

CS25C02 – COMPUTER PROGRAMMING PYTHON

AS PER THE LATEST ANNA UNIVERSITY SYLLABUS

COMMON TO I SEMESTER B.E & B.TECH

(Civil Engineering ,Mechanical Engineering ,Petroleum Engineering)

Regulation 2025

EDITORS & AUTHORS

K.DANIEL RAJ

K.BENITLIN SUBHA

Primary Authors

Dr.J.Japhynth

Dr.T.Jasperline

K.Benitlin Subha

K.Daniel Raj

K.Diala

P.Daniel Raj

J.Vinisha





ISBN: 978-93-48615-60-2

Publisher: Ryan Publishers
15, Sivan Koil 1st Street,
Banker's colony Extension,
Kumaran Nagar,
Trichy - 620017, TamilNadu, India.
Website: www.ryanpublishers.co.in
Email: editor@ryanpublishers.co.in
Ph: +91 6374561101

Copyright © 2025 by Ryan Publishers

No part of this book shall be reproduced or transmitted in any form or by any means (electronic or mechanical including photocopy, recording, or any information storage or retrieval system) without the permission in writing of the publisher.

EDITORS & AUTHORS

EDITOR PROFILE	EDITOR BIOGRAPHY
 <p>Mrs.K.Benitlin Subha Assistant Professor KINGS ENGINEERING COLLEGE</p>	<p>Mrs.K.Benitlin Subha is working as Assistant Professor in Department of Information Technology at Kings Engineering College, has about 10+ years of teaching experience. She received her B.Tech. degree in Information Technology from Dr.Sivanthi Aditanar College of Engineering, Tiruchendur, M.E. degree in Computer Science and Engineering from Anna University Tiruchirapalli. She is pursuing her Ph.D Degree in Anna University, Chennai. Her area of Research includes Machine Learning, IoT, Cloud Computing, etc..</p>
 <p>Mr.K.Daniel Raj passionate educator and researcher</p>	<p>Mr.K.Daniel Raj is a passionate educator and researcher with a background in Information Technology and Computer Science.He has published research articles in reputed journals like IEEE, IGI Global, and SCOPUS, and contributed as both an author and reviewer. He is a reviewer and author of several international and national journals, with interests in Python programming, AI and ML. He is also a member of ISTE and actively contributes to academic and research communities.</p>

PRIMARY AUTHORS

AUTHOR PROFILE	AUTHOR BIOGRAPHY
 <p>Dr.J.Japhynth Principal & Professor Dr.GUPCE</p>	<p>Dr.J.Japhynth Principal of Dr. G.U. Pope College of Engineering, holds an M.E. from Anna University and a Ph.D. in Grid Computing from Karunya University. With over 22 + years of experience in teaching and administration, she has published 20+ research papers and serves as a reviewer for reputed journals. A gold medalist and recipient of the Dale Carnegie Certificate, she is also a frequent guest speaker at academic institutions.</p>
 <p>Dr.T.Jasperline Professor & Head of CSE Dr.GUPCE</p>	<p>Dr. T. Jasperline is a distinguished academician with over 21 years of teaching and research experience. She earned her Doctoral degree in 2018 and currently serves as Professor and Head of the Computer Science and Engineering Department. She has published numerous research articles and is a lifetime member of the (ISTE) and (IE). Her leadership and dedication to academic excellence have significantly contributed to curriculum development, student mentorship, and the advancement of computer science education.</p>



Mrs.K.Diala
Assistant Professor
Dr.GUPCE

Mrs.K.Diala is an Assistant Professor in the Department of Computer Science and Engineering at Dr. G.U. Pope College of Engineering, bringing over 21 years of rich teaching experience. She has published several research papers in reputed journals and holds patents in her field. She is a lifetime member of both (ISTE) and (IE), actively contributing to the academic and research community.



Mr.P.Daniel raj
Assistant Professor
Dr.GUPCE

Mr.P.Daniel raj is an Assistant Professor in the Department of Computer Science and Engineering at Dr. G.U. Pope College of Engineering, bringing over 12+ years of teaching experience. He is committed to fostering academic excellence and contributing to the field of computer science through research and innovation.



Ms.Vinisha J
Assistant Professor
St.Joseph's Institute of
Technology

Ms. Vinisha J is an Assistant Professor in the Department of Information Technology at St. Joseph's Institute of Technology, Chennai, with over five years of teaching experience. Currently pursuing her Ph.D., her expertise includes problem-solving methodologies and intelligent computing systems. She has guided many student projects and actively contributed to workshops on modern programming practices.

Message from the Editorial Board

We are pleased to present this comprehensive guide on **CS25C02 – Computer Programming: Python**, meticulously crafted to align with the Anna University Regulation 2025 syllabus. This book aims to provide students with a solid foundation in Python programming, encompassing both theoretical concepts and practical applications. Through clear explanations, illustrative examples, and hands-on exercises, we endeavor to make programming accessible and engaging for all learners. It is our hope that this resource will inspire students to embrace the world of programming with confidence and curiosity.

Editors & Authors

- K. Daniel Raj
- K. Benitlin Subha

Message from the Author Board

As contributors to this book, our collective goal is to demystify the complexities of Python programming and present it in a manner that is both understandable and enjoyable. Drawing from our diverse academic and professional experiences, we have collaborated to develop content that not only adheres to the prescribed syllabus but also fosters a deeper appreciation for the art of programming. We trust that this book will serve as a valuable companion in your educational journey, equipping you with the skills necessary to excel in the field of computer science and engineering.

Primary Authors:

- Dr. J. Japhynth (Reviewer and Author)
- Dr. T. Jasperline (Reviewer and Author)
- K. Benitlin Subha
- K. Daniel Raj
- K. Diala
- P. Daniel Raj
- J. Vinisha



CS25C02 – COMPUTER PROGRAMMING: PYTHON

SYLLABUS

UNIT I : INTRODUCTION TO PYTHON

Problem solving, Problem Analysis Chart, Developing an Algorithm, Flowchart and Pseudocode, Interactive and Script Mode, Indentation, Comments, Error messages, Variables, Reserved Words, Data Types, Arithmetic operators and expressions, Built-in-Functions, Importing from Packages.

Practical: Problem Analysis Chart, Flowchart and Pseudocode Practices. (Minimum three)

UNIT II : CONTROL STRUCTURES

If, if-else, nested if, multi-way if-elif statements, while loop, for loop, nested loops, pass statements.

Practical: Usage of conditional logics in programs. (Minimum three)

UNIT III : FUNCTIONS

Hiding redundancy, complexity; Parameters, arguments and return values; formal vs actual arguments, named arguments, Recursive & Lambda Functions.

Practical: Usage of functions in programs. (Minimum three)

UNIT IV : STRINGS & COLLECTIONS

String Comparison, Formatting, Slicing, Splitting, Stripping, List, tuples, and dictionaries, basic list operators, searching and sorting lists; dictionary literals, adding and removing keys, accessing and replacing values.

Practical: String manipulations and operations on lists, tuples, sets, and dictionaries. (Minimum three)

UNIT V : FILES OPERATIONS, PACKAGES

Create, Open, Read, Write, Append and Close files. Manipulating directories, OS and Sys modules, reading/writing text and numbers, from/to a file; creating and reading a formatted file (csv, tab-separated, etc.)

Practical: Opening, closing, reading and writing in formatted file format and sort data. (Minimum three),

UNIT VI : PACKAGES

Built-in modules, User-Defined modules, Numpy, SciPy, Pandas, Scikit-learn

Practical: Usage of modules and packages to solve problems. (Minimum three), Project (Minimum Two)

Content

S.No	Title	Pg.No
1	Unit – I - Introduction to Python	
	Fundamentals of computing	01
	Identifications of computational problem	06
	1.1 Problem solving	09
	1.2 Problem Analysis Chart	13
	1.3 Developing an Algorithm	16
	1.4 Flowchart and Pseudocode	19
	1.4.1 Flowchart	19
	1.4.1.1 Flow Chart Symbols	19
	1.4.1.2 Rules for Drawing Flow Chart	20
	1.4.2 Pseudocode	23
	1.5 Python Programming Environment	26
	1.5.1 Interactive Mode	26
	1.5.2 Script Mode	27
	1.6 Python Syntax and Indentation	29
	1.7 Comments and Documentation	31
	1.8 Error Messages and Debugging Basics	34
	1.9 Variables and Reserved Words	38
	1.9.3 Difference between Variables and	40

	Reserved Words	
	1.10 Data Types	42
	1.10.1 Data Types in Python	42
	1.10.2 type Conversion	44
	1.11 Arithmetic Operators and Expressions	45
	1.12 Built-in Functions	49
	1.13 Importing from Packages	51
	1.14 Practical	54
	-Python History Overview-	58
	1.15 Important Two Mark Questions	61
2	Unit – II - Control Structures	
	2.1 Introduction to Control Flow	65
	2.1.1 Functions	66
	2.2 Decision-Making Statements	70
	2.2.1 Conditional If..	70
	2.2.2 If..Else..Condition	72
	2.2.3 Chained Condition	74
	2.3 Iterative Constructs	76
	2.3.1 for loop	76
	2.3.1 While loop	77
	2.3.1 nested loops	78

	2.4 Jump Statements	79
	2.4.1 Break	79
	2.4.2 Continue	80
	2.4.3 pass	81
	2.5 Fruitful Functions -	82
	2.6 Practical	85
	2.6 Important Two Mark Questions	93
3	UNIT- III - Functions	
	3. Introduction to Functions	96
	3.1 Need for Functions: Hiding Redundancy & Complexity	98
	3.2 Function Definition and Calling	101
	3.3 Parameters and Arguments	102
	3.4 Return Values	103
	3.5 Formal vs Actual Arguments	104
	3.6 Named Arguments	106
	3.7 Recursive Functions	107
	3.8 Lambda (Anonymous) Functions	108
	3.9 Practical	107
	3.9.1 Usage of functions in programs	109
	(Minimum three)	

	3.10 Summary Table	110
	3.11 Important Two Mark Questions	112
4	UNIT – IV - Strings and Collections	
	4.1 Strings in Python	116
	4.1.1 String Slicing	117
	4.1.2 Immutability	118
	4.1.3 String Function and Methods	119
	4.2 Python Collections Overview	124
	4.2.1 Lists	124
	4.2.2 Tuples	126
	4.2.3 Sets	130
	4.2.4 Dictionaries	130
	4.3 Basic List Operators	134
	4.3.1 Basic List Operators In Python	134
	4.3.2 List Methods in Python	135
	4.4 Searching and Sorting Lists	137
	4.4.1 Searching in List	138
	4.4.2 Sorting in List	141
	4.5 Dictionary Literals, Keys, and Values	145
	4.5.1 Dictionary Literals	146
	4.5.2 Dictionary Keys	147

	4.5.3 Dictionary Values	148
	4.5.4 Dictionary Items (Keys + Values)	148
	4.6 Accessing Values	150
	4.6.1 Adding Keys	150
	4.7 Replacing Values	151
	4.7.1 Removing Keys	152
	4.8 Dictionary Operations in Python	154
	4.9 Adding Removing ,Replacing Elements	155
	4.10 Practical Exercises	156
	4.10.1 String manipulations	156
	4.10.2 operations on lists	157
	4.10.3 operations on tuples	158
	4.10.4 operations on sets	159
	4.10.5 operations on dictionaries4.8 4.11	159
	Important Two Mark Questions	160
5	UNIT- V -File Operations	
	Introduction	164
	5.1 File Handling Concepts	165
	5.2 Creating and Opening Files	166
	5.3 Reading and Writing Files	168
	5.4 Appending and Closing Files	170

5.5 Directory Manipulation (OS and Sys modules)	171
5.5.1 Accessing Details	172
5.5.2 Returning Path	173
5.5.3 Env & Comm Line Argument	174
5.5.4 Reading/Writing Text and Numbers , From /To a File	174
5.6 Difference Between Os & Sys Module in Python	175
5.7 Reading/Writing Formatted Files (CSV, tab-separated, etc.)	176
5.7.1 Working with CSV Files in Python	176
5.7.1.1 Reading a CSV file	176
5.7.1.2 Reading CSV Files Into a Dictionary With csv	178
5.7.1.3 Writing to a CSV file	178
5.7.1.4 Writing a dictionary to a CSV file	178
5.7.2 Simple Way to Read CSV Files in Python	180
5.8 Practical Exercises	183
5.9 Important Two Marks	186

6	UNIT- VI- Packages	
	6. Introduction to Modules and Packages	189
	6.1 Built-in Modules (math, random, sys, os, etc.)	191
	6.2 Packages	197
	6.3 Build In Modules	200
	6.4 Python Module Summary Table	213
	6.5 User-Defined Modules	214
	6.6 Popular Python Packages for Engineers	218
	6.6.1 Numpy	218
	6.6.2 SciPy	210
	6.6.3 Pandas	220
	6.6.4 Scikit-learn	220
	6.7 Python Library Summary Table	221
	6.8 Case Studies	223
	6.9 Practicals	226
	6.9.1 usage of modules and packages (Minimum three each)	226
	6.9.2 Mini Projects (Minimum Two)	230
	6.10 Important Two Mark Questions	233

UNIT – I

INTRODUCTION TO PYTHON

Problem solving, Problem Analysis Chart, Developing an Algorithm, Flowchart and Pseudocode, Interactive and Script Mode, Indentation, Comments, Error messages, Variables, Reserved Words, Data Types, Arithmetic operators and expressions, Built-in-Functions, Importing from Packages.

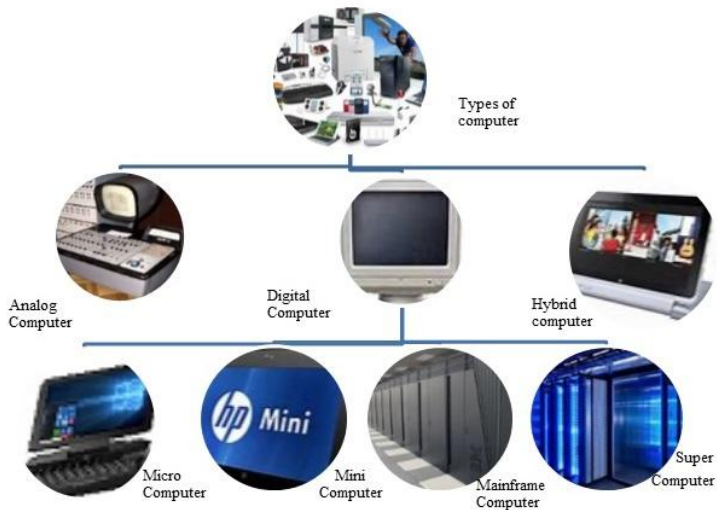
Practical: Problem Analysis Chart, Flowchart and Pseudocode Practices. (Minimum three)

FUNDAMENTALS OF COMPUTING

What is Computer?

- The computer is a super-intelligent electronic device that can perform tasks, process information, and store data.
- It takes the data as an input and processes that data to perform tasks under the control of a program and produces the output.
- A computer is like a personal assistant that follows instructions to get things done quickly and accurately.
- It has memory to store information temporarily so that the computer can quickly access it when needed.

Types of Computer



Functionalities of Computer

- **Input** – Feeding data into the system.
- **Processing** – Working on the data using CPU.
- **Storage** – Saving data temporarily or permanently.
- **Output** – Displaying the result of processing.
- **Control** – Coordinating all operations within the computer.

Components Of Computer

Computer = Hardware (physical parts) + Software (programs to control the hardware)



Hardware Components

These are the physical parts of a computer system.

a) Input Devices

Used to enter data into the computer.

Examples:

- Keyboard
- Mouse
- Scanner

b) Output Devices

Used to display or produce the result of processed data.

Examples:

- Monitor
- Printer
- Speakers

c) Processing Unit

The brain of the computer, also known as the CPU.

It includes:

- **ALU (Arithmetic Logic Unit)** – Performs all arithmetic and logical operations.
- **CU (Control Unit)** – Directs the operation of the processor.
- **Registers** – Small memory units for quick data access.

d) Memory/Storage Devices

Used to store data permanently or temporarily

Primary Memory (Volatile)

- RAM (Random Access Memory)
- ROM (Read Only Memory)

Secondary Storage (Non-Volatile):

- HDD (Hard Disk Drive)
- SSD (Solid State Drive)
- USB Flash Drives
- CDs/DVDs

Software Components

Programs and operating systems that control the hardware.

a) System Software

Manages hardware and basic system operations.

Examples:

- Operating Systems (Windows, Linux, macOS)
- Device Drivers

b) Application Software

Programs that perform specific tasks for users.

Examples:

- MS Word, Excel
- Web Browsers
- Games

c) Utility Software

Helps maintain and optimize the system.

Examples:

- Antivirus
- Disk Cleanup Tools

IDENTIFICATION OF COMPUTATIONAL PROBLEM

Definition:

Identifying a computational problem means recognizing a real-world issue that can be solved using a computer algorithm or program. It involves translating the problem into a format that can be handled using logical steps and computational methods.

Computational thinking can be split into four parts,

1. **Decomposition**
2. **Pattern recognition**
3. **Abstraction**
4. **Algorithmic design**

In order to solve a problem computationally, two things are needed,

- A **representation** that captures all the relevant aspects of the problem
- An **algorithm** that solves the problem by use of the representation.

Problem: Create an app that calculates the average grade for each student.

1. Decomposition:

We need to break the problem down into smaller tasks.

- **Input collection:** Gather the grades for each assignment for every student.
- **Grade calculation:** Calculate the average grade for each student.
- **Display result:** Output the average grade for each student.

Now the problem is divided into manageable chunks.

2. Pattern Recognition:

Identify patterns and repetitive elements in the problem.

- The steps to compute the average for one student can be **repeated** for all students.
- We can recognize that the **calculation of an average** involves a common formula (sum of grades divided by the number of grades).

3. Abstraction:

Generalize the problem to focus only on the important details.

Ignore unnecessary details like the subject of assignments or individual student names.

Abstract the process:

We just need to know the grades for each student and apply the same calculation for all students. Focus on the **structure**: We

will have a list of numbers (grades) for each student, and the task is to calculate the average for that list.

4. Algorithm Design:

Create a step-by-step plan (algorithm) to solve the problem.

Step-by-step algorithm:

Input: Collect the grades for each student (e.g., in a list or table).

Process: For each student:

- Sum all the grades.
- Divide the sum by the number of assignments to find the average.

Output: Display the average grade for each student.

Computational thinking allows us,

1. To take a complex problem
2. Understand what the problem is
3. Develop possible solutions
4. Present these solutions in a way that a computer, a human can understand.

1.1 PROBLEM SOLVING

1.1.1 Introduction

Problem solving is the process of identifying a computational or engineering problem, analyzing its requirements, and developing an efficient solution using systematic methods. In the context of programming, problem solving is the foundation for writing correct, efficient, and reusable code.

1.1.2 Definition

Problem solving in computer science can be defined as:

“A step-by-step process of defining a problem, designing a procedure (algorithm), and implementing it using a programming language to obtain the desired output.”

1.1.3 Characteristics of a Good Problem-Solving Process

1. **Clarity** – The problem statement must be clearly understood.
2. **Systematic Approach** – Steps should follow a logical sequence.
3. **Efficiency** – The solution must minimize time and space complexity.
4. **Correctness** – The solution should produce accurate results for all valid inputs.
5. **Scalability** – The approach should adapt to larger or complex inputs.

1.1.4 General Steps in Problem Solving

1. **Problem Identification** – Clearly state the problem.
2. **Problem Analysis** – Understand input, process, and output requirements.
3. **Solution Design** – Develop an algorithm, flowchart, or pseudocode.
4. **Implementation** – Translate the design into a program (Python).
5. **Testing and Verification** – Run test cases to check correctness.
6. **Maintenance** – Modify or optimize the solution as requirements evolve.

1.1.5 Computational Problem Solving in Engineering

In engineering, problem solving often involves:

- **Mathematical computations** (e.g., stress analysis, area, volume calculations).
- **Data processing** (e.g., analyzing sensor data, simulation outputs).
- **Automation** (e.g., controlling devices, file handling, batch processing).
- **Modeling and Simulation** (e.g., using Python libraries for system modeling).

Example: Calculating Area of a Rectangle

Problem Statement: Write a program to compute the area of a rectangle given its length and breadth.

Step 1 – Problem Identification:

- Input → Length and Breadth
- Process → Multiply Length \times Breadth
- Output → Area of the rectangle

Step 2 – Algorithm:

1. Start
2. Read length and breadth
3. Compute area = length \times breadth
4. Display area
5. Stop

Step 3 – Pseudocode:

BEGIN

 READ length, breadth

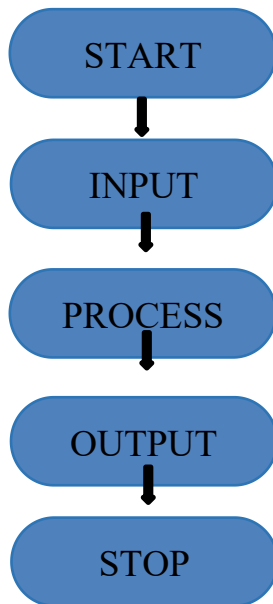
 area \leftarrow length * breadth

 PRINT area

END

Step 5 – Python Implementation:

```
# Program to calculate area of a rectangle
length = float(input("Enter length: "))
breadth = float(input("Enter breadth: "))
area = length * breadth
print("Area of rectangle:", area)
```

Step 4 – Flowchart:**1.1.6 Summary**

- Problem solving is a systematic process that transforms a problem statement into a working solution.
- It involves understanding inputs/outputs, designing algorithms, and implementing solutions in Python.
- Engineering applications frequently rely on computational problem solving to automate and optimize tasks.

1.2 PROBLEM ANALYSIS CHART

1.2.1 Definition

A **Problem Analysis Chart (PAC)** is a tabular representation that breaks down a problem into its fundamental components:

- **Input** – The data required for the problem.
- **Process** – The computational or logical steps needed to transform input into output.
- **Output** – The expected result of the problem.

1.2.2 Components of Problem Analysis Chart (PAC)

Component	Description	Example
Problem Statement	Clearly defines what needs to be solved	Find the area of a rectangle
Input	Data values required from the user	Length, Breadth
Process	Operations or calculations performed on input	$\text{Area} = \text{Length} \times \text{Breadth}$
Output	Final result to be displayed	Area of the rectangle

1.2.3 Steps to Construct a PAC

1. Identify the **problem statement**.
2. Determine all necessary **inputs**.
3. List the **process steps** needed to solve the problem.

4. Define the expected **output**.
5. Tabulate the above information into a PAC.

1.2.4 Example: PAC for Area of Rectangle

Problem	Find the Area of a Rectangle
Input	Length, Breadth
Process	$\text{Area} = \text{Length} \times \text{Breadth}$
Output	Area of the rectangle

1.2.5 Example: PAC for Simple Interest

Problem	Calculate Simple Interest
Input	Principal (P), Rate of Interest (R), Time (T)
Process	$\text{SI} = (\text{P} \times \text{R} \times \text{T}) / 100$
Output	Simple Interest amount

1.2.6 Benefits of PAC

- Provides **clarity** before coding.
- Reduces logical errors by defining all inputs and outputs.
- Acts as a **bridge** between problem statement and algorithm.
- Helps beginners systematically approach programming problems.

1.2.7 Python Example Based on PAC

```
# Program to calculate area of rectangle using PAC
length = float(input("Enter length: "))
breadth = float(input("Enter breadth: "))
area = length * breadth
print("Area of rectangle:", area)
```

1.2.8 Summary

- The Problem Analysis Chart (PAC) is a tool to break a problem into **input, process, and output**.
- It simplifies problem-solving by organizing requirements before coding.
- PAC ensures that the transition from **problem statement** → **algorithm** → **code** is systematic.

1.3 DEVELOPING AN ALGORITHMS

Definition of Algorithm:

An algorithm is a finite set of instructions for performing a particular task. The instructions are nothing but the statements in simple English language.

In computing, we focus on the type of problems categorically known as algorithmic problems, where their solutions are expressible in the form of algorithms. The term “algorithm” was derived from the name of Mohammed al-Khowarizmi, a Persian mathematician in the ninth century.

Al-Khowarizmi → Algorithmus (in Latin) → Algorithm.

An algorithm is a well-defined computational procedure consisting of a set of instructions that takes some value or set of values, as input, and produces some value or set of values, as output.

In other word, an algorithm is a procedure that accepts data; manipulate them following the prescribed steps, so as to eventually fill the required unknown with the desired value(s).



People in various professions follow specific procedures in their work, each referred to by a different name. For example, a cook uses a method known as a recipe, which transforms ingredients (input) into a finished dish (output) through a series of steps.

1.3.1 Characteristics of a Good Algorithm

1. **Precision** – Every step in the algorithm must be clearly and accurately defined.
2. **Uniqueness** – Each step should yield a unique result, depending solely on the given input and the result of previous steps.
3. **Finiteness** – The algorithm must terminate after a limited number of steps.
4. **Effectiveness** – Among all possible approaches, the algorithm should provide the most efficient solution to the problem.
5. **Input** – It should accept one or more inputs to begin the process.
6. **Output** – It must generate at least one output as the result.
7. **Generality** – The algorithm should be applicable to a wide range of input values, not just specific cases.

Example 1.3.1: Algorithm to Check if a Number is Even or Odd**Algorithm Steps:**

Step 1 :	Start
Step 2 :	Take a number as input
Step 3 :	If the number is divisible by 2, it is even
Step 4 :	Else, it is odd
Step 5 :	Display the result
Step 6 :	Stop

Example 1.3.2: Algorithm to Find the Largest of Two Numbers**Algorithm Steps:**



Step 1 :	Start
Step 2 :	Get two numbers as input
Step 3 :	Compare the two numbers
Step 4 :	If the first number is greater, it is the largest
Step 5 :	Else, the second number is the largest
Step 6 :	Display the result
Step 7 :	Stop



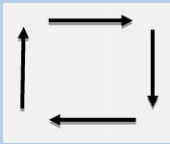
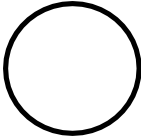
1.4 FLOW CHART AND PSEUDOCODE

1.4.1 Flow Charts

- Flowcharts are the graphical representation of the algorithms.
- The algorithms and flowcharts are the final steps in organizing the solutions.
- Using the algorithms and flowcharts the programmers can find out the bugs in the programming logic and then can go for coding.
- Flowcharts can show errors in the logic and set of data can be easily tested using flowcharts

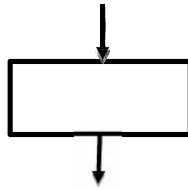
1.4.1.1 Flow chart Symbols

S. No	Name of symbol	Symbol	Type	Description
1.	Terminal Symbol		Oval	Represent the start and stop of the program.
2.	Input/ Output symbol		Parallelogram	Denotes either input or output operation.

