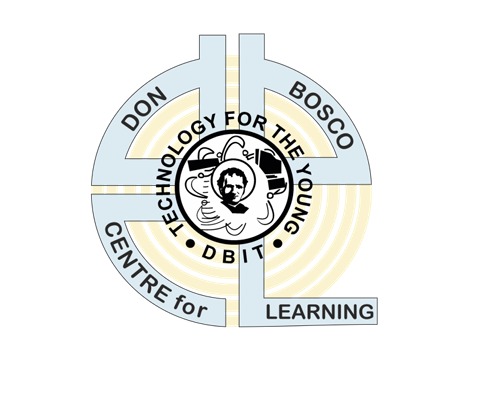
**Python Lab Journal**

Name: Keshav Kuldip Prajapati Roll No. 44

Course/Lab: ITL404

Semester: IV

Academic Year: 2024-2025

**Subject In-charge: Prof. Shiv Negi**

Department of Information Technology

The Bombay Salesian Society’s

Don Bosco Institute of Technology, Mumbai 400070

INDEX PAGE

|  |  |  |
| --- | --- | --- |
| Sr.No | Title | Page No. |
| 1 | Print the grades of students after accepting the marks for 5 subjects from the user. | 4-5 |
| 2 | Accept the range from the user and then display all prime numbers between the given range. | 6-7 |
| 3 | Print an n digit number in reverse order | 8-8 |
| 4 | Display, i) a pattern formed with numbers and, ii) a pattern formed with ‘\*’ using nested looping. | 9-11 |
| 5 | Display the count of elements of different data types present in a LIST. | 12-13 |
| 6 | Check for a given value in the LIST; display total count of occurrences along with the index positions of each occurrence. | 14-15 |
| 7 | Perform sorting of LIST elements; Press 1 for ascending order and, Press 2 for descending order. | 16-17 |
| 8 | Add elements of List2 in List1, then display the updated List | 18-19 |
| 9 | Demonstrate LIST comprehensions using two examples. | 20-20 |
| 10 | Display count of vowels, consonants, blank spaces, special symbols and digits in a given STRING. | 21-22 |
| 11 | Calculate library fine based on 2 conditions, a) Book return date and b) Book condition | 23-25 |
| 12 | Calculate and display the Net monthly salary for the two categories of employees, permanent and temporary, based on following inputs from the user – monthly salary, hourly rate, present/absent days’ count, incentives/bonus, income tax, etc. | 26-29 |
| 13 | Display Fibonacci series elements (n) using recursive method. | 30-31 |
| 14 | Demonstrate passing and returning a List to/from a user defined function. | 32-33 |
| 15 | Create a module having 3 functions – factorial (), primeNumber () and powNumber (). Import this module in the main menu driven program to access all the functions (accept input from the user). | 34-37 |
| 16 | Write a program to demonstrate multiple exceptions handling, specifically, NameError, IndexError, and ZeroDivisionError. | 38-40 |
| 17 | Write a program to implement multiple exceptions handling such as ValueError, KeyError, PermissionError, General exception within a standard LOGIN process (Login successful, User doesn’t exist, Incorrect password, Too many attempts, etc.). | 41-45 |
| 18 | Write an object oriented program to demonstrate working of default and parameterized constructors. | 46-48 |
| 19 | Write an object oriented menu driven program to perform banking operations (New account, Deposit, Withdraw, Balance, Show all, Exit). | 49-54 |
| 20 | Write object oriented program to implement, i) Single level inheritance, and ii) Multilevel inheritance by considering appropriate real life scenarios (use super(), \_\_init\_\_, \_\_str\_\_, and \_\_name\_\_ ). | 55-58 |
| 21 | Design a working interface for the login and registration process with proper form validations and database connection using tkinter and SQLite | 59-65 |
| 22 | Perform CRUD database operations using a menu driven program (User Management - Add, Show, Delete, Update and Search). | 66-74 |
| 23 | Implement a program to accept file content from the user and then display/read file content using 3 different approaches. | 75-78 |
| 24 | Implement a program that reads a large log file (e.g., server.txt), finds ‘error’ word and counts the occurrences and keeps track of line number for each occurrence, and finally saves extracted information into a new file called error.txt. | 79-81 |
| 25 | Implement a program to demonstrate working of multiple-threads for a specific case scenario (food ordering, airport luggage management, ATM, etc.) | 82-86 |
| 26 | Implement a multithreading program for banking scenario to demonstrate RACE condition – 1) without and, ii) with LOCK | 87-90 |
| 27 | Select an appropriate dataset (Kaggle) and perform EDA to answer/perform following. (Jupyter Notebook). | 91-97 |

Project Link: <https://github.com/kpcode11/PythonLabQuestionCodes.git>

**Python Lab Journal**

**Class: SEIT Semester: IV Academic Year: 2024-2025**

**Basic Programs**

[1]. Basic Programming Elements (Data types, print (), input (), operators, if-else, looping, etc.)

……………………………………………………………………………………

1. **Title: Print the grades of students after accepting the marks for 5 subjects from the user.**

…………………………………………………………………………………………………..

1. **Theory:**

The program calculates the average marks of a student based on marks entered for multiple subjects and assigns a grade according to the average.

**Programming Elements Used:**

* Lists: To store marks.
* Input & Typecasting: input() and int() to get user data.
* Loops: for loop to collect subject marks.
* List Functions: append() and sum() for storing and totaling marks.
* Arithmetic: Average calculated using division.
* Conditionals: if-elif-else for grade assignment.
* Formatted Output: f-strings for clean display.

…………………………………………………………………………………………………..

1. **Source Code:**

marks = []

sub = int(input("Enter the number of subjects: "))

for i in range(sub):

mark = int(input(f"Enter the marks of subject {i+1}: "))

marks.append(mark)

average = sum(marks) / sub

if average >= 90:

print("Grade A")

elif average >= 80:

print("Grade B")

elif average >= 70:

print("Grade C")

elif average >= 60:

print("Grade D")

elif average >= 50:

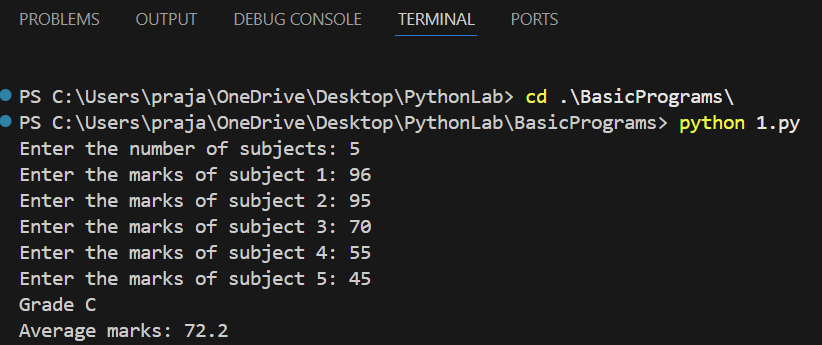
print("Grade E")

else:

print("Grade F")

print(f"Average marks: {average}")

…………………………………………………………………………………………………..

1. **Sample Input/Output:**

…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Accept the range from the user and then display all prime numbers between the given range.**

…………………………………………………………………………………………………..**2.** **Theory:**

**Concept Implemented:**

This program checks and displays all **prime numbers** within a given range provided by the user. A prime number is a number greater than 1 that is divisible only by 1 and itself.

**Programming Elements Used:**

* **Functions:** The program uses a function is\_prime() to determine whether a number is prime. This improves code reusability and clarity.
* **Mathematical Logic:** It uses an efficient approach by checking for factors only up to the square root of the number to reduce unnecessary computations.
* **Loops:** A for loop is used to iterate through the given range of numbers.
* **Conditionals:** if-else statements handle logic like checking if the number is less than 2, if it's divisible by 2, and other potential divisors.
* **Input Handling:** The program takes two numbers as input for the range and swaps them if the start is greater than the end.
* **List Usage:** All found prime numbers are stored in a list and printed at once using unpacking.

This program demonstrates the **use of modular programming** through functions and highlights how **mathematical optimization** (like checking up to √n) can make algorithms more efficient. It also shows how to build robust programs that handle user input dynamically and avoid errors due to input order.

…………………………………………………………………………………………………..**3.** **Source Code:**

# Function to check if a number is prime

def is\_prime(num):

    if num <= 1:

        return False

    if num == 2:

        return True

    if num % 2 == 0:

        return False

    for i in range(3, int(num\*\*0.5) + 1, 2):

        if num % i == 0:

            return False

    return True

# Get range from user

start = int(input("Enter the start of the range: "))

end = int(input("Enter the end of the range: "))

# Swap if start is greater than end

if start > end:

    start, end = end, start

print(f"Prime numbers between {start} and {end} are:")

# Find and display prime numbers in the range

prime\_numbers = []

for num in range(start, end + 1):

    if is\_prime(num):

        prime\_numbers.append(num)

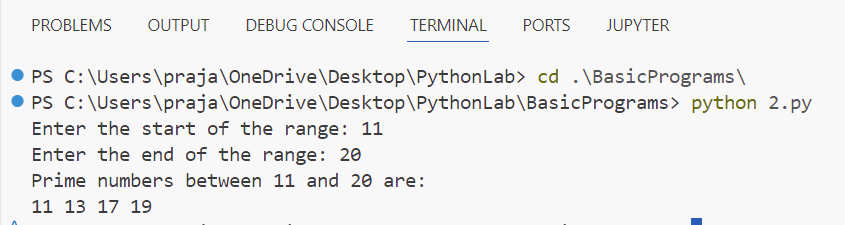
if prime\_numbers:

    print(\*prime\_numbers)

else:

    print("No prime numbers found in the given range.")

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Print an n digit number in reverse order**

…………………………………………………………………………………………………..**2.** **Theory:**

**Concept Implemented:**

This program accepts an integer and prints its **reversed form**, digit by digit.

**Programming Elements Used:**

* **While Loop:** Used to repeatedly extract the last digit and build the reversed number.
* **Arithmetic Operations:** % is used to get the last digit, and // is used to remove it from the original number.
* **Variables:** reversed\_num is updated iteratively to store the result.
* **Input Handling:** The user is prompted to enter a number, ensuring flexibility.

This program is a classic example of **digit manipulation**, commonly used in problems involving palindromes, digital roots, and numerical pattern matching. It teaches the concept of **loops and number theory**, along with how positional value changes when digits are reversed.

…………………………………………………………………………………………………..**3.** **Source Code:**

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

reversed\_num = 0

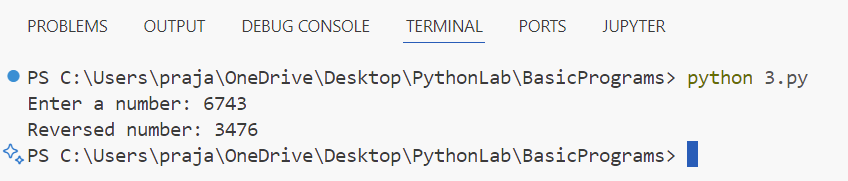
while num > 0:

    digit = num % 10          # Get the last digit

    reversed\_num = reversed\_num \* 10 + digit

    num = num // 10           # Remove the last digit

print("Reversed number:", reversed\_num)

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**

……………………………………………………………………………………

**1. Title: Display, i) a pattern formed with numbers and, ii) a pattern formed with ‘\*’ using nested looping.**

…………………………………………………………………………………………………..**2.** **Theory:**

**Concept Implemented:**

This program prints one of two **text-based patterns** depending on the user’s selection:

1. A number pattern forming a right-angled triangle.
2. A star pattern forming a pyramid.

**Programming Elements Used:**

* **Functions:** Two separate functions (number\_pattern() and star\_pattern()) are defined for each pattern.
* **Nested Loops:** Both patterns utilize nested for loops for constructing rows and columns.
* **Conditionals:** if-elif-else handles user choice and directs the program flow.
* **Input Handling:** The program uses input prompts to allow dynamic selection and row entry.
* **String Formatting:** Printing with alignment is used to shape the star pyramid.

**Theory:**

Pattern printing is an essential practice for beginners to master **nested loops and conditionals**. It also builds logical thinking as it involves careful control of spacing and content on each line. This example showcases **user-driven design**, making the program interactive and responsive to user inputs.

…………………………………………………………………………………………………..**3.** **Source Code:**

def number\_pattern(rows):

    print("\nNumber Pattern (Right-angled Triangle):")

    for i in range(1, rows + 1):

        for j in range(1, i + 1):

            print(j, end=" ")

        print()

def star\_pattern(rows):

    print("\nStar Pattern (Pyramid):")

    for i in range(1, rows + 1):

        # Print leading spaces

        for j in range(rows - i):

            print(" ", end="")

        # Print stars

        for k in range(2 \* i - 1):

            print("\*", end="")

        print()

# Main program

print("Pattern Selector")

print("1. Number Pattern (Right-angled Triangle)")

print("2. Star Pattern (Pyramid)")

choice = int(input("Enter your choice (1 or 2): "))

rows = int(input("Enter the number of rows: "))

if choice == 1:

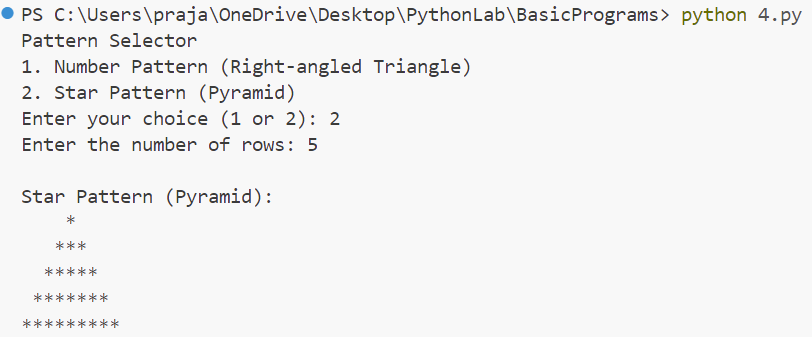
    number\_pattern(rows)

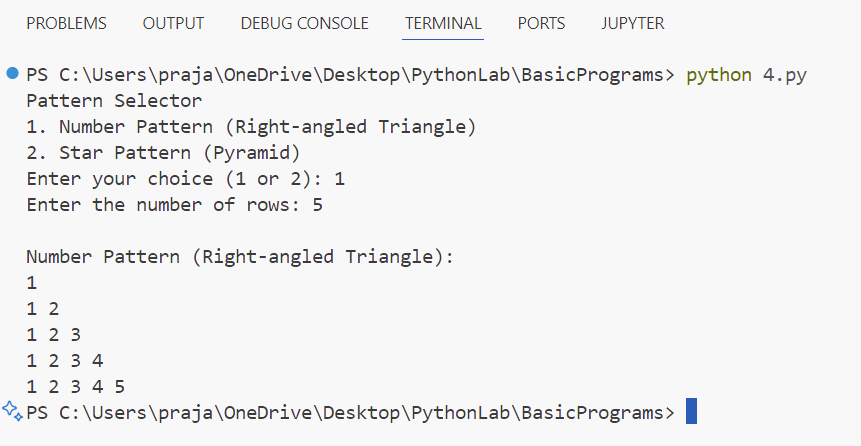
elif choice == 2:

    star\_pattern(rows)

else:

    print("Invalid choice! Please enter either 1 or 2.")

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

**Advanced data types**

[2] Python – LIST - Write Python program to....

