SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur-603203

DEPARTMENT OF MASTER OF COMPUTER APPLICATION

ACADEMIC YEAR: 2026-2027 (ODD SEMESTER)

LAB MANUAL

(REGULATION - 2024)

MC4167 – PYTHON PROGRAMMING LABORATORY

FIRST SEMSTER

Prepared By

Ms. R. Jeevitha , Assistant Professor, Department of Computer Applications

VISION OF THE DEPARTMENT

To educate students with conceptual knowledge and technical skills in the field of Computer Applications with moral and ethical values to achieve excellence in academic, industry, and research- centric environments.

MISSION OF THE DEPARTMENT

- To bring the most brilliant students and faculty together to understand the strengths and limits of computation, invent next-generation computing systems, and create innovative solutions to real-world problems.
- To provide conducive environment so as to achieve excellence in teaching-learning, and research and development activities.
- Deliver knowledge among students through novel pedagogical methods in the varied areas of computer sciences with thrust on applications so as to enable students to undertake research.
- To facilitate students to nurture skills to practice their professions competently to meet the ever-changing needs of society.

INDEX

E. NO	EXPERIMENT NAME	Pg.No.
1	Python programming using simple statements and expressions	
1a	Exchange the values of two variables	1
1b	Circulate the values of n variables	2
1c	Distance between two points	4
2	Scientific problems using Conditionals and Iterative loops.	
2a	Fibonacci series	5
2b	Armstrong Number	6
2c	Palindrome	8
3	Searching	
3a	Linear Search	10
3 b	Binary search	12
4	Sorting	
4a	Selection sort	15
4 b	Insertion sort	17
5a	Merge sort	19
5b	Quick Sort	21
6	Implementing applications	
6a	Using Lists	24
6b	Using Tuples	26
7	Implementing applications using Sets, Dictionaries.	29
8	Implementing programs using Functions.	36
9	Implementing programs using Strings.	40
10	Implementing programs using written modules and Python Standard Libraries (pandas, numpy, Matplotlib, scipy)	44
11	Implementing real-time/technical applications using File handling.	56
12	Implementing real-time/technical applications using Exception handling.	61
13	Creating and Instantiating classes	67
14	Topic Beyond Syllabus: Automation & Scripting in Python	70
15	Sample Viva Questions	74

MC4167 **PYTHON PROGRAMMING LABORATORY** **L T P C**

0 0 4 2

LIST OF EXPERIMENTS:

1. Python programming using simple statements and expressions (exchange the values of two variables, circulate the values of n variables, distance between two points).
2. Scientific problems using Conditionals and Iterative loops.
3. Linear search and Binary search
4. Selection sort, Insertion sort
5. Merge sort, Quick Sort
6. Implementing applications using Lists, Tuples.
7. Implementing applications using Sets, Dictionaries.
8. Implementing programs using Functions.
9. Implementing programs using Strings.
10. Implementing programs using written modules and Python Standard Libraries (pandas,numpy, Matplotlib, scipy)
11. Implementing real-time/technical applications using File handling.
12. Implementing real-time/technical applications using Exception handling.
13. Creating and Instantiating classes

Total: 60 Periods

LIST OF EQUIPMENTS FOR A BATCH OF 30 STUDENTS

Hardware Requirements

Component	Minimum Requirement	Recommended Requirement
Processor (CPU)	Dual-core 2 GHz	Quad-core 2.5 GHz or higher
RAM	4 GB	8 GB or higher
Storage (HDD/SSD)	20 GB free space	50 GB SSD for faster I/O
Graphics	Integrated graphics	Dedicated GPU (optional for ML/AI)
Display	1024×768 resolution	1920×1080 Full HD
Internet	Required for downloading libraries and online resources	High-speed internet for cloud-based datasets

Software Requirements

Software	Purpose	Recommended Version / Notes
Operating System	Run Python and libraries	Windows 10/11, macOS, Linux Ubuntu 20.04+
Python Interpreter	Execute Python code	Python 3.10+
IDE / Editor	Writing and running Python programs	PyCharm, VS Code, Jupyter Notebook, Anaconda
Libraries / Packages	Data Analysis & Visualization	pandas, numpy, matplotlib, seaborn, plotly, scipy, scikit-learn
Web Automation & Scraping	Automation tasks	requests, beautifulsoup4, selenium
Excel / PDF Automation	File handling and reporting	openpyxl, pandas, PyPDF2, reportlab
Version Control	Code management	Git & GitHub / GitLab
Package Manager	Install Python packages	pip (latest version)
Web Browser	Testing web scraping / web apps	Chrome / Firefox / Edge

COURSE OUTCOMES

At the end of the course, students will be able to

CO1: Apply the Python language syntax including control statements, loops and functions to solve a wide variety of problems in mathematics and science.

CO2: Use the core data structures like lists, dictionaries, tuples and sets in Python to store, process and sort the data.

CO3: Create files and perform read and write operations.

CO4: Illustrate the application of python libraries.

CO5: Handle exceptions and create classes and objects for any real time applications.

CO- PO MATRIX

Course Outcomes	PROGRAM OUTCOMES					
	1	**2**	**3**	**4**	**5**	**6**
CO1	2	1	3	3	2	2
CO2	2	1	3	3	2	2
CO3	1	1	3	2	2	2
CO4	2	1	3	2	2	2
CO5	2	1	3	3	2	3
AVG	1.8	1	3	2.6	2	2.2

EVALUATION PROCEDURE FOR EACH EXPERIMENT

S.No	Description	Mark
1.	Aim & Pre-Lab discussion	20
2.	Observation	20
3.	Conduction and Execution	30
4.	Output & Result	10
5.	Viva	20
Total		**100**

INTERNAL ASSESSMENT FOR LABORATORY

S.No	Description	Mark
1.	Conduction & Execution of Experiment	30
2.	Record	10
3.	Model Test	20
Total		**60**

EX.NO 1: PYTHON PROGRAMMING USING SIMPLE STATEMENTS AND EXPRESSIONS

AIM: To develop Python programs to perform basic operations such as exchanging values of two variables, circulating the values of multiple variables, and calculating distance between two points using simple statements and expressions.

 # Ex.No:1A To exchange the given values of two variables

Algorithm: Swapping Two Variables

1. **Start**
2. **Input** values of a and b.
3. **Display** values of a and b before swapping.
4. **Swap** the values using

   temp = a

   a = b

   b = temp

   (or in Python simply a, b = b, a)
5. **Display** values of a and b after swapping.
6. **Stop**

Program:

```
a = input("Enter value for a: ")
b = input("Enter value for b: ")
# Displaying before swapping
print("Before swapping:")
print("a =", a)
print("b =", b)
# Swapping the values
a, b = b, a
# Displaying after swapping
print("After swapping:")
print("a =", a)
print("b =", b)
```

Part A: Fill in the blanks

1. Swapping means exchanging the _____of two variables.
2. In Python, swapping can be done in one line using _____assignment.
3. If a = 5 and b = 10, after swapping, a = _____and b = _____.

Part B: Programming Task

- Write a program to swap three variables (a → b → c → a).

Part C: Code Snippet – Swap Three Variables

```
# Swap three variables in a single statement
p = 1
q = 2
r = 3
print("Before swapping: p =", p, "q =", q, "r =", r) # Swap such that p->q, q->r, r->p
_____= _____, _____, _____  # Fill in the blanks
print("After swapping: p =", p, "q =", q, "r =", r)
```

 # Ex.No:1B Circulate the values of n variables

Algorithm: Circulating Elements of a List

1. **Start**
2. Read the number of elements n.
3. Initialize an empty list list1.
4. Repeat steps 5–6 for i = 1 to n:
    - Read an integer ele.
    - Append ele to list1.
5. Display the original list list1.
6. Repeat steps 7–9 for i = 1 to n:
    - Remove the first element of list1.
    - Append the removed element at the end of list1.
    - Display list1 after the rotation.
7. **Stop**

Program:

```
no_of_terms = int(input("Enter number of values: "))
 list1 = []
# Input values
for val in range(no_of_terms):
        ele = int(input("Enter integer: ")) list1.append(ele)
# Circulate and display values
print("Circulating the elements of list:", list1)
# Circulate one by one and show each step
for val in range(no_of_terms):
        ele = list1.pop(0)    # remove first element
        list1.append(ele)     # append it to the end
        print("After rotation", val+1, ":", list1)
```

Sample Output:

Enter number of values: 4 Enter integer: 10

Enter integer: 20

Enter integer: 30

Enter integer: 40

Circulating the elements of **list: [**10, 20, 30, 40**]**

After rotation 1 **: [**20, 30, 40, 10**]**

After rotation 2 **: [**30, 40, 10, 20**]**

After rotation 3 **: [**40, 10, 20, 30**]**

After rotation 4 **: [**10, 20, 30, 40**]**

Part A: Fill in the blanks

1. Circulation means shifting elements of a list _____by one position.
2. The method pop(0) removes the _____ element of the list.
3. The time complexity of list circulation is_____.

Part B: Programming Task

- Write a program to rotate a list by k positions, where k is given by the user.
- Code & Execute Single Circulation without Looping

 # Ex.No: 1C To Calculate the Distance Between Two Points.

Algorithm:

1. **Start**
2. Read the coordinates of the first point (x1, y1).
3. Read the coordinates of the second point (x2, y2).
4. Apply the distance formula:

$$\text{distance} = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

5. Display the calculated distance.
6. **Stop**

Program:

```
import math
# Input coordinates of first point
x1 = float(input("Enter x1: "))
y1 = float(input("Enter y1: "))
# Input coordinates of second point
x2 = float(input("Enter x2: "))
y2 = float(input("Enter y2: "))
# Calculate distance
distance = math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
# Output the result
print("Distance between the two points is:", distance)
```

Sample Output:

```
Enter x1: 2
Enter y1: 3
Enter x2: 7
Enter y2: 8
Distance between the two points is: 7.0710678118654755
```

Programming Task:

- Write a program to calculate the midpoint between two points (x1, y1) and (x2, y2).

Result:

The program successfully exchanged/circulated values of variables and calculated the distance between two points as expected.

EX.NO 2: SCIENTIFIC PROBLEMS USING CONDITIONALS AND ITERATIVE LOOPS

AIM: To implement Python programs that solve scientific and mathematical problems using conditional statements and iterative loops such as factorial calculation, Fibonacci series, prime number checking, and Armstrong numbers.