3.	Process symbol		Rectangle	Denotes the process to be carried
4.	Decision symbol		Diamond	Represents decision Making and branching
5.	Flow lines		Arrow lines	Represents the sequence of steps and direction of flow. Used to connect symbols.
6.	Connector		Circle	A connector symbol is represented by a circle and a letter or digit is placed in the circle to specify the link. This symbol is used to Connect flow charts.

1.4.1.2 Rules for drawing flow chart

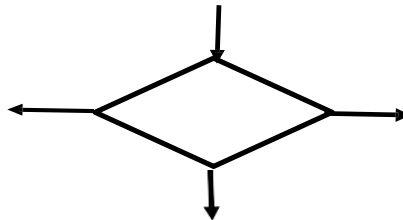
1. In drawing a proper flowchart, all necessary requirements should be listed out in logical order.
2. The flow chart should be clear, neat and easy to follow. There should not be any room for ambiguity in understanding the flowchart.



3. The usual directions of the flow of a procedure or system are from left to right or top to bottom. Only one flow line should come out from a process symbol.



4. Only one flow line should enter a decision symbol, but two or three flow lines, one for each possible answer, can leave the decision symbol.



5. Only one flow line is used in conjunction with terminal symbol.



6. If flow chart becomes complex, it is better to use connect or symbols to reduce the number of flow lines.
7. Ensure that flow chart has logical start and stop.

1.4.1.3 Advantages of Flow chart

1. **Communication:** Flow charts are better way of communicating the logic of the system.
2. **Effective Analysis:** With the help of flow chart, a problem can be analyzed in more effective way.
3. **Proper Documentation:** Flow charts are used for good program documentation, which is needed for various purposes.
4. **Efficient Coding:** The flowcharts act as a guide or blue print during the system analysis and program development phase.
5. **Systematic Testing and Debugging:** The flow chart helps in testing and debugging the program
6. **Efficient Program Maintenance:** The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part.

1.4.1.4 Disadvantages of Flowchart

1. **Complex Logic:** Sometimes, the program logic is quite complicated. In that case flow chart becomes complex and difficult to use.
2. **Alteration and Modification:** If alterations are required the flowchart may require re- drawing completely.
3. **Reproduction:** As the flowchart symbols cannot be typed, reproduction becomes problematic.

1.4.2 Pseudo Code

Pseudocode is an informal high-level description of the operating principle of a computer program or algorithm. It uses the basic structure of a normal programming language, but is intended for human reading rather than machine reading. It is text based detail design tool. Pseudo means false and code refers to instructions written in programming language.

Pseudocode cannot be compiled nor executed, and there are no real formatting or syntax rules. The pseudocode is written in normal English language which cannot be understood by the computer.

Example 1.4.1 : Pseudocode: To find sum of two numbers

```
READ num1, num2
```

```
Sum = num1+num2
```

```
PRINT sum
```

Basic rules to write pseudocode:

1. Only one statement per line. Statements represents single action is written on same line. For example to read the input, all the inputs must be read using single statement.
2. Capitalized initial keywords The keywords should be written in capital letters. Eg: **READ, WRITE, IF, ELSE, ENDIF, WHILE, REPEAT, UNTIL**

Example 1.4.2 : Pseudocode: Find the total and average of three subjects

```
READ name, department, mark1, mark2, mark3
```

```
Total=mark1+mark2+mark3
```

```
Average=Total/3
```

```
WRITE name, department, mark1, mark2, mark3
```

1. Indent to show hierarchy Indentation is a process of showing the boundaries of the structure.
2. End multi-line structures each structure must be ended properly, which provides more clarity.

Example 1.4.3 : Pseudocode: Find greatest of two numbers

```
READ a, b
```

```
IF a>b then
```

```
PRINT a is greater
```

```
ELSE
```

```
PRINT b is greater
```

```
END IF
```

1. Keep statements language independent.

Pseudocode must never written or use any syntax of any programming language.

1.4.2.1 Advantages of Pseudocode

- Can be done easily on a word processor
- Easily modified
- Implements structured concepts well
- It can be written easily
- It can be read and understood easily
- Converting pseudocode to programming language is easy as compared with flowchart

1.4.2.2 Disadvantages of Pseudocode

- It is not visual
- There is no standardized style or format

1.5. PYTHON PROGRAMMING ENVIRONMENT

1. Interactive mode
2. Script mode

1.5.1 INTERACTIVE MODE

- Python has two modes: **normal mode** and **interactive mode**.

Interactive Mode:

Also called **Script Mode**. In this mode, the user can type Python commands directly in the interpreter. The interpreter immediately executes the command and displays the result. Each statement is executed as soon as it is typed. This is particularly useful for testing small code snippets.

- The prompt `>>>` indicates that the interpreter is ready for user input.
- When an expression is typed after the prompt, the interpreter displays the result immediately.
- Proper indentation is required while writing the code on the interpreter shell.

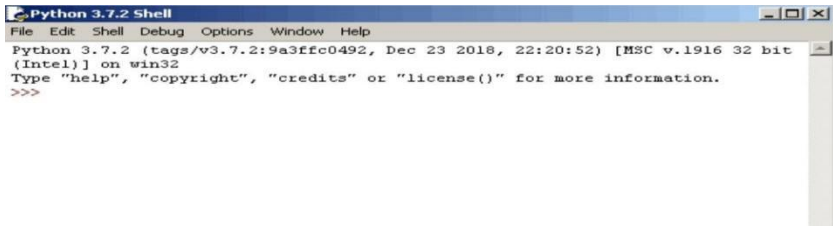


fig 1.5.1 Interactive mode

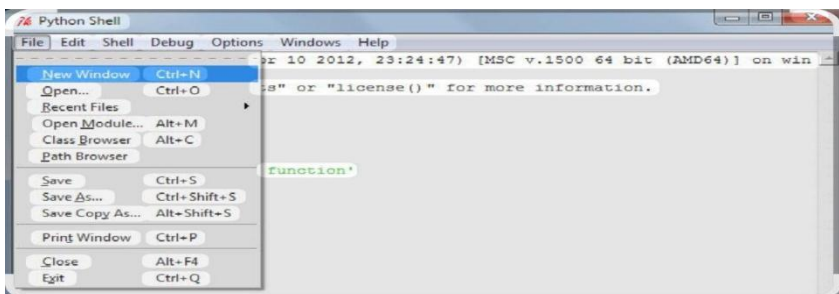
1.5.2. SCRIPT MODE

- Script mode is the **standard mode**.
- In this mode, Python commands are saved in a file with the extension .py.
- The saved file can then be executed repeatedly.

Steps to run Python in Script Mode:

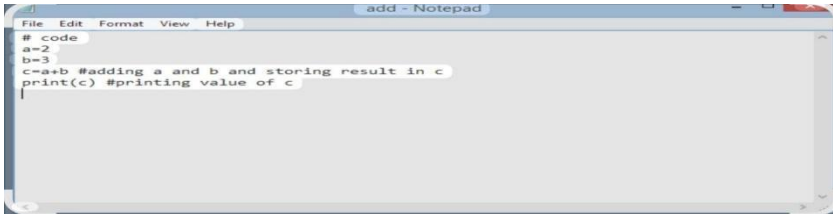
Step 1: Open Python Shell by double-clicking the Python IDE.

Step 2: On the File Menu, click on **New File** option.

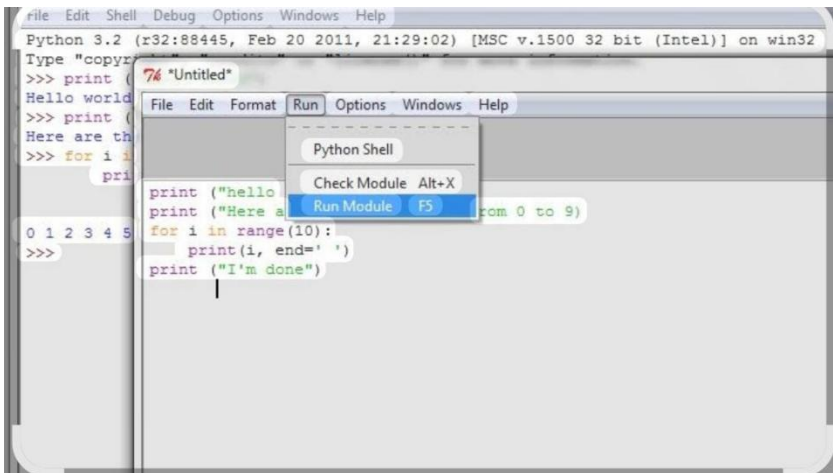


Step 3: Give some suitable file name with extension .py (e.g., example.py).

Step 4: A file will get opened and the type some programming code.



Step 5: Now run your code by clicking on **Run Menu** → **Run Module (F5)**.



The output will be displayed on the python shell.

```
→ ~ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more
>>> raw_input()
I am learning
```

1.6 PYTHON SYNTAX AND INDENTATION

1.6.1 PYTHON SYNTAX

- Python syntax refers to the set of rules that defines how Python programs are written and interpreted.
- Unlike many programming languages (such as C, C++ or Java), Python syntax is simple and emphasizes readability.
- Statements in Python are written line by line and executed sequentially.
- Each statement usually ends with a **newline**, not with a semicolon (;).
- Indentation, rather than braces {} or keywords, is used to define code blocks.

Example: A simple Python statement

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

Example: Multiple statements on one line

```
x = 5; y = 10; print(x + y)
```

1.6.2 INDENTATION IN PYTHON

- **Indentation** means the space at the beginning of a line of code.
- In Python, **indentation is mandatory** and indicates a block of code.
- Other programming languages like C/Java use curly braces { } to define blocks, but Python relies on indentation only.
- The standard indentation is **4 spaces** (tabs may also be used, but mixing tabs and spaces should be avoided).

Example: Using indentation in if statement

```
x = 10
```

```
if x > 5:
```

```
    print("x is greater than 5")
```

```
    print("This line is also inside the block")
```

```
print("This line is outside the block")
```

Output:

x is greater than 5

This line is also inside the block

This line is outside the block

1.6.2.1 INDENTATION ERRORS

- If indentation is not used correctly, Python raises an **IndentationError**.
- This helps prevent ambiguity in program structure.

Example: Error due to incorrect indentation

```
x = 10
```

```
if x > 5:
```

```
    print("x is greater than 5") # ✗ IndentationError
```

Error Message:

IndentationError: expected an indented block

1.7 COMMENTS AND DOCUMENTATION

Comments and documentation are essential for writing **readable, maintainable, and error-free code**.

They help programmers explain the **logic, purpose, and functionality** of code segments without affecting execution.

Python ignores comments during program execution.

1.7.1 Comments in Python

Python provides two types of comments:

a) Single-Line Comments

- Begin with the `#` symbol.
- Everything after `#` on the same line is ignored by the interpreter.

Example:

```
# This is a single-line comment  
  
x = 10 # Assigning value 10 to variable x  
  
print(x)
```

b) Multi-Line Comments

- Python does not have explicit multi-line comment syntax (like `/* ... */` in C).
- Instead, multi-line comments are created using **triple quotes** (`''' ... '''` or `""" ... """`).
- They can span across multiple lines.

Example:

```
"""
This is a multi-line comment
explaining the program logic
Author: Student
Date: 2025-08-30
"""
print("Hello, Python")
```

1.7.2 Documentation Strings (Docstrings)

- **Docstrings** are special string literals used for documenting **modules, classes, functions, or methods**.
- Declared using triple quotes (""" ... """ or ''' ... ''').
- Can be accessed at runtime using the `__doc__` attribute.

Example: Function with Docstring

```
def add(a, b):

    """This function takes two numbers
    and returns their sum."""

    return a + b

print(add.__doc__) # Accessing the docstring
```

Output:

This function takes two numbers
and returns their sum.

1.7.3 Importance of Comments and Documentation

- Increases **readability** of programs.
- Helps in **debugging and maintenance**.
- Acts as **reference material** for future developers.
- Essential for **collaborative coding projects** and **engineering software systems**.

1.8 ERROR MESSAGES AND DEBUGGING BASICS

Errors are inevitable in programming and occur when the Python interpreter fails to execute code correctly. Understanding error messages and learning debugging techniques are essential skills for every programmer. Python provides clear and descriptive error messages that help identify the type and location of the problem. Debugging is the systematic process of detecting, analyzing, and fixing errors to ensure reliable software execution.

1.8.1 Types of Errors in Python

(a) Syntax Errors

- Occur when Python code violates grammatical rules.
- Detected during the parsing stage, before execution.

- Example: Missing colon, wrong indentation.

Syntax Error Example

```
if True
```

```
    print("Hello") # Missing colon after if
```

Error Message:

SyntaxError: expected ':'

(b) Runtime Errors (Exceptions)

- Occur while the program is running.
- Examples: division by zero, accessing invalid index, wrong data type.

Runtime Error Example

```
x = 10 / 0
```

Error Message:

ZeroDivisionError: division by zero

(c) Logical Errors

- The program runs without crashing but produces incorrect results.
- Hardest to detect since Python does not raise an error.

Logical Error Example

Program to calculate average (wrong formula used)

```
marks = [80, 90, 70]
```

```
average = sum(marks) * len(marks) # Incorrect
```

```
print("Average =", average)
```

1.8.2 Debugging Basics

Step 1: Reading Error Messages

- Identify the **type of error** (SyntaxError, ValueError, etc.).
- Locate the **line number** provided in the traceback.

Step 2: Using Print Statements

- Insert print() statements at different points to check variable values.
- Helps trace program flow and detect logic errors.

Step 3: Using Python Debugger (pdb)

- Python has a built-in debugger module: **pdb**.
- Allows step-by-step execution, variable inspection, and breakpoints

Example:

```
Import pdb
```

```
def divide(a, b):
```

```
    pdb.set_trace() # Debugger starts here
```

```
    return a / b
```

```
print(divide(10, 2))
```

Step 4: Exception Handling

- Use try-except blocks to gracefully handle errors.

```
try:
```

```
    num = int("abc")
```

```
except ValueError as e:
```

```
    print("Error occurred:", e)
```

1.8.3 Importance of Debugging in Engineering

- Ensures **reliable and efficient** software.
- Reduces **development time and cost** by preventing failures.
- Improves **system performance and accuracy**.
- Essential for **safety-critical engineering applications** (e.g., healthcare, automotive, aerospace).

1.8.4 Best Practices for Debugging

- Read and interpret error messages carefully.
- Start debugging from the **topmost error line** in the traceback.
- Use **modular programming** to isolate and test components.
- Avoid excessive use of `print()`; rely on debugging tools.
- Write **unit tests** to catch errors early.

1.9 VARIABLES AND RESERVED WORDS

In Python programming, **variables** are symbolic names used to store data values, while **reserved words (keywords)** are predefined identifiers with special meaning. Understanding both concepts is fundamental to writing valid and efficient programs.

1.9.1 VARIABLES IN PYTHON

Definition

A **variable** is a name given to a memory location that holds a value. Python allows dynamic typing, meaning variables do not need explicit declaration of type; their type is determined at runtime.

Rules for Variable Naming

1. Must begin with a letter or underscore (_).
2. Cannot start with a digit.
3. Can contain letters, digits, and underscores.
4. Case-sensitive (Age and age are different).

5. Cannot use reserved keywords as variable names.

Valid Examples:

```
student_name = "John"
```

```
marks = 95
```

```
_age = 20
```

Invalid Examples:

```
1stvalue = 100 # Starts with digit
```

```
for = 25      # 'for' is a reserved word
```

Variable Assignment

- Python uses the = operator for assignment.
- Multiple assignments are allowed in one line.

Examples:

```
x = 10      # Single assignment
```

```
a, b, c = 1, 2, 3 # Multiple assignment
```

```
y = z = 100 # Same value to multiple variables
```

Dynamic Typing

- The type of variable is determined automatically.
- Variable type can change during execution.

Example:

```
x = 50      # integer
x = "Hello" # now a string
```

1.9.2 RESERVED WORDS IN PYTHON

Definition

Reserved words (or **keywords**) are predefined identifiers in Python with special meaning. They are part of the Python language syntax and cannot be used as variable names.

List of Python Reserved Words (Python 3.10+)

Keywords

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Soft Keywords

match	case	-
-------	------	---

1.9.3 DIFFERENCE BETWEEN VARIABLES AND RESERVED WORDS

Aspect	Variables	Reserved Words
Definition	User-defined names storing data	Predefined identifiers with special meaning
Flexibilit	Chosen by programmer	Fixed and cannot be

Aspect	Variables	Reserved Words
y		changed
Example	student, marks, count	for, while, if, class
Usage	Store values, perform computations	Define control flow, logic, and structure

1.9.5 APPLICATIONS : Applications of Variables

1. **Data Storage and Processing** – Used to store sensor values, numerical results, and intermediate computations in engineering programs.
2. **Simulation Models** – Variables represent real-world parameters such as speed, pressure, temperature, or voltage in engineering simulations.
3. **User Input Handling** – Store and manipulate user-provided data in interactive applications.
4. **Mathematical Computations** – Perform calculations in scientific and engineering problems (e.g., solving equations, statistical analysis).

Applications of Reserved Words

1. **Control Structures** – Keywords like if, while, for are used to define logic, iterations, and decision-making.
2. **Function and Class Definitions** – def, class help organize code for modular and object-oriented programming.
3. **Exception Handling** – try, except, finally provide robust error handling in engineering applications.
4. **Resource Management** – Keywords like with ensure safe handling of files and network connections.

1.10 DATA TYPES

In Python, **data types** define the type of values that variables can store. Since Python is a **dynamically typed language**, the interpreter automatically determines the data type of a variable during program execution.

Understanding data types is fundamental in programming because they determine:

- How data is stored in memory.
- What operations can be performed on the data.
- How the program interprets the stored values.

1.10.1 Data Types in Python

Python provides several built-in data types, broadly categorized as follows:

a) Numeric Types

- **int** – Integer values without decimal.
- **float** – Real numbers with decimal points.
- **complex** – Numbers with real and imaginary parts ($a + bj$).

Example:

```
x = 10      # int
y = 12.5    # float
z = 3 + 4j  # complex
```

b) Sequence Types

1. **str (String)** – Collection of characters enclosed in quotes (" " or ' ').
2. **list** – Ordered, mutable collection of items.
3. **tuple** – Ordered, immutable collection of items.
4. **range** – Represents an immutable sequence of numbers.

Example:

```
name = "Python"      # string
numbers = [1, 2, 3, 4] # list
coordinates = (10, 20) # tuple
r = range(1, 5)      # range
```

c) Set Types

- **set** – Unordered collection of unique elements.
- **frozenset** – Immutable version of a set.

Example:

```
s = {1, 2, 3, 4}      # set
fs = frozenset([1, 2, 3]) # frozenset
```

d) Mapping Type

- **dict (Dictionary)** – Collection of key–value pairs.

Example:

```
student = {"name": "John", "age": 20, "marks": 95}
```

e) Boolean Type

- **bool** – Represents truth values: True or False.

Example:

```
is_valid = True  
result = (10 > 5) # evaluates to True
```

f) None Type

- **None** – Special data type representing the absence of a value or null value.

Example:

```
x = None
```

1.10.2 Type Conversion

Python supports **type conversion** in two forms:

1. **Implicit Conversion (Type Casting by Interpreter)** – Automatically converts smaller data type to larger data type to avoid data loss.
2. `a = 10` # int
3. `b = 5.5` # float
4. `c = a + b` # result is float (15.5)

5. **Explicit Conversion (Type Casting by User)** – Programmer manually converts data type using built-in functions (int(), float(), str(), etc.).
6. `x = "100"`
7. `y = int(x)` # converts string to int

1.10.3 Applications of Data Types

1. **Engineering Computations** – Numeric types are used for calculations in simulations, measurements, and modeling.
2. **Data Handling** – Strings, lists, and dictionaries are used for handling user input, datasets, and structured information.
3. **Database and Networking** – Dictionaries and sets are useful for representing key-value pairs and unique identifiers.
4. **Logical Decision-Making** – Boolean values help in control flow, testing conditions, and validations.
5. **Error Representation** – None is useful for representing missing or undefined values in engineering software.

1.11 ARITHMETIC OPERATORS AND EXPRESSIONS

In Python, **arithmetic operators** are used to perform mathematical operations on numerical data types such as integers, floats, and complex numbers. An **expression** is a combination of variables, constants, and operators that evaluates to a value.

Understanding operators and expressions is essential for

developing algorithms, solving engineering computations, and building logic in programs.

1.11.1 ARITHMETIC OPERATORS IN PYTHON:

Definitions of Arithmetic Operators

1. **Addition (+)**

The addition operator is used to add two operands and produce their sum.

Example: $5 + 3 = 8$

2. **Subtraction (-)**

The subtraction operator calculates the difference between two operands.

Example: $10 - 4 = 6$

3. **Multiplication (*)**

The multiplication operator returns the product of two operands.

Example: $7 * 3 = 21$

4. **Division (/)**

The division operator divides the left operand by the right operand and returns a floating-point result.

Example: $10 / 4 = 2.5$

5. **Floor Division (//)**

The floor division operator divides two operands but discards the decimal part, returning the largest integer less than or equal to the result.

Example: $10 // 4 = 2$

6. **Modulus (%)**

The modulus operator returns the remainder of a division operation.

Example: $10 \% 4 = 2$

7. Exponentiation (**)

The exponentiation operator raises the first operand to the power of the second operand.

Example: $2 ** 3 = 8$

Operator	Description	Example	Result
+	Addition	$10 + 5$	15
-	Subtraction	$10 - 5$	5
*	Multiplication	$10 * 5$	50
/	Division (float result)	$10 / 3$	3.333
//	Floor Division	$10 // 3$	3
%	Modulus (Remainder)	$10 \% 3$	1
**	Exponentiation (Power)	$2 ** 3$	8

Arithmetic Expressions

An **arithmetic expression** is a combination of numbers, variables, and operators that produces a numerical result.

Examples:

$a = 10$

$b = 3$

```
expr1 = a + b      # 13
```

```
expr2 = a - b      # 7
```

```
expr3 = a * b + 2    # 32
expr4 = a / b        # 3.333...
expr5 = a % b        # 1
expr6 = a ** b       # 1000
```

1.11.2 Operator Precedence and Associativity

When multiple operators are used in a single expression, Python follows **operator precedence** rules.

Precedence Order (Highest to Lowest):

1. **Parentheses** `()`
2. **Exponentiation** `**`
3. **Multiplication, Division, Floor Division, Modulus** `*`, `/`, `//`, `%`
4. **Addition, Subtraction** `+`, `-`

Associativity:

- Most operators are **left-to-right associative** (evaluated from left to right).
- Exponentiation (`**`) is **right-to-left associative**.

Example:

```
result = 2 + 3 * 4    # 2 + (12) = 14
power = 2 ** 3 ** 2   # 2 ** (9) = 512
```

1.11.3 Applications of Arithmetic Operators and Expressions

1. **Engineering Calculations** – Used for mathematical modeling, solving equations, and scientific simulations.
2. **Data Analysis** – Helps in statistical computations like mean, variance, and regression analysis.
3. **Control Systems** – Expressions are applied in feedback loop calculations and signal processing.
4. **Graphics and Game Development** – Arithmetic is used in positioning, scaling, and motion calculations.
5. **Financial Applications** – Useful in interest calculation, budgeting, and forecasting.

1.12 BUILT-IN FUNCTIONS

Python provides a rich collection of **built-in functions** that can be used directly without importing any external library. These functions simplify programming by performing commonly needed tasks such as mathematical operations, type conversions, input/output handling, and data manipulation.

Characteristics of Built-in Functions

1. Available by default in Python (no need to import).
2. Can be used across different data types and applications.
3. Improve efficiency by reducing the need for manual implementation.
4. Often serve as building blocks for larger programs.

Commonly Used Built-in Functions

Function	Description	Example	Output
<code>print()</code>	Displays output on the	<code>print("Hello")</code>	Hello

Function	Description	Example	Output
	screen		
input()	Reads user input as a string	x = input("Enter: ")	User input
len()	Returns length of a sequence	len("Python")	6
type()	Displays the data type of a variable	type(10)	<class 'int'>
int()	Converts a value to integer	int("5")	5
float()	Converts a value to float	float("3.2")	3.2
str()	Converts a value to string	str(25)	"25"
max()	Returns maximum value from a sequence	max(4, 9, 2)	9
min()	Returns minimum value from a sequence	min(4, 9, 2)	2
abs()	Returns absolute value	abs(-7)	7
round()	Rounds a number to nearest integer or given decimals	round(3.75, 1)	3.8
sum()	Returns sum of elements in a sequence	sum([1, 2, 3])	6

Example Program

Demonstrating built-in functions

```
a = -15
```

```
b = [2, 4, 6, 8]
print("Absolute value of a:", abs(a))
print("Maximum value in list:", max(b))
print("Sum of list:", sum(b))
print("Type of a:", type(a))
```

Output:

Absolute value of a: 15

Maximum value in list: 8

Sum of list: 20

Type of a: <class 'int'>

Applications of Built-in Functions

1. **Data Analysis** – Functions like `sum()`, `max()`, `min()` are widely used in statistical and numerical analysis.
2. **User Interaction** – `input()` and `print()` support user-driven programs and command-line applications.
3. **Type Conversion** – Functions like `int()`, `float()`, `str()` are used in handling user inputs and file data.
4. **Scientific Computation** – Built-in mathematical functions simplify calculations before applying advanced libraries.
5. **Automation and Scripting** – Essential in writing small scripts for system tasks and automation.

1.13 IMPORTING FROM PACKAGES

In Python, a **package** is a collection of modules organized in a directory structure. To use the functions, classes, or variables defined in these modules, they must be **imported** into the program. Importing allows code reusability, modular programming, and efficient development.

Syntax for Importing

Python provides multiple ways to import from packages:

1. Import Entire Module

```
import math
```

```
print(math.sqrt(16)) # Using function with module name
```

2. Import Specific Functions/Classes

```
from math import sqrt, pi
```

```
print(sqrt(25))
```

```
print(pi)
```

3. Import Module with Alias

```
import numpy as np
```

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

4. Import All Contents (Not Recommended)

```
from math import *  
  
print(sin(0))
```

Commonly Used Standard Packages

- **math** – Provides mathematical functions like `sqrt()`, `sin()`, `cos()`.
- **random** – Used for generating random numbers.
- **datetime** – Deals with date and time.
- **os** – Provides operating system-related functions (file handling, directory operations).
- **sys** – Provides system-specific parameters and functions.

