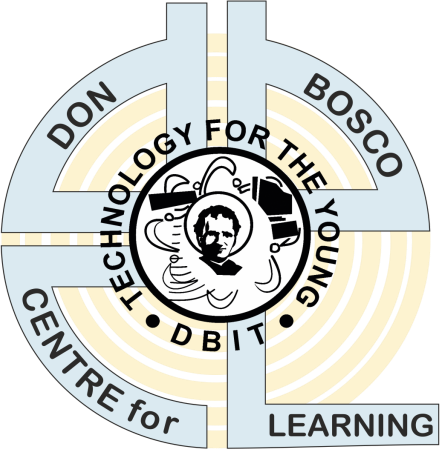
Python Lab Journal

****

Name: Sahil Vikas Khude

Roll No. 31

Course/Lab: Python Lab (ITL404)

Semester: IV

Academic Year: 2024-25

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

Department of Information Technology

The Bombay Salesian Society’s

Don Bosco Institute of Technology, Mumbai 400070.

|  |  |  |
| --- | --- | --- |
| **Sr No.** | **Title** | **Page No.** |
| 1 | Print the grades of students after accepting the marks for 5 subjects from the user. |  |
| 2 | Accept the range from the user and then display all prime numbers between the given range. |  |
| 3 | Print an n digit number in reverse order. |  |
| 4 | Display a pattern formed with numbers. |  |
| 5 | Display a pattern formed with ‘\*’ using nested looping. |  |
| 6 | Display the count of elements of different data types present in a LIST. |  |
| 7 | Check for a given value in the LIST; display total count of occurrences along with the index positions of each occurrence. |  |
| 8 | Perform sorting of LIST elements; Press 1 for ascending order and, Press 2 for descending order. |  |
| 9 | Add elements of List2 in List1, then display the updated List. |  |
| 10 | Demonstrate LIST comprehensions using two examples. |  |
| 11 | Display count of vowels, consonants, blank spaces, special symbols and digits in a given STRING. |  |
| 12 | Calculate library fine based on 2 conditions, a) Book return date and b) Book condition. |  |
| 13 | 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. |  |
| 14 | Display Fibonacci series elements (n) using recursive method. |  |
| 15 | Demonstrate passing and returning a List to/from a user defined function. |  |
| 16 | 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). |  |
| 17 | Write a program to demonstrate multiple exceptions handling, specifically, NameError, IndexError, and ZeroDivisionError. |  |
| 18 | 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.). |  |
| 19 | Write an object oriented program to demonstrate working of default and parameterized constructors. |  |
| 20 | Write an object oriented menu driven program to perform banking operations (New account, Deposit, Withdraw, Balance, Show all, Exit). |  |
| 21 | Write object oriented program to implement Single level inheritance by considering appropriate real life scenarios (use super(), \_\_init\_\_, \_\_str\_\_, and \_\_name\_\_ ). |  |
| 22 | Write object oriented program to implement Multilevel inheritance by considering appropriate real life scenarios (use super(), \_\_init\_\_, \_\_str\_\_, and \_\_name\_\_ ) |  |
| 23 | Design a working interface for the login and registration process with proper form validations and database connection using tkinter and SQLite. |  |
| 24 | Perform CRUD database operations using a menu driven program (User Management - Add, Show, Delete, Update and Search). |  |
| 25 | Implement a program to accept file content from the user and then display/read file content using 3 different approaches. |  |
| 26 | 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. |  |
| 27 | Implement a program to demonstrate working of multiple-threads for a specific case scenario (food ordering, airport luggage management, ATM, etc.). |  |
| 28 | Implement a multithreading program for banking scenario to demonstrate RACE condition without LOCK. |  |
| 29 | Implement a multithreading program for banking scenario to demonstrate RACE condition with LOCK. |  |
| 30 | Select an appropriate dataset (Kaggle) and perform EDA. |  |

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

**Theory:**

In this program, we are taking input of marks for 5 subjects from the user using a for loop and storing them in a list. We then calculate the total and average of these marks using the built-in sum() function and basic arithmetic.

Based on the average marks, we assign grades using if-elif-else conditional statements. These conditions help us decide what grade the student falls into. Finally, we print the total, average, and the grade.

Key Python concepts used here:

* Input and typecasting
* List to store values
* Loops (for loop)
* Conditional statements (if-elif-else)
* Basic arithmetic operations

**Source Code:**

marks = []

for i in range(1, 6):

    mark = float(input(f'Enter marks of subject { i }: '))

    marks.append(mark)

total = 0

for i in marks:

    total = total + i

average = total / 5

if average >= 90:

    grade = 'A+'

elif average >= 80:

    grade = 'A'

elif average >= 70:

    grade = 'B+'

elif average >= 60:

    grade = 'B'

elif average >= 50:

    grade = 'C'

elif average >= 40:

    grade = 'D'

else:

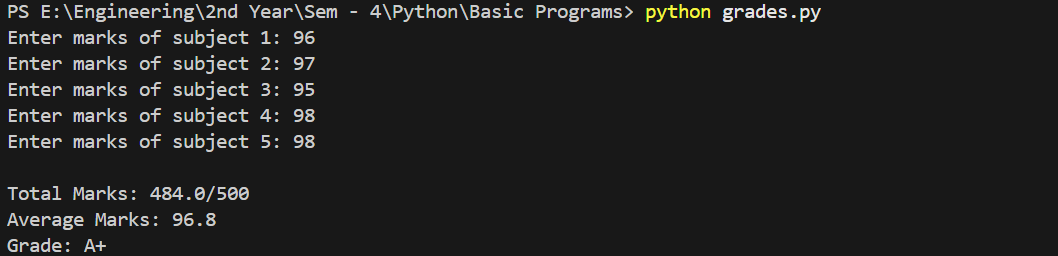
    grade = 'F'

print(f"\nTotal Marks: {total}/500")

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

print(f"Grade: {grade}")

**Sample Output:**

****

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

**Theory:**

This program finds all prime numbers between a user-defined range. A prime number is a number greater than 1 that has no divisors other than 1 and itself.

The user inputs the starting and ending values of the range. The program then uses a for loop to check each number in this range. For each number, we use another loop to check if it's divisible by any number between 2 and its square root (num \*\* 0.5)—which is a shortcut to reduce unnecessary checks and improve performance.

If no divisors are found, that number is printed as a prime number. The logic is built using:

* Loops (for loop)
* Conditional statements (if-else)
* Modulo operator (%) to check divisibility
* Mathematical optimization using square root method to improve efficiency

**Source Code:**

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

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

print(f'\n Prime numbers between {start} and {end} are: ')

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

    if num > 1:

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

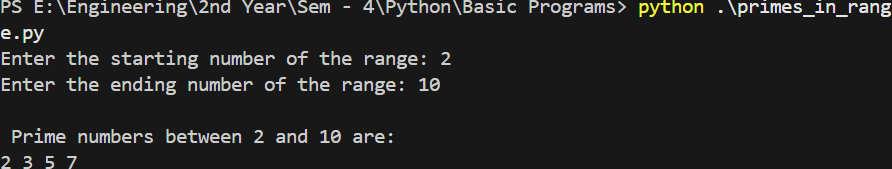
            if num % i == 0:

                break

        else:

                print(num, end=' ')

**Sample Output:**

****

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

**Theory:**

In this program, we accept an n-digit number from the user. Instead of converting it to an integer, we take it as a string so that we can directly reverse it using string slicing. In Python, the slicing syntax [::-1] allows us to reverse any string in a simple and clean way.

This program demonstrates:

* Input handling
* String slicing technique for reversal
* Avoiding complex logic with loops by using Python’s built-in features

**Source Code:**

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

num1 = number

print("The reverse of number", num1, "is ", end='')

while num1 != 0:

    i = num1 % 10

    print(i, end='')

    num1 //= 10

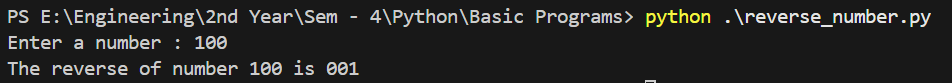
while(num1 != 0):

    i = num1 % 10

    print(i, end='')

    num1 //= 10

**Sample Output:**



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

**Theory:**

In this program, we create two different patterns using nested loops:

1. Number Pattern:
   * For each row, we print numbers starting from 1 up to the row number.
   * This uses two for loops: the outer loop for rows and the inner loop for columns.
2. Star Pattern (\*):
   * For each row, we print that many stars.
   * Again, we use nested loops to control the structure of the triangle.

**Source Code:**

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

**print('\nNumber Pattern')**

**for i in range(1, rows + 1):**

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

**print(j, end=' ')**

**print()**

**print('\nStar Pattern: ')**

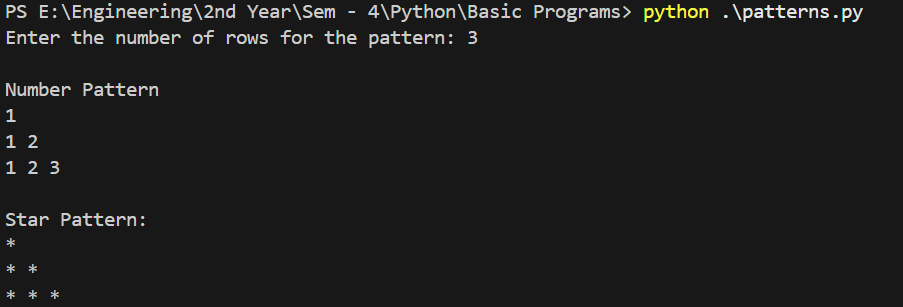
**for i in range(1, rows + 1):**

**for j in range(i):**

**print('\*', end=' ')**

**print()**

**Sample Output:**

****

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

**Theory:**

In this program, we use a list containing elements of multiple data types — like integers, strings, floats, booleans, lists, dictionaries, etc.