 # Ex.No:2A Function to generate Fibonacci series

Algorithm:

1. **Start**
2. Read a number n (number of terms).
3. If n < 0:
   - Display *"Fibonacci series cannot be generated"* and **Stop**.
4. If n == 1:
   - Display [0] and **Stop**.
5. Otherwise:
   - Initialize first two terms: a = 0, b = 1.
   - Repeat n times:
     - Append a to the series list.
     - Update a, b = b, a + b.
   - Display the Fibonacci series.
6. **Stop**

Program:

```
def fibonacci_series(n):
    a, b = 0, 1
    series = []
    for _ in range(n):
        series.append(a) a, b = b, a
        + b
    return series
```

```
# Input from the user
num = int(input("Enter a number: "))

# Check conditions
 if num == 1:
        print("The number is equal to 1. Fibonacci series: [0]")
elif num < 0:
        print("The number is less than 0. Fibonacci series cannot be generated.")
else:
        # Generate and display Fibonacci series
        fib = fibonacci_series(num)
        print("Fibonacci series of", num, "terms:", fib)
```

Sample Output:

```
Enter a number: -3
The number is less than 0. Fibonacci series cannot be generated.
```

Part A: Programming Task

1. Write a Python program to generate the first **10 Fibonacci numbers**.
2. Modify the program so that it prints only **even Fibonacci numbers** up to n terms.
3. Modify the program to **sum all Fibonacci numbers** up to n terms.

Part B: Output Practice

If the input is n = 7, write the output of the program: Fibonacci series of 7 terms: _____

```
# Ex.No:2B Define a function to check Armstrong number
def is_armstrong(num):
        original = num
        # Step 1: Count number of digits
        n = 0
        temp = num
        while temp > 0:
                temp = temp // 10 n += 1
```

```
    # Step 2: Calculate sum of digits raised to power n
    temp = num
    result = 0
    while temp > 0:
        digit = temp % 10
        # Manual power calculation (digit^n)
        power = 1
        for _ in range(n):
            power *= digit
        result += power
        temp = temp // 10

    # Step 3: Compare result with original number
    return result == original

# Main program
number = int(input("Enter a number: "))
if is_armstrong(number):
    print(number, "is an Armstrong number.")
else:
    print(number, "is NOT an Armstrong number.")
```

Sample Output:

Example 1: Armstrong number Enter a number:

153

153 is an Armstrong number.

Example 2: Not an Armstrong number Enter a number:

123

123 is NOT an Armstrong number.

Example 3: Single digit number Enter a number:

7

7 is an Armstrong number.

Part A: Viva Questions

1. What is an Armstrong number?
2. Give an example of a 3-digit Armstrong number.
3. How do we calculate the sum of digits raised to the power of the number of digits?

Part B: Programming Task / Activity

1. Write a program to find all Armstrong numbers between 1 and 1000.
2. Modify the function so that it checks Armstrong numbers for **any base (not only base 10)**.
3. Write a program to count how many Armstrong numbers are present up to n.

Part C: Output Practice

Q1. If input is 370, what will be the output?

_____

Q2. If input is 123, what will be the output?

_____

Q3. If input is 9474, what will be the output?

_____

#Ex.No:2C Define a function to check for palindrome

```
def is_palindrome(num):
    original = num reverse = 0
      # Reverse the number
      while num > 0:
          digit = num % 10
          reverse = reverse * 10 + digit
          num = num // 10
    # Compare original with reversed number
    return original = = reverse
# Main program
number = int(input("Enter a number: "))
if is_palindrome(number):
    print(number, "is a Palindrome.")
else:
```

```
    print(number, "is NOT a Palindrome.")
```

Sample Output:

Example 1: Palindrome number

Enter a number: 121

121 is a Palindrome. **Example 2:**

Not a palindrome Enter a number:

123

123 is NOT a Palindrome. **Example 3:**

Single digit number Enter a number: 7

7 is a Palindrome.

Part A: Programming Task / Activity

1. Write a program to find all palindrome numbers between 1 and 500.
2. Modify the function to check if a **string** (word) is a palindrome (e.g., "radar", "madam").
3. Write a program to check if a number is both **palindrome and Armstrong**.

Part B: Output Practice

Q1. If input is 454, what will be the output?

_____

Q2. If input is 123, what will be the output?

_____

Result:

The program correctly implemented factorials, Fibonacci series, prime number checking, and other mathematical computations using loops and conditional statements.

EX.NO 3: LINEAR SEARCH AND BINARY SEARCH

AIM: To implement Python programs that search for a given element in a list using linear search and binary search techniques and to understand their efficiency.

 # Ex.No:3A Define the linear search function

Algorithm:

1. Start
2. Input n(number of elements).
3. Initialize an empty list arr.
4. Repeat for i = 1to n:

o Read element and insert into arr.

5. Input the target element to search.
6. Define a function linear_search(arr, target)that:
   - Loops through each index of the array.
   - If arr[index] == target, return the index.
   - If element not found, return -1.
7. Call the function and store the result in position.
8. If position != -1:
   - Print the position (1-based) and index (0-based).

   Else:
   - Print “Element not found”.
9. Stop.

Program:

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

# Main program
n = int(input("Enter number of elements: "))
arr = []
```

```
# Get the elements
for i in range(n):
        ele = int(input(f"Enter element {i+1}: "))
        arr.append(ele)

# Target to search
target = int(input("Enter the number to search: "))

# Call the function
position = linear_search(arr, target)
# Display result
if position != -1:
        print(f"Element found at position {position} (Index {position})")
else:
        print("Element not found in the list.")
```

```
Enter number of elements: 5
Enter element 1: 10
Enter element 2: 20
Enter element 3: 30
Enter element 4: 40
Enter element 5: 50
Enter the number to search: 30 Element found at
position 3 (Index 2)
```

Part A: Viva Questions

1. What is the **time complexity** of Linear Search in the best and worst case?
   - Best Case: O(1), Worst Case: O(n).
2. How is **Linear Search different from Binary Search**?
   - Linear Search: Works on both sorted and unsorted lists.
   - Binary Search: Works only on **sorted** lists, faster (O(log n)).
3. What happens if the searched element appears **multiple times** in the list?
   - Linear Search returns the **first occurrence** (unless modified).

Part B: Worksheet (Fill in the blanks / Short Qs)

1. Linear Search compares the target element with each element in the list _____.
2. The return value of the function when an element is not found is _____.
3. Linear Search has a worst-case time complexity of _____.

Part C: Practice Exercises

1. Modify the program to search for **all occurrences** of the element and display their indices.
2. Write a program to count how many times a given element appears in the list using Linear Search.
3. Compare Linear Search program and checking their performance for 1000 elements.
4. Modify the program to handle **strings** instead of numbers.
5. Write a program to search for an element in a list of **student names**.

#Ex.No:3B Define binary search function

Algorithm:

1. Start
2. Input n (number of elements).
3. Initialize an empty list arr.
4. Read n elements into the list.
5. Sort the list (Binary Search requires a sorted list).
6. Input the target element to search.
7. Set low = 0 and high = n - 1.
8. Repeat until low <= high:
   - Find mid = (low + high) // 2.
   - If arr[mid] == target, return mid.
   - If arr[mid] < target, set low = mid + 1.
   - Else, set high = mid - 1.
9. If not found, return -1.
10. Display result.
11. Stop.

Program:

```
def binary_search(arr, target):
    low = 0; high = len(arr) - 1
```

```
        while low <= high:
                mid = (low + high) // 2

            if arr[mid] == target:
                return mid   # Target found
            elif arr[mid] < target: low = mid + 1
            else:
                high = mid - 1

        return -1   # Target not found
# Main program
n = int(input("Enter number of elements: "))
arr = []

# Input the elements
for i in range(n):
        ele = int(input(f"Enter element {i+1}: "))
        arr.append(ele)

# Sort the list (Binary search requires sorted list)
arr.sort()
print("Sorted list:", arr)
# Input the number to search
target = int(input("Enter number to search: "))
# Perform binary search
pos = binary_search(arr, target)

# Output result
 if pos != -1:
        print(f"Element found at position {pos} (Index {pos})")
 else:
        print("Element not found in the list.")
```

Sample Output:

Enter number of elements: 5

Enter element 1: 12

Enter element 2: 7

Enter element 3: 25

Enter element 4: 3

Enter element 5: 18

Sorted list: [3, 7, 12, 18, 25]

Enter number to search: 18

Element found at position 3 (Index 3)

Part A: VIVA QUESTIONS:

1. What is the difference between linear search and binary search?
2. Why does binary search require a sorted list?
3. What is the time complexity of binary search?

Part B: PRACTICE EXERCISES:

1. Modify the program to **search multiple target values** in the same sorted list.
2. Implement **binary search recursively** instead of using a loop.
3. Count the **number of comparisons** made while searching for the target.
4. Modify the program to **return all indices** if the target element appears multiple times.

Result:

The program correctly located elements in a list using linear and binary search methods and returned accurate positions or “not found” messages.

EX.NO: 4 SELECTION SORT AND INSERTION SORT

AIM: To develop Python programs to sort a list of elements using selection sort and insertion sort algorithms and compare their execution.

Algorithm (Selection Sort):

1. Start
2. Input N (Number Of Elements)
3. Initialize An Empty List Arr
4. Read N Elements Into The List
5. For Each Index I From 0 To N-1:
    - Find The **Minimum Element** In The Unsorted Portion
    - Swap It With The Element At Index I
6. Repeat Until The List Is Sorted
7. Display The Sorted List
8. Stop

Program:

```
#Ex.No:4A Define selection sort function
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        # Find the minimum element in the remaining list
        for j in range(i+1, n):
            if arr[j] < arr[min_index]: min_index = j
        # Swap the found minimum with the first unsorted element
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr

# Main program
n = int(input("Enter number of elements: "))
arr = []

# Input elements
```

```
for i in range(n):
        ele = int(input(f"Enter element {i+1}: "))
        arr.append(ele)
print("Original list:", arr)