……………………………………………………………………………………

**1. Title: Display the count of elements of different data types present in a LIST.**

…………………………………………………………………………………………………..**2.** **Theory:**

This Python program accepts a list of elements from the user and determines the count of elements belonging to different data types: integer, float, boolean, string, and None. The user is prompted to enter the number of elements, followed by the list values, which are initially treated as strings. The program then uses type-checking logic: it checks if a value is numeric using .isdigit(), checks for float by allowing one decimal point, identifies boolean values by matching 'true' or 'false' (case-insensitive), and detects None by checking for the string 'none'. Remaining inputs are classified as strings. Each category is stored in a separate list. Finally, the program prints the count and contents of each data type category. This program demonstrates the use of lists, input handling, type identification using string methods, conditional logic, and list operations in Python.…………………………………………………………………………………………………..**3.** **Source Code:**

n=int(input("Enter number of elements in the list : "))

l = []

print("Enter elements in the list : ")

for i in range(n):

l.append(input())

print("Entered list is :",l)

fl = []

ch = []

i = []

b = []

no = []

for k in l:

if k.isdigit():

i.append(int(k))

elif k.replace('.','',1).isdigit():

fl.append(float(k))

elif k.lower() == 'true' or k.lower() == 'false':

b.append(bool(k))

elif k.lower() == 'none':

no.append(k)

else:

ch.append(k)

print("No of elements of each data type are as follows")

print("Integers :",len(i),i)

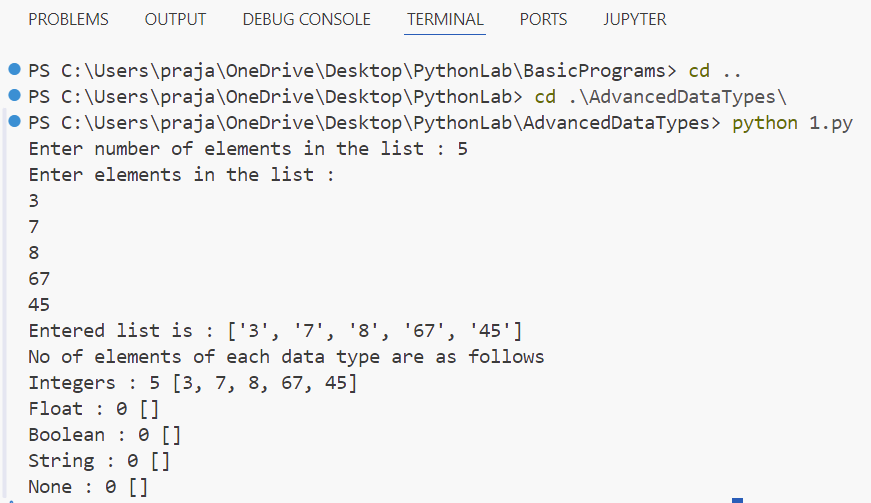
print("Float :",len(fl),fl)

print("Boolean :",len(b),b)

print("String :",len(ch),ch)

print("None :",len(no),no)

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Check for a given value in the LIST; display total count of occurrences along with the index positions of each occurrence.**

…………………………………………………………………………………………………..**2.** **Theory:**

This Python program takes a list of integers from the user and then allows the user to search for a specific value within the list. The program uses a loop to compare each element in the list with the search value and, if a match is found, records its index in a separate list and increments a counter to track the total number of occurrences. After scanning the entire list, it displays the number of times the value occurs along with the exact index positions. If the value is not found, a message is displayed indicating that the element is not present. This program demonstrates the use of list operations, looping constructs, conditional logic, and index tracking, making it a useful example for search operations in collections. …………………………………………………………………………………………………..**3.** **Source Code:**

n=int(input("Enter number of intergers in the list : "))

list = []

print("Enter integers in the list : ")

for i in range(n):

list.append(int(input()))

print("Entered list is :",list)

x = int(input("Enter element you want to search for : "))

ind = []

j = 0

c = 0

for i in list:

if(i == x):

c += 1

ind.append(j)

j += 1

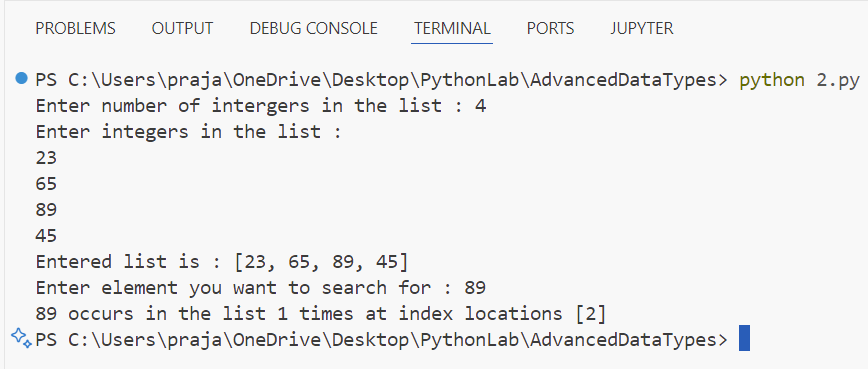
if(c == 0):

print(f"{x} is not present in the list")

else:

print(f"{x} occurs in the list {c} times at index locations {ind}")

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Perform sorting of LIST elements; Press 1 for ascending order and, Press 2 for descending order.**

…………………………………………………………………………………………………..**2.** **Theory:**

This program accepts a list of integers from the user and provides an option to sort the list either in ascending or descending order based on user input. It uses a simple comparison-based sorting technique (similar to bubble sort) implemented using nested loops. Depending on whether the user presses 1 or 2, it sorts the list in ascending or descending order, respectively, by comparing and swapping elements appropriately. After sorting, the program displays the sorted list. It also handles invalid input by checking if the user’s choice is not 1 or 2. The program demonstrates the use of nested loops, conditional branching, swapping, and user interaction through input choices. …………………………………………………………………………………………………..**3.** **Source Code:**

n = int(input("Enter number of integers in the list: "))

num = []

print("Enter integers in the list:")

for i in range(n):

num.append(int(input()))

print("Entered list is:", num)

choice = int(input("Press 1 for ascending order, Press 2 for descending order: "))

l = len(num)

for i in range(l):

for j in range(i + 1, l):

if (choice == 1 and num[i] > num[j]) or (choice == 2 and num[i] < num[j]):

num[i], num[j] = num[j], num[i]

if choice == 1:

print("List sorted in ascending order:", num)

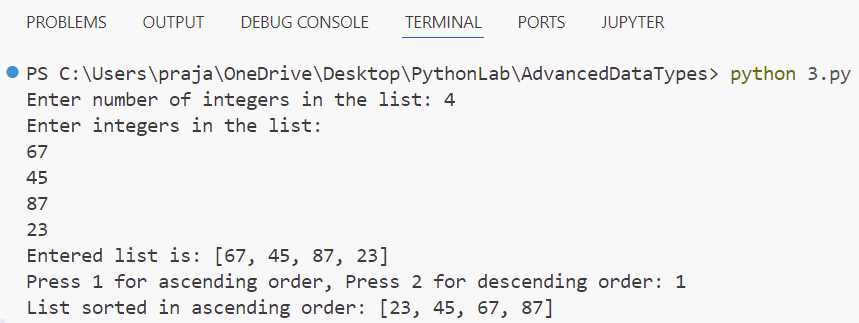
elif choice == 2:

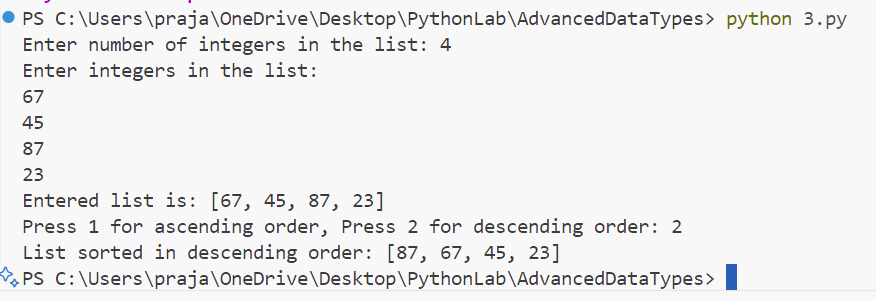
print("List sorted in descending order:", num)

else:

print("Invalid choice! Please enter 1 or 2.")

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**





…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Add elements of List2 in List1, then display the updated List** …………………………………………………………………………………………………..**2.** **Theory:**

This Python program takes a list of integers from the user and then allows the user to search for a specific value within the list. The program uses a loop to compare each element in the list with the search value and, if a match is found, records its index in a separate list and increments a counter to track the total number of occurrences. After scanning the entire list, it displays the number of times the value occurs along with the exact index positions. If the value is not found, a message is displayed indicating that the element is not present. This program demonstrates the use of list operations, looping constructs, conditional logic, and index tracking, making it a useful example for search operations in collections. …………………………………………………………………………………………………..**3.** **Source Code:**

def addlist(L1,L2):

L3=[]

L3 = L1 + L2

return L3

n1 = int(input("Enter number of elements in list 1 : "))

L1 = []

print("Enter elements of list 1 : ")

for i in range(0,n1):

L1.append(input())

print("First list is :",L1)

n2 = int(input("Enter number of elements in list 2 : "))

L2 = []

print("Enter elements of list 2 : ")

for i in range(0,n2):

L2.append(input())

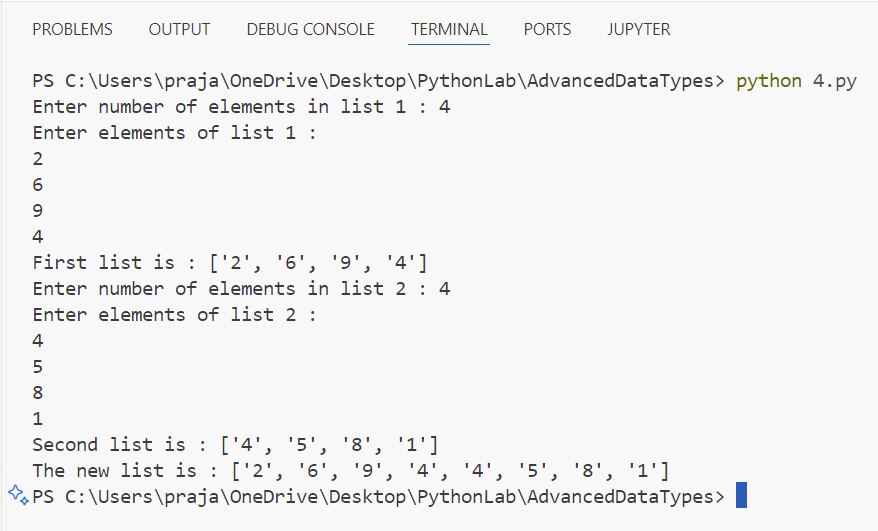
print("Second list is :",L2)

L3 = []

L3 = addlist(L1,L2)

print("The new list is :",L3)

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Demonstrate LIST comprehensions using two examples.** …………………………………………………………………………………………………..**2.** **Theory:**

The given program demonstrates the concept of list comprehension in Python, which is a concise and efficient way to create new lists by applying expressions and conditions on iterable elements. In the first example, a list is constructed by iterating through numbers from 1 to 100 and selecting only those numbers that are divisible by 3, 6, and 9. This is done using a single line of code: [i for i in range(1,101) if (i % 3 == 0) and (i % 6 == 0) and (i % 9 == 0)], which results in a filtered list that satisfies all the specified conditions. In the second example, another list is created to store the squares of even numbers between 1 and 20 using the expression [i\*\*2 for i in range(1, 21) if i % 2 == 0]. Here, only even numbers are selected based on the condition i % 2 == 0, and their squares are calculated and stored in the new list. These examples show how list comprehensions simplify code readability and execution, eliminating the need for explicit loops and multiple lines of logic, making the program more elegant and efficient.

…………………………………………………………………………………………………..**3.** **Source Code:**

print("Numbers divisible by 3,6,9")

list = [i for i in range(1,101) if (i % 3 == 0) and (i % 6 == 0) and (i % 9 == 0)]

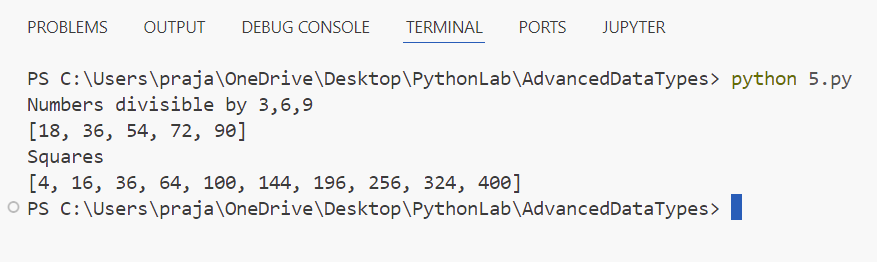
print(list)

print("Squares")

squares = [i\*\*2 for i in range(1, 21) if i % 2 == 0]

print(squares)

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Display count of vowels, consonants, blank spaces, special symbols and digits in a given STRING.** …………………………………………………………………………………………………..**2.** **Theory:**

The provided code is designed to analyze a given string and count the number of vowels, consonants, digits, spaces, and special symbols present. The program starts by accepting a sentence from the user. It then initializes counters for each category: vowels, consonants, digits, spaces, and special symbols, all set to zero. A for loop iterates through each character in the string, checking its type. If the character is a digit, the digits counter is incremented. If the character is alphabetic, it is further checked to see if it's a vowel or a consonant, incrementing the respective counters accordingly. Spaces are counted using a direct comparison to a space character, while any other characters are classified as special symbols. After processing the entire string, the program outputs the counts for each category, providing a comprehensive breakdown of the character types in the sentence.

…………………………………………………………………………………………………..**3.** **Source Code:**

sent = input("Enter a sentence: ")

vowels = 0

consonants = 0

digits = 0

spaces = 0

special\_symbols = 0

for i in sent:

if i.isdigit():

digits += 1

elif i.isalpha():

if i in "aeiouAEIOU":

vowels += 1

else:

consonants += 1

elif i == " ":

spaces += 1

else:

special\_symbols += 1

print(f"Vowels: {vowels}")

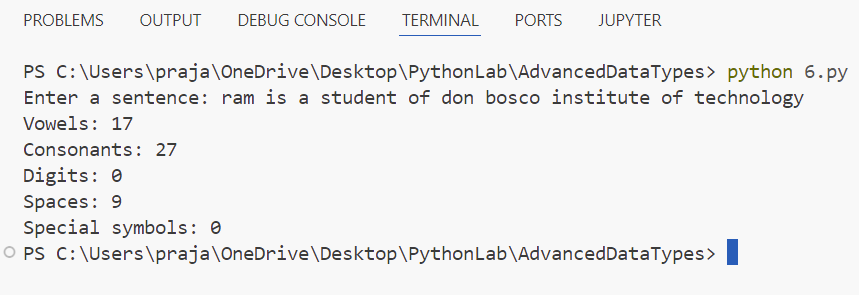
print(f"Consonants: {consonants}")

print(f"Digits: {digits}")

print(f"Spaces: {spaces}")

print(f"Special symbols: {special\_symbols}")

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

**User defined functions**

[3] Python – User Defined Functions (UDF)

Write Python program using UDF to....……………………………………………………………………………………

**1. Title: Calculate library fine based on 2 conditions, a) Book return date and b) Book condition**

…………………………………………………………………………………………………..**2.** **Theory:**

**Concept**: Date handling and conditional logic for calculating fines.  
**Programming Elements**:

* datetime module for date parsing and arithmetic.
* Functions (calculate\_days\_late, calculate\_condition\_fine, calculate\_total\_fine) for modular logic.
* Conditional statements (if-elif-else) to determine fines based on book condition.
* String formatting for displaying the fine report.