Example Program

```
# Importing from math and random packages
```

```
import math  
import random  
num = 49  
print("Square root of num:", math.sqrt(num))  
print("Random number between 1 and 10:", random.randint(1,  
10))
```

Output:

```
Square root of num: 7.0  
Random number between 1 and 10: 6
```

Applications of Importing from Packages

1. **Scientific Computation** – Using math, numpy, scipy for engineering and research calculations.
2. **Data Science and AI** – pandas, matplotlib, scikit-learn for data processing and machine learning.
3. **System Programming** – os and sys for file operations and process management.
4. **Web Development** – Frameworks like flask and django are imported as packages.
5. **Automation** – Packages like time, shutil are used in scripting and automation tasks.

WHY PROGRAMMING LANGUAGE?

1. Time Management.
2. Project Implementation.
3. Communication Process.
4. Calculation.
5. S/W and H/W.
6. Web Developing.

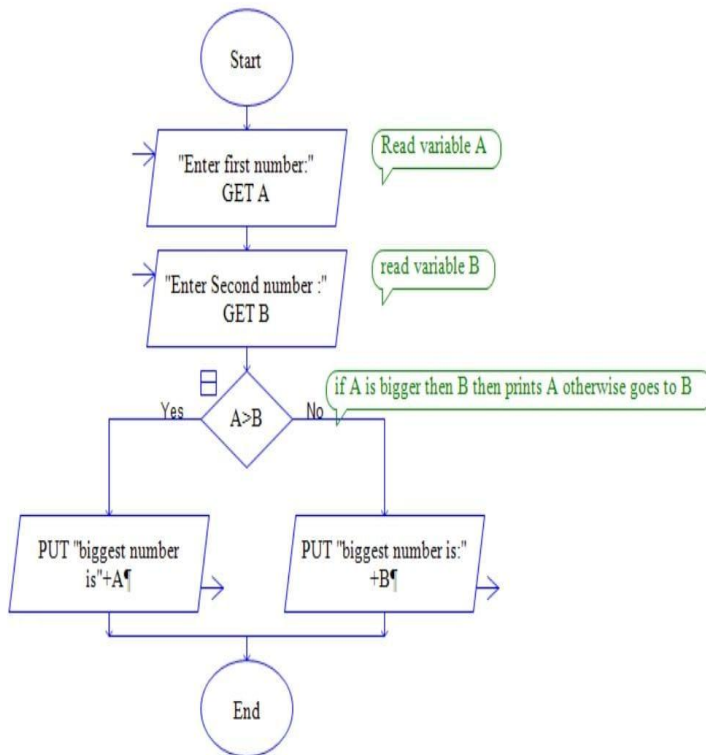
1.14 PRACTICAL EXERCISES

Problem 1: Find the Largest of Two Numbers

Problem Analysis

- **Input:** Two numbers (a, b)
- **Process:** Compare a and b
- **Output:** Largest number

Flowchart



Pseudocode

Start

Input a, b

If $a > b$ then

Print "Largest =", a

Else

Print "Largest =", b

EndIf

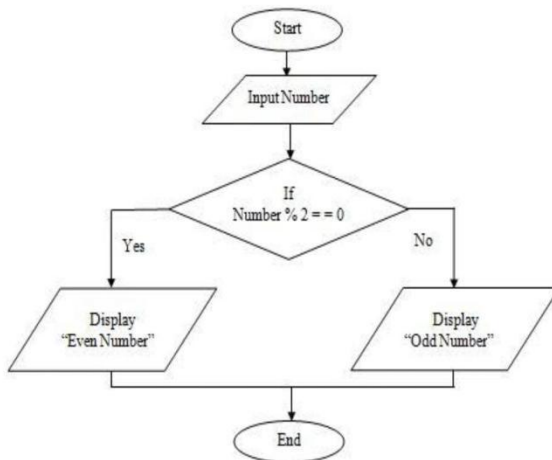
Stop

Problem 2: Check Whether a Number is Even or Odd

Problem Analysis

- **Input:** A number (n)
- **Process:** Divide n by 2, check remainder
- **Output:** Even or Odd

Flowchart



Pseudocode

Start

Input n

If $n \% 2 == 0$ then

Print "Even"

Else

Print "Odd"

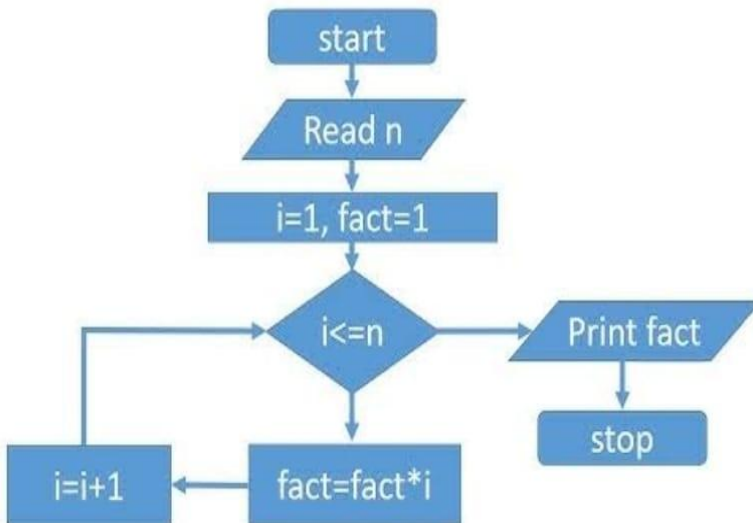
EndIf
Stop

Problem 3: Calculate Factorial of a Number

Problem Analysis

- **Input:** A positive integer (n)
- **Process:** Multiply numbers from 1 to n
- **Output:** Factorial value (n!)

Flowchart



Pseudocode

Start

Input n

fact \leftarrow 1

For i = 1 to n

fact \leftarrow fact * i

EndFor

Print "Factorial =", fact

Stop

PYTHON:

- Python is a high level, interpreted, interactive and object oriented- scripting language.
- Python was designed to be highly readable which uses English keyword frequently where as other languages use punctuation and it has fewer syntactical constructions than other languages.
- Python is an interpreted, interactive object oriented and beginner's friendly language.

HISTORY OF PYTHON:

- Python was developed by guide Van Rossum in the late eighties and early nineties at the National research Institute for Mathematics and Computer science in the Netherlands.

- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol- 68, Small Talk and Unix shell and other scripting languages.
- Python is copyrighted like Perl, Python source code is now available under GNU General Public License (GPU)
- Python is now maintained by a core development team as the institute although Guido Van Rossum still holds a vital role in directing in progress.
- **Python is Interpreted:** This means that it is processed at runtime by the interpreter and go do need to complete your program before executing it. This is similar to PERL and PHP
- **Python Interactive:** You can actually sit at a python prompt and interact with the interpreter directly to write your program
- **Python is Object – Oriented:** Python supports object oriented style or technique of programming that encapsulates code within objects.

PYTHON IS A BEGINNER’S LANGUAGE:

Python is the great language for the Beginner- level programmers and supports the development of a wide range of applications.

Application:

- Bit torrent file sharing
- Google search engine, youtube
- Inter, cisco, HP, IBM
- I-Robot
- NASA
- Face book, Drop box

Example :

```
def newton_sqrt (n):
```

```
    approx = 0.5*n
```

```
    better = 0.5* (approx + n / approx)
```

```
    while better != approx
```

```
        approx = better
```

```
        better = 0.5* (approx + n/approx)
```

```
    Return approx
```

```
S= int (input ("Enter the number"))
```

```
Print ('The square root is', newton_sqrt)
```

1.15 IMPORTANT TWO MARKS :**1) What is problem solving?**

Problem solving is the step-by-step process of identifying a problem, analyzing it, and creating a sequence of instructions to achieve a correct solution. It helps in developing logical thinking.

2) What is problem analysis?

Problem analysis is the stage where the problem is clearly understood by identifying inputs, outputs, and constraints. It ensures clarity before designing the solution.

3) What is an algorithm?

An algorithm is a finite sequence of well-defined steps used to solve a problem. It must always terminate and give the expected output.

4) State two characteristics of an algorithm.

An algorithm must be finite, meaning it should end after a limited number of steps. It must also be unambiguous so that every step is clearly defined.

5) What is a flowchart?

A flowchart is a graphical representation of an algorithm using standard symbols. It shows the logical flow of the program from start to end.

6) Give two flowchart symbols with their meaning.

Oval represents the Start or End of a program. Parallelogram is used to show Input and Output operations.

7) **What is pseudocode?**

Pseudocode is a method of expressing algorithms in plain English with programming-like structures. It helps in designing before actual coding.

8) **What is interactive mode in Python?**

Interactive mode executes Python statements line by line in the Python shell. It is useful for testing small code snippets quickly.

9) **What is script mode in Python?**

Script mode allows writing Python code in a file and running it as a complete program. It is used for larger and reusable programs.

10) **What is indentation in Python?**

Indentation is the use of spaces or tabs at the beginning of a line to indicate a block of code. Python relies on indentation instead of braces.

11) **How are comments written in Python?**

Single-line comments use # and multi-line comments are enclosed within triple quotes ("\" ... \"). Comments improve readability and are ignored by the interpreter.

12) **What are error messages?**

Error messages are system-generated notifications that indicate mistakes in the code. They may be syntax errors, runtime errors, or logical errors.

13) **What is a variable?**

A variable is a named memory location that stores data

which can change during program execution. Example: `x = 10`.

14) State two rules for naming variables.

Variable names must not begin with digits and should not use reserved keywords. They can contain letters, digits, and underscores.

15) What are reserved words?

Reserved words are special keywords predefined by Python. They have fixed meanings and cannot be used as variable names, e.g., `if`, `while`, `class`.

16) List four Python data types.

Common data types are: `int` (integer values), `float` (decimal values), `str` (string/text), and `bool` (Boolean values `True/False`).

17) What are arithmetic operators?

Arithmetic operators are symbols used to perform basic mathematical operations. Examples: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulus).

18) Define expression with example.

An expression is a combination of variables, values, and operators that produces a result. Example: `a + b * 5`.

19) Give two built-in functions in Python.

The `print()` function displays output on the screen. The `len()` function returns the length of a sequence like a string or list.

20) Write an example of importing from a package.

Python allows importing functions from packages to reuse code. Example: from math import sqrt imports the square root function.

UNIT II

CONTOL STRUCTURES

If, if-else, nested if, multi-way if-elif statements, while loop, for loop, nested loops, pass statements. Practical: Usage of conditional logics in programs. (Minimum three)

2.1 INTODUCTION TO CONTROL FLOW

Control flow refers to the **order in which individual instructions, statements, or function calls are executed** within a program. In Python and other programming languages, control flow is determined by **decision-making statements, loops, and function calls**, which allow programmers to alter the natural top-to-bottom execution sequence.

1. Sequential Execution

By default, program statements execute one after another in the order they appear.

Example: Assigning variables and printing values.

2. Decision Making (Selection)

Control flow can branch based on conditions using if, if-else, and if-elif-else.

Example: Checking if a number is positive or negative.

3. Repetition (Iteration)

Loops (for, while) allow repeating a block of code until a condition is met.

Example: Printing numbers from 1 to 10.

4. Transfer of Control

Special keywords like break, continue, and pass can alter the normal flow within loops.

5. Function Calls

Execution can jump to a function block and return to the caller after completion.

2.1.1 FUNCTIONS

Function is a sub program which consists of sets of instructions used to perform a specific task, a large perform is divided into basic building blocks called functions.

Need for function:

When the program is too complex and large they are divided into part. Each is separately coded and combined into single program. Each subprogram is called as function.

- Debugging, Testing and maintenance becomes easy when the program is divided into subprograms.
- Functions are used to avoid rewriting same code again and again in a program
- Function provides code re-usability.
- The length of the program is reduced.

Types of function:

1. User-defined function (programmer create)
2. Built-in function (already created & Stored)

Function Definition: (sub program)

- Def keyword is used to define a fn
- Give the fn name after def keyword followed by which arguments are given
- End with colon (:)
- Inside the fn add the prgm statements to be executed
- End with or without return statement

Syntax:

```
def fun-name (Parameter 1,.. Parametern):
```

```
Statement 1..... Statements
```

```
return [expression]
```

Eg:

```
def my_add (a,b):
```

```
    c = a+b
```

```
    return c.
```

function name → my_add

Statement → c=a+b

Function Prototype:

- Function without Argument & return & without return types
- Function with Argument & without return type
- Function without Argument & with return type
- Function with Argument & with return type

Without Return Type

Without argument:

```
def add ():
```

```
    a = int (input("enter a :"))
```

```
    b = int (input("enter b:"))
```

```
    c = a+b
```

```
    print (c)
```

add ()

With argument:

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

```
    c = a+b
```

```
    print (c)
```

```
    a = int (input("enter a:"))
```

```
    b = int (input("enter b:"))
```

```
add (a, b)
```

With Return Type:

Without argument:

```
def add ():
```

```
    a = int (input("enter a:"))
```

```
    b = int (input("enter b:"))
```

```
    c = a+b
```

```
    Return c
```

```
    Print (c)
```

```
add ()
```

With argument:

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

```
    C = a+b
```

```
    Return c
```

```
    Print (c)
```

```
    a = int (input("enter a:"))
```

```
    b = int (input("enter b:"))
```

```
add (a, b)
```

2.2 DECISION-MAKING STATEMENTS

2.2.1 if ...

2.2.2 if...else

2.2.3 if...elif...else

2.2.4 if...if

2.2.1 .Conditional (if):

conditional (if) is used to test a condition, if the condition is true the statements inside if will be executed.

Syntax:

i. if condition:

```
    if condition:
```

Statement

Eg:

If $a < 10$:

Print ("The number is less than 10")

Flowchart:

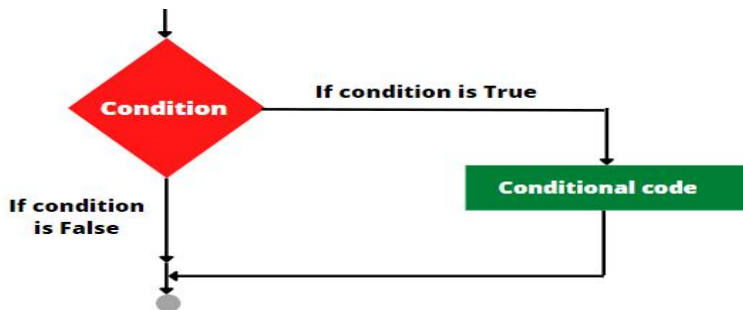


Fig: Flowchart of single selection if statement

EXAMPLE :

Program to provide bonus mark if the category is sports	output
<pre> m=eval(input("enter ur mark out of 100")) c=input("enter ur category G/S") if(c=="S"): m=m+5 print("mark is",m) </pre>	<pre> enter your mark out of 100 85 enter your category G/S S mark is 90 </pre>

2.2.2 if – else condition (alternative statement)

In the alternative the condition must be true or false. In this else statement can be combined with if statement. The else statement contains the block of code that executes when the condition is false. If the condition is true statements inside the if get executed otherwise else part gets executed.

The alternatives are

called branches, because they are branches in the flow of execution.

Syntax:

if condition:

Statement 1

else:

Statement 2

Eg:

if a<10:

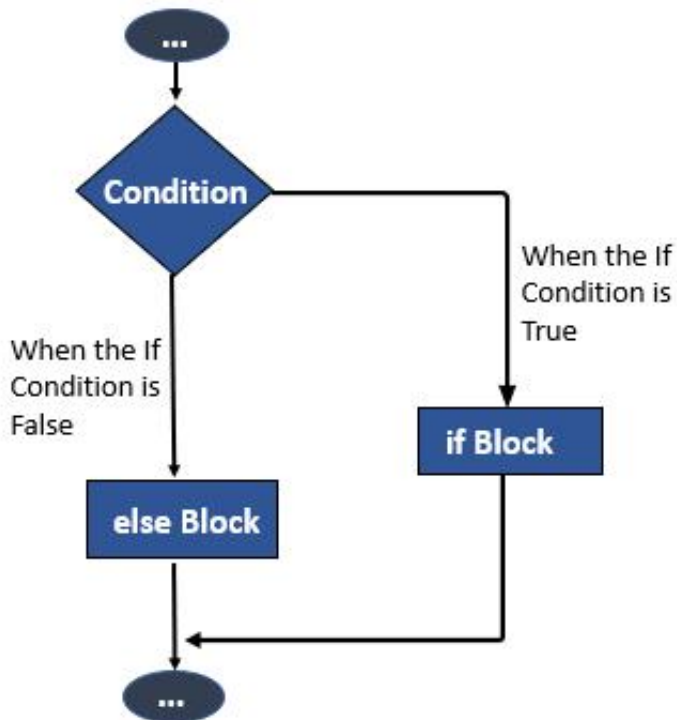
Print (“The number is less than 10”)

else 2

Print (“The number is not less than 10”)

Examples:

1. odd or even number
2. positive or negative number
3. leap year or not

Flowchart:

2.2.3 Chained condition:

if ...elif...else

- The elif is short for else if.
- This is used to check more than one condition.
- If the condition1 is False, it checks the condition2 of the elif block. If all the conditions are False, then the else part is executed.
- Among the several if...elif...else part, only one part is executed according to the condition.
- The if block can have only one else block. But it can have multiple elif blocks.
- The way to express a computation like that is a chained conditional.

Syntax:

```
if condition:
    .
    .
elif condition:
    .
    .
else: statement
```

Eg:

```
def datatypes():
```

```
if a = A,a,B,b.....Z,z:
```

```
print ("a:", type (a))
```

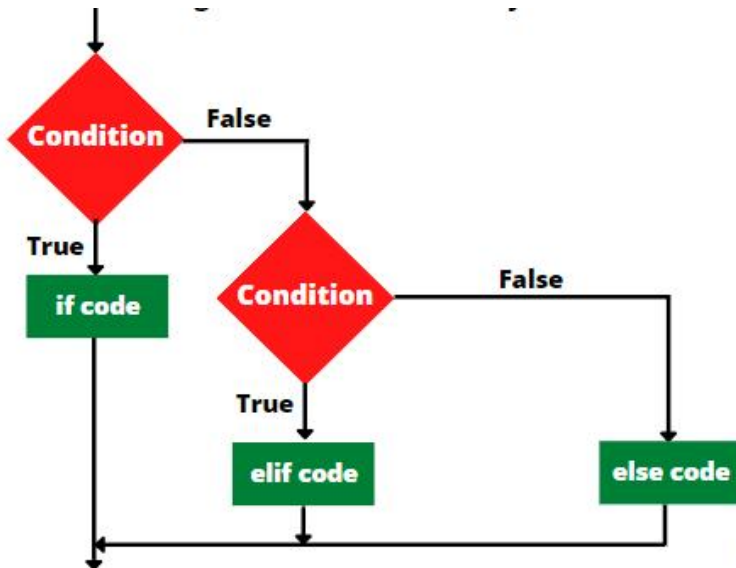
```
elif a=0,1,....:
```

```
print ("a:", type (a))
```

```
else:
```

```
print ("invalid data")
```

Flow Chart:



2.3 ITERATIVE CONSTRUCTS

2.3.1 FOR LOOP

The for loop is used to iterate over a sequence or other iterable object. It executes the block of code once for each item in the sequence.

Syntax:

For variable in sequence: Body of for loop.

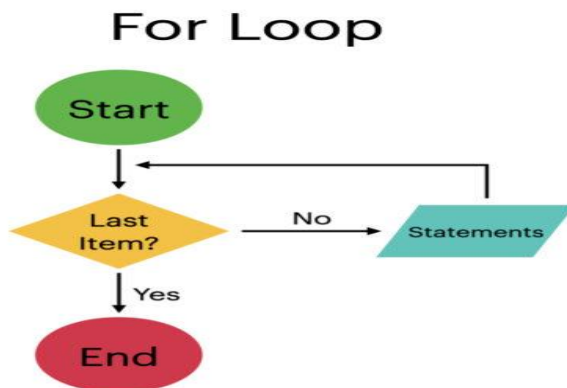
Eg:

```
for i in range(10):  
    print(i)
```

O/P:

1,2,3,4,5,6,7,8,9.

Flow chart



2.3.2 WHILE LOOP

While loop repeatedly executes a block of code as long as a specified condition is true. If the condition becomes false, the loop terminates.

Syntax:

```
While condition  
# code block
```

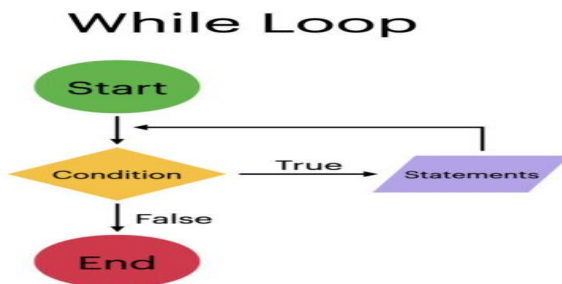
Eg:

```
Count = 0  
While count < 5  
Print (count)  
Count = 1
```

O/P:

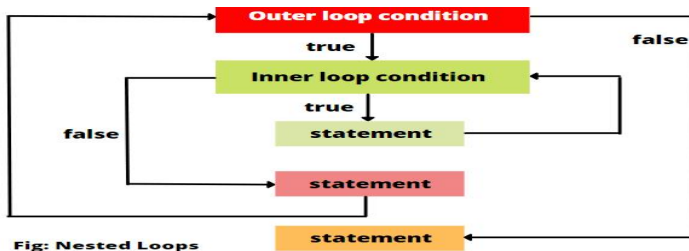
0,1,2,3,4

Flowchart:



2.3.3 NESTED LOOP

Flow Chart :



A **nested while loop** is a while loop inside another while loop. The inner loop completes all its iterations for every single iteration of the outer loop.

Syntax:

while condition1:

 while condition2:

 # code block of inner loop

 # code block of outer loop

Example:

```
I = 0
while i < 3:    # Outer loop
```

```
    j = 0
```

```
    while j < 2:    # Inner loop
```

```
        print(f"i = {i}, j = {j}")
```

```
        j = j + 1
```

```
        i = i + 1    Output: i = 0, j = 0 i = 0, j = 1 i = 1, j = 0 i = 1, j = 1 i = 2, j = 0 i = 2,
```

```
        j = 1
```

2.4 JUMP STATEMENTS

2.4.1 BREAK STATEMENT

The break statement is used to exit a loop permanently. Regardless of the loop condition when encountered, it immediately terminates the inner most loop.

Syntax: break

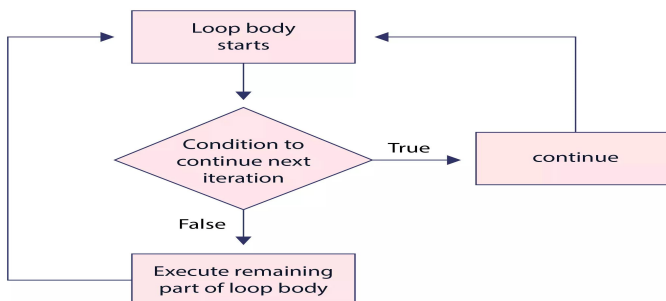
Eg:

```
for i in range (10):  
    if 1 == 3  
        break  
  
print (i)
```

O/P:

1,2

Flowchart:



2.4.2 CONTINUE STATEMENT

- The continue statement skips the current iteration of the loop and moves to the next iteration
- It does not exit the loop

Syntax:

Continue

Eg:

```
for i in range (10):
```

```
    if i == 3
```

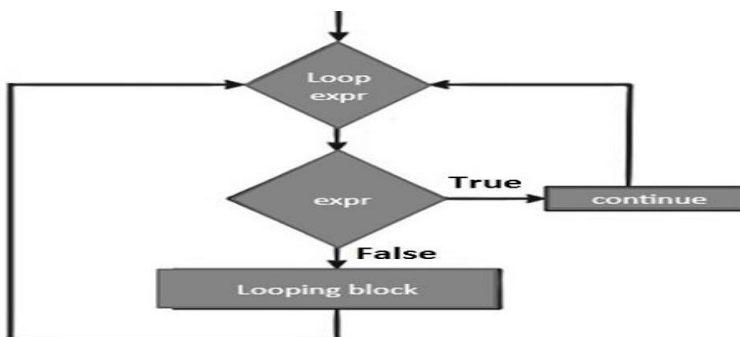
```
        Continue
```

```
    print (i)
```

O/P:

1,2,3,4,5,6,7,8,9

Flow Chart:



2.4.3 PASS: (DO NOTHING)

Pass statement is a placeholder

It is used when a statement is syntactically required but you don't want to execute any code

Eg:

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

```
    if i == 3
```

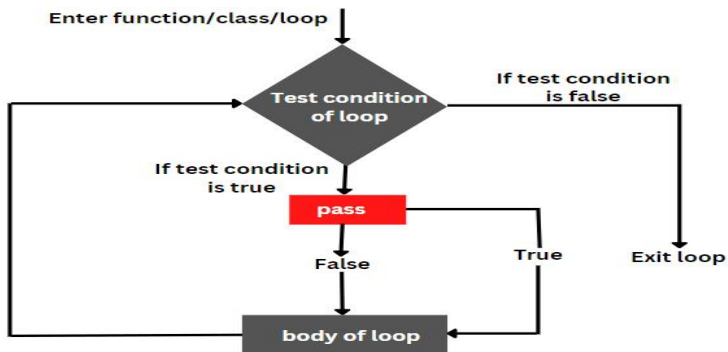
```
        pass
```

```
print(i)
```

O/P:

1,2,3,4

Flow chart :



2.5 FRUITFUL FUNCTIONS

Function that returns values are called as fruitful functions.

Input the value → fruitful fun → return value

- Return values
- Parameters
- Local and global scope
- Function composition
- recursion

2.5.1 Return values

A fruitful function is one that computes a result and returns a value to the caller using the return statement

Syntax:

```
def function _ name (parameter)
```

```
    # function logic
```

```
return value
```

Eg:

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

```
    return a +b
```

```
result = add (3,5)
```

Print (result)

O/P: 7

2.5.2 Parameters

Parameters are the values that are passed into a function. When its called they are place holders that allow the function to accept input and perform its task with different values.