# Call the sort function
sorted_arr = selection_sort(arr)

# Display sorted list
print("Sorted list using Selection Sort:", sorted_arr)
```

Sample Output:

Enter number of elements: 5
Enter element 1: 40
Enter element 2: 10
Enter element 3: 30
Enter element 4: 20
Enter element 5: 50
Sorted list: [10, 20, 30, 40, 50] Enter number to search: 30
Element found at position 3 (Index 2)

Give the Output For: Element Not Found

Part A: VIVA QUESTIONS:

1. Explain the working principle of **Selection Sort**.
2. What is the **time complexity** of Selection Sort in best, worst, and average cases?
3. How does Selection Sort differ from Insertion Sort?

Part B: PRACTICE EXERCISES:

1. Modify the program to **sort in descending order**.
2. Count the **number of swaps** performed during sorting.
3. Test the algorithm with a list containing **duplicate elements**.

#Ex.No:4B Define insertion sort function

Program:

```
def insertion_sort(arr):
        n = len(arr)
       for i in range(1, n):
              key = arr[i]
              j = i – 1
   key# Move elements of arr[0..i-1] that are greater than

       while j >= 0 and arr[j] > key:

                    arr[j + 1] = arr[j]
                    j -= 1
                  # Insert the key at correct position
                  arr[j + 1] = key
       return arr
# Main program
n = int(input("Enter number of elements: "))
arr = []
# Input elements
for i in range(n):
        ele = int(input(f"Enter element {i+1}: "))
        arr.append(ele)
print("Original  list:",  arr) # Call insertion sort function
sorted_arr = insertion_sort(arr) # Display sorted list
print("Sorted list using Insertion Sort:", sorted_arr)
```

Sample Output:

```
Enter number of elements: 5
Enter element 1: 50
Enter element 2: 20
Enter element 3: 40
Enter element 4: 10
Enter element 5: 30
```

Original **list: [**50**,** 20**,** 40**,** 10**,** 30**]**

Sorted **list** using Insertion Sort**: [**10**,** 20**,** 30**,** 40**,** 50**]**

Part A: Viva Questions (Conceptual Q&A)

1. Why does Binary Search work only on sorted lists?
2. What is the time complexity of Binary Search in the best and worst case?
3. Compare Linear Search and Binary Search in terms of performance.

Part B: Worksheet (Fill in the blanks / Short Qs – DIFFERENT from viva)

1. In Binary Search, the middle index is calculated as mid = (_____+ _____) // 2.
2. If the target element is greater than arr[mid], then the new search range becomes from index _____to _____.
3. Suppose the list is [2, 4, 6, 8, 10, 12]. If target = 8, then the element is found at index _____.
4. Binary Search stops when low becomes _____than high.
5. If the list has n = 1,000,000 elements, the maximum number of comparisons required in Binary Search is about _____.

Part C: Practice Exercises

1. Modify the program to perform Binary Search using **recursion**.
2. Write a program that counts how many times an element occurs in a sorted list using Binary Search.
3. Implement Binary Search for a list of **student names (strings)**.
4. Compare execution times of **Linear vs Binary Search** for n = 1000 and n = 100000.
5. Modify the program to return the **first and last occurrence** of the element in case of duplicates.

Result:

The program sorted the given list of numbers correctly using both selection sort and insertion sort algorithms.

EX.NO 5: MERGE SORT AND QUICK SORT

AIM: To implement recursive sorting algorithms (merge sort and quick sort) in Python to efficiently sort a list of elements.

#Ex.No: 5A Define merge sort function

Algorithm:

1. Start
2. Input n (number of elements).
3. Initialize an empty list arr.
4. Read n elements into the list.
5. Define the recursive function merge_sort(arr) that:
    - If length of list > 1:
    - Split list into two halves → left_half and right_half.
    - Recursively call merge_sort on both halves.
    - Merge the two sorted halves back into arr.
    - Else, return (base case).
6. Print the original list.
7. Call merge_sort(arr).
8. Print the sorted list.
9. Stop.

Program:

```
def merge_sort(arr):
        if len(arr) > 1:
            mid = len(arr) // 2  # Find the middle point
            left_half = arr[:mid]
            right_half = arr[mid:]
            # Recursively sort both halves
            merge_sort(left_half)
            merge_sort(right_half)
            # Merge the sorted halves
            i = j = k = 0
            # Copy data to temp arrays left_half[] and right_half[]
```

```
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1
        # Check for any leftover elements
        while i < len(left_half):
            arr[k] = left_half[i]
             i += 1
             k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
# Main program
n = int(input("Enter number of elements: "))
 arr = []
# Input elements
for i in range(n):
    ele = int(input(f"Enter element {i+1}: "))
    arr.append(ele)
print("Original list:", arr)
# Call merge sort function
merge_sort(arr)
# Display sorted list
print("Sorted list using Merge Sort:", arr)
```

Sample Output:

```
Enter number of elements: 6
Enter element 1: 45
```

Enter element 2: 12

Enter element 3: 78

Enter element 4: 34

Enter element 5: 23

Enter element 6: 56

Original **list: [**45, 12, 78, 34, 23, 56**]**

Sorted **list** using Merge Sort**: [**12, 23, 34, 45, 56, 78**]**

Part A: Viva Questions

1. What is the **time complexity** of Merge Sort in best, average, and worst case?
    - O(n log n) in all cases.
2. Why is Merge Sort considered a **divide and conquer** algorithm?
3. Compare Merge Sort with Quick Sort in terms of space and speed.

Part B: Worksheet (Fill in the blanks / Short Qs)

1. Merge Sort divides the array into _____halves recursively.
2. The merging step of Merge Sort has a time complexity of _____.
3. Merge Sort is a _____(stable / unstable) sorting algorithm.
4. If the list has n = 16 elements, the recursion tree will have _____levels.

Part C: Practice Exercises

1. Modify the program to sort the list in **descending order**.
2. Implement Merge Sort for a list of **strings** (e.g., names).
3. Compare Merge Sort with Bubble Sort by timing both algorithms for large input sizes.
4. Write a program to count the number of **comparisons** made during Merge Sort.
5. Modify the program to merge two **already sorted lists** without recursion.

#Ex.No:5B Develop a Python Program for Quick Sort function

Algorithm:

1. Start
2. Input n (number of elements).
3. Initialize an empty list arr.
4. Read n elements into the list.
5. Define a function quick_sort(arr) that:
    - If list length ≤ 1, return the list (base case).
    - Else, choose the first element as pivot.

- Divide remaining elements into two sublists:
  - left: elements smaller than pivot
  - right: elements greater than or equal to pivot
- Recursively sort left and right.
- Return left + [pivot] + right.

6. Print the original list.
7. Call quick_sort(arr) and store in sorted_arr.
8. Print sorted list.
9. Stop.

Program:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr   # Base case: already sorted
    else:
        pivot = arr[0]
        left = [x for x in arr[1:] if x < pivot]    #Elements less than pivot
        right = [x for x in arr[1:] if x >= pivot] # Elements greater than or equal
        return quick_sort(left) + [pivot] + quick_sort(right)
# Main program
n = int(input("Enter number of elements: "))
arr = []
# Input elements
for i in range(n):
    ele = int(input(f"Enter element {i+1}: ")) arr.append(ele)
print("Original list:", arr) # Apply Quick Sort
 sorted_arr = quick_sort(arr) # Output sorted list
print("Sorted list using Quick Sort:", sorted_arr)
```

Sample Output:

```
Enter number of elements: 6
Enter element 1: 45
Enter element 2: 12
Enter element 3: 78
```

Enter element 4: 34

Enter element 5: 23

Enter element 6: 56

Original list: [45, 12, 78, 34, 23, 56]

Sorted list using Quick Sort: [12, 23, 34, 45, 56, 78]

Part A: Viva Questions

1. What is the **time complexity** of Quick Sort in best, average, and worst case?
   - Best: O(n log n), Average: O(n log n), Worst: O(n^2).
2. Why is Quick Sort called an **in-place sorting algorithm**?
3. Compare Quick Sort with Merge Sort in terms of space usage.

Part B: Worksheet (Fill in the blanks / Short Qs — not same as viva)

1. In Quick Sort, the element chosen to divide the array is called the _____.
2. The worst-case time complexity of Quick Sort occurs when the pivot is always the _____ element.
3. Quick Sort uses the _____**and** _____strategy.
4. If the list is [8, 3, 1, 7] and the pivot is 8, then the left sublist is _____.
5. Quick Sort is generally faster than Bubble Sort because its average time complexity is _____instead of _____.

Part C: Practice Exercises

1. Modify the program to sort the list in **descending order**.
2. Implement Quick Sort using the **last element as pivot** instead of the first.
3. Implement Quick Sort iteratively (without recursion).
4. Compare execution time of Quick Sort vs Merge Sort for 1000 random elements.
5. Modify Quick Sort to count the **number of recursive calls** made during sorting.

Result:

The program successfully sorted the input list using merge sort and quick sort and displayed the sorted output.

EX.NO 6: IMPLEMENTING APPLICATIONS USING LISTS AND TUPLES

AIM: To study and implement Python programs that utilize lists and tuples for storing, accessing, updating, slicing, and manipulating data.

#Ex.No:6A IMPLEMENTATION OF LISTS

Algorithm:

1. Start.
2. Create a list my_list with some integer values.
3. Display the original list.
4. Perform slicing operations on the list:
    - Elements from index 1 to 3.
    - Elements from start to index 2.
    - Last 3 elements.
5. Modify a particular element in the list.
6. Delete an element using its index.
7. Access elements using positive and negative indexing.
8. Display the results of each operation.
9. Stop.

Program:

```
# 1. Create a list
my_list = [10, 20, 30, 40, 50, 60]
print("Original List:", my_list)
# 2. Slice the list
print("Sliced [1:4] ->", my_list[1:4])   # Elements from index 1 to 3
print("Sliced [:3] ->", my_list[:3])     # Elements from start to index 2
print("Sliced [-3:] ->", my_list[-3:])   # Last 3 elements

# 3. Change elements
my_list[2] = 300
print("After changing index 2 to 300:", my_list)
```

```
# 4. Delete an element
del my_list[4]                # Delete element at index 4
print("After deleting element at index 4:", my_list)

# 5. Access elements by index print("Element at index 0:",
my_list[0])
print("Element at last index:", my_list[-1])
```

Sample Output:

Original List: [10, 20, 30, 40, 50, 60]

Sliced [1:4] -> [20, 30, 40]

Sliced [:3] -> [10, 20, 30]

Sliced [-3:] -> [40, 50, 60]

After changing index 2 to 300: [10, 20, 300, 40, **50 , 60]**

After deleting element at index 4: [10, 20, 300, 40, 60]

Element at index 0: 10

Element at last index: 60

Part A: Viva Questions

1. What is the difference between **list indexing** and **list slicing**?
2. How are **negative indices** useful in Python lists?
3. What is the difference between del, remove(), and pop() in lists?

