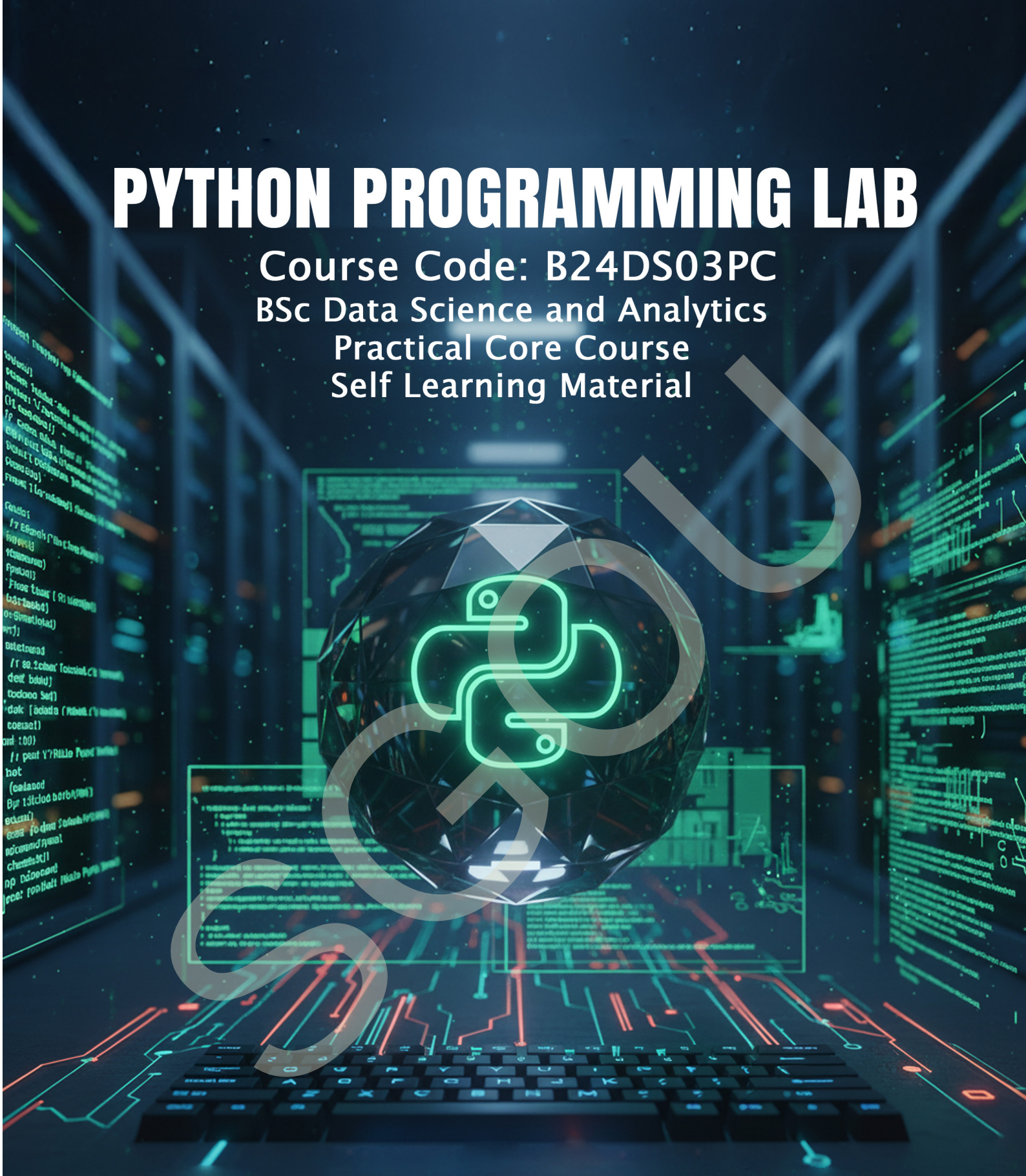
PYTHON PROGRAMMING LAB

Course Code: B24DS03PC

BSc Data Science and Analytics
Practical Core Course
Self Learning Material



SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU OPEN UNIVERSITY

Vision

To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.

Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

Pathway

Access and Quality define Equity.

Python Programming Lab

Course Code: B24DS03PC

Semester - III

Practical Core Course
Undergraduate Programme
BSc Data Science and Data Analytics
Self Learning Material

SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU
OPEN UNIVERSITY

PYTHON PROGRAMMING LAB

Course Code: B24DS03PC
Semester- III
Practical Core Course
BSc Data Science and Data Analytics

Scan this QR Code for reading the SLM on a digital device.

Edition
September 2025

www.sgou.ac.in

Visit and Subscribe our Social Media Platforms

MESSAGE FROM VICE CHANCELLOR

Dear learner,

I extend my heartfelt greetings and profound enthusiasm as I warmly welcome you to Sreenarayanaguru Open University. Established in September 2020 as a state-led endeavour to promote higher education through open and distance learning modes, our institution was shaped by the guiding principle that access and quality are the cornerstones of equity. We have firmly resolved to uphold the highest standards of education, setting the benchmark and charting the course.

The courses offered by the Sreenarayanaguru Open University aim to strike a quality balance, ensuring students are equipped for both personal growth and professional excellence. The University embraces the widely acclaimed "blended format," a practical framework that harmoniously integrates Self-Learning Materials, Classroom Counseling, and Virtual modes, fostering a dynamic and enriching experience for both learners and instructors.

The University is committed to providing an engaging and dynamic educational environment that encourages active learning. The Study and Learning Material (SLM) is specifically designed to offer you a comprehensive and integrated learning experience, fostering a strong interest in exploring advancements in information technology (IT). The curriculum has been carefully structured to ensure a logical progression of topics, allowing you to develop a clear understanding of the evolution of the discipline. It is thoughtfully curated to equip you with the knowledge and skills to navigate current trends in IT, while fostering critical thinking and analytical capabilities.The Self-Learning Material has been meticulously crafted, incorporating relevant examples to facilitate better comprehension.

Rest assured, the university's student support services will be at your disposal throughout your academic journey, readily available to address any concerns or grievances you may encounter. We encourage you to reach out to us freely regarding any matter about your academic programme. It is our sincere wish that you achieve the utmost success.

Regards,
Dr. Jagathy Raj V. P.

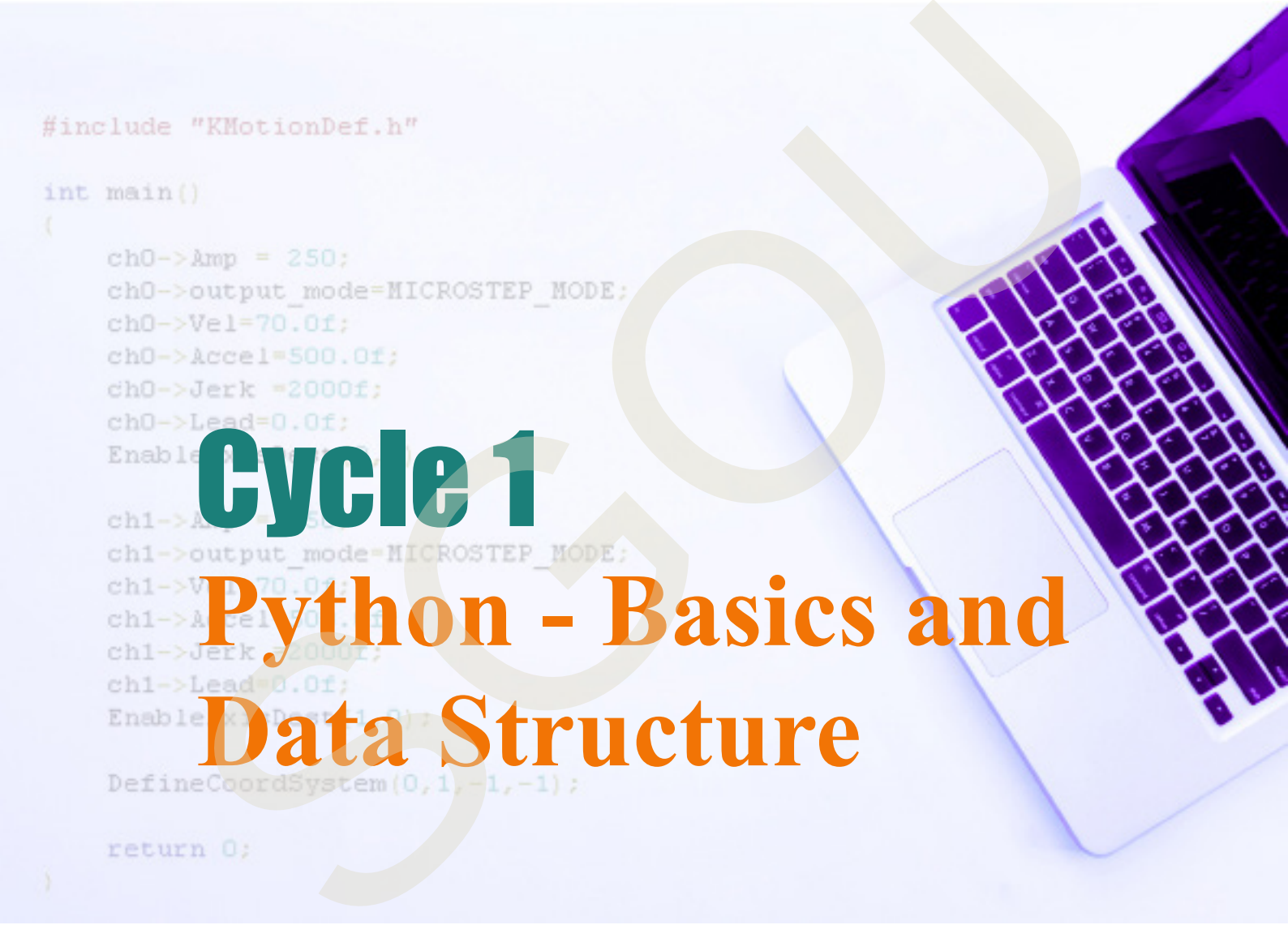
01-09-2025

Contents

Lab Manual

Python Programming

B24DS03PC



Cycle 1

Python - Basics and Data Structure

Introduction

Python is a powerful, high-level programming language known for its readability and versatility. It is widely used for web development, data science, cybersecurity, artificial intelligence, automation, and more. Python is platform-independent, running seamlessly on Windows, macOS, Linux, and embedded systems. As a high-level language, it handles memory management automatically, simplifying complex programming tasks. It offers extensibility by integrating with C, C++, and Java, while also enabling embedding in other applications. Python also supports multi-threading and concurrency, allowing for efficient execution of programs using libraries like multiprocessing. Additionally, it is widely used in cybersecurity, ethical hacking, and automation, making it a valuable tool for penetration testing, encryption, and network analysis. In the context of Data Science, Python has become the dominant language for tasks such as data cleaning, exploratory data analysis (EDA), statistical modeling, and building predictive models. It integrates seamlessly with tools for big data processing, cloud computing, and business intelligence, making it an essential skill for data analysts and scientists.

Objectives of the Python Lab

- Learn fundamental Python syntax, data types, and execution methods to build a strong programming foundation.
- Work with data structures such as lists, tuples, sets, and dictionaries to understand how to store and manipulate data effectively.
- Implement control statements like loops, conditionals, and functions to structure programs efficiently.
- Perform file input and output operations to read and write data for processing tasks.
- Develop problem-solving skills through debugging exercises and logical reasoning tasks.
- Gain hands-on experience in importing, cleaning, and transforming datasets using Python libraries.
- Apply Python for statistical analysis, hypothesis testing, and visualization of trends in real-world datasets.
- Implement basic machine learning workflows to understand model training and evaluation.

Tools and software used for python programming

There are several tools and software commonly used for Python programming, depending on the development needs. Here are some essential ones:

Integrated Development Environments (IDEs)

- PyCharm – A powerful IDE with advanced debugging and code analysis features.

- Visual Studio Code (VS Code) – A lightweight, customizable editor with Python extensions.
- Jupyter Notebook – Ideal for data science and interactive coding.
- Spyder – Designed for scientific computing and data analysis.
- Google Colab – A cloud-based Jupyter-like environment ideal for collaborative data science work without local setup.
- RStudio (with Python integration) – Useful for students working on hybrid R–Python data projects.

Package Management Tools

- pip – The default package manager for installing Python libraries.
- conda – A package and environment manager, useful for data science projects.

Version Control Systems

- Git – Used for tracking code changes and collaborating on projects.
- GitHub/GitLab/Bitbucket – Platforms for hosting and managing repositories.

Testing & Debugging Tools

- pytest – A framework for writing and running tests.
- pdb – Python's built-in debugger for troubleshooting code.

Virtual Environment Tools

- venv – A built-in tool for creating isolated Python environments.
- virtualenv – An alternative for managing dependencies separately.

Data Science & Machine Learning Libraries

- NumPy, Pandas, Matplotlib – Essential for data analysis and visualization.
- TensorFlow, PyTorch, Scikit-learn – Used for AI and machine learning applications.
- Seaborn – For advanced statistical data visualization.
- Statsmodels – For econometrics and statistical modeling.
- Plotly – For interactive dashboards and data visualizations.

Web Development Frameworks

- Flask – A lightweight framework for building web applications.
- Django – A full-featured framework for scalable web development.