Syntax:

```
def _ function name (p1, p2):
```

```
    # function body
```

Eg:

```
def greet (name):
```

```
    return f "hello, {name} !"
```

```
message = greet ("joe")
```

```
print (message)
```

O/P:

Hello Joe!

2.5.3 Function composition

It is the process of combining two or more functions to produce a new fn.

In python you can compose functions by calling one fn inside another fn.

Eg:

```
def square(x):
```

```
    return x*x
```

```
def double (x):
```

```
    return x*2
```

```
def square _ then _ double (x)
```

```
    return double (square(x))
```

```
result = square _ then _ double(3)
```

```
print (result)
```

2.5.4 Recursion

Property in which one fn call its repeatedly parameters are changed on each all

Syntax:

```
def recursive function (parameter)
```

```
    if base _ condition
```

```
        return base value
```

else:

 return recursive_ function (modified _ parameter)

Eg:

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

O/P:

120

2.6 PRACTICAL

Practical Model – 1: Pass or Fail

Q: Write a program to check whether a student has passed or failed using conditional logic.

Algorithm

1. Start
2. Read the marks of the student

3. If marks ≥ 50 , then display "Pass"
4. Else, display "Fail"
5. Stop

Pseudocode

BEGIN

 READ marks

 IF marks ≥ 50 THEN

 PRINT "Pass"

 ELSE

 PRINT "Fail"

 ENDIFEND

Flowchart

Draw Flowchart Manually

Python Program

```
# Program to check pass or fail

marks = int(input("Enter your marks: "))

if marks  $\geq 50$ :
    print("Pass")else:
```

```
print("Fail")
```

Sample Output

Enter your marks: 65Pass

Enter your marks: 40Fail

Practical Model – 2: Age Validation

Q: Write a program to check if a person is a minor, adult, or invalid age.

Algorithm

1. Start
2. Read age of the person
3. If age < 0, display "Invalid age"
4. Else if age < 18, display "Minor"
5. Else, display "Adult"
6. Stop

Pseudocode

BEGIN

 READ age

 IF age < 0 THEN

 PRINT "Invalid age"

```
ELSE IF age < 18 THEN
```

```
    PRINT "Minor"
```

```
ELSE
```

```
    PRINT "Adult"
```

```
ENDIFEND
```

Flowchart

Draw Flowchart Manually

Python Program

```
# Program for age validation
```

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

```
if age < 0:
```

```
    print("Invalid age")elif age < 18:
```

```
    print("Minor")else:
```

```
    print("Adult")
```

Sample Output

```
Enter your age: 15Minor
```

```
Enter your age: 23Adult
```


Enter your age: -5Invalid age

Practical Model – 3: Menu Driven Calculator

Q: Write a program to perform addition or subtraction based on user's choice.

Algorithm

1. Start
2. Display menu with choices: 1. Addition, 2. Subtraction
3. Read user choice
4. If choice = 1, read two numbers and display their sum
5. Else if choice = 2, read two numbers and display their difference
6. Else, display "Invalid choice"
7. Stop

Pseudocode

BEGIN

DISPLAY "1. Addition"

DISPLAY "2. Subtraction"

READ choice

IF choice = 1 THEN

 READ a, b

```
    PRINT a + b

ELSE IF choice = 2 THEN

    READ a, b

    PRINT a - b

ELSE

    PRINT "Invalid choice"

ENDIFEND
```

Flowchart

Draw Flowchart Manually

Python Program

```
# Menu-driven calculator
print("1. Addition")
print("2. Subtraction")
```

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

if choice == 1:

    a = int(input("Enter first number: "))

    b = int(input("Enter second number: "))

    print("Result:", a + b)
elif choice == 2:
```

```
a = int(input("Enter first number: "))  
b = int(input("Enter second number: "))  
print("Result:", a - b)else:  
print("Invalid choice")
```

Sample Output

1. Addition2. SubtractionEnter your choice: 1Enter first number:
20Enter second number: 15Result: 35

1. Addition2. SubtractionEnter your choice: 3Invalid choice

Practical Model – 4: Largest of Three Numbers

Q: Write a program to find the largest among three numbers.

Algorithm

1. Start
2. Read three numbers a, b, c
3. If $a > b$ and $a > c$, then a is largest
4. Else if $b > c$, then b is largest
5. Else, c is largest
6. Stop

Pseudocode

BEGIN

```
READ a, b, c

IF a > b AND a > c THEN

    PRINT "a is largest"

ELSE IF b > c THEN

    PRINT "b is largest"

ELSE

    PRINT "c is largest"

ENDIF

END
```

Flowchart

Draw Flowchart Manually

Python Program

```
# Program to find largest of three numbers

a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = int(input("Enter third number: "))

if a > b and a > c:

    print("Largest number is:", a)elif b > c:
```

```
print("Largest number is:", b)else:
```

```
print("Largest number is:", c)
```

Sample Output

Enter first number: 10Enter second number: 25Enter third
number: 7Largest number is: 25

2.6 IMPORTANT TWO MARKS

1. What is the use of the if statement in Python?

The if statement is used to test a condition. If the condition is true, the block of code under if is executed.

2. Differentiate between if and if-else.

if executes a block only when the condition is true, while if-else executes one block if the condition is true and another block if it is false.

3. What is a nested if statement?

A nested if is an if statement inside another if statement. It is used when multiple levels of decision-making are needed.

4. What is an if-elif ladder (multi-way decision)?

The if-elif ladder is used when multiple conditions need to be checked sequentially. Only one block of code is executed

depending on the condition that evaluates to true.

5. **Write a simple syntax of while loop.**

while condition:

 # statements

It executes repeatedly until the condition becomes false.

6. **What is the use of a for loop in Python?**

The for loop is used for iteration over a sequence (like list, string, range). It executes once for each item in the sequence.

7. **What is a nested loop?**

A nested loop is a loop inside another loop. For each iteration of the outer loop, the inner loop executes fully.

8. **What is the purpose of the pass statement?**

The pass statement is a null operation. It is used as a placeholder when no action is required in a block.

9. **Write the output of the following code:**

```
for i in range(3):
```

```
    for j in range(2):
```

```
        print(i, j)
```

Output:

0 0

0 1

1 0

1 1

2 0

2 1

10. **What happens if we forget to update the condition inside a while loop?**

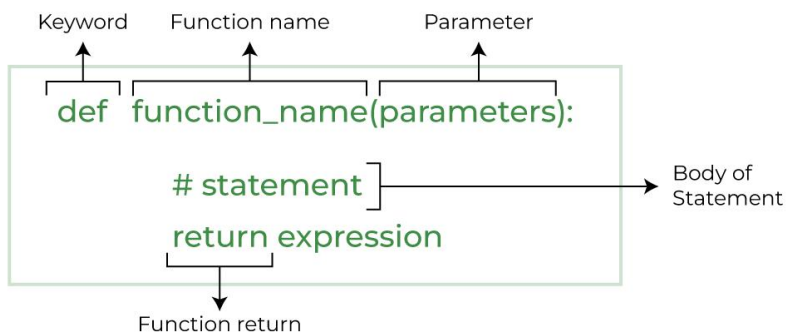
If the condition always remains true, the loop runs infinitely, leading to an **infinite loop**.

UNIT III

FUNCTIONS

Hiding redundancy, complexity; Parameters, arguments and return values; formal vs actual arguments, named arguments, Recursive & Lambda Functions. Practical: Usage of functions in programs. (Minimum three)

3. INTRODUCTION TO FUNCTIONS



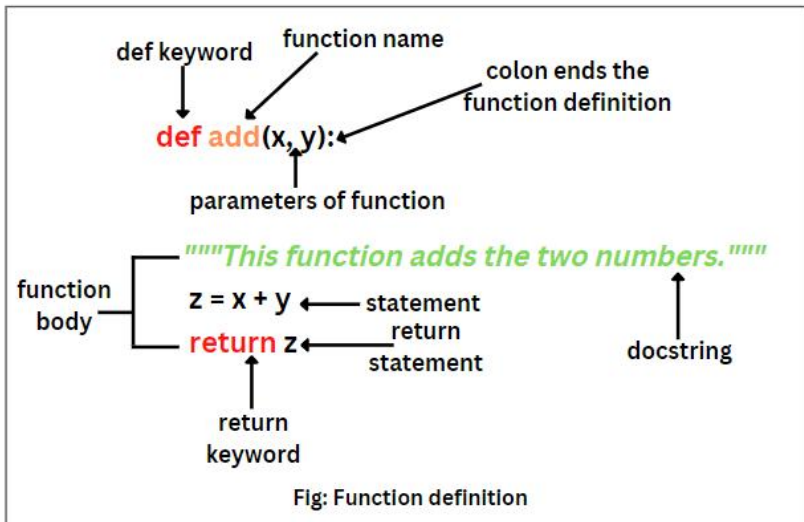
Definition

A **function** is a reusable block of code that performs a specific task. Instead of writing the same code again and again, we put it inside a function and reuse it.

Why Functions are Needed?

- **Hiding redundancy:** Repeated code can be avoided.

- **Hiding complexity:** Large programs can be divided into smaller, simple parts.
- **Reusability:** Write once, use many times.
- **Readability:** Programs become easier to understand.



Example Program

Without function

```
print("Area of rectangle:", 10 * 5)  
print("Area of rectangle:", 7 * 4)
```

With function

```
def area(length, width):
```

```
    return length * width
```

```
print("Area of rectangle:", area(10, 5))
```

```
print("Area of rectangle:", area(7, 4))
```

Output

Area of rectangle: 50Area of rectangle: 28

3.1 NEED FOR FUNCTIONS: HIDING REDUNDANCY & COMPLEXITY

Definition

In programming, **functions** are used to *hide redundancy* and *reduce complexity*.

Hiding Redundancy

- Redundancy means repeating the same code multiple times.
- Functions help us avoid writing the same block of code again and again by grouping it into a reusable function.

Hiding Complexity

- Complex problems can be divided into smaller, easy-to-understand tasks using functions.
- Each function handles only one task, making the overall program easier to read, understand, and debug.

Example 1: Without Function (Redundant Code)

```
# Program without using functions
```

```
print("Square of 2 is:", 2*2)

print("Square of 3 is:", 3*3)

print("Square of 4 is:", 4*4)
```

Output:

```
Square of 2 is: 4
Square of 3 is: 9
Square of 4 is: 16
```

Here, the logic $x*x$ is repeated again and again → **Redundancy**.

Example 2: With Function (Redundancy Removed)

```
def square(x):      # Function definition
    return x * x

# Function call

print("Square of 2 is:", square(2))

print("Square of 3 is:", square(3))

print("Square of 4 is:", square(4))
```

Output:

```
Square of 2 is: 4
Square of 3 is: 9
Square of 4 is: 16
```

Now, the code `x*x` is written only **once** inside the function → **Redundancy hidden**.

Example 3: Hiding Complexity with Functions

```
def get_marks():  
    return int(input("Enter marks: "))  
  
def calculate_grade(marks):  
    if marks >= 90:  
        return "A"  
    elif marks >= 75:  
        return "B"  
    elif marks >= 50:  
        return "C"  
    else:  
        return "Fail"  
  
def display_result(grade):  
    print("Your Grade is:", grade)  
  
# Main Program  
marks = get_marks()
```

```
grade = calculate_grade(marks)
```

```
display_result(grade)
```

Output (Sample Run):

Enter marks: 82Your Grade is: B

Here, the **complex task (grading system)** is split into 3 smaller functions:

1. `get_marks()` – Input handling
2. `calculate_grade()` – Logic part
3. `display_result()` – Output display

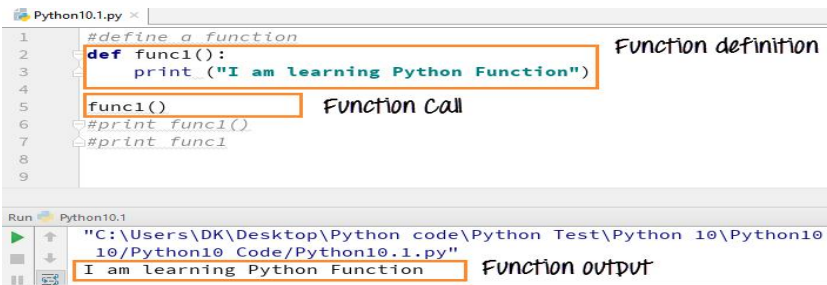
This makes the program easier to read and maintain → **Complexity hidden.**

3.2 FUNCTION DEFINITION AND CALLING

A function in Python is defined using the `def` keyword, followed by the function name, parameters, and a block of statements. Functions must be called to execute their code.

Syntax:

```
def function_name(parameters):  
    """Optional docstring: explains purpose of the function"""  
    # block of code  
    return value
```



The screenshot shows a Python IDE window titled 'Python10.1.py'. The code is as follows:

```
1 #define a function
2 def func1():
3     print ("I am learning Python Function")
4
5 func1()
6 #print func1()
7 #print func1
8
9
```

Annotations in the image:

- 'Function definition' points to the `def func1():` line.
- 'Function Call' points to the `func1()` line.

Below the code editor, the 'Run' output window shows the execution path and the output:

```
"C:\Users\DK\Desktop\Python code\Python Test\Python 10\Python10
10\Python10 Code\Python10.1.py"
I am learning Python Function
```

Annotations in the image:

- 'Function output' points to the output string 'I am learning Python Function'.

Fig 3.2 Function Definition And Calling

Example:

```
def greet():
    print("Welcome to Python Functions!")
```

```
# Function call
greet()
```

- **Definition:** Creates the function and assigns it a name.
- **Call:** Executes the function by its name, optionally passing arguments.

3.3 PARAMETERS AND ARGUMENTS

Functions can receive **inputs** through parameters and arguments.

- **Parameters:** Variables defined inside parentheses of the function definition.
- **Arguments:** Actual values or variables passed during the function call.

The diagram illustrates the components of a Python function. It shows a function definition: `# Function Definition`
`def add(a, b):`
 `return a + b`
and a function call: `# Function Call`
`add(2, 3)`. A callout bubble labeled "Parameters" points to the variables `a` and `b` in the function definition. Another callout bubble labeled "Arguments" points to the values `2` and `3` in the function call.

Example:

```
def add(x, y): # x, y are parameters
    return x + y

print(add(5, 7)) # 5, 7 are arguments
```

Types of Arguments:

- **Positional Arguments** – Order matters.
- **Default Arguments** – Provide fallback values.
- **Variable-Length Arguments** – `*args` (non-keyword), `**kwargs` (keyword arguments).

3.4 RETURN VALUES

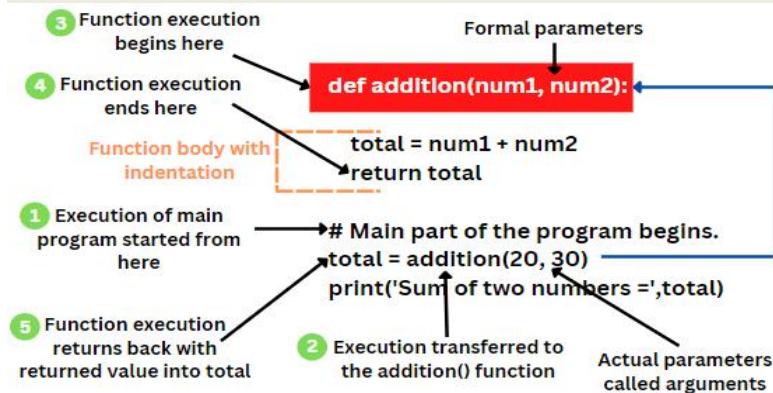
Functions can send results back using the return statement.

- **Single Return:** Function returns one value.
- **Multiple Returns:** Functions can return tuples containing multiple values.

Example:

```
def divide(a, b):  
    quotient = a // b  
    remainder = a % b  
    return quotient, remainder
```

```
q, r = divide(17, 5)  
print("Quotient:", q, "Remainder:", r)
```



Here, the function hides the complexity of division and exposes clear output.

3.5 FORMAL VS ACTUAL ARGUMENTS

Definition

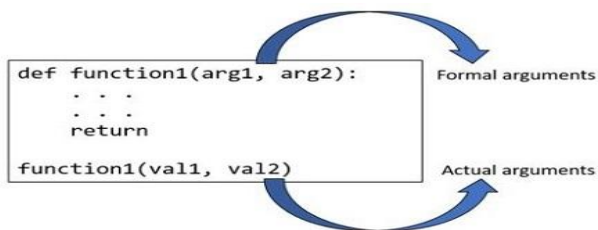
Formal Arguments

- The variables declared in the function definition.

- They act as placeholders to receive values when the function is called.
- Example: In `def add(a, b):`, `a` and `b` are **formal arguments**

Actual Arguments

- The real values (or variables) that are passed when calling the function.
- They replace the formal arguments during execution.
- Example: In `add(5, 10)`, the values `5` and `10` are **actual arguments**.



Example Program 1: Simple Addition

```
def add(a, b):    # a and b are formal arguments  
  
    print("Sum:", a + b)  
  
add(5, 10)       # 5 and 10 are actual arguments
```

Output:

Sum: 15

Example Program 2: Greeting Function

```
def greet(name):    # name → formal argument
    print("Hello,", name)
greet("Daniel")    # "Daniel" → actual argument
```

Output:

Hello, Daniel

This distinction is important because **formal arguments act as placeholders**, while **actual arguments supply data**.

Basis	Formal Argument	Actual Argument
Definition	Variables in the function definition	Values given in the function call
Existence	Exists only inside the function	Exists in the calling program
Example (in add(a,b))	a, b	5, 10 in add(5,10)

3.6 NAMED ARGUMENTS

In Python, arguments can be passed by explicitly mentioning the parameter name, making code **clearer** and avoiding positional confusion.

Example:

```
def student(name, age, course):  
    print(name, "is", age, "years old studying", course)  
  
student(age=19, course="CSE", name="Rahul")
```

Advantages:

- Improves readability
- Reduces errors in large functions
- Allows skipping some parameters if defaults are defined

3.7 RECURSIVE FUNCTIONS

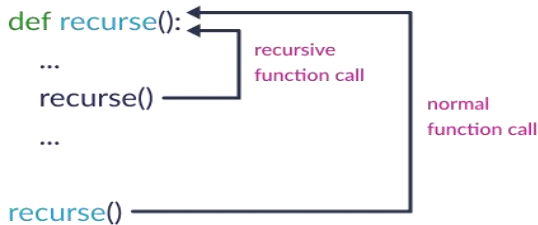
A recursive function is one that calls itself. It is often used to solve problems that can be broken down into smaller, similar sub-problems.

Rules for Recursion:

1. A **base case** must stop recursion.
2. Each recursive call should simplify the problem.

Example – Factorial:

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n - 1)  
  
print("Factorial of 5 is", factorial(5))
```



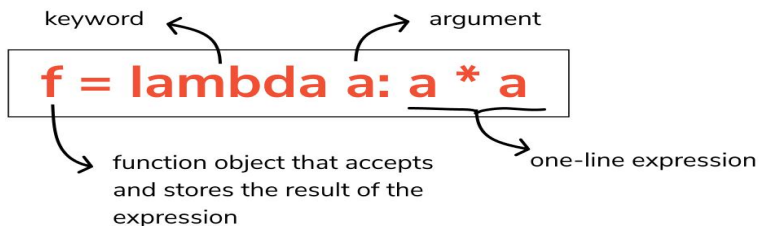
- **Advantages:** Reduces code length, natural representation of mathematical definitions.
- **Disadvantages:** Higher memory usage (stack calls), risk of infinite recursion if base case is missing.

3.8 LAMBDA (ANONYMOUS) FUNCTIONS

Lambda functions are small, unnamed functions created with the lambda keyword. They are **useful for short operations** where defining a full function is unnecessary.

Syntax:

lambda arguments: expression



Example:

```
square = lambda x: x * x
print(square(7))
```

- Can be used with higher-order functions like `map()`, `filter()`, and `reduce()`.

Example with filter():

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

3.9 PRACTICAL**3.9.1 USAGE OF FUNCTIONS IN PROGRAMS
(MINIMUM THREE)****Program 1 – Prime Number Checker**

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

num = 29
print(num, "is prime?", is_prime(num))
```

Program 2 – Fibonacci Sequence (Recursion)

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

for i in range(6):
    print(fibonacci(i), end=" ")
```

Program 3 – Using Lambda with Map & Filter

```
numbers = [1, 2, 3, 4, 5, 6]

squares = list(map(lambda x: x**2, numbers))
evens = list(filter(lambda x: x % 2 == 0, numbers))

print("Squares:", squares)
print("Evens:", evens)
```

3.10 SUMMARY TABLE :

Concept	Definition (Easy Words)	Example Program (Simple)	Output
Hiding Redundancy	Avoids repeating the same code again and again by grouping it into	python\ndef square(x):\n return x*x\nprint(square(5))\nprint(25100

Concept	Definition (Easy Words)	Example Program (Simple)	Output
	a function.	square(10))	
Hiding Complexity	Breaks a big/complex problem into smaller functions to make it easy to understand.	python\ndef add(a,b):\n return a+b\n\ndef sub(a,b):\n return a-b\n\nprint(add(10,5))\nprint(sub(10,5))	155
Parameters & Arguments	Parameters are variables in function definition. Arguments are actual values passed when calling the function.	python\ndef greet(name):\n print(\"Hello\", name)\n\ngreet(\"Daniel\")	Hello Daniel
Return Values	A function can return a result back to the main program.	python\ndef cube(x):\n return x**3\n\nprint(cube(3))	27
Formal vs Actual Arguments	Formal Arguments – Variables defined inside function. Actual Arguments – Values passed while calling.	python\ndef add(a,b): # a,b are formal\n return a+b\n\nprint(add(3,4)) # 3,4 are actual	7
Named Arguments	We can pass arguments by name instead of position.	python\ndef student(name, age):\n print(name, age)\n\nstudent(age=20, name=\"Jojo\")	Jojo 20

Concept	Definition (Easy Words)	Example Program (Simple)	Output
Recursive Function	A function calling itself repeatedly until a condition is met.	python\ndef fact(n):\n if n==1:\n return 1\n return n*fact(n-1)\n\nprint(fact(5))	120
Lambda (Anonymous) Function	A small one-line function without a name, often used for simple tasks.	python\square = lambda x: x*x\nprint(square(6))	36

3.11 IMPORTANT TWO MARKS

1. Need for functions

Functions help us avoid repetition and make programs simple by dividing them into smaller, reusable parts. They improve readability and reduce errors.

2. Redundancy

Redundancy means writing the same code again and again. Functions remove redundancy by storing repeated code in one place and reusing it.

3. Complexity

Large programs are difficult to understand. Functions reduce complexity by breaking them into smaller, manageable sections.

4. Parameter

A parameter is a variable used in the function definition. It acts as a placeholder to accept input values when the function is called.

5. Argument

An argument is the actual value or data passed to a function during the call. It replaces the parameter temporarily while the function executes.

6. Return value

Functions can send back a result to the main program using the return statement. If no return is given, Python automatically returns None.

7. Formal arguments

Formal arguments are the variables defined in the function header. They act as placeholders for the values passed during a function call.

8. Actual arguments

Actual arguments are the real values or variables passed to a function during its call. They are substituted into the formal arguments.

9. Named arguments

Named arguments are passed using the parameter name during the function call. This improves clarity and order, for example: `greet(name="John")`.

10. Recursion

Recursion means a function calling itself again and again until

a base condition is reached. It is useful in problems like factorial and Fibonacci.

11. **Example of recursion**

The factorial of a number can be calculated by recursion where $n! = n * (n-1)!$. The base condition stops the function when $n = 0$.

12. **Lambda function**

A lambda is an anonymous (nameless) function written in one line using the lambda keyword. It is often used for small calculations.

13. **Example of lambda**

For example, `add = lambda x, y: x+y` creates a function that returns the sum of two numbers without using `def`.

14. **Code reusability**

Functions provide reusability, meaning once written, the same function can be used many times in a program or in other programs.

15. **Multiple return values**

In Python, functions can return more than one value by separating them with commas. The values are returned as a tuple.

16. **Default return**

If a function does not use the return statement, it automatically returns the special value `None` in Python.

17. Built-in vs User-defined functions

Built-in functions are provided by Python (like `len()`, `max()`), while user-defined functions are created by the programmer.

18. Default arguments

Parameters can have default values which are used when no argument is passed. For example, `def greet(name="Guest")`.

