**Your Name: Munazza Ramzan**

**Course : HCCD-AI**

**Python Lab**

**Lab 1: Conditions**

**Code:**

# get input from user

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

# check if the num  is postive, negtive or zero

if num>0:

    print(" The num is postive")

if num<0:

    print("The number in negitive ")

else:

    print(" The num is zero")

# Get input from the user

num = int(input("Enter an integer: "))

# Check if the number is even or odd

if num % 2 == 0:

    print("The number is even.")

else:

    print("The number is odd.")

# Get input from the user

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

# Check voting eligibility

if age >= 18:

    print("You are eligible to vote.")

else:

    print("You are not eligible to vote.")

# Get input from the user

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

# Check if the number is a multiple of both 3 and 5

if num % 3 == 0 and num % 5 == 0:

    print("The number is a multiple of both 3 and 5.")

else:

    print("The number is not a multiple of both 3 and 5.")

# Get input from the user

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

# Get the last digit

last\_digit = abs(num) % 10  # abs() handles negative numbers

# Check if the last digit is divisible by 3

if last\_digit % 3 == 0:

    print("The last digit is divisible by 3.")

else:

    print("The last digit is not divisible by 3.")

# Get input from the user

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

# Check if it is a leap year

if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):

    print(f"{year} is a leap year.")

else:

    print(f"{year} is not a leap year.")

# Dictionary to map numbers to days

days\_of\_week = {

    1: "Sunday",

    2: "Monday",

    3: "Tuesday",

    4: "Wednesday",

    5: "Thursday",

    6: "Friday",

    7: "Saturday"

}

# Get user input

try:

    num = int(input("Enter a number (1 to 7): "))

    # Check if number is in valid range

    if 1 <= num <= 7:

        print("The day of the week is:", days\_of\_week[num])

    else:

        print("Invalid input. Please enter a number between 1 and 7.")

except ValueError:

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

# Get user input

try:

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

    # Check if the number is a multiple of 5

    if number % 5 == 0:

        print("Hello")

    else:

        print("Bye")

except ValueError:

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

# Get user input

try:

    num1 = float(input("Enter the first number: "))

    num2 = float(input("Enter the second number: "))

    # Compare the numbers

    if num1 > num2:

        print(f"{num1} is greater than {num2}.")

    elif num2 > num1:

        print(f"{num2} is greater than {num1}.")

    else:

        print("Both numbers are equal.")

except ValueError:

    print("Invalid input. Please enter valid numbers.")

# Get user input

char = input("Enter a single alphabet character: ")

# Check if input is a single alphabet character

if len(char) == 1 and char.isalpha():

    # Convert to lowercase for simplicity

    char = char.lower()

    # Check if the character is a vowel

    if char in ['a', 'e', 'i', 'o', 'u']:

        print(f"'{char}' is a vowel.")

    else:

        print(f"'{char}' is a consonant.")

else:

    print("Invalid input. Please enter a single alphabet character.")

# Get user input

try:

    num2 = float(input("Enter the second number: "))

    num3 = float(input("Enter the third number: "))

    # Use if-elif-else to find the largest

    if num1 >= num2 and num1 >= num3:

        print(f"The largest number is: {num1}")

    elif num2 >= num1 and num2 >= num3:

        print(f"The largest number is: {num2}")

    else:

        print(f"The largest number is: {num3}")

except ValueError:

    print("Invalid input. Please enter valid numbers.")

# Get user input

try:

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

    if num <= 1:

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

    else:

        is\_prime = True

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

            if num % i == 0:

                is\_prime = False

                break

        if is\_prime:

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

        else:

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

except ValueError:

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

# Get user input

try:

    marks = float(input("Enter your marks (0 to 100): "))

    # Check for valid marks

    if 0 <= marks <= 100:

        if marks >= 90:

            print("Grade: A")

        elif marks >= 80:

            print("Grade: B")

        elif marks >= 70:

            print("Grade: C")

        elif marks >= 60:

            print("Grade: D")

        else:

            print("Grade: F")

    else:

        print("Invalid input. Marks should be between 0 and 100.")

except ValueError:

    print("Invalid input. Please enter a valid number.")

# Item price

item\_price = 100

# Get user input

try:

    amount = float(input("Enter your available balance (in Rs.): "))

    # Check if the balance is enough

    if amount >= item\_price:

        print("Purchase successful.")

    else:

        print("Insufficient balance.")

except ValueError:

    print("Invalid input. Please enter a valid amount.")

# Get user input

try:

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

    abs\_num = abs(num)  # Consider absolute value to ignore negative sign

    if abs\_num < 10:

        print("It is a single-digit number.")

    elif abs\_num < 100:

        print("It is a two-digit number.")

    elif abs\_num < 1000:

        print("It is a three-digit number.")

    else:

        print("The number has more than three digits.")

except ValueError:

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

**Lab 2: Loops**

**Code:**

# Program to print all even numbers from 1 to 20

for number in range(1, 21):

    if number % 2 == 0:

        print(number)

# Program to print numbers from 10 to 1 in reverse order using a while loop

number = 10

while number >= 1:

    print(number)

    number -= 1

# Program to print the square of numbers from 1 to 5

for number in range(1, 6):

    square = number \*\* 2

    print(f"The square of {number} is {square}")

# Program to calculate factorial using a while loop

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

if num < 0:

    print("Factorial is not defined for negative numbers.")

else:

    factorial = 1

    i = 1

    while i <= num:

        factorial \*= i

        i += 1

    print(f"The factorial of {num} is {factorial}")

# Program to print characters of "Programming" skipping 'g'

for char in "Programming":

    if char == 'g':

        continue

    print(char)

# Program to print numbers from 1 to 10, stopping if number > 6

for number in range(1, 11):

    if number > 6:

        break

    print(number)

# Program to check if a number is positive, negative, or zero

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

if num > 0:

    print("The number is positive.")

elif num < 0:

    print("The number is negative.")

else:

    print("The number is zero.")

# Program to print multiplication table of a given number

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

print(f"Multiplication table of {num}:")

for i in range(1, 11):

    print(f"{num} x {i} = {num \* i}")

# Program to display the day of the week based on user input

day\_num = int(input("Enter a number (1 to 7): "))

days = {

    1: "Sunday",

    2: "Monday",

    3: "Tuesday",

    4: "Wednesday",

    5: "Thursday",

    6: "Friday",

    7: "Saturday"

}

if 1 <= day\_num <= 7:

    print(f"The day is {days[day\_num]}")

else:

    print("Invalid input! Please enter a number between 1 and 7.")

# Program to keep asking for password until the correct one is entered

password = ""

while password != "python123":

    password = input("Enter the password: ")

    if password != "python123":

        print("Incorrect password, try again.")

print("Access granted!")

**Lab 3: Strings/ Lists**

**Code:**

def main():

    # Get user input and convert it to a list of numbers

    user\_input = input("Enter a list of numbers separated by spaces: ")

    try:

        numbers = list(map(float, user\_input.split()))

    except ValueError:

        print("Please enter only numbers separated by spaces.")

        return

    if not numbers:

        print("The list is empty. Please enter at least one number.")

        return

    # Find and print the maximum and minimum values

    maximum = max(numbers)

    minimum = min(numbers)

    print(f"Maximum value: {maximum}")

    print(f"Minimum value: {minimum}")

    # Calculate and print the sum

    total = sum(numbers)

    print(f"Sum of elements: {total}")

    # Compute and display the average

    average = total / len(numbers)

    print(f"Average of elements: {average}")

# Run the program

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

    main()

def main():

    numbers = []

    print("Enter 10 integers:")

    for i in range(10):

        while True:

            try:

                num = int(input(f"Enter integer #{i + 1}: "))

                numbers.append(num)

                break

            except ValueError:

                print("Invalid input. Please enter an integer.")

