AI Assisted Coding Lab 5.3

Veekshitha Adharasani

2303A52175

Batch – 41

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

Task 1: Insecure Code Prompt

Generate a simple python code for a simple login system where the code identifies the user given password and username is correct credentials. If it is correct print Login Successful, if not print Invalid credentials.

CODE:

```
username = 'admin'
password = 'password123'

input_username = input('Enter username: ')
input_password = input('Enter password: ')

if input_username == username and input_password == password:
    print('Login Successful')
else:
    print('Invalid credentials')
```

OUTPUT:

```
Enter username: vicky
Enter password: vicky@14
Invalid credentials
```

Task 2: Secure Prompt

Generate a python code for login system for privacy and data security scenario. Keep choice that user selects wether the user wants to register or login. Encrypt the data and password. Check whether the credentials are hardcoded. If the user registered then store the passwords and details and check whether the passwords are stored or compared in plain text. Improve the overall security. Generate an efficient and readable code.

CODE:

```python
import hashlib
import os

# 1. Initialize user data storage
users = {}
print('Initialized an empty dictionary named users.')

# 2. Implement secure password hashing function
def hash_password(password):
    """
    Generates a unique salt and hashes the password using SHA256.
    Returns the hashed password and the salt.
    """
    salt = os.urandom(16)  # Generate a random 16-byte salt
    hashed_password = hashlib.sha256(salt + password.encode('utf-8')).hexdigest()
    return hashed_password, salt

# 3. Implement password verification function
def verify_password(password, salt, stored_hashed_password):
    """
    Verifies a plain-text password against a stored hashed password and salt.
    Returns True if the password matches, False otherwise.
    """
    hashed_input_password = hashlib.sha256(salt + password.encode('utf-8')).hexdigest()
    return hashed_input_password == stored_hashed_password

# 4. Implement user registration logic
def register_user():
    """
    username = input('Enter desired username: ')
    password = input('Enter desired password: ')

    if username in users:
        print(f'Username "{username}" is already taken. Please choose another one.')
        return

    hashed_password, salt = hash_password(password)
    users[username] = {'hashed_password': hashed_password, 'salt': salt}

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

# 5. Implement user login logic
def login_user():
    """
    Allows a user to log in by prompting for a username and password,
    and verifying credentials using the stored salt and hashed password.
    """
    username = input('Enter username: ')
    password = input('Enter password: ')

    if username not in users:
        print('Username not found.')
        return

    user_data = users[username]
    stored_hashed_password = user_data['hashed_password']
    salt = user_data['salt']

    if verify_password(password, salt, stored_hashed_password):
        print(f'Login successful! Welcome, {username}.')
    else:
        print('Invalid password.')

# Main application loop
print('\n--- Welcome to the User Authentication System ---')
while True:
    choice = input('\nEnter choice (register/login/exit): ').lower()

    if choice == 'register':
        register_user()
    elif choice == 'login':
        login_user()
    elif choice == 'exit':
        print('Exiting the application. Goodbye!')
        break
    else:
        print('Invalid choice. Please enter "register", "login", or "exit".')
```

OUTPUT:

```
Initialized an empty dictionary named users.

--- Welcome to the User Authentication System ---

Enter choice (register/login/exit): register
Enter desired username: vicky
Enter desired password: 12345
User "vicky" registered successfully!

Enter choice (register/login/exit): login
Enter username: vicky
Enter password: 12345
Login successful! Welcome, vicky.

Enter choice (register/login/exit): exit
Exiting the application. Goodbye!
```

JUSTIFICATION:

The original code was insecure because it used hardcoded credentials and compared passwords in plain text. The revised version removes hardcoded data and allows users to register dynamically. The structure separates registration and login logic, improving clarity and maintainability. Overall, the updated design follows basic security best practices for handling user credentials.

