

Assignment-5.3

Sindhu Katakam

Batch-41

2303A52481

Task 1: Privacy and Data Security in AI-Generated Code

PROMPT (insecure):

generate a python code for login management . based on users choice user may login or create an account.

CODE1:

```
while True:
    print("\n--- Login Management System ---")
    print("1. Create Account")
    print("2. Login")
    print("3. Exit")

    choice = input("Enter your choice (1-3): ")

    if choice == '1':
        username = input("Enter new username: ")
        password = input("Enter new password: ")
        create_account(username, password)
    elif choice == '2':
        username = input("Enter username: ")
        password = input("Enter password: ")
        login(username, password)
    elif choice == '3':
        print("Exiting system. Goodbye!")
        break
    else:
        print("Invalid choice. Please enter 1, 2, or 3.")

accounts = {}

def create_account(username, password):
    if username in accounts:
        print("Account already exists. Please choose a different username.")
        return False
    else:
        accounts[username] = password
        print(f"Account for '{username}' created successfully!")
        return True

def login(username, password):
    if username not in accounts:
        print("Username not found.")
        return False
    elif accounts[username] != password:
        print("Incorrect password.")
        return False
    else:
        print(f"Welcome, {username}! You are logged in.")
        return True
```

Output:

```
***
--- Login Management System ---
1. Create Account
2. Login
3. Exit
Enter your choice (1-3): 2
Enter username: sindhuu
Enter password: sindhu21
Invalid credentials.

--- Login Management System ---
1. Create Account
2. Login
3. Exit
Enter your choice (1-3): 3
Exiting system. Goodbye!
```

Prompt secure: (code2)

generate a readable and efficient code for login system .based on user choice the user may login or register. the data has to be secured and encrypted . check whether the credentials are hardcoded and passwords are stored or compared in plain text. .improves the overall security . after registration store the data of user securely and it should not be displayed in the output.

CODE:

```
import sys
!{sys.executable} -m pip install bcrypt
import bcrypt
import getpass # Import the getpass module

accounts = {}

def create_account(username, password):
    if username in accounts:
        print("Account already exists. Please choose a different username.")
        return False
    else:
        hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
        accounts[username] = hashed_password
        print(f"Account for '{username}' created successfully!")
        return True

def login(username, password):
    if username not in accounts:
        print("Invalid credentials.")
        return False
    else:
        stored_hashed_password = accounts[username]
        if bcrypt.checkpw(password.encode('utf-8'), stored_hashed_password):
            print(f"Welcome, {username}! You are logged in.")
            return True
        else:
            print("Invalid credentials.") # Generic message for security
            return False

while True:
    print("\n--- Login Management System ---")
    print("1. Create Account")
    print("2. Login")
    print("3. Exit")

    choice = input("Enter your choice (1-3): ")

    if choice == '1':
        username = input("Enter new username: ")
        password = getpass.getpass("Enter new password: ") # Use getpass for secure input
        create_account(username, password)
    elif choice == '2':
        username = input("Enter username: ")
        password = getpass.getpass("Enter password: ") # Use getpass for secure input
        login(username, password)
    elif choice == '3':
        print("Exiting system. Goodbye!")
        break
    else:
        print("Invalid choice. Please enter 1, 2, or 3.")
```

Output:

```

*** Requirement already satisfied: bcrypt in /usr/local/lib/python3.12/dist-packages (5.0.0)

--- Login Management System ---
1. Create Account
2. Login
3. Exit
Enter your choice (1-3): 1
Enter new username: sindhukatakam
Enter new password: .....
Account for 'sindhukatakam' created successfully!

--- Login Management System ---
1. Create Account
2. Login
3. Exit
Enter your choice (1-3): 2
Enter username: sindhukatakam
Enter password: .....
Welcome, sindhukatakam! You are logged in.

--- Login Management System ---
1. Create Account
2. Login
3. Exit
Enter your choice (1-3): 3
Exiting system. Goodbye!

```

Explanation:

Code1 is less secure because it stores and compares passwords in plain text, shows specific error messages that can reveal valid usernames, and displays the password while typing. Code2 improves security by hashing passwords with bcrypt, using secure password comparison, and hiding typed passwords with getpass. Code2 also gives generic error messages, which makes guessing credentials harder. In terms of readability, Code1 is simpler and easier for beginners because it has fewer imports and no hashing logic. Code2 is slightly more complex due to extra steps but is still well-organized and worth using for real security.

Task 2: Bias Detection in AI-Generated Decision Systems

Prompt: (biased)

Generate a simple Python program for a loan approval system that includes biased logic based on applicant names and gender. The program should evaluate each applicant and print Loan approved if they meet the specified biased conditions, otherwise print Not eligible. Ensure the code is efficient.