Part B: Worksheet (Fill in the blanks / Short Qs — different from viva)

1. In Python, a list is defined using _____brackets.
2. If my_list = [5, 10, 15, 20], then my_list[2] = ____.
3. The last element of a list can be accessed using index _____.
4. The slice my_list[:3] returns the first _____elements.
5. Write the output for:
6. L = [2, 4, 6, 8, 10]
7. print(L[-2])

Part C: Practice Exercises

1. Write a program to create a list of 5 numbers and display their **sum** and **average**.
2. Write a program to find the **maximum and minimum** elements in a list.
3. Write a program to count how many times a particular element occurs in a list.
4. Write a program to reverse a list without using built-in functions.
5. Write a program to check if a given list is a **palindrome**.

#Ex.No:6B IMPLEMENTATION OF TUPLES

Key Properties of a Tuple:

- Tuples are **immutable** → you **cannot change** or **delete individual items**.
- But you **can create, access using index, concatenate, slice**, and **delete the entire tuple**.

Algorithm:

1. Start.
2. Create a tuple my_tuple1with integer values.
3. Display the original tuple.
4. Access elements from the tuple using:
   - Positive indexing
   - Negative indexing
5. Concatenate two tuples.
6. Perform slicing operations on the tuple.
7. Delete the entire tuple using del.
8. Display results for each operation.
9. Stop.

Program:

```
# 1. Create a tuple
my_tuple1 = (10, 20, 30, 40, 50, 60)
my_tuple2 = (10, 20, 30, 40, 50, 60)
print("Original Tuple:", my_tuple1)
# 2. Access elements by index
print("Element at index 0:", my_tuple1[0])
print("Element at last index:", my_tuple1[-1])
```

```
# 3. Concatenate tuples
my_tuple3 = my_tuple1 + my_tuple2 print(my_tuple3)
# 4. Slice the tuple
print("Sliced [1:4] ->", my_tuple1[1:4])# From index 1 to 3
print("Sliced [:3] ->", my_tuple1[:3]) # From start to index 2
print("Sliced [-3:] ->", my_tuple1[-3:])  # Last 3 elements
# 5. Delete entire tuple
del my_tuple1
print("Tuple deleted.")
```

Sample Output:

```
Original Tuple: (10, 20, 30, 40, 50, 60)
Element at index 0: 10
Element at last index: 60
Concatenated Tuple: (10, 20, 30, 40, 50, 60, 10, 20, 30, 40, 50, 60)
Sliced [1:4] -> (20, 30, 40)
Sliced [:3] -> (10, 20, 30)
Sliced [-3:] -> (40, 50, 60)
Tuple deleted.
```

Part A: Viva Questions

1. What is the main difference between **lists and tuples** in Python?
2. Why are tuples considered **immutable**?
3. Can a tuple contain **duplicate elements**? Explain.

Part B: Worksheet (Fill in the blanks / Short Qs – different from viva)

1. A tuple in Python is defined using _____brackets.
2. Tuples are **(mutable / immutable)**.
3. If t = (5, 10, 15, 20), then t[-2] =_____.
4. The expression (1, 2) + (3, 4) results in_____.
5. Write the output:
6. t = (2, 4, 6, 8, 10)
7. print(t[:3])

Part C: Practice Exercises

1. Write a program to create a tuple of **5 student names** and display them.
2. Write a program to find the **maximum and minimum** elements in a tuple.
3. Write a program to convert a **list into a tuple**.
4. Write a program to check if a given element exists in a tuple.
5. Write a program to create a tuple and find its **length**.

Result:

The program effectively performed operations like adding, deleting, updating, and slicing elements in lists and tuples.

EX.NO 7: IMPLEMENTING APPLICATIONS USING SETS AND DICTIONARIES

AIM: To develop Python programs that perform set operations (union, intersection) and dictionary operations (add, update, delete) to manage collections of unique items and key-value pairs.

check if a Given String Is Hetero gram or Not using sets Python sets are unordered collections of unique, immutable objects. They can be created using braces {} or the set() constructor.

#Ex.No:7 IMPLEMENTATION OF SETS

Algorithm:

1. Start the program.
2. Create sets with integers, strings, and mixed data types.
3. Perform the following operations on sets:
   - Add elements using add().
   - Update multiple elements using update().
   - Remove elements using discard().
   - Perform **Union** using | and union().
   - Perform **Intersection** using & and intersection().
4. Display the results after each operation.
5. Stop the program

Program:

```
# create a set of integer type
student_id = {112, 114, 116, 118, 115}
print('Student ID:', student_id)
# create a set of string type
vowel_letters = {'a', 'e', 'i', 'o', 'u'}
print('Vowel Letters:', vowel_letters)
# create a set of mixed data types
mixed_set = {'Hello', 101, -2, 'Bye'}
print('Set of mixed data types:', mixed_set)
```

```
# Add items to a Set
numbers = {21, 34, 54, 12}
print('Initial Set:',numbers)

# using add() method
numbers.add(32)
print('Updated Set:', numbers)

# Update an items to a Set
companies = {'Lacoste', 'Ralph Lauren'}
tech_companies = ['apple', 'google', 'apple']

# using update() method
companies.update(tech_companies)
 print(companies)

# Output: {'google', 'apple', 'Lacoste', 'Ralph Lauren'} # Remove an items to a Set
languages = {'Swift', 'Java', 'Python'}
 print('Initial Set:',languages)

# remove 'Java' from a set
removedValue = languages.discard('Java')
print('Set after remove():', languages)

# Set Operations - Union # first set
A = {1, 3, 5}
# second set B = 0, 2, 4}
# perform union operation using |
 print('Union using |:', A | B)

# perform union operation using union()
print('Union using union():', A.union(B))
```

```
# Set Operations - Intersection # first set
A = {1, 3, 5}
# second set B = {1, 2, 3}
# perform intersection operation using &
print('Intersection using &:', A & B)

# perform intersection operation using intersection()
 print('Intersection using intersection():', A.intersection(B))
```

Sample Output:

```
Student ID: {112, 114, 115, 116, 118}
Vowel Letters: {'a', 'e', 'i', 'o', 'u'}
Set of mixed data types: {'Bye', 101, 'Hello', -2}
Initial Set: {34, 12, 21, 54}
Updated Set: {34, 32, 12, 21, 54}
After update(): {'google', 'apple', 'Lacoste', 'Ralph Lauren'}
Initial Set: {'Python', 'Java', 'Swift'} Set after remove():
{'Python', 'Swift'} Union using |: {0, 1, 2, 3, 4, 5}
Union using union(): {0, 1, 2, 3, 4, 5}
Intersection using &: {1, 3}
Intersection using intersection(): {1, 3}
```

Part A: Fill in the Blanks (Worksheet)

1. The add() method is used to _____a single element to a set.
2. The update() method can add _____elements to a set at once.
3. The _____operation returns elements common to both sets.
4. The discard() method does not raise an _____if the element is not found in the set.

Part B: Viva Questions

1. What is the difference between a **set** and a **list** in Python?
2. Explain the difference between remove() and discard() in sets.
3. Why are sets called **unordered collections**?

 # IMPLEMENTATION OF DICTIONARIES

#Implement Dictionaries - Inventory dictionary

Algorithm:

1. Start the program.
2. Initialize an empty dictionary inventory = {}.
3. Define function add_item(name, quantity) to add a new key-value pair.
   - If item already exists, display a warning.
   - Else, insert item with given quantity.
4. Define function update_item(name, quantity) to modify existing item.
   - If item exists, update its value.
   - Else, display “not found” message.
5. Define function display_inventory() to print all items in the dictionary.
6. Create a menu-driven interface:
   - **1. Add Item**
   - **2. Update Item**
   - **3. Display Inventory**
   - **4. Exit**
7. Based on user choice, call respective functions.
8. Stop when user selects Exit.

Program:

```
inventory = {}

def add_item(name, quantity): if name in
        inventory:
                print(f"Item '{name}' already exists. Use 'update' to change quantity.")
        else:
                inventory[name] = quantity
                print(f"Item'{name}' added with quantity{quantity}.")
```

```
def update_item(name, quantity):
    if name in inventory:
        inventory[name] = quantity
        print(f"Item '{name}' updated to quantity
                {quantity}.")
    else:
        print(f"Item '{name}' not found. Use 'add' to add new item.")

def display_inventory():
    if not inventory:
        print("Inventory is empty.")
        else:
            print("\nCurrent Inventory:")
        for item, qty in inventory.items():
            print(f"{item} : {qty}")

# Menu-driven interface
while True:
    print("\n--- Inventory Menu ---")
    print("1. Add Item")
    print("2. Update Item")
    print("3. Display Inventory")
    print("4. Exit")

    choice = input("Enter your choice (1-4): ")

    if choice == '1':
        name = input("Enter item name: ").lower() quantity =
        int(input("Enter quantity: ")) add_item(name, quantity)

    elif choice == '2':
        name = input("Enter item name to update: ").lower() quantity = int(input("Enter
        new quantity: ")) update_item(name, quantity)

    elif choice == '3': display_inventory()
```

```
    elif choice == '4':
        print("Exiting Inventory Program.") break

    else:
        print("Invalid choice. Please enter 1 to 4.")
```

Sample Output:

--- Inventory Menu ---

1. Add Item

2. Update Item

3. Display Inventory

4. Exit

Enter your choice (1-4): 1 Enter item name: pen

Enter quantity: 20

Item 'pen' added with quantity 20.

--- Inventory Menu ---

1. Add Item

2. Update Item

3. Display Inventory

4. Exit

Enter your choice (1-4): 3 Current Inventory:

pen : 20

--- Inventory Menu ---

1. Add Item

2. Update Item

3. Display Inventory

4. Exit

Enter your choice (1-4): 2 Enter item name to update: pen Enter new quantity: 50

Item 'pen' updated to quantity 50.