Question 2: Bias Detection in AI-Generated Decision Systems

Task 1: Biased Prompt

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

CODE:

```python
import random

# Define applicant names and genders
applicant_names = ['Alice', 'Bob', 'Charlie', 'Diana', 'Eve', 'Frank', 'Grace', 'Henry']
applicant_genders = ['Male', 'Female']

# Define base loan approval conditions
MIN_AGE = 18
MAX_AGE = 65
MIN_INCOME = 30000
BASE_MIN_CREDIT_SCORE = 650

# Introduce bias: Female applicants need a higher credit score
BIAS_CREDIT_SCORE_INCREASE_FOR_FEMALE = 50 # Female applicants need 50 points higher credit score

print("--- Loan Application System ---")

# Simulate loan applications for a few random applicants
for _ in range(5): # Process 5 random applicants
    applicant = {
        'name': random.choice(applicant_names),
        'gender': random.choice(applicant_genders),
        'age': random.randint(18, 70), # Age between 18 and 70
        'income': random.randint(20000, 80000), # Income between 20k and 80k
        'credit_score': random.randint(500, 850) # Credit score between 500 and 850
    }

    # Apply gender-based credit score requirement
    min_credit_score_for_applicant = BASE_MIN_CREDIT_SCORE
```

```python
    applicant = {
        'name': random.choice(applicant_names),
        'gender': random.choice(applicant_genders),
        'age': random.randint(18, 70), # Age between 18 and 70
        'income': random.randint(20000, 80000), # Income between 20k and 80k
        'credit_score': random.randint(500, 850) # Credit score between 500 and 850
    }

    # Apply gender-based credit score requirement
    min_credit_score_for_applicant = BASE_MIN_CREDIT_SCORE
    if applicant['gender'] == 'Female':
        min_credit_score_for_applicant += BIAS_CREDIT_SCORE_INCREASE_FOR_FEMALE

    print(f"\nProcessing application for: {applicant['name']} ({applicant['gender']})")
    print(f"  Age: {applicant['age']}, Income: ${applicant['income']}, Credit Score: {applicant['credit_score']}")
    print(f"  Required Credit Score (based on gender): {min_credit_score_for_applicant}")

    # Check loan approval conditions with bias
    if (
        applicant['age'] >= MIN_AGE and
        applicant['age'] <= MAX_AGE and
        applicant['income'] >= MIN_INCOME and
        applicant['credit_score'] >= min_credit_score_for_applicant
    ):
        print('  Loan approved!')
    else:
        print('  Not eligible.')
```

OUTPUT:

```
--- Biased Loan Application System ---

Processing application for: Grace (Female)
  Age: 35, Income: $48445, Credit Score: 523
  Required Credit Score (based on gender): 700
  Not eligible.

Processing application for: Eve (Female)
  Age: 46, Income: $75212, Credit Score: 815
  Required Credit Score (based on gender): 700
  Loan approved!

Processing application for: Henry (Male)
  Age: 23, Income: $41554, Credit Score: 643
  Required Credit Score (based on gender): 650
  Not eligible.

Processing application for: Charlie (Male)
  Age: 29, Income: $74984, Credit Score: 505
  Required Credit Score (based on gender): 650
  Not eligible.

Processing application for: Alice (Male)
  Age: 29, Income: $27853, Credit Score: 543
  Required Credit Score (based on gender): 650
  Not eligible.
```

Task 2: Un biased Prompt

Generate a python code for loan approval system. Vary applicant names and genders. Generate an output that prints Loan approved if the conditions satisfy, if not print Not eligible. Give an improvised code that is not biased. Generate a efficient, improvised and readable code.