To solve this:

* We use a for loop to go through each element.
* The type() function (with \_\_name\_\_) helps identify the data type of each item.
* Lists
* Dictionaries
* Loops
* Built-in type() function
* Conditional logic

**Source Code:**

**mixed\_list = [1, 'string', 2.2, True, None, 42, 'world', False, [1,3], {'a': 1}]**

**type\_count = {}**

**for item in mixed\_list:**

**item\_type = type(item).\_\_name\_\_**

**if item\_type in type\_count:**

**type\_count[item\_type] += 1**

**else:**

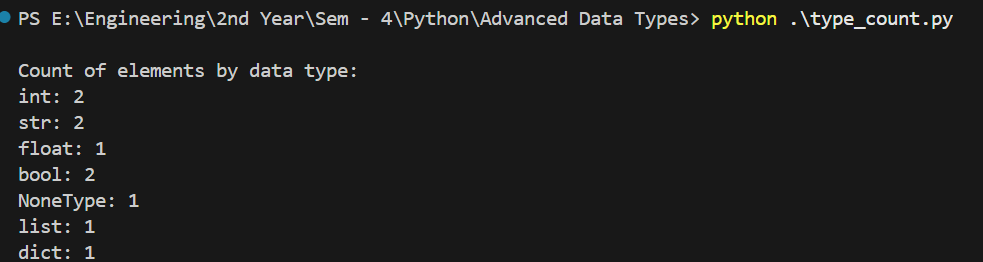
**type\_count[item\_type] = 1**

**print('\nCount of elements by data type: ')**

**for dtype, count in type\_count.items():**

**print(f'{dtype}: {count}')**

**Sample Output:**

****

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

**Theory:**

**This program searches for a user-specified value in a list and finds:**

* **The total number of times the value appears**
* **The index positions at which it occurs**

**We use:**

* **enumerate() function to get both index and value during the loop**
* **A counter variable to track the number of occurrences**
* **A list to store the positions where matches are found**
* **An if-else condition to check and respond accordingly**

**Source Code:**

**my\_list = [10,20,30,10,40,10,50,20,10]**

**search\_value = int(input("Enter the value to search in the list: "))**

**count = 0**

**positions = []**

**for index, value in enumerate(my\_list):**

**if value == search\_value:**

**count += 1**

**positions.append(index)**

**print(f"\nThe value {search\_value} occurred {count} times.")**

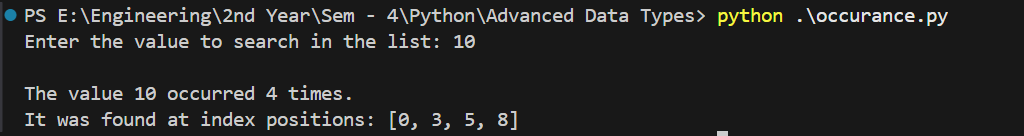
**if count > 0:**

**print("It was found at index positions:", positions)**

**else:**

**print("The value was not found in the list.")**

**Sample Output:**

****

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

**Theory:**

In this program, we sort list elements without using built-in sorting methods.  
We use Bubble Sort — a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they're in the wrong order.

We implemented:

* Ascending order: by swapping if the left element is greater than the right.
* Descending order: by swapping if the left element is smaller than the right.

Key Python concepts used:

* Nested loops
* Swapping using tuple unpacking
* User input-based conditional logic

**Source Code:**

my\_list = [45, 12, 78, 34, 89, 23, 67]

print("Original List: ", my\_list)

choice = int(input("Press 1 for Ascending Order or Press 2 for Descending Order: "))

n = len(my\_list)

if choice == 1:

    for i in range(n):

        for j in range(0, n-i-1):

            if my\_list[j] > my\_list[j+1]:

                my\_list[j], my\_list[j+1] = my\_list[j+1], my\_list[j]

    print("List in Ascending Order:", my\_list)

elif choice == 2:

    for i in range(n):

        for j in range(0, n-i-1):

            if my\_list[j] < my\_list[j+1]:

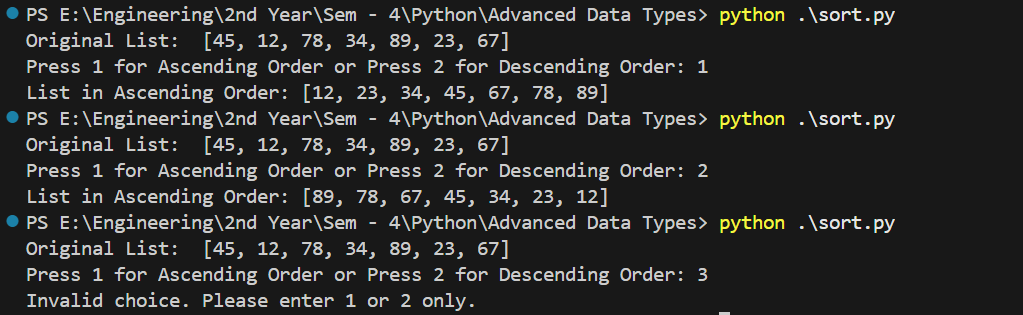
                my\_list[j], my\_list[j+1] = my\_list[j+1], my\_list[j]

    print("List in Ascending Order:", my\_list)

else:

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

**Sample Output:**

****

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

**Theory:**

In this program, we are merging two lists: List2 is added element-by-element into List1.

To do this:

* We use a for loop to iterate through List2
* For each element, we use the append() method of Python lists to add it to List1

This avoids using + or extend() — and manually demonstrates the process of list merging.

Key Python concepts:

* Lists
* Loops
* append() method to add individual elements

**Source Code:**

list1 = [10, 20, 30]

list2 = [40, 50, 60]

print("List1 before adding:", list1)

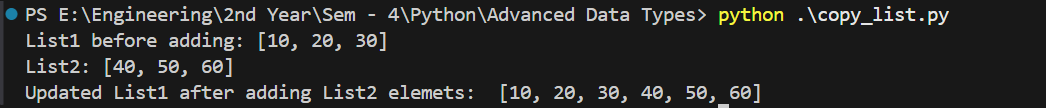
print("List2:", list2)

for element in list2:

list1.append(element)

print("Updated List1 after adding List2 elements:", list1)

**Sample Output:**

****

**Title: Demonstrate LIST comprehensions using two examples.**

**Theory:**

List Comprehension is a short and elegant way to create lists in Python.  
Instead of writing multiple lines using loops, we can write the logic in a single line.

In Example 1, we generate a list of squares using:

Python

In Example 2, we use a condition inside the list comprehension to filter even numbers:

Key Python concepts used:

* List Comprehension syntax
* Looping and conditional filtering
* Mathematical operations within list creation

**Source Code:**

end = int(input("Enter number to find squares till that number: "))

squares = [x\*\*2 for x in range(1, end + 1)]

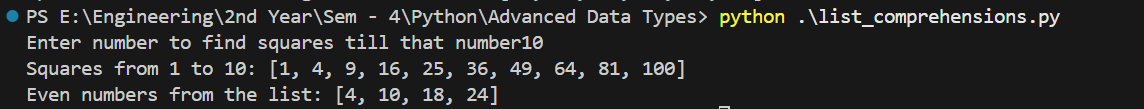
print("Squares from 1 to 10:", squares)

numbers = [1, 4, 7, 10, 15, 18, 21, 24]

even\_numbers = [num for num in numbers if num % 2 == 0]

print("Even numbers from the list:", even\_numbers)

**Sample Output:**

****

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

**Theory:**

This program takes a string input from the user and counts different types of characters:

* Vowels (a, e, i, o, u)
* Consonants (other alphabets)
* Digits (0–9)
* Blank spaces (isspace())
* Special characters (punctuations, symbols etc.)

We use:

* String methods like .isalpha(), .isdigit(), .isspace() to classify characters
* Looping through each character for detailed analysis
* Counter variables for tracking each type

**Source Code:**

text = input("Enter a string: ")

vowels = consonants = spaces = digits = special\_chars = 0

vowel\_set = 'aeiouAEIOU'

for char in text:

    if char in vowel\_set:

        vowels += 1

    elif char.isalpha():

        consonants += 1

    elif char.isdigit():

        digits += 1

    elif char.isspace():

        spaces += 1

    else:

        special\_chars += 1

print("\nAnalysis of the given string:")

print("Vowels:", vowels)

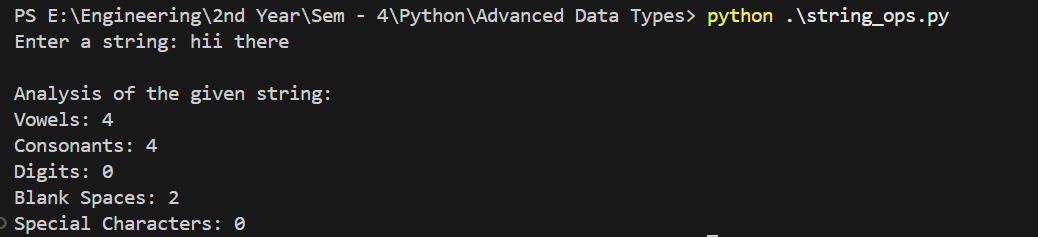
print("Consonants:", consonants)

print("Digits:", digits)

print("Blank Spaces:", spaces**)**

print("Special Characters:", special\_chars)

**Sample Output:**

****

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

**Theory:**

This program demonstrates User Defined Functions (UDF) in Python — a way to break the problem into reusable blocks of logic.