Scope of the Lab

This Python lab serves as a fundamental introduction to programming, equipping students with essential skills for software development, automation, and data processing. Through hands-on exercises, students will gain a strong foundation in Python syntax, data structures, and control flow. They will learn to write efficient and structured code using loops, conditionals, and functions to solve computational problems effectively. Additionally, the lab emphasizes practical applications by guiding students in handling file input and output operations, working with built-in libraries, and managing external modules. This lab also prepares students to use Python for extracting insights from structured and unstructured data, performing statistical analysis, and creating effective visualizations to communicate findings. Students will learn to integrate Python with tools such as SQL databases, Excel, and APIs to work with real-world datasets.

Expectations from Students

1. Students must actively participate in all lab sessions by engaging in coding exercises, discussions, and practical applications to deepen their understanding of Python. They should come prepared by reviewing the lab manual and any assigned materials beforehand to approach tasks with confidence.

2. Consistency in completing and submitting lab assignments on time is essential for reinforcing programming skills and building technical proficiency.

3. Collaboration with peers is encouraged, as working on group projects and coding challenges fosters teamwork and enhances problem-solving abilities.

4. Students should take the initiative to explore beyond the structured exercises, ask questions, and apply their learning to real-world applications.

5. Maintaining discipline and focus throughout the lab sessions will ensure maximum learning and retention of concepts.

6. They should demonstrate curiosity and engagement by experimenting with Python's capabilities and integrating their knowledge into practical coding projects.

7. By fulfilling these expectations, students will develop a strong foundation in Python programming that will be valuable for future studies and professional endeavors.

8. Students are encouraged to bring their own datasets or find publicly available ones to explore during lab sessions. They should also document their analysis process and share visual reports or dashboards as part of project submissions.

Lab Guidelines

1. Read and understand the lab manual, experiment guidelines, and instructor directions before starting any task.

2. Keep the workspace clean and organized to prevent any interference during practical exercises.

3. Use only the approved software and tools specified by the instructor or lab administrator.

4. Ensure that all lab exercises and coding assignments are completed and submitted within the given deadlines.

5. Write original code and follow ethical programming practices, avoiding plagiarism.

6. Participate actively in discussions and collaborative activities to improve understanding and problem-solving skills.

7. Seek help from the instructor or peers when facing challenges to clarify doubts and enhance learning.

8. Regularly save and back up coding projects to prevent data loss due to unexpected errors or system failures.

9. Explore beyond the basic exercises by experimenting with new concepts and applying Python to real-world scenarios.

10. Submit lab assignments and reports within the specified deadlines to avoid penalties.

11. Ensure all submitted work is original, avoiding plagiarism or copying from others to maintain academic integrity.

12. Prepare for viva or oral exams to effectively demonstrate understanding of the experiments conducted.

13. Non-compliance with lab guidelines, misuse of lab equipment, or engaging in unethical practices will result in disciplinary measures, which may include:
    - Warnings.
    - Deduction of grades.
    - Suspension from lab sessions.

14. All data used in lab projects should comply with data privacy laws and ethical guidelines. Students must avoid using sensitive or personal information unless explicit permission is granted and anonymization measures are applied.

Code of Conduct

- Be respectful to instructors, lab assistants, and peers.

- Always seek help when required, but do not disrupt others' work.
- Strive to develop problem-solving skills and explore advanced features of DBMS tools.
- Respect the integrity of datasets and avoid misrepresenting results.
- Clearly label visualizations, cite sources, and maintain transparency in analytical methods.

Steps to Install and Configure Python

Installing and configuring Python

Step 1: Download Python

1. Go to the official Python website: https://www.python.org.
2. Navigate to the Downloads section and choose the appropriate version for your operating system (Windows, macOS, or Linux).
3. Click the download link for the latest stable release.

Step 2: Install Python

For Windows:

4. Run the downloaded .exe file.
5. Check "Add Python to PATH" (important for command-line usage).
6. Click Install Now and wait for the installation to complete.
7. Verify installation by opening Command Prompt (cmd) and typing:

python --version

For data science projects, it is recommended to install **Anaconda Distribution**, which comes with Python, Jupyter Notebook, and essential data science libraries pre-installed. This avoids manual library installations and simplifies environment management.

Some of the IDEs for Python development are:

1. PyCharm

- Developed by JetBrains, PyCharm is a powerful IDE with advanced features like code analysis, debugging, and version control.
- Ideal for professional developers working on large projects.

2. Visual Studio Code (VS Code)

- A lightweight, highly customizable code editor with Python extensions.
- Great for beginners and experienced developers alike.

3. Jupyter Notebook

- Best suited for data science and machine learning projects.
- Allows interactive coding with inline visualization.

4. Spyder

- Designed for scientific computing and data analysis.
- Comes with built-in tools for debugging and variable exploration.

Experiment 1

Title: Setting up Python environment (Anaconda, Jupyter Notebooks), Basic Python syntax review, Writing and running Python scripts

Objectives:

- To learn how to install and use Anaconda for Python programming.
- To understand the basic rules of Python syntax (keywords, identifiers, comments, operators).
- To practice writing and running Python scripts.
- To solve simple problems using Python programs.
- To learn how to take input from the user and display output.

Theory

1.1.1 Steps to Install Anaconda Distribution

Step 1: Download Anaconda

1. Go to the official Anaconda website: https://www.anaconda.com/download.
2. Choose your operating system (Windows, macOS, or Linux).
3. Select the **Python 3.x** version (always download the latest stable release).
4. Click **Download** to get the installer file.

Step 2: Install Anaconda (Windows Example)

1. Locate the downloaded .exe file and double-click to start the installer.
2. In the setup window:
   - » Click **Next** to proceed.
   - » Accept the license agreement.
   - » Choose **Just Me** (recommended) or **All Users** depending on your needs.

3. Select the installation location (default is fine).
4. Tick the box **"Add Anaconda to my PATH environment variable"**..
5. Click **Install** and wait for the process to complete.
6. When finished, click **Finish**.

Step 3: Verify Installation

- Open the **Anaconda Navigator** from the Start Menu.
- Alternatively, open **Anaconda Prompt** and type:

```
conda --version
```

You should see the installed version number.

Step 4: Using Anaconda for Data Science

- Launch **Jupyter Notebook** from Anaconda Navigator to start coding interactively.
- Use **Spyder** for scientific computing.
- Manage environments and install libraries with:

```
conda create --name myenv python=3.10
conda activate myenv
conda install pandas numpy matplotlib seaborn
```

Step 5: Updating Anaconda

Run in **Anaconda Prompt**:

```
conda update conda
conda update anaconda
```

Step 6: Verify installation:

Open Anaconda Prompt

Type:

```
conda --version
python --version
```

Launch Jupyter Notebook

1. Open Anaconda Navigator.
2. Launch Jupyter Notebook.
3. Create a new notebook: File → New Notebook → Python 3.
4. Type : print("Hello, Python!")

5. Press Shift + Enter to execute.

Write and Run a Python Script

1. Open a text editor (Notepad/VS Code).
2. Write: # simple_script.py

   print("Welcome to MCA Python Lab")
3. Save as simple_script.py.
4. Open terminal/command prompt, navigate to the file location, and run the script.

python simple_script.py

1.1.2 Introduction to Basic Python Syntax

Python is a high-level, interpreted programming language known for its simple and readable syntax. Unlike many programming languages that use braces {} to define code blocks, Python uses indentation (spaces or tabs). This makes the code clean and structured.

Table 1.1.1 : Basic Syntax Elements in Python

Concept	Description	Examples
Keywords	Keywords in Python are reserved words that have a predefined meaning in the language. They are part of Python's syntax and cannot be used as identifiers (like variable names, function names, or class names).	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
Identifiers	An identifier is the name given to entities such as variables, functions, classes, objects, or modules in	name = "Alice" age = 22 student_marks = 85 rollNumber1 = 101

Reading input from the user and display the output	input()- It is used to read the data
	print()- To display the data
		x=int(Input()) //Integer
		f=float(Input()) //Float
		s=Input() //String
		a=10// Initialize value directly.

Operators	♦ Arithmetic Operators: +, -, *, /, %, ** (exponentiation), // (floor division). ♦ Relational (Comparison) Operators: ==, !=, >, <, >=, <= (return boolean values). ♦ Logical Operators: and, or, not (used to combine conditional statements). ♦ Assignment Operators: =, +=, -=, *=, /=, %=, **=, //=. ♦ Bitwise Operators: & (AND), | (OR), ^ (XOR), ~ (NOT), << (left shift), >> (right shift). ♦ Identity Operators: is, is not (check if two variables refer to the same object in memory). ♦ Membership Operators: in, not in (check if a value is present in a sequence).	Arithmetic operators 5 + 3 → 8 10 - 4 → 6 6 * 2 → 12 8 / 2 → 4.0 10 % 3 → 1 2 ** 3 → 8 10 // 3 → 3 Relational (Comparison) Operators ● 5 == 5 → True ● 7 != 3 → True ● 10 > 2 → True ● 4 < 2 → False ● 6 >= 6 → True ● 8 <= 5 → False Logical Operators ♦ (5 > 2) and (3 < 4) → True ♦ (7 < 2) or (4 == 4) → True ♦ not (5 == 5) → False

		Assignment Operators ♦ x = 5 ♦ x += 2 → x = 7 ♦ x -= 3 → x = 4 **Bitwise Operators** ♦ 5 & 3 → 1 ♦ 5 | 3 → 7 ♦ 5 ^ 3 → 6 **Identity Operators** ♦ x = [1, 2] ♦ y = x ♦ x is y → True ♦ x is not y → False **Membership Operators** ♦ 3 in [1, 2, 3] → True ♦ 5 not in [1, 2, 3] → True

Character Set	Letters → A–Z, a–z
	Digits → 0–9
	Special Symbols → + - * / % = < > ( ) [ ] { } @ # , . : ‘ “
	Whitespace → space, tab (\t), newline (\n)
	Escape Sequences → \n (new line), \t (tab), \” (double quote), \\ (backslash)
		x = “Hello\nWorld” # escape sequence
		y = 25 # digits

Comments	Single-line comment → starts with # Multi-line comment → written with ‘’’ or “””	# This is a comment ‘’’ This is a multi-line comment ‘’’

1.1.2.1 operator precedence and associativity

Precedence	Operator(s)	Description
1 (Highest)	()	Parentheses (grouping)
2	**	Exponentiation
3	+x, -x, ~x	Unary plus, minus, bitwise NOT
4	*, /, //, %	Multiplication, Division, Floor Div, Mod
5	+, -	Addition, Subtraction
6	<<, >>	Bitwise Shift
7	&	Bitwise AND
8	^	Bitwise XOR
9	==, !=, >, <, >=, <=	Comparison Operators
10	not	Logical NOT
11	and	Logical AND
12	or	Logical OR
14	if else	Conditional expressions
15	=, +=, -=, etc.	Assignment operators

1.1.3 Writing and running Python scripts

Writing and running Python scripts is the process of creating a sequence of Python instructions in a text file (usually saved with the .py extension) and then executing them to perform a specific task or solve a problem. Unlike running code interactively in the Python shell or Jupyter Notebook, scripts allow you to store code permanently, reuse it, and execute it as a standalone program. To write a Python script, you typically use a text editor or an Integrated Development Environment (IDE) such as VS Code, PyCharm, or even simple editors like Notepad, where you type Python statements such as variable definitions, functions, loops, and logic. Once the script is saved, it can be run by opening a command line or terminal, navigating to the folder containing the file, and using the command python filename.py (or python3 filename.py depending

on the system). When executed, the Python interpreter reads the script line by line, converts it into machine-understandable instructions, and produces the desired output or performs the intended action. Scripts are especially useful for automating repetitive tasks, analyzing data, building applications, or running experiments, and they can also accept user input, import modules, and handle errors, making them powerful building blocks for both simple programs and large-scale projects.

1.1.4 Python sample questions

1. Write a program to perform all arithmetic operations on two numbers entered by the user.

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)
print("Floor Division:", a // b)
print("Modulus:", a % b)
print("Exponentiation:", a ** b)
```

Output

```
Enter first number: 40
Enter second number: 20
Addition: 60
Subtraction: 20
Multiplication: 800
Division: 2.0
Floor Division: 2
Modulus: 0
Exponentiation: 1099511627776000000000000000000
```

2. Write a program to find the area and perimeter of a rectangle.

```
length = float(input("Enter the length of the rectangle: "))
breadth = float(input("Enter the breadth of the rectangle: "))
```

```
area = length * breadth

perimeter = 2 * (length + breadth)

print("Area of the rectangle:", area)

print("Perimeter of the rectangle:", perimeter)
```

Output

```
Enter the length of the rectangle: 3

Enter the breadth of the rectangle: 2

Area of the rectangle: 6.0

Perimeter of the rectangle: 10.0
```

3. Write a Python program to convert a given temperature in Celsius to Fahrenheit and display both the Celsius and Fahrenheit values.