CODE:

```python
import random

# --- Parameters for Applicant Data Generation (Unbiased) ---
applicant_names = ['Alice', 'Bob', 'Charlie', 'Diana', 'Eve', 'Frank', 'Grace', 'Henry']
applicant_genders = ['Male', 'Female']

# General ranges for all applicants (no gender-specific penalties)
MIN_AGE_GEN = 18
MAX_AGE_GEN = 70
MIN_INCOME_GEN = 20000
MAX_INCOME_GEN = 80000
MIN_CREDIT_SCORE_GEN = 500
MAX_CREDIT_SCORE_GEN = 850

# --- Neutral Loan Approval Conditions ---
MIN_AGE_APPROVAL = 18
MAX_AGE_APPROVAL = 65
MIN_INCOME_APPROVAL = 30000
MIN_CREDIT_SCORE_APPROVAL = 650

# --- Loan Approval Function (Unbiased Logic) ---
def is_loan_approved(applicant):
    """
    Determines if a loan applicant is approved based on neutral criteria.
    """
    if (
        applicant['age'] >= MIN_AGE_APPROVAL and
        applicant['age'] <= MAX_AGE_APPROVAL and
        applicant['income'] >= MIN_INCOME_APPROVAL and
```

```python
        else:
            return False

# --- Main Application Logic ---
unbiased_applicants = []
NUM_APPLICANTS_TO_SIMULATE = 10 # Simulate a few applicants for demonstration

print("--- Unbiased Loan Application System ---")

# Generate unbiased applicant data and process applications
for _ in range(NUM_APPLICANTS_TO_SIMULATE):
    applicant = {
        'name': random.choice(applicant_names),
        'gender': random.choice(applicant_genders),
        'age': random.randint(MIN_AGE_GEN, MAX_AGE_GEN),
        'income': random.randint(MIN_INCOME_GEN, MAX_INCOME_GEN),
        'credit_score': random.randint(MIN_CREDIT_SCORE_GEN, MAX_CREDIT_SCORE_GEN)
    }
    unbiased_applicants.append(applicant)

    print(f"\nProcessing application for: {applicant['name']} ({applicant['gender']})")
    print(f"  Age: {applicant['age']}, Income: ${applicant['income']}, Credit Score: {applicant['credit_score']}")

    if is_loan_approved(applicant):
        print('  Loan approved!')
    else:
        print('  Not eligible.')

print(f"\nProcessed {len(unbiased_applicants)} unbiased loan applications.")
```

OUTPUT:

```
--- Unbiased Loan Application System ---

Processing application for: Eve (Male)
  Age: 66, Income: $43251, Credit Score: 639
  Not eligible.

Processing application for: Bob (Male)
  Age: 53, Income: $64315, Credit Score: 558
  Not eligible.

Processing application for: Charlie (Male)
  Age: 27, Income: $30046, Credit Score: 526
  Not eligible.

Processing application for: Bob (Male)
  Age: 30, Income: $62242, Credit Score: 617
  Not eligible.

Processing application for: Bob (Male)
  Age: 66, Income: $22707, Credit Score: 824
  Not eligible.

Processing application for: Grace (Female)
  Age: 25, Income: $71553, Credit Score: 778
  Loan approved!

Processing application for: Alice (Male)
  Age: 69, Income: $56335, Credit Score: 704
  Not eligible.
```

```
Processing application for: Alice (Male)
  Age: 69, Income: $56335, Credit Score: 704
  Not eligible.

Processing application for: Frank (Female)
  Age: 51, Income: $57467, Credit Score: 565
  Not eligible.

Processing application for: Eve (Male)
  Age: 24, Income: $21682, Credit Score: 575
  Not eligible.

Processing application for: Alice (Female)
  Age: 35, Income: $50964, Credit Score: 763
  Loan approved!

Processed 10 unbiased loan applications.
```

JUSTIFICATION:

The biased version unfairly changes eligibility by increasing credit score requirements based on gender. Such logic introduces discrimination and violates fairness principles in system design. The improved version removes gender-based conditions and uses only neutral factors like age, income, and credit score. This ensures that all applicants are evaluated equally under the same criteria. The unbiased approach improves ethical quality and real-world applicability of the system.

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

Task 1: Non-Transparency Prompt

Generate a simple python code for implementation of Binary search using recursion. A list and element is given by the user. Then sort the user given list, check for the user given element in the sorted list and return the index of the searched element.