19. Syntax of function

The general syntax is:

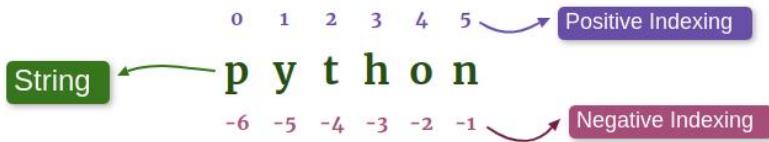
```
def function_name(parameters):  
    statements
```

UNIT IV

STRINGS & COLLECTIONS

String Comparison, Formatting, Slicing, Splitting, Stripping, List, tuples, and dictionaries, basic list operators, searching and sorting lists; dictionary literals, adding and removing keys, accessing and replacing values. Practical: String manipulations and operations on lists, tuples, sets, and dictionaries. (Minimum three)

4.1 STRINGS:



String is basically sequence of characters

Eg:

```
>>>str = "hello python"
```

```
>>> str 1 = str.upper ()
```

```
print (str 1)
```

```
print (str)
```

```
len (str)
```

O/P:

HELLO PYTHON

Hello python

12

4.1.1 STRING SLICING:

A substring of a string is obtained by taking a slice. Similarly we can slice a list to refer to sublist of the items in the list

String Slicing

str1 ⇒

F	A	C	E
---	---	---	---

0 1 2 3 ⇒ Positive indexing

-4 -3 -2 -1 ⇒ Negative indexing

str1[1:3] = AC

str1[-3:-1] = AC

1

Eg:

```
>>>str = "Hello python"
```

```
>>>print (str [0.6])  
>>>print (str [0:])  
>>>print (str [9:12])  
>>>print (str[1:11])
```

O/P

Hello

Hello python

hon

ello python

4.1.2 IMMUTABILITY:

We cannot change the existing string

```
>>>str = "Hello python"  
>>> print (str[0])  
>>>'h'
```

Type error

To make the desire changes we need to take new string and manipulate it as per our requirements.

```
>>>> str = "Hello python"
```

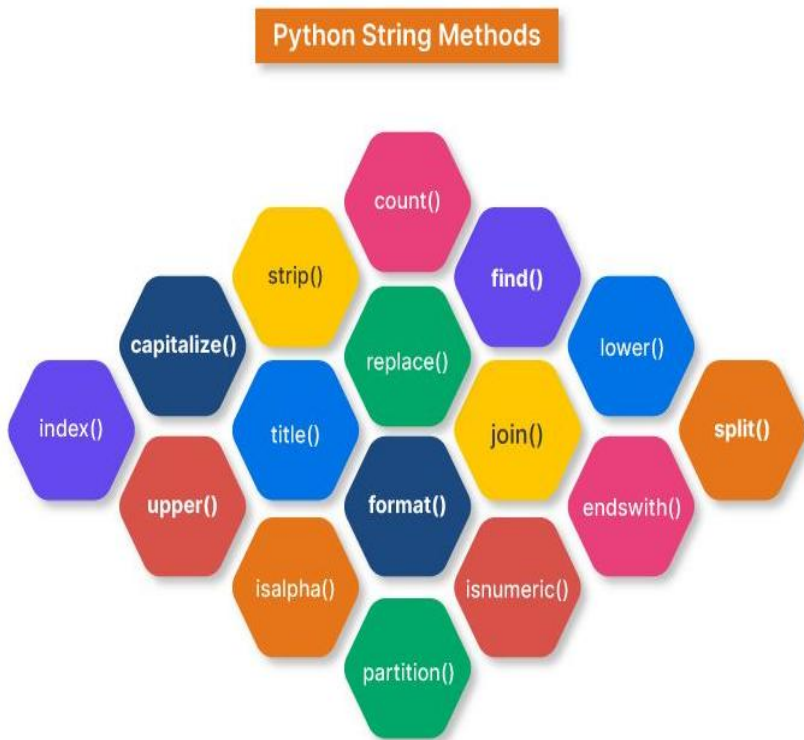
```
>>> new_str = 'P' + str [6:]
```

```
>>> print (new_str)
```

O/P:

Hello Python

4.1.3 STRING FUNCTIONS AND METHODS:



1.String concatenation:

Joining of two or more string is called string Concatenation.

```
>>>str1= "Hello"
```

```
>>>str2= "Python".
```

```
>>> Print (str1+ str2)
```

2.string comparison:

```
>,<=,==
```

```
>>> str = "Hello"
```

```
>>> str1 "Python"
```

```
>>> print (str > Str1)
```

False

3.String Repeatation:

we can repeat the string using * Operator

```
>>>str = "Hello"
```

```
print ( str*4)
```

Hello Hello Hello Hello.

4.Membership test:

The membership of particular character is determined using Keyword

```
>>>str1 = "Hello"
```

```
>>> 'H' in Str1
```

True

```
>>>'E' in str1
```

False.

PYTHON STRING METHODS:

1. Count()

Count the list values

2. Capitalize()

Capitalize the first letter

3. find ()

Finds the needed value

4. Index

5. isainum ()

return true if all are alphabets or numbers

6. is digit()

Return true if all are numbers

7. is lower ()
8. is upper()

STRING MODULE:

The String module contains number of constants and functions to process the strings. We use the string module python program we need to import it at the beginning,

1. The cap words function to display first letter capital
2. The upper function and lower case
3. Translation of character to other form

1. The capwords function to display first letter capital.

Syntax:

String.capwords (str)

```
>>>str="hello Python Program"
```

```
>>> String. Capwords (str)
```

O/P:

Hello Python Program

2.The upper function and lower case

Syntax:

String. upper(str)

String. lower (str)

3. Translation of character to other form:

The maketrans() returns the translation table for passing to translate(), that will map each character in from-ch into the character at the same position in to-ch. The from-ch and to-ch must have the same lengths

Syntax:

string. maketrans (from-ch,to_ch)

Eg:

String module 2.py

```
from-ch= "aco"
```

```
to-ch= "012"
```

```
new_str = str. maketrans (from-ch, to-ch)
```

```
str = "I love programming in python"
```

```
print (str)
```

```
print (str. translate ( new_str))
```

O/P:

I love programming in python

I l2v1 pr2gramming in python

In above program we have generated a translation table for the characters a, e and o characters. These characters will be mapped to 0, 1 and 2 respectively using the function maketrans.

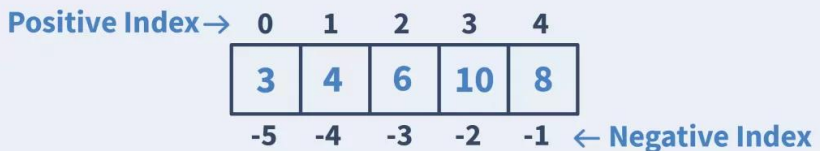
The actual conversation of the given string will take place using the function translate.

4.2 PYTHON COLLECTIONS OVERVIEW

4.2.1 LISTS

- Arrays and list are both used in python to store data
- Number is used to create an array in python
- It's a module which contains array they are used to store any type of data and can be indexed
- The functions which can be performed in list and array are distinct

My_List = [3, 4, 6, 10, 8]



Call by Value (Local variable)

```
def modify (x):  
    x=x+10  
    Print ("Inside function:",x)  
  
num = 5  
    modify (num)  
print("Outside function:", num)
```

O/P:

Inside function: 15

Outside function: 5

- Cannot be Modified in place
- so, when passed to a function a new copy is created
- Tuples are Immutable

Global variable

- Mutable object can be modifies in place
- So changes inside a fn reflect outside
- Lists are mutable

Call by Reference

Def modify (lst):

Print (“Inside function: lst”)

Numbers = [1,2,3]

Modify (numbers)

Print (“outside function”, number)

O/P:

Inside function: [1,2,3,10]

Outside function: [1,2,3,10]

4.2.2 TUPLE

- A Tuple is a sequence of values. The values can be of any type and they are indexed as Integers.
- Tuples are immutable.
- Tuples are comma separated list of values.

t = (1 , 2 , 'python' , tuple() , (42 , 'hi'))

The diagram illustrates the indexing of the tuple `t`. It shows the tuple `t = (1 , 2 , 'python' , tuple() , (42 , 'hi'))` with lines connecting each element to its corresponding index label below it. The first four elements are connected to `t[0]`, `t[1]`, `t[2]`, and `t[3]` respectively. The last element, a nested tuple `(42, 'hi')`, is enclosed in a curly brace and connected to `t[4]`.

Eg:

```
>>> Tuple=(A', 'a', 'E', 'e', 'I', 'i', 'O', 'o', 'U', 'u')
```

(or)

```
Tuple = A', 'a', 'E', 'e', 'I', 'i', 'O', 'o', 'U', 'u'
```

It is not necessary to enclose Tuples within paranthesis. To create a tuple with a single element we have to include a common in final

Eg:

```
>>> t1 = 'a'
```

```
>>> type (t1)
```

```
>>> class. Tuple
```

Another way to create a tuple is by built in function. If we create a Tuple with no arguments that is referred as empty tuple.

Eg:

```
>>>t2 = Tuple ('fruit')
```

```
>>>print (t2)
```

```
('f', 'r', 'u', 'i', 't')
```

```
>>>print (t2 [2])
```

O/P

Tuples are immutable which means we cannot update or change the values of tuple but we can replace one tuple with another.

Eg:

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

```
>>>t = ('a',) + t[1]
```

```
>>>print (t)
```

O/P

```
(a,2,3,4,5)
```

4.2.2.1 TUPLE ASSIGNMENT

1. An assignment to all of the elements in a tuple using a single assignment statement
2. Python has a very powerful tuple assignment feature that allows a tuple of variable on the left of an assignment to be assigned values from a tuple on the right of the assignment
3. The left side is a tuple of variables the right side is a tuple of values each value is assigned to its respective variable

4. All the expressions on the right side are evaluated before any of the assignments this feature makes tuple assignment quite versatile.
5. Naturally, the no. of variables on the left and the no. of values of the right have to be the same.

Tuple Assignment

```
>>> (a,b,c,d) = (1,2,3)
```

Value Error: need more than 3 values to Unpack,

4.2.2.2 TUPLE PACKING / UNPACKING

In tuple packing, the values on the left are 'packed' together in a tuple

```
>>> b=("George", 25, "20000") # tuple packing
```

In tuple unpacking, the values in a tuple on the right are 'unpacked' into the variable

The right side can be any kind of sequence

```
>>> b= "George", 25, "20000") # tuple packing.
```

```
>>> (name, age, salary)=b #tuple unpacking.
```

```
>>> name
```

```
'George'
```

```
>>>age
```

25

```
>>>salary
'20000'
```

4.2.3 SETS

- **Definition:** Unordered collection of **unique** items, written inside {}.
- **Key features:**
 - No duplicate elements
 - Supports mathematical operations like union, intersection, difference

Example:


```
nums = {1, 2, 2, 3, 4}
print(nums) # {1, 2, 3, 4}
```

4.2.4 DICTIONARIES:

A Dictionary organises information position in python by association. Not position in python. In python a dictionary associates a set of keys with data values.

```
d = {'a':1, 'b':2}

d.keys()          d.values()
  ↓              ↓
dict_keys(['a', 'b']) dict_values([1, 2])
```

 **Python Dictionary keys() and values()**

A Python dictionary is written as a sequence of keys or value pairs separated by commas. These pairs are called entities. The entire sequence of entity is enclosed in curly braces.

NOTE:

Element in dictionaries are not accessed via keys and not by their position

Dict {'Name': 'Joe', 'Age': 7, 'class': 'first'}

4.2.4.1 DICTIONARY OPERATION

Operation	Description	Example	Output
Create Dictionary	Define using {} with key–value pairs	student = {"name": "Daniel", "age": 23}	{'name': 'Daniel', 'age': 23}
Access Value	Use key inside []	student["name"]	Daniel
Get Value (safe)	Use .get() method	student.get("age")	23
Add Item	Assign new key–value	student["course"] = "Python"	{'name': 'Daniel', 'age': 23, 'course': 'Python'}
Update Value	Reassign value to key	student["age"] = 24	{'name': 'Daniel', 'age': 24}
Remove Item	pop(key)	student.pop("age")	{'name': 'Daniel'}

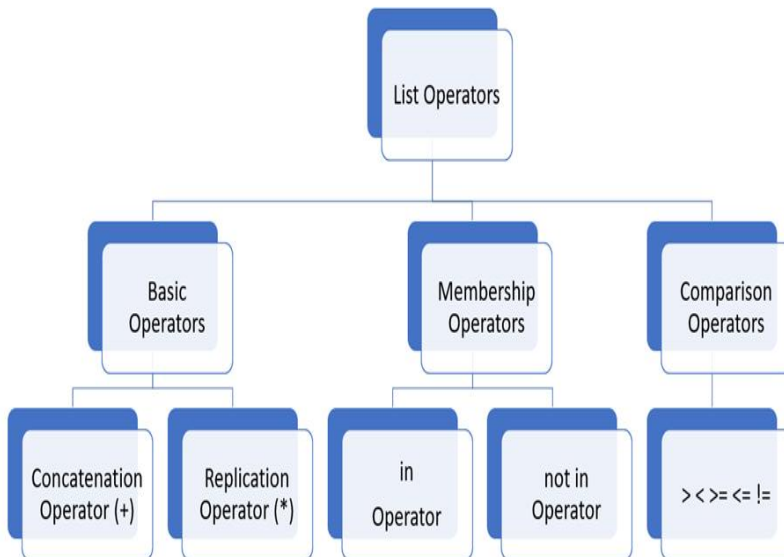
Operation	Description	Example	Output
	removes by key		
Remove Last Item	popitem() removes last inserted	student.popitem()	Remaining dict without last pair
Delete Key	Use del keyword	del student["name"]	{}
Clear Dictionary	Remove all items	student.clear()	{}
Keys	Get all keys	student.keys()	dict_keys(['name', 'age'])
Values	Get all values	student.values()	dict_values(['Daniel', 23])
Items	Get key–value pairs	student.items()	dict_items([('name', 'Daniel'), ('age', 23)])
Loop Keys	Iterate dictionary	for k in student: print(k)	name age
Loop Values	Iterate values	for v in student.values(): print(v)	Daniel 23
Loop Items	Iterate key–value pairs	for k,v in student.items(): print(k,v)	name Daniel age 23

4.2.4.2 DICTIONARY METHODS

Method	Description	Example	Output
<code>clear()</code>	Removes all elements from the dictionary	<code>student.clear()</code>	<code>{}</code>
<code>copy()</code>	Returns a shallow copy of the dictionary	<code>x = student.copy()</code>	<code>{'name':'Daniel','age':23}</code>
<code>fromkeys(seq, value)</code>	Creates a new dictionary with keys from a sequence and a common value	<code>dict.fromkeys(['a','b'],'zero')</code>	<code>{'a':'zero','b':'zero'}</code>
<code>get(key, default)</code>	Returns value for key; returns default if key not found	<code>student.get('age', 0)</code>	23
<code>items()</code>	Returns view object with all key–value pairs	<code>student.items()</code>	<code>dict_items([('name','Daniel'), ('age',23)])</code>
<code>keys()</code>	Returns view object with all keys	<code>student.keys()</code>	<code>dict_keys(['name','age'])</code>
<code>values()</code>	Returns view object with all values	<code>student.values()</code>	<code>dict_values(['Daniel', 23])</code>
<code>pop(key, default)</code>	Removes item with given key and returns its value	<code>student.pop('age')</code>	23
<code>popitem()</code>	Removes and returns the last inserted key–value	<code>student.popitem()</code>	<code>('age', 23)</code>

Method	Description	Example	Output
	pair		
setdefault(key, default)	Returns value of key; if key not present, inserts with default value	student.setdefault('course', 'Python')	Adds 'course': 'Python' if not present
update(dict2)	Updates dictionary with key–value pairs from another dictionary	student.update({'age': 24})	{'name': 'Daniel', 'age': 24}

4.3 BASIC LIST OPERATIONS



4.3.1 BASIC LIST OPERATORS IN PYTHON

Operator	Description	Example	Output
+	Concatenates two lists	[1,2,3] + [4,5]	[1,2,3,4,5]
*	Repeats list elements	[1,2] * 3	[1,2,1,2,1,2]
in	Checks if element exists in list	3 in [1,2,3]	True
not in	Checks if element does not exist	5 not in [1,2,3]	True
len()	Returns number of items in list	len([10,20,30])	3
min()	Returns smallest element	min([4,7,1])	1
max()	Returns largest element	max([4,7,1])	7
sum()	Returns sum of elements	sum([1,2,3,4])	10

4.3.2 LIST METHODS IN PYTHON

Method	Description	Example	Output
<code>append(x)</code>	Adds element to the end	<code>lst.append(10)</code>	<code>[1,2,3,10]</code>
<code>insert(i, x)</code>	Inserts element at index i	<code>lst.insert(1,99)</code>	<code>[1,99,2,3]</code>
<code>extend(list2)</code>	Adds all elements of another list	<code>lst.extend([4,5])</code>	<code>[1,2,3,4,5]</code>
<code>remove(x)</code>	Removes first occurrence of element	<code>lst.remove(2)</code>	<code>[1,3,4]</code>
<code>pop(i)</code>	Removes and returns element at index i (default last)	<code>lst.pop()</code>	Returns last element
<code>clear()</code>	Removes all elements	<code>lst.clear()</code>	<code>[]</code>
<code>index(x)</code>	Returns index of first occurrence of x	<code>lst.index(3)</code>	2
<code>count(x)</code>	Returns number of times x appears	<code>lst.count(2)</code>	1
<code>sort()</code>	Sorts list in ascending order (by default)	<code>lst.sort()</code>	<code>[1,2,3,4]</code>

Method	Description	Example	Output
<code>reverse()</code>	Reverses the order of elements	<code>lst.reverse()</code>	<code>[4,3,2,1]</code>
<code>copy()</code>	Returns a shallow copy of list	<code>lst.copy()</code>	<code>[1,2,3]</code>

4.4 SEARCHING AND SORTING LISTS

Python is an interpreted, object-oriented, high level, programming language with dynamic semantics. Developed by Guido Van Rossum in 1991. It supports multiple programming paradigms, including structured, object-oriented and functional programming.

List are mutable means that you can change the values of list, like we can add an element, remove, repeat and allows indexing and slicing like strings an element from a list by using different operators.

List can be created by using square brackets. By placing elements inside square bracket[],separate by commas. For example: `list1= ["a","b"]`. There is a one more type of list known as "Nested list". It is list within list.

For example: `list1=["a","b"[1,2]"c"]`

Lists are one of the most important **collection data structures** in Python, and often we need to **search** for elements or **sort** the list for better organization.

4.4.1 SEARCHING IN LISTS

Searching means **finding whether an element exists in a list or locating its position**. Python provides multiple ways to search for elements, from simple membership operators to efficient searching algorithms.

1. Using Membership Operators (in, not in)

These are the simplest ways to check if an element exists in a list.

Example:

```
numbers = [10, 20, 30, 40, 50]
print(20 in numbers)    # True
print(100 not in numbers) # True
```

2. Using index() Method

The index() method returns the **first index position** of the given element.

If the element is not found, it raises a ValueError.

Example:

```
fruits = ["apple", "banana", "cherry", "banana"]
print(fruits.index("banana")) # 1
```

3. Linear Search (Manual Search)

Linear search checks each element in the list one by one until the target element is found. This method works on **unsorted lists**.

Example:

```
def linear_search(lst, target):  
    for i in range(len(lst)):  
        if lst[i] == target:  
            return i # return index  
    return -1 # not found
```

```
numbers = [15, 8, 42, 23, 4]print(linear_search(numbers, 23)) #  
3print(linear_search(numbers, 100)) # -1
```

4. Binary Search (Efficient for Sorted Lists)

Binary search is faster but works only on **sorted lists**. It repeatedly divides the list into halves to find the target element.

Example using bisect module:

```
import bisect  
  
numbers = [5, 10, 15, 20, 25, 30]  
  
pos = bisect.bisect_left(numbers, 20)
```

```
if pos < len(numbers) and numbers[pos] == 20:
```

```
    print("Found at index:", pos) # 3else:
```

```
    print("Not Found")
```

5. Searching with List Comprehension

List comprehensions can be used to find all positions of an element.

Example:

```
fruits = ["apple", "banana", "cherry", "banana", "grape"]
```

```
positions = [i for i, x in enumerate(fruits)
```

```
if x == "banana"]
```

```
print(positions) # [1, 3]
```

Summary

- **Simple Search** → in, not in, index()
- **Linear Search** → Works for any list but slower for large data
- **Binary Search** → Works only on sorted lists, faster (log n time)
- **List Comprehension** → Finds multiple positions easily

Quick Exercise

- 1) Write a program to check if a number is present in a list.
- 2) Find the index of "cherry" in the list ["apple","cherry","grape"].
- 3) Implement a linear search function and test it with a list of marks.
- 4) Use binary search to find the position of 45 in [10,20,30,40,45,50].
- 5) Print all index positions of "apple" in a list using list comprehension.

4.4.2 SORTING IN LISTS

Sorting means arranging the elements of a list in a particular order, either **ascending** or **descending**. Python provides multiple ways to sort lists, from built-in methods to custom functions.

1. Using sort() Method

- Sorts the list **in place** (changes the original list).
- Default is **ascending order**.
- Accepts two optional arguments:

reverse=True → sort in descending order.

key=function → sort based on custom logic.

Example:

```
numbers = [5, 2, 9, 1, 7]
```

```
numbers.sort()print(numbers) # [1, 2, 5, 7, 9]
```

```
numbers.sort(reverse=True)print(numbers) # [9, 7, 5, 2, 1]
```

2. Using sorted() Function

- Returns a **new sorted list** (does not change the original).
- Same parameters as sort().

Example:

```
marks = [45, 89, 23, 67]
```

```
sorted_marks = sorted(marks)
```

```
print(sorted_marks) # [23, 45, 67, 89]
```

```
print(marks)      # [45, 89, 23, 67]
```

(unchanged)

3. Sorting with key Parameter

The key parameter allows sorting based on a specific rule or property.

Sort by string length:

```
fruits = ["apple", "banana", "kiwi", "cherry"]
```

```
fruits.sort(key=len)print(fruits)
```

```
# ['kiwi', 'apple', 'cherry', 'banana']
```

Sort case-insensitive:

```
names = ["Zara", "daniel", "Lulu", "joselin"]  
names.sort(key=str.lower)  
print(names) # ['daniel', 'joselin', 'Lulu', 'Zara']
```

4. Sorting with Custom Function

You can define your own function and use it in sorting.

Example: Sort numbers by remainder when divided by 5

```
def remainder5(x):  
    return x % 5  
  
nums = [15, 22, 8, 19, 31]  
nums.sort(key=remainder5)  
print(nums) # [15, 31, 22, 8, 19]
```

5. Reversing a List (Not Sorting)

Sometimes we just want to **reverse order** without sorting.

Example:

```
letters = ['a','b','c','d']  
letters.reverse()print(letters) # ['d','c','b','a']
```

4.4.2.1 Difference between `sort()` and `sorted()`

Feature	<code>sort()</code>	<code>sorted()</code>
Changes Original List	Yes	No (returns new list)
Return Value	None	New sorted list
Speed	Slightly faster (in-place)	Slightly slower (creates copy)

Summary

- `sort()` → modifies the original list.
- `sorted()` → returns a new sorted list.
- Use `reverse=True` for descending order.
- Use `key` for custom sorting.
- `reverse()` only reverses, does not sort.

Quick Exercise

- 1) Sort a list of numbers [45, 12, 78, 34] in ascending and descending order.
- 2) Sort the list ["dog","cat","elephant","bee"] by string length.
- 3) Use `sorted()` to sort a list without changing the original.
- 4) Write a custom sort that arranges numbers based on their last digit.
- 5) Reverse a list ["one","two","three"] without sorting.

4.4.3 PRACTICAL APPLICATIONS

Searching:

- Finding a student record by roll number.
- Checking availability of an item in a shopping cart.

Sorting:

- Arranging exam scores from highest to lowest.
- Sorting product names alphabetically.
- Sorting transactions by date or amount.

Summary:

- Searching → Linear search (in, index()), Binary search (bisect).
- Sorting → sort(), sorted(), custom key-based sorting.
- Python uses **Timsort (Hybrid of Merge + Insertion Sort)** internally for efficiency.

4.5 DICTIONARY LITERALS, KEYS, AND VALUES

A **dictionary** in Python is an **unordered, mutable collection of key-value pairs**. It allows fast access to data using a **unique key** instead of an index (as in lists). Dictionaries are widely used for **mapping relationships** (like student roll numbers to names, product IDs to prices, etc.).

4.5.1 DICTIONARY LITERALS

- A **dictionary literal** is created using curly braces {} with key-value pairs.
- The **syntax** is:
- `dictionary = {key1: value1, key2: value2, key3: value3}`
- Keys must be **unique** and **immutable** (string, number, tuple).
- Values can be **any data type** (string, list, another dictionary, etc.).

Syntax:

```
my_dict = {key1: value1, key2: value2, key3: value3}
```

Example:

```
student = {  
    "name": "Daniel",  
    "age": 23,  
    "department": "CSE"  
}  
print(student)
```

Output:

```
{'name': 'Daniel', 'age': 23, 'department': 'CSE'}
```

Table for Dictionary Literal Example:

Key	Value
name	Daniel
age	23
department	CSE

4.5.2 DICTIONARY KEYS

- Keys act as **identifiers** for values.
- Must be **unique** → if a duplicate key is added, the latest value **overwrites** the old one.
- Keys must be **hashable** (strings, numbers, tuples).

Example:

```
student = {"id": 101, "name": "Arun", "id": 102}  
print(student)
```

Output: {'id': 102, 'name': 'Arun'} → The first id is overwritten.

Accessing Keys:

```
student = {"name": "Arun", "age": 20, "course": "CSE"}  
print(student.keys()) # dict_keys(['name', 'age', 'course'])
```

4.5.3 DICTIONARY VALUES

- Values can be **any data type**: string, int, list, dictionary, etc.
- Values can be **repeated** (not unique).

Example:

```
student = {"name": "Arun", "age": 20, "marks": [80, 85, 90]}  
print(student.values()) # dict_values(['Arun', 20, [80, 85, 90]])
```

Accessing a value using a key:

```
print(student["name"]) # Arun  
print(student.get("marks")) # [80, 85, 90]
```

4.5.4 DICTIONARY ITEMS (KEYS + VALUES)

`items()` returns both keys and values as pairs (tuples).

Example:

```
student = {"name": "Arun", "age": 20, "course": "CSE"}  
print(student.items())
```

Output: dict_items([('name', 'Arun'), ('age', 20), ('course', 'CSE')])

4.5.5 PRACTICAL EXAMPLE

```
# Storing product prices  
products = {"pen": 10, "notebook": 40, "bag": 500}
```

```
# Accessing keys
for item in products.keys():
    print(item)

# Accessing values
for price in products.values():
    print(price)

# Accessing items
for item, price in products.items():
    print(item, ":", price)
```

Output:

```
pen
notebook
bag
10
40
500
pen : 10
notebook : 40
bag : 500
```

4.5.6 APPLICATIONS OF DICTIONARY KEYS & VALUES

- **Student Database:** Roll number → Name.
- **Inventory Management:** Product ID → Quantity.
- **Phonebook:** Name → Phone number.
- **Language Translation:** Word → Equivalent meaning.
- **Configuration Settings:** Option → Value.

Summary:

- **Dictionary Literals** define key-value pairs inside {}.
- **Keys** must be unique and immutable.
- **Values** can be any type and repeated.
- Useful methods: `.keys()`, `.values()`, `.items()`.

4.6 ACCESSING VALUES

Values are accessed using their **keys**.

Syntax:

```
value = dictionary[key]
```

Example:

```
print("Name:", student["name"])print("Age:", student["age"])
```

Output:

Name: DanielAge: 23

You can also use `dictionary.get(key)` which returns `None` if the key doesn't exist (instead of an error).

4.6.1 ADDING KEYS

You can add a new key-value pair by assigning a value to a new key.

Syntax:

```
dictionary[new_key] = new_value
```

Example:

```
student["year"] = 3print(student)
```

Output:

```
{'name': 'Daniel', 'age': 23, 'department': 'CSE', 'year': 3}
```

Table after Adding a Key:

Key	Value
name	Daniel
age	23
department	CSE
year	3

4.7. REPLACING VALUES

You can replace a value by assigning a new value to an **existing** key.