--- Inventory Menu ---

1. Add Item

2. Update Item

3. Display Inventory

4. Exit

Enter your choice **(**1-4**):** 4 Exiting Inventory

Program**.**

PartA: Fill in the Blanks (Worksheet)

1. A _____is a collection of key-value pairs in Python.
2. The inventory.items() method returns both _____and _____.
3. In the program, inventory[name] = quantity is used to _____or _____the value of a dictionary key.
4. The dictionary keys must be _____, while values can be any data type.

Part B: Viva Questions:

1. What is the difference between **list** and **dictionary** in Python?
2. Can dictionary keys be duplicated? Why or why not?
3. How does the items() method differ from keys() and values()?

Result:

The program correctly demonstrated set operations (union, intersection, add, remove) and dictionary operations (insert, update, display).

EX.NO 8: IMPLEMENTING PROGRAMS USING FUNCTIONS

AIM: To create reusable Python functions to perform specific tasks and develop modular programs such as arithmetic calculations.

Algorithm – Simple Calculator Using Functions

Step 1: Start

Step 2: Define functions: add(a,b), subtract(a,b), multiply(a,b), divide(a,b)

Step 3: Display the calculator menu (1–Add, 2–Subtract, 3–Multiply, 4–Divide, 5–Exit)

Step 4: Accept user's choice

Step 5:

- If choice is 1–4 → input two numbers
- Call the respective function and display result

Step 6: If choice = 5 → Exit program

Step 7: If choice is invalid → Display error message

Step 8: Repeat until user exits

Step 9: Stop

Program:

```
# Function definitions For Arithmetic Calculations
def add(a, b):
    return a + b
def subtract(a, b): return a - b

def multiply(a, b): return a * b

def divide(a, b): if b != 0:
        return a / b
    else:
        return "Cannot divide by zero"

# Main program loop
while True:
    print("\n--- Simple Calculator ---")
    print("1. Add")
```

```
    print("2. Subtract")
    print("3. Multiply")
    print("4. Divide")
    print("5. Exit")
    choice = input("Enter your choice (1-5): ")
    if choice in ['1', '2', '3', '4']:
        num1 = float(input("Enter first number: "))
        num2 = float(input("Enter second number: "))
        if choice == '1':
            print("Result:", add(num1, num2)) elif choice =='2':
            print("Result:", subtract(num1, num2)) elif choice == '3':
            print("Result:", multiply(num1, num2)) elif choice == '4':
            print("Result:", divide(num1, num2))

    elif choice == '5':
        print("Exiting Calculator. Goodbye!")
        break
    else:
        print("Invalid input. Choose a number from 1 to 5.")
```

Sample Output:

--- Simple Calculator ---

1. Add

2. Subtract

3. Multiply

4. Divide

5. Exit

Enter your choice (1-5): 1 Enter first

number: 10 Enter second number: 5

Result: 15.0

--- Simple Calculator ---

1. Add

2. Subtract

3. Multiply

4. Divide

5. Exit

Enter your choice (1-5): 4 Enter first number: 8 Enter second number: 0

Result: Cannot divide by zero

--- Simple Calculator ---

1. Add

2. Subtract

3. Multiply

4. Divide

5. Exit

Enter your choice (1-5): 5 Exiting Calculator.

Goodbye!

Part A: Viva Questions

Q1. What is the difference between a function definition and a function call?

Function definition describes what the function does. Function call executes it.

Q2. Why do we use return in functions?

return sends back the result of a function to the caller, so it can be stored or reused.

Q3. What is the advantage of writing a calculator using functions instead of writing code directly?

Functions make code reusable, modular, easier to debug, and reduce repetition.

Part B: Worksheet – Fill in the Blanks

1. The keyword to define a function in Python is _____.
2. A function without a return statement returns _____by default.
3. Functions help in writing _____and _____code.
4. Complete the code

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

```
# Function to check even number
def is_even(num):
    if num % 2 == 0:
    return _____
    else:
        return _____
```

Part C: Extra Practice Exercises

1. Write a program with functions to calculate **area of circle, rectangle, and triangle**.
2. Write a function is_prime(n) that checks whether a number is prime.
3. Extend the calculator program to perform **square root** and **percentage**.
4. Write a menu-driven program to manage **student marks**:
   - Add student and marks
   - Update marks
   - Display all students

Result:

The program successfully modularized tasks into functions such as arithmetic operations, palindrome check, and returned correct outputs.

EX.NO 9: IMPLEMENTING PROGRAMS USING STRINGS

AIM: To implement Python programs that manipulate strings for operations like reversing, checking palindromes, counting vowels/consonants, word count, and character frequency analysis.

string operations and functions

palindrome check, reversing a string, counting characters, word occurrences

Aim

To implement Python programs using **string operations and functions** like palindrome check, reversing a string, counting characters, and word occurrences.

Algorithm:

1. Start the program.
2. Get input string from the user.
3. Perform the following string operations:
   - Reverse a string using slicing [::-1].
   - Check if string equals its reverse → palindrome.
   - Count vowels and consonants by checking each character.
   - Count words using split().
   - Count frequency of characters using dictionary.
4. Display the results.
5. Stop.

Program:

```
#Example 1: Reverse a String
def reverse_string(s):
    return s[::-1]

text = input("Enter a string: ") print("Reversed string:",
reverse_string(text))
#Example 2: Check Palindrome
def is_palindrome(s):
    return s == s[::-1]

text = input("Enter a string: ")
```

```
if is_palindrome(text):
        print("Palindrome")
        else:
            print("Not a Palindrome")


#Example 3: Count Vowels and Consonants
def count_vowels_consonants(s):
        vowels = "aeiouAEIOU"
        v = c = 0
        for ch in s:
            if ch.isalpha():
                if ch in vowels: v += 1
                else:
                    c += 1
        return v, c

text = input("Enter a string: ")
v, c = count_vowels_consonants(text)
print("Vowels:", v, " Consonants:", c)


#Example 4: Count Words in a Sentence
def word_count(s):
        words = s.split() return
        len(words)
sentence = input("Enter a sentence: ")
print("Number of words:", word_count(sentence))


#Example 5: Frequency of Each Character
def char_frequency(s):
        freq = {} for ch in
        s:
            freq[ch] = freq.get(ch, 0) + 1
        return freq
```

```
text = input("Enter a string: ")
print("Character frequency:", char_frequency(text))
```

Sample Output:

```
Enter a string: hello Reversed string:
olleh

Enter a string: madam
Palindrome

Enter a string: python Vowels: 1
Consonants: 5

Enter a sentence: Python is fun Number of
words: 3

Enter a string: apple
Character frequency: {'a': 1, 'p': 2, 'l': 1, 'e': 1}
```

Part A: Viva Questions

Q1. What is the difference between split() and join() in Python strings?

split() breaks a string into a list of words, join() combines list items into a string.

Q2. What is slicing in strings?

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

Q3. Are Python strings mutable or immutable?

Strings are **immutable** (cannot be changed once created).

Part B – Fill in the Blanks

1. Strings in Python are enclosed in _____or _____.
2. The function used to find length of a string is _____.
3. "hello".upper() will output _____.

Part C – Complete the Code

```
# Function to check if string contains digit def has_digit(s):
    for ch in s:
        if ch._: return True
    return False
```

Part D: Practice Exercises

1. Write a function to check whether a given string is a **pangram** (contains all alphabets).
2. Write a function to find the **longest word** in a sentence.
3. Write a program to **remove all duplicate characters** from a string.
4. Write a program to count the number of **uppercase, lowercase, digits, and special characters** in a string.

Result:

The program correctly reversed strings, checked palindromes, counted vowels/consonants, counted words, and calculated character frequency.

EX.NO 10: IMPLEMENTING PROGRAMS USING WRITTEN MODULES AND PYTHON STANDARD LIBRARIES

AIM: To use Python standard libraries (pandas, numpy, matplotlib, scipy) to perform data analysis, numerical computations, visualizations, and scientific computations efficiently.

Installation (if necessary):

If these libraries are **not** pre-installed, use pip install pandas numpy matplotlib scipy **in** your terminal **or** command prompt.

Importing:

import pandas **as** pd **(for** data manipulation **with** DataFrames**) import** numpy **as** np **(for** numerical operations **with** arrays**) import** matplotlib.pyplot **as** plt **(for** creating plots**) from** scipy **import** stats **(or** other specific submodules **for**

scientific computing**) Pandas:**

Create DataFrames **from** various data sources **(**CSV**,** Excel**,** dictionaries**).**

Perform data cleaning**,** filtering**,** aggregation**, and**

transformation**.**

Example**:** df **=** pd**.**read_csv**(**'data.csv'**),** df**[**'column'**].**mean**()** **NumPy:**

Create **and** manipulate multi-dimensional arrays **(**ndarray**).**

Perform element-wise operations**,** linear algebra**, and**

statistical computations**.**

Example**:** arr **=** np**.**array**([**1**,** 2**,** 3**]),** np**.sum(**arr**)** **Matplotlib:**

Generate various types of plots **(**line plots**,** scatter plots**,** bar charts**,** histograms**).**

Customize plot aesthetics **(**labels**,** titles**,** colors**).** Example**:** plt**.**plot**(**x**,** y**),** plt**.**title**(**'My Plot'**),** plt**.**show**()**

SciPy:

Access advanced scientific **and** technical computing tools**.** Perform optimization**,** signal processing**,** statistical analysis **and** more**.**

Example**:** scipy**.**stats**.**linregress**(**x**,** y**) for** linear regression**.**

Aim

To implement a Python program using the **pandas** library for performing operations such as reading CSV data, displaying statistics, grouping, and filtering.