CODE:

```
def binary_search_recursive(arr, low, high, x):
    # Base case: Element is not present in the array
    if high >= low:
        mid = low + (high - low) // 2

        # If element is at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it can only be present in left subarray
        elif arr[mid] > x:
            return binary_search_recursive(arr, low, mid - 1, x)

        # Else the element can only be present in right subarray
        else:
            return binary_search_recursive(arr, mid + 1, high, x)

    else:
        # Element is not present in the array
        return -1

# --- User Input ---

# Get list of numbers from user
list_input = input("Enter a list of numbers separated by spaces: ")
user_list = [int(num) for num in list_input.split()]

# Get element to search from user
```

```
        # Element is not present in the array
        return -1

# --- User Input ---

# Get list of numbers from user
list_input = input("Enter a list of numbers separated by spaces: ")
user_list = [int(num) for num in list_input.split()]

# Get element to search from user
search_element = int(input("Enter the element to search for: "))

# --- Sort the list first ---
user_list.sort()
print(f"Sorted list: {user_list}")

# --- Perform binary search ---
result = binary_search_recursive(user_list, 0, len(user_list) - 1, search_element)

# --- Display result ---
if result != -1:
    print(f"Element {search_element} is present at index {result} in the sorted list.")
else:
    print(f"Element {search_element} is not present in the list.")
```

OUTPUT:

```
Enter a list of numbers separated by spaces: 6 7 2 9 1
Enter the element to search for: 1
Sorted list: [1, 2, 6, 7, 9]
Element 1 is present at index 0 in the sorted list.
```

Task 2: Transparency Prompt

Generate a simple python code for implementation of Binary search using recursion. A list and element is given by the user. Then sort the user given list, check for the user given element in the sorted list and return the index of the searched element. Include clear inline comments and step by step explanation of recursive logic. The code and output should be efficient, well-structured and readable.

CODE:

```python
def binary_search_recursive(arr, low, high, x):
    # Step 1: Base Case - If the search range is invalid (high < low), the element is not in the array.
    if high >= low:
        # Step 2: Calculate the middle index of the current search segment.
        # Using low + (high - low) // 2 prevents potential integer overflow for very large low and high values.
        mid = low + (high - low) // 2

        # Step 3: Compare the middle element with the target element x.
        # If the element at mid is the target, we have found it. Return its index.
        if arr[mid] == x:
            return mid

        # If the target element x is smaller than the middle element,
        # it means x can only be present in the left subarray.
        # Recursively call binary search on the left half: from 'low' to 'mid - 1'.
        elif arr[mid] > x:
            return binary_search_recursive(arr, low, mid - 1, x)

        # If the target element x is larger than the middle element,
        # it means x can only be present in the right subarray.
        # Recursively call binary search on the right half: from 'mid + 1' to 'high'.
        else: # arr[mid] < x
            return binary_search_recursive(arr, mid + 1, high, x)

    else:
        # Step 1 (continued): If 'high' becomes less than 'low', the element is not present in the array.
        return -1

# --- User Input ---
```

```python
# --- User Input ---

# Get list of numbers from user
list_input = input("Enter a list of numbers separated by spaces: ")
user_list = [int(num) for num in list_input.split()]

# Get element to search from user
search_element = int(input("Enter the element to search for: "))

# --- Sort the list first ---
# Binary search requires the list to be sorted to work correctly.
user_list.sort()

# --- Perform binary search ---
# Call the recursive binary search function with the sorted list, initial range, and target element.
result = binary_search_recursive(user_list, 0, len(user_list) - 1, search_element)

# --- Display result clearly and well structured ---
print("\n--- Binary Search Results ---")
print(f"Search Element: {search_element}")
print(f"Sorted List: {user_list}")

if result != -1:
    print(f"Result: Element {search_element} is present at index {result}.")
else:
    print(f"Result: Element {search_element} is not present in the list.")
```

OUTPUT:

```
Enter a list of numbers separated by spaces: 2 0 1 6 3 4
Enter the element to search for: 0

--- Binary Search Results ---
Search Element: 0
Sorted List: [0, 1, 2, 3, 4, 6]
Result: Element 0 is present at index 0.
```

JUSTIFICATION:

The non-transparent version performs correctly but does not clearly explain how recursion and decisions work. The transparent version adds meaningful comments that describe each step of the algorithm. This makes it easier for readers to understand the flow of recursion and base cases. Improved structure helps learners debug and maintain the code more effectively. Clear explanations increase trust in the correctness of the algorithm. Overall, the revised code improves readability, learning value, and transparency.