…………………………………………………………………………………………………..**3.** **Source Code:**

from datetime import datetime, timedelta

def calculate\_days\_late(return\_date, due\_date):

    """Calculate days between return date and due date"""

    delta = return\_date - due\_date

    return delta.days if delta.days > 0 else 0

def calculate\_condition\_fine(condition):

    """Calculate fine based on book condition"""

    condition = condition.lower()

    if condition == "excellent":

        return 0

    elif condition == "good":

        return 10

    elif condition == "fair":

        return 25

    elif condition == "poor":

        return 50

    elif condition == "damaged":

        return 100

    else:

        return 0  # default if condition not recognized

def calculate\_total\_fine(return\_date\_str, due\_date\_str, condition):

    """Calculate total fine combining late days and condition penalties"""

    try:

        # Convert string dates to datetime objects

        due\_date = datetime.strptime(due\_date\_str, "%Y-%m-%d").date()

        return\_date = datetime.strptime(return\_date\_str, "%Y-%m-%d").date()

        # Calculate days late

        days\_late = calculate\_days\_late(return\_date, due\_date)

        # Calculate fines

        late\_fine = days\_late \* 2  # ₹2 per day late

        condition\_fine = calculate\_condition\_fine(condition)

        total\_fine = late\_fine + condition\_fine

        # Generate report

        print("\n=== Library Fine Calculation ===")

        print(f"Due Date: {due\_date}")

        print(f"Return Date: {return\_date}")

        print(f"Days Late: {days\_late}")

        print(f"Book Condition: {condition.capitalize()}")

        print("-----------------------------")

        print(f"Late Fine: ₹{late\_fine}")

        print(f"Condition Fine: ₹{condition\_fine}")

        print(f"Total Fine: ₹{total\_fine}")

        return total\_fine

    except ValueError:

        print("Invalid date format! Please use YYYY-MM-DD")

        return 0

def main():

    print("Library Fine Calculator")

    print("Enter dates in YYYY-MM-DD format")

    # Get user input

    due\_date = input("Enter due date (YYYY-MM-DD): ")

    return\_date = input("Enter return date (YYYY-MM-DD): ")

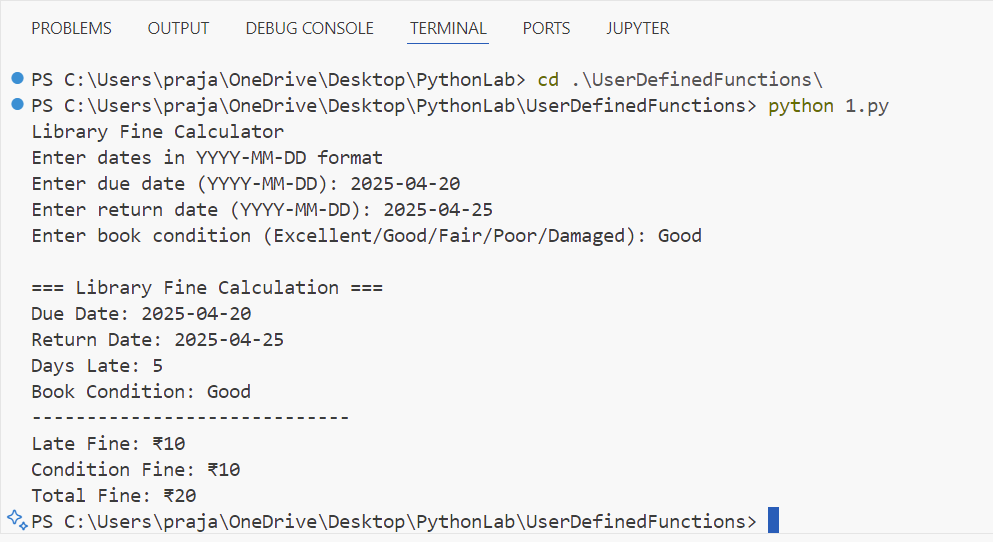
    condition = input("Enter book condition (Excellent/Good/Fair/Poor/Damaged): ")

    # Calculate and display fine

    calculate\_total\_fine(return\_date, due\_date, condition)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**

……………………………………………………………………………………

**1. Title: Calculate and display the Net monthly salary for the two categories of employees, permanent and temporary, based on following inputs from the user – monthly salary, hourly rate, present/absent days’ count, incentives/bonus, income tax, etc.** …………………………………………………………………………………………………..**2.** **Theory:**

**Concept**: Object-oriented programming (OOP) with inheritance for salary calculation.  
**Programming Elements**:

* Classes (Employee, PermanentEmployee, TemporaryEmployee) to model different employee types.
* Inheritance (PermanentEmployee and TemporaryEmployee inherit from Employee).
* Method overriding (calculate\_net\_salary implemented differently in child classes).
* Encapsulation (attributes like tax\_rate, pf\_rate are part of the class).
* User input handling for dynamic salary calculation.

…………………………………………………………………………………………………..**3.** **Source Code:**

class Employee:

    def \_\_init\_\_(self, emp\_id, name, emp\_type):

        self.emp\_id = emp\_id

        self.name = name

        self.emp\_type = emp\_type

        self.tax\_rate = 0.10  # 10% income tax

    def calculate\_net\_salary(self):

        pass  # To be implemented in child classes

    def display\_salary\_slip(self, basic, deductions, allowances, net):

        print("\n=== Salary Slip ===")

        print(f"Employee ID: {self.emp\_id}")

        print(f"Name: {self.name}")

        print(f"Type: {self.emp\_type}")

        print("-----------------------------")

        print(f"Basic Salary: ₹{basic:.2f}")

        print(f"Allowances: ₹{allowances:.2f}")

        print(f"Deductions: ₹{deductions:.2f}")

        print("-----------------------------")

        print(f"Net Salary: ₹{net:.2f}")

        print("=============================")

class PermanentEmployee(Employee):

    def \_\_init\_\_(self, emp\_id, name, monthly\_salary):

        super().\_\_init\_\_(emp\_id, name, "Permanent")

        self.monthly\_salary = monthly\_salary

        self.pf\_rate = 0.12  # 12% PF

    def calculate\_net\_salary(self, present\_days, bonus=0):

        total\_days = 22  # Working days in month

        basic = (self.monthly\_salary / total\_days) \* present\_days

        # Deductions

        pf = basic \* self.pf\_rate

        tax = basic \* self.tax\_rate

        deductions = pf + tax

        # Allowances

        allowances = bonus

        net\_salary = basic + allowances - deductions

        self.display\_salary\_slip(basic, deductions, allowances, net\_salary)

        return net\_salary

class TemporaryEmployee(Employee):

    def \_\_init\_\_(self, emp\_id, name, hourly\_rate):

        super().\_\_init\_\_(emp\_id, name, "Temporary")

        self.hourly\_rate = hourly\_rate

    def calculate\_net\_salary(self, hours\_worked, overtime=0, bonus=0):

        regular\_hours = min(hours\_worked, 176)  # 176 = 22 days \* 8 hours

        overtime\_hours = max(hours\_worked - 176, 0)

        # Basic pay calculations

        basic = regular\_hours \* self.hourly\_rate

        overtime\_pay = overtime\_hours \* (self.hourly\_rate \* 1.5)

        # Deductions

        tax = (basic + overtime\_pay) \* self.tax\_rate

        deductions = tax

        # Allowances

        allowances = overtime\_pay + overtime + bonus

        net\_salary = basic + allowances - deductions

        self.display\_salary\_slip(basic, deductions, allowances, net\_salary)

        return net\_salary

def main():

    print("Employee Salary Calculator")

    # Permanent Employee Calculation

    print("\nPermanent Employee Details:")

    perm\_emp = PermanentEmployee("P1001", "Rahul Sharma", 50000)

    present\_days = int(input("Enter present days (out of 22): "))

    bonus = float(input("Enter bonus amount (₹): "))

    perm\_emp.calculate\_net\_salary(present\_days, bonus)

    # Temporary Employee Calculation

    print("\nTemporary Employee Details:")

    temp\_emp = TemporaryEmployee("T2001", "Priya Patel", 250)

    hours\_worked = float(input("Enter total hours worked: "))

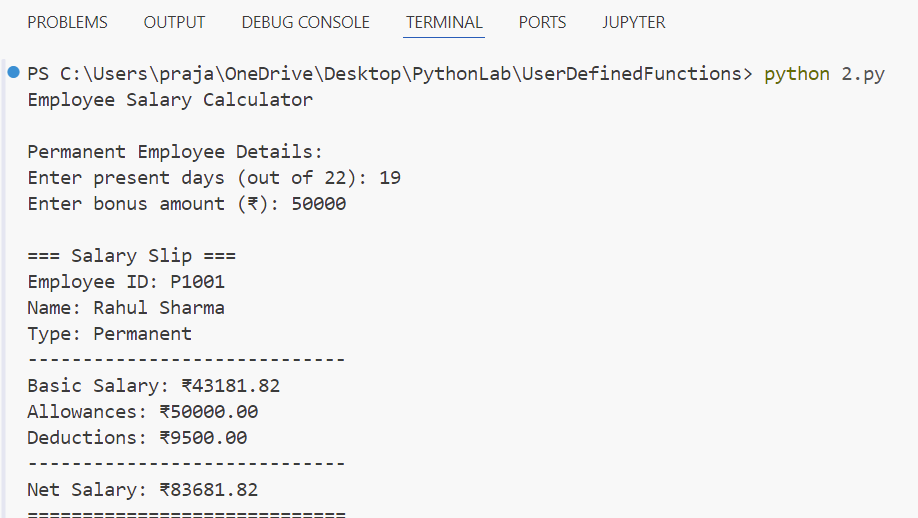
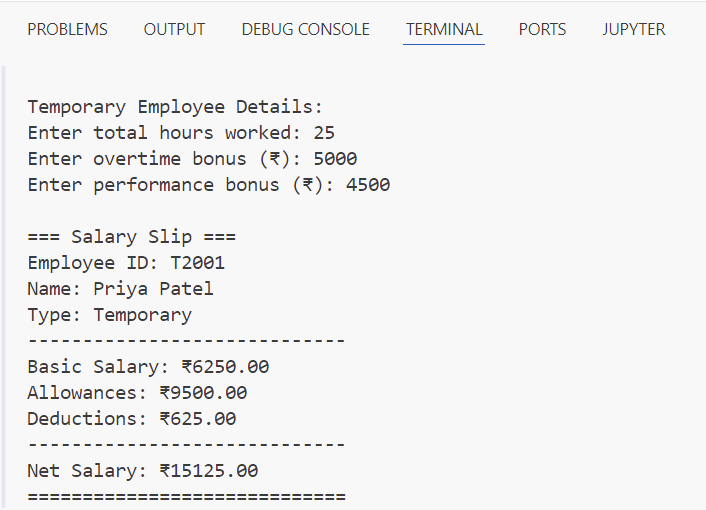
    overtime = float(input("Enter overtime bonus (₹): "))

    bonus = float(input("Enter performance bonus (₹): "))

    temp\_emp.calculate\_net\_salary(hours\_worked, overtime, bonus)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**

…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Display Fibonacci series elements (n) using recursive method.**

…………………………………………………………………………………………………..**2.** **Theory:**

**Concept**: Recursion for mathematical series generation.  
**Programming Elements**:

* Recursive function (fibonacci) to compute Fibonacci numbers.
* Looping (for loop in display\_fibonacci\_series) to display the series.
* Input validation (try-except block) to handle non-integer inputs.

…………………………………………………………………………………………………..**3.** **Source Code:**

def fibonacci(n):

    """Recursive function to return the nth Fibonacci number"""

    if n <= 0:

        return 0

    elif n == 1:

        return 1

    else:

        return fibonacci(n-1) + fibonacci(n-2)

def display\_fibonacci\_series(n):

    """Display Fibonacci series up to n elements"""

    print(f"Fibonacci series (first {n} elements):")

    for i in range(n):

        print(fibonacci(i), end=" ")

    print()  # For new line

# Main program

if \_\_name\_\_ == "\_\_main\_\_":

    try:

        n = int(input("Enter the number of Fibonacci elements to display: "))

        if n < 0:

            print("Please enter a non-negative integer.")

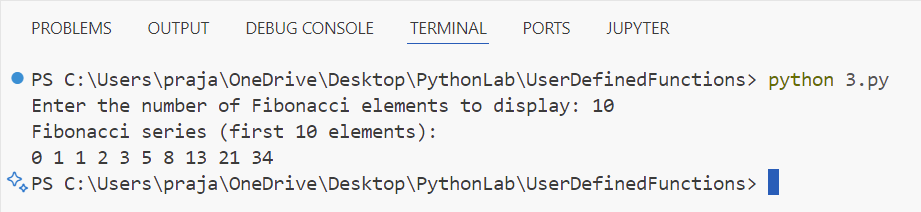
        else:

            display\_fibonacci\_series(n)

    except ValueError:

        print("Invalid input! Please enter a valid integer.")

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Demonstrate passing and returning a List to/from a user defined function.** …………………………………………………………………………………………………..**2.** **Theory:**

**Concept:** List manipulation and function parameter passing (by value vs. by reference).  
**Programming Elements:**

* Functions (process\_list, add\_greeting\_to\_names, split\_odd\_even) demonstrating different ways to handle lists.
* List comprehensions (in split\_odd\_even for filtering odd/even numbers).
* Type checking (isinstance) to handle different data types in lists.
* In-place list modification (add\_greeting\_to\_names modifies the original list).
* Returning multiple values (tuple unpacking for odd\_nums and even\_nums).

…………………………………………………………………………………………………..**3.** **Source Code:**

def fibonacci(n):

    """Recursive function to return the nth Fibonacci number"""

    if n <= 0:

        return 0

    elif n == 1:

        return 1

    else:

        return fibonacci(n-1) + fibonacci(n-2)

def display\_fibonacci\_series(n):

    """Display Fibonacci series up to n elements"""

    print(f"Fibonacci series (first {n} elements):")

    for i in range(n):

        print(fibonacci(i), end=" ")

    print()  # For new line

# Main program

if \_\_name\_\_ == "\_\_main\_\_":

    try:

        n = int(input("Enter the number of Fibonacci elements to display: "))

        if n < 0:

            print("Please enter a non-negative integer.")

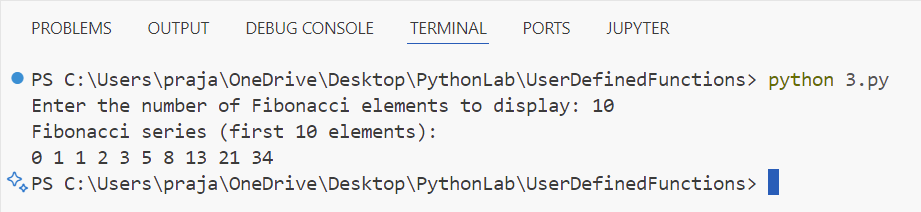
        else:

            display\_fibonacci\_series(n)

    except ValueError:

        print("Invalid input! Please enter a valid integer.")

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

**Exploring concept of modules and exception handling**

[4] Python – User defined modules and exception handling

……………………………………………………………………………………

**1. Title: Create a module having 3 functions – factorial (), primeNumber () and powNumber (). Import this module in the main menu driven program to access all the functions (accept input from the user).**

…………………………………………………………………………………………………..**2.** **Theory:**

This program is a **menu-driven calculator** that performs three operations:

1. **Factorial** (iterative calculation with edge-case handling).
2. **Prime check** (optimized with square root limit).
3. **Power** (using Python’s \*\* operator).

**Key Programming Elements:**

* **Modular design**: Logic is split into math\_operations.py for reusability.
* **Menu loop**: while True with input() for user choices.
* **Error handling**: try-except blocks for invalid inputs.
* **Efficient algorithms**: Iterative factorial, optimized prime check.

…………………………………………………………………………………………………..**3.** **Source Code (main\_program.py)**

from math\_operations import factorial, primeNumber, powNumber

def main():

    while True:

        print("\nMENU")

        print("1. Calculate Factorial")

        print("2. Check Prime Number")

        print("3. Calculate Power")

        print("4. Exit")

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

        if choice == '1':

            try:

                num = int(input("Enter a non-negative integer: "))

                print(f"Factorial of {num} is: {factorial(num)}")

            except ValueError:

                print("Please enter a valid integer!")

        elif choice == '2':

            try:

                num = int(input("Enter a number to check: "))

                if primeNumber(num):

                    print(f"{num} is a prime number")

                else:

                    print(f"{num} is not a prime number")

            except ValueError:

                print("Please enter a valid integer!")

        elif choice == '3':

            try:

                base = float(input("Enter base: "))

                exponent = float(input("Enter exponent: "))

                print(f"{base}^{exponent} = {powNumber(base, exponent)}")

            except ValueError:

                print("Please enter valid numbers!")

        elif choice == '4':

            print("Exiting program...")

            break

        else:

            print("Invalid choice! Please enter 1-4")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**(math\_operations.py)**

def factorial(n):

    """Calculate factorial of a number"""

    if n < 0:

        return "Factorial doesn't exist for negative numbers"

    elif n == 0 or n == 1:

        return 1

    else:

        fact = 1

        for i in range(2, n+1):

            fact \*= i

        return fact

def primeNumber(num):