CODE:

```

def predict_loan_approval_biased_simplified(applicant_name, gender):
    """
    Demonstrates a *biased* loan approval prediction function based on name and gender.
    WARNING: This function illustrates discriminatory practices and is for educational purposes ONLY.
    Implementing such biases in real-world financial systems is illegal and unethical.
    """
    is_approved = False
    biased_reasons = []

    # --- Name Bias (purely illustrative of discrimination) ---
    # In a fair system, an applicant's name should not influence loan decisions.
    if applicant_name.lower().startswith(('a', 'm', 's')):
        is_approved = True # Arbitrary bias: names starting with A, M, S are favored
        biased_reasons.append(f"Name '{applicant_name}' starts with a preferred letter.")
    else:
        biased_reasons.append(f"Name '{applicant_name}' does not start with a preferred letter.")

    # --- Gender Bias (purely illustrative of discrimination) ---
    # In a fair system, an applicant's gender should never be a factor in loan approval.
    if gender.lower() == 'female' and is_approved: # Arbitrary bias: further favors females if already partially approved
        biased_reasons.append(f"Gender '{gender}' is explicitly favored in this model.")
    elif gender.lower() == 'male':
        if is_approved: # If already approved by name, gender acts as a slight demerit
            biased_reasons.append(f"Gender '{gender}' is explicitly disfavored in this model.")
            is_approved = False # Overriding name-based approval for demonstration
        else:
            biased_reasons.append(f"Gender '{gender}' is explicitly disfavored in this model.")

    # Final decision based on the biased logic
    if is_approved:
        print(f"Loan approved for {applicant_name}.")
    else:
        print(f"Not eligible for {applicant_name}.")
    print("\tFactors Considered:", biased_reasons)

# --- Example Usage of the Biased System (for demonstration ONLY) ---
print("\n--- Simple Loan Approval System Examples (for demonstration ONLY) ---")

# Example 1: Favored name and favored gender
predict_loan_approval_biased_simplified("Alice", "Female")

# Example 2: Favored name but disfavored gender (will be declined due to gender override)
predict_loan_approval_biased_simplified("Mark", "Male")

# Example 3: Disfavored name and favored gender (will be declined due to name)
predict_loan_approval_biased_simplified("Zara", "Female")

# Example 4: Disfavored name and disfavored gender
predict_loan_approval_biased_simplified("Brendan", "Male")

```

OUTPUT:

```

--- Simple Loan Approval System Examples (for demonstration ONLY) ---
Loan approved for Alice.
  Factors Considered: ["Name 'Alice' starts with a preferred letter.", "Gender 'Female' is explicitly favored in this model."]
Not eligible for Mark.
  Factors Considered: ["Name 'Mark' starts with a preferred letter.", "Gender 'Male' is explicitly disfavored in this model."]
Not eligible for Zara.
  Factors Considered: ["Name 'Zara' does not start with a preferred letter."]
Not eligible for Brendan.
  Factors Considered: ["Name 'Brendan' does not start with a preferred letter.", "Gender 'Male' is explicitly disfavored in this model."]

```

Prompt: (Unbiased)

generate a simple python code for loan approval system. that vary applicant names and gender . Generate an output that prints Loan approved if the conditions satisfy, if not print Not eligible.

Code:

```
def approve_loan_status(applicant_name, age, gender, credit_score, annual_income):
    # Basic conditions for loan approval
    is_approved = True
    reasons = []

    if age < 18:
        is_approved = False
        reasons.append("Applicant is too young.")
    if credit_score < 600:
        is_approved = False
        reasons.append("Credit score is too low.")
    if annual_income < 30000:
        is_approved = False
        reasons.append("Annual income is insufficient.")

    # Gender is included for context but does not affect approval in this simple example.
    # In a real-world scenario, this would be handled carefully to avoid discrimination.
    if gender.lower() not in ['male', 'female', 'other']:
        reasons.append(f"Unrecognized gender: {gender}.")

    if is_approved:
        print(f"Loan approved for {applicant_name}.")
    else:
        print(f"Not eligible for {applicant_name}. Reasons: {', '.join(reasons) if reasons else 'Undisclosed criteria not met.'}")

# --- Example Usage ---
print("--- Simple Loan Approval System ---")

# Applicant 1: Approved
approve_loan_status("Alice Smith", 28, "Female", 720, 60000)

# Applicant 2: Not eligible (low credit score)
approve_loan_status("Bob Johnson", 35, "Male", 550, 45000)

# Applicant 3: Not eligible (low income)
approve_loan_status("Charlie Brown", 42, "Other", 680, 25000)

# Applicant 4: Not eligible (too young)
approve_loan_status("Diana Prince", 17, "Female", 700, 50000)

# Applicant 5: Approved with a high income
approve_loan_status("Eve Adams", 50, "Female", 750, 90000)
```