Question 4: Ethical Evaluation of AI-Based Scoring Systems Scenario

Task: Biased Prompt

Generate a Python program for a job applicant scoring system based on skills, experience, and education. Calculate a final score and display whether the applicant is selected or not. The logic may include one additional condition that is not directly related to job performance, so that the program can later be analyzed for fairness.

CODE:

```python
def calculate_applicant_score(name, skills, experience_years, education_level):
    score = 0

    # Scoring based on skills (example weights)
    skill_weights = {
        "python": 10,
        "java": 8,
        "sql": 7,
        "ml": 12,
        "statistics": 8,
        "communication": 5
    }
    for skill in skills:
        score += skill_weights.get(skill.lower(), 0) # Add points for known skills, 0 for unknown

    # Scoring based on experience
    score += experience_years * 7 # 7 points per year of experience

    # Scoring based on education level
    education_scores = {
        "high school": 10,
        "bachelor's": 30,
        "master's": 50,
        "phd": 70
    }
    score += education_scores.get(education_level.lower(), 0)
```

```python
    # This condition is arbitrary and included *only* for the purpose of later fairness analysis.
    # In an ethical, real-world system, such conditions should be avoided.
    if len(name) % 2 == 0: # If name has an even number of characters
        score += 5 # Arbitrary bonus points
        print(f"  (Arbitrary Condition Bonus: +5 points for even name length)")
    else:
        score -= 5 # Arbitrary penalty points
        print(f"  (Arbitrary Condition Penalty: -5 points for odd name length)")
    # ----------------------------------------------------------------

    return score

# --- Predefined Applicant Data (instead of user input) ---
applicants_data = [
    {
        "name": "Alice",
        "skills": ["python", "sql"],
        "experience_years": 3,
        "education_level": "bachelor's"
    },
    {
        "name": "Bob",
        "skills": ["java", "python", "ml"],
        "experience_years": 5,
        "education_level": "master's"
    },
    {
        "name": "Charlie",
        "skills": ["statistics", "python", "communication"],
```

```python
        {
            "name": "Eve",
            "skills": ["sql", "python"],
            "experience_years": 4,
            "education_level": "bachelor's"
        }
    ]

    # --- Evaluate Applicants ---
    print("\n--- Applicant Evaluation Results ---")
    selection_threshold = 100 # Example threshold for selection

    for applicant in applicants_data:
        print(f"\nApplicant Name: {applicant['name']}")
        print(f"  Skills: {', '.join(applicant['skills'])}")
        print(f"  Experience: {applicant['experience_years']} years")
        print(f"  Education: {applicant['education_level']}")

        final_score = calculate_applicant_score(
            applicant['name'],
            applicant['skills'],
            applicant['experience_years'],
            applicant['education_level']
        )
        print(f"  Final Score: {final_score}")

        if final_score >= selection_threshold:
            print("  Decision: SELECTED")
        else:
            print("  Decision: NOT SELECTED")
```

OUTPUT:

```
--- Applicant Evaluation Results ---

Applicant Name: Alice
  Skills: python, sql
  Experience: 3 years
  Education: bachelor's
  (Arbitrary Condition Penalty: -5 points for odd name length)
  Final Score: 63
  Decision: NOT SELECTED

Applicant Name: Bob
  Skills: java, python, ml
  Experience: 5 years
  Education: master's
  (Arbitrary Condition Penalty: -5 points for odd name length)
  Final Score: 110
  Decision: SELECTED

Applicant Name: Charlie
  Skills: statistics, python, communication
  Experience: 2 years
  Education: phd
  (Arbitrary Condition Penalty: -5 points for odd name length)
  Final Score: 102
  Decision: SELECTED

Applicant Name: Eve
  Skills: sql, python
  Experience: 4 years
  Education: bachelor's
  (Arbitrary Condition Penalty: -5 points for odd name length)
```

Task 2: Un Biased Prompt

Analyze the baised part and remove bias for the code. Generate a efficient, well structured and improvized code.