    """Check if a number is prime"""

    if num <= 1:

        return False

    if num == 2:

        return True

    if num % 2 == 0:

        return False

    for i in range(3, int(num\*\*0.5)+1, 2):

        if num % i == 0:

            return False

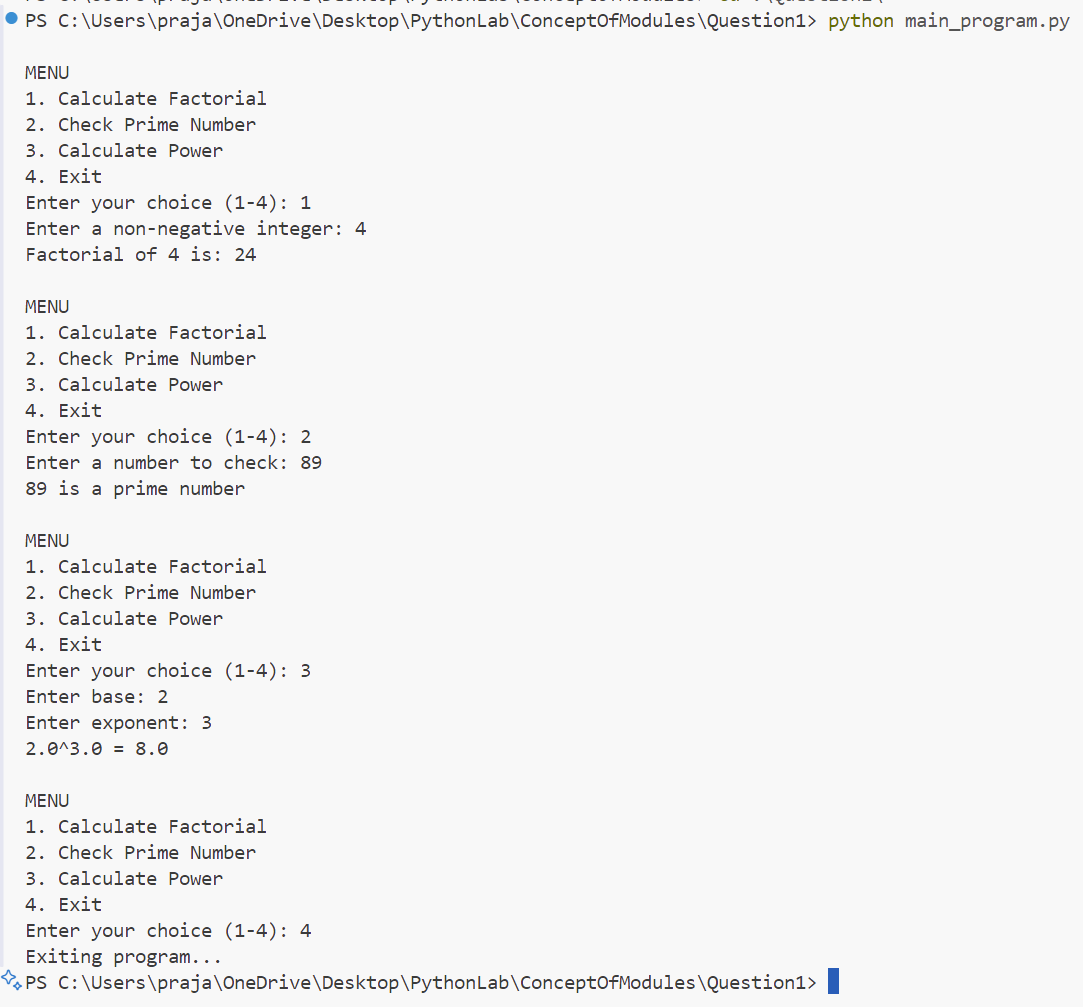
    return True

def powNumber(base, exponent):

    """Calculate power of a number"""

    return base \*\* exponent

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Write a program to demonstrate multiple exceptions handling, specifically, NameError, IndexError, and ZeroDivisionError.** …………………………………………………………………………………………………..**2.** **Theory:**

This program demonstrates **common Python exceptions** through a menu-driven interface. Users can trigger and handle:

1. NameError (undefined variable)
2. IndexError (out-of-bounds list access)
3. ZeroDivisionError (division by zero)

**Key Programming Elements:**

* **Exception handling**: Uses try-except blocks to catch and explain errors.
* **Menu loop**: while True with user input for interactive testing.
* **Error recovery**: Specific except blocks for each exception type.
* **Cleanup**: finally block executes after each attempt.

…………………………………………………………………………………………………..**3.** **Source Code:**

def demonstrate\_exceptions():

    while True:

        print("\nException Demonstration Menu:")

        print("1. Trigger NameError")

        print("2. Trigger IndexError")

        print("3. Trigger ZeroDivisionError")

        print("4. Exit")

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

        try:

            if choice == '1':

                # NameError demonstration

                print("\nAttempting to use an undefined variable...")

                print(undefined\_variable)  # This variable doesn't exist

            elif choice == '2':

                # IndexError demonstration

                my\_list = [10, 20, 30]

                print("\nCurrent list:", my\_list)

                index = int(input("Enter an index to access (0-2): "))

                print("Value at index", index, "is:", my\_list[index])

            elif choice == '3':

                # ZeroDivisionError demonstration

                numerator = int(input("\nEnter numerator: "))

                denominator = int(input("Enter denominator: "))

                result = numerator / denominator

                print("Division result:", result)

            elif choice == '4':

                print("Exiting program...")

                break

            else:

                print("Invalid choice! Please enter 1-4")

        except NameError:

            print("Error: You tried to use a variable that doesn't exist!")

        except IndexError:

            print("Error: You tried to access an index that's out of range!")

            print("Remember list indices start from 0!")

        except ZeroDivisionError:

            print("Error: You cannot divide by zero!")

        except ValueError:

            print("Error: Please enter valid integers where required!")

        except Exception as e:

            print("An unexpected error occurred:", type(e).\_\_name\_\_, "-", e)

        else:

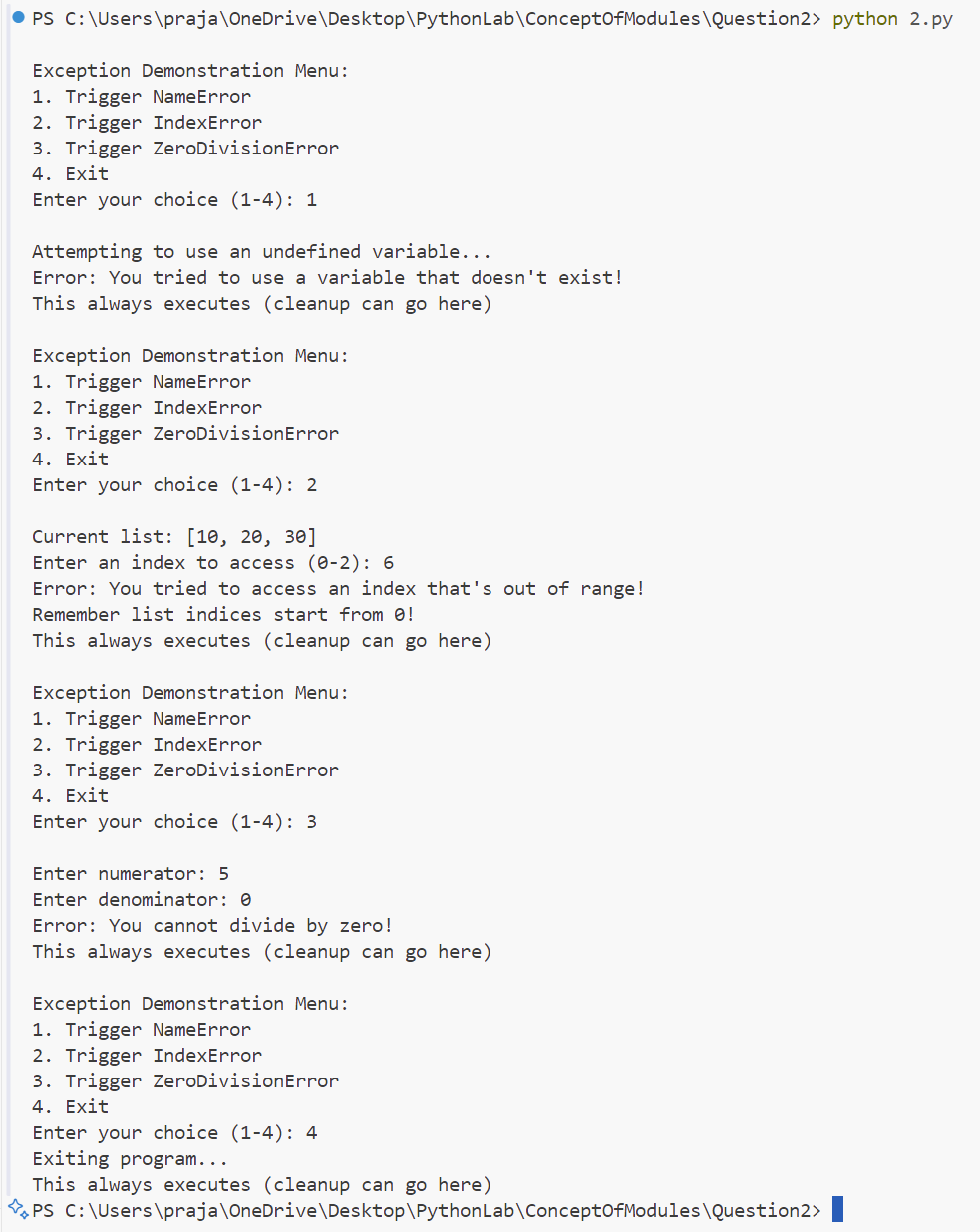
            print("Operation completed successfully!")

        finally:

            print("This always executes (cleanup can go here)")

# Run the demonstration

demonstrate\_exceptions()

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**

……………………………………………………………………………………

**1. Title: Write a program to implement multiple exceptions handling such as ValueError, KeyError, PermissionError, General exception within a standard LOGIN process (Login successful, User doesn’t exist, Incorrect password, Too many attempts, etc.).** …………………………………………………………………………………………………..**2.** **Theory:**

This program implements a **secure login system** with:

1. **Password authentication** (using getpass for hidden input)
2. **Account locking** after 3 failed attempts
3. **Activity logging** to track login attempts
4. **Error handling** for security and system failures

**Key Programming Elements**

1. **Security Features**
   * getpass hides password input
   * Account locking after MAX\_ATTEMPTS
   * Password validation without storing plaintext (mock database)
2. **Error Handling**
   * Custom exceptions (KeyError, ValueError, PermissionError)
   * Logs all attempts (success/failure) to login\_audit.log
   * Handles unexpected errors gracefully
3. **Audit Logging**
   * Timestamped logs for security monitoring
   * File write error handling (PermissionError)
4. **User Flow**
   * Menu-driven interface (while loop)
   * Clean exit on KeyboardInterrupt or system errors

…………………………………………………………………………………………………..**3.** **Source Code:**

import getpass  # For secure password input

import sys

from datetime import datetime

# Mock user database (in real app, use proper database with hashed passwords)

USER\_DB = {

    "admin": {"password": "Admin@123", "attempts": 0, "locked": False},

    "user1": {"password": "Password1!", "attempts": 0, "locked": False},

    "user2": {"password": "Secure#456", "attempts": 0, "locked": False}

}

MAX\_ATTEMPTS = 3

LOG\_FILE = "login\_audit.log"

def write\_log(message):

    """Log events to a file with timestamp"""

    try:

        with open(LOG\_FILE, 'a') as f:

            f.write(f"{datetime.now()} - {message}\n")

    except PermissionError:

        print("Warning: Could not write to log file (permission denied)")

    except Exception as e:

        print(f"Logging error: {type(e).\_\_name\_\_} - {e}")

def login():

    try:

        username = input("Enter username: ").strip()

        # Check if user exists

        if username not in USER\_DB:

            write\_log(f"Login failed - User '{username}' doesn't exist")

            raise KeyError("User doesn't exist")

        # Check if account is locked

        if USER\_DB[username]["locked"]:

            write\_log(f"Login blocked - Account '{username}' is locked")

            raise PermissionError("Account locked (too many failed attempts)")

        # Get password securely

        password = getpass.getpass("Enter password: ")

        # Validate password

        if password != USER\_DB[username]["password"]:

            USER\_DB[username]["attempts"] += 1

            write\_log(f"Login failed - Incorrect password for '{username}' (Attempt {USER\_DB[username]['attempts']})")

            if USER\_DB[username]["attempts"] >= MAX\_ATTEMPTS:

                USER\_DB[username]["locked"] = True

                write\_log(f"Account '{username}' locked due to too many attempts")

                raise PermissionError("Too many attempts - account locked")

            else:

                raise ValueError("Incorrect password")

        # Successful login

        USER\_DB[username]["attempts"] = 0  # Reset attempts

        write\_log(f"Login successful - User '{username}'")

        return f"Welcome, {username}! Login successful."

    except KeyError as e:

        return f"Error: {e}"

    except ValueError as e:

        return f"Error: {e}"

    except PermissionError as e:

        return f"Security Error: {e}"

    except Exception as e:

        write\_log(f"Unexpected error during login: {type(e).\_\_name\_\_} - {e}")

        return f"System Error: Please try again later"

def main():

    print("\n=== Secure Login System ===")

    while True:

        print("\n1. Login")

        print("2. Exit")

        choice = input("Select option: ")

        if choice == '1':

            result = login()

            print(f"\n{result}")

        elif choice == '2':

            print("Exiting system...")

            sys.exit(0)

        else:

            print("Invalid choice")

if \_\_name\_\_ == "\_\_main\_\_":

    try:

        main()

    except KeyboardInterrupt:

        print("\nProgram terminated by user")

        sys.exit(1)

    except Exception as e:

        write\_log(f"Critical system failure: {type(e).\_\_name\_\_} - {e}")

        print("A critical error occurred. See logs for details.")

        sys.exit(1)

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

**Object Oriented Programming**

[5] Python – OOPs

……………………………………………………………………………………

**1. Title: Write an object oriented program to demonstrate working of default and parameterized constructors.** …………………………………………………………………………………………………..**2.** **Theory:**

**Concept**: Class initialization using constructors  
**Key Elements**:

* **Default constructor** (\_\_init\_\_ with no parameters) and **parameterized constructor** (with parameters)
* **Class variables** (company = "ABC Corp") shared across instances
* **Instance methods** (display\_info()) for object-specific operations
* **Object creation** demonstrating both constructor types

…………………………………………………………………………………………………..**3.** **Source Code:**

class Employee:

    # Class variable

    company = "ABC Corp"

    # Default Constructor (no parameters)

    def \_\_init\_\_(self):

        self.name = "New Employee"

        self.id = 0000

        self.salary = 0.0

        print("Default constructor called - Employee created with default values")

    # Parameterized Constructor

    def \_\_init\_\_(self, name, emp\_id, salary):

        self.name = name

        self.id = emp\_id

        self.salary = salary

        print(f"Parameterized constructor called - Employee {self.name} created")

    def display\_info(self):

        print(f"\nEmployee Details:")

        print(f"Name: {self.name}")

        print(f"ID: {self.id}")

        print(f"Salary: ${self.salary}")

        print(f"Company: {Employee.company}")

class Laptop:

    # Default Constructor

    def \_\_init\_\_(self):

        self.brand = "Dell"

        self.model = "XPS 15"

        self.price = 1499.99

        print("\nDefault constructor called - Laptop created with default values")

    def display\_specs(self):

        print(f"\nLaptop Specifications:")

        print(f"Brand: {self.brand}")

        print(f"Model: {self.model}")

        print(f"Price: ${self.price}")

# Main program

if \_\_name\_\_ == "\_\_main\_\_":

    print("=== Constructor Demonstration ===")

    # Creating objects using different constructors

    print("\nCreating Employee objects:")

    emp1 = Employee("John Doe", 1001, 75000.50)  # Parameterized constructor

    emp1.display\_info()

    print("\nCreating Laptop objects:")

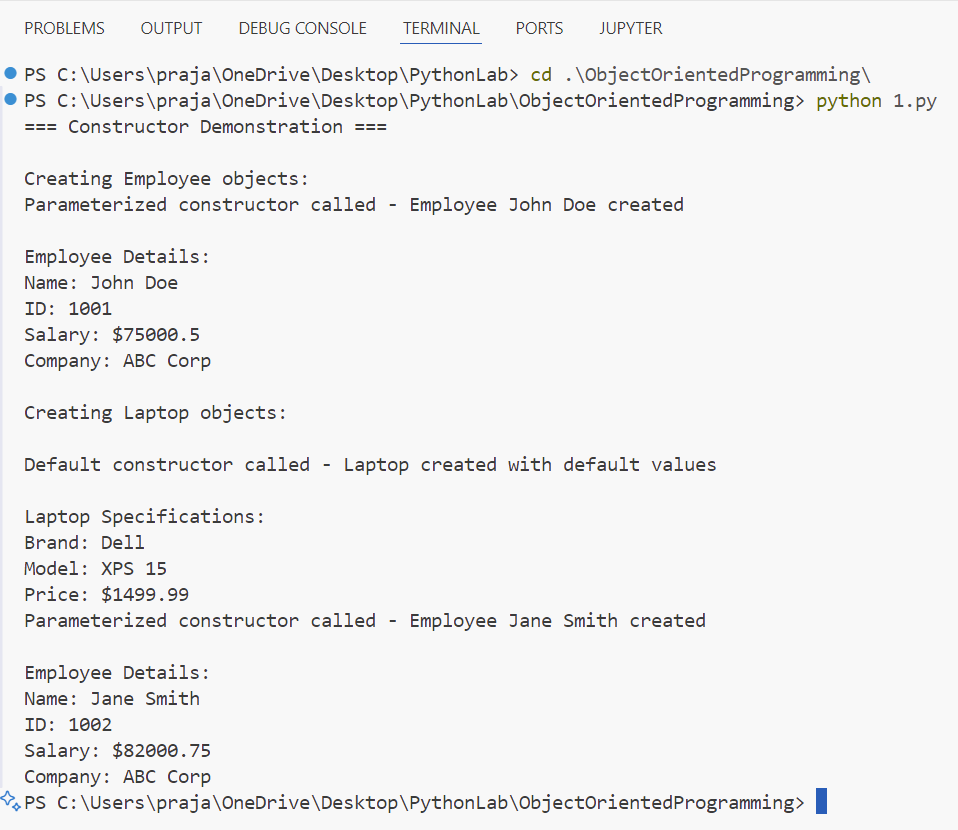
    laptop1 = Laptop()  # Default constructor

    laptop1.display\_specs()