Here, we define a function calculate\_fine() which takes:

* Return date
* Due date
* Book condition

It then:

* Calculates fine for late returns at Rs.2 per day.
* Adds additional charges if the book is damaged (Rs.50) or lost (Rs.200).

**Source Code:**

from datetime import datetime

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

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

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

    fine = 0

    if return\_date > due\_date:

        days\_late = (return\_date - due\_date).days

        fine += days\_late \* 2

    if condition.lower() == 'damaged':

        fine += 50

    elif condition.lower() == 'lost':

        fine += 200

    return fine

return\_date = input("Enter return date (dd-mm-yyyy): ")

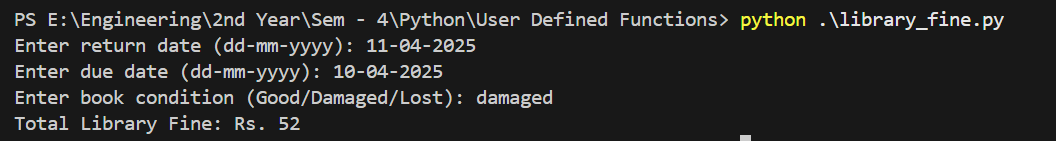
due\_date = input("Enter due date (dd-mm-yyyy): ")

condition = input("Enter book condition (Good/Damaged/Lost): ")

total\_fine = calculate\_fine(return\_date, due\_date, condition)

print("Total Library Fine: Rs.", total\_fine)

**Sample Output:**

****

**Title: Calculate and display the Net Monthly Salary for two categories of employees (Permanent and**

**Temporary) based on the following inputs:**

* **Monthly salary (for permanent)**
* **Hourly rate (for temporary)**
* **Number of present/absent days**

**Theory:**

This program calculates the Net Monthly Salary for:

1. Permanent Employees – based on a fixed monthly salary and days present out of 30.
2. Temporary Employees – based on an hourly rate, fixed 8 working hours per day, and number of present days.

To solve this:

* We used a User Defined Function (calculate\_net\_salary) with optional parameters.
* Used conditional statements to separate logic for permanent and temporary employees.
* For permanent, salary is prorated based on days present.
* For temporary, salary is based on total hours worked (hourly\_rate \* hours \* days).

**Source Code:**

def calculate\_net\_salary(emp\_type, monthly\_salary=0, hourly\_rate=0, present\_days=0,

                         bonus=0, income\_tax=0):

    total\_days = 30

    working\_hours\_per\_day = 8

    if emp\_type.lower() == "permanent":

        per\_day\_salary = monthly\_salary / total\_days

        gross\_salary = per\_day\_salary \* present\_days + bonus

    elif emp\_type.lower() == "temporary":

        gross\_salary = hourly\_rate \* working\_hours\_per\_day \* present\_days + bonus

    else:

        print("Invalid employee type.")

        return 0

    net\_salary = gross\_salary - income\_tax

    return net\_salary

emp\_type = input("Enter employee type (Permanent/Temporary): ").strip()

if emp\_type.lower() == "permanent":

    monthly\_salary = float(input("Enter monthly salary: "))

    present\_days = int(input("Enter number of present days (out of 30): "))

    bonus = float(input("Enter incentives/bonus (if any): "))

    income\_tax = float(input("Enter income tax amount: "))

    net = calculate\_net\_salary(emp\_type, monthly\_salary=monthly\_salary,

                                present\_days=present\_days,

                                bonus=bonus, income\_tax=income\_tax)

elif emp\_type.lower() == "temporary":

    hourly\_rate = float(input("Enter hourly rate: "))

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

    bonus = float(input("Enter incentives/bonus (if any): "))

    income\_tax = float(input("Enter income tax amount: "))

    net = calculate\_net\_salary(emp\_type, hourly\_rate=hourly\_rate,

                                present\_days=present\_days,

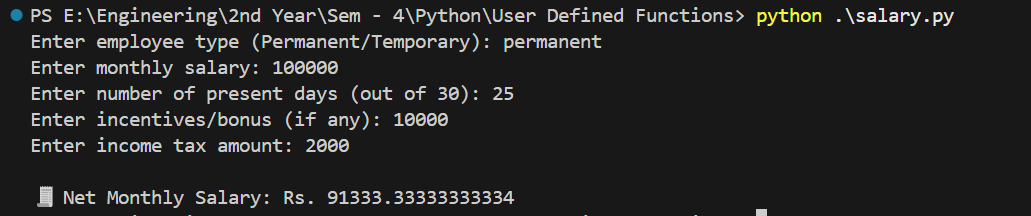
                                bonus=bonus, income\_tax=income\_tax)

else:

    net = 0

print("\n🧾 Net Monthly Salary: Rs.", net)

**Sample Output:**

****

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

**Theory:**

This program uses a recursive function to display the Fibonacci series up to n terms. In the Fibonacci series, each number is the sum of the two preceding ones, starting from:

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

We defined a function fibonacci(n) where:

* If n <= 0: return 0
* If n == 1: return 1
* Else: return fibonacci(n-1) + fibonacci(n-2)

This is a classic example of recursion, where the function calls itself with smaller inputs until it reaches the base condition.

Programming Concepts Used:

* Recursion
* Looping for display
* Conditional statements
* Function definition

**Source Code:**

def fibonacci(n):

    if n <= 0:

        return 0

    elif n == 1:

        return 1

    else:

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

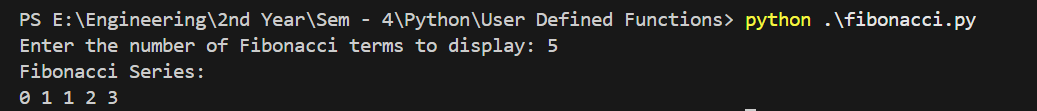
terms = int(input("Enter the number of Fibonacci terms to display: "))

print("Fibonacci Series: ")

for i in range(terms):

    print(fibonacci(i), end=' ')

**Sample Output:**

****

**Title: Demonstrate passing a list to a user-defined function and returning a modified list from the function.**

**Theory:**

In this program:

* We demonstrate how to pass a list as an argument to a function and how to return a modified list from that function.
* The function modify\_list(input\_list) takes a list, doubles each element using a loop, and returns a new list.

Key Programming Concepts Used:

* User Defined Functions (UDF)
* List creation and manipulation
* Looping through list elements
* Returning values from functions

**Source Code:**

def modify\_list(input\_list):

    new\_list = []

    for element in input\_list:

        new\_list.append(element \* 2)

    return new\_list

original\_list = []

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

for i in range(n):

    val = int(input(f'Enter element {i+1}: '))

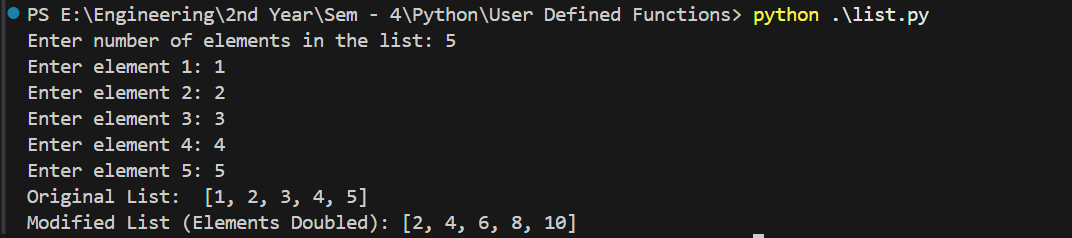
    original\_list.append(val)

modified = modify\_list(original\_list)

print("Original List: ", original\_list)

print("Modified List (Elements Doubled):", modified)

**Sample Output:**

****

**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).**

**Theory:**

This program demonstrates the use of User Defined Modules and Exception Handling in Python:

* We created a custom module mymodule.py containing three reusable functions:
  + factorial(n) – returns the factorial of a number
  + primeNumber(n) – checks if a number is prime
  + powNumber(base, exponent) – calculates base^exponent
* In the main program, we:
  + Imported the module using import mymodule
  + Created a menu-driven interface
  + Used try-except blocks to handleinvalid inputs and prevent crashes

**Source Code:**

def factorial(n):

    if n < 0:

        return 'Factorial not defined for negative numbers'

    result = 1

    for i in range(1, n + 1):

        result \*= i

    return result

def primeNumber(n):

    if n < 2:

        return False

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

        if n % i == 0:

            return False

    return True

def powNumber(base, exponent):

    return base \*\* **exponent**

import functions

def main():

    while True:

        print("\n===== MENU =====")

        print("1. Calculate Factorial")

        print("2. Check Prime Number")

        print("3. Calculate Power")

        print("4. Exit")

        try:

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

            if choice == 1:

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

                print("Factorial: ", **functions.factorial(num))**

elif choice == 2:

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

                if functions.primeNumber(num):

                    print(num, 'Is a prime number.')

                else:

                    print(num, 'is NOT a Prime Number.')

            elif choice == 3:

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

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

                print("Power: ", functions.powNumber(base, exponent))

            elif choice == 4:

                print("Exiting the program. Goodbye!")

                break

            else:

                print("Invalid choice! Please select from 1 to 4.")

        except ValueError:

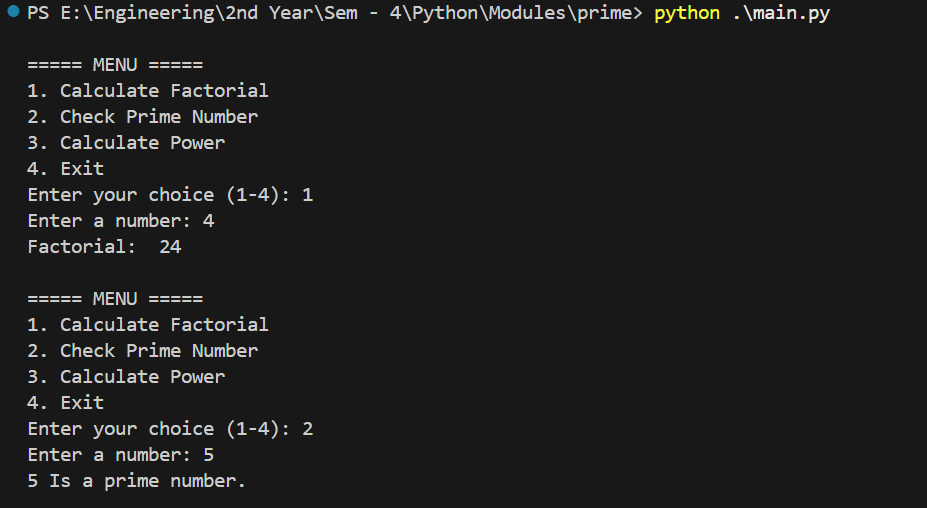
            print("Invalid input! Please enter numeric values only.")

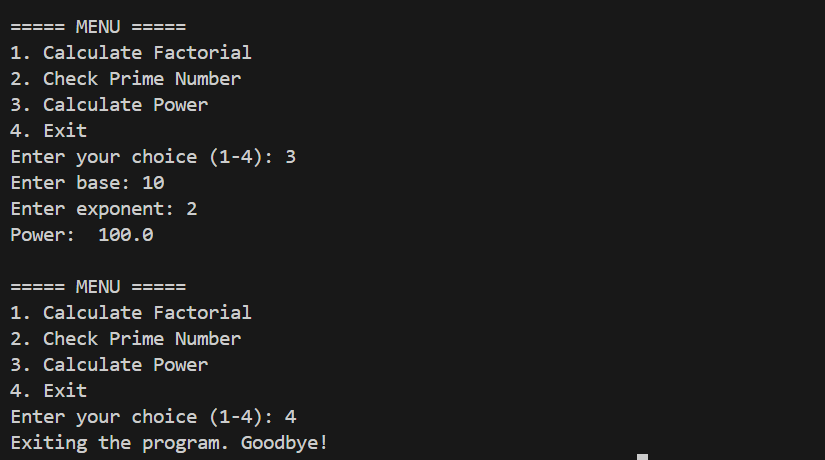
        except Exception as e:

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

main()

**Sample Output:**

****

****

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

**Theory:**

This program demonstrates how to handle multiple specific exceptions in Python using try-except blocks.

* NameError occurs when a variable is used without being defined.
* IndexError occurs when accessing an element from a list using an invalid index.
* ZeroDivisionError occurs when attempting to divide a number by zero.

We used multiple except blocks to catch and handle each error individually, which is good practice for debugging and user-friendly error messages.

Key Concepts:

* Exception Handling (try, except, finally)
* Specific error types: NameError, IndexError, ZeroDivisionError
* Graceful program execution even with errors

**Source Code:**

my\_list = [10,20,30]

try:

    print("Value of x is:", x)

    print("Fourth element of list is: ", my\_list[3])

    result = 100/0

    print("Result:", result)

except NameError as ne:

    print("Caught a NameError:", ne)

except IndexError as ie:

    print("Caught an IndexError:", ie)

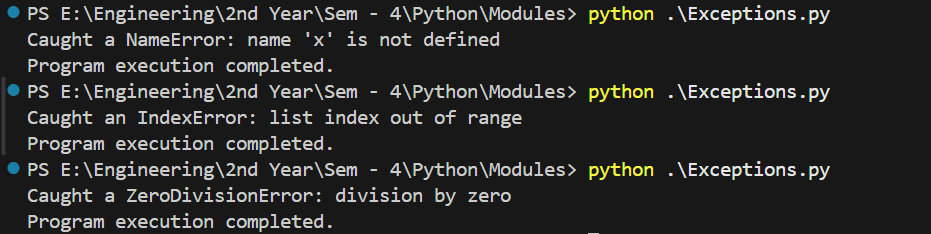
except ZeroDivisionError as ze:

    print("Caught a ZeroDivisionError:",ze)

finally:

    print("Program execution completed.")

**Sample Output:**

****

**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.).**

**Theory:**

This program simulates a login system with proper handling of multiple exceptions:

* ValueError if the username contains non-alphanumeric characters.
* KeyError if the entered username is not found in the database.
* PermissionError for incorrect password attempts.
* A general Exception to catch any other unexpected error.
* It also tracks login attempts and locks the user out after 3 failed tries.

**Source Code:**

user\_db = {

    "sahil": "sahil@1234",

    "aditya": 'aditya@256',

    "rushi": "rushi@1234"

}

MAX\_ATTEMPTS = 3

def login():

    attempts = 0

    while attempts < MAX\_ATTEMPTS:

        try:

            username = input("Enter your username: ")

            if not username.isalnum():

                raise ValueError("Username should be alphanumeric.")

            if username not in user\_db:

                raise KeyError("User does not exist.")

            password = input("Enter your password: ")

            if user\_db[username] != password:

                raise PermissionError("Incorrect password")

            print("Login Successful. Welcome,", username + '!')

            return

        except ValueError as ve:

            print("ValueError: ", ve)

except KeyError as ke:

            print("KeyError:", ke)

        except PermissionError as pe:

            print("PermissionError:", pe)

        except Exception as e:

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

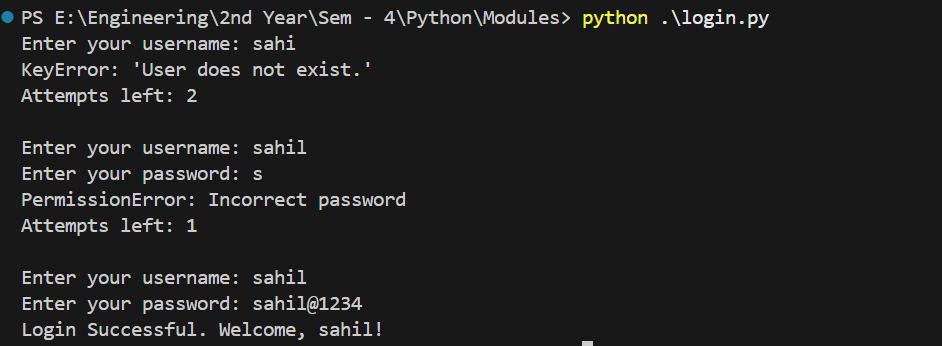
        attempts += 1

        print(f"Attempts left: {MAX\_ATTEMPTS - attempts}\n")

    print("Too many failed attempts. Please try again later.")

login()

**Sample Output:**

****

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

**Theory:**

This program demonstrates Object-Oriented Programming in Python through the concept of constructors:

* A constructor is a special method (\_\_init\_\_) that is automatically called when an object of the class is created.
* A default constructor provides default values if no arguments are passed.
* A parameterized constructor allows us to pass specific values during object creation.

Key Concepts Used:

* Class and Object
* Constructor overloading using default arguments
* Method definition inside a class
* Reusability of code through object instances

**Source Code:**

class Student:

    def \_\_init\_\_(self, name='unknown', roll\_no=0):

        self.name = name

        self.roll\_no = roll\_no

    def display\_info(self):

        print(f"Name: {self.name}")

        print(f"Roll No: {self.roll\_no}")

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

# Default constructor

student1 = Student()

print("Using Default Constructor: ")

student1.display\_info()

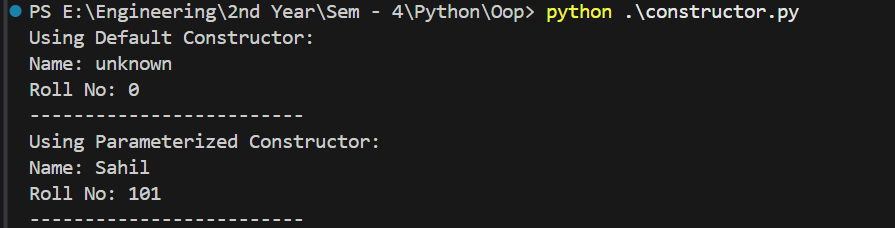
# Parameterized constructor

student2 = Student("Sahil", 101)

print("Using Parameterized Constructor:")

student2.display\_info()

**Sample Output:**

****

**Title: Write an object-oriented, menu-driven program to perform basic banking operations like:**

* **New Account**
* **Deposit**
* **Withdraw**
* **Balance Inquiry**
* **Show All Accounts**
* **Exit**

**Theory:**

This program demonstrates Object-Oriented Programming (OOP) by simulating a basic banking system with multiple operations:

* A class BankAccount is defined to represent each user with attributes like name, account number, and balance.
* Each operation (deposit, withdraw, etc.) is implemented as a method.
* A dictionary is used to store multiple accounts.
* A menu-driven interface allows the user to interact with the system.