CODE:

```python
def calculate_applicant_score_unbiased(name, skills, experience_years, education_level):
    score = 0

    # Scoring based on skills (example weights)
    skill_weights = {
        "python": 10,
        "java": 8,
        "sql": 7,
        "ml": 12,
        "statistics": 8,
        "communication": 5
    }
    for skill in skills:
        score += skill_weights.get(skill.lower(), 0) # Add points for known skills, 0 for unknown

    # Scoring based on experience
    score += experience_years * 7 # 7 points per year of experience

    # Scoring based on education level
    education_scores = {
        "high school": 10,
        "bachelor's": 30,
        "master's": 50,
        "phd": 70
    }
    score += education_scores.get(education_level.lower(), 0)

    # The previous arbitrary condition based on name length has been intentionally removed
    # to ensure an unbiased evaluation based purely on job-related qualifications.

    return score

# --- Predefined Applicant Data ---
applicants_data_unbiased = [
    {
        "name": "Alice",
        "skills": ["python", "sql"],
        "experience_years": 3,
        "education_level": "bachelor's"
    },
    {
        "name": "Bob",
        "skills": ["java", "python", "ml"],
        "experience_years": 5,
        "education_level": "master's"
    },
    {
        "name": "Charlie",
        "skills": ["statistics", "python", "communication"],
        "experience_years": 2,
        "education_level": "phd"
    },
    {
        "name": "Eve",
        "skills": ["sql", "python"],
        "experience_years": 4,
        "education_level": "bachelor's"
    }
]

# --- Evaluate Applicants with Unbiased System ---
print("\n--- Unbiased Applicant Evaluation Results ---")
selection_threshold = 100 # Example threshold for selection

for applicant in applicants_data_unbiased:
    print(f"\nApplicant Name: {applicant['name']}")
    print(f"  Skills: {', '.join(applicant['skills'])}")
    print(f"  Experience: {applicant['experience_years']} years")
    print(f"  Education: {applicant['education_level']}")

    final_score = calculate_applicant_score_unbiased(
        applicant['name'],
        applicant['skills'],
        applicant['experience_years'],
        applicant['education_level']
    )
    print(f"  Final Score: {final_score}")

    if final_score >= selection_threshold:
        print("  Decision: SELECTED")
    else:
        print("  Decision: NOT SELECTED")
```

OUTPUT:

```
--- Unbiased Applicant Evaluation Results ---

Applicant Name: Alice
  Skills: python, sql
  Experience: 3 years
  Education: bachelor's
  Final Score: 68
  Decision: NOT SELECTED

Applicant Name: Bob
  Skills: java, python, ml
  Experience: 5 years
  Education: master's
  Final Score: 115
  Decision: SELECTED

Applicant Name: Charlie
  Skills: statistics, python, communication
  Experience: 2 years
  Education: phd
  Final Score: 107
  Decision: SELECTED

Applicant Name: Eve
  Skills: sql, python
  Experience: 4 years
  Education: bachelor's
  Final Score: 75
  Decision: NOT SELECTED
```

JUSTIFICATION:

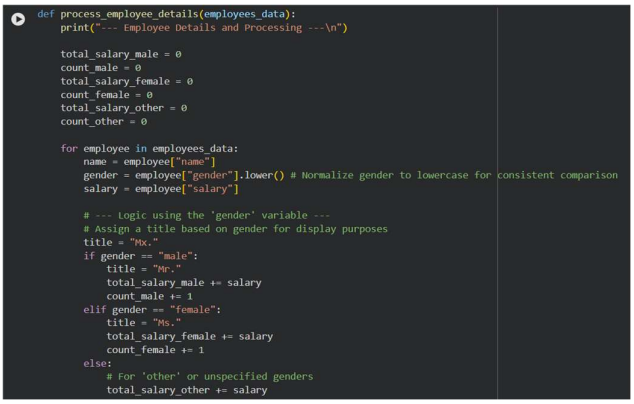
The biased version introduced an arbitrary rule unrelated to job performance, which can unfairly influence outcomes. The revised code removes this non-job related condition to ensure decisions rely only on relevant qualifications. Scoring is now based strictly on skills, experience, and education, improving fairness and consistency. This makes the system more transparent and easier to justify in real hiring scenarios.