Output:

```
... --- Simple Loan Approval System Example ---
```

```
Applicant 1 Status: Approved  
Feedback: ['All basic criteria met.']
```

```
Applicant 2 Status: Declined  
Feedback: ['Credit score is too low (below 600).']
```

```
Applicant 3 Status: Approved  
Feedback: ['All basic criteria met.']
```

Explanation:

The biased code intentionally demonstrates discriminatory bias by making loan decisions based on irrelevant attributes such as applicant name and gender, which leads to unfair and unethical outcomes. It treats certain names and genders preferentially, showing how biased logic can create disparate treatment. In contrast, the unbiased code evaluates applicants using relevant financial criteria like age, credit score, and income, which are appropriate and legitimate factors in real loan assessments. Although gender and ethnicity are collected, they do not influence the decision, avoiding discriminatory impacts. Overall, the first code illustrates harmful bias, while the second reflects fairer and more responsible decision-making.

Task 3: Transparency and Explainability in AI-Generated Code (Recursive Binary Search)

Prompt:

generate a binary search python code using recursion. the list has to be sorted before searching for the target element.

Code:

```

def binary_search_recursive(arr, target, low, high):
    # Base case: target not found
    if low > high:
        return -1

    mid = (low + high) // 2

    # If target is at mid
    if arr[mid] == target:
        return mid
    # If target is smaller than mid, search in the left half
    elif arr[mid] > target:
        return binary_search_recursive(arr, target, low, mid - 1)
    # If target is larger than mid, search in the right half
    else:
        return binary_search_recursive(arr, target, mid + 1, high)

def find_element_with_binary_search(data_list, target_element):
    # Ensure the list is sorted for binary search to work correctly
    sorted_list = sorted(data_list)
    print(f"Original list: {data_list}")
    print(f"Sorted list: {sorted_list}")

    n = len(sorted_list)
    result_index = binary_search_recursive(sorted_list, target_element, 0, n - 1)

    if result_index != -1:
        print(f"Target element {target_element} found at index {result_index} in the sorted list.")
    else:

```

```

        print(f"Target element {target_element} not found in the list.")
    return result_index

# --- Dynamic Input Usage ---
print("\n--- Interactive Binary Search ---")
while True:
    try:
        list_input_str = input("Enter numbers for the list, separated by commas (e.g., 10, 20, 5, 30): ")
        my_dynamic_list = [int(x.strip()) for x in list_input_str.split(',')]

        target_input = int(input("Enter the target element to search for: "))

        find_element_with_binary_search(my_dynamic_list, target_input)

        another_search = input("\nPerform another search? (yes/no): ")
        if another_search.lower() != 'yes':
            break

    except ValueError:
        print("Invalid input. Please ensure you enter numbers separated by commas for the list and a single number for the target.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")

print("Exiting interactive binary search. Goodbye!")

```

Output:


```

...
--- Interactive Binary Search ---
Enter numbers for the list, separated by commas (e.g., 10, 20, 5, 30): 87,35,97,24,53,26
Enter the target element to search for: 5
Original list: [87, 35, 97, 24, 53, 26]
Sorted list: [24, 26, 35, 53, 87, 97]
Target element 5 not found in the list.

Perform another search? (yes/no): no
Exiting interactive binary search. Goodbye!

```

Prompt:

generate a simple python code for binary search using recursion.sort the list first and then search for the target element. mention clear inline comments that includes step by step explanation of recursive logic and code has to be structured , understandable for beginners and readable.

Code:

```

🎮 def binary_search_recursive(arr, target, low, high):
    """
    Performs a binary search on a sorted list recursively.

    Args:
        arr (list): The sorted list to search within.
        target: The element to search for.
        low (int): The starting index of the current search range.
        high (int): The ending index of the current search range.

    Returns:
        int: The index of the target element if found, otherwise -1.
    """

```

```

# Step 1: Base Case - If the search range is invalid (low > high),
#           it means the target is not in the list.
if low > high:
    return -1

# Step 2: Calculate the middle index of the current search range.
#           Using (low + high) // 2 prevents potential overflow for very large lists.
mid = (low + high) // 2

# Step 3: Compare the middle element with the target.
#           If they are equal, the target is found at 'mid' index.
if arr[mid] == target:
    return mid

# Step 4: If the middle element is greater than the target,
#           the target must be in the left half of the current range.
#           Recursively call the function for the left half.
elif arr[mid] > target:
    return binary_search_recursive(arr, target, low, mid - 1)

# Step 5: If the middle element is less than the target,
#           the target must be in the right half of the current range.
#           Recursively call the function for the right half.
else:
    return binary_search_recursive(arr, target, mid + 1, high)