Key Concepts:

* OOP (Class, Object, Methods)
* Encapsulation
* Menu-driven programming using while loop and conditional branching

**Source Code:**

class BankAccount:

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

        self.name = name

        self.acc\_number = acc\_number

        self.balance = balance

    def deposit(self, amount):

        self.balance += amount

        print(f'{amount}Rs deposited successfully!')

    def withdraw(self, amount):

        if amount > self.balance:

            print("Insufficient balance!")

        else:

            self.balance -= amount

            print(f'{amount}Rs withdrawn successfully!')

    def display\_balance(self):

        print(f'Current balance; {self.balance}Rs')

    def display\_info(self):

        print(f"Name: {self.name}, Account Number: {self.acc\_number}, Balance: {self.balance}Rs")

bank\_accounts = {}

def create\_account():

    name = input("Enter name: ")

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

    if acc\_number in bank\_accounts:

        print("Account already Exists! ")

    else:

        bank\_accounts[acc\_number] = BankAccount(name, acc\_number)

        print("New account created successfully!")

def deposit\_amount():

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

    if acc\_number in bank\_accounts:

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

        bank\_accounts[acc\_number].deposit(amount)

    else:

        print("Account not found!")

def withdraw\_amount():

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

    if acc\_number in bank\_accounts:

        amount = float(input("Enter amount to withdraw: "))

        bank\_accounts[acc\_number].withdraw(amount)

    else:

**print(" Account not found!")**

def check\_balance():

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

    if acc\_number in bank\_accounts:

        bank\_accounts[acc\_number].display\_balance()

    else:

        print("Account not found!")

def show\_all\_accounts():

    if bank\_accounts:

        for account in bank\_accounts.values():

            account.display\_info()

    else:

        print("No accounts to display.")

while True:

    print("\n====== Bank Menu ======")

    print("1. New Account")

    print("2. Deposit")

    print("3. Withdraw")

    print("4. Check Balance")

    print("5. Show All Accounts")

    print("6. Exit")

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

    if choice == '1':

        create\_account()

    elif choice == '2':

        deposit\_amount()

    elif choice == '3':

        withdraw\_amount()

    elif choice == '4':

        check\_balance()

    elif choice == '5':

        show\_all\_accounts()

    elif choice == '6':

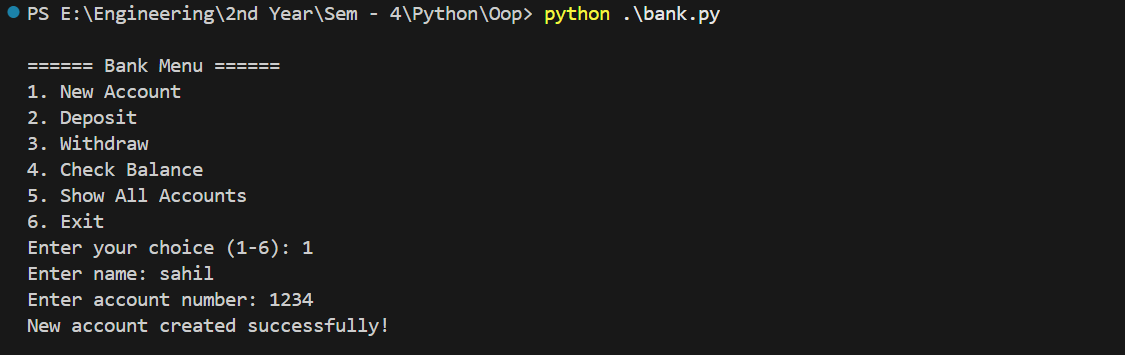
        print("👋 Exiting... Thank you for banking with us!")

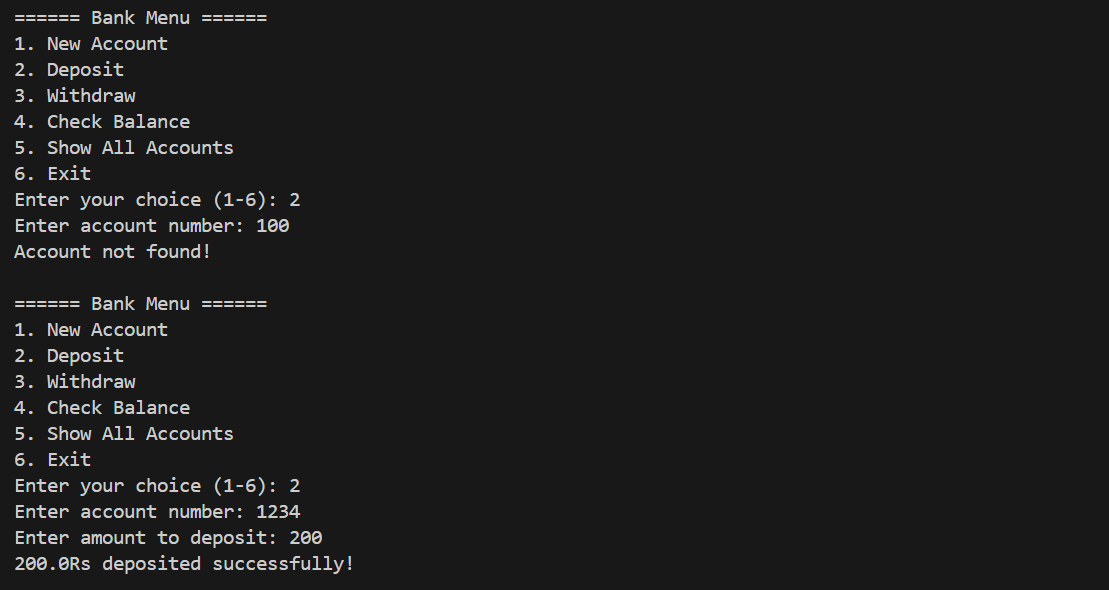
        break

    else:

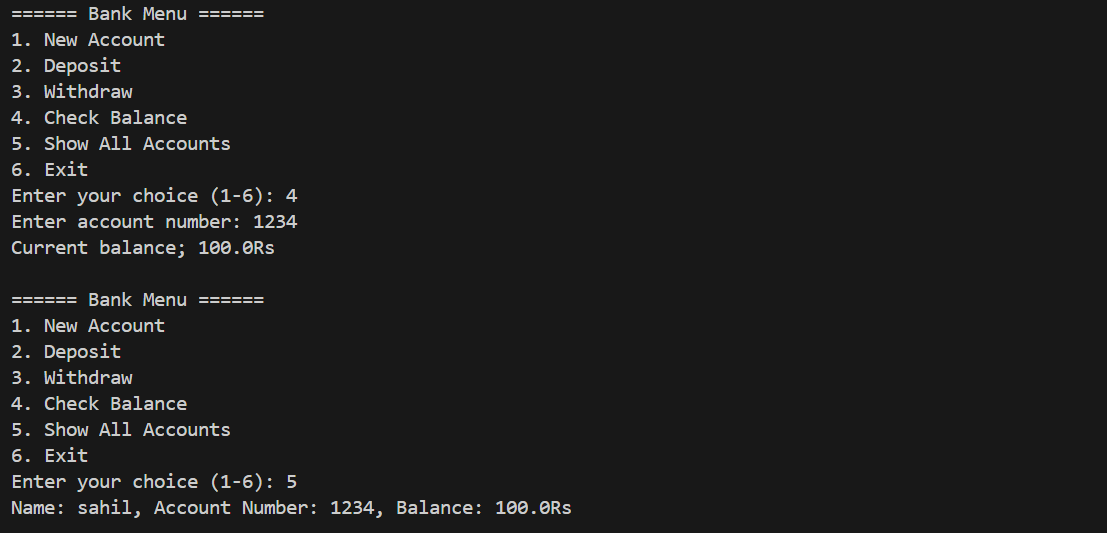
        print(" Invalid choice! Please try again.")

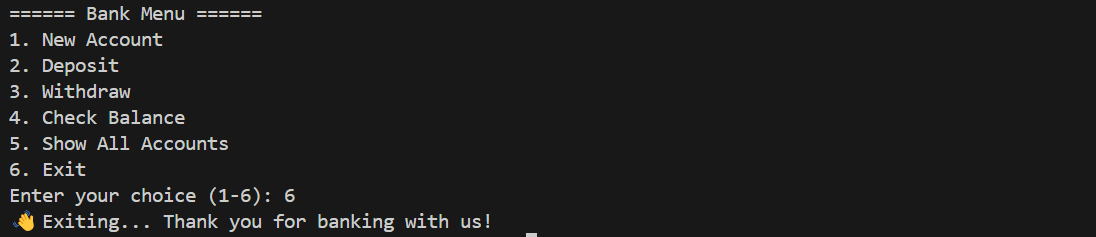
**Sample Output:**











**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\_\_ ).**

**Theory:**

This program demonstrates inheritance in Python using real-world scenarios of a person → student → college student.