Question 5: Inclusiveness and Ethical Variable Design Scenario

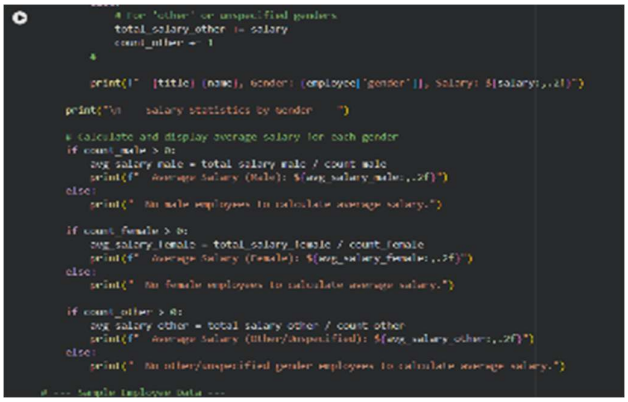
Task 1: Non – Inclusiveness Prompt

Generate a Python code that processes employee details such as name, gender, and salary. The program should use the gender variable in the logic and display the final result. Keep the code simple and realistic.

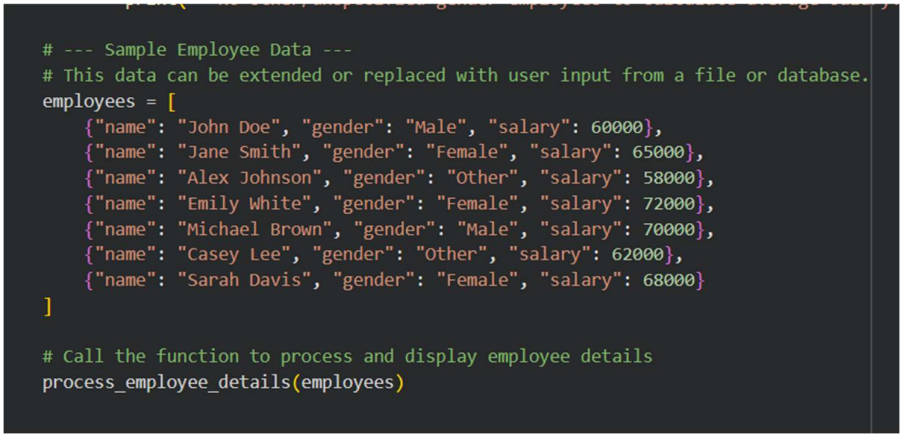
CODE:



```
def process_employee_details(employees_data):
    print("--- Employee Details and Processing ---\n")

    total_salary_male = 0
    count_male = 0
    total_salary_female = 0
    count_female = 0
    total_salary_other = 0
    count_other = 0

    for employee in employees_data:
        name = employee["name"]
        gender = employee["gender"].lower() # Normalize gender to lowercase for consistent comparison
        salary = employee["salary"]

        # --- Logic using the 'gender' variable ---
        # Assign a title based on gender for display purposes
        title = "Mx."
        if gender == "male":
            title = "Mr."
            total_salary_male += salary
            count_male += 1
        elif gender == "female":
            title = "Ms."
            total_salary_female += salary
            count_female += 1
        else:
            # For 'other' or unspecified genders
            total_salary_other += salary
```



```
            # For 'other' or unspecified genders
            total_salary_other += salary
            count_other += 1

        print(f"  {title} {name}, Gender: {employee['gender']}, Salary: ${salary:,.2f}")

    print("\n--- Salary Statistics by Gender ---")

    # Calculate and display average salary for each gender
    if count_male > 0:
        avg_salary_male = total_salary_male / count_male
        print(f"  Average Salary (Male): ${avg_salary_male:,.2f}")
    else:
        print("  No male employees to calculate average salary.")

    if count_female > 0:
        avg_salary_female = total_salary_female / count_female
        print(f"  Average Salary (Female): ${avg_salary_female:,.2f}")
    else:
        print("  No female employees to calculate average salary.")

    if count_other > 0:
        avg_salary_other = total_salary_other / count_other
        print(f"  Average Salary (Other/Unspecified): ${avg_salary_other:,.2f}")
    else:
        print("  No other/unspecified gender employees to calculate average salary.")

# --- Sample Employee Data ---
```



```
# --- Sample Employee Data ---
# This data can be extended or replaced with user input from a file or database.
employees = [
    {"name": "John Doe", "gender": "Male", "salary": 60000},
    {"name": "Jane Smith", "gender": "Female", "salary": 65000},
    {"name": "Alex Johnson", "gender": "Other", "salary": 58000},
    {"name": "Emily White", "gender": "Female", "salary": 72000},
    {"name": "Michael Brown", "gender": "Male", "salary": 70000},
    {"name": "Casey Lee", "gender": "Other", "salary": 62000},
    {"name": "Sarah Davis", "gender": "Female", "salary": 68000}
]

# Call the function to process and display employee details
process_employee_details(employees)
```