    # Display even numbers

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

    print("\nEven numbers:", even\_numbers)

    # Display odd numbers

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

    print("Odd numbers:", odd\_numbers)

    # Count numbers divisible by 5

    divisible\_by\_5 = [num for num in numbers if num % 5 == 0]

    print(f"Numbers divisible by 5: {divisible\_by\_5}")

    print(f"Count of numbers divisible by 5: {len(divisible\_by\_5)}")

# Run the program

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

    main()

def reverse\_list(original\_list):

    reversed\_list = []

    for item in original\_list:

        reversed\_list.insert(0, item)  # Insert each item at the beginning

    return reversed\_list

# Example usage

sample\_list = [1, 2, 3, 4, 5]

reversed\_result = reverse\_list(sample\_list)

print("Original list:", sample\_list)

print("Reversed list:", reversed\_result)

def process\_names(names):

    # Sort the names alphabetically

    sorted\_names = sorted(names, key=str.lower)  # Case-insensitive sorting

    print("Names in alphabetical order:")

    for name in sorted\_names:

        print(name)

    # Find the longest name

    longest\_name = max(names, key=len)

    print(f"\nLongest name: {longest\_name}")

    # Count names starting with a vowel (case insensitive)

    vowels = {'a', 'e', 'i', 'o', 'u'}

    vowel\_count = sum(1 for name in names if name and name[0].lower() in vowels)

    print(f"Number of names starting with a vowel: {vowel\_count}")

# Example usage

name\_list = [

    "Alice", "Eve", "Oscar", "Uma", "Igor",

    "Bob", "Charlie", "David", "Zara", "Andrew"

]

process\_names(name\_list)

def merge\_lists(list1, list2):

    merged = []

    for item in list1:

        merged.append(item)

    for item in list2:

        merged.append(item)

    return merged

# Example usage

list\_a = [1, 2, 3]

list\_b = [4, 5, 6]

result = merge\_lists(list\_a, list\_b)

print("Merged list:", result)

def process\_string():

    user\_input = input("Enter a string: ")

    # Convert to uppercase

    upper\_str = user\_input.upper()

    print("\nUppercase:", upper\_str)

    # Convert to lowercase

    lower\_str = user\_input.lower()

    print("Lowercase:", lower\_str)

    # Count the number of vowels (case-insensitive)

    vowels = "aeiou"

    vowel\_count = sum(1 for char in user\_input.lower() if char in vowels)

    print(f"Number of vowels: {vowel\_count}")

# Run the function

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

    process\_string()

def is\_palindrome(s):

    # Remove spaces and convert to lowercase for uniform comparison

    cleaned = ''.join(char.lower() for char in s if char.isalnum())

    # Compare cleaned string to its reverse

    return cleaned == ''.join(reversed(cleaned))

# Example usage

test\_string = input("Enter a string to check if it's a palindrome: ")

if is\_palindrome(test\_string):

    print("The string is a palindrome.")

else:

    print("The string is not a palindrome.")

def count\_char\_frequency(text):

    # Convert to lowercase and remove spaces

    cleaned\_text = text.lower().replace(" ", "")

    # Dictionary to store frequency

    frequency = {}

    # Count each character

    for char in cleaned\_text:

        if char in frequency:

            frequency[char] += 1a

        else:

            frequency[char] = 1

    # Display the frequencies

    print("Character frequencies (ignoring case and spaces):")

    for char, count in frequency.items():

        print(f"{char}: {count}")

# Example usage

user\_input = input("Enter a string: ")

count\_char\_frequency(user\_input)

def remove\_special\_and\_digits(s):

    result = ''.join(char for char in s if char.isalpha())

    return result

# Example usage

input\_str = "Hello, World! 1234 #Python3"

clean\_str = remove\_special\_and\_digits(input\_str)

print("Original string:", input\_str)

print("Cleaned string:", clean\_str)

def process\_sentence():

    # Take input from the user

    sentence = input("Enter a sentence: ")

    # Split the sentence into words

    words = sentence.split()

    print("\nWords in the sentence:", words)

    # Count the number of words

    word\_count = len(words)

    print("Number of words:", word\_count)

     # Reverse the order of words and create a new sentence

    reversed\_words = []

    for word in words:

        reversed\_words.insert(0, word)  # Insert at the beginning

    reversed\_sentence = ' '.join(reversed\_words)

    print("Reversed sentence:", reversed\_sentence)

# Run the function

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

    process\_sentence()

**lab 4 : Functions/ Lists/ String/Dictionary:**

**code:**

def print\_receipt(customer\_name, item\_purchased):

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

    print("          PURCHASE RECEIPT         ")

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

    print(f"Customer Name : {customer\_name}")

    print(f"Item Purchased: {item\_purchased}")

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

    print("        THANK YOU FOR SHOPPING!     ")

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

def calculate\_zakat(total\_savings):

    zakat\_percentage = 2.5 / 100  # 2.5%

    zakat\_amount = total\_savings \* zakat\_percentage

    return zakat\_amount

def get\_basic\_salary():

    # Get basic salary from the user

    salary = float(input("Enter the basic salary of the employee: "))

    return salary

def get\_employee\_bonus():

    # Call get\_basic\_salary() to retrieve the salary

    basic\_salary = get\_basic\_salary()

    # Calculate 10% bonus

    bonus = basic\_salary \* 0.10

    return bonus

# Driver code

bonus\_amount = get\_employee\_bonus()

print(f"The employee's bonus is: {bonus\_amount}")

def get\_income():

    # Get income from the user

    income = float(input("Enter your annual income: "))

    return income

def calculate\_annual\_tax():

    # Get income from get\_income()

    income = get\_income()

    # Calculate 15% tax

    tax = income \* 0.15

    return tax

# Driver code

annual\_tax = calculate\_annual\_tax()

print(f"The annual tax to be paid is: {annual\_tax}")

def get\_total\_marks():

    # Get marks for 3 subjects from the user

    marks1 = float(input("Enter marks for Subject 1: "))

    marks2 = float(input("Enter marks for Subject 2: "))

    marks3 = float(input("Enter marks for Subject 3: "))

    # Calculate total

    total = marks1 + marks2 + marks3

    return total

def assign\_grade(total\_marks):

    # Assign grade based on total marks

    if total\_marks >= 270:

        return 'A'

    elif total\_marks >= 240:

        return 'B'

    elif total\_marks >= 210:

        return 'C'

    elif total\_marks >= 180:

        return 'D'

    else:

        return 'F'

# === Driver Code ===

total = get\_total\_marks()

grade = assign\_grade(total)

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

print(f"Assigned Grade: {grade}")

def calculate\_total\_bill():

    # Get prices of three items from the user

    item1 = float(input("Enter price of Item 1: "))

    item2 = float(input("Enter price of Item 2: "))

    item3 = float(input("Enter price of Item 3: "))

    total = item1 + item2 + item3

    return total

def apply\_discount(bill\_amount):