* Single-level inheritance: The Student class inherits from Person, reusing and extending its properties.
* Multilevel inheritance: The CollegeStudent class further extends Student, showing deep inheritance.
* super() is used to call parent class constructors.
* \_\_str\_\_() gives a readable string output of object data.
* The condition if \_\_name\_\_ == "\_\_main\_\_": ensures the demo runs only when the file is executed directly.

Key OOP Concepts:

* Inheritance (Single & Multilevel)
* Constructor chaining with super()
* Polymorphism using \_\_str\_\_
* Realistic object structure and readability

**Source Code:**

class Person:

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

        self.name = name

        self.age = age

    def \_\_str\_\_(self):

        return f'Name: {self.name}, Age: {self.age}'

class Student(Person):

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

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

        self.student\_id = student\_id

    def \_\_str\_\_(self):

        return super().\_\_str\_\_() + f', Student ID: {self.student\_id}'

class CollegeStudent(Student):

    def \_\_init\_\_(self, name, age, student\_id, college\_name):

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

        self.college\_name = college\_name

    def \_\_str\_\_(self):

        return super().\_\_str\_\_() + f'College: {self.college\_name}'

if \_\_name\_\_ == '\_\_main\_\_':

    print("\n---Single Level Inheritanxe Demo---")

    student1 = Student("Sahil", 20, "ST101")

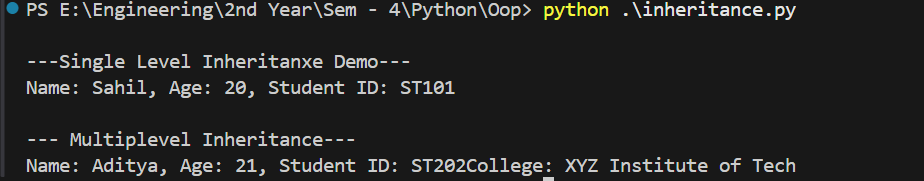
    print(student1)

    print("\n--- Multiplevel Inheritance---")

    college\_student1 = CollegeStudent("Aditya", 21, "ST202", "XYZ Institute of Tech")

    print(college\_student1)

**Sample Output:**



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

**Theory:**

**Source Code:**

import tkinter as tk

from tkinter import messagebox

import sqlite3

# Database setup

conn = sqlite3.connect("users.db")

cursor = conn.cursor()

cursor.execute('''

CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

username TEXT UNIQUE NOT NULL,

password TEXT NOT NULL

)

''')

conn.commit()

# Main Window

root = tk.Tk()

root.title("Login & Register System")

root.geometry("450x400")

root.resizable(False, False)

root.configure(bg="#f0f0f0")

# Fonts and Colors

font\_title = ("Arial", 18, "bold")

font\_label = ("Arial", 12)

btn\_color = "#4CAF50"

entry\_bg = "#ffffff"

highlight = "#222222"

# Frame Setup

frame\_register = tk.Frame(root, bg="white")

frame\_login = tk.Frame(root, bg="white")

for frame in (frame\_register, frame\_login):

frame.place(x=0, y=0, width=450, height=400)

# REGISTER PAGE

def register\_user():

username = reg\_username.get()

password = reg\_password.get()

if username == "" or password == "":

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

return

try:

cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)", (username, password))

conn.commit()

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

reg\_username.set("")

reg\_password.set("")

show\_login()

except sqlite3.IntegrityError:

messagebox**.showerror("Error", "Username already exists!")**

def show\_login():

frame\_register.place\_forget()

frame\_login.place(x=0, y=0, width=450, height=400)

def show\_register():

frame\_login.place\_forget()

frame\_register.place(x=0, y=0, width=450, height=400)

reg\_username = tk.StringVar()

reg\_password = tk.StringVar()

tk.Label(frame\_register, text="Register", font=font\_title, bg="white", fg=highlight).pack(pady=20)

tk.Label(frame\_register, text="Username", font=font\_label, bg="white").pack(pady=5)

tk.Entry(frame\_register, textvariable=reg\_username, bg=entry\_bg, width=30).pack()

tk.Label(frame\_register, text="Password", font=font\_label, bg="white").pack(pady=5)

tk.Entry(frame\_register, textvariable=reg\_password, bg=entry\_bg, show="\*", width=30).pack()

tk.Button(frame\_register, text="Register", command=register\_user, bg=btn\_color, fg="white", width=20).pack(pady=20)

tk.Button(frame\_register, text="Already have an account? Login", command=show\_login, bg="white", fg="blue", bd=0).pack()

# LOGIN PAGE

def login\_user():

username = login\_username.get()

password = login\_password.get()

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

result = cursor.fetchone()

if result:

messagebox.showinfo("Success", f"Welcome, {username}!")

else:

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

login\_username = tk.StringVar()

login\_password = tk.StringVar()

tk.Label(frame\_login, text="Login", font=font\_title, bg="white", fg=highlight).pack(pady=20)

tk.Label(frame\_login, text="Username", font=font\_label, bg="white").pack(pady=5)

tk.Entry(frame\_login, textvariable=login\_username, bg=entry\_bg, width=30).pack()

tk.Label(frame\_login, text="Password", font=font\_label, bg="white").pack(pady=5)

tk.Entry(frame\_login, textvariable=login\_password, bg=entry\_bg, show="\*", width=30).pack()

tk.Button(frame\_login, text="Login", command=login\_user, bg=btn\_color, fg="white", width=20).pack(pady=20)

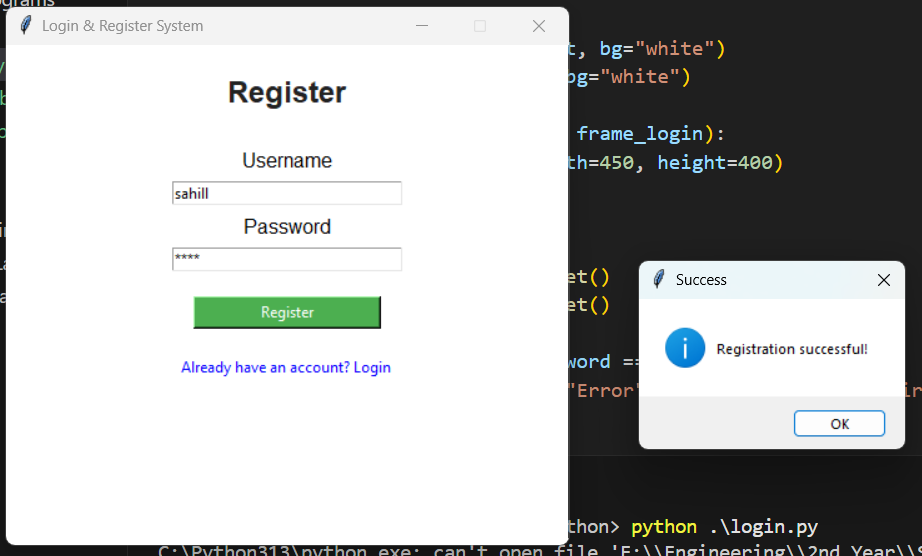
tk.Button(frame\_login, text="New user? Register here", command=show\_register, bg="white", fg="blue", bd=0).pack()

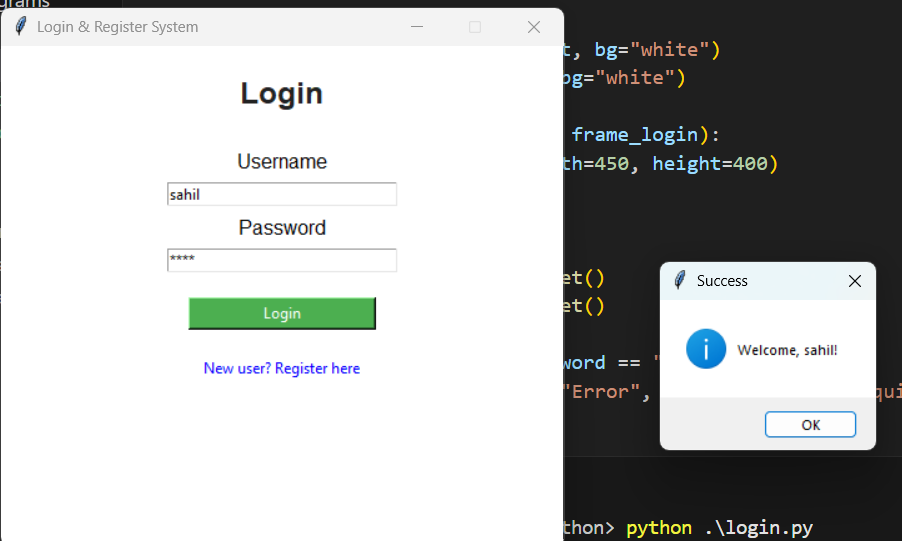
# Start on Register page

show\_register()

root.mainloop()

**Sample Output:**





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

**Theory:**

SQLite Database:

* A database named user\_management.db is created (or connected if already exists).
* The table users contains fields for id, username, email, and age.

CRUD Functions:

* Add User: Takes input from the user and inserts it into the database.
* Show All Users: Fetches and displays all users in the database.
* Delete User: Deletes a user based on the username entered.
* Update User: Allows the user to update the email and age of a specific user.
* Search User: Searches for a user by their username and displays their details if found.

Menu-Driven Program: The program continuously asks the user for input and performs the respective CRUD operation until the user selects "Exit".