Algorithm

1. **Start**
2. Import the pandas module.
3. Define a function read_data(file_path) to read a CSV file into a DataFrame.
4. Define another function analyze_data(data) to perform operations:
   - Display first few rows of data.
   - Display basic statistics of numerical columns.
   - Group data by a categorical column and calculate mean values.
   - Filter rows based on a condition.
5. In the main function:
   - Call read_data() with the file path.
   - Call analyze_data() to process the data.
6. **Stop**

#PROGRAM ON PANDAS:

```
import pandas as pd
# Read data from a CSV filedef read_data(file_path):
data = pd.read_csv(file_path)
return data# Perform basic operations on the datadef analyze_data(data):
# Display the first few rows of the data
print("First few rows of the data:")
print(data.head())
```

```
# Get basic statistics of the data
print("\nBasic statistics of the data:")
print(data.describe())
# Group data by a column and calculate the mean
grouped_data = data.groupby('Category')['Value'].mean()
print("\nAverage value per category:")
print(grouped_data)
# Filter data based on a condition
filtered_data = data[data['Value'] > 50]
print("\nData where Value > 50:")
print(filtered_data)
# Main program
if __name__ == "__main__":
        file_path = "data.csv"
# Replace with the path to your CSV file
        data = read_data(file_path) analyze_data(data)
```

Output:

Part A: Viva Questions

Q1. What is the difference between loc[] and iloc[] in pandas?

loc[] selects rows/columns by **labels**, iloc[] selects by **index positions**.

Q2. What does describe() function return?

It returns basic statistics like count, mean, min, max, std, etc.

Q3. What is the use of groupby() in pandas?

It is used to split data into groups and apply aggregate functions (like mean, sum).

Part B: Worksheet – Fill in the Blanks

1. Pandas is built on top of _____library.
2. The two main data structures in pandas are _____and _____.
3. The function to read a CSV file in pandas is _____.

Part C – Complete the Code

```
import pandas as pd
data = pd._____("data.csv")          # Fill here
print(data._____())                  # Print first 5 rows
```

Part D – Debug the Code

```
import pandas as pd df = pd.DataFrame({"Name": ["A", "B", "C"],"Marks": [85, 90, 75]})
print(df.groupby("Name")["Marks"].mean())  # Fix error
```

Part E: Practice Exercises

1. Write a program to read a CSV file and display the **maximum value** of a column.
2. Write a program to count the number of **unique values** in a column.
3. Write a program to add a new column to a DataFrame that contains the **square** of another column.
4. Write a program to sort the DataFrame by a column in **descending order**.

Implementing programs using written modules and Python Standard Libraries

numpy

Aim:

To implement a Python program using the **NumPy** library for performing basic operations such as mean, sum, standard deviation, mathematical functions, and transpose of 2D arrays.

Algorithm:

1. **Start**
2. Import the numpy module.
3. Define a function basic_numpy_operations() that performs:
   - Create a NumPy array from a Python list.
   - Display the array.
   - Calculate mean, sum, and standard deviation.

- Apply a mathematical function (square).
- Create a 2D array and display it.
- Transpose the 2D array.

4. In the main function, call basic_numpy_operations().
5. **Stop**

#PROGRAM ON NUMPY

```
import numpy as npdef basic_numpy_operations(): # Create a NumPy array from a Python list
arr = np.array([1, 2, 3, 4, 5]) # Display the array
print("Array:", arr)
 # Calculate the mean of the array
mean_value = np.mean(arr)
print("Mean:", mean_value)
 # Calculate the sum of the array
 sum_value = np.sum(arr)
 print("Sum:", sum_value)
 # Calculate the standard deviation of the array
 std_deviation = np.std(arr)
 print("Standard Deviation:", std_deviation) # Apply a
 mathematical function to the array
 squared_values = np.square(arr)
 print("Squared Values:", squared_values)
 # Create a 2D NumPy array
 two_dimensional_arr = np.array([[1, 2, 3], [4, 5, 6]])
 print("2D Array:")
 print(two_dimensional_arr)
 # Transpose the 2D array
 transposed_arr = np.transpose(two_dimensional_arr)
 print("Transposed 2D Array:")
 print(transposed_arr)


if __name__== "__main__":
      basic_numpy_operations()
```

Sample Output:

```
Array: [1 2 3 4 5]
Mean: 3.0
Sum: 15
Standard Deviation: 1.4142135623730951
Squared Values: [ 1  4  9 16 25] 2D Array:
[[1 2 3]
 [4 5 6]]
Transposed 2D Array:
 [[1 4]
  [2 5]
  [3 6]]
```

Part A: Viva Questions

Q1. What is the difference between a Python list and a NumPy array?

A list can hold mixed data types, while a NumPy array is optimized for numerical computations and stores homogeneous data.

Q2. What does np.mean() function return?

It returns the average value of elements in an array.

Q3. What is the use of np.transpose()?

It swaps the rows and columns of a 2D array.

Part B: Worksheet – Fill in the Blanks

1. NumPy stands for _____.
2. The function used to calculate the standard deviation in NumPy is _____.
3. The shape of a 2D array can be obtained using the attribute _____.

Part C – Complete the Code

```
import numpy as np
arr = np.array([10, 20, 30])
print("Mean:", np._____(arr))          # Fill here
print("Sum:", np._____(arr))           # Fill here
```

Part D – Debug the Code

```
import numpy as np
matrix = np.array([[1, 2], [3, 4]]) print("Original Matrix:\n", matrix)
print("Transpose:\n", np.transpose(matrix)) # Fix error if any
```

Part E: Practice Exercises

1. Create a NumPy array of numbers from **1 to 20** and display only even numbers.
2. Create a 3x3 NumPy array with random values and calculate its **determinant**.
3. Write a program to create a NumPy array of 10 zeros and reshape it into a **2x5 matrix**.
4. Write a program to generate two arrays and perform **element-wise addition, subtraction, and multiplication**.

Implementing programs using written modules and Python Standard Libraries Matplotlib

Aim:

To implement a Python program using **Matplotlib** library to plot a simple line graph with labels and title.

Algorithm:

1. **Start**
2. Import matplotlib.pyplot as plt.
3. Define a function simple_line_plot() that:
   - Creates data for x-axis and y-axis.
   - Plots a line graph.
   - Adds labels for x-axis and y-axis.
   - Adds a title to the graph.
   - Displays the graph using plt.show().
4. In the main function, call simple_line_plot().
5. **Stop**

#PROGRAM ON MATPLOTLIB:

```
import matplotlib.pyplot as plt def
simple_line_plot():
# Data for the x-axis (time or any other independent variable)
```

```
x = [1, 2, 3, 4, 5]
# Data for the y-axis (dependent variable)
y = [2, 4, 6, 8, 10]
# Create a line plot
 plt.plot(x, y)
# Add labels to the x and y axes
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
# Add a title to the plot
plt.title('Simple Line Plot') # Show the plot
plt.show()
if __name__== "__main__": simple_line_plot()
# Calculate the mean of the array
 mean_value = np.mean(arr)
print("Mean:", mean_value)
# Calculate the sum of the array
sum_value = np.sum(arr)
print("Sum:", sum_value)
# Calculate the standard deviation of the array
 std_deviation = np.std(arr)
print("Standard Deviation:", std_deviation) # Apply a mathematical function to the array
squared_values = np.square(arr)
print("Squared Values:", squared_values)
# Create a 2D NumPy array
two_dimensional_arr = np.array([[1, 2, 3], [4, 5, 6]])
print("2D Array:")
print(two_dimensional_arr)
# Transpose the 2D array
transposed_arr = np.transpose(two_dimensional_arr)
print("Transposed 2D Array:")
print(transposed_arr)
if __name__== "__main__": basic_numpy_operations()
```

Output:

Part A: Viva Questions

Q1. What is the purpose of plt.show() in Matplotlib?

It displays the graph on the screen.

Q2. What is the difference between plt.plot() and plt.scatter()?

plt.plot() is used for line plots, while plt.scatter() is used for scatter plots.

Q3. Can we plot multiple lines on the same graph in Matplotlib?

Yes, by calling plt.plot() multiple times before plt.show().

Part B: Worksheet – Fill in the Blanks

1. The function used to display a graph in Matplotlib is _____.
2. To set the label for the x-axis, we use _____.
3. The function used to give a title to the plot is _____.

Part C – Complete the Code

```
import matplotlib.pyplot as plt
x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]
plt.plot(x, y)
plt.___('X-axis')   # Fill here
plt.___('Y-axis')   # Fill here
plt._('My Plot') # Fill here
plt.show()
```

Part D – Debug the Code

```
import matplotlib.pyplot as plt
x = [1, 2, 3]
y = [2, 4, 6]
plt.plot(x, y)
plt.lable('X-axis')                    # Fix spelling mistake
plt.ylabel('Y-axis')
plt.title('Line Plot') plt.show()
```

Part E: Practice Exercises

1. Plot a line graph for the function **y = x²** for x values from -10 to 10.
2. Draw a line graph with **two lines**:

- y = 2x
- y = 3x

  (Give different colors and labels for both lines).

3. Create a bar chart showing marks of 5 subjects.
4. Plot a sine wave using numpy.linspace() and plt.plot().

Implementing programs using written modules and Python Standard Libraries scipy

Aim:

To implement a Python program using **SciPy** library to perform optimization (finding the minimum of a function).

Algorithm:

1. **Start**
2. Import **NumPy** and optimize from **SciPy**.
3. Define an **objective function** to minimize (e.g., $f(x) = x^2 + 2x + 1$).
4. Provide an **initial guess** for the optimization.
5. Use optimize.minimize() to find the minimum value of the function.
6. Print the **optimal x value** and the **minimum function value**.
7. Call the function from the main block.
8. **Stop**

#PROGRAM ON SCIPY :

```
import numpy as np
from scipy import optimize

def scipy_example():
# Define a function to optimize (e.g., find the minimum value)
def objective_function(x):
    return x**2 + 2*x + 1
  # Initial guess for the optimization
   initial_guess = 0
  # Use the minimize function from scipy.optimize to find the minimum
  result = optimize.minimize(objective_function, initial_guess)
```

```
# Extract the optimal value
optimal_x = result.x[0]
optimal_value = result.fun print(f"Optimal x: {optimal_x}")
print(f"Optimal value: {optimal_value}")
if __name__== "__main__":
    scipy_example()
```

Output:



Part A: Viva Questions

Q1. What is SciPy used for in Python?

It is used for scientific and technical computing (optimization, integration, statistics, etc.).

Q2. What does optimize.minimize() do?

It finds the minimum value of a function.

Q3. What is the difference between NumPy and SciPy?

NumPy handles arrays and basic numerical operations, while SciPy builds on NumPy and provides advanced scientific algorithms like optimization and integration.