```
celsius = float(input("Enter temperature in Celsius: "))

# Process: Convert Celsius to Fahrenheit

fahrenheit = (celsius * 9/5) + 32

# Output: Display result

print("Celsius Temperature:", celsius)

print("Fahrenheit Temperature:", fahrenheit)
```

Output

```
Enter temperature in Celsius: 25
Celsius Temperature: 25.0
Fahrenheit Temperature: 77.0
```

4. Write a Python Program to Swap Two Variables

```
# Python program to swap two variables

# To take inputs from the user

x = input('Enter value of x: ')

y = input('Enter value of y: ')

# create a temporary variable and swap the values

temp = x
```

```
x = y

y = temp

print('The value of x after swapping: {}'.format(x))

print('The value of y after swapping: {}'.format(y))
```

Output

```
Enter value of x: 70

Enter value of y: 77

The value of x after swapping: 77

The value of y after swapping: 70
```

1.1.5 Lab Practice Questions

1. Write a program to compute the simple interest (SI = P × R × T / 100).
2. Write a program to find the area and perimeter of a square.
3. Write a python program to read a number and print the cube of a given number.
4. Write a Python program to display your name, age, and city.
5. Write a Python program to find the average of three numbers.
6. Write a Python program to calculate the **area** and **circumference** of a circle .

Experiment 2

Title: Operators and String Handling in Python

Objectives

- To gain knowledge of the different categories of operators provided in Python.
- To practice the use of arithmetic, relational, logical, assignment, bitwise, and identity operators.
- To write Python programs that demonstrate the working of various operators.
- To explore and implement different string handling operations and built-in string methods in Python.
- To write programs that efficiently perform manipulation of string data.

Theory

1.2.1 Operators in Python

Operators in Python are special symbols or keywords that are used to perform operations on variables and values. They allow programmers to carry out computations, comparisons, logical decisions, and data manipulations efficiently. Python supports a wide variety of operators such as arithmetic, relational, logical, assignment, bitwise, identity, and membership operators (Table 1.2.1), each serving a specific purpose in program execution. Mastering operators is essential, as they form the foundation of writing expressions and implementing logic in Python programs.

Table 1.2.1 Types of Operators in Python

Operator Type	Operators	Description
Arithmetic Operators	+, -, *, /, %, ** (exponentiation), // (floor division)	Used to perform mathematical calculations such as addition, subtraction, etc.
Relational Operators	==, !=, >, <, >=, <=	Used to compare values and return Boolean results (True or False).
Logical Operators	and, or, not	Used to combine or modify conditional statements.
Assignment Operators	=, +=, -=, *=, /=, %=, **=, //=	Used to assign values to variables and update them with operations.
Bitwise Operators	& (AND), `	(OR),^(XOR),~(NOT),<<(left shift),>>` (right shift)
Identity Operators	is, is not	Check whether two variables refer to the same object in memory.
Membership Operators	in, not in	Check whether a value is present in a sequence (e.g., string, list, tuple).

1.2.1.1 Operator precedence and Associativity

In Python, when an expression contains multiple operators, the order in which the operations are performed is determined by **operator precedence**. Operators with higher precedence are evaluated first, while those with lower precedence are evaluated later. For example, multiplication has higher precedence than addition, so in the expression `2 + 3 * 4`, multiplication is performed before addition, resulting in `14`. When two operators of the same precedence appear in an expression, **associativity** rules decide the order of evaluation. Most operators in Python follow **left-to-right associativity**, meaning they are evaluated from left to right, while a few like exponentiation (`**`) follow

right-to-left associativity (Table 1.2.2). Understanding precedence and associativity is essential to avoid ambiguity and ensure accurate results in expressions.

Table 1.2.2 Operator precedence and Associativity

Precedence Level	Operator(s)	Description	Associativity
1 (Highest)	()	Parentheses (grouping)	Left to Right
2	**	Exponentiation	Right to Left
3	+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT	Right to Left
4	*, /, //, %	Multiplication, Division, Floor division, Modulo	Left to Right
5	+, -	Addition, Subtraction	Left to Right
6	<<, >>	Bitwise shift operators	Left to Right
7	&	Bitwise AND	Left to Right
8	^	Bitwise XOR	Left to Right
9	`	Bitwise OR	Left to Right
10	==, !=, >, <, >=, <=	Comparison (Relational) operators	Left to Right
11	is, is not, in, not in	Identity and Membership operators	Left to Right
12	not	Logical NOT	Right to Left
13	and	Logical AND	Left to Right
14	or	Logical OR	Left to Right
15 (Lowest)	=, +=, -=, *=, /=, %=, etc.	Assignment operators	Right-to-Left

1.2.2 Reading input from the user

In Python, programs often need to interact with users by accepting input data during execution. Reading input from the user allows a program to become dynamic, responding to different values provided at runtime rather than using fixed data. Python provides the built-in `input()` function to accept input from the keyboard as a string, which can then be converted into other data types like integers or floats as needed. Understanding how to read and process user input is fundamental for creating interactive programs

and performing real-time computations. **Commonly used functions and techniques related to reading input from the user in Python** are listed below (Table 1.2.3).

Table 1.2.3 **Functions related to reading input from the user in Python**

Function / Method	Description	Example
input(prompt)	Reads input from the user as a string. The optional prompt is displayed before input.	name = input("Enter your name: ")
int()	Converts a string input into an integer.	age = int(input("Enter your age: "))
float()	Converts a string input into a floating-point number.	height = float(input("Enter your height: "))
str()	Converts input or other data types to a string (if needed).	num_str = str(123)
eval()	Evaluates a string as a Python expression (can read numbers, lists, etc.).	result = eval(input("Enter a number or expression: "))
split()	Used to split multiple inputs given in a single line (after using input()).	a, b = input("Enter two numbers: ").split()
map()	Converts multiple inputs to a specific data type (like int) after splitting.	x, y = map(int, input("Enter two integers: ").split())

Exercise Questions

Exercise 1: Write a Python program to input two numbers and display the results of all arithmetic operators.

Source code

```
# Input two numbers
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
# Arithmetic operations
print("Addition:", a + b)
```

```
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)
print("Floor Division:", a // b)
print("Modulus:", a % b)
print("Exponentiation:", a ** b)
```

Output

```
Enter first number: 5
Enter second number: 2
Addition: 7
Subtraction: 3
Multiplication: 10
Division: 2.5
Floor Division: 2
Modulus: 1
Exponentiation: 25
```

Exercise 2: Write a program to find the area and perimeter of a rectangle.

Source code

```
# Input the length of a side
side = float(input("Enter the length of the side of the square: "))
# Calculate area and perimeter
area = side * side
perimeter = 4 * side
# Display the results
print("Area of the square:", area)
print("Perimeter of the square:", perimeter)
```

Output

```
Enter the length of the side of the square: 5
Area of the square: 25.0
```

Perimeter of the square: 20.0

Exercise 3: Write a program to check whether a given number is greater than, less than, or equal to another number using relational operators.

Source code

 # Input two numbers

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

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

 # Compare the numbers

if a > b:

    print(a, "is greater than", b)

elif a < b:

    print(a, "is less than", b)

else:

    print(a, "is equal to", b)

Output

Enter the first number: 7

Enter the second number: 10

7 is less than 10

1.2.3 Strings in python

In Python, a **string** is a sequence of characters used to represent text. It is one of the most important and commonly used data types in programming. Strings can include letters, numbers, symbols, and even spaces, and are always enclosed within single quotes (' '), double quotes (" "), or triple quotes (''' ''' or """ """). For example, "Python", 'Hello World', and '''Multi-line string''' are valid string objects. Since strings are **immutable**, once created, their contents cannot be changed, but Python provides various operations and built-in methods to manipulate and process them efficiently.

1.2.3.1 Common String Operations and Methods

Strings are one of the most frequently used data types in Python, and working with them efficiently is essential in programming. To make string handling easier, Python provides a variety of operations such as concatenation, repetition, membership testing, indexing, and slicing. Along with these, it also offers many built-in methods like lower(), upper(), replace(), split(), join(), and find() that allow programmers to manipulate, search, and transform strings quickly. These operations and methods together make

string processing simple, flexible, and powerful. Some common string operations and methods in Python are listed below (Table1.2.4).

Table 1.2.4 Common string operations and methods in Python.

Operation / Method	Description	Example	Output
Concatenation (+)	Joins two or more strings together	"Hello" + " World"	Hello World
Repetition (*)	Repeats the string multiple times	"Hi! " * 3	Hi! Hi! Hi!
Slicing ([:])	Extracts a part of the string using index range	"Python"[0:4]	Pyth
len()	Returns the length of the string	len("Python")	6
upper()	Converts string to uppercase	"python".upper()	PYTHON
lower()	Converts string to lowercase	"PYTHON".lower()	python
capitalize()	Converts first character to uppercase and rest to lowercase	"hello world".capitalize()	Hello world
islower()	Checks if all characters are in lowercase	"python".islower()	True
isupper()	Checks if all characters are in uppercase	"PYTHON".isupper()	True
split()	Splits string into a list of words	"a b c".split()	['a', 'b', 'c']

Exercise Questions

Exercise 1: Write a Python program to concatenate two strings `"Hello"` and `"World"`, and then repeat the result three times.

Source code

```
str1 = “Hello”

str2 = “World”

result = (str1 + “ “ + str2) * 3

print(result)
```

Output

Hello WorldHello WorldHello World

Exercise 2: Write a Python program to slice the first four characters from the string "Programming", convert it to uppercase, and then convert the whole string to lowercase.

Source code

```
text = "Programming"
print(text[0:4].upper())   # slicing + uppercase
print(text.lower())        # lowercase
```

Output

PROG

Programming

Exercise 3: Write a Python program to perform the following on the string "welcome to python LAB".

Find its length

Capitalize it

Check if it is lowercase

Check if it is uppercase

Split it into words

Source code

```
s = "welcome to python LAB"
print("Length:", len(s))
print("Capitalize:", s.capitalize())
print("Is Lower:", s.islower())
print("Is Upper:", s.isupper())
print("Split:", s.split())
```

Output

Length: 22

Capitalize: Welcome to python lab

Is Lower: False

Is Upper: False

Split: [‘welcome’, ‘to’, ‘python’, ‘LAB’]

Lab Practice Questions

1. Write a program to find the area and perimeter of a rectangle.
2. Write a program to compute the simple interest (SI = P × R × T / 100).
3. Write a python program to read a number and print the cube of a given number.
4. Write a Python program to concatenate two strings entered by the user and display the result.
5. Write a Python program to input a string from the user and:
   - ♦ Convert it to uppercase
   - ♦ Convert it to lowercase
   - ♦ Capitalize it
6. Given the string “Data Science Lab”, write a program to:
   - ♦ Find its length using len()
   - ♦ Check whether it is in uppercase using isupper()
   - ♦ Split the string into separate words using split()

Experiment 3

Title: Conditional Statements in Python

Objectives

By the end of this experiment, you will be able to:

1. Identify and explain different types of control and looping statements used in Python programming.
2. Apply conditional and iterative statements to solve simple computational problems.
3. Develop Python programs using decision-making and looping constructs to implement real-world logic effectively.

Theory

1.3.1 Conditional Statements

Control and looping statements in Python are fundamental constructs that allow

programmers to make decisions and perform repetitive tasks efficiently. Conditional statements such as **if**, **if-else**, **nested if**, and **if-elif-else ladder** enable a program to execute specific blocks of code based on given conditions, making the program dynamic and logical. Looping statements like **while**, **for**, and simulated **do-while** loops help in executing a set of instructions repeatedly until a particular condition is met. These constructs reduce code redundancy, improve readability, and are essential for implementing decision-making, automation, and iterative operations in Python programs.

1.3.2 Python Example questions

Control Structures: if, if-else, nested if, if-elif-else ladder, switch-case, while, for, and do-while

1. The if Statement

The if statement is used to execute a block of code only when a specific condition is true.

Example 1: Check whether a number is positive

Code:

```
num = int(input("Enter a number: "))
if num > 0:
    print("The number is positive.")
```

Input: 8

Output: The number is positive.

Explanation: The condition num > 0 is true for 8, so the message is displayed.

Example 2: Check if a person is an adult

Code:

```
age = int(input("Enter your age: "))
if age >= 18:
    print("You are an adult.")
```

Input: 22

Output: You are an adult.

Explanation: The condition age >= 18 evaluates to true, so the block executes.

Example 3: Check if a letter is uppercase

Code:

```
ch = input("Enter a character: ")
```

```
if ch.isupper():
    print("The character is uppercase.")
```

Input: G

Output: The character is uppercase.

Explanation: isupper() returns True for uppercase letters.

2. The if-else Statement

The if-else statement lets you execute one block if the condition is true and another if it's false.

Example 1: Check if a number is even or odd

Code:

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

Input: 9

Output: Odd number

Explanation: The number 9 is not divisible by 2, so the else part runs.

Example 2: Check pass or fail

Code:

```
mark = int(input("Enter your mark: "))
if mark >= 50:
    print("Pass")
else:
    print("Fail")
```

Input: 45

Output: Fail

Explanation: Marks below 50 trigger the else block.

Example 3: Compare two numbers

Code:

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

```
b = int(input("Enter second number: "))
if a == b:
    print("Both numbers are equal")
else:
    print("Numbers are different")
```

Input: 5, 9

Output: Numbers are different

Explanation: Since 5 ≠ 9, the else part executes.

3. The Nested if Statement

A nested if is when one if statement is placed inside another to check multiple conditions.

Example 1: Find the largest of three numbers

Code:

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

Input: 7, 9, 5

Output: Largest: 9

Explanation: The inner if conditions compare the values step by step.

Example 2: Classify number as positive, negative, or zero

Code:

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

```
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive")
else:
    print("Negative")
```

Input: -3

Output: Negative

Explanation: Outer if fails, so it executes the else part.

Example 3: Check eligibility for driving

Code:

```
age = int(input("Enter age: "))
if age >= 18:
    license = input("Do you have a license (yes/no)? ").lower()
    if license == "yes":
        print("You can drive.")
    else:
        print("You must get a license first.")
else:
    print("You are too young to drive.")
```

Input: 20, no

Output: You must get a license first.

Explanation: Both conditions are checked in nested form.

4. The if-elif-else Ladder

This structure is used when multiple conditions need to be checked in sequence.

Example 1: Grade calculation

Code:

```
mark = int(input("Enter marks: "))
if mark >= 90:
    print("Grade A")
```

```
elif mark >= 75:
    print("Grade B")
elif mark >= 50:
    print("Grade C")
else:
    print("Fail")
```

Input: 82

Output: Grade B

Explanation: The second condition mark >= 75 is true.

Example 2: Temperature indication message

Code:

```
temp = int(input("Enter temperature: "))
if temp > 40:
    print("Very Hot")
elif temp > 30:
    print("Hot")
elif temp > 20:
    print("Warm")
else:
    print("Cold")
```

Input: 18

Output: Cold

Explanation: None of the above are true, so else executes.

Example 3: Identify weekday

Code:

```
day = int(input("Enter day number (1–7): "))
if day == 1:
    print("Sunday")
elif day == 2:
    print("Monday")
elif day == 3:
```

```
    print("Tuesday")
elif day == 4:
    print("Wednesday")
elif day == 5:
    print("Thursday")
elif day == 6:
    print("Friday")
elif day == 7:
    print("Saturday")
else:
    print("Invalid day number")
```

Input: 5

Output: Thursday

5. The switch-case (match-case statement in Python)

The match-case statement in Python acts like a switch in other languages.

Example 1: Display month name

Code:

```
month = int(input("Enter month number (1-12): "))
match month:
    case 1:
        print("January")
    case 5:
        print("May")
    case 12:
        print("December")
    case _:
        print("Invalid month number")
```

Input: 12

Output: December

Example 2: Simple calculator

Code:

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

```
b = int(input("Enter second number: "))
op = input("Enter operator (+, -, *, /): ")
match op:
    case '+':
        print("Result:", a + b)
    case '-':
        print("Result:", a - b)
    case '*':
        print("Result:", a * b)
    case '/':
        print("Result:", a / b)
    case _:
        print("Invalid operator")
```

Input: 8, 4, *

Output: Result: 32

Example 3: Traffic signal message

Code:

```
signal = input("Enter signal color: ").lower()
match signal:
    case 'red':
        print("Stop")
    case 'yellow':
        print("Get Ready")
    case 'green':
        print("Go")
    case _:
        print("Invalid color")
```

Input: green

Output: Go

Looping Statements

6. The while Loop

A while loop repeats a block of code as long as a given condition remains true.

Example 1: Print first 5 natural numbers

Code:

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

Output:

1

2

3

4

5

Example 2: Sum of digits

Code:

```
num = int(input("Enter a number: "))
sum = 0
while num > 0:
    sum += num % 10
    num //= 10
print("Sum of digits:", sum)
```

Input: 1234

Output: Sum of digits: 10

Example 3: Factorial of a number

Code:

```
num = int(input("Enter a number: "))
fact = 1
while num > 0:
    fact *= num
    num -= 1
print("Factorial:", fact)
```

Input: 5

Output: Factorial: 120

7. The for Loop

The for loop iterates over a sequence (like a list, string, or range).

Example 1: Multiplication table

Code:

```
n = int(input("Enter a number: "))
for i in range(1, 6):
    print(f"{n} x {i} = {n*i}")
```

Input: 4

Output:

4 x 1 = 4

4 x 2 = 8

4 x 3 = 12

4 x 4 = 16

4 x 5 = 20

Example 2: Print each letter of a word

Code:

```
word = input("Enter a word: ")
for ch in word:
    print(ch)
```

Input: Code

Output:

C

o

d

e

Example 3: Find the sum of a list

Code:

```
nums = [2, 4, 6, 8, 10]
total = 0
for n in nums:
    total += n
print("Sum =", total)
```

Output: Sum = 30

8. The do-while Equivalent in Python

Python doesn't have a do-while loop, but the same logic can be implemented using while True and break.

Example 1: Menu-driven calculator

Code:

```
while True:
    print("\n1. Add\n2. Subtract\n3. Exit")
    ch = int(input("Enter your choice: "))
    if ch == 3:
        break
    a = int(input("Enter first number: "))
    b = int(input("Enter second number: "))
    if ch == 1:
        print("Sum =", a + b)
    elif ch == 2:
        print("Difference =", a - b)
```

Sample Input/Output:

1

10

20

Sum = 30

3

Example 2: Repeat until valid password

Code:

```
while True:
    password = input("Enter password: ")
    if password == "python123":
        print("Access Granted")
        break
    else:
        print("Wrong password! Try again.")
```

Input:

abc123 → Wrong password! Try again.

python123 → Access Granted

Example 3: Continue user input until they type stop

Code:

```
while True:
    word = input("Enter a word (type 'stop' to exit): ").lower()
    if word == 'stop':
        print("Program ended.")
        break
    print("You entered:", word)
```

Input:

hello

Output:

You entered: hello

stop

Program ended.

Nested Loop

Example 1: Program to print an isosceles triangle pattern using stars (*)

```
# Read number of rows from the user
rows = int(input("Enter the number of rows for the triangle: "))
# Outer loop for rows
for i in range(1, rows + 1):
    # Inner loop for spaces
    for j in range(rows - i):
        print(" ", end="")
    # Inner loop for stars
    for k in range(2 * i - 1):
        print("*", end="")
    # Move to next line
    print()
```

Input:

Enter the number of rows for the triangle: 6

Output:

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

Lab Practice Questions

1. Implement a Python program to check whether a given year is a leap year or not using an if-else statement.
2. Write a program to determine whether a given number is divisible by both 3 and 5 using an if statement.
3. Develop a program to find the largest among four numbers using a nested if structure.
4. Implement a program to check whether a character entered by the user is a vowel or a consonant using an if-elif-else ladder.
5. Write a Python program that accepts a number and checks whether it lies between 1 and 100 using conditional statements.
6. Create a menu-driven program using match-case that performs addition, subtraction, multiplication, and division based on user choice.
7. Write a program using a match-case structure to display the type of shape based on the number of sides (e.g., 3 → Triangle, 4 → Quadrilateral, etc.).

Looping Statements

8. Write a Python program using a while loop to print all even numbers between 1 and 50.
9. Implement a program using a for loop to calculate the factorial of a given number.
10. Develop a program using a while loop to reverse a given number.
11. Write a program using a for loop to count the number of vowels in a given string.
12. Implement a Python program using a for loop to generate the Fibonacci series up to n terms.
13. Write a program using a while loop that keeps accepting numbers from the user until a negative number is entered.
14. Create a Python program using a do-while equivalent loop to repeatedly ask for a username until the user enters “admin”.
15. Implement a nested loop program to print a right-angled triangle pattern using stars (*).

Experiment 4

Title:To demonstrate different comprehensions - List, Set and Generator Comprehensions

Objectives:

- To practice writing list, set and generator comprehensions.
- To learn how to add, update, and remove elements in Python collections.
- To explore the unique features and applications of each type of collection.

To develop problem-solving skills by applying collections in real-life examples and programs.

Theory

1.4.1 List, Tuple, Set, Dictionary and Generator

In Python, **collections** are used to store and manage groups of data. They help in organizing, accessing, and processing data efficiently. The main built-in collection types are list, tuple, set, dictionary and generator.

Table 1.4.1: Comparison of List, Tuple, Set, Dictionary, and Generator in Python

List	A **list** is a **mutable, ordered collect-ion** of elements in Python. Lists can hold **different data types** (numbers, strings, objects, etc.). Lists are writt-en using **square brackets [ ]**.	numbers = [10, 20, 30, 40]
		mixed = [1, "Alice", 3.5, True]
		# Accessing elements: numbers[0] # first element numbers[-1] # last element
		#Updating elements:numbers[1] = 25
		#Adding elements:numbers.append(50)
		numbers.insert(2, 60)
		#Deleting elements:numbers.remove(25)
		numbers.pop() del numbers[0]
		#Other methods: sort() → Sorts list in ascending order reverse() → Reverses the list len() → Returns the number of elements

Tuple	A **tuple** is an **ordered, immutable collection** of elements in Python. Unlike lists, tuples **cannot be modified** (no insertion, deletion, or updating of elements). Tuples are written using **parentheses ( )**.	numbers = (10, 20, 30, 40) mixed = (1, "Alice", 3.5, True) single = (5,) # single-element tuple #Accessing elements: numbers[0] # first element numbers[-1] # last element len(tuple) → Number of elements max(tuple) → Maximum value min(tuple) → Minimum value sum(tuple) → Sum of elements (if numeric) tuple(list) → Convert list to tuple #Slicing: print(numbers[1:3])

Set	A set is an unordered collection of unique elements in Python. Duplicates are not allowed. Sets are defined using curly braces { } or the set() function.	my_set = {1, 2, 3, 4, 5}
		print(my_set) # Output: {1, 2, 3, 4, 5}
		A = {1, 2, 3, 4}
		B = {3, 4, 5, 6}
		print(A.union(B)) # {1, 2, 3, 4, 5, 6}
		print(A.intersection(B)) # {3, 4}
		print(A.difference(B)) # {1, 2}
		print(B.difference(A)) # {5, 6}
		s = {10, 20, 30}
		s.add(40) # Add single item
		print(s) # {40, 10, 20, 30}
		s.update([50, 60]) # Add multiple items
		print(s) # {40, 10, 50, 20, 60, 30}
		s.remove(20) # Remove item (error if not found)
		print(s) # {40, 10, 50, 60, 30}
		s.discard(100) # Remove item (no error if not found)
		print(s) # {40, 10, 50, 60, 30}

Dictionary	♦ A dictionary is a collection of key–value pairs.
	♦ Each key must be unique and immutable (string, number, tuple).
	♦ Values can be of any data type (mutable or immutable).
	♦ Dictionaries are unor-dered, mutable, and indexed by keys.
	♦ Defined using curly braces {} with key: value format.
		# Empty dictionary
		my_dict = {}
		# Dictionary with data
		student = {
		"name": "Anu",
		"age": 21,
		"course": "MCA"
		}
		print(student)
		# Output: {'name': 'Anu', 'age': 21, 'course': 'MCA'}
		print(student["name"]) # Anu
		print(student.get("age")) # 21
		student["grade"] = "A" # Add new key-value pair
		student["age"] = 22 # Update value
		print(student)
		# Output: {'name': 'Anu', 'age': 22, 'course': 'MCA', 'grade': 'A'}
		student.pop("course") # Removes key 'course'
		print(student)
		student.popitem() # Removes last inserted item
		print(student)
		del student["age"] # Deletes key 'age'
		print(student)
		student.clear() # Empties the dictionary
		print(student) # {}
		person = {"name": "Rahul", "age": 25, "city": "Kochi"}
		print(person.keys()) # dict_keys(['name', 'age', 'city'])
		print(person.values()) # dict_values(['Rahul', 25, 'Kochi'])
		print(person.items()) # dict_items([('name', 'Rahul'), ('age', 25), ('city', 'Kochi')])

Generator	A generator is a function that gives values one by one instead of all at once. It uses the yield keyword instead of return. Generators do not store all data in memory → they are memory efficient.	def my_gen(): yield 1 yield 2 yield 3 g = my_gen() print(next(g)) # 1 print(next(g)) # 2 print(next(g)) # 3

1.4.2 List, Set, Dictionary and Generator comprehensions

Table 1.4.2: Comparison of List, Set, Dictionary, and Generator Comprehensions in Python

Type of comprehensions	**Descriptions**	**Syntax & Examples**
List comprehensions	List comprehension is a short and simple way to create lists in Python. Instead of writing long for loops, we can create lists in one line.	Syntax: [expression for item in iterable if condition] squares = [x*x for x in range(6)] print(squares) # [0, 1, 4, 9, 16, 25] even_numbers = [x for x in range(10) if x % 2 == 0] print(even_numbers) # [0, 2, 4, 6, 8]
Python there is no direct tuple comprehension.		
Set Comprehensions	Set comprehension is just like list comprehension, but it creates a set instead of a list.	Syntax: {expression for item in iterable if condition} squares = {x*x for x in range(6)} print(squares) # {0, 1, 4, 9, 16, 25} even_numbers = {x for x in range(10) if x % 2 == 0} print(even_numbers) # {0, 2, 4, 6, 8}

Dictionary comprehension	**Dictionary comprehension** is a short way to create dictionaries. It works similar to list/ set comprehensions but creates **key–value pairs**.	{key_expression: value_expression for item in iterable if condition} even_squares = {x: x*x for x in range(10) if x % 2 == 0} print(even_squares)

Generator Comprehensions	A generator comprehension is similar to list comprehension, but it creates a generator object instead of storing all elements in memory.
	
	It uses parentheses ( ) instead of square brackets [ ].
	
	Generators produce values one by one (lazy evaluation).
		Syntax:
		
		(expression for item in iterable if condition)
		
		gen = (x for x in range(5))
		
		print(gen)
		
		for val in gen:
		
		print(val)
		
		squares = (x*x for x in range(6))
		
		for num in squares:
		
		print(num)

1.4.3 Python sample questions

1. Write a program to Find all of the numbers from 1-1000 that are divisible by 7 (using list comprehension)..

```
div7 = [n for n in range(1,100) if n % 7 == 0]

print(div7)
```

Output

[7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98]

2. Write a program to extract vowels from a string (Using list comprehensions).

```
text = “Python Programming”

vowels = [ch for ch in text if ch.lower() in “aeiou”]

print(“Vowels in the string:”, vowels)
```

Output

Vowels in the string: [‘o’, ‘o’, ‘a’, ‘i’]

3. Write a python program to create a list of even numbers from 1 to 20 using list comprehension.

```
evens = [x for x in range(1, 21) if x % 2 == 0]

print(“Even numbers from 1 to 20:”, evens)
```

Output

Even numbers from 1 to 20: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

4. Write a Python program to remove the odd numbers from a given set of integers and print the new set (using set comprehension).

```
mySet = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

print(“The existing set is:”)

print(mySet)

mySet = {element for element in mySet if element % 2 == 0}

print(“The modified set is:”)

print(mySet)
```

Output

The existing set is:

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

The modified set is:

{2, 4, 6, 8, 10}

5. Write a Python program using set comprehension to generate a set of numbers between 1 and 50 that are divisible by 3 or 5. Display the set.