**Source Code:**

import sqlite3

conn = sqlite3.connect('users\_management.db')

cursor = conn.cursor()

cursor.execute('''

    CREATE TABLE IF NOT EXISTS users (

        id INTEGER PRIMARY KEY AUTOINCREMENT,

        username TEXT UNIQUE NOT NULL,

        email TEXT NOT NULL,

        age INTEGER NOT NULL

    )

''')

conn.commit()

def add\_user():

    username = input("Enter username: ")

    email = input("Enter email: ")

    age = int(input("Enter age: "))

    try:

        cursor.execute("INSERT INTO users (username, email, age) VALUES (?,?,?)", (username, email, age))

        conn.commit()

        print("user added successfully")

    except sqlite3.IntegrityError:

        print("Error: Username already exists!")

def show\_users():

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

    users = cursor.fetchall()

    if users:

        print("\n User List: ")

        for user in users:

            print(f"ID: {user[0]}, Username: {user[1]}, Email: {user[2]}, Age: {user[3]}")

    else:

        print("No users found!")

def delete\_user():

    username = input("Enter username to delete: ")

    cursor.execute("DELETE FROM users WHERE username = ? ", (username,))

    conn.commit()

    if cursor.rowcount > 0:

        print(f"User '{username}' deleted successfully!")

    else:

        print("User not found!")

def update\_user():

    username = input("Enter username to update: ")

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

    user = cursor.fetchone()

    if user:

        print(f'Current details: Username: {user[1]}, Email: {user[2]}, Age: {user[3]}')

        email = input("Enter new Email: ")

        age = int(input("Enter new age: "))

        cursor.execute("UPDATE users SET email = ?, age = ? WHERE username = ?", (email, age, username))

        conn.commit()

        print("User details updated successfully!")

    else:

        print("User not found!")

def search\_user():

    username = input("Enter username to search: ")

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

    user = cursor.fetchone()

    if user:

        print(f'User found: ID: {user[0]}, Username: {user[1]}, Email: {user[2]}, Age: {user[3]}')

    else:

        print("User not found!")

def menu():

    while True:

        print("\nUser Management System")

        print("1. Add User")

        print("2. Show All Users")

        print("3. Delete User")

        print("4. Update User")

        print("5. Search User")

        print("6. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":

            add\_user()

        elif choice == "2":

            show\_users()

        elif choice == "3":

            delete\_user()

        elif choice == "4":

            update\_user()

        elif choice == "5":

            search\_user()

        elif choice == "6":

            print("Exiting...")

            break

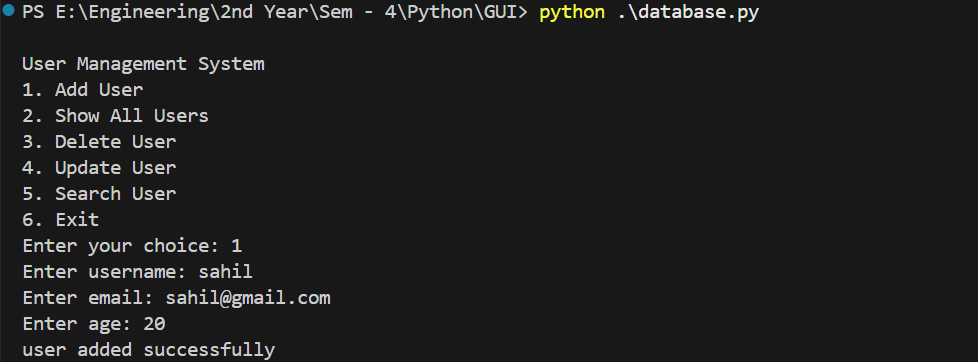
        else:

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

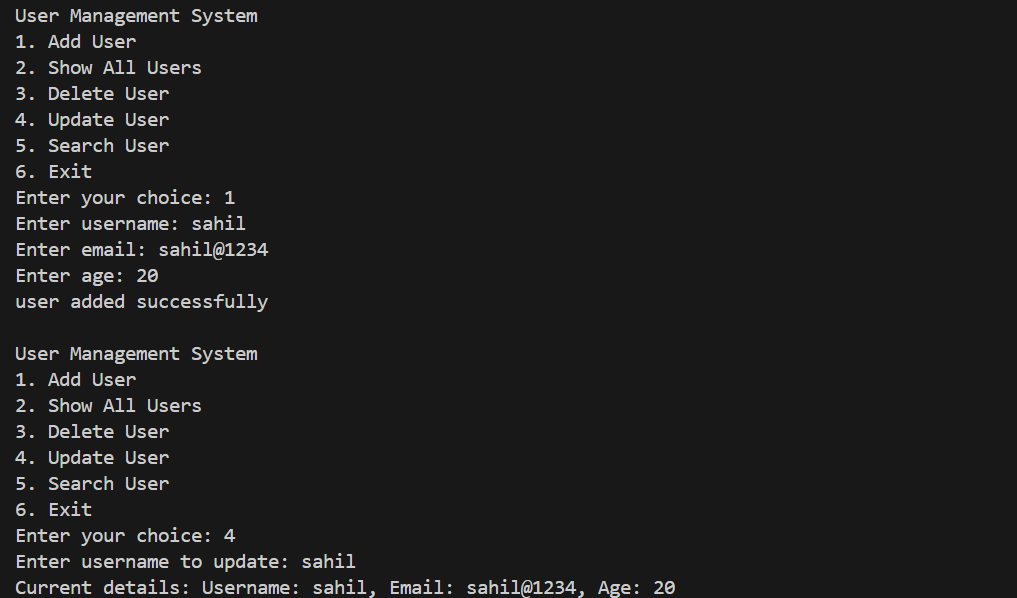
menu()

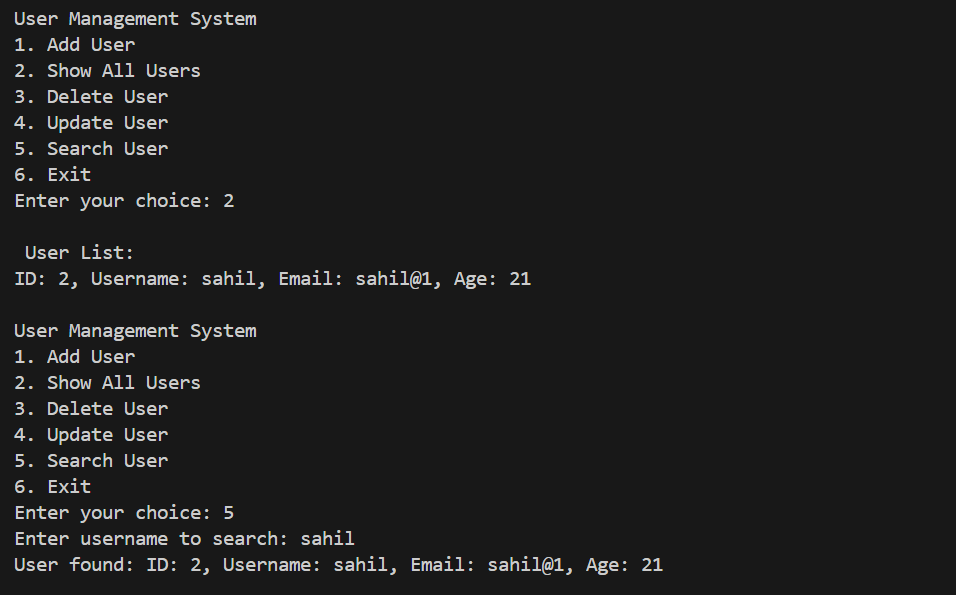
conn.close()

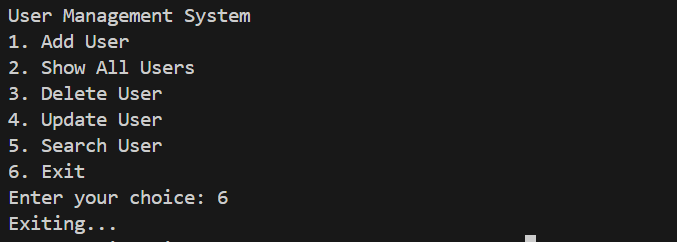
**Sample Output:**











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

**Theory:**

In this program, we use file handling in Python, which involves creating, writing, and reading a file.  
To read a file, we used three different approaches:

* read() – Reads the whole content of the file as a single string.
* readline() – Reads one line at a time.
* readlines() – Reads the entire file and returns a list of lines.

The with open(...) syntax is used to handle files safely, ensuring the file is properly closed after the operation.