Part B: Worksheet – Fill in the Blanks

1. The SciPy module used for optimization is _____.
2. The method used to minimize a function is _____.
3. SciPy is built on top of _____.

Part C – Complete the Code

```
from scipy import optimize
def f(x):
    return x**2 + 4*x + 4
result = optimize._____(f, 0)  # Fill here
print("Optimal x:", result.x[0])
print("Optimal value:", result.fun)
```

Part D – Debug the Code

```
from scipy import optimize
def obj(x):
        return x**2 - 6*x + 9

# Wrong function call
result = optimize.minimum(obj, 2)                    # Fix here
print(result)
```

Part E: Practice Exercises

1. Use SciPy to minimize the function: **$f(x) = x^2 - 4x + 5$**.
2. Find the minimum of the function **$f(x) = \sin(x) + x^2$**.
3. Solve an equation using optimize.root() for **$x^2 - 4 = 0$**.
4. Use scipy.integrate.quad() to integrate **$f(x) = x^2$** between 0 and 2.

Result:

The program successfully used pandas, numpy, matplotlib, and scipy to perform data analysis, computations, visualizations, and optimization tasks.

EX.NO 11: IMPLEMENTING REAL-TIME/ TECHNICAL APPLICATIONS USING FILE HANDLING

AIM: To develop Python programs that perform file handling operations such as creating, reading, copying, and updating files, and processing data to count words or find the longest word.

Algorithm:

Part A: Create an Employee List

1. Start the program.
2. Open the file in **append mode**.
3. Accept roll no, name, and marks from the user.
4. Write the details into the file.
5. Close the file.
6. Open the file in **read mode** and display all records.
7. Stop.

Part B: Copy from one file to another

1. Open the source file in **read mode**.
2. Open the destination file in **write mode**.
3. Read line by line and write into the destination.
4. Close both files.
5. Display a success message.

Part C: Word Count in a File

1. Open the file in **read mode**.
2. Read the entire content as a string.
3. Split the text into a list of words.
4. Count the total number of words.
5. Display the count.

Part D: Longest Word in a File

1. Use the word list obtained from word count.
2. Apply max(words, key=len) to find the longest word.
3. Display the word and its length.

Program:

```
# File to create an Employee List def
add_student(filename):
    with open(filename, "a") as f:
        roll = input("Enter Roll No: ")
        name = input("Enter Name: ")
        marks = input("Enter Marks: ")
        f.write(f"{roll},{name},{marks}\n")
        print("Student added successfully!\n")
def view_students(filename):
    with open(filename, "r") as f: print("\n--- Student
        Records ---")
    for line in f:
            roll, name, marks = line.strip().split(",")
            print(f"Roll: {roll}, Name: {name}, Marks:{marks}")

filename = "students.txt"
while True:
    print("\n1. Add Student  2. View Students  3. Exit")
    ch = int(input("Enter choice: "))
    if ch == 1:
        add_student(filename)
        elif ch == 2:
            _students(filename)
        else:
            break
# Implementing Copy from one file to another

def copy_file(src, dest):
    """Copy content from one file to another"""
    with open(src, "r") as f1, open(dest, "w") as f2:
        for line in f1:
            f2.write(line)
    print(f"Contents copied from {src} to {dest}\n")
    # Implementing Count words in a file
```

```
def word_count(filename):
    """Count total words in a file"""
    with open(filename, "r") as f:
        text = f.read() words =
        text.split()
        print(f"Total words in {filename}: {len(words)}")
        return words

# Implementing find the longest word in a file

def longest_word(words):
    """Find the longest word from a list of words"""
    lw = max(words, key=len)
    print("Longest word:", lw, " (Length:", len(lw), ")")

# ---------- Main Program ----------
src_file = "input.txt" dest_file = "output.txt"

# 1. Copy file
copy_file(src_file,dest_file)

# 2. Count words
words = word_count(src_file)

# 3. Find longest word
longest_word(words)
```

Sample Output:

```
1. Add Student  2. View Students  3. Exit
Enter choice: 1 Enter Roll
No: 101 Enter Name: John
Enter Marks: 87
Student added successfully!
```

1. Add Student 2. View Students 3. Exit

Enter choice: 2

--- Student Records ---

Roll: 101, Name: John, Marks: 87

Contents copied from input.txt to output.txt

Total words in input.txt: 6

Longest word: programming (Length: 11)

Part A: Viva Questions

1. What is the difference between **read mode (r)** and **append mode (a)** in file handling?
2. How can you copy one file's contents to another in Python?
3. What function is used to find the **longest word** from a list of words?

Part B: Worksheet - Fill in the Blanks

1. The function to read all lines from a file is _____.
2. Opening a file in "w" mode will _____ the file contents if it already exists.
3. The method used to split text into words is _____.

Part C: Code Completion

```
# Fill the missing parts

# Open file in write mode f =
open("data.txt", "w")
f._____("Hello World")
f.close()

# Count words
with open("data.txt", "r") as f:
    text = f.read()
    words = text._____()
    print("Word count:", len(words))
```

Part D: Practice Exercises

1. Write a program to **count the number of lines** in a file.
2. Write a program to **reverse the contents** of a file.
3. Write a program to **copy only unique words** from one file to another.
4. Extend the employee list program to **search an employee by roll number**.

Result:

The program correctly created, read, copied, and processed files, including word count and finding the longest word in the file.

EX.NO 12: IMPLEMENTING REAL-TIME/ TECHNICAL APPLICATIONS USING EXCEPTION HANDLING

AIM: To implement Python programs that handle runtime errors and exceptions gracefully using try-except blocks, including custom exceptions for applications like ATM withdrawals, safe division, and file handling.

Algorithm:

1. **Start** the program.
2. Define a **custom exception** InsufficientBalanceError.
3. Implement the following functions:
   - **safe_division()** – Perform division with try-except handling for ZeroDivisionError and ValueError.
   - **file_reader()** – Open a file and read contents, handle FileNotFoundError.
   - **age_checker()** – Validate age input and raise an error if negative.
   - **atm_withdrawal()** – Simulate ATM withdrawal and raise InsufficientBalanceError if amount > balance.
4. Display a **menu** with 5 options:
   1. Safe Division
   2. File Reader
   3. Age Checker
   4. ATM Withdrawal
   5. Exit
5. Based on user choice, call the respective function.
6. Continue until user chooses Exit.
7. **Stop** the program.

Program:

```
# Custom exception for insufficient balance
class InsufficientBalanceError(Exception): pass
def safe_division():
    try:
            num1 = int(input("\nEnter numerator: "))
```

```
            num2 = int(input("Enter denominator: ")) result = num1 /
            num2
            print("Division Result =", result)
            except ZeroDivisionError:
                print("Error: Cannot divide by zero!")
            except ValueError:
                print("Error: Please enter valid integers.")

def file_reader():
    try:
            filename = input("\nEnter filename to open: ")
            with open(filename, "r") as f:
                print("\nFile Contents:\n", f.read())
    except FileNotFoundError:
            print("Error: File not found!")
    except Exception as e:
            print("Unexpected error:", e)

def age_checker():
    try:
            age = int(input("\nEnter your age: "))
            if age < 0:
                raise ValueError("Age cannot be negative!")
                print("Your age is:", age)
    except ValueError as e:
            print("Invalid input:", e)

def atm_withdrawal():
    balance = 5000
    try:
        amount = int(input("\nEnter withdrawal amount: "))
        if amount > balance:
                raise InsufficientBalanceError("Insufficient balance in account!")
            balance -= amount
```

```
            print("Withdrawal successful. Remaining balance:", balance)
        except InsufficientBalanceError as e:
            print("Transaction Failed:", e)
        except ValueError:
            print("Error: Please enter a valid amount.")

# ---------- Main Menu ----------
while True:
    print("\n--- Exception Handling Applications ---")
    print("1. Safe Division")
    print("2. File Reader")
    print("3. Age Checker")
    print("4. ATM Withdrawal")
    print("5. Exit")

    choice = input("Enter choice: ")
    if choice == "1":
        safe_division()
    elif choice == "2":
        file_reader()
    elif choice == "3":
        age_checker()
    elif choice == "4":
        atm_withdrawal()
    elif choice == "5":
        print("Exiting program... Goodbye!")
        break
    else:
        print("Invalid choice! Please try again.")
```

Sample Output:

```
--- Exception Handling Applications ---
1. Safe Division
2. File Reader
```

3. Age Checker

4. ATM Withdrawal

5. Exit

Enter choice: 1
Enter numerator: 10
Enter denominator: 0

Error: Cannot divide by zero!

--- Exception Handling Applications ---

1. Safe Division

2. File Reader

3. Age Checker

4. ATM Withdrawal

5. Exit

Enter choice: 2

Enter filename to open: sample.txt

Error: File not found!

--- Exception Handling Applications ---

1. Safe Division

2. File Reader

3. Age Checker

4. ATM Withdrawal

5. Exit

Enter choice: 3

Enter your age: -5

Invalid input: Age cannot be negative!

--- Exception Handling Applications ---

1. Safe Division

2. File Reader

3. Age Checker

4. ATM Withdrawal

5. Exit

Enter choice: 4

Enter withdrawal amount: 6000

```
Transaction Failed: Insufficient balance in account!

Enter choice: 5
Exiting program... Goodbye!
```

Part A: Viva Questions

1. What is the difference between **built-in exceptions** and **user-defined exceptions** in Python?
2. Why do we use the **finally** block in exception handling?
3. How does exception handling improve the reliability of a program?

Part B: Practice Exercises

1. Write a program to handle multiple exceptions when converting a string to integer (e.g., "abc", "12.5").
2. Implement a **Bank Account class** with deposit and withdrawal methods using exception handling for insufficient funds.
3. Modify the **file_reader()** function to also count and display the number of lines in the file.