```
nums = {x for x in range(1, 51) if x % 3 == 0 or x % 5 == 0}

print(“Numbers divisible by 3 or 5:”, nums)
```

Output

Numbers divisible by 3 or 5: {3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24, 25, 27, 30, 33, 35, 36, 39, 40, 42, 45, 48, 50}

6. Write a Python program to extract uppercase letters using generator comprehension

```
text = "GeneratorComprehensionInPython"
uppercase = (ch for ch in text if ch.isupper())
print("Uppercase letters:")
for ch in uppercase:
    print(ch, end=" ")
```

Output

Uppercase letters:

G C I P

7. Program to generate words with length greater than 4 using generator comprehension

```
sentence = "Python generator comp creates ite "
long_words = (word for word in sentence.split() if len(word) > 4)
print("Words longer than 4 letters:")
for w in long_words:
    print(w, end=" ")
```

Output

Words longer than 4 letters:

Python generator creates

1.4.4 Lab Practice Questions

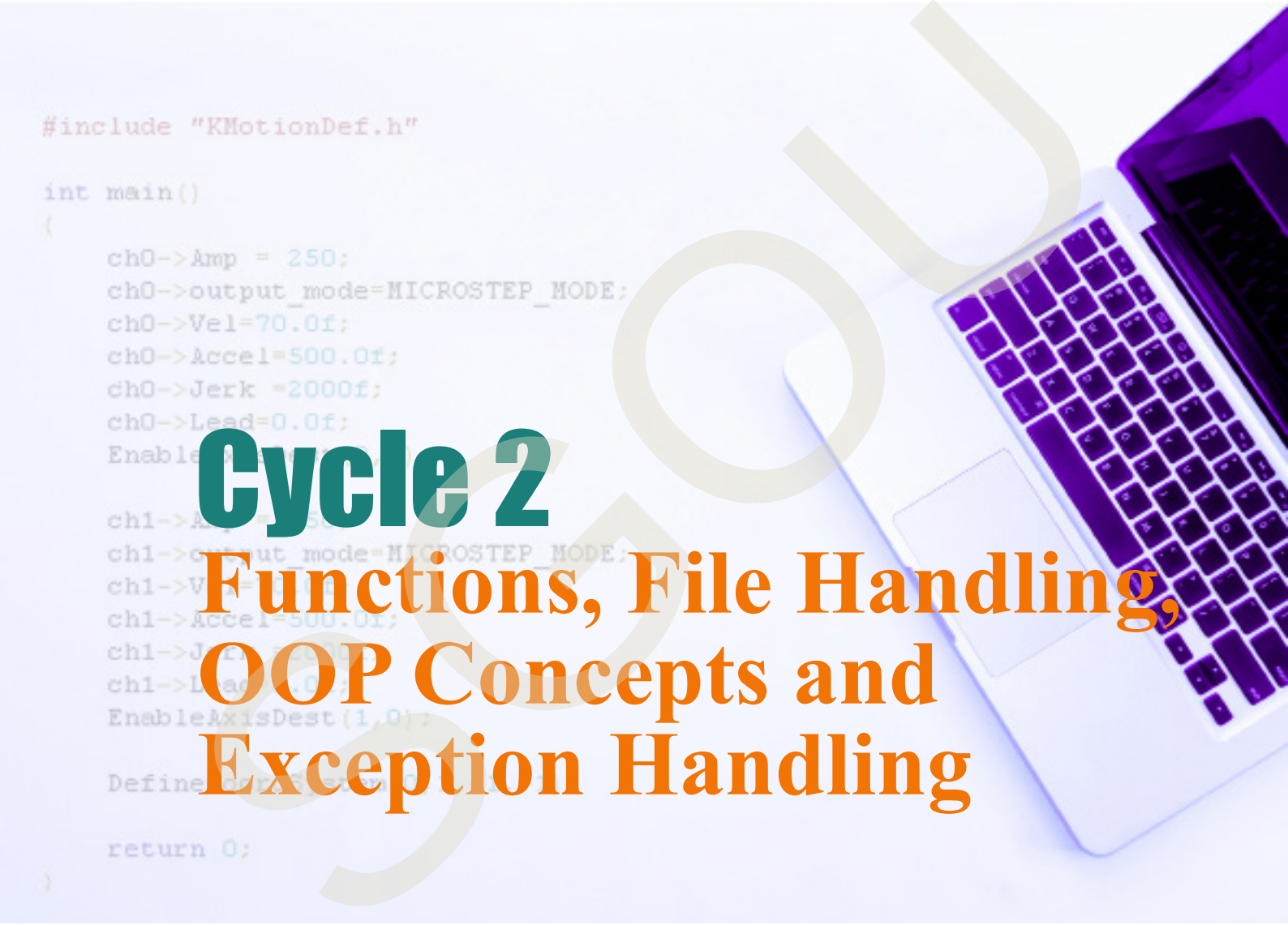
1. Write a python program to extract all uppercase letters from a string using list comprehension.
2. Write a python program using list comprehension to generate a list of all numbers from **1 to 50** that are divisible by **5**.
3. Write a Python program using set comprehension to generate ASCII values of all characters in a given string.
4. Write a Python program using set comprehension to find all words longer than 4 characters in a sentence.

5. Write a Python program using set comprehension to generate all prime numbers between 1 and 50.

6. Write a Python program using generator comprehension to generate the first letters of each word in a sentence.

7. Write a Python program using generator comprehension to extract digits from a given alphanumeric string.

8. Write a Python program using generator comprehension to generate cubes of odd numbers between 1 and 15.



Cycle 2

Functions, File Handling, OOP Concepts and Exception Handling

Experiment 1

Title: Functions and Lambda Functions

Objectives:

To understand and implement different applications using user-defined functions and lambda (anonymous) functions.

Theory

2.1.1 User Defined Functions

A function is a block of reusable code that performs a specific task when called. It improves modularity and reusability in programs.

Syntax:

```
def function_name(parameters):
    #  function body
    return result
```

2.1.2 Lambda Functions

A lambda function is a small anonymous function defined using the lambda keyword. It can take any number of arguments but can only have one expression. Typically used with map(), filter(), and reduce( ) .The map( ) function applies a given function to each item in an iterable. The filter() applies a conditional function to filter out elements and it returns only elements that return True for the function. The reduce( ) applies a function cumulatively to the items of an iterable and it returns a single value.

Syntax:

```
lambda arguments: expression
```

Example:

```
square = lambda x: x * x
print("Square of 4 is:", square(4))
```

2.1.3 Sample Questions

1. Write a Python program using a user-defined function simple_interest(p, r, t) to calculate and display the Simple Interest.

```
def simple_interest(p, r, t):
    si = (p * r * t) / 100
    print("Simple Interest =", si)
```

```
# Input from the user
principal = float(input("Enter Principal amount: "))
rate = float(input("Enter Rate of Interest: "))
time = float(input("Enter Time in years: "))
# Function call
simple_interest(principal, rate, time)
```

Output

```
Enter Principal amount: 200000
Enter Rate of Interest: 15
Enter Time in years: 3
Simple Interest = 90000.0
```

2. Write a Python program using a user-defined function factorial(n) to compute the factorial of a given number.

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
# Input from user
num = int(input("Enter a number: "))
if num < 0:
    print("Factorial is not defined for negative numbers.")
else:
    print(f"Factorial of {num} is {factorial(num)}")
```

Output

```
Enter a number: 5
Factorial of 5 is 120
```

3. Write a Python program to check whether a number is prime using a user-defined function.

```
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 = int(input("Enter a number: "))
if is_prime(num):
    print(num, "is a prime number.")
else:
    print(num, "is not a prime number.")
```

Output

Enter a number: 17

17 is a prime number.

4. Write a Python program to extract all even numbers from a given list using a lambda function with filter( )

```
numbers = [10, 15, 22, 33, 40, 55, 60] # List of numbers
# Using lambda with filter to get even numbers
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
# Display result
print("Original list:", numbers)
print("Even numbers:", even_numbers)
```

Output

Original list: [10, 15, 22, 33, 40, 55, 60]

Even numbers: [10, 22, 40, 60]

5. Write a Python program to find the square of each number in a given list using the map( ) function with a lambda expression.

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

print(“Squared List:”, squares)

Output

Original List: [1, 2, 3, 4, 5]

Squared List: [1, 4, 9, 16, 25]

2.1.4 Lab Practice Questions

1. Write a Python program using a user-defined function to calculate and display the circumference and area of a circle by accepting its radius.

2. Write a Python program using a user-defined function to determine whether a given year is a leap year or not.

3. Write a Python program using a user-defined function to find the sum of all elements in a given list.

4. Write a Python program to cube each element in a given list using the map( ) function along with a lambda expression.

5. Write a Python program to filter out negative numbers from a list using the filter( ) function with a lambda expression.

Experiment 2

Title: Implement different applications using Nested structures, Modules and File handling operations.

Objectives

- To learn how to define and use nested structures in C programming to represent complex real-world entities
- To understand and implement modular programming using functions for better code readability and reusability.
- To learn how to perform file input/output operations like reading from and writing to text files.

Theory

2.2.1 Nested Structures

Nested structures happen when you put one data structure inside another. For example, a list can contain other lists, or a dictionary can contain other dictionaries or lists as values.This allows you to organize data in multiple layers or levels, making it easier to represent complex or hierarchical information.

2.2.1.1 Use of Nested Structures

- To represent multi-level data naturally.
- To group related information together inside a container.
- To model real-world relationships like tables, trees, or networks.

2.2.1.2 Python Sample Questions

Define a structure Student with fields: Roll Number Name Marks in 3 subjects Address (contains: city, pin code) Write a program to accept and display the student details using nested structures.

```
class Address:
    def __init__(self, city, pin):
        self.city = city
        self.pin = pin
class Student:
    def __init__(self, roll_no, name, marks, address):
        self.roll_no = roll_no
        self.name = name
        self.marks = marks  # List of marks in 3 subjects
        self.address = address
def main():
    roll_no = int(input("Enter Roll Number: "))
    name = input("Enter Name: ")
    marks = []
    print("Enter marks in 3 subjects:")
    for i in range(3):
        mark = int(input(f"Subject {i + 1}: "))
        marks.append(mark)
    city = input("Enter city: ")
    pin = int(input("Enter pin code: "))
```

```
    addr = Address(city, pin)
    student = Student(roll_no, name, marks, addr)
    print("\n--- Student Details ---")
    print(f"Roll Number: {student.roll_no}")
    print(f"Name: {student.name}")
    print(f"Marks: {', '.join(map(str, student.marks))}")
    print(f"City: {student.address.city}")
    print(f"Pin Code: {student.address.pin}")
if __name__ == "__main__":
    main()
```

Output

```
Enter Roll Number: 101
Enter Name: Alice Johnson
Enter marks in 3 subjects:
Subject 1: 85
Subject 2: 90
Subject 3: 88
Enter city: New York
Enter pin code: 10001
--- Student Details ---
Roll Number: 101
Name: Alice Johnson
Marks: 85, 90, 88
City: New York
Pin Code: 10001
```

2.2.2 Modules

In Python, modules are files that contain Python code like functions, classes, and variables that can be reused in other Python programs. Modules help you organize your

code into manageable sections and promote code reusability.A module is just a **.py** file containing definitions and statements.

2.2.2.1 Types of Modules

1. **Built-in modules**: Already included with Python (e.g., math, sys, os).
2. **User-defined modules**: Created by the user (like my_module.py).
3. **Third-party modules**: External modules installed via tools like pip (e.g., requests, numpy, pandas).

2.2.2.2 Python Sample Questions

1. Write a modular program to:

- Accept an array of n integers
- Find the sum and average using functions

```
def accept_numbers():
    n = int(input("Enter the number of elements: "))
    numbers = []
    for i in range(n):
        num = int(input(f"Enter element {i + 1}: "))
        numbers.append(num)
    return numbers
def calculate_sum(arr):
    return sum(arr)
def calculate_average(arr):
    return sum(arr) / len(arr) if len(arr) > 0 else 0
def main():
    nums = accept_numbers()
    total = calculate_sum(nums)
    avg = calculate_average(nums)
    print("\n--- Results ---")
    print("Entered Numbers:", nums)
```

```
    print("Sum:", total)
    print("Average:", round(avg, 2))
if __name__ == "__main__":
    main()
```

Output

```
Enter the number of elements: 5
Enter element 1: 10
Enter element 2: 20
Enter element 3: 30
Enter element 4: 40
Enter element 5: 50
--- Results ---
Entered Numbers: [10, 20, 30, 40, 50]
Sum: 150
Average: 30.0
```

2.2.3 File Handling Operations

File handling in Python enables us to create, update, read, and delete the files stored on the file system. A file is a named location on a disk to store related data.File Handling consists of following three steps:

- Open the file.
- Process file i.e. perform read or write operation.
- Close the file.

2.2.3.1 Types of File

1. **Text Files**- A file whose contents can be viewed using a text editor is called a text file. A text file is simply a sequence of ASCII or Unicode characters.

2. **Binary Files**-A binary file stores the data in the same way as as stored in the memory. The .exe files, mp3 file, image files, word documents are some of the examples of binary files.