Syntax:

```
dictionary[key] = new_value
```

Example:

```
student["age"] = 24print(student)
```

Output:

```
{'name': 'Daniel', 'age': 24, 'department': 'CSE', 'year': 3}
```

Table after Replacing Value:

Key	Value
name	Daniel
age	24
department	CSE
year	3

4.7.1 REMOVING KEYS

Keys can be removed using

- `del dictionary[key]`
- `dictionary.pop(key)`

Example:

```
# Using del
del student["department"]
print(student)

# Using pop
removed_value = student.pop("year")
print("Removed:", removed_value)
print(student)
```

Output:

```
{'name': 'Daniel', 'age': 24}
```

```
Removed: 3
```

```
{'name': 'Daniel', 'age': 24}
```

Table after Removing Keys:

Key	Value
name	Daniel
age	24

4.8 DICTIONARY OPERATIONS IN PYTHON

Operation	Syntax / Method	Example Code	Output
Dictionary Literal (Creation)	<code>dict_name = {key: value, ...}</code>	<code>student = {"name": "Daniel", "age": 23}</code>	<code>{'name': 'Daniel', 'age': 23}</code>
Access Value	<code>dict_name[key]</code> <code>dict_name.get(key)</code>	<code>student["name"]</code> <code>student.get("age")</code>	Daniel 23
Add New Key	<code>dict_name[new_key] = value</code>	<code>student["year"] = 3</code>	<code>{'name': 'Daniel', 'age': 23, 'year': 3}</code>
Replace Value	<code>dict_name[existing_key] = new_value</code>	<code>student["age"] = 24</code>	<code>{'name': 'Daniel', 'age': 24, 'year': 3}</code>
Remove Key (del)	<code>del dict_name[key]</code>	<code>del student["year"]</code>	<code>{'name': 'Daniel', 'age': 24}</code>
Remove Key (pop)	<code>dict_name.pop(key)</code>	<code>student.pop("age")</code>	Returns 24 Remaining: <code>{'name': 'Daniel'}</code>
Remove Last Item	<code>dict_name.popitem()</code>	<code>student.popitem()</code>	Removes last inserted item
Check Key Exists	<code>key in dict_name</code>	<code>"name" in student</code>	True
Get All Keys	<code>dict_name.keys()</code>	<code>student.keys()</code>	<code>dict_keys(['name',</code>

Operation	Syntax / Method	Example Code	Output
			'age'])
Get All Values	<code>dict_name.values()</code>	<code>student.values()</code>	<code>dict_values(['Daniel', 23])</code>
Get All Items	<code>dict_name.items()</code>	<code>student.items()</code>	<code>dict_items([('name', 'Daniel'), ('age', 23)])</code>
Clear Dictionary	<code>dict_name.clear()</code>	<code>student.clear()</code>	<code>{}</code> (empty dictionary)

4.9 ADDING, REMOVING, AND REPLACING ELEMENTS

Lists:

```
fruits = ["apple", "banana"]
fruits.append("cherry") # add
fruits.remove("banana") # remove
fruits[0] = "grape"     # replace
```

Dictionaries:

```
student = {"name": "Alice", "age": 20}
student["dept"] = "CSE" # add
del student["age"]      # remove
student["name"] = "Bob" # replace
```

Sets:

```
nums = {1, 2, 3}
nums.add(4)
nums.discard(2)
```

4.10 PRACTICAL EXERCISES**4.10.1 STRING MANIPULATIONS****Exercise 1: Palindrome Check**

Write a Python program to check whether a given string is a palindrome or not.

Example:

Input: madam → Output: Palindrome

```
word = input("Enter a string: ")
if word == word[::-1]:
    print("Palindrome")
else:
    print("Not a Palindrome")
```

Exercise 2: Word Frequency Counter

Write a program to count how many times each word occurs in a given sentence.

```
sentence = "python is easy and python is powerful"
words = sentence.split()
freq = {}
for word in words:
```

```
freq[word] = freq.get(word, 0) + 1  
print(freq)
```

Exercise 3: String Formatting

Write a Python program to input name, age, and grade of a student and display them using string formatting.

```
name = "Arun"  
age = 20  
grade = "A"  
print(f'Student {name} is {age} years old and scored grade {grade}.')
```

4.10.2 OPERATIONS ON LISTS

Exercise 1: Find Maximum and Minimum

Write a Python program to find the largest and smallest number in a list.

```
numbers = [10, 25, 7, 56, 89, 3]  
print("Max:", max(numbers))  
print("Min:", min(numbers))
```

Exercise 2: Sorting a List

Write a program to sort a list of numbers in ascending and descending order.

```
numbers = [5, 3, 8, 6, 2]  
print("Ascending:", sorted(numbers))
```

```
print("Descending:", sorted(numbers, reverse=True))
```

Exercise 3: List Comprehension

Write a program to generate a list of squares of numbers from 1 to 10.

```
squares = [x**2 for x in range(1, 11)]  
print(squares)
```

4.10.3 OPERATIONS ON TUPLES

Exercise 1: Tuple Packing and Unpacking

```
person = ("Arun", 20, "CSE")  
name, age, course = person  
print(name, age, course)
```

Exercise 2: Find Element in Tuple

```
t = (1, 3, 5, 7, 9)  
print("Is 5 in tuple?", 5 in t)
```

Exercise 3: Tuple Concatenation

```
t1 = (1, 2, 3)  
t2 = (4, 5, 6)  
print("Concatenated Tuple:", t1 + t2)
```

4.10.4 OPERATIONS ON SETS

Exercise 1: Set Union and Intersection

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
print("Union:", a | b)
print("Intersection:", a & b)
```

Exercise 2: Removing Duplicates

```
numbers = [1, 2, 2, 3, 4, 4, 5]
unique = set(numbers)
print("Unique Numbers:", unique)
```

Exercise 3: Set Difference

```
a = {1, 2, 3, 4}
b = {2, 4, 6}
print("Difference:", a - b)
```

4.10.5 OPERATIONS ON DICTIONARIES

Exercise 1: Student Marks Dictionary

Write a program to store student names as keys and marks as values, then print them.

```
marks = {"Arun": 85, "Kavi": 92, "Meena": 78}
for name, score in marks.items():
    print(name, ":", score)
```

Exercise 2: Updating Dictionary

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

Exercise 3: Frequency Count Using Dictionary

```
text = "apple banana apple orange banana apple"  
words = text.split()  
freq = {}  
for word in words:  
    freq[word] = freq.get(word, 0) + 1  
print(freq)
```

Summary of Practical Exercises:

- **Strings** → Manipulation, formatting, word frequency.
- **Lists** → Sorting, searching, comprehensions.
- **Tuples** → Packing/unpacking, searching, concatenation.
- **Sets** → Union, intersection, removing duplicates.
- **Dictionaries** → Storing, updating, counting frequencies.

4.11 IMPORTANT TWO MARKS

STRINGS

1. **What is string comparison in Python?**

String comparison is checking whether two strings are equal or which one is greater/lesser based on ASCII/Unicode values using operators like `==`, `!=`, `<`, `>`.

2. **Give an example of string comparison.**

```
print("apple" < "banana") # True
```

3. **What is string formatting?**

String formatting allows inserting variables into strings using `format()` or f-strings.

Example:

```
name = "Daniel"print(f'Hello {name}')
```

4. **What is string slicing?**

Extracting a part of a string using `[start:end:step]`.

Example: `"Python"[0:3] → 'Pyt'`

5. **Differentiate between `split()` and `strip()`.**

- `split()` breaks a string into a list based on a separator.
- `strip()` removes leading and trailing whitespaces.

Example for `split()` and `strip()`.

```
"hi,hello".split(",") → ['hi','hello']
```

```
" hello ".strip() → 'hello'
```

LISTS, TUPLES, DICTIONARIES

1. What is the difference between lists and tuples?

- List: Mutable (can be changed).
- Tuple: Immutable (cannot be changed).

Example of list and tuple

```
list1 = [1, 2, 3]
```

```
tuple1 = (1, 2, 3)
```

2. What are dictionary literals?

Dictionary literals are created using {key: value} notation.

Example: student = {"name": "Daniel", "age": 23}

3. How to add and remove keys in a dictionary?

I. Add: dict["key"] = value

II. Remove: dict.pop("key") or del dict["key"]

4. How to access dictionary values?

Using keys: dict[key] or dict.get(key)

5. How to replace a dictionary value?

Reassign value to the key.

Example: student["age"] = 2

BASIC LIST OPERATIONS

6. **What are basic list operators in Python?**

+ (concatenation), * (repetition), in (membership), len() (length).

7. **Give an example of list concatenation.**

$$[1,2] + [3,4] \rightarrow [1,2,3,4]$$

8. **How to search an element in a list?**

Using in operator or list.index().

Example: 3 in [1,2,3] \rightarrow True

9. **How to sort a list in Python?**

Using sort() (in-place) or sorted() (returns new list).

Example: nums = [3,1,2]; nums.sort() \rightarrow [1,2,3]

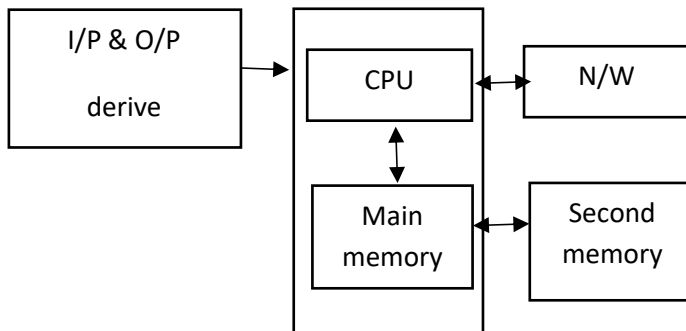
UNIT V

FILES OPERATIONS

Create, Open, Read, Write, Append and Close files. Manipulating directories, OS and Sys modules, reading/writing text and numbers, from/to a file; creating and reading a formatted file (csv, tab-separated, etc.); Practical: Opening, closing, reading and writing in formatted file format and sort data. (Minimum three)

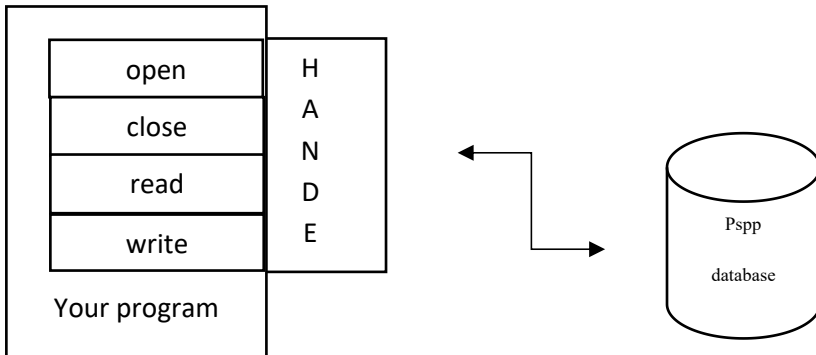
5.INTRODUCTION:

File is a named location on disk to store related information. Since, RAM (Random Access Memory) is volatile. Which loses its data when computer is turned off we use files for future use of the data.



5.1 FILE HANDLING COCEPTS

Opening and closing file:



File types:

- Text file
- Binary file

Text file:

- It is a type of file which is used to store textual information
- Structured as a sequence of lines
- Once you store a data in a text file it remains even if you shutdown and restart a computer

Binary file:

Is any type of file that, is not a text file.

5.2 CREATING AND OPENING FILES

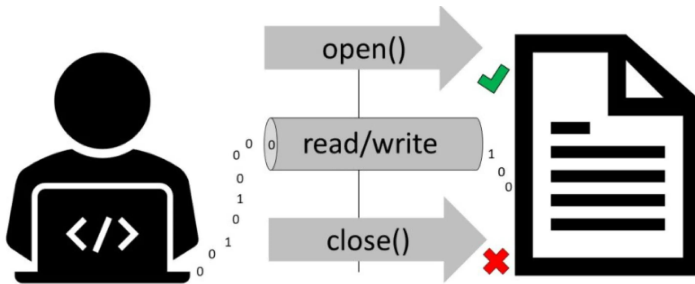


Fig 5.2 CREATING AND OPENING FILES

Opening a file:

In python there is an in built fn called `open()` to open a file

The open function:

Syntax:

```
file object = open(file _name  
[, access-mode] [, buffering])
```

`file_name` → that contains the name of the file that we want to access

`access mode` → It determines the mode in which the file has to be opened.

`buffering` → If the buffering Value is set too, no buffering takes place

If the value is 1, line buffering takes place.

Buffering Value > 1, the buffering action is performed with the indicated buffer size.

If -ve the buffer size & system default.

Eg:

```
f=open("Popes.txt", 'r', "encoding="utf-8")
```

Python file modes:

Mode	Description
'r'	Open a file for reading
'w'	Open a file for writing creates a new file if it does not exist.
'rb'	Open only a file for reading only in a binary format
'wb'	Opening a file for writing only in a binary format
'rt'	Opens a file for both reading and writing
'wt'	Opens a file for both writing and reading
'rbt'	Open a file for reading and writing in a binary file
'wbt'	Open a file for writing and reading in a binary file.
'a'	Open a file for appending at the end.

'ab'	Open a file for appending in binary format
'a+'	Open a file for both appending and reading
'ab+'	Open a file for both appending and reading in binary format
'+'	Open a file for update

Closing a file:

In python for closing a file

The close method: the close() is used.

Syntax

```
file object. close()
```

Eg:

```
f = open("text.txt",'r', encoding="utf-8")
```

```
f.close()
```

5.3 READING AND WRITING FILES

Reading a file

- To read a file in python we must open a file in reading mode.
- We can use the read() size method to read the size of data.

- If size parameter is not specified it reads and file return the whole

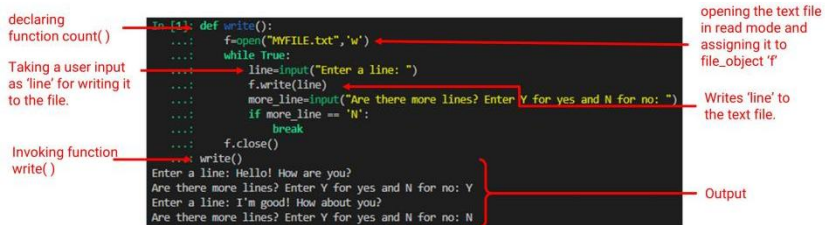


Fig 5.3 READING AND WRITING FILES

Eg:

`f= open ("Popes.txt", "r", "encoding="utf-8")`

`f. read(4) → read till the 4th word Eg(Dr.G.)`

`f.read() → reads the whole file`

`f.tell () → tells where we are reading (4)`

`f.seek → tells 10th word. (e)`

`f. read line () → reads line by line.`

Writing a file for writing the contents to the file we must 1st open it in the write mode. Here we use 'w', 'a' or 'r' as a file mode.

`F= open ("Popes.txt", 'w', encoding= 'utf-8')`

`f.write("Dr. G. U. Pope"\n)`

```
f.write ("College of “\n")  
f.write ("Engg "\n)  
f. write("sawyerpuram" \n)  
f. write ("AUNIT OF" \n)
```

5.4 APPEND AND CLOSE FILES IN PYTHON

Append Mode ('a' or 'a+')

Adds data **at the end** of an existing file without deleting previous content.

'a' → Append only

'a+' → Append and Read

Syntax:

```
file = open("filename.txt", "a")  
file.write("Text to append\n")  
file.close()
```

Using with (Recommended):

with open("filename.txt", "a") as file:

```
file.write("Text to append\n")# Automatically closes file
```

Close a File

- `close()` saves changes and frees system resources.
- Using `with` handles closing automatically.

Key Points:

- Always close files after operations.
- Append mode prevents overwriting existing data.

5.5 DIRECTORY MANIPULATION (OS AND SYS MODULES)

Python provides a lot of libraries to interact with the development environment. If you need help using the OS and Sys Module in Python, you have landed in the right place. This article covers a detailed explanation of the OS and Sys Module including their comparison. By the end of this article, you will be able to easily decide which module suits your Python application development.

What is the OS Module?

OS stands for the Operating System module in Python that allows the developers to interact with the operating system. It provides functions to perform tasks like file and directory manipulation, process management, and more. For example, we can get information about the operating system, such as the current working directory and the user's home directory. This makes it suitable for managing the system-level operations.

What is the Sys Module?

On the other hand, the System, abbreviated as the Sys Module helps us to interact with the Python runtime environment. It

allows access to variables used or maintained by the Python interpreter. It also provides tools such as 'sys.argv' for command-line arguments, system-specific parameters, or exiting scripts. Also, developers can easily manipulate parameters related to the interpreter's configuration, standard streams, and exit codes. Hence, we can easily manipulate the various parts of the Python Runtime environment.

The **os** and **sys** modules help manage files and directories.

Sys. Module Vs OS Module

Below are some of the examples by which we can understand the difference between [sys module](#) and [os module](#) in [Python](#):

5.5.1 Accessing Details

The Sys Module can access the command line arguments and the [os module](#) can access the location of the script as shown in the below code.

```
# PythonCode.py executed with 'python PythonCode.py arg1 arg2'
import sys
import os

# Accessing command-line arguments using sys.argv
script_name = sys.argv[0]
arguments = sys.argv[1:]

# Getting the absolute path of the script using os.path
script_path = os.path.abspath(script_name)

print("&quot;Script Name:&quot;, script_name)
print("&quot;Arguments:&quot;, arguments)
print("&quot;Script Path:&quot;, script_path)
```

Output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\fchbh\Desktop> python '.\import os.py' arg1 arg2
Script Name: .\import os.py
Arguments: ['arg1', 'arg2']
Script Path: C:\Users\fchbh\Desktop\import os.py
PS C:\Users\fchbh\Desktop> 
```

Output

5.5.2 Returning Path

The OS Module gives the Python Path within the system and the Sys module gives the Python Path that is shown below:

```
import os
import sys

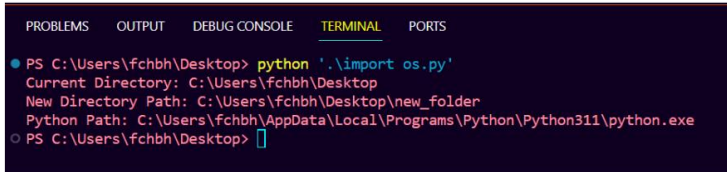
# Getting current working directory using os.getcwd()
current_directory = os.getcwd()

# Creating a new directory using os.mkdir()
new_directory1 = os.path.join(current_directory, 'new_folder')
os.mkdir(new_directory1)

# Checking Python path using sys.executable
python_path = sys.executable

print(&quot;Current Directory:&quot;, current_directory)
print(&quot;New Directory Path:&quot;, new_directory1)
print(&quot;Python Path:&quot;, python_path)
```

Output:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\fchbh\Desktop> python .\import os.py
Current Directory: C:\Users\fchbh\Desktop
New Directory Path: C:\Users\fchbh\Desktop\new_folder
Python Path: C:\Users\fchbh\AppData\Local\Programs\Python311\python.exe
PS C:\Users\fchbh\Desktop>
```

Output

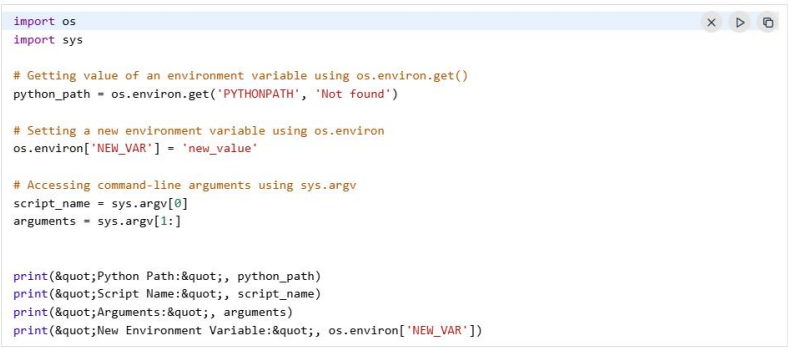
5.5.3 Environment Variables and Command Line Arguments

You can use OS Module to get the Environment Variables and Sys module to access the Command line arguments. This is illustrated as shown below.

```
import os, sys
```

```
print(os.getcwd())    # Current directory
```

```
os.mkdir("NewFolder") # Create directory
os.rename("old.txt", "new.txt")
os.remove("new.txt") # Delete file
os.rmdir("NewFolder") # Remove directory
print(sys.version) # Python version info
```

A screenshot of a Python script in an IDE. The script imports os and sys, gets the PYTHONPATH environment variable, sets a new environment variable NEW_VAR, accesses command-line arguments, and prints the Python path, script name, arguments, and the new environment variable.

```
import os
import sys

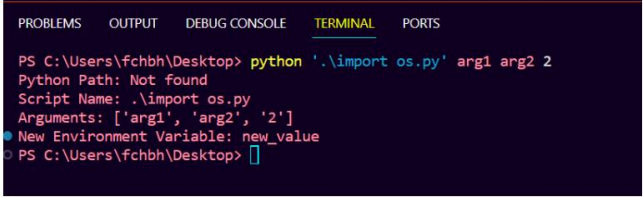
# Getting value of an environment variable using os.environ.get()
python_path = os.environ.get('PYTHONPATH', 'Not found')

# Setting a new environment variable using os.environ
os.environ['NEW_VAR'] = 'new_value'

# Accessing command-line arguments using sys.argv
script_name = sys.argv[0]
arguments = sys.argv[1:]

print(""Python Path:", python_path)
print(""Script Name:", script_name)
print(""Arguments:", arguments)
print(""New Environment Variable:", os.environ['NEW_VAR'])
```

Output:

A screenshot of a terminal window showing the output of the Python script. The terminal displays the command to run the script with arguments, followed by the output of the script, which includes the Python path, script name, arguments, and the new environment variable.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\fchbh\Desktop> python '.\import os.py' arg1 arg2 2
Python Path: Not found
Script Name: .\import os.py
Arguments: ['arg1', 'arg2', '2']
New Environment Variable: new_value
PS C:\Users\fchbh\Desktop>
```

Output

5.5.4 Reading/Writing Text and Numbers

Writing numbers:

```
f = open("numbers.txt", "w")
for i in range(1, 6):
    f.write(str(i) + "\n")
f.close()
```

Reading numbers:

```
f = open("numbers.txt", "r")
nums = f.readlines()
nums = [int(x.strip()) for x in nums]
print(nums)
f.close()
```

5.6 Difference between os and sys module in Python

Feature	os Module	sys Module
Purpose	Deals with operating system dependent functions like file handling, process management, directory operations.	Provides access to Python interpreter variables and functions that interact with the runtime environment.
Usage	Used for file paths, creating/removing directories, environment variables, process management.	Used for command-line arguments, exiting programs, controlling standard I/O, Python version info.
Example Functions	os.getcwd(), os.mkdir(), os.remove(), os.environ	sys.argv, sys.exit(), sys.version, sys.path
Example Code	python import os; print(os.getcwd()) → shows current directory	python import sys; print(sys.argv) → shows command-line args

5.7 READING/WRITING FORMATTED FILES (CSV, TAB-SEPARATED, ETC.)

5.7.1 Working with csv files in Python

A **CSV (Comma Separated Values) file** is a plain text file where each line represents a data record, and fields within each record are separated by commas. It's commonly used for spreadsheets and databases due to its simplicity and readability.

Below are some operations that we perform while working with Python CSV files in Python

5.7.1.1 Reading a CSV file

Reading from a CSV file is done using the reader object. The CSV file is opened as a text file with Python's built-in `open()` function, which returns a file object. In this example, we first open the CSV file in **READ** mode, file object is converted to `csv.reader` object and further operation takes place. Code and detailed explanation is given below.

```
import csv
filename = "AAPL.csv" # File name
fields = [] # Column names
rows = [] # Data rows

with open(filename, 'r') as csvfile:
    csvreader = csv.reader(csvfile) # Reader object

    fields = next(csvreader) # Read header
    for row in csvreader: # Read rows
        rows.append(row)

    print("Total no. of rows: %d" % csvreader.line_num) # Row count

print('Field names are: ' + ', '.join(fields))

print('\nFirst 5 rows are:\n')
for row in rows[:5]:
    for col in row:
        print("%10s" % col, end=" ")
    print('\n')
```

Output

```
Total no. of rows: 185
Field names are:Date, Open, High, Low, Close, Adj Close, Volume

First 5 rows are:

2014-09-29 100.589996 100.690002 98.040001 99.620003 93.514290 142718700
2014-10-06 99.949997 102.379997 98.309998 100.730003 94.556244 280258200
2014-10-13 101.330002 101.779999 95.180000 97.669998 91.683792 358539800
2014-10-20 98.320000 105.489998 98.220001 105.220001 98.771042 358532900
2014-10-27 104.849998 108.040001 104.699997 108.000000 101.380676 220230600
```


Explanation:

- **with open(...)** opens the CSV file in read mode safely using a context manager.
- **csv.reader(csvfile)** turns the file into a CSV reader object.
- **next(csvreader)** extracts the first row as column headers.
- Loop through **csvreader** to append each row (as a list) to rows.
- Print total rows, headers and first 5 data rows in a formatted view.

5.7.1.2 Reading CSV Files Into a Dictionary With csv

We can read a CSV file into a dictionary using the csv module in Python and the csv.DictReader class. Here's an example:

Suppose, we have a **employees.csv** file and content inside it will be:

```
name,department,birthday_month
John Smith,HR,July
Alice Johnson,IT,October
Bob Williams,Finance,January
```

Example: This reads each row as a dictionary (headers as keys), then appends it to list .

```
import csv
with open('employees.csv', mode='r') as file:
    csv_reader = csv.DictReader(file) # Create DictReader

    data_list = [] # List to store dictionaries
    for row in csv_reader:
        data_list.append(row)

for data in data_list:
    print(data)
```

Output:

```
{'name': 'John Smith', 'department': 'HR', 'birthday_month': 'July'}
{'name': 'Alice Johnson', 'department': 'IT', 'birthday_month': 'October'}
{'name': 'Bob Williams', 'department': 'Finance', 'birthday_month': 'January'}
```

Explanation:

- **with open(...)** opens the file using a context manager.

- **csv.DictReader(file)** reads each row as a dictionary using headers as keys.
- **data_list.append(row)** stores each dictionary in a list.