Part C: Code Snippet Completion

Q1. Safe Division – Fill in the missing parts:

```
def safe_division():
    try:
        num1 = int(input("Enter numerator: "))
        num2 = int(input("Enter denominator: "))
        result = num1 / num2
        print("Division Result =", result)
    except _____:
        print("Error: Cannot divide by zero!")
    except _____:
        print("Error: Please enter valid integers.")
```

Q2. ATM Withdrawal – Fill in the missing parts:

```
class InsufficientBalanceError(_____):
    pass
def atm_withdrawal():
    balance = 5000 try:
        amount = int(input("Enter withdrawal amount: "))
```

```
            if amount > balance:
                raise InsufficientBalanceError("Insufficient balance in account!")
            balance -= amount
            print("Withdrawal successful. Remaining balance:", balance)
        except _____as e:
            print("Transaction Failed:", e)
        except _____:
            print("Error: Please enter a valid amount.")
```

Result:

The program handled runtime errors gracefully, including division by zero, invalid input, file not found, and custom exceptions for ATM withdrawal.

EX.NO 13: CREATING AND INSTANTIATING CLASSES

AIM: To develop Python programs that implement object-oriented programming concepts by creating classes, instantiating objects, and performing operations such as employee payroll calculation with departments.

Algorithm:

1. Start the program.
2. Define a class **Employee** with:
   - Data members: emp_id, name, dept, basic_pay, da, hra, gross_pay, pf, net_pay.
   - Method __init__() to initialize employee details.
   - Method calculate_payroll() to compute salary components:
     - DA = 80% of Basic Pay
     - HRA = 15% of Basic Pay
     - Gross Pay = Basic Pay + DA + HRA
     - PF = 12% of Basic Pay
     - Net Pay = Gross Pay – PF
   - Method display() to print employee details with payroll.
3. In the main program:
   - Get employee details from the user.
   - Create an object of the **Employee** class.
   - Call calculate_payroll() to calculate salary.
   - Call display() to print payroll details.
4. Stop the program.

Program:

```
class Employee:
    def _init_(self, emp_id, name, dept, basic_pay): self.emp_id = emp_id
        self.name = name self.dept = dept
        self.basic_pay = basic_pay self.da = 0.0
        self.hra = 0.0
        self.gross_pay = 0.0
        self.pf = 0.0
        self.net_pay = 0.0
```

```
    def calculate_payroll(self):
        # Salary components (assumed %)
        self.da = 0.80 * self.basic_pay    # Dearness Allowance # House Rent Allowance
        self.hra = 0.15 * self.basic_pay
        self.gross_pay = self.basic_pay + self.da + self.hra self.pf = 0.12 *
        self.basic_pay                                    # Provident Fund
        self.net_pay = self.gross_pay - self.pf
    def display(self):
        print("\n--- Employee Payroll ---")
        print(f"ID                  : {self.emp_id}")
        print(f"Name                : {self.name}")
        print(f"Department: {self.dept}")
        print(f"Basic Pay : {self.basic_pay}")
        print(f"DA                  : {self.da}")
        print(f"HRA                 : {self.hra}")
        print(f"Gross Pay : {self.gross_pay}")
        print(f"PF                  : {self.pf}")
        print(f"Net Pay             : {self.net_pay}")

# -------- Main Program --------
emp_id = input("Enter Employee ID: ")
name = input("Enter Employee Name: ")
dept = input("Enter Department: ")
basic = float(input("Enter Basic Pay: "))

emp = Employee(emp_id, name, dept, basic)                    #instantiate object
emp.calculate_payroll()
emp.display()
```

Sample Output:

```
Enter Employee ID: E102 Enter
Employee Name: MANIRAJ Enter
Department: IT
Enter Basic Pay: 25000
```

--- Employee Payroll ---ID :

E102

Name : Rahul

Department: IT Basic Pay :

25000.0 DA : 20000.0

HRA : 3750.0

Gross Pay : 48750.0 PF :

3000.0 Net Pay : 45750.0

Part A: Viva Questions

1. What is the difference between a **class** and an **object** in Python?
2. Why do we use the __init__() constructor method in classes?
3. Explain the difference between **gross pay** and **net pay** in payroll systems.

Part B: Worksheet –Fill in the blanks

1. The function used to initialize objects in a class is _____.
2. In payroll, **PF** stands for _____.
3. **DA** in employee salary means _____.
4. The keyword used to create an object in Python is _____.
5. The formula for **Gross Pay** = Basic Pay + _____+ _____.

Part C: Practice Exercises

1. Modify the program to calculate **income tax (10% of Gross Pay)** and deduct it from Net Pay.
2. Create a class Department that stores department details and link it with Employee.
3. Extend the program to input and display **details of 5 employees**.

Result:

The program successfully implemented class creation, object instantiation, and payroll calculations for employees including DA, HRA, PF, and Net Pay.

14. TOPIC BEYOND THE SYLLABUS: AUTOMATION & SCRIPTING IN PYTHON

Library: requests+ BeautifulSoup

Purpose: Extract information from websites automatically.

Sample Program – Scrape Headlines from a News Website

```
import requests
from bs4 import BeautifulSoup # URL to
scrape
url = "https://www.bbc.com/news" # Send HTTP
request
response = requests.get(url) # Parse
HTML content
soup = BeautifulSoup(response.text, "html.parser")
# Find all headlines
headlines = soup.find_all('h3')  # Assuming headlines are in
<h3> tags
print("Top Headlines:")
for i, headline in enumerate(headlines[:5], 1): print(f"{i}. {headline.text.strip()}")
```

Sample Output:

Top Headlines:

1. Global economic growth slows
2. New tech innovation unveiled
3. Climate change report released
4. Sports championship results
5. Health experts give advice

1. Excel Automation

Library: pandas+ openpyxl

Purpose: Read, write, and update Excel files automatically.

Sample Program – Create and Update Excel File

```
import pandas as pd

# Create a DataFrame data = {
        "Name": ["Alice", "Bob", "Charlie"], "Age": [25, 30, 22],
        "Department": ["HR", "IT", "Finance"]
}
df = pd.DataFrame(data)

# Write to Excel df.to_excel("employees.xlsx",index=False)
print("Excel file created.")

# Read Excel and add a new row
df_new = pd.read_excel("employees.xlsx")
new_employee = {"Name": "David", "Age": 28, "Department": "Marketing"}
df_new = df_new.append(new_employee, ignore_index=True)

# Save updated Excel df_new.to_excel("employees.xlsx",index=False)
print("Excel file updated.")
```

Sample Output:

Excel file employees.xlsxcreated and updated with the new employee.

2. PDF Automation

Library: PyPDF2

Purpose: Read, merge, or extract content from PDF files.

Sample Program – Merge Two PDFs

```
from PyPDF2 import PdfMerger

pdfs = ["file1.pdf", "file2.pdf"] merger = PdfMerger()

for pdf in pdfs: merger.append(pdf)
```

```
merger.write("merged.pdf") merger.close()
print("PDF files merged successfully.")
```

Sample Output:

merged.pdf is created with content from file1.pdf and file2.pdf.

3. Task Scheduling

Library: schedule

Purpose: Automate repetitive tasks at scheduled intervals.

Sample Program – Print Message Every 10 Seconds

```
import schedule
import time

def job():
    print("Task executed!")


# Schedule the job every 10 seconds
schedule.every(10).seconds.do(job)
while True:
    schedule.run_pending() time.sleep(1)
```

Sample Output:

Task executed! Task executed! Task executed!

... (every 10 seconds)

Part A: Practice Exercises

1. Scrape the top 10 trending repositories from GitHub and save them to Excel.
2. Merge multiple PDF reports from a folder into a single PDF.
3. Schedule a script to check the weather every morning and send an email alert.

4. Extract tables from a PDF and store them in CSV.
5. Automatically update stock prices in an Excel file every hour using web scraping.

Recommended Practice Exercises

1. Implement a **custom decorator** to measure function execution time.
2. Create a **multi-threaded file downloader**.
3. Develop a **Flask API** that returns JSON data from a CSV file.
4. Build a **simple ML model** using scikit-learn and visualize results.
5. Scrape a **website table** and store the data in Excel using Python.

15. SAMPLE VIVA QUESTIONS

1. Basics of Python

1. What is Python, and what are its key features?
2. How is Python different from other programming languages like Java or C++?
3. Explain the concept of an interpreter in Python.
4. What are Python's main data types?
5. What is the difference between is and == in Python?

2. Python Syntax and Semantics

6. How do you define a variable in Python?
7. What are Python keywords, and how many are there in the current version?
8. Explain the significance of indentation in Python.
9. What are comments in Python, and how are they written?
10. Can Python have multiline comments? If yes, how?

3. Control Flow

11. What is the difference between if, elif, and else in Python?
12. How does a while loop differ from a for loop in Python?
13. What is the purpose of the break and continue statements?
14. How can you iterate over a range of numbers in Python?
15. Explain list comprehensions with an example.

4. Functions and Modules

16. What are functions, and why are they used in Python?
17. How do you define and call a function in Python?
18. What are default arguments and keyword arguments in Python functions?
19. How do you import a module in Python? Give an example.
20. What is the difference between import and from ... import?

5. File Handling

21. How do you open a file in Python?
22. Explain the difference between reading (r), writing (w), and appending (a) modes.
23. What is the with statement in file handling?
24. How can you read a file line by line in Python?
25. What happens if you try to read a file that doesn't exist?

6. Object-Oriented Programming (OOP)

26. What is a class in Python, and how is it different from an object?
27. What is the significance of the self keyword in Python classes?
28. How do you create an instance of a class in Python?
29. What are instance variables and class variables in Python?
30. Explain inheritance and polymorphism in Python.

7. Exception Handling

31. What are exceptions in Python?
32. How is exception handling implemented in Python?
33. What is the difference between try-except and try-finally?
34. What is the use of the raise keyword in Python?
35. How can you create custom exceptions in Python?

8. Python Collections

36. What is the difference between a list, tuple, set, and dictionary in Python?
37. How can you access elements in a tuple?
38. What is the significance of the keys() and values() methods in dictionaries?
39. How do you remove duplicate elements from a list in Python?
40. How is slicing implemented in Python lists?

9. Advanced Topics

41. What are decorators in Python?
42. How does Python manage memory?
43. Explain the difference between mutable and immutable objects in Python.
44. What is the difference between shallow copy and deep copy?
45. How does Python handle multi-threading?

10. Libraries and Frameworks

46. What are some commonly used Python libraries for data analysis?
47. Explain the difference between NumPy and Pandas.
48. How is Django different from Flask?
49. What is the purpose of Matplotlib in Python?
50. How can you install third-party libraries in Python?

*****