   - Text Files – .txt, .csv (data is stored in human-readable format)
   - Binary Files – .dat, .bin (data is stored in machine-readable format)

2.2.3.2 File Operations in Python

a.open() - open the file

b. read() – reads entire content

c. readline() – reads one line at a time

d. readlines() – returns list of lines

e. write() – writes string data

f. writelines() – writes a list of strings

g.close()-close the file.

h.. File open using with Statement - Automatically handles closing of file.(with open("sample.txt", "r") as f:)

2.2.3.3 Python Sample Questions

1. Write a program to create a text file, accept name and age of a person from the user, and write it to the file.

```
def write_to_file():
    # Accept user input
    name = input("Enter your name: ")
    age = input("Enter your age: ")
    # Open file in write mode (creates file if it doesn't exist)
    with open("person.txt", "w") as file:
        file.write(f"Name: {name}\n")
        file.write(f"Age: {age}\n")
    print("\nData written to 'person.txt' successfully.")
# Run the function
write_to_file()
```

Output

```
Enter your name: John Doe
Enter your age: 25
Data written to 'person.txt' successfully.
```

Contents of person.txt (Generated File):

Name: John Doe

Age: 25

2. Write a Python program to read the contents of a text file named merge.txt and count the following: Total number of uppercase letters, Total number of lowercase letters, Total number of digits

```
# Create the file "merge.txt"

def program1():
    with open("merge.txt","r") as f1:
      data=f1.read()
    cnt_ucase =0
    cnt_lcase=0
    cnt_digits=0
    for ch in data:
       if ch.islower():
          cnt_lcase+=1
       if ch.isupper():
          cnt_ucase+=1
       if ch.isdigit():
          cnt_digits+=1
    print("Total Number of UpperCase letters are:",cnt_ucase)
    print("Total Number of LowerCase letters are:",cnt_lcase)
    print("Total Number of  digits are:",cnt_digits)
program1()
```

Output

Total Number of UpperCase letters are: 9

Total Number of LowerCase letters are: 10

Total Number of digits are: 5

2.2.4 Lab Practice Questions

1. Create a nested class structure for a Book with attributes like title, price, and a nested Author class with name and nationality. Input and print details of a book.

2. Write a menu-driven program with functions to:

   - Find factorial of a number
   - Check if a number is prime
   - Generate Fibonacci series up to n terms

3. Accept multiple student records (name and marks) from the user and write them to a file called students.txt.

4. Write a Python program to read a text file and count the following:

   - Total number of lines
   - Total number of words
   - Total number of characters

Experiment 3

Title: Object-Oriented Programming (OOP) in Python

Objectives

By the end of this experiment, you will be able to:

1. Understand the fundamental concepts of Object-Oriented Programming.
2. Design and implement Python programs using OOP principles.
3. Apply OOP techniques to solve real-world problems.

Theory

2.3.1 Object-oriented programming (OOPs)

Object-Oriented Programming (OOP) is a programming approach where programs are structured around objects.

Each object contains data (attributes) and behavior (methods).

2.3.2 Basic Concepts of OOP

Table 2.3.1 Basic concepts in python

Concept	Description	Example
Class	Blueprint for creating objects	class Student:
Object	Instance of a class	s1 = Student()
Encapsulation	Wrapping up of data and methods together	self.name inside a class
Abstraction	Hiding unnecessary details	car.start() hides engine detail
Inheritance	One class inherits properties of another class.	class Car(Vehicle)
Polymorphism	Same method behaves differently in different classes	len("Hello") vs len([1,2,3])

2.3.3 Class and Object in Python

Classes and objects are the two core concepts in object-oriented programming. A class defines what an object should look like, and an object is created based on that class. For example:

Table 2.3.2 Classes and Objects

Class	Objects
Fruit	Apple, Banana, Mango
Car	Volvo, Audi, Toyota

When you create an object from a class, it inherits all the variables and functions defined inside that class.

2.3.4 Encapsulation

Encapsulation means combining data and related operations into one unit and restricting direct access to the internal details. This ensures that sensitive information is protected and only accessible in controlled ways. In Python, encapsulation is achieved by hiding certain attributes and exposing only the necessary operations.

Consider a bank account. The account balance should not be directly accessible to everyone. Instead, customers interact with the account through controlled methods like deposit or withdrawal. In this way, encapsulation hides the balance details and only allows changes through authorized operations.

2.3.5 Inheritance

Inheritance is a mechanism where one class (child class) can reuse and extend the

functionality of another class (parent class). This avoids code duplication and helps build hierarchical relationships among entities. The child class can inherit properties and methods from the parent class while also adding its own unique features.

Think of an Animal as a parent class. It defines basic characteristics such as breathing and moving. A Dog class and a Cat class can inherit these common features but also define their own unique behaviors like barking or meowing. This avoids repeating the basic functions in every new animal class.

2.3.6 Abstraction

Abstraction focuses on hiding unnecessary details and showing only the essential aspects of an object. It allows users to interact with objects without worrying about the complexity of how things work internally. This provides simplicity and a clear interface to the user.

When you drive a car, you only use the steering wheel, brakes, and accelerator. You do not need to understand the complex internal working of the engine or transmission system. In the same way, abstraction allows programmers to use objects without needing to know their full implementation details.

2.3.7 Polymorphism

Polymorphism means “many forms” and allows the same operation or method name to behave differently depending on the object. This means that objects from different classes can respond to the same action in their own way, as long as they share a common interface.

If different animals have a method called speak, a dog may respond with a bark, while a cat may respond with a meow. Even though the same action is used, the outcome differs depending on the object. This flexibility makes programs more dynamic and reusable.

2.3.8 Python sample questions

1. Write a Python program to create a class Person with attributes name and age. Create a child class Student that adds an attribute student_id and a method display(). Create objects and display their details.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
class Student(Person):   # Inheritance
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
```

```
        self.student_id = student_id
    def display(self):
        print(f"Name: {self.name}, Age: {self.age}, Student ID: {self.student_id}")
# Test
s1 = Student("Alice", 20, "S101")
s2 = Student("Bob", 22, "S102")
s1.display()
s2.display()
```

Output:

Name: Alice, Age: 20, Student ID: S101

Name: Bob, Age: 22, Student ID: S102

2. Using the abc module, create an abstract class Appliance with an abstract method turn_on(). Create two subclasses WashingMachine and Refrigerator that implement turn_on(). Demonstrate abstraction by creating objects and calling the method.

```
from abc import ABC, abstractmethod
class Appliance(ABC):
    @abstractmethod
    def turn_on(self):
        pass
class WashingMachine(Appliance):
    def turn_on(self):
        return "Washing Machine is now running"
class Refrigerator(Appliance):
    def turn_on(self):
        return "Refrigerator is cooling"
# Test
a1 = WashingMachine()
a2 = Refrigerator()
```

```
print(a1.turn_on())
print(a2.turn_on())
```

Output:

Washing Machine is now running

Refrigerator is cooling

3. Create a class Student with private attributes __marks and __grade.

Provide setter and getter methods to set marks and calculate grade automatically:

- Marks ≥ 90 → Grade A
- Marks ≥ 75 → Grade B
- Marks ≥ 50 → Grade C
- Otherwise → Grade F

Demonstrate encapsulation by setting marks and displaying grades.

```
class Student:
    def __init__(self, name):
        self.name = name
        self.__marks = 0
        self.__grade = "F"
    def set_marks(self, marks):
        self.__marks = marks
        if marks >= 90:
            self.__grade = "A"
        elif marks >= 75:
            self.__grade = "B"
        elif marks >= 50:
            self.__grade = "C"
        else:
            self.__grade = "F"
```

```
    def get_details(self):
        return f"Name: {self.name}, Marks: {self.__marks}, Grade: {self.__grade}"
# Test
s1 = Student("Alice")
s1.set_marks(92)
s2 = Student("Bob")
s2.set_marks(67)
print(s1.get_details())
print(s2.get_details())
```

Output:

Name: Alice, Marks: 92, Grade: A

Name: Bob, Marks: 67, Grade: C

4. Create a base class Animal with a method make_sound(). Derive three classes Dog, Cat, and Cow that implement the method differently Demonstrate polymorphism by iterating through a list of animals and calling make_sound().

```
class Animal:
    def make_sound(self):
        pass   # base method (to be overridden)
class Dog(Animal):
    def make_sound(self):
        return "Dog barks"
class Cat(Animal):
    def make_sound(self):
        return "Cat meows"
class Cow(Animal):
    def make_sound(self):
        return "Cow moos"
# Demonstrating Polymorphism
```

```
animals = [Dog(), Cat(), Cow()]

for animal in animals:

    print(animal.make_sound())
```

Output:

Dog barks

Cat meows

Cow moos

2.3.9 Lab Practice Questions

1. Write a Python program to create a class Student with attributes name, marks1, marks2, and marks3.Add a method to calculate the total marks and average marks, and another method to display the student's details.

2. Write a Python program to create a base class Employee with attributes name and basic_salary. Create a derived class Manager that adds an attribute allowance and a method to calculate and display the total salary.

3. Write a Python program to create a base class Shape with a method area().

   Derive two classes Rectangle and Circle that override the area() method to calculate and display the area of the respective shapes.

4. Write a Python program to demonstrate encapsulation by creating a class BankAccount.

   - The class should have private attributes __account_number and __balance.
   - Provide public methods deposit(amount), withdraw(amount), and get_balance() to access and modify the balance safely.

     Ensure that the balance cannot be accessed directly from outside the class.

Experiment 4

Title: Data analysis using Libraries- NumPy, Pandas and Matplotlib

Objectives

- To understand the importance of NumPy arrays compared to Python lists.
- To apply NumPy for data cleaning, reshaping, and normalization.
- To use NumPy as a base tool for data science and machine learning applications.

Theory

2.4.1 Numpy in Python

NumPy (Numerical Python) is a powerful Python library used for scientific computing and data science. Its main feature is the ndarray (N-dimensional array), which is faster and more efficient than Python lists. NumPy provides built-in functions for mathematics, statistics, and linear algebra, making it easy to perform operations like mean, median, standard deviation, matrix multiplication, and reshaping data. It also supports random number generation, which is useful in simulations and machine learning. Since libraries like Pandas, Matplotlib, Scikit-learn, TensorFlow are built on top of NumPy, it is considered the foundation of data science in Python.

2.4.2 Basic operations of Numpy

Features	Examples
NumPy Arrays	import numpy as np # 1D array arr1 = np.array([1, 2, 3, 4, 5]) print(arr1) # 2D array arr2 = np.array([[1, 2, 3], [4, 5, 6]]) print(arr2)
Array Properties	print(arr2.shape) # (2,3) → 2 rows, 3 columns print(arr2.ndim) # 2 → dimensions print(arr2.size) # 6 → total elements print(arr2.dtype) # int64 → data type
Array Creation	np.zeros((2,3)) # 2x3 matrix of zeros np.ones((3,3)) # 3x3 matrix of ones np.arange(1,10,2) # array from 1 to 9 step 2 np.linspace(0,1,5) # 5 numbers between 0 and 1 np.eye(3) # 3x3 identity matrix

Array Operations	a = np.array([1,2,3]) b = np.array([4,5,6]) print(a + b) # [5 7 9] print(a * b) # [4 10 18] print(a ** 2) # [1 4 9] print(np.sqrt(a)) # [1.0 1.41 1.73]
Indexing & Slicing of an array	arr = np.array([10,20,30,40,50]) print(arr[0]) # 10 print(arr[1:4]) # [20 30 40] mat = np.array([[1,2,3],[4,5,6],[7,8,9]]) print(mat[0,1]) # 2 print(mat[:,2]) # [3 6 9]
Functions	x = np.array([1,2,3,4,5]) print(np.mean(x)) # 3.0 print(np.median(x)) # 3.0 print(np.std(x)) # standard deviation print(np.sum(x)) # 15
Random Numbers	np.random.rand(2,2) # random 2x2 matrix (0-1) np.random.randint(1,10,5) # 5 random integers between 1–9

2.4.2 Pandas in Python

Pandas is one of the most essential Python libraries for data handling. It makes tasks like cleaning, analyzing, and processing data much easier and faster, which is why it is a core skill for Data Science and Machine Learning. Pandas is an open-source Python library designed for data analysis and manipulation. It provides efficient and flexible data structures:

Series → 1D labeled data.

DataFrame → 2D labeled tabular data.

Table 2.4.2: Basic Syntax and data structures

Features	Examples
Importing Pandas	import pandas as pd
Series	s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd']) print(s)
DataFrame	data = {'Name': ['John', 'Anna', 'Peter'], 'Age': [25, 28, 22], 'City': ['Delhi', 'Mumbai', 'Chennai']} df = pd.DataFrame(data) print(df)
Viewing Data	print(df.head()) # First 5 rows print(df.tail()) # Last 5 rows print(df.info()) # Summary of DataFrame
Selecting Data	print(df['Name']) # Single column print(df[['Name','Age']]) # Multiple columns print(df.iloc[0]) # First row by index print(df.loc[1, 'City']) # Specific cell
Filtering	print(df[df['Age'] > 23]) # Filter rows
Adding New Column	df['Salary'] = [50000, 60000, 45000]
Sorting	print(df.sort_values('Age'))
Grouping	print(df.groupby('City')['Age'].mean())

2.4.3 Matplotlib in python

Matplotlib is one of the most powerful and widely used data visualization libraries in Python, designed to create high-quality static, interactive, and even animated plots with ease. Originally developed by John D. Hunter in 2003, it has now become the foundation of data visualization in the Python ecosystem. Matplotlib integrates seamlessly with NumPy arrays and pandas DataFrames, making it highly efficient for handling large

datasets in scientific computing and data analysis. One of its biggest strengths is the fine-grained control it offers over every element of a figure, including line styles, colors, markers, axes, labels, titles, legends, grids, and more, allowing users to produce simple graphs for learning as well as professional-quality visualizations for publications. Beyond just plotting data, Matplotlib serves as a critical tool for exploring, analyzing, and presenting data trends, which makes it invaluable for scientists, engineers, statisticians, data analysts, and students alike. Its versatility ensures that it remains a go-to library in fields ranging from research and academia to finance and artificial intelligence, where clear and accurate data representation is essential.

To install Matplotlib:	pip install matplotlib
Importing Matplotlib	import matplotlib.pyplot as plt
Line Plot	plt.plot(x, y, color='green', linestyle='--', marker='o')
Bar Chart	categories = ['A', 'B', 'C', 'D'] values = [10, 24, 36, 40] plt.bar(categories, values, color='purple') plt.title("Bar Chart Example") plt.show()
Histogram	import numpy as np data = np.random.randn(1000) # 1000 random values plt.hist(data, bins=20, color='skyblue', edgecolor='black') plt.title("Histogram Example") plt.show()
Scatter Plot	x = [5, 7, 8, 7, 6, 9, 5, 6] y = [99, 86, 87, 88, 100, 86, 103, 87] plt.scatter(x, y, color='red') plt.title("Scatter Plot Example") plt.xlabel("X-axis") plt.ylabel("Y-axis") plt.show()

Pie Chart	labels = ['Python', 'Java', 'C++', 'C#'] sizes = [45, 30, 15, 10] plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90) plt.title("Programming Language Popularity") plt.show()
Customization in Matplotlib	a) Adding Labels and Titles plt.title("My Graph") plt.xlabel("X-axis Label") plt.ylabel("Y-axis Label") b) Grid Lines plt.grid(True) c) Legends plt.plot(x, y, label="Line 1") plt.legend() d) Subplots (multiple graphs in one figure) plt.subplot(1, 2, 1) # 1 row, 2 columns, 1st plot plt.plot([1,2,3], [4,5,6]) plt.subplot(1, 2, 2) # 1 row, 2 columns, 2nd plot plt.plot([1,2,3], [6,5,4]) plt.show()
Line Properties	color → line color ('red', 'blue', 'green', etc.) linestyle → '-', '--', ':', '-.' linewidth → thickness of the line marker → 'o', '^', 's', 'd' Example: plt.plot(x,y,color='purple',linestyle='--',linewidth=2, marker='o')

Ticks (Marks on Axes)	plt.xticks([0, 2, 4, 6, 8], ["Zero", "Two", "Four", "Six", "Eight"]) plt.yticks(rotation=45) # Rotate labels
Axis Limits	plt.xlim(0, 10) # X-axis range plt.ylim(0, 50) # Y-axis range
Figure Size	plt.figure(figsize=(8, 5)) # width=8, height=5
Saving the Graph	plt.savefig("graph.png", dpi=300, bbox_inches='tight')

2.4.4 Python sample questions

1. Write a Python program using NumPy to create an array with elements [5, 10, 15, 20, 25] and find the maximum, minimum, and mean of the array.