    # Apply discount based on bill amount

    if bill\_amount >= 500:

        discount = bill\_amount \* 0.20

    elif bill\_amount >= 300:

        discount = bill\_amount \* 0.10

    else:

        discount = 0.0

    final\_amount = bill\_amount - discount

    print(f"\nOriginal Bill Amount: ₹{bill\_amount:.2f}")

    print(f"Discount Applied: ₹{discount:.2f}")

    print(f"Final Amount to Pay: ₹{final\_amount:.2f}")

# === Driver Code ===

total\_bill = calculate\_total\_bill()

apply\_discount(total\_bill)

def get\_bonus(years):

    # Calculate Rs.1000 bonus per completed year

    bonus = years \* 1000

    return bonus

# === Driver Code ===

years\_worked = int(input("Enter number of years you have worked: "))

bonus\_amount = get\_bonus(years\_worked)

print(f"Your bonus amount is: ₹{bonus\_amount}")

def calculate\_tax(income):

    # Calculate tax based on income slabs

    if income <= 200000:

        tax = 0

    elif income <= 500000:

        tax = (income - 200000) \* 0.05

    else:

        tax = (300000 \* 0.05) + (income - 500000) \* 0.10

    return tax

# === Driver Code ===

income = float(input("Enter your annual income: ₹"))

tax\_amount = calculate\_tax(income)

print(f"Your calculated tax is: ₹{tax\_amount:.2f}")

def calculate\_zakat(weight):

    price\_per\_gram = 8000  # Rs. 8000 per gram

    total\_value = weight \* price\_per\_gram

    zakat = total\_value \* 0.025  # 2.5% of the total value

    return zakat

# === Driver Code ===

gold\_weight = float(input("Enter the weight of gold in grams: "))

zakat\_amount = calculate\_zakat(gold\_weight)

print(f"Zakat to be paid: ₹{zakat\_amount:.2f}")

def calculate\_profit\_loss(cost\_price, selling\_price):

    if selling\_price > cost\_price:

        # Profit

        profit = selling\_price - cost\_price

        return (profit / cost\_price) \* 100

    elif selling\_price < cost\_price:

        # Loss

        loss = cost\_price - selling\_price

        return -((loss / cost\_price) \* 100)

    else:

        # No profit, no loss

        return 0

# === Driver Code ===

cp = float(input("Enter the cost price: ₹"))

sp = float(input("Enter the selling price: ₹"))

result = calculate\_profit\_loss(cp, sp)

# Display result with appropriate message

if result > 0:

    print(f"You made a profit of {result:.2f}%.")

elif result < 0:

    print(f"You incurred a loss of {-result:.2f}%.")

else:

    print("No profit, no loss.")

def calculate\_discounted\_price(price, discount\_percent):

    discount\_amount = (discount\_percent / 100) \* price

    final\_price = price - discount\_amount

    return final\_price

# Driver code

original\_price = float(input("Enter the original price: ₹"))

discount\_percent = float(input("Enter the discount percentage: "))

discounted\_price = calculate\_discounted\_price(original\_price, discount\_percent)

print(f"Discounted price: ₹{discounted\_price:.2f}")

def calculate\_electricity\_bill(units):

    if units <= 100:

        bill = units \* 5

    elif units <= 200:

        bill = (100 \* 5) + (units - 100) \* 7

    else:

        bill = (100 \* 5) + (100 \* 7) + (units - 200) \* 10

    return bill

# === Driver Code ===

units\_consumed = int(input("Enter the number of units consumed: "))

total\_bill = calculate\_electricity\_bill(units\_consumed)

print(f"Total electricity bill: ₹{total\_bill:.2f}")

def calculate\_basic\_bill(units):

    # Calculate bill based on slab rates

    if units <= 100:

        bill = units \* 5

    elif units <= 200:

        bill = (100 \* 5) + (units - 100) \* 7

    else:

        bill = (100 \* 5) + (100 \* 7) + (units - 200) \* 10

    return bill

def calculate\_total\_bill(basic\_bill):

    # Add 10% surcharge if basic bill exceeds Rs.1500

    if basic\_bill > 1500:

        surcharge = basic\_bill \* 0.10

        total\_bill = basic\_bill + surcharge

    else:

        total\_bill = basic\_bill

    return total\_bill

# Driver code

units\_consumed = int(input("Enter the number of units consumed: "))

basic\_bill = calculate\_basic\_bill(units\_consumed)

total\_payable = calculate\_total\_bill(basic\_bill)

print(f"Basic Bill: ₹{basic\_bill:.2f}")

if total\_payable > basic\_bill:

    print(f"Surcharge Applied: ₹{total\_payable - basic\_bill:.2f}")

print(f"Total Payable Amount: ₹{total\_payable:.2f}")

def second\_largest(numbers):

    # Remove duplicates by converting list to a set

    unique\_numbers = set(numbers)

    # Convert back to list and sort in descending order

    sorted\_numbers = sorted(unique\_numbers, reverse=True)

    # Check if there are at least two unique numbers

    if len(sorted\_numbers) < 2:

        print("There is no second largest number.")

    else:

        print("The second largest number is:", sorted\_numbers[1])

# Example usage:

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

second\_largest(numbers)

def merge\_sorted\_lists(list1, list2):

    merged\_list = []

    i = j = 0

    # Traverse both lists and append smaller element to merged\_list

    while i < len(list1) and j < len(list2):

        if list1[i] < list2[j]:

            merged\_list.append(list1[i])

            i += 1

        else:

            merged\_list.append(list2[j])

            j += 1

    # Append remaining elements of list1 (if any)

    while i < len(list1):

        merged\_list.append(list1[i])

        i += 1

    # Append remaining elements of list2 (if any)

    while j < len(list2):

        merged\_list.append(list2[j])

        j += 1

    return merged\_list

# Driver code

list1 = [1, 3, 5, 7]

list2 = [2, 4, 6, 8]

result = merge\_sorted\_lists(list1, list2)

print("Merged sorted list:", result)

def find\_max\_min\_avg(numbers):

    maximum = max(numbers)

    minimum = min(numbers)

    average = sum(numbers) / len(numbers)

    return maximum, minimum, average

# Driver code

nums = [10, 25, 5, 40, 15]

max\_num, min\_num, avg\_num = find\_max\_min\_avg(nums)

print(f"Maximum: {max\_num}")

print(f"Minimum: {min\_num}")

print(f"Average: {avg\_num:.2f}")

# Example list

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

# Initialize sum

total = 0

# Loop through the list and add each element to total

for num in numbers:

    total += num

# Print the result

print("Sum of all elements:", total)

# Input string

text = input("Enter a string: ")

# Initialize counters

vowels = 0

consonants = 0

# Define vowel characters

vowel\_chars = "aeiouAEIOU"

# Loop through each character in the string

for char in text:

    if char.isalpha():  # Check if the character is a letter

        if char in vowel\_chars:

            vowels += 1

        else:

            consonants += 1

# Output the results

print("Number of vowels:", vowels)

print("Number of consonants:", consonants)

# Input string

text = input("Enter a string: ")

# Step 1: Replace spaces with hyphens

text = text.replace(' ', '-')

# Step 2: Convert to lowercase

text = text.lower()

# Step 3: Remove all digits

result = ''.join(char for char in text if not char.isdigit())

# Output the result

print("Processed string:", result)

import string

# Input sentence

sentence = input("Enter a sentence: ")

# Step 1: Remove punctuation

no\_punct = ''.join(char for char in sentence if char not in string.punctuation)