    # Creating another employee

    emp2 = Employee("Jane Smith", 1002, 82000.75)

    emp2.display\_info()

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**

…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Write an object oriented menu driven program to perform banking operations (New account, Deposit, Withdraw, Balance, Show all, Exit).** …………………………………………………………………………………………………..**2.** **Theory:**

**Concept:** Class composition and encapsulation  
**Key Elements:**

* BankAccount class with core banking operations (deposit/withdraw/balance)
* BankingSystem class manages multiple accounts (composition)
* Input validation for transactions
* Menu-driven interface for user interaction
* str method for clean account representation

…………………………………………………………………………………………………..**3.** **Source Code:**

class BankAccount:

    def \_\_init\_\_(self, account\_number, name, account\_type, balance=0):

        self.account\_number = account\_number

        self.name = name

        self.account\_type = account\_type

        self.balance = balance

    def deposit(self, amount):

        if amount > 0:

            self.balance += amount

            print(f"\nDeposited ₹{amount}. New balance: ₹{self.balance}")

        else:

            print("\nInvalid deposit amount!")

    def withdraw(self, amount):

        if amount > 0:

            if self.balance >= amount:

                self.balance -= amount

                print(f"\nWithdrew ₹{amount}. New balance: ₹{self.balance}")

            else:

                print("\nInsufficient funds!")

        else:

            print("\nInvalid withdrawal amount!")

    def display\_balance(self):

        print(f"\nAccount Balance for {self.name}: ₹{self.balance}")

    def \_\_str\_\_(self):

        return (f"Account Number: {self.account\_number}\n"

                f"Account Holder: {self.name}\n"

                f"Account Type: {self.account\_type}\n"

                f"Balance: ₹{self.balance}\n")

class BankingSystem:

    def \_\_init\_\_(self):

        self.accounts = {}

    def create\_account(self):

        print("\n--- Create New Account ---")

        account\_number = input("Enter account number: ")

        if account\_number in self.accounts:

            print("Account number already exists!")

            return

        name = input("Enter account holder name: ")

        account\_type = input("Enter account type (Savings/Current): ")

        initial\_deposit = float(input("Enter initial deposit amount: ₹"))

        if initial\_deposit < 0:

            print("Invalid initial deposit amount!")

            return

        self.accounts[account\_number] = BankAccount(account\_number, name, account\_type, initial\_deposit)

        print("\nAccount created successfully!")

    def deposit(self):

        account\_number = input("\nEnter account number: ")

        if account\_number in self.accounts:

            amount = float(input("Enter deposit amount: ₹"))

            self.accounts[account\_number].deposit(amount)

        else:

            print("Account not found!")

    def withdraw(self):

        account\_number = input("\nEnter account number: ")

        if account\_number in self.accounts:

            amount = float(input("Enter withdrawal amount: ₹"))

            self.accounts[account\_number].withdraw(amount)

        else:

            print("Account not found!")

    def check\_balance(self):

        account\_number = input("\nEnter account number: ")

        if account\_number in self.accounts:

            self.accounts[account\_number].display\_balance()

        else:

            print("Account not found!")

    def show\_all\_accounts(self):

        print("\n--- All Accounts ---")

        if not self.accounts:

            print("No accounts found!")

        else:

            for account in self.accounts.values():

                print(account)

    def run(self):

        while True:

            print("\n==== Banking System ====")

            print("1. Create New Account")

            print("2. Deposit Money")

            print("3. Withdraw Money")

            print("4. Check Balance")

            print("5. Show All Accounts")

            print("6. Exit")

            choice = input("Enter your choice (1-6): ")

            if choice == '1':

                self.create\_account()

            elif choice == '2':

                self.deposit()

            elif choice == '3':

                self.withdraw()

            elif choice == '4':

                self.check\_balance()

            elif choice == '5':

                self.show\_all\_accounts()

            elif choice == '6':

                print("\nThank you for using our banking system!")

                break

            else:

                print("\nInvalid choice! Please try again.")

# Main program

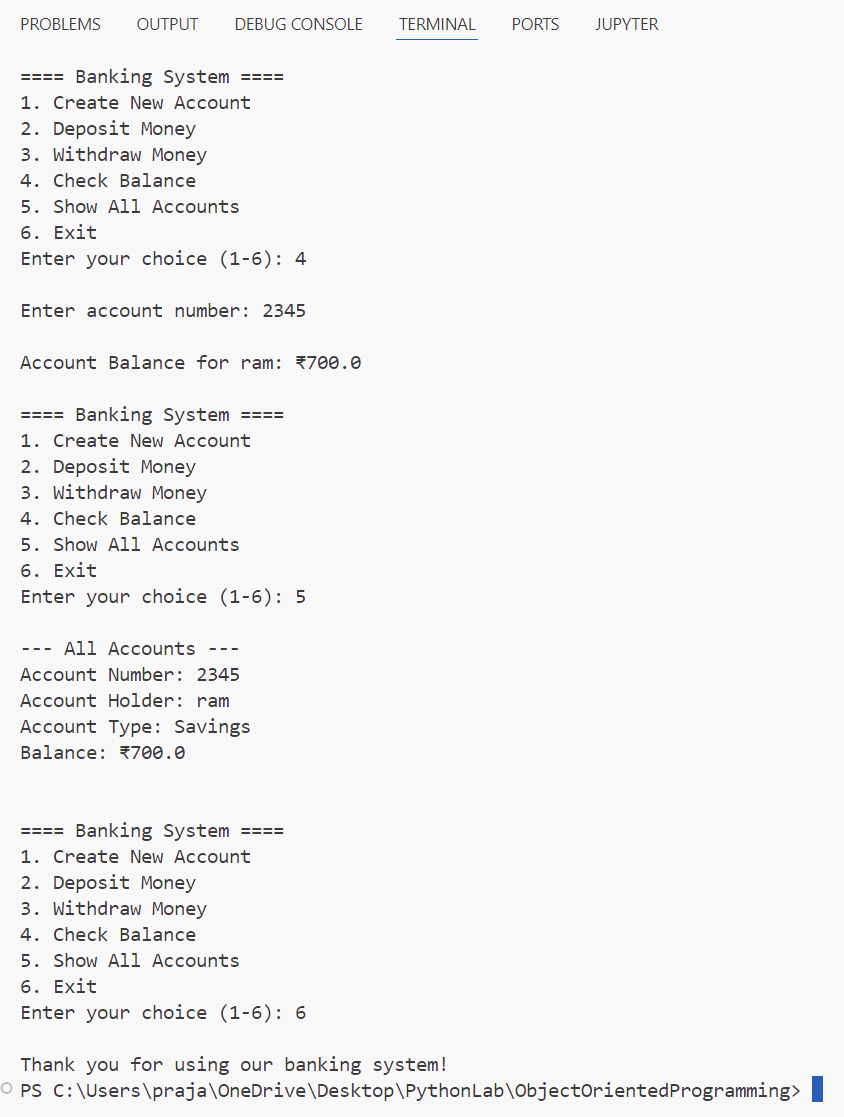
if \_\_name\_\_ == "\_\_main\_\_":

    bank = BankingSystem()

    bank.run()

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**





…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Write object oriented program to implement, i) Single level inheritance, and ii) Multilevel inheritance by considering appropriate real life scenarios (use super(), \_\_init\_\_, \_\_str\_\_, and \_\_name\_\_ ).** …………………………………………………………………………………………………..**2.** **Theory:**

**Concept**: Single-level and multilevel inheritance  
**Key Elements**:

* **Single inheritance**: Car inherits from Vehicle
* **Multilevel inheritance**: Person → Employee → Manager
* **super()** for parent class method calls
* **Method overriding** (custom \_\_str\_\_ in child classes)
* **Class-specific methods** (honk(), conduct\_meeting())

…………………………………………………………………………………………………..**3.** **Source Code:**

# Single-Level Inheritance Example

class Vehicle:

    def \_\_init\_\_(self, make, model, year):

        self.make = make

        self.model = model

        self.year = year

        print(f"{self.\_\_class\_\_.\_\_name\_\_} instance created")

    def \_\_str\_\_(self):

        return f"{self.year} {self.make} {self.model}"

    def start\_engine(self):

        print("Engine started")

class Car(Vehicle):  # Single inheritance

    def \_\_init\_\_(self, make, model, year, num\_doors):

        super().\_\_init\_\_(make, model, year)  # Using super() to call parent's \_\_init\_\_

        self.num\_doors = num\_doors

    def \_\_str\_\_(self):

        return f"{super().\_\_str\_\_()} with {self.num\_doors} doors"

    def honk(self):

        print("Beep beep!")

# Multilevel Inheritance Example

class Person:

    def \_\_init\_\_(self, name, age):

        self.name = name

        self.age = age

        print(f"{self.\_\_class\_\_.\_\_name\_\_} instance created")

    def \_\_str\_\_(self):

        return f"Person: {self.name}, {self.age} years old"

class Employee(Person):  # First level inheritance

    def \_\_init\_\_(self, name, age, emp\_id):

        super().\_\_init\_\_(name, age)

        self.emp\_id = emp\_id

        self.department = None

    def \_\_str\_\_(self):

        return f"{super().\_\_str\_\_()} | ID: {self.emp\_id}"

    def assign\_department(self, dept):

        self.department = dept

        print(f"{self.name} assigned to {dept} department")

class Manager(Employee):  # Second level inheritance (multilevel)

    def \_\_init\_\_(self, name, age, emp\_id, team\_size):

        super().\_\_init\_\_(name, age, emp\_id)

        self.team\_size = team\_size

    def \_\_str\_\_(self):

        return f"{super().\_\_str\_\_()} | Manager of {self.team\_size} people"

    def conduct\_meeting(self):

        print(f"{self.name} is conducting a meeting with {self.team\_size} team members")

def main():

    print("\n=== Single-Level Inheritance Demo ===")

    my\_car = Car("Toyota", "Camry", 2022, 4)

    print(my\_car)  # Uses Car's \_\_str\_\_

    my\_car.start\_engine()  # Inherited from Vehicle

    my\_car.honk()  # Car's own method

    print("\n=== Multilevel Inheritance Demo ===")

    manager = Manager("Sarah Johnson", 35, "MGR100", 8)

    print(manager)  # Uses Manager's \_\_str\_\_

    manager.assign\_department("Marketing")  # Inherited from Employee

    manager.conduct\_meeting()  # Manager's own method

if \_\_name\_\_ == "\_\_main\_\_":

    print(f"\nRunning {\_\_name\_\_} program")

    main()

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

**GUI & Database programming**

[6] Python – GUI and Database Connection

……………………………………………………………………………………

**1. Title: Design a working interface for the login and registration process with proper form validations and database connection using tkinter and SQLite**

…………………………………………………………………………………………………..**2.** **Theory:**

**Concept**: Secure user authentication with form validation and database storage.  
**Key Elements**:

* **Tkinter GUI**: Login/registration forms with input validation.
* **SQLite Database**: Stores user credentials (username, email, password).
* **Form Validation**: Regex for email, length checks for username/password.
* **Security**: Basic password matching (in production, use hashing).

**Example Flow**:

1. User registers with valid credentials → stored in DB.
2. User logs in with correct credentials → "Login successful".

…………………………………………………………………………………………………..**3.** **Source Code:**

import tkinter as tk

from tkinter import messagebox

import sqlite3

import re

# Database setup

def initialize\_db():

    conn = sqlite3.connect('user\_database.db')

    cursor = conn.cursor()

    cursor.execute('''

        CREATE TABLE IF NOT EXISTS users (

            id INTEGER PRIMARY KEY AUTOINCREMENT,

            username TEXT UNIQUE NOT NULL,

            email TEXT UNIQUE NOT NULL,

            password TEXT NOT NULL

        )

    ''')

    conn.commit()

    conn.close()

# Form validation functions

def validate\_email(email):

    pattern = r'^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    return re.match(pattern, email)

def validate\_password(password):

    return len(password) >= 8

def validate\_username(username):

    return len(username) >= 4 and ' ' not in username

# Main application class

class AuthApp:

    def \_\_init\_\_(self, root):

        self.root = root

        self.root.title("Authentication System")

        self.root.geometry("400x300")

        # Initialize database

        initialize\_db()

        # Show login frame by default

        self.show\_login\_frame()

    def clear\_frame(self):

        for widget in self.root.winfo\_children():

            widget.destroy()

    def show\_login\_frame(self):

        self.clear\_frame()

        # Login Frame

        tk.Label(self.root, text="Login", font=("Arial", 16)).pack(pady=10)

        tk.Label(self.root, text="Username:").pack()

        self.login\_username = tk.Entry(self.root)

        self.login\_username.pack()

        tk.Label(self.root, text="Password:").pack()

        self.login\_password = tk.Entry(self.root, show="\*")

        self.login\_password.pack()

        tk.Button(self.root, text="Login", command=self.login\_user).pack(pady=10)

        tk.Button(self.root, text="Register", command=self.show\_register\_frame).pack()

    def show\_register\_frame(self):

        self.clear\_frame()

        # Registration Frame

        tk.Label(self.root, text="Register", font=("Arial", 16)).pack(pady=10)

        tk.Label(self.root, text="Username:").pack()

        self.reg\_username = tk.Entry(self.root)

        self.reg\_username.pack()

        tk.Label(self.root, text="Email:").pack()

        self.reg\_email = tk.Entry(self.root)

        self.reg\_email.pack()

        tk.Label(self.root, text="Password:").pack()

        self.reg\_password = tk.Entry(self.root, show="\*")

        self.reg\_password.pack()

        tk.Label(self.root, text="Confirm Password:").pack()

        self.reg\_confirm\_password = tk.Entry(self.root, show="\*")

        self.reg\_confirm\_password.pack()

        tk.Button(self.root, text="Register", command=self.register\_user).pack(pady=10)

        tk.Button(self.root, text="Back to Login", command=self.show\_login\_frame).pack()

    def login\_user(self):

        username = self.login\_username.get()

        password = self.login\_password.get()

        if not username or not password:

            messagebox.showerror("Error", "All fields are required!")

            return

        conn = sqlite3.connect('user\_database.db')

        cursor = conn.cursor()

        cursor.execute("SELECT \* FROM users WHERE username = ?", (username,))

        user = cursor.fetchone()

        conn.close()

        if user and user[3] == password:  # Simple password check (in real apps, use hashing)

            messagebox.showinfo("Success", "Login successful!")