```
import numpy as np

arr = np.array([5, 10, 15, 20, 25])

print("Array Elements:", arr)

print("Maximum:", np.max(arr))

print("Minimum:", np.min(arr))

print("Mean:", np.mean(arr))
```

Output

Array Elements: [5 10 15 20 25]

Maximum: 25

Minimum: 5

Mean: 15.0

2. Write a Python program using NumPy to create a 1D array of numbers from 1 to 10 and display only the even numbers from the array.

```
import numpy as np

arr = np.arange(1, 11)

even_numbers = arr[arr % 2 == 0]

print("Array from 1 to 10:", arr)
```

print(“Even numbers:”, even_numbers)

Output

Array from 1 to 10: [1 2 3 4 5 6 7 8 9 10]

Even numbers: [2 4 6 8 10]

3. Write a Python program using NumPy to create an array [10, 20, 30, 40, 50] and find the square root of each element.

```
import numpy as np

arr = np.arange(1, 11)

even_numbers = arr[arr % 2 == 0]

print(“Array from 1 to 10:”, arr)

print(“Even numbers:”, even_numbers)
```

Output

Array from 1 to 10: [1 2 3 4 5 6 7 8 9 10]

Even numbers: [2 4 6 8 10]

4. Write a python program to replace all even numbers in a 1D array with their negative.

```
import numpy as np

arr = np.arange(1, 10)

arr[arr % 2 == 0] *= -1

print(arr)
```

Output

5. Write a Pandas program to perform arithmetic operations on two Pandas Series.

```
import pandas as pd

ds1 = pd.Series([3, 6, 9])

ds2 = pd.Series([2, 4, 6])

ds = ds1 + ds2

print(“Add two Series:”)

print(ds)
```

```
print("Subtract two Series:")
ds = ds1 - ds2
print(ds)
print("Multiply two Series:")
ds = ds1 * ds2
print(ds)
print("Divide Series1 by Series2:")
ds = ds1 / ds2
print(ds)
```

Output

```
Add two Series:
0   5
1   10
2   15
dtype: int64
Subtract two Series:
0   1
1   2
2   3
dtype: int64
Multiply two Series:
0   6
1   24
2   54
dtype: int64
Divide Series1 by Series2:
0   1.5
```

1 1.5

2 1.5

dtype: float64

6. Write a Pandas program to select the rows where the percentage greater than 70

```
import pandas as pd

import numpy as np

exam_data  = {'name': ['Aman', 'Kamal', 'Amjad', 'Rohan', 'Amit', 'Sumit', 'Matthew', 'Kartik', 'Kavita', 'Pooja'],

        'perc': [79.5, 29, 90.5, np.nan, 32, 65, 56, np.nan, 29, 89],

         'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']}

labels = ['A', 'B', 'C', 'B', 'E', 'F', 'G', 'H', 'I', 'J']

df = pd.DataFrame(exam_data , index=labels)

print("Number of student whose percentage more than 70:")

print(df[df['perc'] > 70])
```

Output

Number of student whose percentage more than 70:

```
   name  perc qualify
A  Aman  79.5     yes
C  Amjad  90.5     yes
J  Pooja  89.0     yes
```

7. To write a Python program using **Pandas** to create two Series, one representing the **population** of four zones and another representing the **average income** of the same zones. Then calculate the **per capita income** (average income ÷ population) and store it in a third Series. Finally, print all three Series.

```
import pandas as pd

# Create Series for population and average income of four zones

population = pd.Series([500000, 1200000, 800000, 600000],

                       index=['North', 'South', 'East', 'West'])
```

```
avg_income = pd.Series([20000, 25000, 22000, 18000],
                       index=['North', 'South', 'East', 'West'])
print("Population of Zones:")
print(population, "\n")
print("Average Income of Zones:")
print(avg_income, "\n")
# Calculate per capita income (average income / population)
per_capita_income = avg_income / population
print("Per Capita Income of Zones:")
print(per_capita_income)
```

Output

```
Population of Zones:
North     500000
South    1200000
East      800000
West      600000
dtype: int64
Average Income of Zones:
North    20000
South    25000
East     22000
West     18000
dtype: int64
Per Capita Income of Zones:
North    0.040000
South    0.020833
East     0.027500
```

West 0.030000

dtype: float64

8. Write a Python program using NumPy and pandas to create a 5×5 DataFrame with random integers between 1 and 100. Replace all the elements on the main diagonal with 0, and display the modified DataFrame.

```
iimport pandas as pd

import numpy as np

# Create a 5x5 DataFrame with random integers between 1 and 100

df = pd.DataFrame(np.random.randint(1, 100, 25).reshape(5, 5))

print("Original DataFrame:")

print(df)

# Replace main diagonal elements with 0

for i in range(min(df.shape)):

    df.iat[i, i] = 0

print("\nModified DataFrame (Main diagonal replaced with 0):")

print(df)
```

Output

Original DataFrame:

```
   0   1   2   3   4
0  34  65  83  69  47
1  91  45  66  59  45
2  11  55  59  64  50
3  88  72  10  29  23
4  57  41  56   6  59
```

Modified DataFrame (Main diagonal replaced with 0):

```
   0   1   2   3   4
0   0  65  83  69  47
1  91   0  66  59  45
```

2 11 55 0 64 50

3 88 72 10 0 23

4 57 41 56 6 0

9. Write a Pandas program to add a new column Total which stores the sum of marks in each row

```
import pandas as pd

# Create a sample DataFrame with student marks

data = {

    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],

    'Maths': [78, 85, 92, 66, 74],

    'Physics': [88, 79, 95, 80, 72],

    'Chemistry': [90, 82, 89, 76, 85]

}

df = pd.DataFrame(data)

print("Original DataFrame:")

print(df)

# Add a new column 'Total' that stores the sum of marks in each row

df['Total'] = df[['Maths', 'Physics', 'Chemistry']].sum(axis=1)

print("\nDataFrame after adding 'Total' column:")

print(df)
```

Output

Original DataFrame:

	Name	Maths	Physics	Chemistry
0	Alice	78	88	90
1	Bob	85	79	82
2	Charlie	92	95	89
3	David	66	80	76
4	Eva	74	72	85

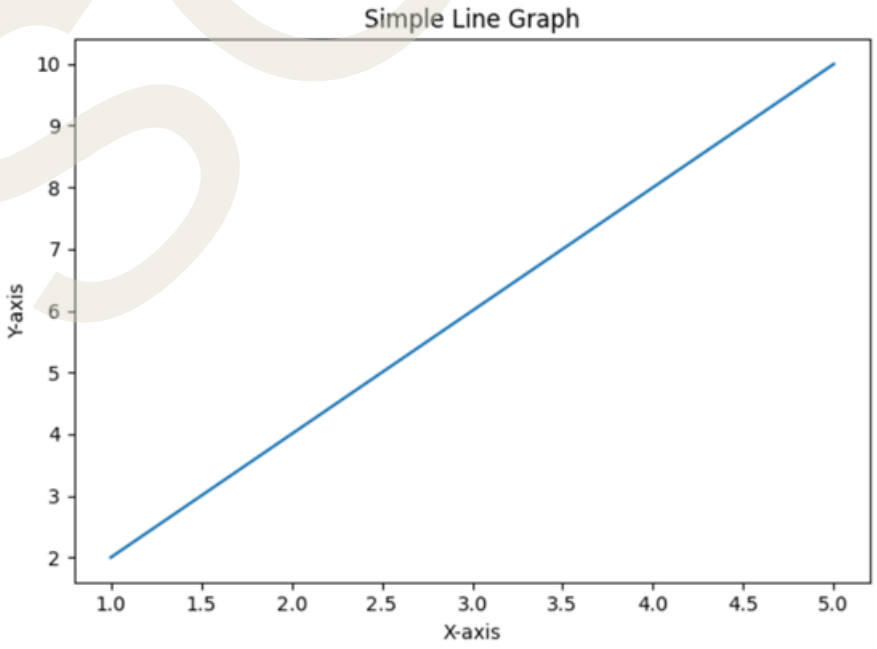
DataFrame after adding 'Total' column:

	Name	Maths	Physics	Chemistry	Total
0	Alice	78	88	90	256
1	Bob	85	79	82	246
2	Charlie	92	95	89	276
3	David	66	80	76	222
4	Eva	74	72	85	23

10. Write a Python program using Matplotlib to plot a simple line graph for x = [1, 2, 3, 4, 5] and y = [2, 4, 6, 8, 10]. Add a title, labels for axes, and display the graph.

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]

y = [2, 4, 6, 8, 10]

plt.plot(x, y)

plt.title("Simple Line Graph")

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.show()
```

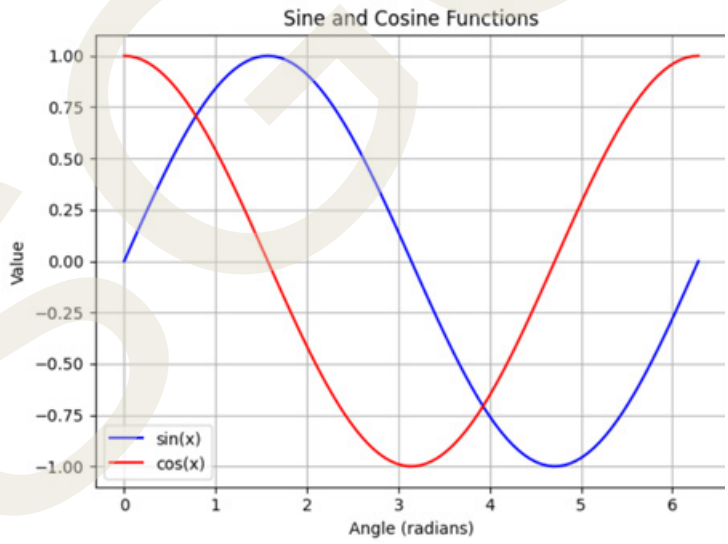
Output



11. Write a Python program using Matplotlib to plot sine and cosine curves for values from 0 to 2π.

```
import matplotlib.pyplot as plt

import numpy as np

x = np.linspace(0, 2*np.pi, 100)   # 100 values between 0 and 2π

y1 = np.sin(x)

y2 = np.cos(x)

plt.plot(x, y1, label="sin(x)", color='blue')

plt.plot(x, y2, label="cos(x)", color='red')

plt.title("Sine and Cosine Functions")

plt.xlabel("Angle (radians)")

plt.ylabel("Value")

plt.legend()

plt.grid(True)

plt.show()
```