5.7.1.3 Writing to a CSV file

To write to a CSV file, we first open the CSV file in WRITE mode. The file object is converted to csv.writer object and further operations takes place. Code and detailed explanation is given below.

```
import csv

# Define header and data rows
fields = ['Name', 'Branch', 'Year', 'CGPA']
rows = [
    ['Nikhil', 'COE', '2', '9.0'],
    ['Sanchit', 'COE', '2', '9.1'],
    ['Aditya', 'IT', '2', '9.3'],
    ['Sagar', 'SE', '1', '9.5'],
    ['Prateek', 'MCE', '3', '7.8'],
    ['Sahil', 'EP', '2', '9.1']
]

filename = "university_records.csv"
with open(filename, 'w') as csvfile:
    csvwriter = csv.writer(csvfile)           # Create writer object
    csvwriter.writerow(fields)                # Write header
    csvwriter.writerows(rows)                 # Write multiple rows
```

Explanation:

- **fields** defines the column headers and rows contains the data as a list of lists.
- with **open(..., 'w')** opens the file in write mode using a context manager.
- **csv.writer(csvfile)** creates a writer object for writing to the CSV.
- **writerow(fields)** writes the header row to the file.
- **writerows(rows)** writes all data rows to the CSV at once.

5.7.1.4 Writing a dictionary to a CSV file

To write a dictionary to a CSV file, the file object (csvfile) is converted to a DictWriter object. Detailed example with explanation and code is given below.

```
# importing the csv module
import csv

# my data rows as dictionary objects
mydict = [{'branch': 'COE', 'cgpa': '9.0',
            'name': 'Nikhil', 'year': '2'},
          {'branch': 'COE', 'cgpa': '9.1',
            'name': 'Sanchit', 'year': '2'},
          {'branch': 'IT', 'cgpa': '9.3',
            'name': 'Aditya', 'year': '2'},
          {'branch': 'SE', 'cgpa': '9.5',
            'name': 'Sagar', 'year': '1'},
          {'branch': 'MCE', 'cgpa': '7.8',
            'name': 'Prateek', 'year': '3'},
          {'branch': 'EP', 'cgpa': '9.1',
            'name': 'Sahil', 'year': '2'}]

# field names
fields = ['name', 'branch', 'year', 'cgpa']

# name of csv file
filename = "university_records.csv"

# writing to csv file
with open(filename, 'w') as csvfile:
    # creating a csv dict writer object
    writer = csv.DictWriter(csvfile, fieldnames=fields)

    # writing headers (field names)
    writer.writeheader()

    # writing data rows
    writer.writerows(mydict)
```

Output

name	branch	year	cgpa
Nikhil	COE	2	9.0
Sanchit	COE	2	9.1
Aditya	IT	2	9.3
Sagar	SE	1	9.5
Prateek	MCE	3	7.8
Sahil	EP	2	9.1

csv file

Consider that a CSV file looks like this in plain text:

```
university_records - Notepad
File Edit Format View Help
Name;Branch;Year;CGPA
Nikhil;COE;2;9.0
Sanchit;COE;2;9.1
Aditya;IT;2;9.3
Sagar;SE;1;9.5
university record
```

Explanation:

- **with open(...)** opens file safely using a context manager.
- **csv.DictWriter(...)** maps dictionary keys to CSV columns.
- **writeheader()** writes column headers.
- **writerows(mydict)** writes all dictionaries as CSV rows.

5.7.2 SIMPLE WAYS TO READ TSV FILES IN PYTHON

Reading a TSV (Tab-Separated Values) file in [Python](#) involves parsing the file and extracting structured data where each field is separated by a tab character (`\t`). In this article, we will use a sample file named **Dani.tsv** to demonstrate how to read data from a TSV file.

GeekforGeeks - Notepad

File Edit Format View Help

student_name	school_name	address	county	phone
Aleshia	Alan D Rosenberg Cpa Pc	14 Taylor St	Kent	01944-36997
Evan	Cap Gemini America	5 Binney St	Buckinghamshire	01714-73768
France	"Elliott, John W Esq"	8 Moor Place	Bournemouth	01935-82166
Ulysses	"Mcmahan, Ben L"	505 Exeter Rd	Lincolnshire	01302-60130
Tyisha	Champagne Room	5396 Forth Street	West Midlands	01290-36728
Eric	"Thompson, Michael C Esq"	9472 Lind St	Northamptonshire	01545-81735
Marg	Wrangle Hill Auto Auct & Slvg	7457 Cowl St #70	Southampton	01362-62052
Laquita	In Communications Inc	20 Gloucester Pl #96	Tyne & Wear	01590-98248
Lura	Bizerba Usa Inc	929 Augustine St	South Gloucestershire	01340-71391
Yvette	Max Video	45 Bradfield St #166	Derbyshire	01933-51253

5.7.2.1 Using `read_csv(sep='\t')`

Pandas library provides a powerful `read_csv()` function and by simply setting `sep='\t'`, you can handle TSV files effortlessly. It reads the entire file into a structured DataFrame, making it perfect for data analysis and manipulation.

```
import pandas as pd
df = pd.read_csv('dani.tsv', sep='\t')
print(df)
```

	student_name	school_name	address	\
0	Aleshia	Alan D Rosenberg Cpa Pc	14 Taylor St	NaN
1	Evan	Cap Gemini America	5 Binney St	NaN
2	France	"Elliott, John W Esq"	8 Moor Place -	NaN
3	Ulysses	"McMahon, Ben L"	505 Exeter Rd	NaN
4	Tyisha	Champagne Room	5396 Forth Street West	NaN
5	Eric	"Thompson, Michael C Esq"	9472 Lind St	NaN
6	Marg	Wrangle Hill Auto Auct & Slvg	7457 Cowl st	#70
7	Laquita	In Communications Inc	20 Gloucester Pl #96 Tyne &	NaN
8	Lura	Bizerba Usa Inc	929 Augustine St South	NaN
9	Yvette	Max Video	45 Bradfield St	#166

	county	phone
0	Kent.	1944-36997
1	Buckinghamshire	01714-73768
2	Bournemouth	1935-82166
3	Lincolnshire	01302-60130
4	Midlands	91290-36728
5	Northamptonshire	1545-81735
6	Southampton	01362-62052
7	Wear	01590-98248
8	Gloucestershire	01340-71391
9	Derbyshire	1933-51253

Explanation: `pd.read_csv(...)` reads a tab-separated file using pandas and stores the data in **df**, a DataFrame that simplifies viewing, analyzing and handling tabular data.

5.7.2.2 Using `csv.reader()`

This method is great if you're looking for a lightweight and built-in solution. Python's `csv` module can read TSV files just by setting the delimiter to `'\t'`. It reads the file line by line and returns each row as a list.

```
import csv
with open("dani.tsv", newline="") as file:
    tsv_reader = csv.reader(file, delimiter="\t")
    for row in tsv_reader:
        print(row)
```

```
[
    'student_name', 'school_name', 'address', 'county', 'phone'
]
['Aleshia', 'Alan D Rosenburg Cpa Pc 14 Taylor St', '', 'Kent.', '1944-36997']
['Evan', 'Cap Gemini America 5 Binney St', '', 'Buckinghamshire', '01714-73768']
['France', '"Elliott, John W Esq" 8 Moor Place -', '', 'Bournemouth', '1935-82166']
['Ulysses', '"McMahon, Ben L" 505 Exeter Rd', '', 'Lincolnshire', '01302-60130']
['Iyisha', 'Champagne Room 5396 Forth Street West', '', 'Midlands', '91290-36728']
['Eric', '"Thompson, Michael C Esq" 9472 Lind St', '', 'Northamptonshire', '1545-81735']
['Marg', 'Wrangle Hill Auto Auct & Slvg 7457 Cowel st', '#70', 'Southampton', '01362-62052']
['Laquita', 'In Communications Inc 20 Gloucester Pl #96 Tyne &', '', 'Wear', '01590-98248']
['Lura', 'Bizerba Usa Inc 929 Augustine St South', '', 'Gloucestershire', '01340-71391']
['Yvette', 'Max Video 45 Bradfield St', '#166', 'Derbyshire', '1933-51253']
```

Explanation: `csv.reader()` reads the TSV file line by line, splitting each row into a list using the tab ('\t') delimiter. Each line is printed as a list of values.

5.7.2.3 Using Generator expression

This is a minimal and memory-friendly approach for quick tasks or when working with very large files line by line. It doesn't require any libraries and processes each line as it's read.

```
with open("dani.tsv") as f:
    data = (line.strip().split('\t') for line in f)
    for row in data:
        print(row)
```

```
[
    'student_name', 'school_name', 'address', 'county', 'phone'
]
['Aleshia', 'Alan D Rosenburg Cpa Pc 14 Taylor St', '', 'Kent.', '1944-36997']
['Evan', 'Cap Gemini America 5 Binney St', '', 'Buckinghamshire', '01714-73768']
['France', '"Elliott, John W Esq" 8 Moor Place -', '', 'Bournemouth', '1935-82166']
['Ulysses', '"McMahon, Ben L" 505 Exeter Rd', '', 'Lincolnshire', '01302-60130']
['Iyisha', 'Champagne Room 5396 Forth Street West', '', 'Midlands', '91290-36728']
['Eric', '"Thompson, Michael C Esq" 9472 Lind St', '', 'Northamptonshire', '1545-81735']
['Marg', 'Wrangle Hill Auto Auct & Slvg 7457 Cowel st', '#70', 'Southampton', '01362-62052']
['Laquita', 'In Communications Inc 20 Gloucester Pl #96 Tyne &', '', 'Wear', '01590-98248']
['Lura', 'Bizerba Usa Inc 929 Augustine St South', '', 'Gloucestershire', '01340-71391']
['Yvette', 'Max Video 45 Bradfield St', '#166', 'Derbyshire', '1933-51253']
```

Explanation: **Generator expression** read the file line by line, strips any leading spaces, splits the line by tabs and prints each row as a list.

5.7.2.4 Using csv.DictReader()

If you want to access each row using column headers as keys, **csv.DictReader()** is a perfect fit. It reads each row into a dictionary where keys are the column names, making your code more readable.

```
import csv
with open('dani.tsv', newline='') as f:
    reader = csv.DictReader(f, delimiter='\t')
    for row in reader:
        print(row)
```

```
[{'student_name': 'Aleshia', 'school_name': 'Alan D Rosenberg Cpa Pc 14 Taylor St', 'address': 'Kent.', 'county': '1944-36997'}]
[{'student_name': 'Evan', 'school_name': 'Cap Gemini America 5 Binney St', 'address': 'Buckinghamshire', 'county': '01714-73768'}]
[{'student_name': 'France', 'school_name': 'Elliott, John W Esq 8 Moor Place -', 'address': 'Bournemouth', 'county': '1935-82166'}]
[{'student_name': 'Ulysses', 'school_name': 'McMahon, Ben L 505 Exeter Rd', 'address': 'Lincolnshire', 'county': '01302-60130'}]
[{'student_name': 'Tyisha', 'school_name': 'Champagne Room 5396 Forth Street West', 'address': 'Midlands', 'county': '91290-36728'}]
[{'student_name': 'Eric', 'school_name': 'Thompson, Michael C Esq 9472 Lind St', 'address': 'Northamptonshire', 'county': '1545-81735'}]
[{'student_name': 'Marg', 'school_name': 'Wrangle Hill Auto Auct & Slvg 7457 Cowl st', 'address': 'Southampton', 'county': '01362-62052'}]
[{'student_name': 'Laquita', 'school_name': 'In Communications Inc 20 Gloucester Pl #96 Tyne &', 'address': 'Wear', 'county': '01590-98248'}]
[{'student_name': 'Lura', 'school_name': 'Bizerba Usa Inc 929 Augustine St South', 'address': 'Gloucestershire', 'county': '01340-71391'}]
[{'student_name': 'Yvette', 'school_name': 'Max Video 45 Bradfield St', 'address': 'Derbyshire', 'county': '1933-51253'}]
```

Using csv.DictReader()

Explanation: **csv.DictReader()** reads the TSV file and returns each row as an ordered dictionary, using the first line of the file as column headers.

5.8 Practical Exercise :

Opening, Closing, Reading, Writing, and Sorting Data in Files

Objective:

- Learn to open and close files.

- Read and write data in a formatted way.
- Sort data from files.

Practical 1: Writing and Reading Data in a File

Writing formatted data to a file with `open("data1.txt", "w")` as file:

```
file.write("ID\tName\tMarks\n")  
  
file.write("1\tAlice\t85\n")  
  
file.write("2\tBob\t90\n")  
  
file.write("3\tCharlie\t78\n")
```

Reading data from the file with `open("data1.txt", "r")` as file:

```
for line in file:  
    print(line, end="")
```

Output:

ID	Name	Marks
1	Alice	85
2	Bob	90
3	Charlie	78

Practical 2: Appending Data to a File

Appending new student data with open("data1.txt", "a") as file:

```
file.write("4\tDavid\t88\n")
```

Reading updated file
with open("data1.txt", "r") as file:

for line in file:

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

Output:

ID	Name	Marks
1	Alice	85
2	Bob	90
3	Charlie	78
4	David	88

Practical 3: Sorting Data from a File

Reading file and sorting based on marks

data = [] with open("data1.txt", "r") as file:

```
next(file) # Skip header
```

for line in file:

```
parts = line.strip().split("\t")
```

```
data.append((parts[0], parts[1], int(parts[2])))
```

Sorting based on marks

```
data.sort(key=lambda x: x[2], reverse=True)
```

```
print("Sorted Data (Highest Marks  
First):")print("ID\tName\tMarks")for record in data:
```

```
    print(f'{record[0]}\t{record[1]}\t{record[2]}")
```

Output:

Sorted Data (Highest Marks First):

ID	Name	Marks
2	Bob	90
4	David	88
1	Alice	85
3	Charlie	78

Key Points:

- Always **close files** or use `with` to manage automatically.
- Use **append mode** to add data without overwriting.
- Use **lists and sorting** to organize file data.

5.9 IMPORTANT TWO MARKS

1. What is the difference between `'open()'` modes `'w'` and `'a'`?

The `'w'` mode creates a new file or overwrites an existing file, while the `'a'` mode adds new data at the end of the file without removing old content.

2. What does the `'close()'` method do in file handling?

It closes the file object and ensures all data is properly saved from the buffer to the disk.

3. How do you read a file line by line in Python?

We can use a loop like `'for line in file:'` or use the `'readline()'` method to read one line at a time.

4. Which function is used to write data into a file?

The `'write()'` function is used to write strings or converted numbers into a file.

5. Which function is used to create a new directory in Python?

The `'os.mkdir('foldername')'` function is used to create a new directory in the given path.

6. What is the use of `os.getcwd()`?

It returns the current working directory of the program where files are being executed.

7. What does `sys.argv` represent?

It is a list in Python that stores all command-line arguments passed to a script, including the script name.

8. How to remove a directory using the OS module?

The `os.rmdir('foldername')` function is used to remove an empty directory from the system.

9. How do you write numbers to a text file in Python?

Numbers must be converted to string using `str()` and then written into the file using `write()`.

10. Which method is used to read the entire file content at once?

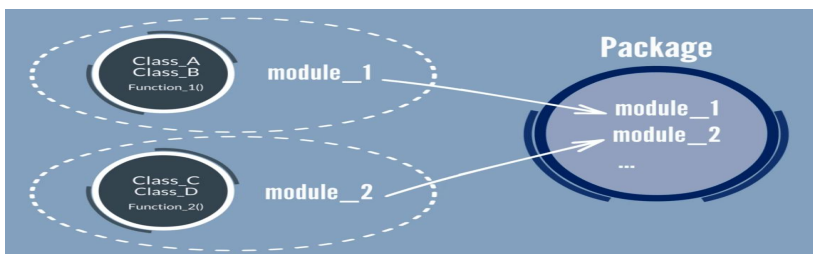
The `read()` method reads all the contents of a file as a single string.

UNIT VI

PACKAGES

Built-in modules, User-Defined modules, Numpy, SciPy, Pandas, Scikitlearn. Practical: Usage of modules and packages to solve problems. (Minimum three), Project (Minimum Two)

6. INTRODUCTION TO MODULES AND PACKAGES



What is Module in Python?

The module is a simple Python file that contains collections of functions and global variables and with having a .py extension file. It is an executable file and to organize all the modules we have the concept called Package in Python.

Examples of modules:

1. [Datetime](#)
2. [Regex](#)
3. [Random](#) etc.

Example: Save the code in a file called demo_module.py

```
def myModule(name):
```

```
    print("This is My Module : "+ name)
```

Import module named demo_module and call the myModule function inside it.

```
import demo_module
```

```
demo_module.myModule("Math")
```

Output:

```
This is My Module : Math
```

What is Package in Python?

The package is a simple directory having collections of modules. This directory contains Python modules and also having `__init__.py` file by which the interpreter interprets it as a Package. The package is simply a namespace. The package also contains sub-packages inside it.

Examples of Packages:

1. [Numpy](#)
2. [Pandas](#)

Example:

```
Student(Package)
```

```
| __init__.py (Constructor)
```

```
| details.py (Module)
```

```
| marks.py (Module)
```

```
| collegeDetails.py (Module)
```

What is Library in Python

The **library** is having a collection of related functionality of codes that allows you to perform many tasks without writing your code. **It is a reusable chunk of code that we can use by importing it into our program**, we can just use it by importing that library and calling the method of that library with a period(.). However, it is often assumed that while a package is a collection of modules, a library is a collection of packages.

Example:

Importing pandas library and call read_csv method using an alias of pandas i.e. pd.

```
import pandas as pd
df = pd.read_csv("file_name.csv")
```

6.1 Built-in MODULES

Python Module is a file that contains built-in functions, classes, **its** and variables. There are many **Python modules**, each with its specific work.

In this article, we will cover all about Python modules, such as How to create our own simple module, Import Python modules, From statements in Python, we can use the alias to rename the module, etc.

6.1.1 What is Python Module

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code.

Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

6.1.2 Create a Python Module

To create a Python module, write the desired code and save that in a file with **.py** extension. Let's understand it better with an example:

Example:

Let's create a simple `calc.py` in which we define two functions, one **add** and another **subtract**.

```
# A simple module, calc.py
def add(x, y):
    return (x+y)

def subtract(x, y):
    return (x-y)
```

6.1.3 Import module in Python

We can import the functions, and classes defined in a module to another module using the **import statement** in some other Python source file.

When the interpreter encounters an import statement, it imports the module if the module is present in the search path.

***Note:** A search path is a list of directories that the interpreter searches for importing a module.*

For example, to import the module `calc.py`, we need to put the following command at the top of the script.

6.1.4 Syntax to Import Module in Python

```
import module
```

Note: This does not import the functions or classes directly instead imports the module only. To access the functions inside the module the dot(.) operator is used.

Importing modules in Python Example

Now, we are importing the `calc` that we created earlier to perform add operation.

```
# importing module calc.pyimport calc  
print(calc.add(10, 2))
```

Output:

```
12
```

6.1.5 Python Import From Module

Python's `from` statement lets you import specific attributes from a module without importing the module as a whole.

Import Specific Attributes from a Python module

Here, we are importing specific `sqrt` and `factorial` attributes from the `math` module.

```
# importing sqrt() and factorial from the  
# module math
```

```
from math import sqrt, factorial
```

```
# if we simply do "import math", then  
# math.sqrt(16) and math.factorial()  
# are required.
```

```
print(sqrt(16))  
print(factorial(6))
```

Output:

```
4.0  
720
```

6.1.6 Import all Names

The * symbol used with the import statement is used to import all the names from a module to a current namespace.

Syntax:

```
from module_name import *
```

6.1.6 What does import * do in Python?

The use of * has its advantages and disadvantages. If you know exactly what you will be needing from the module, it is not recommended to use *, else do so.

```
# importing sqrt() and factorial from the  
# module math  
from math import *
```

```
# if we simply do "import math", then  
# math.sqrt(16) and math.factorial()
```

are required.

```
print(sqrt(16))
```

```
print(factorial(6))
```

Output

```
4.0
```

```
720
```

6.1.7 Locating Python Modules

Whenever a module is imported in Python the interpreter looks for several locations. First, it will check for the [built-in module](#), if not found then it looks for a list of directories defined in the [sys.path](#). Python interpreter searches for the module in the following manner -

- First, it searches for the module in the current directory.
- If the module isn't found in the current directory, Python then searches each directory in the shell variable [PYTHONPATH](#). The PYTHONPATH is an environment variable, consisting of a list of directories.
- If that also fails python checks the installation-dependent list of directories configured at the time Python is installed.

6.1.8 Directories List for Modules

Here, sys.path is a built-in variable within the sys module. It contains a list of directories that the interpreter will search for the required module.

```
# importing sys module
```

```
import sys
```

```
# importing sys.path
```

```
print(sys.path)
```

Output:

```
['/home/nikhil/Desktop/gfg',  
'/usr/lib/python38.zip',      '/usr/lib/python3.8',  
'/usr/lib/python3.8/lib-dynload',  
'/home/nikhil/.local/lib/python3.8/site-packages',  
'/usr/local/lib/python3.8/dist-packages',  
'/usr/lib/python3/dist-packages',  
'/usr/local/lib/python3.8/dist-  
packages/IPython/extensions',  
'/home/nikhil/.ipython']
```

6.1.9 Renaming the Python Module

We can rename the module while importing it using the keyword.

Syntax: *Import Module_name as Alias_name*

```
# importing sqrt() and factorial from the  
# module math  
import math as mt  
  
# if we simply do "import math", then  
# math.sqrt(16) and math.factorial()  
# are required.  
print(mt.sqrt(16))  
print(mt.factorial(6))
```

Output

4.0

720

6.2 PACKAGES

- Packages are namespaces which contains many modules, many packages.
- Along with packages and modules the package contains a file named.
- `__init__.py`. In fact to be a package there must be a file called `__init__.py` in the folder.

6.2.1 How to create a Package

Step1:

Create a folder named

My Maths in your working drive.

Step 2: Inside My Maths directory create a folder `--init-Py`

step 3: create a folder named Add. inside

step 4: Inside Add create a Python file named.

Add.py.

```
def add- fun (a,b):  
    Add atb  
    return Add.
```

step 5: Similarly, Inside Add create a python file named sub.py.

```
def a sub- fun (a,b),
```

```
Sub = a-b  
return sub
```

step 6: similarly, Inside Add python file named Mul.py

```
def mul_fun(a,b)  
Mul = a*b  
return Mul
```

Step 7: Inside Add create a python file named Div.Py

```
def div_fun (a, b)  
Div a/b;  
return div
```

Step 8 : Now on the D: create a driven program which will involve all the above this functionalities present in the My Maths package. We have named this driver testing_arithmetic.

Step 9 : Next Page

Step 10:

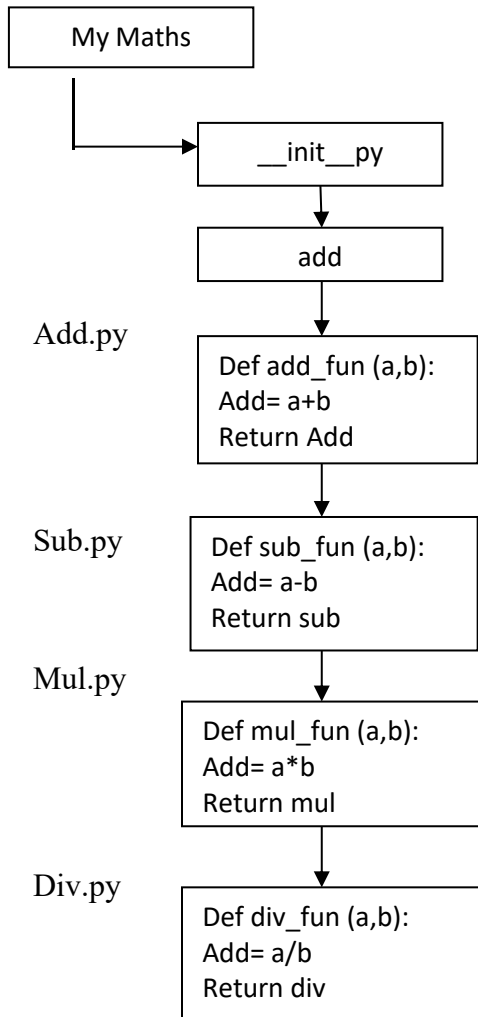
Output:

The addition of 10 and 20 is: 30

The Subtraction of 10 and 20 is: -10

The Multiplication of 10 and 20 is: 200

The Division of 10 and 20 is: 0.5



Step 9:

```
Import MyMaths.Add.add
```

```
Import MyMaths.Sub.sub
```

```
Import MyMaths.Mul.mul
```

```
Import MyMaths.Div.div
```

```
Print (“The Addition of 10 and 20 is:”)
```

```
MyMaths.Add.add.add_fun(10,20)
```

```
Print (“The Subraction of 10 and 20 is:”)
```

```
MyMaths.Sub.sub.sub_fun(10,20)
```

```
Print (“The Multiplication of 10 and 20 is:”)
```

```
MyMaths.Mul.mul.mul_fun(10,20)
```

```
Print (“The Division of 10 and 20 is:”)
```

```
MyMaths.Div.div.div_fun(10,20)
```

6.3 BUILD-IN MODULES

Python is one of the most popular programming languages because of its vast collection of modules which make the work of developers easy and save time from writing the code for a particular task for their program. Python provides various types of modules which include Python built-in modules and external

modules. In this article, we will learn about the built-in modules in Python and how to use them.

What are Python Built-in modules?

Python built-in modules are a set of libraries that come pre-installed with the Python installation. It will become a tedious task to install any required modules in between the process of development that's why Python comes with some most used modules required. These modules provide a wide range of functionalities, from file operations and system-related tasks to mathematical computations and web services. The use of these modules simplifies the development process, as developers can leverage the built-in functions and classes for common tasks, providing code reusability, and efficiency. Some examples of Python built-in modules include **"os"**, **"sys"**, **"math"**, and **"datetime"**.