# Step 2: Split into words

words = no\_punct.split()

# Step 3: Reverse word order

reversed\_words = words[::-1]

# Step 4: Capitalize first letter of each word

capitalized\_words = [word.capitalize() for word in reversed\_words]

# Join and print results

print("Words in reverse order:", ' '.join(reversed\_words))

print("Capitalized words:", ' '.join(capitalized\_words))

print("Sentence without punctuation:", no\_punct)

# Initialize an empty dictionary

students = {}

# Input names and marks for 5 students

for i in range(5):

    name = input(f"Enter name of student {i+1}: ")

    marks = float(input(f"Enter marks of {name}: "))

    students[name] = marks

# Display all student names and marks

print("\nStudent Names and Marks:")

for name, marks in students.items():

    print(f"{name}: {marks}")

# Example dictionaries

dict1 = {'a': 10, 'b': 20, 'c': 30}

dict2 = {'b': 5, 'c': 15, 'd': 40}

# Merged dictionary

merged\_dict = dict1.copy()

# Merge dict2 into merged\_dict

for key, value in dict2.items():

    if key in merged\_dict:

        merged\_dict[key] += value  # Sum values if key exists

    else:

        merged\_dict[key] = value   # Add new key-value pair

# Display the result

print("Merged Dictionary:", merged\_dict)

# Example dictionary

original\_dict = {

    'a': 10,

    'b': 20,

    'c': 10,

    'd': 30,

    'e': 20

}

# Set to track seen values

seen\_values = set()

# New dictionary with unique values only

unique\_dict = {}

for key, value in original\_dict.items():

    if value not in seen\_values:

        unique\_dict[key] = value

        seen\_values.add(value)

# Display the result

print("Dictionary after removing duplicate values:", unique\_dict)

# Sample nested dictionary of students and their subject marks

students = {

    'Alice': {'Math': 85, 'English': 78, 'Science': 92},

    'Bob': {'Math': 75, 'English': 88, 'Science': 80},

    'Charlie': {'Math': 90, 'English': 82, 'Science': 85}

}

# Calculate and print average marks for each student

print("Average Marks for Each Student:")

for student, subjects in students.items():

    total\_marks = sum(subjects.values())

    num\_subjects = len(subjects)

    average = total\_marks / num\_subjects

    print(f"{student}: {average:.2f}")

# Sample dictionary of products and prices

products = {

    'apple': 0.5,

    'banana': 0.3,

    'orange': 0.7

}

# Ask user for product name

product\_name = input("Enter the product name to update: ").lower()

# Check if product exists and update price

if product\_name in products:

    try:

        new\_price = float(input(f"Enter the new price for {product\_name}: "))

        products[product\_name] = new\_price

        print(f"Price of '{product\_name}' updated to {new\_price}")

    except ValueError:

        print("Invalid price entered. Please enter a numeric value.")

else:

    print(f"Error: Product '{product\_name}' does not exist.")

# Optional: print updated dictionary

print("Updated product prices:", products)

**Lab 5: OOP EXPENTION HANDLING, NUM PY**

**Code:**

try:

    amount\_str = input("Enter the amount to withdraw: ")

    # Try converting input to float

    amount = float(amount\_str)

    # Check if amount is positive

    if amount <= 0:

        raise ValueError("Amount must be greater than zero.")

    print(f"Withdrawal amount accepted: ${amount:.2f}")

except ValueError as ve:

    print(f"Invalid input: {ve}")

except TypeError:

    print("Invalid type entered. Please enter a numeric value.")

class Book:

    def \_\_init\_\_(self, title, author):

        self.title = title

        self.author = author

        self.available = True  # Book is available by default

    def borrow\_book(self):

        if self.available:

            self.available = False

            print(f"You have successfully borrowed '{self.title}'.")

        else:

            print(f"Sorry, '{self.title}' is currently not available.")

    def return\_book(self):

        if not self.available:

            self.available = True

            print(f"You have returned '{self.title}'. Thank you!")

        else:

            print(f"'{self.title}' was not borrowed.")

    def \_\_str\_\_(self):

        status = "Available" if self.available else "Not Available"

        return f"'{self.title}' by {self.author} - {status}"

# Base class

class Animal:

    def make\_sound(self):

        print("Some generic animal sound")

# Derived class: Dog

class Dog(Animal):

    def make\_sound(self):

        print("Woof! Woof!")

# Derived class: Cat

class Cat(Animal):

    def make\_sound(self):

        print("Meow!")

# Derived class: Bird

class Bird(Animal):

    def make\_sound(self):

        print("Tweet! Tweet!")

# Demonstration of polymorphism

def animal\_sounds\_demo():

    # Create a list of animal objects

    animals = [Dog(), Cat(), Bird(), Dog(), Bird()]

    print("Animal Shelter Sound Check:")

    for animal in animals:

        animal.make\_sound()  # Polymorphic call

# Run the demo

animal\_sounds\_demo()

import numpy as np

# Sample 2D array: rows = cities, columns = daily temperatures

# Example: 3 cities, 4 days each

temperature\_data = np.array([

    [30, 32, 33, 31],   # City 0

    [25, 27, 26, 28],   # City 1

    [35, 36, 34, 37]    # City 2

])

# 1. Max temperature in each city

max\_temps = np.max(temperature\_data, axis=1)

print("Maximum temperature in each city:", max\_temps)

# 2. Average temperature across all cities

overall\_avg = np.mean(temperature\_data)

print("Overall average temperature:", round(overall\_avg, 2))

# 3. Identify cities with average temp above overall average

city\_averages = np.mean(temperature\_data, axis=1)

above\_avg\_cities = np.where(city\_averages > overall\_avg)[0]

print("Cities with average temperatures above overall average:", above\_avg\_cities)

import numpy as np

# Sample 2D temperature data (rows = cities, columns = daily temps)

temperature\_data = np.array([

    [30, 32, 33, 31],   # City 0

    [25, 27, 26, 28],   # City 1

    [35, 36, 34, 37]    # City 2

])

# 1. Find maximum temperature in each city (row-wise max)

max\_temps = np.max(temperature\_data, axis=1)

print("Maximum temperature in each city:", max\_temps)

# 2. Calculate overall average temperature

overall\_avg = np.mean(temperature\_data)

print("Overall average temperature:", round(overall\_avg, 2))

# 3. Identify cities with average temperatures above overall average

city\_averages = np.mean(temperature\_data, axis=1)

above\_avg\_indices = np.where(city\_averages > overall\_avg)[0]

# Display results

print("Cities with average temp above overall average:", above\_avg\_indices)

# Optional: show city data

for index in above\_avg\_indices:

    print(f"City {index} data: {temperature\_data[index]} (Avg: {city\_averages[index]:.2f})")

**lab 6: Pandas**

**code:**

#Summary of Actions Performed:

#Viewed first & last rows

#Described the dataset (data types, stats)

#Found missing values:

#Age → filled with median

#Fare → dropped 1 row

#Cabin → dropped column (78% missing)

#Created new column: FamilySize

#Dropped irrelevant column: Ticket

#Filtered dataset: Survived & Pclass = 1

#Let me know if you’d like:

#Visualizations (e.g., survival rate by class/gender)

#Exporting the cleaned dataset

#Advanced analysis or modeling

import numpy as np

import pandas as pd

df = pd.read\_csv('/content/Titanicdataset (1)(1).csv')

print(df)