Output



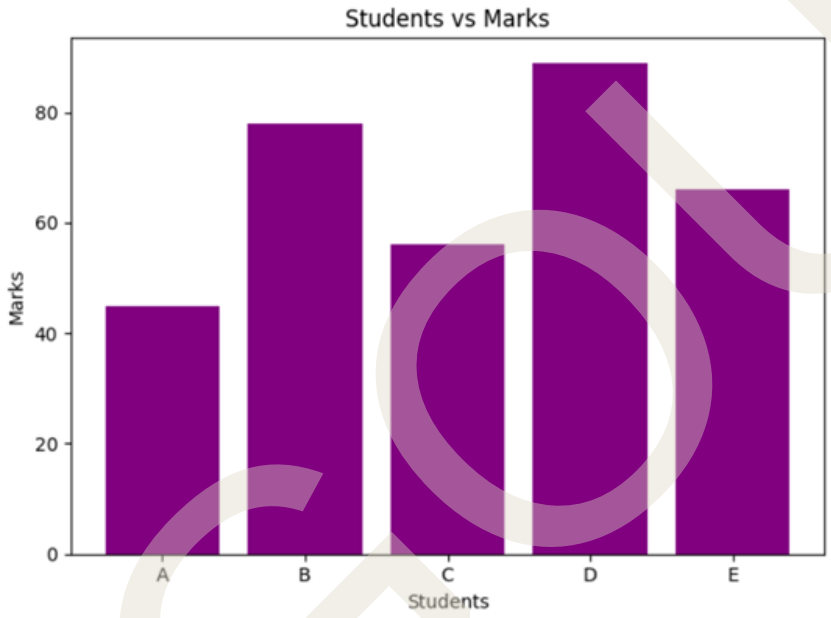
12. Write a Python program using Matplotlib to create a bar chart of students and their marks:

Students = ['A', 'B', 'C', 'D', 'E'], Marks = [45, 78, 56, 89, 66].

```
import matplotlib.pyplot as plt

students = ['A', 'B', 'C', 'D', 'E']
```

```
marks = [45, 78, 56, 89, 66]
plt.bar(students, marks, color='purple')
plt.title("Students vs Marks")
plt.xlabel("Students")
plt.ylabel("Marks")
plt.show()
```

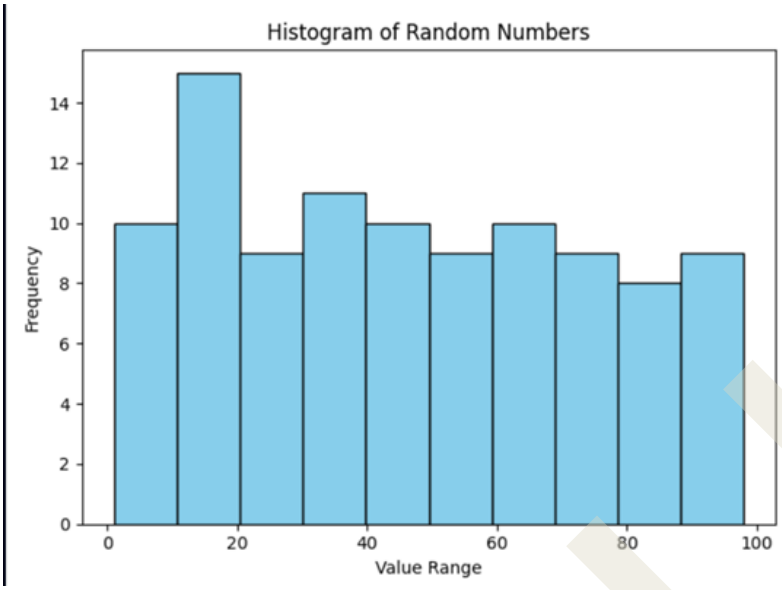
Output



13. Write a Python program using Matplotlib to generate 100 random numbers between 1 and 100 and display their frequency distribution using a histogram with 10 bins.

```
import matplotlib.pyplot as plt
import numpy as np
data = np.random.randint(1, 101, 100)
plt.hist(data, bins=10, color='skyblue', edgecolor='black')
plt.title("Histogram of Random Numbers")
plt.xlabel("Value Range")
plt.ylabel("Frequency")
plt.show()
```

Output

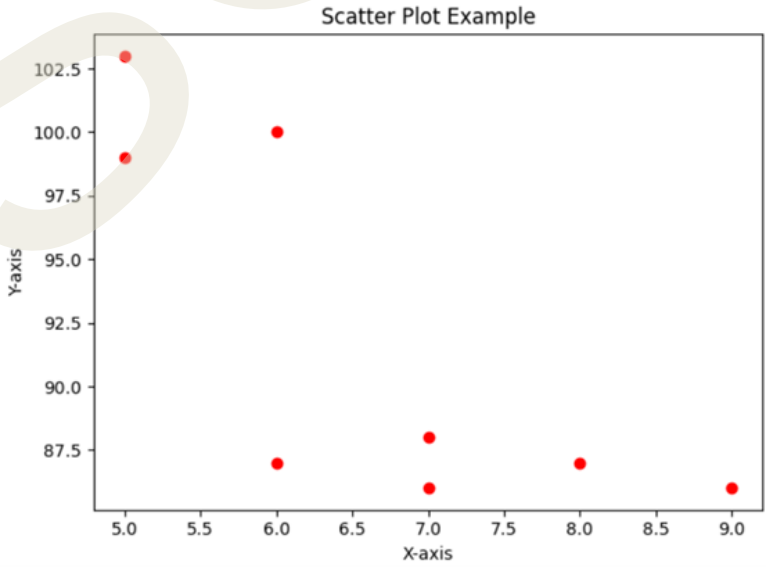


14. Write a Python program using Matplotlib to create a scatter plot of the points:

x = [5, 7, 8, 7, 6, 9, 5, 6] y = [99, 86, 87, 88, 100, 86, 103, 87]

```
import matplotlib.pyplot as plt
x = [5, 7, 8, 7, 6, 9, 5, 6]
y = [99, 86, 87, 88, 100, 86, 103, 87]
plt.scatter(x, y, color='red', marker='o')
plt.title("Scatter Plot Example")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

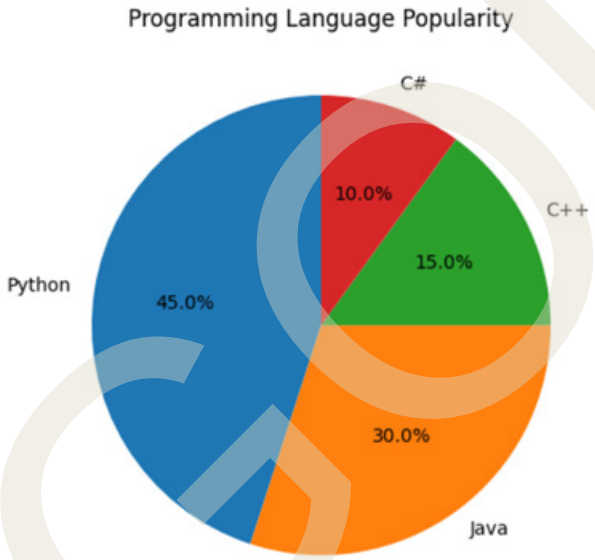
Output



15. Write a Python program using Matplotlib to draw a pie chart showing the popularity of programming languages: Python = 45, Java = 30, C++ = 15, C# = 10.

```
import matplotlib.pyplot as plt

languages = ['Python', 'Java', 'C++', 'C#']

popularity = [45, 30, 15, 10]

plt.pie(popularity, labels=languages, autopct='%1.1f%%', startangle=90)

plt.title("Programming Language Popularity")

plt.show()
```

Output



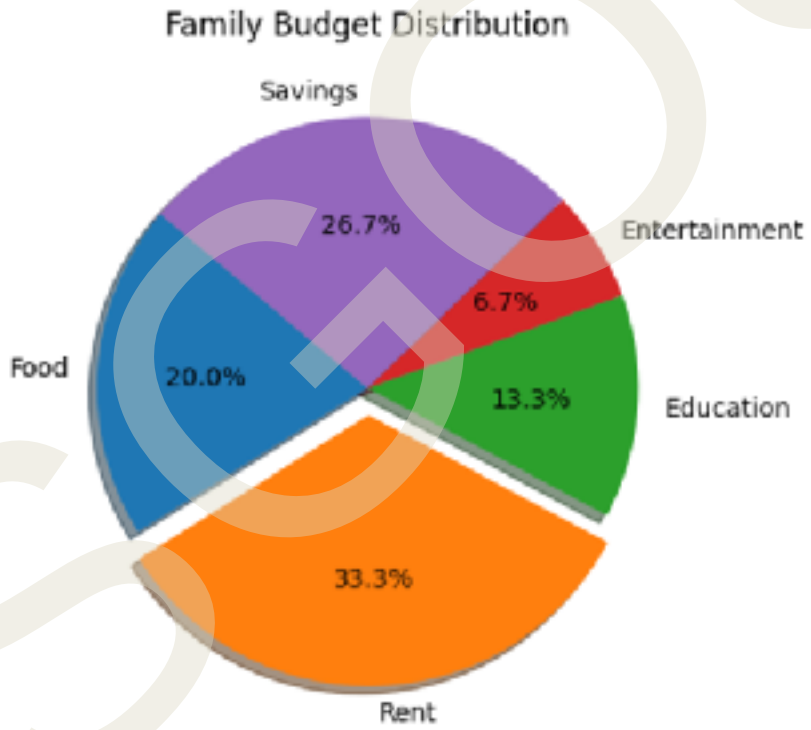
16. Write a Python program using Matplotlib to display a pie chart of expenses in a family budget.

- Categories: Food, Rent, Education, Entertainment, Savings
- Values: Food = ₹3000, Rent = ₹5000, Education = ₹2000, Entertainment = ₹1000, Savings = ₹4000
- Calculate the percentage share of each expense and display it in the chart.
- Highlight the largest expense (Rent) using the explode function.

```
import matplotlib.pyplot as plt

# Data

categories = ["Food", "Rent", "Education", "Entertainment", "Savings"]
```

```
expenses = [3000, 5000, 2000, 1000, 4000]

# Explode: highlight the largest expense (Rent)

explode = [0, 0.1, 0, 0, 0]

# Plot pie chart

plt.pie(expenses, labels=categories, autopct='%1.1f%%', startangle=140, explode=explode, shadow=True)

# Title

plt.title("Family Budget Distribution")

# Show

plt.show()
```

Output



2.4.6 Lab Practice Questions

1. Write a Python program using NumPy to generate 15 random integers between 1 and 100 and find the maximum, minimum, and mean.

2. Write a Python program using NumPy to create an array containing numbers from 1 to 12. Reshape the array into: A 3×4 matrix and A 4×3 matrix

3. Write a Python program using NumPy to create two 3×3 matrices with random integers between 1 and 50. Perform the following operations:Matrix Addition and Matrix Multiplication.

4. Write a Python program using NumPy to create an array of 10 random integers between 1 and 50. Sort the array in ascending order and display the result.

5. Write a Python program using NumPy and pandas to create a 5×5 DataFrame with random integers between 1 and 100. Replace all the elements on the anti-diagonal with 0, and display the modified DataFrame.

6. Write a Pandas program to add a new column Average which stores the average marks of each student in a DataFrame.

7. Write a Pandas program to find the mean, median, and standard deviation of numerical columns in a DataFrame.

8. Write a Python program using Matplotlib to plot $y = x^2$ for $x = [1,2,3,4,5]$ with: Line color = green, Line style = dashed, Markers = circles, Line width = 2.

9. Write a Python program using Matplotlib to plot the exponential function y=exy = e^xy=ex and the logarithmic function y=ln⁡(x)y = \ln(x)y=ln(x) for values of xxx from 0.1 to 5.

10. Write a Python program using Matplotlib to plot two line graphs on the same figure.
    - The first line should represent the function y = 2x.
    - The second line should represent the function y = x.
    - Label the axes, add a title, and include a legend to differentiate the two lines.

11. Write a Python program using Matplotlib to plot a pie chart showing the distribution of marks in four subjects: Math, Science, English, and History.

12. Write a Python program using Matplotlib to create a pie chart showing the percentage usage of social media apps: WhatsApp, Instagram, Facebook, and Twitter. Highlight the app with the maximum usage. [explode = [0.1, 0, 0, 0], # Plot pie chart

plt.pie(usage, labels=apps, autopct='%1.1f%%', explode=explode, startangle=90)]

13. Write a Python program using Matplotlib to plot a Histogram of student marks.
    - Divide the marks into 5 intervals (bins).
    - Display frequency (number of students) on the Y-axis.

- Add suitable labels and a title.

14. Write a Python program using Matplotlib to create a scatter plot of the heights and weights of students.
    - Plot height on the X-axis and weight on the Y-axis.
    - Use different colors and markers for better visualization.
    - Add labels and a title.

സർവ്വകലാശാലാഗീതം

വിദ്യയാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കൂരിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ് വിളങ്ങണം
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരീപ്പുഴ ശ്രീകുമാർ

SREENARAYANAGURU OPEN UNIVERSITY

Regional Centres

Kozhikode
Govt. Arts and Science College
Meenchantha, Kozhikode,
Kerala, Pin: 673002
Ph: 04952920228
email: rckdirector@sgou.ac.in

Thalassery
Govt. Brennen College
Dharmadam, Thalassery,
Kannur, Pin: 670106
Ph: 04902990494
email: rctdirector@sgou.ac.in

Tripunithura
Govt. College
Tripunithura, Ernakulam,
Kerala, Pin: 682301
Ph: 04842927436
email: rcedirector@sgou.ac.in

Pattambi
Sree Neelakanta Govt. Sanskrit College
Pattambi, Palakkad,
Kerala, Pin: 679303
Ph: 04662912009
email: rcpdirector@sgou.ac.in

DON'T LET IT BE TOO LATE

SAY NO TO DRUGS

LOVE YOURSELF AND ALWAYS BE HEALTHY

SREENARAYANAGURU OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

PYTHON PROGRAMMING LAB

COURSE CODE: B24DS03PC

Sreenarayanaguru Open University
Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841