            # Here you would typically open the main application window

        else:

            messagebox.showerror("Error", "Invalid username or password")

    def register\_user(self):

        username = self.reg\_username.get()

        email = self.reg\_email.get()

        password = self.reg\_password.get()

        confirm\_password = self.reg\_confirm\_password.get()

        # Validation

        if not all([username, email, password, confirm\_password]):

            messagebox.showerror("Error", "All fields are required!")

            return

        if not validate\_username(username):

            messagebox.showerror("Error", "Username must be at least 4 characters with no spaces")

            return

        if not validate\_email(email):

            messagebox.showerror("Error", "Invalid email format")

            return

        if not validate\_password(password):

            messagebox.showerror("Error", "Password must be at least 8 characters")

            return

        if password != confirm\_password:

            messagebox.showerror("Error", "Passwords don't match")

            return

        # Database operation

        conn = sqlite3.connect('user\_database.db')

        cursor = conn.cursor()

        try:

            cursor.execute(

                "INSERT INTO users (username, email, password) VALUES (?, ?, ?)",

                (username, email, password)  # In real apps, store hashed password

            )

            conn.commit()

            messagebox.showinfo("Success", "Registration successful!")

            self.show\_login\_frame()

        except sqlite3.IntegrityError:

            messagebox.showerror("Error", "Username or email already exists")

        finally:

            conn.close()

# Run the application

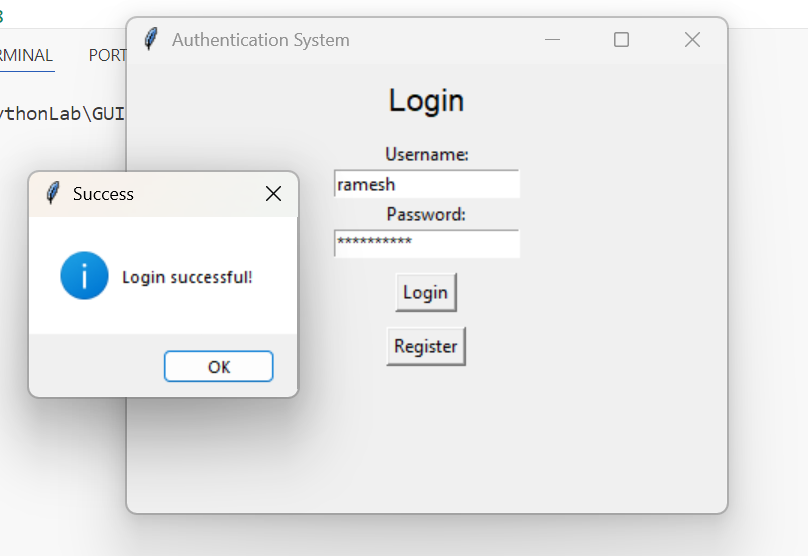
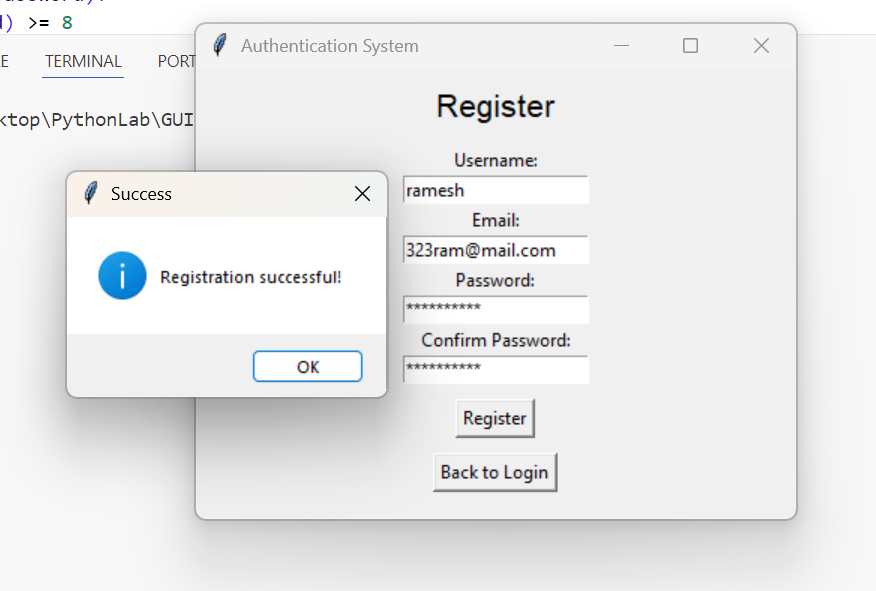
if \_\_name\_\_ == "\_\_main\_\_":

    root = tk.Tk()

    app = AuthApp(root)

    root.mainloop()

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Perform CRUD database operations using a menu driven program (User Management - Add, Show, Delete, Update and Search).** …………………………………………………………………………………………………..**2.** **Theory:**

**Concept**: CRUD (Create, Read, Update, Delete) operations for user management.  
**Key Elements**:

* **Tkinter + Treeview**: Tabular display of users with search functionality.
* **SQLite Database**: Stores user details (name, email, phone, address).
* **CRUD Operations**:
  + **Add/Update**: Input validation for required fields.
  + **Delete**: Confirmation dialog before removal.
  + **Search**: Filters users by name/email.
* **Clean Architecture**: Separates DB operations from GUI logic.

**Example Flow**:

1. Admin adds a user → appears in Treeview.
2. Admin searches for "John" → filters results.
3. Admin updates/deletes a selected user → DB reflects changes.

…………………………………………………………………………………………………..**3.** **Source Code:**

import tkinter as tk

from tkinter import messagebox, ttk

import sqlite3

class UserManagementSystem:

    def \_\_init\_\_(self, root):

        self.root = root

        self.root.title("User Management System")

        self.root.geometry("800x600")

        # Initialize database

        self.initialize\_db()

        # Create GUI components

        self.create\_widgets()

        # Load initial data

        self.load\_users()

    def initialize\_db(self):

        """Initialize the SQLite database and create table if not exists"""

        self.conn = sqlite3.connect('user\_management.db')

        self.cursor = self.conn.cursor()

        self.cursor.execute('''

            CREATE TABLE IF NOT EXISTS users (

                id INTEGER PRIMARY KEY AUTOINCREMENT,

                name TEXT NOT NULL,

                email TEXT UNIQUE NOT NULL,

                phone TEXT,

                address TEXT

            )

        ''')

        self.conn.commit()

    def create\_widgets(self):

        """Create all GUI components"""

        # Main frame

        self.main\_frame = tk.Frame(self.root, padx=20, pady=20)

        self.main\_frame.pack(fill=tk.BOTH, expand=True)

        # Title

        tk.Label(self.main\_frame, text="User Management System", font=("Arial", 16)).grid(row=0, column=0, columnspan=4, pady=10)

        # Entry fields

        tk.Label(self.main\_frame, text="Name:").grid(row=1, column=0, sticky=tk.W)

        self.name\_entry = tk.Entry(self.main\_frame, width=30)

        self.name\_entry.grid(row=1, column=1, padx=5, pady=5)

        tk.Label(self.main\_frame, text="Email:").grid(row=2, column=0, sticky=tk.W)

        self.email\_entry = tk.Entry(self.main\_frame, width=30)

        self.email\_entry.grid(row=2, column=1, padx=5, pady=5)

        tk.Label(self.main\_frame, text="Phone:").grid(row=3, column=0, sticky=tk.W)

        self.phone\_entry = tk.Entry(self.main\_frame, width=30)

        self.phone\_entry.grid(row=3, column=1, padx=5, pady=5)

        tk.Label(self.main\_frame, text="Address:").grid(row=4, column=0, sticky=tk.W)

        self.address\_entry = tk.Entry(self.main\_frame, width=30)

        self.address\_entry.grid(row=4, column=1, padx=5, pady=5)

        # Buttons

        self.add\_btn = tk.Button(self.main\_frame, text="Add User", command=self.add\_user)

        self.add\_btn.grid(row=5, column=0, pady=10)

        self.update\_btn = tk.Button(self.main\_frame, text="Update User", command=self.update\_user)

        self.update\_btn.grid(row=5, column=1, pady=10)

        self.delete\_btn = tk.Button(self.main\_frame, text="Delete User", command=self.delete\_user)

        self.delete\_btn.grid(row=5, column=2, pady=10)

        self.clear\_btn = tk.Button(self.main\_frame, text="Clear Fields", command=self.clear\_fields)

        self.clear\_btn.grid(row=5, column=3, pady=10)

        # Search

        tk.Label(self.main\_frame, text="Search:").grid(row=6, column=0, sticky=tk.W)

        self.search\_entry = tk.Entry(self.main\_frame, width=30)

        self.search\_entry.grid(row=6, column=1, padx=5, pady=5)

        self.search\_btn = tk.Button(self.main\_frame, text="Search", command=self.search\_users)

        self.search\_btn.grid(row=6, column=2, padx=5)

        # Treeview for displaying users

        self.tree = ttk.Treeview(self.main\_frame, columns=("ID", "Name", "Email", "Phone", "Address"), show="headings")

        self.tree.grid(row=7, column=0, columnspan=4, pady=10, sticky=tk.NSEW)

        # Configure treeview columns

        self.tree.heading("ID", text="ID")

        self.tree.heading("Name", text="Name")

        self.tree.heading("Email", text="Email")

        self.tree.heading("Phone", text="Phone")

        self.tree.heading("Address", text="Address")

        self.tree.column("ID", width=50)

        self.tree.column("Name", width=150)

        self.tree.column("Email", width=200)

        self.tree.column("Phone", width=100)

        self.tree.column("Address", width=200)

        # Add scrollbar

        scrollbar = ttk.Scrollbar(self.main\_frame, orient=tk.VERTICAL, command=self.tree.yview)

        self.tree.configure(yscroll=scrollbar.set)

        scrollbar.grid(row=7, column=4, sticky=tk.NS)

        # Bind treeview selection

        self.tree.bind("<<TreeviewSelect>>", self.on\_tree\_select)

    def load\_users(self):

        """Load all users from database into treeview"""

        # Clear current items

        for item in self.tree.get\_children():

            self.tree.delete(item)

        # Fetch all users

        self.cursor.execute("SELECT \* FROM users")

        users = self.cursor.fetchall()

        # Insert into treeview

        for user in users:

            self.tree.insert("", tk.END, values=user)

    def add\_user(self):

        """Add a new user to the database"""

        name = self.name\_entry.get()

        email = self.email\_entry.get()

        phone = self.phone\_entry.get()

        address = self.address\_entry.get()

        if not name or not email:

            messagebox.showerror("Error", "Name and Email are required!")

            return

        try:

            self.cursor.execute(

                "INSERT INTO users (name, email, phone, address) VALUES (?, ?, ?, ?)",

                (name, email, phone, address)

            )

            self.conn.commit()

            messagebox.showinfo("Success", "User added successfully!")

            self.load\_users()

            self.clear\_fields()

        except sqlite3.IntegrityError:

            messagebox.showerror("Error", "Email already exists!")

    def update\_user(self):

        """Update selected user in the database"""

        selected\_item = self.tree.selection()

        if not selected\_item:

            messagebox.showerror("Error", "Please select a user to update!")

            return

        user\_id = self.tree.item(selected\_item)['values'][0]

        name = self.name\_entry.get()

        email = self.email\_entry.get()

        phone = self.phone\_entry.get()

        address = self.address\_entry.get()

        if not name or not email:

            messagebox.showerror("Error", "Name and Email are required!")

            return

        try:

            self.cursor.execute(

                "UPDATE users SET name=?, email=?, phone=?, address=? WHERE id=?",

                (name, email, phone, address, user\_id)

            )

            self.conn.commit()

            messagebox.showinfo("Success", "User updated successfully!")

            self.load\_users()

        except sqlite3.IntegrityError:

            messagebox.showerror("Error", "Email already exists!")

    def delete\_user(self):

        """Delete selected user from the database"""

        selected\_item = self.tree.selection()

        if not selected\_item:

            messagebox.showerror("Error", "Please select a user to delete!")

            return

        user\_id = self.tree.item(selected\_item)['values'][0]

        if messagebox.askyesno("Confirm", "Are you sure you want to delete this user?"):

            self.cursor.execute("DELETE FROM users WHERE id=?", (user\_id,))

            self.conn.commit()

            messagebox.showinfo("Success", "User deleted successfully!")

            self.load\_users()

            self.clear\_fields()

    def search\_users(self):

        """Search users by name or email"""

        search\_term = self.search\_entry.get()

        # Clear current items

        for item in self.tree.get\_children():

            self.tree.delete(item)

        # Search in database

        self.cursor.execute(

            "SELECT \* FROM users WHERE name LIKE ? OR email LIKE ?",

            (f"%{search\_term}%", f"%{search\_term}%")

        )

        users = self.cursor.fetchall()

        # Insert results into treeview

        for user in users:

            self.tree.insert("", tk.END, values=user)

    def on\_tree\_select(self, event):

        """Populate entry fields when user is selected in treeview"""

        selected\_item = self.tree.selection()

        if selected\_item:

            user\_data = self.tree.item(selected\_item)['values']

            self.clear\_fields()

            self.name\_entry.insert(0, user\_data[1])

            self.email\_entry.insert(0, user\_data[2])

            self.phone\_entry.insert(0, user\_data[3])

            self.address\_entry.insert(0, user\_data[4])

    def clear\_fields(self):

        """Clear all entry fields"""

        self.name\_entry.delete(0, tk.END)

        self.email\_entry.delete(0, tk.END)

        self.phone\_entry.delete(0, tk.END)

        self.address\_entry.delete(0, tk.END)

        self.search\_entry.delete(0, tk.END)

        self.tree.selection\_remove(self.tree.selection())

    def \_\_del\_\_(self):

        """Close database connection when object is destroyed"""

        if hasattr(self, 'conn'):

            self.conn.close()

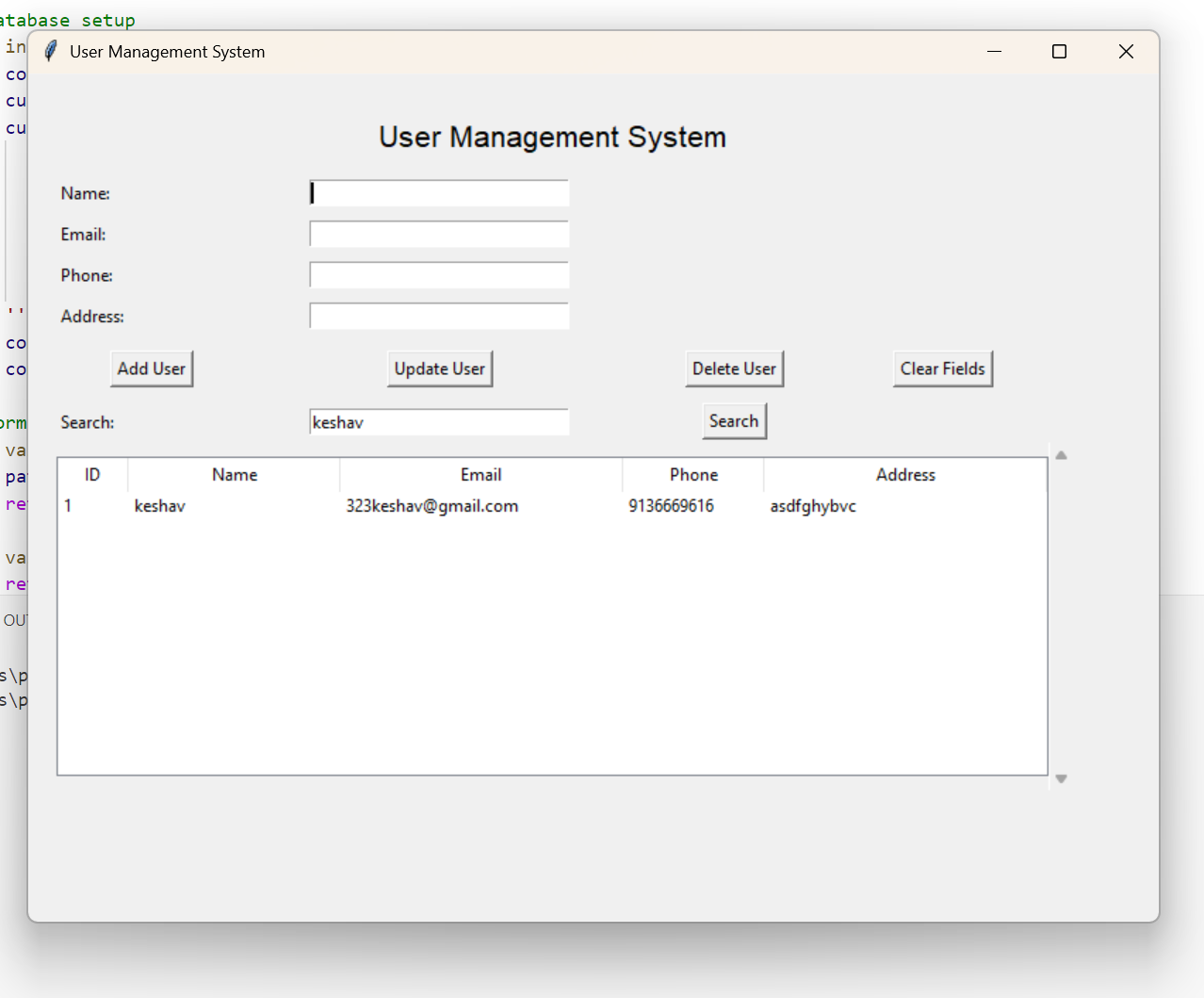
if \_\_name\_\_ == "\_\_main\_\_":

    root = tk.Tk()

    app = UserManagementSystem(root)

    root.mainloop()