```

```

def find_element_with_binary_search(data_list, target_element):
    """
    Prepares a list for binary search by sorting it, then performs the search.

    Args:
        data_list (list): The original unsorted list.
        target_element: The element to search for.

    Returns:
        int: The index of the target element in the sorted list if found, otherwise -1.
    """
    # Step 1: Binary search requires the list to be sorted.
    #           Create a sorted copy of the input list.
    sorted_list = sorted(data_list)
    print(f"Original list: {data_list}")
    print(f"Sorted list: {sorted_list}")

    # Step 2: Get the size of the sorted list.
    n = len(sorted_list)

    # Step 3: Call the recursive binary search function.
    #           Start with the full range of the sorted list (from index 0 to n-1).
    result_index = binary_search_recursive(sorted_list, target_element, 0, n - 1)

```

```

# Step 4: Interpret and print the result.
if result_index != -1:
    print(f"Target element {target_element} found at index {result_index} in the sorted list.")
else:
    print(f"Target element {target_element} not found in the list.")
return result_index

```

```
# --- Dynamic Input Usage ---
print("\n--- Interactive Binary Search ---")
while True:
    try:
        list_input_str = input("Enter numbers for the list, separated by commas (e.g., 10, 20, 5, 30): ")
        my_dynamic_list = [int(x.strip()) for x in list_input_str.split(',')]

        target_input = int(input("Enter the target element to search for: "))

        find_element_with_binary_search(my_dynamic_list, target_input)

        another_search = input("\nPerform another search? (yes/no): ")
        if another_search.lower() != 'yes':
            break

    except ValueError:
        print("Invalid input. Please ensure you enter numbers separated by commas for the list and a single number for the target.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")

print("Exiting interactive binary search. Goodbye!")
```

Output:

```
...
--- Interactive Binary Search ---
Enter numbers for the list, separated by commas (e.g., 10, 20, 5, 30): 24,56,78,25,96,15,37
Enter the target element to search for: 46
Original list: [24, 56, 78, 25, 96, 15, 37]
Sorted list: [15, 24, 25, 37, 56, 78, 96]
Target element 46 not found in the list.

Perform another search? (yes/no): no
Exiting interactive binary search. Goodbye!
```

Explanation:

Both versions of the binary search code correctly implement recursive binary search logic and ensure the input list is sorted before searching, which is essential for correctness. The first version focuses on functionality with minimal comments, while the second version is more detailed and documentation-rich, explaining each step for better readability and learning. Both codes include interactive input handling, type validation, and user prompts, demonstrating good user interaction flow. Error handling is present in both, preventing crashes due to invalid input formats. Overall, the second implementation is more structured and educational, while the first is simpler but functionally equivalent and efficient.

Task-4 : Ethical Evaluation of AI-Based Scoring Systems

Code: (biased)

```

def evaluate_applicant_biased(applicant_name, skills, experience_years, education_level, gender):
    """
    Evaluates a job applicant, introducing *explicit biases* based on name and gender.
    WARNING: This function demonstrates discriminatory practices and is for educational purposes ONLY.
    Implementing such biases in real-world hiring is illegal and unethical.
    """

    total_score = 0
    feedback = []

    # --- Score based on Skills ---
    skill_weights = {
        'Python': 3,
        'SQL': 3,
        'Machine Learning': 4,
        'Data Analysis': 3,
        'Cloud Computing': 2,
        'Project Management': 2
    }
    skill_score = 0
    for skill in skills:
        if skill in skill_weights:
            skill_score += skill_weights[skill]
            feedback.append(f"- Skill '{skill}' awarded {skill_weights[skill]} points.")
        else:
            feedback.append(f"- Skill '{skill}' is relevant but not specifically weighted in this system.")
    total_score += skill_score
    feedback.append(f"Total skill score: {skill_score}")

    # --- Score based on Experience ---
    experience_score = 0
    if experience_years >= 5:
        experience_score = 10
        feedback.append(f"- {experience_years} years of experience awarded 10 points (highly experienced).")
    elif experience_years >= 2:
        experience_score = 5
        feedback.append(f"- {experience_years} years of experience awarded 5 points (moderately experienced).")
    else:
        experience_score = 1
        feedback.append(f"- {experience_years} years of experience awarded 1 point (entry-level experience).")
    total_score += experience_score
    feedback.append(f"Total experience score: {experience_score}")

    # --- Score based on Education ---
    education_score = 0
    education_level_lower = education_level.lower()
    if 'phd' in education_level_lower:
        education_score = 8

