**AI ASSISTED CODING**

**LAB-5.2**

TASK-1:

Use an AI tool (e.g., Copilot, Gemini, Cursor) to generate a login system. Review the  
generated code for hardcoded passwords, plain-text storage, or lack of encryption

def create\_student\_account():

    """Creates a new student account with a username and password."""

    username = input("Enter a username for the new student: ")

    password = input("Enter a password for the new student: ")

    # In a real system, you would securely store this data (e.g., in a database)

    # For this example, we'll just store it in a dictionary

    student\_accounts[username] = password

    print("Account created successfully!")

def student\_login():

    """Allows a student to log in using their username and password."""

    username = input("Enter your username: ")

    password = input("Enter your password: ")

    if username in student\_accounts and student\_accounts[username] == password:

        print("Login successful!")

        return True

    else:

        print("Invalid username or password.")

        return False

# Dictionary to store student accounts (for demonstration purposes)

student\_accounts = {}

# Example usage:

while True:

    print("\nStudent Login System Menu:")

    print("1. Create new account")

    print("2. Login")

    print("3. Exit")

    choice = input("Enter your choice: ")

    if choice == '1':

        create\_student\_account()

    elif choice == '2':

        student\_login()

    elif choice == '3':

        print("Exiting program.")

        break

    else:

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

OUTPUT:

Student Login System Menu:

1. Create new account

2. Login

3. Exit

Enter your choice: 1

Enter a username for the new student: karthik

Enter a password for the new student: karthik123

Account created successfully!

Student Login System Menu:

1. Create new account

2. Login

3. Exit

Enter your choice: 2

Enter your username: karthik

Enter your password: karthik123

Login successful!

Student Login System Menu:

1. Create new account

2. Login

3. Exit

Enter your choice: 3

Exiting program.

Explanation:

This code implements a simple student login system in Python. Here's a breakdown of how it works:

1. **create\_student\_account() function:**
   * This function prompts the user to enter a username and password for a new student account.
   * It then stores the entered username and password in the student\_accounts dictionary. In a real-world application, you would use a more secure method to store this data, like a database with password hashing.
   * Finally, it prints a success message.
2. **student\_login() function:**
   * This function prompts the user to enter their username and password to log in.
   * It checks if the entered username exists as a key in the student\_accounts dictionary.
   * If the username exists, it then checks if the entered password matches the password stored for that username in the dictionary.
   * If both the username and password match, it prints "Login successful!" and returns True.
   * If the username is not found or the password doesn't match, it prints "Invalid username or password." and returns False.
3. **student\_accounts = {}:**
   * This line initializes an empty dictionary called student\_accounts. This dictionary is used to store the usernames and passwords of the student accounts created during the program's execution.
4. **while True: loop:**
   * This is the main loop of the program, which runs continuously until the user chooses to exit.
   * Inside the loop, it displays a menu with options to "Create new account," "Login," or "Exit."
   * It takes the user's choice as input.
5. **if/elif/else block:**
   * This block checks the user's input choice.
   * If the choice is '1', it calls the create\_student\_account() function.
   * If the choice is '2', it calls the student\_login() function.
   * If the choice is '3', it prints an exit message and uses break to exit the while loop, ending the program.
   * If the choice is anything else, it prints an "Invalid choice" message.

In summary, the program provides a command-line interface for users to create new student accounts and log in using those accounts. The account information is stored in a dictionary for the duration of the program's execution.

TASK-2:

Use prompt variations like: “loan approval for John”, “loan approval for Priya”, etc.  
Evaluate whether the AI-generated logic exhibits bias or differing criteria based on  
names or genders

def approve\_loan(name, income, loan\_amount, credit\_score):

    """

    Approves a loan based on income and credit score.

    Args:

        name (str): The name of the applicant.

        income (float): The applicant's income.

        loan\_amount (float): The requested loan amount.

        credit\_score (int): The applicant's credit score.

    Returns:

        bool: True if the loan is approved, False otherwise.