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



**Python – File handling**

[7] Python – File opening, reading, copying, content searching, etc

……………………………………………………………………………………

**1. Title: Implement a program to accept file content from the user and then display/read file content using 3 different approaches.**

…………………………………………………………………………………………………..**2.** **Theory:**

**Concept**: File handling with different reading approaches  
**Key Elements**:

* **Three reading methods**:
  1. read(): Reads entire file content at once
  2. readlines(): Reads file line-by-line into a list
  3. pathlib: Modern file system path operations (create/write/read)
* **Error handling**: try-except for missing files
* **Interactive menu**: User can choose reading method or create new files
* **File creation**: Dynamic content input with line-by-line writing

…………………………………………………………………………………………………..**3.** **Source Code:**

import os

from pathlib import Path

def approach\_1\_read():

    """Approach 1: Using read() method"""

    print("\n=== Approach 1: Using read() ===")

    try:

        with open('user\_file.txt', 'r') as file:

            content = file.read()

        print("File content:")

        print(content)

    except FileNotFoundError:

        print("File not found. Please create the file first using Approach 3.")

def approach\_2\_readlines():

    """Approach 2: Using readlines() method"""

    print("\n=== Approach 2: Using readlines() ===")

    try:

        with open('user\_file.txt', 'r') as file:

            lines = file.readlines()

        print("File content (line by line):")

        for i, line in enumerate(lines, 1):

            print(f"Line {i}: {line.strip()}")

    except FileNotFoundError:

        print("File not found. Please create the file first using Approach 3.")

def approach\_3\_pathlib():

    """Approach 3: Using pathlib (create file and read)"""

    print("\n=== Approach 3: Using pathlib ===")

    print("Enter your file content (press Enter on an empty line to finish):")

    content\_lines = []

    while True:

        line = input()

        if line == "":

            break

        content\_lines.append(line + "\n")  # Add newline character

    # Write to file using pathlib

    file\_path = Path('user\_file.txt')

    file\_path.write\_text(''.join(content\_lines))

    print(f"File saved to: {file\_path.absolute()}")

    # Read back using pathlib

    print("\nFile content from pathlib:")

    print(file\_path.read\_text())

def main():

    while True:

        print("\n=== File Content Reader ===")

        print("1. Display using read()")

        print("2. Display using readlines()")

        print("3. Create file and read using pathlib")

        print("4. Exit")

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

        if choice == '1':

            approach\_1\_read()

        elif choice == '2':

            approach\_2\_readlines()

        elif choice == '3':

            approach\_3\_pathlib()

        elif choice == '4':

            print("Exiting program...")

            break

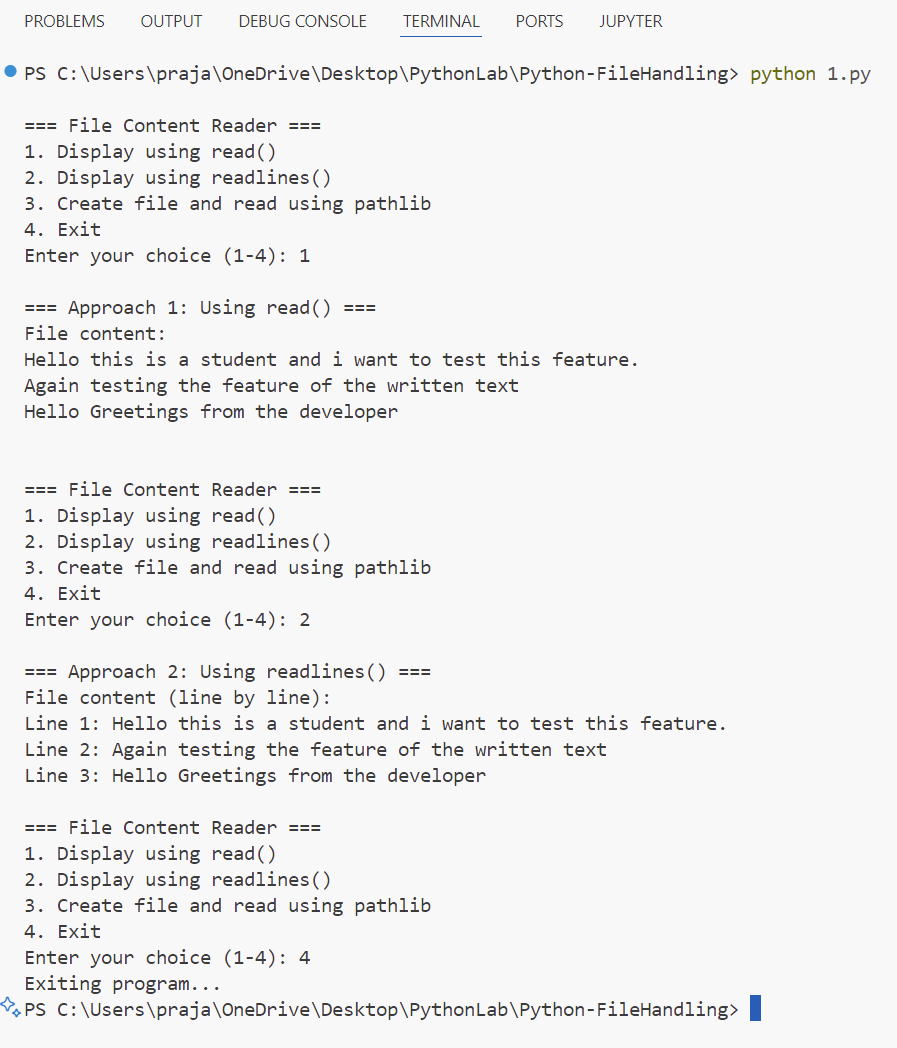
        else:

            print("Invalid choice! Please try again.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Implement a program that reads a large log file (e.g., server.txt), finds ‘error’ word and counts the occurrences and keeps track of line number for each occurrence, and finally saves extracted information into a new file called error.txt.**

…………………………………………………………………………………………………..**2.** **Theory:**

**2.py - Log File Analyzer**

**Concept**: Log file processing and error detection  
**Key Elements**:

* **Error scanning**: Case-insensitive search for "error" in log lines
* **Result reporting**:
  + Counts total errors
  + Records error lines with line numbers
* **Output generation**: Creates error.txt with analysis results
* **Robust error handling**: Catches file access issues and unexpected errors

*Example Workflow*:

1. Scans server.txt for errors
2. Generates error.txt with:
   * Total error count
   * Each error line with line number
3. Handles missing files gracefully

**Supporting Files**

* **user\_file.txt**: Sample file created by Approach 3 in 1.py
* **error.txt**: Analysis output from 2.py (empty when no errors found)

…………………………………………………………………………………………………..**3.** **Source Code:**

def analyze\_log\_file(input\_file, output\_file):

    """

    Analyzes a log file for 'error' occurrences and saves results to a new file

    :param input\_file: Path to the log file

    :param output\_file: Path to save error information

    """

    error\_count = 0

    error\_lines = []

    try:

        with open(input\_file, 'r', encoding='utf-8') as log\_file:

            for line\_num, line in enumerate(log\_file, 1):

                # Case-insensitive search for 'error'

                if 'error' in line.lower():

                    error\_count += 1

                    error\_lines.append((line\_num, line.strip()))

        # Save results to error.txt

        with open(output\_file, 'w', encoding='utf-8') as err\_file:

            err\_file.write(f"Total 'error' occurrences: {error\_count}\n")

            err\_file.write("=" \* 50 + "\n")

            for line\_num, line\_content in error\_lines:

                err\_file.write(f"Line {line\_num}: {line\_content}\n")

        print(f"Analysis complete. Found {error\_count} errors.")

        print(f"Results saved to {output\_file}")

    except FileNotFoundError:

        print(f"Error: The file {input\_file} was not found.")

    except Exception as e:

        print(f"An error occurred: {str(e)}")

def main():

    # File paths

    log\_file = 'server.txt'  # Change to your log file path

    error\_file = 'error.txt'

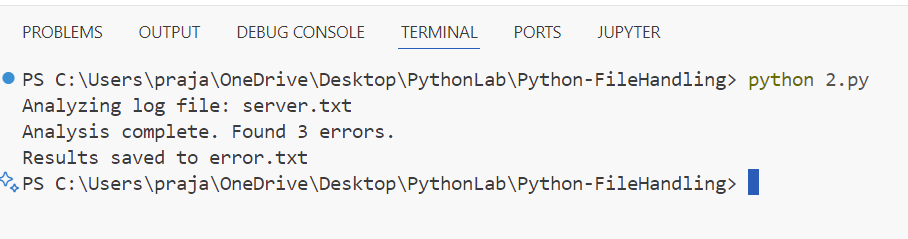
    print(f"Analyzing log file: {log\_file}")

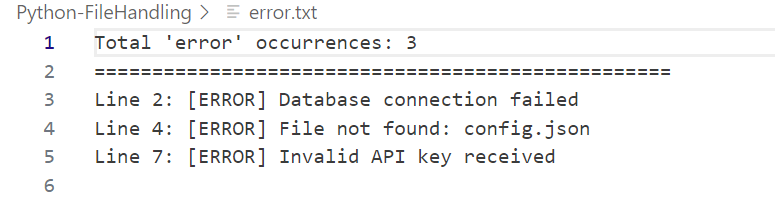
    analyze\_log\_file(log\_file, error\_file)

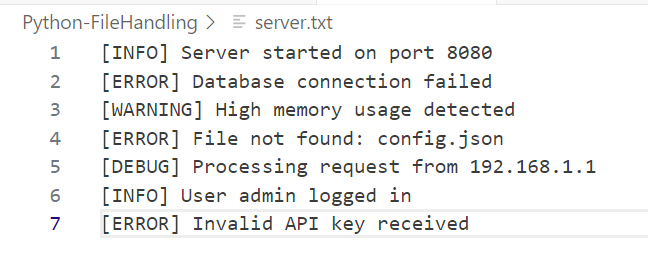
if \_\_name\_\_ == "\_\_main\_\_":

    main()

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**







…………………………………………………………………………………………………..

**Python – Multithreading**

[8] Python – Multithread programming

……………………………………………………………………………………

**1. Title: Implement a program to demonstrate working of multiple-threads for a specific case scenario (food ordering, airport luggage management, ATM, etc.)**

…………………………………………………………………………………………………..**2.** **Theory:**

**Concept**: Multi-threaded order processing with thread synchronization  
**Key Elements**:

* **Producer-Consumer Pattern**:
  + Customer threads (producers) place orders into a shared Queue
  + Chef threads (consumers) process orders from the queue
* **Thread Synchronization**:
  + Lock (order\_lock) protects shared order\_status dictionary
  + Queue handles safe order passing between threads
* **Monitoring**:
  + Daemon thread (display\_system\_status) shows real-time order status
* **Real-world Simulation**:
  + Random delays simulate thinking/cooking times
  + Order tracking from "Placed" → "Preparing" → "Ready"

…………………………………………………………………………………………………..**3.** **Source Code:**

import threading

import time

import random

from queue import Queue

# Shared data structures

order\_queue = Queue()

order\_status = {}

order\_lock = threading.Lock()

class Customer(threading.Thread):

    def \_\_init\_\_(self, customer\_id):

        threading.Thread.\_\_init\_\_(self)

        self.customer\_id = customer\_id

        self.orders = ["Pizza", "Burger", "Pasta", "Salad", "Sandwich"]

    def run(self):

        order = random.choice(self.orders)

        order\_time = random.randint(1, 3)

        # Simulate time taken to decide order

        time.sleep(order\_time)

        with order\_lock:

            order\_id = f"ORD-{self.customer\_id}-{time.time\_ns()}"

            order\_queue.put((order\_id, order))

            order\_status[order\_id] = "Placed"

            print(f"Customer {self.customer\_id} ordered {order} (Order ID: {order\_id})")

class Chef(threading.Thread):

    def \_\_init\_\_(self, chef\_id):

        threading.Thread.\_\_init\_\_(self)

        self.chef\_id = chef\_id

    def run(self):

        while True:

            if not order\_queue.empty():

                order\_id, order = order\_queue.get()

                with order\_lock:

                    order\_status[order\_id] = "Preparing"

                    print(f"Chef {self.chef\_id} is preparing {order} (Order ID: {order\_id})")

                # Simulate cooking time

                cook\_time = random.randint(2, 5)

                time.sleep(cook\_time)

                with order\_lock:

                    order\_status[order\_id] = "Ready"

                    print(f"Chef {self.chef\_id} completed {order} (Order ID: {order\_id}) in {cook\_time} secs")

                order\_queue.task\_done()

            else:

                # No orders left, chefs take a break

                time.sleep(1)

def display\_system\_status():

    while True:

        with order\_lock:

            print("\n=== Current System Status ===")

            print(f"Orders in queue: {order\_queue.qsize()}")

            for order\_id, status in order\_status.items():

                print(f"{order\_id}: {status}")

            print("===========================\n")

        time.sleep(3)

def main():

    # Start system status monitor

    status\_thread = threading.Thread(target=display\_system\_status, daemon=True)

    status\_thread.start()

    # Create chefs

    chefs = [Chef(i+1) for i in range(2)]  # 2 chefs

    for chef in chefs:

        chef.daemon = True

        chef.start()

    # Simulate customers placing orders

    customers = [Customer(i+1) for i in range(5)]  # 5 customers

    for customer in customers:

        customer.start()

        time.sleep(random.randint(1, 2))  # Stagger customer orders

    # Wait for all customers to place orders

    for customer in customers:

        customer.join()

    # Wait for all orders to be processed

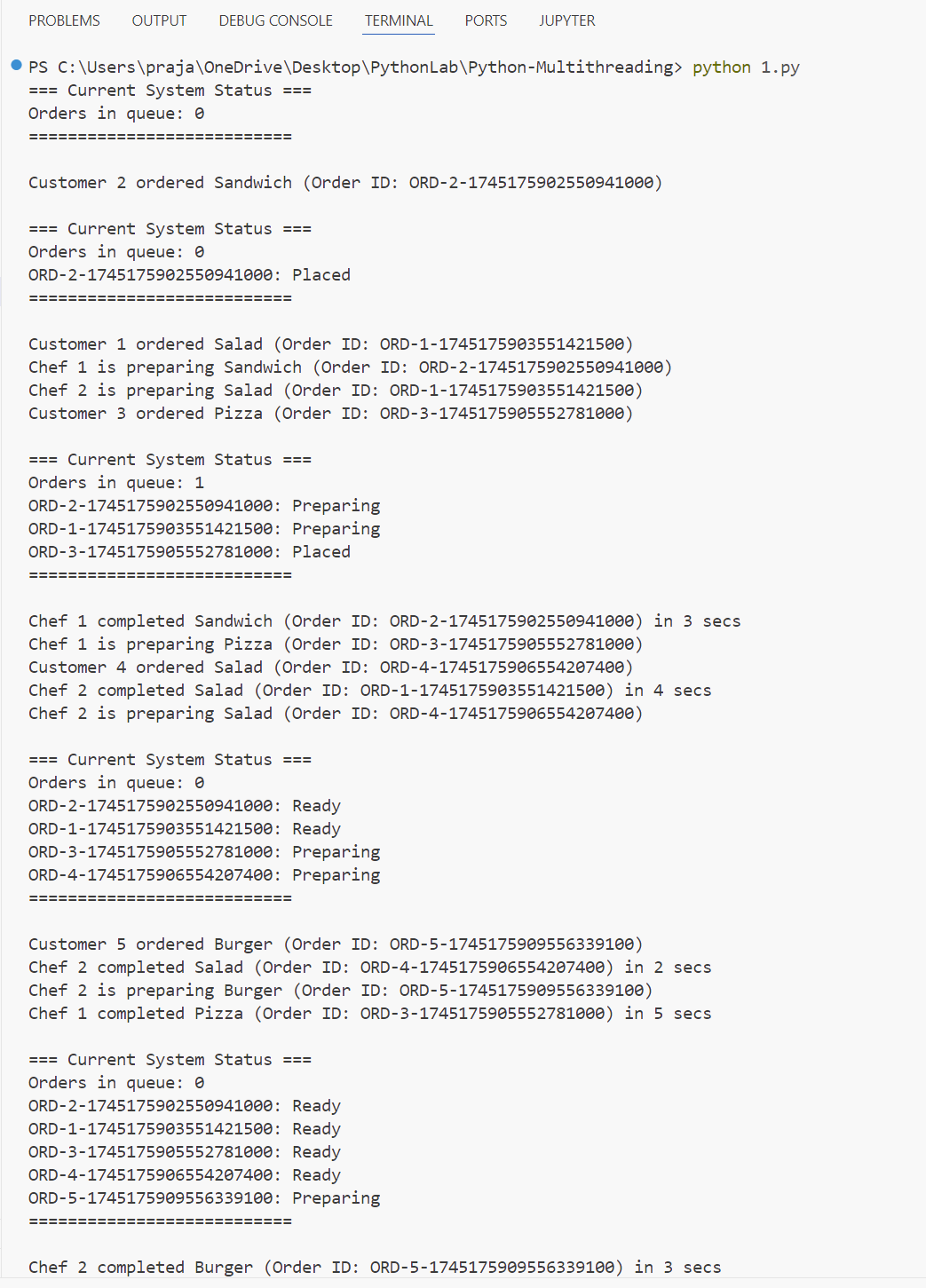
    order\_queue.join()

    print("\nAll orders processed! System shutting down.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

…………………………………………………………………………………………………..**4.** **Sample Input/Output:**



…………………………………………………………………………………………………..