```

```

elif 'masters' in education_level_lower:
    education_score = 6
    feedback.append(f"- Education level '{education_level}' awarded 6 points (Masters).")
elif 'bachelors' in education_level_lower:
    education_score = 4
    feedback.append(f"- Education level '{education_level}' awarded 4 points (Bachelors).")
else:
    education_score = 2
    feedback.append(f"- Education level '{education_level}' awarded 2 points (Other/High School).")
total_score += education_score
feedback.append(f"Total education score: {education_score}")

# --- INTRODUCING EXPLICIT BIAS (FOR DEMONSTRATION ONLY) ---
# This logic is purely illustrative of discriminatory practices.
bias_feedback = []

# Name Bias: Penalize names starting with 'Z' or 'Q'
if applicant_name.lower().startswith(('z', 'q')):
    total_score -= 5
    bias_feedback.append(f"- Applicant name '{applicant_name}' starts with a disfavored letter (BIAS INTRODUCED).")

# Gender Bias: Favor 'Female' applicants, penalize 'Male' applicants
if gender.lower() == 'female':
    total_score += 3

    bias_feedback.append(f"- Gender '{gender}' received bonus points (BIAS INTRODUCED).")
elif gender.lower() == 'male':
    total_score -= 2
    bias_feedback.append(f"- Gender '{gender}' received penalty points (BIAS INTRODUCED).")

if bias_feedback:
    feedback.append("--- Bias Adjustments (for demonstration ONLY) ---")
    feedback.extend(bias_feedback)

# --- Determine Suitability ---
suitability_message = ""
if total_score >= 15:
    suitability_message = "Highly Recommended"
elif total_score >= 10:
    suitability_message = "Recommended"
elif total_score >= 5:
    suitability_message = "Considerable"
else:
    suitability_message = "Not Recommended"

return total_score, suitability_message, feedback

# --- Example Usage of the Biased System ---
print("--- Job Applicant Scoring System with EXPLICIT BIAS ---")

```

```

# Applicant 1: Favored name, favored gender, strong qualifications
name1 = "Alice Smith"
skills1 = ['Python', 'SQL', 'Machine Learning', 'Data Analysis']
experience1 = 6
education1 = "Masters in Data Science"
gender1 = "Female"
score1, suitability1, feedback1 = evaluate_applicant_biased(name1, skills1, experience1, education1, gender1)
print(f"\nApplicant: {name1}")
print(f"Total Score: {score1}")
print(f"Suitability: {suitability1}")
print("Detailed Feedback:")
for item in feedback1:
    print(item)
# Applicant 2: Good qualifications, disfavored gender
name2 = "Bob Johnson"
skills2 = ['Python', 'SQL']
experience2 = 5 # Changed to 5 for higher base score
education2 = "Bachelors in Computer Science"
gender2 = "Male"
score2, suitability2, feedback2 = evaluate_applicant_biased(name2, skills2, experience2, education2, gender2)
print(f"\nApplicant: {name2}")
print(f"Total Score: {score2}")
print(f"Suitability: {suitability2}")
print("Detailed Feedback:")

```

Output:

--- Job Applicant Scoring System with EXPLICIT BIAS ---

Applicant: Alice Smith
 Total Score: 32
 Suitability: Highly Recommended
 Detailed Feedback:
 - Skill 'Python' awarded 3 points.
 - Skill 'SQL' awarded 3 points.
 - Skill 'Machine Learning' awarded 4 points.
 - Skill 'Data Analysis' awarded 3 points.
 Total skill score: 13
 - 6 years of experience awarded 10 points (highly experienced).
 Total experience score: 10
 - Education level 'Masters in Data Science' awarded 6 points (Masters).
 Total education score: 6
 --- Bias Adjustments (for demonstration ONLY) ---
 - Gender 'Female' received bonus points (BIAS INTRODUCED).

Applicant: Bob Johnson
 Total Score: 18
 Suitability: Highly Recommended
 Detailed Feedback:
 - Skill 'Python' awarded 3 points.
 - Skill 'SQL' awarded 3 points.
 Total skill score: 6
 - 5 years of experience awarded 10 points (highly experienced).
 - 5 years of experience awarded 10 points (highly experienced).
 Total experience score: 10
 - Education level 'Bachelors in Computer Science' awarded 4 points (Bachelors).
 Total education score: 4
 --- Bias Adjustments (for demonstration ONLY) ---
 - Gender 'Male' received penalty points (BIAS INTRODUCED).

Prompt: (unbiased)

generate a python code for job applicant scoring system based on their skills experience and education check whether there is any bias based on gender ,name or unrelated features influence scoring, Whether the logic is fair and objective.