**Source Code:**

filename = input("Enter file name to create: ")

content = input("Enter content to write in the file")

with open(filename, 'w') as file:

    file.write(content)

print("\n Content written successfully!")

print("\n--- Reading using read()---")

with open(filename, 'r') as file:

    data = file.read()

    print(data)

print("\n--- Reading using readline() ---")

with open(filename, 'r') as file:

    file.seek(0)

    line = file.readline()

    while line:

        print(line.strip())

        line = file.readline()

print("\n--- Reading using readlines() ---")

with open(filename, "r") as file:

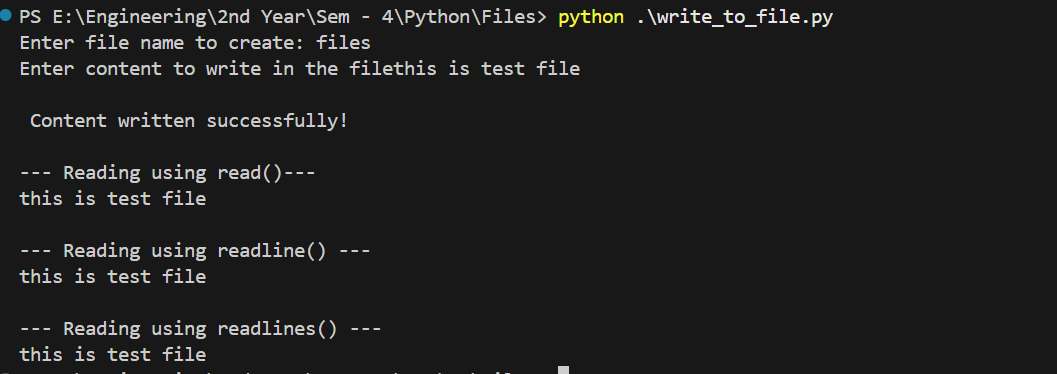
    file.seek(0)

    lines = file.readlines()

    for line in lines:

        print(line.strip())

**Sample Output:**



**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.**

**Theory:**

This program demonstrates file handling and string manipulation in Python.  
We read a file line by line using a for loop with enumerate() to get both content and line numbers.  
We used str.lower() to ensure the word "error" is matched regardless of case.  
Finally, the results (error lines and counts) are written to a new file (error.txt) using file output functions.

This type of log scanning is often used in real-world debugging and system monitoring tools.

**Source Code:**

error\_count = 0

error\_lines = []

try:

    with open('server.txt', 'r') as logfile:

        for line\_number, line in enumerate(logfile, start=1):

            if "error" in line.lower():

                error\_count += 1

                error\_lines.append(f'Line {line\_number}: {line.strip()}')

    with open("error.txt", 'w') as errorfile:

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

        errorfile.write("Details: \n")

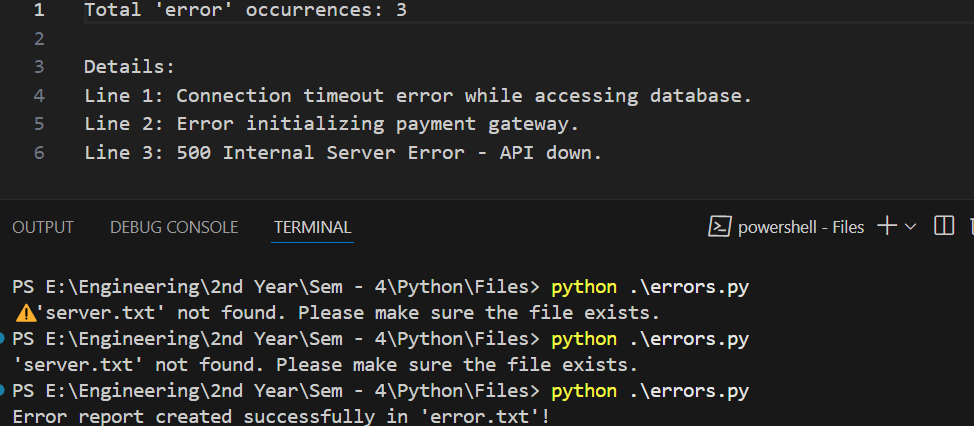
        errorfile.write("\n".join(error\_lines))

    print("Error report created successfully in 'error.txt'!")

except FileNotFoundError:

    print("'server.txt' not found. Please make sure the file exists.")

**Sample Output:**



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

**Theory:**

This program demonstrates the use of multithreading using Python’s threading module.

* Each user represents a separate thread, simulating parallel access to the ATM.
* To avoid data corruption (like two users accessing the balance at once), we use a Lock to create a critical section, allowing only one user to withdraw at a time.
* start() begins a thread and join() ensures the main thread waits until all user threads are finished.

This is a great example of thread synchronization, used in banking systems, ticket booking platforms, and more.

**Source Code:**

import threading

import time

account\_balance = 1000

lock = threading.Lock()

def withdraw\_money(name, amount):

    global account\_balance

    print(f'{name} is trying to withdraw {amount}Rs...')

    with lock:

        if account\_balance >= amount:

            print(f'{name} - Withdrawal Approved!')

            time.sleep(1)

            account\_balance -= amount

            print(f'{name} - {amount}Rs withdrwan. Remaining balance: {account\_balance}Rs')

        else:

            print(f'{name} - Withdrawal denied Insufficient balance')

user1 = threading.Thread(target=withdraw\_money, args=("sahil", 200))

user2 = threading.Thread(target=withdraw\_money, args=("Aditya", 700))

user3 = threading.Thread(target=withdraw\_money, args=("Max", 2000))

user1.start()

user2.start()

user3.start()

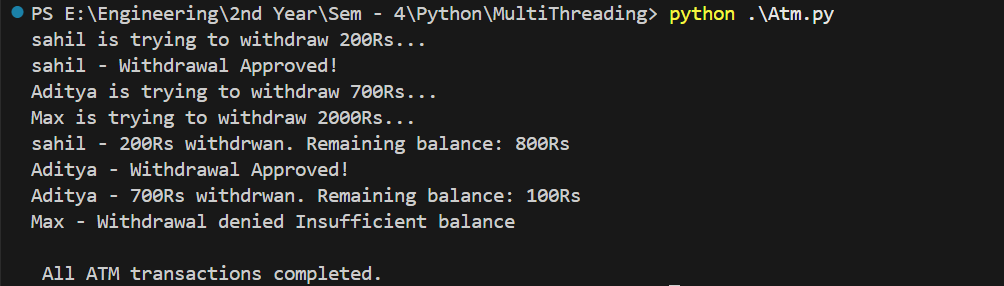
user1.join()

user2.join()

user3.join()

print("\n All ATM transactions completed.")

**Sample Output:**



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

**Theory:**

This program demonstrates race conditions in multithreaded environments and how to prevent them using Locks:

* When multiple threads access shared data simultaneously without synchronization, data inconsistency can occur — this is called a race condition.
* We fix this by using thread locks (threading.Lock()), ensuring that only one thread can access the critical section at a time.
* Real-world applications like banking, airline booking, and e-commerce carts heavily depend on these mechanisms.

**Source Code:**

**Without Lock:**

import threading

import time

balance = 1000

def withdraw(name, amount):

    global balance

    print(f'{name} is trying to withdraw {amount}Rs')

    if balance >= amount:

        print(f'{name} - Approved!')

        time.sleep(2)

        balance -= amount

        print(f'{name} - withdrawn {amount}Rs, Remaining: {balance}Rs')

    else:

        print(f'{name} - Denied Insufficient Balance')

t1 = threading.Thread(target=withdraw, args=("sahil", 300))

t2 = threading.Thread(target=withdraw, args=("aditya", 800))

t1.start()

t2.start()

t1.join()

t2.join()

print(f'\n Final Account Balance (without lock) {balance}Rs')

**With Lock**

import threading

import time

balance = 1000

lock = threading.Lock()

def withdraw\_locked(name, amount):

    global balance

    print(f"{name} is trying to withdraw ₹{amount}")

    with lock:

        if balance >= amount:

            print(f"{name} - Approved")

            time.sleep(1)

            balance -= amount

            print(f"{name} - Withdrawn ₹{amount}, Remaining: ₹{balance}")

        else:

            print(f"{name} - Denied Insufficient balance")

# Threads

t1 = threading.Thread(target=withdraw\_locked, args=("Alice", 700))

t2 = threading.Thread(target=withdraw\_locked, args=("Bob", 500))

# Start threads

t1.start()

t2.start()

# Wait

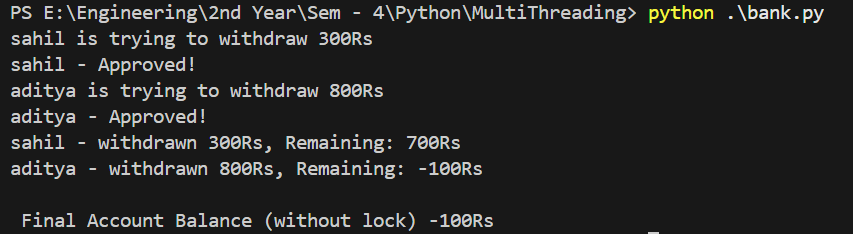
t1.join()

t2.join()

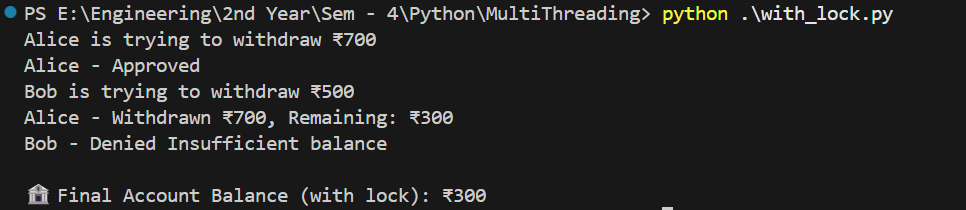
print(f"\nFinal Account Balance (with lock): ₹{balance}")

**Sample Output:**

**Without lock**



With Lock



**Title:**

**Theory:**

**Source Code:**

**Sample Output:**

**Title:**

**Theory:**

**Source Code:**

**Sample Output:**

**Title:**

**Theory:**

**Source Code:**

**Sample Output:**