We can run the below command to get the all available modules in Python:

```
help('modules')
```

```
C:\Users\kamalzpython
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v.1934 64 b
it (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> help('modules')

Please wait a moment while I gather a list of all available modules...

test_sqlite3: testing with version '2.6.0', sqlite_version '3.42.0'
C:\Program Files\Python311\Lib\pkgutil.py:92: UserWarning: The numpy.array_a
pi submodule is still experimental. See NEP 47.
  _import__(_info.name)
C:\Program Files\Python311\Lib\site-packages\distutils_hack\__init__.py:33:
UserWarning: Setuptools is replacing distutils.
  warnings.warn("Setuptools is replacing distutils.")
pIL
__future__          _testconsole         fontTools           quopri
__hello__           _testconsole_d       fractions           random
_phello__           _testimportmultiple  ftplib              re
__abc               _testimportmultiple_d  functools           reprlib
__abc               _testinternalcapi     gc                  requests
__aix_support       _testinternalcapi_d   genericpath         rfcompleter
__ast               _testmultiphase       getopt              runpy
__asyncio            _testmultiphase_d     getpass             sched
__asyncio_d         _thread              glob                 secrets
__bisect             _threading_local      graphlib            select
__blake2             tkinter               gzip                 select_d
__bootsubprocess    _tokenize             hashlib             selectors
__bz2               _tracemalloc          heapq               setuptools
__bz2_d             _typing              html                 shelve
__codecs             _uuid                 http                 shlex
__codecs_cn         _warnings             idlelib             shutil
__codecs_hk         _weakref              idna                 signal
__codecs_iso2022    _weakrefset           imaplib             site
__codecs_jp         _winapi               imghdr              six
__codecs_lr         _xsubinterpreters     imp                 smtpd
__codecs_tw         _zoneinfo             importlib            smtpplib
__collections        _zoneinfo             importlib            sndhdr
__collections_abc    _zoneinfo             importlib            socket
```

6.3.1 ADVANTAGES OF BUILT-IN MODULES IN PYTHON

- **Reduced Development Time :** Built-in modules in Python are made to perform various tasks that are used without installing external module or writing the lengthy code to perform that specific task.
- **Optimized Performance:** Some Python built-in modules are optimized for performance, using low-level code or native bindings to execute tasks efficiently.
- **Consistency:** Python Built-in modules provide consistent way to solve problems. Developers familiar with these modules can easily understand and collaborate on codebases across different projects.
- **Standardization :** As these modules are part of the Python Standard Library, they establish a standard way of performing tasks.
- **Documentation :** Python's official documentation is comprehensive and includes detailed explanations and examples for Python built-in modules. This makes it easier to learn and utilize them.
- **Maintainability :** Python Built-in modules are maintained by the core Python team and community, regular updates, bug fixes, and improvements are to be expected, ensuring long-term viability.

- **Reduced Risk** :Using third-party libraries can introduce risks, like discontinued support or security vulnerabilities.

6.3.2 JSON MODULE IN PYTHON

"json" module in Python is used to encode and decode JSON data. JSON format is a widely used on the web to exchange the information. It is extremely useful for reading and writing data in the JSON format.

Example:

In the below example, we have used json module to convert the Python dictionary into JSON format. Firstly, we import the **"json"** module and then define the Python dictionary after that we convert the Python dictionary into JSON format using **"json.dumps()"** method of **"json"** module in Python and finally print the JSON data.

```
import json data = {  
    "name": "Jonny",  
    "age": 30,  
    "is_student": True,  
    "courses": ["Web Dev", "CP"]}  
json_string = json.dumps(data,  
indent=4)  
print(json_string)
```

OUTPUT

```
{  
    "name": "Jonny",  
    "age": 30,  
    "is_student": true,
```

```
"courses": [  
    "Web Dev",  
    "CP"  
]  
}
```

6.3.3 TKINTER MODULE IN PYTHON

"tkinter" is the standard GUI (Graphical User Interface) library in Python. **"tkinter"** module is used to create windows, buttons, text boxes, and other UI elements for desktop applications.

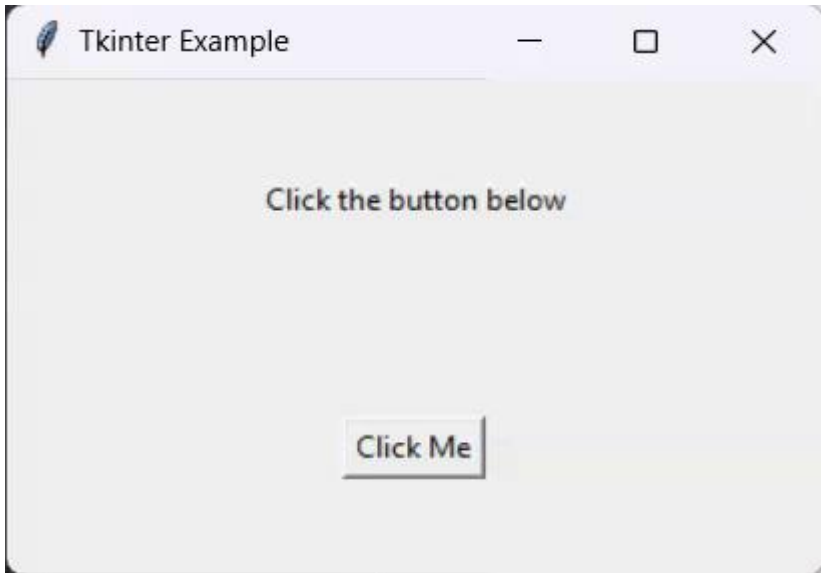
Example

Firstly we imports the "tkinter" module as "tk" and defines a function, **"on_button_click"**, which updates a label's text. A main GUI window titled **"Tkinter Example"** is created, containing a label and a button. The label displays **"Click the button below"**, and the button, labeled **"Click Me"**, triggers the previously defined function when pressed. The application remains responsive to user interactions by using the **"root.mainloop()"** event loop, allowing the label's message to change upon button clicks.

```
import tkinter as tk def on_button_click():  
    label.config(text="Hello, Geeks!")  
root = tk.Tk()root.title("Tkinter Example") label = tk.Label(root,  
text="Click the button below") label.pack(pady=40) button =
```

```
tk.Button(root, text="Click Me", command=on_button_click)  
button.pack(pady=40) root.mainloop()
```

Output: In the below output we can see that when we click on "Click Me" button the text changes.



6.3.4 RANDOM MODULE IN PYTHON

The "random" module in Python is used to generate random numbers and provides the functionality of various random operations such as '**random.randint()**', '**random.choice()**', '**random.random()**', '**random.shuffle()**' and many more.

Example

In the below code, firstly we import the random module, then we are printing the random number from "1 to 10" and

random item from the list by using **random.randint()** and **random.choice()** methods of random module respectively.

```
import random
num = random.randint(1, 10)
print(f"Random integer between 1 and 10: {num}")
fruits = ["Java", "C", "C++", "Python"]
chosen_fruit = random.choice(fruits)
print(f"Randomly chosen language: {chosen_fruit}")
```

Output

Random integer between 1 and 10: 9

Randomly chosen language: Python

6.3.5 MATH MODULE IN PYTHON

The math module offers mathematical functions used for advanced arithmetic operations. This includes trigonometric functions, logarithmic functions, and constants like **pi** and **e**. This module is used to perform complex calculations using Python program.

Example

In the below example, we have used math module to find the square root of a number using **math.sqrt()** method and the value of PI using **math.pi** method and then print the result using **print()** function of Python.

```
import math
sqrt_val = math.sqrt(64)
pi_const = math.pi
print(sqrt_val)
print(pi_const)
```

Output

8.0

3.141592653589793

6.3.6 DATETIME MODULE IN PYTHON

The "**datetime**" module allows for manipulation and reading of date and time values. Some of the basic method of "**datetime**" module are "**datetime.date**", "**datetime.time**", "**datetime.datetime**", and "**datetime.timedelta**".

Example

In the below example, we have print the today' date and current time by using **datetime.date.today()** method and **datetime.datetime.now().time()** method of "**datetime**" module in Python.

```
import datetime
date_today = datetime.date.today()
time_now = datetime.datetime.now().time()
print(date_today)
print(time_now)
```

Output

```
2023-10-19
```

```
07:28:16.279090
```

6.3.7 OS MODULE IN PYTHON

The "**os**" module in Python is used to interact with the operating system and offers OS-level functionalities. For example, interacting with file system, reading directory, and launching application.

Example

In the below example, we have used "**os**" module to fetch the directory path using **os.getcwd()** method and then print the path.

```
import os
directory = os.getcwd()
print(directory)
```

Output

```
/home/guest/sandbox
```

6.3.8 SYS MODULE IN PYTHON

The "sys" module in Python provides functions and variables that interact with the Python runtime environment. It allows developers to access Python interpreter attributes and manipulate them. It offers a range of other functionalities, such as input/output stream access, memory info access, and more.

In the below example, we have used sys module to print the current version of Python programming language and also print the list of command line arguments passed while running the Python program.

```
import sysprint("Python version:", sys.version)print("Command  
line arguments:", sys.argv)sys.exit(1)
```

In the below output, we can see the current version is printed and command line arguments in the form of list along with the name of Python program name.

```
D:\Office\Python>python temp.py hello, How are you doing?  
Python version: 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v.1  
934 64 bit (AMD64)]  
Command line arguments: ['temp.py', 'hello,', 'How', 'are', 'you', 'doing?']
```

6.3.9 RE MODULE IN PYTHON

The re module in Python provides support for working with regular expressions. Regular expressions (often abbreviated as "regex") are powerful tools for matching strings or sets of

strings using a specialized syntax that allows for flexible pattern matching.

Example

In the below example, we write an simple example of finding the matching string using "re" module in Python. After importing "re" module we define a pattern and a text. We search for the email pattern that may present in the text using search() function and then print the first found email in the text. After that we find all occurrences of mails in the text and then print them.

```
import re
pattern = r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}"
text = "Contact us at info@example.com and support@example.org for more details."
match = re.search(pattern, text)
if match:
    print("First found email:", match.group())
emails = re.findall(pattern, text)
print("All found emails:", emails)
```

Output

First found email: info@example.com

All found emails: ['info@example.com', 'support@example.org']

6.3.10 HASHLIB MODULE IN PYTHON

The **"hashlib"** module in Python provides algorithms for message digests or hashing. It allows for data integrity checks using algorithms like SHA-256, MD5, and more.

Example

In this example, firstly, we import the hashlib module then define a sample message string. Convert the string into its SHA-256 hash representation using hashlib.sha256() function. Print the resulting hash.

```
import hashlib
message = "Hello, World!"
hashed = hashlib.sha256(message.encode()).hexdigest()
print(hashed)
```

Output

```
dffd6021bb2bd5b0af676290809ec3a53191dd81c7f70a4b28688
a362182986f
```

6.3.11 CALENDAR MODULE IN PYTHON

The calendar module allows operations and manipulations related to calendars.

Example

In this example, we print the October month of year 2023. Firstly, we import the "calendar" module. Generate a string representation of October 2023 using the **month()** function. Print the calendar for October 2023.

```
import calendar
cal_october = calendar.month(2023, 10)
print(cal_october)
```

Output

```
October 2023
Mo Tu We Th Fr Sa Su
      1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

6.3.12 HEAPQ MODULE IN PYTHON

The **"heapq"** module in Python provides a collection of functions for implementing heaps based on regular lists. A heap is a special tree-based data structure that satisfies the heap property, and it is commonly used to implement priority queues.

Example

In the below example, we have implemented the heap data structure using **"heapq"** module in Python and applied various heap methods on heap such as **heapify()**, **heappush()**, and **heappop()**;

```
import heapq
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
heapq.heapify(numbers)
heapq.heappush(numbers, 7)
print(heapq.heappop(numbers))
```

```
print(heapq.heappushpop(numbers, 8)) print(heapq.nlargest(3,  
numbers)) print(heapq.nsmallest(3, numbers))
```

Output

```
1  
1  
[9, 8, 7]  
[2, 3, 3]
```

6.4 PYTHON MODULES SUMMARY TABLE

Module	Purpose	Example Code	Output
json	Encode/Decode JSON data	<pre>import json; data={"name":"Jonny"}; print(json.dumps(data))</pre>	<pre>{"name": "Jonny"}</pre>
tkinter	GUI creation (windows, buttons, labels)	<pre>import tkinter as tk; tk.Tk().title("Example")</pre>	Opens GUI window
random	Generate random numbers & selections	<pre>import random; print(random.randint(1,10))</pre>	Random integer (e.g., 7)
math	Mathematical functions & constants	<pre>import math; print(math.sqrt(64), math.pi)</pre>	8.0 3.14159265358 9793

Module	Purpose	Example Code	Output
datetime	Work with date & time	<pre>import datetime; print(datetime.date.today())</pre>	2023-10-19
os	Interact with Operating System	<pre>import os; print(os.getcwd())</pre>	Prints current directory
sys	Python runtime & system-specific parameters	<pre>import sys; print(sys.version)</pre>	Python version
re	Regular Expressions (pattern matching)	<pre>import re; print(re.findall(r"\w+@\w+\.\w+", "mail@abc.com"))</pre>	['mail@abc.com']
hashlib	Hashing (MD5, SHA-256, etc.)	<pre>import hashlib; print(hashlib.sha256("hi".encode()).hexdigest())</pre>	SHA-256 hash string
calendar	Calendar operations	<pre>import calendar; print(calendar.month(2023,10))</pre>	Prints October 2023 calendar
heapq	Heap / Priority Queue operations	<pre>import heapq; nums=[3,1,4]; heapq.heapify(nums); print(heapq.heappop(nums))</pre>	1

6.5 USER DEFINED MODULES

User-defined modules in Python:

User-defined python modules are the modules, which are created by the user to simplify their project. These modules can contain functions, classes, variables, and other code that you can reuse across multiple scripts.

How to create a user-defined module?

We will create a module calculator to perform basic mathematical operations.

```
# calculator.py
```

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

```
    return a + b
```

```
def sub(a, b):
```

```
    return a - b
```

```
def mul(a, b):
```

```
    return a * b
```

```
def div(a, b):
```

```
    return a / b
```

In the above code, we have implemented basic mathematic operations. After that, we will save the above python file as calculator.py.

Now, we will use the above user-defined python module in another python program.

```
# main.py

import calculator

print("Addition of 5 and 4 is:", calculator.add(5, 4))

print("Subtraction of 7 and 2 is:", calculator.sub(7, 2))

print("Multiplication of 3 and 4 is:", calculator.mul(3, 4))

print("Division of 12 and 3 is:", calculator.div(12, 3))
```

Output:

Addition of 5 and 4 is: 9

Subtraction of 7 and 2 is: 5

Multiplication of 3 and 4 is: 12

Division of 12 and 3 is: 4.0

How to import python modules?

We can import python modules using keyword import. The syntax to import python modules is given below.

Syntax to Import Python Modules

```
import module_name
```

Example to Import Python Modules:

```
import math  
  
print(math.sqrt(4))
```

Output:

```
2.0
```

In the above program, we imported all the attributes of module math and we used the sqrt function of that module.

Now, let's see how we can import specific attributes from the python module.

To import specific attributes or functions from a particular module we can use keywords from along with import.

Syntax to import python module using Attribute:

```
from module_name import attribute_name
```


Example of import python module using Attribute:

```
from math import sqrt  
  
print(sqrt(4))
```

Output:

```
2.0
```

Now, let's see how we can import all the attributes or functions from the module at the same time.

We can import all the attributes or functions from the module at the same time using the * sign.

Syntax to import python module using all Attributes:

```
from module_name import *
```

Example to import python module using all Attributes:

```
from math import *  
  
print(sqrt(4))  
  
print(log2(8))
```

Output:

```
2.0
```

3.0

Python modules are instrumental in code reuse, program organization, and enhancing the functionality of Python programs. They enable developers to separate implementation details, improve code readability, and maintain large-scale projects effectively. With a vast collection of built-in modules, Python provides a rich ecosystem of functionalities, making it easier to accomplish a wide range of tasks without the need for additional package installations.

6.6 POPULAR PYTHON PACKAGES FOR ENGINEERS

Python is a very versatile language, thanks to its huge set of libraries which makes it functional for many kinds of operations. Its versatile nature makes it a favorite among new as well as old developers. As we have reached the year 2025 **Python** language continues to evolve with **new libraries and updates** getting added to it which enhance its capabilities.

The developers must be familiar with at least the most popular libraries. In this article, we will look at some of the **Python libraries that every developer should explore at least once**.

6.6.A.What is a Library?

In a programming language context, **a library refers to the collection of pre-written code modules that serve a specific functionality. These modules are reusable, these are integrated into the programmer's code which increases the development process and functionality of the software.** It is

an **encapsulation of common tasks or complex sets of algorithms** that provide a set of functions that a developer can use to his advantage without having to create software from scratch. These are repositories of code that promote code reuse, modularization, and collaboration with the programming community. Popular languages such as **Java, Python, and JavaScript** have many libraries that cover diverse domains making software development easier.

6.6 .B What are Python Libraries?

Python libraries are reusable code modules that contain **pre-written code**. You can integrate it into your code to save time and effort. They cover many diverse domains, such as **NumPy, which stands out for numerical computation and can very easily perform operations on large arrays and matrices. Pandas, another trendy library, is widely used for data manipulation and analysis and contains efficient data structures like DataFrames.** These and many more libraries collectively contribute to Python's popularity by making the development process easier and promoting a collaborative ecosystem.

6.6.1 NumPy

- Supports large arrays, matrices, and mathematical operations.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
```

```
print("Mean:", np.mean(arr))
```

6.6.2 SciPy

- Used for **scientific computing**.

```
from scipy import stats
data = [10, 20, 30, 40, 50]
print("Mean:", stats.tmean(data))
```

6.6.3 Pandas

- Data analysis and manipulation.

```
import pandas as pd
data = {"Name": ["A", "B"], "Age": [20, 21]}
df = pd.DataFrame(data)
print(df)
```

6.6.4 Scikit-learn

- Machine learning package.

```
from sklearn.linear_model import LinearRegression
import numpy as np
x = np.array([[1], [2], [3]])
y = np.array([2, 4, 6])
model = LinearRegression().fit(x, y)
print("Prediction:", model.predict([[4]]))
```

6.7 PYTHON LIBRARIES SUMMARY TABLE

Library	Purpose	Key Features
NumPy	Numerical computing	Multidimensional arrays, element-wise operations, linear algebra
Pandas	Data analysis	DataFrame & Series, time series handling, missing data cleaning
Matplotlib	Data visualization	Line, bar, scatter plots; pyplot for simple plotting
TensorFlow	Deep learning	Data flow graphs, runs on CPU/GPU, Google's ML framework
PyTorch	Deep learning	Tensors, dynamic computation graph, fast training
Scikit-learn	Machine learning	Classification, regression, clustering, easy API
Requests	HTTP requests	Supports GET, POST, cookies, API interaction
Keras	Neural networks (high-level)	Easy API, works with TensorFlow, supports CNN & RNN
Seaborn	Statistical visualization	Works with Pandas, attractive plots, color themes
Plotly	Interactive plots	Works with Pandas/NumPy, 3D & web-

		based charts
NLTK	Natural Language Processing	Tokenization, stemming, text analysis
Beautiful Soup	Web scraping	Parse HTML/XML, extract data easily
Pygame	Game development	Graphics & sound for 2D games
Gensim	NLP & similarity	Handles large text, topic modeling, cosine similarity
spaCy	Industrial NLP	Fast tokenization, POS tagging, entity recognition
SciPy	Scientific computing	Integrals, optimization, linear algebra
Theano	Numerical computation	GPU acceleration, integrates with NumPy
PyBrain	Neural networks	Modular design, supervised/unsupervised learning
Bokeh	Interactive visualization	Zoomable, web-embedded charts
Hebel	Deep learning on GPU	GPU acceleration, NumPy integration

6.8 Case Studies: Using File Operations & Packages in Problem Solving

Introduction

In real-world applications, data must be **stored, retrieved, and processed** efficiently. File operations help in **storing data permanently**, while Python packages provide **ready-to-use functions** to simplify problem-solving.

Case Study Example: Student Result Management System

Problem Statement

A college wants to maintain student marks in a text file. The system should:

- 1) Store student names and marks in a file.
- 2) Read data back and process it.
- 3) Use packages (statistics, os) to find average, highest scorer, and total students.

Step 1: Writing Student Data into File

```
# student_results.py

filename = "results.txt"

# Writing student marks into file

with open(filename, "w") as f:
```

```
f.write("Daniel,85\n")
```

```
f.write("Joselin,92\n")
```

```
f.write("Sam,78\n")
```

```
f.write("Lulu,88\n")
```

Step 2: Reading and Processing Data

```
import statistics
```

```
import os
```

```
students = {}
```

```
# Reading student data from file
```

```
with open("results.txt", "r") as f:
```

```
    for line in f:
```

```
        name, mark = line.strip().split(",")
```

```
        students[name] = int(mark)
```

```
# Extract marks
```

```
marks = list(students.values())
```

```
print("Student Records:", students)
```

```
print("Total Students:", len(students))
```



```
print("Average Marks:", statistics.mean(marks))  
print("Highest Marks:", max(marks))  
print("Top Scorer:", max(students, key=students.get))  
print("File Size:", os.path.getsize("results.txt"), "bytes")
```

Output

Student Records: {'Daniel': 85, 'Joselin': 92, 'Sam': 78, 'Lulu': 88}

Total Students: 4

Average Marks: 85.75

Highest Marks: 92

Top Scorer: Joselin

File Size: 34 bytes

Explanation

File Operations:

- "w" mode used to write data.
- "r" mode used to read data line by line.
- strip() removes newline \n.
- Data split into **name** and **marks** using split(",").

Packages Used:

- `statistics.mean()` → To calculate average.
- `max()` with dictionary → To find top scorer.
- `os.path.getsize()` → To check file size.

6.9 PRACTICALS

6.9.1 : Usage of modules and packages to solve problems

Practical 1: File Handling with OS Module

Aim:

To list all `.txt` files in a directory and calculate their total size using the `os` module.

Algorithm:

1. Import the `os` module.
2. Define a function to scan through a directory.
3. Use `os.listdir()` to get all files in the directory.
4. Check if the file ends with `.txt`.
5. Display the filename and add its size using `os.path.getsize()`.
6. Display the total size of all text files.

Program:

```
import os

def txt_file_info(directory):
    total_size = 0
    print("Text files in directory:")
```

```
for file in os.listdir(directory):
    if file.endswith(".txt"):
        print(file)
        total_size += os.path.getsize(os.path.join(directory, file))
print("Total size of text files:", total_size, "bytes")

txt_file_info(".")
```

Output:

Text files in directory:

notes.txt

data.txt

report.txt

Total size of text files: 5682 bytes

Result:

Thus, the program successfully listed all `.txt` files and displayed their total size.

Practical 2: Data Analysis with Math & Statistics**Aim:**

To calculate mean, median, standard deviation, and square root of maximum marks using statistics and math modules.

Algorithm:

1. Import statistics and math modules.
2. Define a list of student marks.
3. Use mean(), median(), and stdev() to calculate statistics.
4. Use math.sqrt() to calculate square root of the highest mark.
5. Print results.

Program:

```
import statistics as stats
import math

marks = [78, 82, 69, 90, 76, 84, 95]

print("Mean:", stats.mean(marks))
print("Median:", stats.median(marks))
print("Standard Deviation:", stats.stdev(marks))
print("Square root of highest mark:", math.sqrt(max(marks)))
```

Output:

```
Mean: 82.0
Median: 82
Standard Deviation: 8.54
Square root of highest mark: 9.74
```

Result:

Thus, statistical operations were successfully performed using math and statistics modules.

Practical 3: OTP Generator using Random & Datetime

Aim:

To generate a random OTP and display it with the current timestamp using random and datetime modules.

Algorithm:

1. Import random and datetime modules.
2. Use random.randint() to generate a 6-digit OTP.
3. Display the OTP.
4. Display the current date and time using datetime.now().

Program:

```
import random
from datetime import datetime

def generate_otp():
    otp = random.randint(100000, 999999)
    print("Your OTP is:", otp)
    print("Generated at:", datetime.now())

generate_otp()
```

Output:

Your OTP is: 472918

Generated at: 2025-08-31 10:15:32.452818

Result:

Thus, the program successfully generated an OTP with timestamp.

6.9..2 PROJECT**Project 1: Student Grade Management System****Aim:**

To manage student marks and calculate average scores with pass/fail status using the csv and statistics modules.

Algorithm:

1. Import csv and statistics modules.
2. Open the CSV file containing student names and marks.
3. Read data using csv.DictReader().
4. Convert marks into a list of integers.
5. Calculate the average using statistics.mean().
6. Print student name, average, and result (Pass/Fail).

Program:

```
import csv
import statistics as stats

with open("students.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
```

```
marks = list(map(int, row["marks"].split()))
avg = stats.mean(marks)
result = "Pass" if avg >= 50 else "Fail"
print(row["name"], "Average:", avg, "-", result)
```

Output:

John Average: 71.67 - Pass

Mary Average: 47.67 - Fail

David Average: 87.67 - Pass

Project 2: Weather Data Logger**Aim:**

To fetch live weather data of a city and store it in a JSON log file with timestamp using requests, json, and datetime modules.

Algorithm:

1. Import requests, json, and datetime modules.
2. Fetch weather data from wttr.in API using requests.get().
3. Extract current temperature.
4. Create a dictionary with city, temperature, and timestamp.
5. Write the dictionary into a JSON file.
6. Display the logged data.

Program:

```
import requests
import json
from datetime import datetime

def fetch_weather(city):
    url = f"https://wttr.in/{city}?format=j1"
    response = requests.get(url).json()
    temp = response['current_condition'][0]['temp_C']
    data = {"city": city, "temperature": temp, "time":
str(datetime.now())}

    with open("weather_log.json", "a") as f:
        f.write(json.dumps(data) + "\n")

    print("Logged:", data)

fetch_weather("Chennai")
```

Output:

Logged: {'city': 'Chennai', 'temperature': '31', 'time': '2025-08-31
10:25:43.378425'}

6.10 IMPORTANT TWO MARKS

1. Give two examples of built-in modules in Python?

Examples of built-in modules are ``os``, ``sys``, ``math``, and ``datetime``.

2.What is a User-Defined Module?

A user-defined module is a ``py`` file created by the programmer which can be imported and reused in other programs.

3.What is NumPy mainly used for?

NumPy is mainly used for handling large arrays, performing numerical computations, and supporting mathematical operations.

4.What is SciPy in Python?

SciPy is a scientific library built on NumPy that provides advanced features like integration, optimization, and signal processing.

5.What is Pandas mainly used for?

Pandas is mainly used for data manipulation and analysis, especially handling tabular data with Series and DataFrames.

6.What is Scikit-learn in Python?

Scikit-learn is a machine learning library that provides tools for classification, regression, clustering, and model evaluation.

7.Which data structures are mainly used in Pandas?

The two primary data structures are Series, which represents 1D data, and DataFrame, which represents 2D tabular data.

8.How do you import NumPy in Python?

NumPy is generally imported using the statement ``import numpy as np``.

```
*****
```