# 1. View the first and last few rows

print(df.head())

print(df.tail())

# 2. Basic info and statistical summary

print(df.info())

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

# 3. Check for missing/null values

missing\_values = df.isnull().sum()

missing\_percent = (missing\_values / len(df)) \* 100

missing\_summary = pd.DataFrame({

    'Missing Values': missing\_values,

    'Percentage (%)': missing\_percent.round(2)

})

print(missing\_summary)

 # 4. Handle missing values

# Fill 'Age' with median

df['Age'].fillna(df['Age'].median(), inplace=True)

# Drop row with missing Fare

df.dropna(subset=['Fare'], inplace=True)

# Drop 'Cabin' column (too many missing)

df.drop(columns=['Cabin'], inplace=True)

# 5. Add a new column: FamilySize

df['FamilySize'] = df['SibSp'] + df['Parch'] + 1

# 6. Drop an irrelevant column: Ticket

df.drop(columns=['Ticket'], inplace=True)

# 7. Filter dataset: passengers who survived and were in 1st class

filtered\_df = df[(df['Survived'] == 1) & (df['Pclass'] == 1)]

print(filtered\_df.head())

# Group by Sex and Pclass, with aggregation on Age and Survived

grouped = df.groupby(['Sex', 'Pclass']).agg({

    'Age': 'mean',

    'Survived': ['mean', 'count']

}).reset\_index()

# Rename columns

grouped.columns = ['Sex', 'Pclass', 'AvgAge', 'SurvivalRate', 'Count']

print(grouped.head())

import numpy as np

import pandas as pd

df = pd.read\_csv('/content/salesdataset - salesdataset.csv')

print(df)

# 1. View first and last rows

print(df.head())

print(df.tail())

# 2. Info & Statistical Summary

print(df.info())

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

# 3. Check for missing values

missing = df.isnull().sum()

missing\_percent = (missing / len(df)) \* 100

print(pd.DataFrame({'Missing Values': missing, 'Percentage (%)': missing\_percent.round(2)}))

# 4. Handle missing values

df.drop(columns=['ADDRESSLINE2'], inplace=True)  # drop mostly-missing column

df['STATE'].fillna('Unknown', inplace=True)       # fill missing states

df.dropna(subset=['POSTALCODE'], inplace=True)    # drop few rows

# 5. Add new column: REVENUE = QUANTITYORDERED × PRICEEACH

df['REVENUE'] = df['QUANTITYORDERED'] \* df['PRICEEACH']

# 6. Drop irrelevant column

df.drop(columns=['PHONE'], inplace=True)

# 7. Filter dataset: orders from USA with large deal size

filtered\_df = df[(df['COUNTRY'] == 'USA') & (df['DEALSIZE'] == 'Large')]

print(filtered\_df.head())

#partB

# Group by COUNTRY — total, average, and count of SALES

country\_group = df.groupby('COUNTRY')['SALES'].agg(['sum', 'mean', 'count']).reset\_index()

print(country\_group.head())

# Group by PRODUCTLINE and DEALSIZE — total SALES

product\_deal\_group = df.groupby(['PRODUCTLINE', 'DEALSIZE'])['SALES'].sum().reset\_index()

print(product\_deal\_group.head())

**lab 7 : matplotlib**

**code:**

import pandas as pd

import matplotlib.pyplot as plt

# Load dataset

df = pd.read\_csv("/content/Titanicdataset (1)(1).csv")

print(df)

# ---------------- Box Plot ----------------

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

sns.boxplot(x='Survived', y='Age', data=df)

plt.title('Box Plot of Age by Survival')

plt.xlabel('Survived (0 = No, 1 = Yes)')

plt.ylabel('Age')

plt.tight\_layout()

plt.show()

import pandas as pd

import matplotlib.pyplot as plt

# Load dataset

df = pd.read\_csv("/content/Titanicdataset (1)(1).csv")

print(df)

 # ---------------- Scatter Plot ----------------

plt.figure(figsize=(8, 7))

sns.scatterplot(x='Age', y='Fare', hue='Survived', data=df)

plt.title('Scatter Plot of Age vs Fare')

plt.xlabel('Age')

plt.ylabel('Fare')

plt.tight\_layout()

plt.show()

import pandas as pd

import matplotlib.pyplot as plt

# Load dataset

df = pd.read\_csv("/content/Titanicdataset (1)(1).csv")

print(df)

# ---------------- Histogram ----------------

plt.figure(figsize=(9, 8))

sns.histplot(df['Age'].dropna(), bins=20, kde=True)

plt.title('Age Distribution')

plt.xlabel('Age')

plt.ylabel('Frequency')

plt.tight\_layout()

plt.show()

import pandas as pd

import matplotlib.pyplot as plt

# Load dataset

df = pd.read\_csv("/content/Titanicdataset (1)(1).csv")

print(df)

# ---------------- Line Plot ----------------

plt.figure(figsize=(9, 8))

sns.lineplot(x='PassengerId', y='Fare', data=df)

plt.title('Line Plot of Fare by Passenger ID')

plt.xlabel('Passenger ID')

plt.ylabel('Fare')

plt.tight\_layout()

plt.show()

 import pandas as pd

import matplotlib.pyplot as plt

# Load dataset

df = pd.read\_csv("/content/Titanicdataset (1)(1).csv")

print(df)

 # ---------------- Pie Chart ----------------

survived\_counts = df['Survived'].value\_counts()

plt.figure(figsize=(8, 7))

plt.pie(survived\_counts, labels=['Not Survived', 'Survived'], autopct='%1.1f%%', startangle=90)

plt.title('Survival Rate')

plt.tight\_layout()

plt.show()

import pandas as pd

import matplotlib.pyplot as plt

# Load dataset

df = pd.read\_csv("/content/Titanicdataset (1)(1).csv")

print(df)

# ---------------- Bar Plot ----------------

plt.figure(figsize=(8, 6))

sns.countplot(x='Embarked', data=df)

plt.title('Count of Passengers by Embarked')

plt.xlabel('Port of Embarkation')

plt.ylabel('Count')

plt.tight\_layout()

plt.show()

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Load your dataset

df = pd.read\_csv("/content/Titanicdataset (1)(1).csv")

print(df)

# Create 2x3 subplots

fig, axes = plt.subplots(2, 3, figsize=(18, 10))

plt.subplots\_adjust(hspace=0.4, wspace=0.4)

# 1. Bar Plot - Frequency of 'Embarked'

sns.countplot(x='Embarked', data=df, ax=axes[0, 0])

axes[0, 0].set\_title('Count of Passengers by Embarked')

axes[0, 0].set\_xlabel('Port of Embarkation')

axes[0, 0].set\_ylabel('Count')

# 2. Scatter Plot - Relationship between 'Age' and 'Fare'

sns.scatterplot(x='Age', y='Fare', hue='Survived', data=df, ax=axes[0, 1])

axes[0, 1].set\_title('Scatter: Age vs Fare')

axes[0, 1].set\_xlabel('Age')

axes[0, 1].set\_ylabel('Fare')

# 3. Histogram - Distribution of 'Age'

sns.histplot(df['Age'].dropna(), bins=20, kde=True, ax=axes[0, 2])

axes[0, 2].set\_title('Histogram of Age')

axes[0, 2].set\_xlabel('Age')

axes[0, 2].set\_ylabel('Frequency')