Code:

```
def evaluate_applicant(applicant_name, skills, experience_years, education_level):
    """
    Evaluates a job applicant based on their skills, experience, and education.
    Assigns a score and determines suitability.

    Args:
        applicant_name (str): The name of the applicant.
        skills (list): A list of skills the applicant possesses (e.g., ['Python', 'SQL', 'ML']).
        experience_years (int): Years of professional experience.
        education_level (str): The highest education level (e.g., 'High School', 'Bachelors', 'Masters', 'PhD').

    Returns:
        tuple: A tuple containing (total_score, suitability_message, detailed_feedback).
    """
    total_score = 0
    feedback = []

    # --- Score based on Skills ---
    # Define ideal skills and their weights
    skill_weights = {
        'Python': 3,
        'SQL': 3,
        'Machine Learning': 4,
        'Data Analysis': 3,
        'Cloud Computing': 2,
        'Project Management': 2
    }
    skill_score = 0
    for skill in skills:
        if skill in skill_weights:
            skill_score += skill_weights[skill]
            feedback.append(f"- Skill '{skill}' awarded {skill_weights[skill]} points.")
        else:
            feedback.append(f"- Skill '{skill}' is relevant but not specifically weighted in this system.")
    total_score += skill_score
    feedback.append(f"Total skill score: {skill_score}")

    # --- Score based on Experience ---
    experience_score = 0
    if experience_years >= 5:
        experience_score = 10
        feedback.append(f"- {experience_years} years of experience awarded 10 points (highly experienced).")
    elif experience_years >= 2:
        experience_score = 5
        feedback.append(f"- {experience_years} years of experience awarded 5 points (moderately experienced).")
    else:
        experience_score = 1
```

Toggle Gemini

```

        experience_score = 1
        feedback.append(f"- {experience_years} years of experience awarded 1 point (entry-level experience).")
    total_score += experience_score
    feedback.append(f"Total experience score: {experience_score}")

# --- Score based on Education ---
education_score = 0
education_level_lower = education_level.lower()
if 'phd' in education_level_lower:
    education_score = 8
    feedback.append(f"- Education level '{education_level}' awarded 8 points (PhD).")
elif 'masters' in education_level_lower:
    education_score = 6
    feedback.append(f"- Education level '{education_level}' awarded 6 points (Masters).")
elif 'bachelors' in education_level_lower:
    education_score = 4
    feedback.append(f"- Education level '{education_level}' awarded 4 points (Bachelors).")
else:
    education_score = 2
    feedback.append(f"- Education level '{education_level}' awarded 2 points (Other/High School).")
total_score += education_score
feedback.append(f"Total education score: {education_score}")

# --- Determine Suitability ---

```



```

# --- Determine Suitability ---
suitability_message = ""
if total_score >= 15:
    suitability_message = "Highly Recommended"
elif total_score >= 10:
    suitability_message = "Recommended"
elif total_score >= 5:
    suitability_message = "Considerable"
else:
    suitability_message = "Not Recommended"

return total_score, suitability_message, feedback

# --- Example Usage ---
print("\n--- Job Applicant Scoring System ---")
# Applicant 1: Strong candidate
name1 = "Alice Smith"
skills1 = ['Python', 'SQL', 'Machine Learning', 'Data Analysis']
experience1 = 6
education1 = "Masters in Data Science"
score1, suitability1, feedback1 = evaluate_applicant(name1, skills1, experience1, education1)
print(f"\nApplicant: {name1}")
print(f"Total Score: {score1}")
print(f"Suitability: {suitability1}")
print("Detailed Feedback:")

```



```

▶ for item in feedback1:
    print(item)
# Applicant 2: Entry-level candidate
name2 = "Bob Johnson"
skills2 = ['Python', 'SQL']
experience2 = 1
education2 = "Bachelors in Computer Science"
score2, suitability2, feedback2 = evaluate_applicant(name2, skills2, experience2, education2)
print(f"\nApplicant: {name2}")
print(f"Total Score: {score2}")
print(f"Suitability: {suitability2}")
print("Detailed Feedback:")
for item in feedback2:
    print(item)
# Applicant 3: Candidate with some experience but fewer relevant skills
name3 = "Carol White"
skills3 = ['Project Management', 'Cloud Computing']
experience3 = 3
education3 = "High School Diploma"
score3, suitability3, feedback3 = evaluate_applicant(name3, skills3, experience3, education3)
print(f"\nApplicant: {name3}")
print(f"Total Score: {score3}")
print(f"Suitability: {suitability3}")
print("Detailed Feedback:")

```

Output:

```

--- Job Applicant Scoring System ---

*** Applicant: Alice Smith
Total Score: 29
Suitability: Highly Recommended
Detailed Feedback:
- Skill 'Python' awarded 3 points.
- Skill 'SQL' awarded 3 points.
- Skill 'Machine Learning' awarded 4 points.
- Skill 'Data Analysis' awarded 3 points.
Total skill score: 13
- 6 years of experience awarded 10 points (highly experienced).
Total experience score: 10
- Education level 'Masters in Data Science' awarded 6 points (Masters).
Total education score: 6

Applicant: Bob Johnson
Total Score: 11
Suitability: Recommended
Detailed Feedback:
- Skill 'Python' awarded 3 points.
- Skill 'SQL' awarded 3 points.
Total skill score: 6
- 1 years of experience awarded 1 point (entry-level experience).
Total experience score: 1
- Education level 'Bachelors in Computer Science' awarded 4 points (Bachelors).
Total education score: 4

```

Explanation:

The biased code unfairly changes scores based on name and gender, even though these have nothing to do with job ability. This creates discrimination and unequal treatment. The unbiased code only uses skills, experience, and education, which are relevant to hiring. This makes it more fair and objective. So, the biased code is unethical, while the unbiased code is suitable for real evaluations.