……………………………………………………………………………………

**1. Title: Implement a multithreading program for banking scenario to demonstrate RACE condition – 1) without and, ii) with LOCK** …………………………………………………………………………………………………..**2.** **Theory:**

**Concept**: Thread synchronization pitfalls and solutions  
**Key Elements**:

1. **Race Condition Demo**:
   * Without locks, simultaneous withdrawals cause negative balance  
     *Output*: Final balance: $-20 (Due to race condition)
2. **Lock Solution**:
   * BankAccountWithLock uses threading.Lock for atomic operations  
     *Output*: Final balance: $0 (Correct with locks)

* **Thread Safety**:
  + with lock: blocks ensure exclusive access during balance checks/updates
* **Educational Contrast**:
  + Shows identical operations with/without locks to demonstrate synchronization importance

…………………………………………………………………………………………………..**3.** **Source Code:**

    import threading

    import time

    class BankAccount:

        def \_\_init\_\_(self, use\_lock=True):

            self.balance = 100  # Initial balance

            self.use\_lock = use\_lock

            if use\_lock:

                self.lock = threading.Lock()

        def withdraw(self, amount):

            if self.use\_lock:

                with self.lock:

                    return self.\_withdraw(amount)

            else:

                return self.\_withdraw(amount)

        def \_withdraw(self, amount):

            # Simulate processing time

            time.sleep(0.1)

            if self.balance >= amount:

                self.balance -= amount

                return True

            return False

    def customer\_action(account, customer\_id):

        for attempt in range(10):

            # Check balance (with lock if account uses it)

            if account.use\_lock:

                with account.lock:

                    current\_balance = account.balance

                    time.sleep(0.001)  # Simulate decision delay

            else:

                current\_balance = account.balance

                time.sleep(0.001)

            if current\_balance >= 20:

                success = account.withdraw(20)

                if success:

                    print(f"Customer-{customer\_id} withdrew $20 (Attempt {attempt + 1})")

                else:

                    print(f"Customer-{customer\_id} - Withdrawal failed (Attempt {attempt + 1})")

            else:

                print(f"Customer-{customer\_id} - Insufficient funds (Attempt {attempt + 1})")

                break

    def banking\_demo(use\_lock=True):

        account = BankAccount(use\_lock=use\_lock)

        print("\n" + "="\*50)

        print(f"=== {'With Lock' if use\_lock else 'Without Lock'} Demo ===")

        print(f"Initial balance: ${account.balance}\n")

        customers = [threading.Thread(target=customer\_action, args=(account, i)) for i in range(5)]

        start\_time = time.time()

        for customer in customers:

            customer.start()

        for customer in customers:

            customer.join()

        duration = time.time() - start\_time

        print(f"\nFinal balance: ${account.balance}")

        print(f"Completed in {duration:.2f} seconds")

        print("="\*50 + "\n")

    if \_\_name\_\_ == "\_\_main\_\_":

        # Run demo without lock first

        print("WARNING: Running without locks will show race condition effects")

        banking\_demo(use\_lock=False)

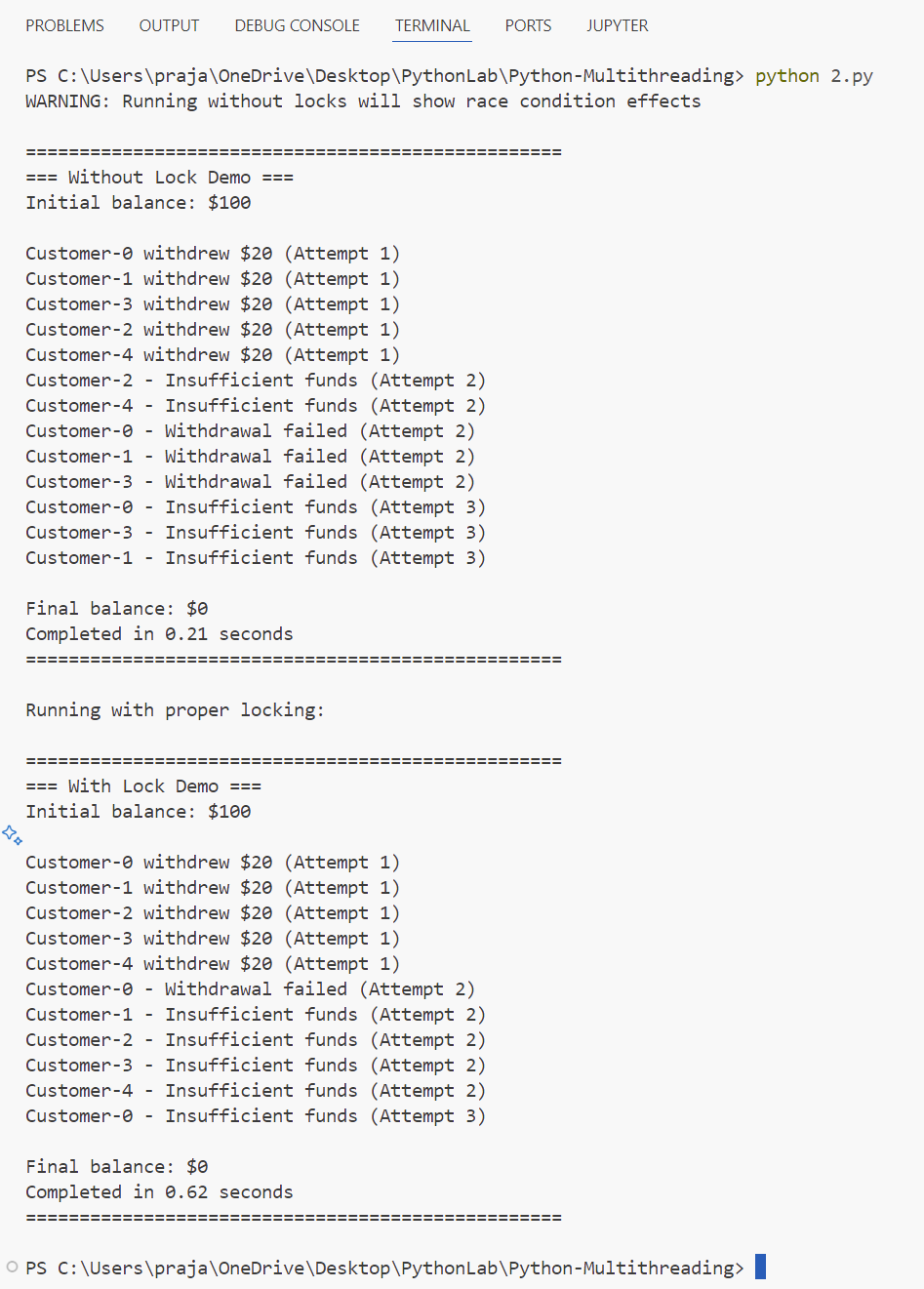
        time.sleep(2)

        # Then run with proper locking

        print("Running with proper locking:")

        banking\_demo(use\_lock=True)

…………………………………………………………………………………………………..

**4.** **Sample Input/Output:**

**Data Analysis and Visualization**

[9] Python – Exploratory Data Analysis – pandas, dataframe, descriptive and correlation analysis, matplotlib, etc.

……………………………………………………………………………………

**1. Title: Select an appropriate dataset (Kaggle) and perform EDA to answer/perform following. (Jupyter Notebook).**

…………………………………………………………………………………………………..**2.** **Theory:**

**Concept Implemented:**  
The notebook performs an **Exploratory Data Analysis (EDA)** on a dataset of engineering colleges in India. The analysis includes:

1. **Data Loading and Initial Inspection**: Loading the dataset and displaying the first and last few rows to understand its structure.
2. **Descriptive Statistics**: Calculating basic statistics to summarize the data.
3. **Handling Missing Values**: Identifying and addressing missing values in the dataset.
4. **Data Visualization**: Using heatmaps to visualize missing data patterns.
5. **Variable Analysis**: Identifying independent and dependent variables, and analyzing their distributions.

**Programming Elements Used:**

1. **Pandas (**pd**)**: Used for data manipulation, including loading the dataset (pd.read\_csv), displaying data (df.head(), df.tail()), and calculating descriptive statistics (df.describe()).
2. **Matplotlib (**plt**) and Seaborn (**sns**)**: Used for data visualization, such as creating heatmaps to visualize missing values (sns.heatmap).
3. **Data Cleaning**: Handling missing values by filling numeric columns with their mean (df['Average Fees'].fillna(df['Average Fees'].mean())).
4. **Basic Data Analysis**: Identifying variables with the most missing values and summarizing the dataset's structure.

This EDA provides a foundational understanding of the dataset, which is crucial for further analysis or modeling tasks. The notebook leverages Python's data analysis and visualization libraries to explore and clean the data effectively.

…………………………………………………………………………………………………..**3.** **Source Code:**

# EDA Project - Engineering Colleges Dataset

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Load the dataset

df = pd.read\_csv("engineering colleges in India.csv")

# Display top 5 and last 5 rows

display(df.head())

display(df.tail())

# Count of independent and dependent variables

dependent\_var = 'Rating'

independent\_vars = [col for col in df.columns if col != dependent\_var]

print("Number of Independent Variables:", len(independent\_vars))

print("Number of Dependent Variables:", 1)

# Descriptive statistics

display(df.describe(include='all'))

# Convert numeric columns to numeric dtype

numeric\_cols = ['Total Student Enrollments', 'Total Faculty', 'Established Year', 'Average Fees']

for col in numeric\_cols:

df[col] = pd.to\_numeric(df[col], errors='coerce')

# Independent variable with minimum average

avg\_values = df[numeric\_cols].mean()

print("Independent Variable with Minimum Average Value:", avg\_values.idxmin())

# Independent variable with highest standard deviation

std\_values = df[numeric\_cols].std()

print("Independent Variable with Highest Standard Deviation:", std\_values.idxmax())

# Total count of missing values in each column

missing\_counts = df[independent\_vars].isnull().sum()

print("\nMissing Values in Independent Variables:\n", missing\_counts)

# Visualize missing values

plt.figure(figsize=(10,6))

sns.heatmap(df[independent\_vars].isnull(), cbar=False, yticklabels=False, cmap='viridis')

plt.title("Missing Values Heatmap")

plt.show()

# Variable with max missing values

print("Independent Variable with Maximum Missing Values:", missing\_counts.idxmax())

# Replace missing values in a numeric independent variable with average (e.g. 'Average Fees')

df['Average Fees'].fillna(df['Average Fees'].mean(), inplace=True)

# Histogram of an independent variable (e.g. 'Established Year')

plt.figure(figsize=(8,5))

df['Established Year'].dropna().astype(int).hist(bins=30)

plt.title("Frequency Distribution of Established Year")

plt.xlabel("Year")

plt.ylabel("Frequency")

plt.show()

# Box-plot to identify outliers (e.g. 'Average Fees')

plt.figure(figsize=(8,5))

sns.boxplot(data=df, x='Average Fees')

plt.title("Boxplot for Average Fees")

plt.show()

# Line chart for correlation (e.g. between 'Established Year' and 'Average Fees')

df\_sorted = df.sort\_values('Established Year')

plt.figure(figsize=(10,6))

plt.plot(df\_sorted['Established Year'], df\_sorted['Average Fees'])

plt.title("Line Chart: Established Year vs Average Fees")

plt.xlabel("Established Year")

plt.ylabel("Average Fees")

plt.show()

# Correlation Matrix

correlation\_matrix = df[numeric\_cols].corr()

sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm')

plt.title("Correlation Matrix")

plt.show()

# Scatter plots for two pairs

# Assuming high positive correlation: 'Total Faculty' vs 'Total Student Enrollments'

sns.scatterplot(data=df, x='Total Faculty', y='Total Student Enrollments')

plt.title("Scatter Plot - Total Faculty vs Student Enrollments")

plt.show()

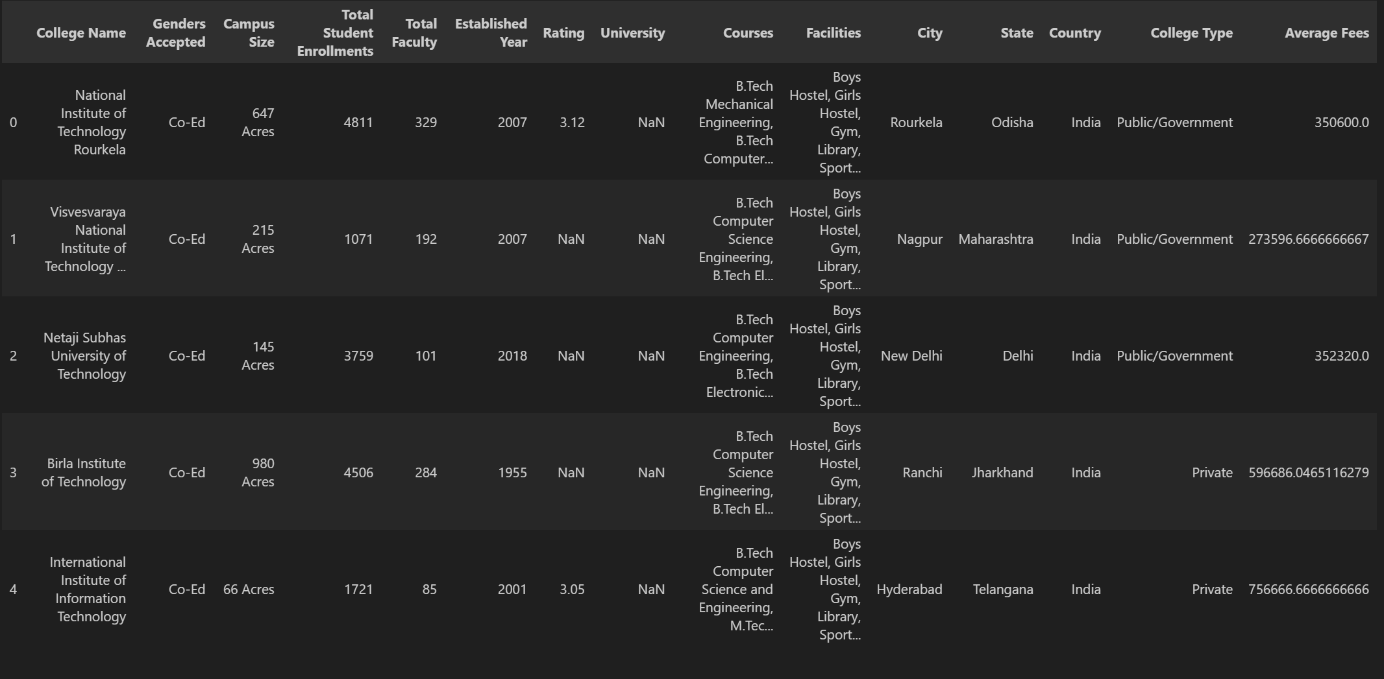
# Assuming weak/negative correlation: 'Average Fees' vs 'Rating'

sns.scatterplot(data=df, x='Average Fees', y='Rating')

plt.title("Scatter Plot - Average Fees vs Rating")

plt.show()

…………………………………………………………………………………………………..

**4.** **Sample Input/Output:**