# 4. Line Plot - Fare by Passenger ID

sns.lineplot(x='PassengerId', y='Fare', data=df, ax=axes[1, 0])

axes[1, 0].set\_title('Line Plot: Fare by Passenger ID')

axes[1, 0].set\_xlabel('Passenger ID')

axes[1, 0].set\_ylabel('Fare')

# 5. Pie Chart - Survival Rate

survived\_counts = df['Survived'].value\_counts()

axes[1, 1].pie(survived\_counts, labels=['Not Survived', 'Survived'], autopct='%1.1f%%', startangle=90)

axes[1, 1].set\_title('Pie Chart: Survival Rate')

# 6. Box Plot - Age by Survived

sns.boxplot(x='Survived', y='Age', data=df, ax=axes[1, 2])

axes[1, 2].set\_title('Box Plot: Age by Survival')

axes[1, 2].set\_xlabel('Survived (0 = No, 1 = Yes)')

axes[1, 2].set\_ylabel('Age')

# Show all plots

plt.tight\_layout()

plt.show()

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Load dataset

df = pd.read\_csv("/content/IRIS - IRIS.csv")

print(df)

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Load dataset

df = pd.read\_csv("/content/IRIS - IRIS.csv")

print(df)

#Bar plot

sns.countplot(data=df, x='species')

plt.title("Count of Each Species")

plt.xlabel("Species")

plt.ylabel("Count")

plt.show()

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Load dataset

df = pd.read\_csv("/content/IRIS - IRIS.csv")

print(df)

#Line plot

sns.lineplot(data=df, x=df.index, y='sepal\_length', hue='species')

plt.title("Sepal Length Trend Across Index")

plt.xlabel("Index")

plt.ylabel("Sepal Length")

plt.show()

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Load dataset

df = pd.read\_csv("/content/IRIS - IRIS.csv")

print(df)

#Scatter plot

sns.scatterplot(data=df, x='sepal\_length', y='sepal\_width', hue='species')

plt.title("Sepal Length vs Sepal Width")

plt.xlabel("Sepal Length")

plt.ylabel("Sepal Width")

plt.show()

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Load dataset

df = pd.read\_csv("/content/IRIS - IRIS.csv")

print(df)

#histogram plot

sns.histplot(df['petal\_length'], bins=20, kde=True)

plt.title("Distribution of Petal Length")

plt.xlabel("Petal Length")

plt.ylabel("Frequency")

plt.show()

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Load dataset

df = pd.read\_csv("/content/IRIS - IRIS.csv")

print(df)

# pie chart

species\_counts = df['species'].value\_counts()

plt.pie(species\_counts, labels=species\_counts.index, autopct='%1.1f%%', startangle=140)

plt.title("Species Distribution")

plt.axis('equal')

plt.show()

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Load dataset

df = pd.read\_csv("/content/IRIS - IRIS.csv")

print(df)

#Box plot

sns.boxplot(data=df, x='species', y='petal\_width')

plt.title("Boxplot of Petal Width by Species")

plt.xlabel("Species")

plt.ylabel("Petal Width")

plt.show()

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Load dataset

df = pd.read\_csv("/content/IRIS - IRIS.csv")

print(df)

# Create a 2x3 subplot layout

fig, axes = plt.subplots(2, 3, figsize=(18, 10))

plt.subplots\_adjust(hspace=0.4, wspace=0.4)

# 1. Bar Plot - Frequency of 'species'

sns.countplot(x='species', data=df, ax=axes[0, 0])

axes[0, 0].set\_title('Count of Each Species')

# 2. Scatter Plot - sepal\_length vs petal\_length

sns.scatterplot(x='sepal\_length', y='petal\_length', hue='species', data=df, ax=axes[0, 1])

axes[0, 1].set\_title('Sepal Length vs Petal Length')

# 3. Histogram - Distribution of 'sepal\_width'

sns.histplot(df['sepal\_width'], bins=20, kde=True, ax=axes[0, 2])

axes[0, 2].set\_title('Distribution of Sepal Width')

# 4. Line Plot - Petal Length over Index

sns.lineplot(x=df.index, y='petal\_length', hue='species', data=df, ax=axes[1, 0])

axes[1, 0].set\_title('Petal Length over Index')

# 5. Pie Chart - Share of Each Species

species\_counts = df['species'].value\_counts()

axes[1, 1].pie(species\_counts, labels=species\_counts.index, autopct='%1.1f%%', startangle=90)

axes[1, 1].set\_title('Species Distribution')

# 6. Box Plot - Sepal Length by Species

sns.boxplot(x='species', y='sepal\_length', data=df, ax=axes[1, 2])

axes[1, 2].set\_title('Sepal Length by Species')

# Show all plots

plt.tight\_layout()

plt.show()

**lab 8: longistic regression**

**code:**

import numpy as np

import pandas as pd

df = pd.read\_csv('/content/Breast Cancer Diagnostic Dataset - Breast Cancer Diagnostic Dataset.csv')

print(df)

# 1. Check for missing/null values

print("Missing values in each column:\n", df.isnull().sum())

# 2. Handle missing data

# Option A: Fill missing values with mean (for numerical columns)

df\_filled = df.fillna(df.mean(numeric\_only=True))

# Option B: Or Drop rows with any missing values

# df\_dropped = df.dropna()

# 3. Add a new column (example: 'radius\_mean\_squared' = radius\_mean ^ 2)

if 'radius\_mean' in df.columns:

    df\_filled['radius\_mean\_squared'] = df\_filled['radius\_mean'] \*\* 2

# 4. Drop an irrelevant column (example: 'id')

if 'id' in df\_filled.columns:

    df\_filled = df\_filled.drop(columns=['id'])

# 5. Filter the dataset (example: keep only rows with diagnosis == 'M')

if 'diagnosis' in df\_filled.columns:

    df\_filtered = df\_filled[df\_filled['diagnosis'] == 'M']

    print("Filtered rows (Malignant cases):", df\_filtered.shape[0])

# Display final few rows

print(df\_filtered.head())

# Display basic information

print("Columns in dataset:", df.columns.tolist())

print("\nFirst 5 rows:\n", df.head())

#  6 Group by 'diagnosis' and count number of instances

grouped\_count = df.groupby('diagnosis').size()

print("\nCount of diagnoses:\n", grouped\_count)

# 7 Group by 'diagnosis' and calculate mean of numerical features

grouped\_mean = df.groupby('diagnosis').mean()

print("\nMean values grouped by diagnosis:\n", grouped\_mean)

# 8 Group by 'diagnosis' and calculate sum of numerical features

grouped\_sum = df.groupby('diagnosis').sum()

print("\nSum of values grouped by diagnosis:\n", grouped\_sum)

# Step 1: Import required libraries

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

# Step 2: Load the dataset

df = pd.read\_csv("Breast Cancer Diagnostic Dataset - Breast Cancer Diagnostic Dataset.csv")

# Step 3: Explore the dataset

print("First 5 rows:\n", df.head())

print("\nDataset info:")

print(df.info())

# Step 4: Drop any irrelevant columns

# Assuming 'id' or unnamed columns exist

df = df.drop(columns=[col for col in df.columns if 'Unnamed' in col or col.lower() == 'id'])

# Step 5: Encode the target column

# Assuming target column is 'diagnosis' with values 'M' and 'B'

df['diagnosis'] = df['diagnosis'].map({'M': 1, 'B': 0})