Task 5: Inclusiveness and Ethical Variable Design

Prompt:

Write a simple and realistic Python program that processes employee information including name, gender, and salary. The program should use the gender variable in its logic and display a final output based on the processed details.

Code:

```
def process_employee_data(employee_name, gender, salary):  
    """  
    Processes employee information, using gender for statistical grouping.  
    This function demonstrates a realistic and ethical use of gender data for reporting.  
  
    Args:  
        employee_name (str): The name of the employee.  
        gender (str): The gender of the employee (e.g., 'Male', 'Female', 'Other').  
        salary (float): The employee's annual salary.  
  
    Returns:  
        dict: A dictionary containing the processed employee details.  
    """  
    processed_data = {  
        'Name': employee_name,  
        'Gender': gender,  
        'Salary': salary  
    }  
    return processed_data  
  
def generate_gender_report(all_employee_data):  
    """  
    Generates a summary report based on gender from processed employee data.  
    """
```

```

gender_counts = {}
gender_salaries = {}

for employee in all_employee_data:
    gender = employee['Gender'].capitalize() # Normalize gender string
    salary = employee['Salary']

    gender_counts[gender] = gender_counts.get(gender, 0) + 1
    gender_salaries[gender] = gender_salaries.get(gender, 0) + salary

print("\n--- Employee Gender Report ---")
for gender, count in gender_counts.items():
    total_salary = gender_salaries[gender]
    average_salary = total_salary / count if count > 0 else 0
    print(f"Gender: {gender}")
    print(f"  Number of Employees: {count}")
    print(f"  Total Salary: ${total_salary:,.2f}")
    print(f"  Average Salary: ${average_salary:,.2f}")
print("-----")

# --- Example Usage ---
print("--- Employee Information Processing ---")

employees = []

# Process individual employee data
employees.append(process_employee_data("Alice Smith", "Female", 75000.00))
employees.append(process_employee_data("Bob Johnson", "Male", 80000.00))
employees.append(process_employee_data("Charlie Davis", "Other", 68000.00))
employees.append(process_employee_data("Diana Prince", "Female", 90000.00))
employees.append(process_employee_data("Eve Adams", "Female", 72000.00))
employees.append(process_employee_data("Frank White", "Male", 92000.00))

# Display individual processed data (for verification)
print("\nProcessed Individual Employee Data:")
for emp in employees:
    print(f"  Name: {emp['Name']}, Gender: {emp['Gender']}, Salary: ${emp['Salary']:,.2f}")

# Generate a report using the gender variable
generate_gender_report(employees)

... --- Employee Information Processing ---

Processed Individual Employee Data:
  Name: Alice Smith, Gender: Female, Salary: $75,000.00
  Name: Bob Johnson, Gender: Male, Salary: $80,000.00
  Name: Charlie Davis, Gender: Other, Salary: $68,000.00
  Name: Diana Prince, Gender: Female, Salary: $90,000.00
  Name: Eve Adams, Gender: Female, Salary: $72,000.00
  Name: Frank White, Gender: Male, Salary: $92,000.00

```

```

--- Employee Gender Report ---
Gender: Female
  Number of Employees: 3
  Total Salary: $237,000.00
  Average Salary: $79,000.00
Gender: Male
  Number of Employees: 2
  Total Salary: $172,000.00
  Average Salary: $86,000.00
Gender: Other
  Number of Employees: 1
  Total Salary: $68,000.00
  Average Salary: $68,000.00
-----

```

Prompt:

Write a Python program that works with employee details. Use simple and gender-neutral variable names. Do not use gender or identity in the program decisions unless you really need to. The program should make decisions only based on job things like role, performance, or experience.

```

def evaluate_worker(worker_id, role, experience_years, performance_rating):
    """
    Evaluates a worker based on job-related factors: role, experience, and performance.
    Assigns an evaluation score and determines a recommendation.

    Args:
        worker_id (str): A unique identifier for the worker (gender-neutral).
        role (str): The worker's job role (e.g., 'Junior', 'Mid-level', 'Senior', 'Lead').
        experience_years (int): Years of professional experience in their field.
        performance_rating (int): A rating of 1 to 5 (1=Needs Improvement, 5=Outstanding).

    Returns:
        tuple: A tuple containing (evaluation_score, recommendation_message, detailed_feedback).
    """
    evaluation_score = 0
    feedback = []

    # --- Score based on Role ---
    role_lower = role.lower()
    if 'lead' in role_lower or 'senior' in role_lower:
        evaluation_score += 8
        feedback.append(f"- Role '{role}' awarded 8 points (Senior/Lead). ")
    elif 'mid-level' in role_lower or 'specialist' in role_lower:
        evaluation_score += 5

    feedback.append(f"- Role '{role}' awarded 5 points (Mid-level). ")
else: # e.g., 'Junior', 'Associate'
    evaluation_score += 2
    feedback.append(f"- Role '{role}' awarded 2 points (Entry-level). ")

# --- Score based on Experience ---