    """

    # Simple rules: approve if income is sufficient AND credit score is above a threshold

    min\_credit\_score = 600  # Example threshold

    if income >= loan\_amount and credit\_score >= min\_credit\_score:

        print(f"Loan approved for {name}!")

        return True

    else:

        print(f"Loan denied for {name}. Income is not sufficient or credit score is too low.")

        return False

# List of applicants

applicants = ["John", "Priya", "Alice", "Bob", "Charlie"]

# Process each applicant

for name in applicants:

    print(f"\nEnter details for {name}:")

    income = float(input(f"Enter {name}'s income: "))

    loan\_amount = float(input(f"Enter the loan amount requested by {name}: "))

    credit\_score = int(input(f"Enter {name}'s credit score: "))

    # Approve loan for the current applicant

    approve\_loan(name, income, loan\_amount, credit\_score)

output:

Enter details for John:

Enter John's income: 100000

Enter the loan amount requested by John: 500000

Enter John's credit score: 550

Loan denied for John. Income is not sufficient or credit score is too low.

Enter details for Priya:

Enter Priya's income: 50000

Enter the loan amount requested by Priya: 200000

Enter Priya's credit score: 680

Loan denied for Priya. Income is not sufficient or credit score is too low.

Enter details for Alice:

Enter Alice's income: 20000

Enter the loan amount requested by Alice: 30000

Enter Alice's credit score: 600

Loan denied for Alice. Income is not sufficient or credit score is too low.

Enter details for Bob:

Enter Bob's income: 100000

Enter the loan amount requested by Bob: 100000

Enter Bob's credit score: 650

Loan approved for Bob!

Enter details for Charlie:

Enter Charlie's income: 60000

Enter the loan amount requested by Charlie: 100000

Enter Charlie's credit score: 687

Loan denied for Charlie. Income is not sufficient or credit score is too low.

Explanation:

Certainly! This code implements a loan approval system that processes a list of applicants, taking into account their income, requested loan amount, and credit score.

Here's a breakdown of the code:

1. **approve\_loan(name, income, loan\_amount, credit\_score) function:**
   * This function takes four arguments: the applicant's name, income, loan\_amount they are requesting, and their credit\_score.
   * It defines a min\_credit\_score threshold (set to 600 in this example).
   * It then checks if the applicant's income is greater than or equal to the loan\_amount *and* their credit\_score is greater than or equal to the min\_credit\_score.
   * If both conditions are true, the loan is approved, a success message is printed including the applicant's name, and the function returns True.
   * If either condition is false, the loan is denied, a message explaining why is printed, and the function returns False.
2. **applicants = ["John", "Priya", "Alice", "Bob", "Charlie"]:**
   * This line creates a list called applicants containing the names of the individuals applying for a loan. You can easily add or remove names from this list to process different applicants.
3. **for name in applicants: loop:**
   * This for loop iterates through each name in the applicants list.
   * For each applicant, it:
     + Prints a message indicating that it's time to enter details for that specific applicant.
     + Prompts the user to enter the applicant's income, loan\_amount, and credit\_score using the input() function. The input is converted to the appropriate data type (float for income and loan amount, int for credit score).
     + Calls the approve\_loan() function, passing the current applicant's name, income, loan\_amount, and credit\_score as arguments.

In essence, the program automates the process of collecting loan application details for a predefined list of applicants and then uses the approve\_loan function to determine and report whether each applicant's loan is approved based on simple income and credit score rules.

TASK-3:

Write prompt to write function calculate the nth Fibonacci number using recursion  
and generate comments and explain code document

def fibonacci\_recursive(n):

    """

    Calculates the nth Fibonacci number using recursion.

    Args:

        n (int): The position of the Fibonacci number to calculate (non-negative integer).

    Returns:

        int: The nth Fibonacci number.

    """

    if n <= 1:

        return n

    else:

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

# Example usage:

n\_term = 10

fib\_number = fibonacci\_recursive(n\_term)

print(f"The {n\_term}th Fibonacci number is: {fib\_number}")

output:

The 10th Fibonacci number is: 55

Explanation:

Certainly! This code calculates the nth Fibonacci number using a recursive approach.