# Step 6: Split features and target

X = df.drop('diagnosis', axis=1)

y = df['diagnosis']

# Step 7: Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Step 8: Create and train the logistic regression model

model = LogisticRegression(max\_iter=10000)

model.fit(X\_train, y\_train)

# Step 9: Make predictions

y\_pred = model.predict(X\_test)

# Step 10: Evaluate the model

accuracy = accuracy\_score(y\_test, y\_pred)

precision = precision\_score(y\_test, y\_pred)

recall = recall\_score(y\_test, y\_pred)

f1 = f1\_score(y\_test, y\_pred)

# Step 11: Print the results

print("\nModel Evaluation Metrics:")

print(f"Accuracy  : {accuracy:.4f}")

print(f"Precision : {precision:.4f}")

print(f"Recall    : {recall:.4f}")

print(f"F1 Score  : {f1:.4f}")

import numpy as np

import pandas as pd

df = pd.read\_csv('/content/healthcare-dataset-stroke-data - healthcare-dataset-stroke-data.csv')

print(df)

# Logistic Regression on the Stroke Dataset

# Part 1: With preprocessing

# Part 2: Without preprocessing

# Evaluation + automatic comparison & short conclusion

import os

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import OneHotEncoder, StandardScaler

from sklearn.impute import SimpleImputer

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import (

    accuracy\_score,

    confusion\_matrix,

    precision\_recall\_fscore\_support,

    classification\_report,

)

RANDOM\_STATE = 42

np.random.seed(RANDOM\_STATE)

# -----------------------------

# Helper printing/evaluation

# -----------------------------

def evaluate\_model(y\_true, y\_pred, label="Model"):

    acc = accuracy\_score(y\_true, y\_pred)

    precision, recall, f1, \_ = precision\_recall\_fscore\_support(

        y\_true, y\_pred, average="binary", zero\_division=0

    )

    cm = confusion\_matrix(y\_true, y\_pred)

    print(f"\n===== {label} =====")

    print(f"Accuracy     : {acc:.4f}")

    print(f"Precision    : {precision:.4f}")

    print(f"Recall       : {recall:.4f}")

    print(f"F1-Score     : {f1:.4f}")

    print("Confusion Matrix:\n", cm)

    print("\nClassification Report:\n", classification\_report(y\_true, y\_pred, zero\_division=0))

    return {"acc": acc, "precision": precision, "recall": recall, "f1": f1, "cm": cm}

# Target column

TARGET = "stroke"

if TARGET not in df.columns:

    raise ValueError(f"Target column '{TARGET}' not found in the dataset.")

# -----------------------------

# PART 1: WITH PREPROCESSING

# -----------------------------

# Separate X, y

X = df.drop(columns=[TARGET])

y = df[TARGET].astype(int)

# Identify column types

numeric\_features = X.select\_dtypes(include=["int64", "float64"]).columns.tolist()

# Do not scale 'id' (it's just an identifier) -> better to drop it

if "id" in numeric\_features:

    numeric\_features.remove("id")

    X = X.drop(columns=["id"])

categorical\_features = X.select\_dtypes(include=["object", "category"]).columns.tolist()

# Build preprocessing pipeline

numeric\_transformer = Pipeline(

    steps=[

        ("imputer", SimpleImputer(strategy="median")),

        ("scaler", StandardScaler()),

    ]

)

categorical\_transformer = Pipeline(

    steps=[

        ("imputer", SimpleImputer(strategy="most\_frequent")),

        ("onehot", OneHotEncoder(handle\_unknown="ignore")),

    ]

)

preprocessor = ColumnTransformer(

    transformers=[

        ("num", numeric\_transformer, numeric\_features),

        ("cat", categorical\_transformer, categorical\_features),

    ],

    remainder="drop",

)

# Full pipeline with Logistic Regression

log\_reg = LogisticRegression(max\_iter=1000, random\_state=RANDOM\_STATE)

clf = Pipeline(steps=[("preprocessor", preprocessor), ("model", log\_reg)])

# Train/test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, y, test\_size=0.2, random\_state=RANDOM\_STATE, stratify=y

)

# Fit and predict

clf.fit(X\_train, y\_train)

y\_pred\_pre = clf.predict(X\_test)

metrics\_pre = evaluate\_model(y\_test, y\_pred\_pre, label="Logistic Regression WITH preprocessing")

# -----------------------------

# PART 2: WITHOUT PREPROCESSING

# (Load data, keep only numeric columns, drop NaNs, no scaling/encoding/imputation beyond the drops)

# -----------------------------

X2 = df.drop(columns=[TARGET]).copy()

y2 = df[TARGET].astype(int).copy()

# Keep numeric columns only (no encoding). Also drop 'id' if exists.

X2 = X2.select\_dtypes(include=["int64", "float64"])

if "id" in X2.columns:

    X2 = X2.drop(columns=["id"])

# Drop any rows with NaNs to avoid imputation

mask\_no\_na = X2.dropna().index

X2 = X2.loc[mask\_no\_na]

y2 = y2.loc[mask\_no\_na]

# If the target became single-class after dropping, warn

if y2.nunique() < 2:

    raise ValueError(

        "After dropping NaNs, the target has <2 classes. "

        "Cannot train a binary classifier without both classes present."

    )

X2\_train, X2\_test, y2\_train, y2\_test = train\_test\_split(

    X2, y2, test\_size=0.2, random\_state=RANDOM\_STATE, stratify=y2

)

log\_reg\_no\_prep = LogisticRegression(max\_iter=1000, random\_state=RANDOM\_STATE)

log\_reg\_no\_prep.fit(X2\_train, y2\_train)

y\_pred\_no = log\_reg\_no\_prep.predict(X2\_test)

metrics\_no = evaluate\_model(y2\_test, y\_pred\_no, label="Logistic Regression WITHOUT preprocessing")

# -----------------------------

# COMPARISON & SHORT CONCLUSION

# -----------------------------

def pct(a):

    return f"{100\*a:.2f}%"

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

print(f"Accuracy  : with prep = {pct(metrics\_pre['acc'])} | without prep = {pct(metrics\_no['acc'])}")

print(f"Precision : with prep = {pct(metrics\_pre['precision'])} | without prep = {pct(metrics\_no['precision'])}")

print(f"Recall    : with prep = {pct(metrics\_pre['recall'])} | without prep = {pct(metrics\_no['recall'])}")

print(f"F1-Score  : with prep = {pct(metrics\_pre['f1'])} | without prep = {pct(metrics\_no['f1'])}")

# Generate a short auto-conclusion

better = []

worse = []

same = []

for k in ["acc", "precision", "recall", "f1"]:

    if np.isclose(metrics\_pre[k], metrics\_no[k], atol=1e-4):

        same.append(k)

    elif metrics\_pre[k] > metrics\_no[k]:

        better.append(k)

    else:

        worse.append(k)

print("\n---------------- Conclusion (Auto-generated) ----------------")

lines = []

if better:

    lines.append(f"Preprocessing improved {', '.join(better)}.")

if worse:

    lines.append(f"Without preprocessing performed better on {', '.join(worse)} (possibly due to data leakage/variance).")

if same:

    lines.append(f"Both approaches were similar on {', '.join(same)}.")

if not lines:

    lines.append("Both approaches performed very similarly across all metrics.")

lines.append(

    "Overall, proper preprocessing (imputing missing values, encoding categoricals, and scaling numerics) "

    "typically yields a more stable and often better-performing Logistic Regression model, especially on imbalanced, mixed-type datasets."