```

```

if experience_years >= 7:
    evaluation_score += 10
    feedback.append(f"- {experience_years} years of experience awarded 10 points (Highly Experienced). ")
elif experience_years >= 3:
    evaluation_score += 6
    feedback.append(f"- {experience_years} years of experience awarded 6 points (Experienced). ")
else:
    evaluation_score += 3
    feedback.append(f"- {experience_years} years of experience awarded 3 points (Developing Experience). ")

# --- Score based on Performance Rating ---
if performance_rating == 5:
    evaluation_score += 7
    feedback.append(f"- Performance rating of {performance_rating} awarded 7 points (Outstanding). ")
elif performance_rating == 4:
    evaluation_score += 5
    feedback.append(f"- Performance rating of {performance_rating} awarded 5 points (Exceeds Expectations). ")
elif performance_rating == 3:
    evaluation_score += 3
    feedback.append(f"- Performance rating of {performance_rating} awarded 3 points (Meets Expectations). ")
else: # 1 or 2
    evaluation_score += 1
    feedback.append(f"- Performance rating of {performance_rating} awarded 1 point (Needs Improvement). ")

```

```

# --- Determine Recommendation ---
recommendation_message = ""
if evaluation_score >= 20:
    recommendation_message = "Highly Recommended for Promotion/Bonus"
elif evaluation_score >= 15:
    recommendation_message = "Eligible for Promotion Consideration"
elif evaluation_score >= 10:
    recommendation_message = "Meets Expectations - Good Contributor"
else:
    recommendation_message = "Requires Development and Review"

return evaluation_score, recommendation_message, feedback

# --- Example Usage (using gender-neutral worker_id) ---
print("--- Worker Evaluation System ---")

# Worker 1: High performer, experienced Senior
worker_id_1 = "WK001"
score1, recommendation1, feedback1 = evaluate_worker(worker_id_1, "Senior Engineer", 8, 5)
print(f"\nWorker ID: {worker_id_1}")
print(f"Total Score: {score1}")
print(f"Recommendation: {recommendation1}")
print("Detailed Feedback:")
for item in feedback1:
    print(item)

# Worker 2: Mid-level, good performance, moderate experience
worker_id_2 = "WK002"
score2, recommendation2, feedback2 = evaluate_worker(worker_id_2, "Mid-level Developer", 4, 4)
print(f"\nWorker ID: {worker_id_2}")
print(f"Total Score: {score2}")
print(f"Recommendation: {recommendation2}")
print("Detailed Feedback:")
for item in feedback2:
    print(item)

# Worker 3: Junior, new to role, average performance
worker_id_3 = "WK003"
score3, recommendation3, feedback3 = evaluate_worker(worker_id_3, "Junior Analyst", 1, 3)
print(f"\nWorker ID: {worker_id_3}")
print(f"Total Score: {score3}")
print(f"Recommendation: {recommendation3}")
print("Detailed Feedback:")
for item in feedback3:
    print(item)

```

```

*** --- Worker Evaluation System ---

```

```

Worker ID: WK001
Total Score: 25
Recommendation: Highly Recommended for Promotion/Bonus
Detailed Feedback:
- Role 'Senior Engineer' awarded 8 points (Senior/Lead).
- 8 years of experience awarded 10 points (Highly Experienced).
- Performance rating of 5 awarded 7 points (Outstanding).

```

Output:

```
Worker ID: WK002
Total Score: 16
Recommendation: Eligible for Promotion Consideration
Detailed Feedback:
- Role 'Mid-level Developer' awarded 5 points (Mid-level).
- 4 years of experience awarded 6 points (Experienced).
- Performance rating of 4 awarded 5 points (Exceeds Expectations).

Worker ID: WK003
Total Score: 8
Recommendation: Requires Development and Review
Detailed Feedback:
- Role 'Junior Analyst' awarded 2 points (Entry-level).
- 1 years of experience awarded 3 points (Developing Experience).
- Performance rating of 3 awarded 3 points (Meets Expectations).
```

Explanation:

The first code uses gender only for reporting and grouping salaries, not for deciding how good an employee is. It collects names, gender, and salary, then shows how many employees are in each gender group and their average salary. The second code does not use gender at all and instead focuses on fair job factors like role, experience, and performance. It gives each worker a score and recommendation based only on work-related things. So, the first code uses gender just for statistics, while the second code avoids gender and makes decisions based only on job skills and performance.