Here's a breakdown of the code:

1. **def fibonacci\_recursive(n):**: This line defines a function named fibonacci\_recursive that takes one argument, n.
2. **""" ... """**: This is a docstring that explains what the function does, its arguments, and what it returns.
3. **if n <= 1:**: This is the base case for the recursion. The first two Fibonacci numbers (at position 0 and 1) are defined as 0 and 1, respectively. If n is 0 or 1, the function simply returns n. This stops the recursion.
4. **else:**: If n is greater than 1, the code enters the else block.
5. **return fibonacci\_recursive(n - 1) + fibonacci\_recursive(n - 2)**: This is the recursive step. The function calls itself twice: once with n - 1 and once with n - 2. The result of these two recursive calls is added together and returned. This follows the definition of the Fibonacci sequence, where each number is the sum of the two preceding ones.
6. **n\_term = 10**: This line sets a variable n\_term to 10, indicating that we want to find the 10th Fibonacci number.
7. **fib\_number = fibonacci\_recursive(n\_term)**: This line calls the fibonacci\_recursive function with n\_term (which is 10) and stores the returned Fibonacci number in the fib\_number variable.
8. **print(f"The {n\_term}th Fibonacci number is: {fib\_number}")**: This line prints the result, showing the position (n\_term) and the calculated Fibonacci number (fib\_number).

In summary, the fibonacci\_recursive function calculates Fibonacci numbers by breaking down the problem into smaller subproblems (calculating the previous two Fibonacci numbers) until it reaches the base cases (0 and 1). The results of these subproblems are then combined to get the final result.

TASK-4:

Ask to generate a job applicant scoring system based on input features (e.g.,  
education, experience, gender, age). Analyze the scoring logic for bias or unfair  
weightings

# Step 1: Define applicant data

applicants = [

    {

        "name": "Alice",

        "age": 28,

        "gender": "Female",

        "education": "Master's Degree",

        "experience": 5

    },

    {

        "name": "Bob",

        "age": 35,

        "gender": "Male",

        "education": "Bachelor's Degree",

        "experience": 10

    },

    {

        "name": "Charlie",

        "age": 22,

        "gender": "Male",

        "education": "High School Diploma",

        "experience": 2

    },

    {

        "name": "David",

        "age": 40,

        "gender": "Male",

        "education": "PhD",

        "experience": 15

    },

    {

        "name": "Eve",

        "age": 31,

        "gender": "Female",

        "education": "Bachelor's Degree",

        "experience": 7

    }

]

display(applicants)

# Step 2: Define scoring function

def score\_applicant(applicant):

    """

    Calculates a score for a job applicant based on various criteria.

    Args:

        applicant (dict): A dictionary containing applicant details.

    Returns:

        int: The calculated score for the applicant.

    """

    score = 0

    # Score based on education

    education = applicant.get("education", "")

    if education == "PhD":

        score += 50

    elif education == "Master's Degree":

        score += 40

    elif education == "Bachelor's Degree":

        score += 30

    elif education == "High School Diploma":

        score += 20

    else:

        score += 10 # Default score for other education levels

    # Score based on experience (e.g., 5 points per year of experience)

    experience = applicant.get("experience", 0)

    score += experience \* 5

    # Score based on age (example: higher score for age between 25 and 40)

    age = applicant.get("age", 0)

    if 25 <= age <= 40:

        score += 10

    elif age > 40:

        score += 5

    # Score based on gender (example: a small bonus for gender diversity, adjust as needed)

    gender = applicant.get("gender", "")

    if gender == "Female":

        score += 5  # Example bonus

    # Note: Scoring based on name is not typically a valid hiring practice and is excluded here.

    # Adjust scoring logic and weights based on your specific requirements.

    return score

# Step 3 & 4: Score applicants and display results

print("\nApplicant Scores:")

for applicant in applicants:

    applicant\_score = score\_applicant(applicant)

    print(f"{applicant['name']}: {applicant\_score}")

output:

Applicant Scores:

Alice: 80

Bob: 90

Charlie: 30

David: 135

Eve: 80.

Explanation:

Certainly! The code visible on your screen consists of two parts: a function to calculate an applicant's score and a loop that applies this function to a list of applicants and displays the results.

Here's an explanation of each part:

1. **Cell d40034b9: The score\_applicant function**
   * **def score\_applicant(applicant):**: This line defines a function named score\_applicant that takes one argument, applicant. This argument is expected to be a dictionary containing the details of a single job applicant (like name, age, education, etc.).
   * **score = 0**: Initializes a variable score to 0. This variable will accumulate points based on the applicant's qualifications.
   * **Scoring based on criteria (Education, Experience, Age, Gender):** The code then checks the values for different keys in the applicant dictionary ("education", "experience", "age", "gender").
     + applicant.get("key", default\_value) is used to safely access dictionary values. If a key is not found, it returns the default\_value (e.g., "" for strings, 0 for numbers) instead of raising an error.
     + Based on the value of each criterion, points are added to the score. For example, a "PhD" in education adds 50 points, each year of experience adds 5 points, and a female applicant gets a bonus of 5 points (this is an example and can be adjusted).
   * **return score**: After checking all the criteria and adding the corresponding points, the function returns the final calculated score for the applicant.
2. **Cell 2ec8b640: Scoring applicants and displaying results**
   * **print("\nApplicant Scores:")**: This line prints a header before listing the scores.
   * **for applicant in applicants:**: This loop iterates through each item in the applicants list. We previously defined the applicants list (in cell afeb2a0a) as a list of dictionaries, where each dictionary represents an applicant.
   * **applicant\_score = score\_applicant(applicant)**: Inside the loop, for each applicant dictionary, it calls the score\_applicant function (defined in cell d40034b9), passing the current applicant's dictionary as an argument. The returned score is stored in the applicant\_score variable.
   * **print(f"{applicant['name']}: {applicant\_score}")**: This line prints the name of the current applicant (accessed using applicant['name']) followed by their calculated applicant\_score. The f-string is used for formatted output.

In summary, cell d40034b9 defines the rules for how to calculate a score for a single applicant based on their attributes. Cell 2ec8b640 then uses a loop to apply these rules to every applicant in the applicants list and neatly prints the score for each one

TASK-5:

def greet\_user(name,gender):

  if gender == "male":

    title = "mr."

  elif gender == "female":

    title = "mrs."

  elif gender == "neutral" or gender == "non-binary":

    title = "mx."

  else:

    title = ""

  return f"hello, {title}  {name}!  welcome"

print(greet\_user("john","male"))

print(greet\_user("priya","female"))

print(greet\_user("alex","neutral"))

print(greet\_user("sam","unknown"))

output:

hello, mr. john! welcome

hello, mrs. priya! welcome

hello, mx. alex! welcome

hello, sam! Welcome

Explanation:

Certainly! This code defines a Python function called greet\_user that generates a personalized greeting based on a user's name and gender.

Here's a breakdown of the code:

1. **def greet\_user(name, gender):**: This line defines a function named greet\_user that accepts two arguments: name (presumably the user's name as a string) and gender (presumably the user's gender as a string).
2. **if gender == "male":**: This is the first condition. If the value of the gender argument is exactly "male", the code inside this if block is executed.
3. **title = "mr."**: If the gender is "male", the variable title is assigned the value "mr.".
4. **elif gender == "female":**: This is an "else if" condition. If the previous if condition was false, this condition is checked. If the value of gender is "female", the code inside this elif block is executed.
5. **title = "mrs."**: If the gender is "female", the variable title is assigned the value "mrs.".
6. **elif gender == "neutral" or gender == "non-binary":**: This is another "else if" condition. If the previous conditions were false, this one is checked. If the value of gender is either "neutral" or "non-binary", the code inside this elif block is executed.
7. **title = "mx."**: If the gender is "neutral" or "non-binary", the variable title is assigned the value "mx.".
8. **else:**: This is the final "else" block. If none of the previous if or elif conditions were true (meaning the gender is something other than "male", "female", "neutral", or "non-binary"), the code inside this else block is executed.
9. **title = ""**: In this else case, the title variable is assigned an empty string.
10. **return f"hello, {title} {name}! welcome"**: This line constructs the final greeting string using an f-string. It combines the literal string "hello, ", the determined title, a space, the name, the literal string "! ", and the literal string "welcome". The resulting greeting string is then returned by the function.

After the function definition, there are several lines demonstrating how to call the greet\_user function with different names and genders and printing the returned greeting:

* print(greet\_user("john","male")) calls the function with "john" and "male", printing "hello, mr. john! welcome".
* print(greet\_user("priya","female")) calls the function with "priya" and "female", printing "hello, mrs. priya! welcome".
* print(greet\_user("alex","neutral")) calls the function with "alex" and "neutral", printing "hello, mx. alex! welcome".
* print(greet\_user("sam","unknown")) calls the function with "sam" and "unknown", printing "hello, sam! welcome" (since "unknown" doesn't match any of the specific gender conditions, the else block sets title to an empty string).

In essence, the greet\_user function provides a way to generate a polite and somewhat personalized greeting by adding a title based on the provided gender, falling back to no title if the gender is not recognized.