)

print("\n".join(lines))

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

**lab 9: Decision Tree and Random Forest :**

**code:**

**A) Dcision Tree**

import numpy as np

import pandas as pd

df = pd.read\_csv('/content/Titanicdataset (1)(1).csv')

print(df)

# View the first and last few rows

print("First 5 rows:")

print(df.head())

print("\nLast 5 rows:")

print(df.tail())

# Get basic information about columns, data types

print("\nDataset Info:")

print(df.info())

# Get statistical summaries

print("\nStatistical Summary:")

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

# Check for Missing Values

print("\nMissing Values:")

print(df.isnull().sum())

# Handle Missing Values (example: fill Age with mean if column exists)

if 'Age' in df.columns:

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

# Drop an irrelevant column (example: PassengerId)

if 'PassengerId' in df.columns:

    df.drop('PassengerId', axis=1, inplace=True)

# Add a new column (example: FamilySize = SibSp + Parch + 1)

if 'SibSp' in df.columns and 'Parch' in df.columns:

    df['FamilySize'] = df['SibSp'] + df['Parch'] + 1

# Filter the dataset (example: Age > 30)

if 'Age' in df.columns:

    filtered\_df = df[df['Age'] > 30]

    print("\nFiltered Data (Age > 30):")

    print(filtered\_df.head())

# --- Part B: Grouping and Aggregation ---

# Group the data by 'Pclass' and calculate mean of Age and Fare

if 'Pclass' in df.columns:

    grouped = df.groupby('Pclass').agg({

        'Age': 'mean',

        'Fare': 'mean'

    })

    print("\nGrouped Results by Pclass:")

    print(grouped)

# Decision Tree Model with Evaluation on Titanic Dataset

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score, confusion\_matrix, ConfusionMatrixDisplay

import matplotlib.pyplot as plt

# Load dataset

df = pd.read\_csv("Titanicdataset (1)(1).csv")

# Basic data cleaning (drop unnecessary columns, handle NaN)

df = df.drop(columns=['Name', 'Ticket', 'Cabin'], errors='ignore')  # remove non-numeric columns if present

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

df['Embarked'].fillna(df['Embarked'].mode()[0], inplace=True)

# Convert categorical columns to numeric

df = pd.get\_dummies(df, drop\_first=True)

# Define features (X) and target (y)

X = df.drop('Survived', axis=1)

y = df['Survived']

# Split data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Train Decision Tree

model = DecisionTreeClassifier(random\_state=42)

model.fit(X\_train, y\_train)

# Predictions

y\_pred = model.predict(X\_test)

# Evaluation

accuracy = accuracy\_score(y\_test, y\_pred)

cm = confusion\_matrix(y\_test, y\_pred)

print("Accuracy Score:", accuracy)

print("Confusion Matrix:\n", cm)

# Confusion Matrix Visualization

disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=model.classes\_)

disp.plot(cmap=plt.cm.Blues)

plt.show()

B) Randon Forest:

import numpy as np

import pandas as pd

df = pd.read\_csv('/content/Loan\_dataset - Loan\_dataset.csv')

print(df)

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.compose import ColumnTransformer

from sklearn.preprocessing import OneHotEncoder

from sklearn.impute import SimpleImputer

from sklearn.pipeline import Pipeline

from sklearn.metrics import (

    accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report

)

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

#  Pick/guess the target

# -----------------------------

possible\_targets = ["Loan\_Status", "loan\_status", "Status", "status", "Target", "target", "Outcome", "outcome"]

TARGET\_COLUMN = next((c for c in possible\_targets if c in df.columns), df.columns[-1])

y\_raw = df[TARGET\_COLUMN]

# map target to 0/1 if it's a yes/no or Y/N; otherwise factorize

if y\_raw.dtype == object or str(y\_raw.dtype).startswith("category"):

    up = [str(u).upper() for u in y\_raw.dropna().unique()]

    if set(up) == {"Y", "N"}:

        y = y\_raw.map({"N": 0, "Y": 1}).astype(int)

    elif set(up) == {"YES", "NO"}:

        y = y\_raw.map({"NO": 0, "YES": 1}).astype(int)

    else:

        y, \_ = pd.factorize(y\_raw)

else:

    y = y\_raw.copy()

X = df.drop(columns=[TARGET\_COLUMN])

# -----------------------------

# Separate categorical & numeric; force binary categoricals to 0/1

# -----------------------------

cat\_cols = X.select\_dtypes(include=["object", "category"]).columns.tolist()

num\_cols = X.select\_dtypes(exclude=["object", "category"]).columns.tolist()

binary\_cols = [c for c in cat\_cols if X[c].nunique(dropna=True) == 2]

for c in binary\_cols:

    uniq = X[c].dropna().unique()

    up = [str(u).upper() for u in uniq]

    if set(up) == {"Y", "N"}:

        mapping = {"N": 0, "Y": 1}

    elif set(up) == {"YES", "NO"}:

        mapping = {"NO": 0, "YES": 1}

    else:

        mapping = {uniq[0]: 0, uniq[1]: 1}

    X[c] = X[c].map(mapping)

    num\_cols.append(c)

cat\_cols = [c for c in cat\_cols if c not in binary\_cols]

# -----------------------------

#  Preprocess (impute + encode)

# -----------------------------

numeric\_transformer = SimpleImputer(strategy="median")

categorical\_transformer = Pipeline(steps=[

    ("imputer", SimpleImputer(strategy="most\_frequent")),

    ("onehot", OneHotEncoder(drop="first", handle\_unknown="ignore"))

])

preprocess = ColumnTransformer(

    transformers=[

        ("num", numeric\_transformer, num\_cols),

        ("cat", categorical\_transformer, cat\_cols)

    ],

    remainder="drop"

)

# -----------------------------

# 5) Train / test split

# -----------------------------

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, y, test\_size=0.2, random\_state=42,

    stratify=y if len(np.unique(y)) > 1 else None

)

# -----------------------------

#  Models

# -----------------------------

dt\_clf = Pipeline(steps=[

    ("preprocess", preprocess),

    ("model", DecisionTreeClassifier(random\_state=42))

])

rf\_clf = Pipeline(steps=[

    ("preprocess", preprocess),

    ("model", RandomForestClassifier(random\_state=42))

])

dt\_clf.fit(X\_train, y\_train)

rf\_clf.fit(X\_train, y\_train)

y\_pred\_dt = dt\_clf.predict(X\_test)

y\_pred\_rf = rf\_clf.predict(X\_test)

avg = "binary" if len(np.unique(y)) == 2 else "weighted"

def print\_metrics(name, y\_true, y\_pred):

    print(f"\n=== {name} ===")

    print(f"Accuracy : {accuracy\_score(y\_true, y\_pred):.4f}")

    print(f"Precision: {precision\_score(y\_true, y\_pred, average=avg, zero\_division=0):.4f}")

    print(f"Recall   : {recall\_score(y\_true, y\_pred, average=avg, zero\_division=0):.4f}")

    print(f"F1-score : {f1\_score(y\_true, y\_pred, average=avg, zero\_division=0):.4f}")

    print("\nClassification report:\n", classification\_report(y\_true, y\_pred, zero\_division=0))

print\_metrics("Decision Tree", y\_test, y\_pred\_dt)

print\_metrics("Random Forest", y\_test, y\_pred\_rf)