OUTPUT:

```
--- Employee Details and Processing ---

  Mr. John Doe, Gender: Male, Salary: $60,000.00
  Ms. Jane Smith, Gender: Female, Salary: $65,000.00
  Mx. Alex Johnson, Gender: Other, Salary: $58,000.00
  Ms. Emily White, Gender: Female, Salary: $72,000.00
  Mr. Michael Brown, Gender: Male, Salary: $70,000.00
  Mx. Casey Lee, Gender: Other, Salary: $62,000.00
  Ms. Sarah Davis, Gender: Female, Salary: $68,000.00

--- Salary Statistics by Gender ---
  Average Salary (Male): $65,000.00
  Average Salary (Female): $68,333.33
  Average Salary (Other/Unspecified): $60,000.00
```

Task 2: Inclusiveness Prompt

Generate a Python code that processes employee details using inclusive coding practices. Use gender-neutral variable names and avoid using gender or identity in the program logic unless strictly necessary. The logic should be fair and based only on job-related factors such as role, performance, or experience.

CODE:

```python
def process_inclusive_employee_data(workforce_data):
    print("--- Inclusive Employee Data Processing ---\n")

    total_employees = len(workforce_data)
    total_compensation = 0
    total_performance_score = 0
    total_experience_years = 0

    # Dictionaries to store aggregated data by role
    role_compensation = {}
    role_performance = {}
    role_counts = {}

    for person in workforce_data:
        name = person["name"]
        role = person["role"]
        compensation = person["compensation"]
        performance_rating = person["performance_rating"]
        years_of_experience = person["years_of_experience"]

        total_compensation += compensation
        total_performance_score += performance_rating
        total_experience_years += years_of_experience

        # Aggregate data by role (job-related factor)
        if role not in role_counts:
            role_counts[role] = 0
            role_compensation[role] = 0
            role_performance[role] = 0
```

```
        role_counts[role] += 1
        role_compensation[role] += compensation
        role_performance[role] += performance_rating

        # Display basic employee information (no identity-based titles or logic)
        print(f"  {name}, Role: {role}, Compensation: ${compensation:,.2f}, Performance: {performance_rating}/5, Experience: {years_of_experience} years")

    print("\n--- Overall Workforce Statistics ---")
    if total_employees > 0:
        print(f"  Total Employees: {total_employees}")
        print(f"  Average Compensation: ${total_compensation / total_employees:,.2f}")
        print(f"  Average Performance Rating: {total_performance_score / total_employees:.2f}/5")
        print(f"  Average Years of Experience: {total_experience_years / total_employees:.2f} years")
    else:
        print("  No employee data to process.")

    print("\n--- Statistics by Role ---")
    if role_counts:
        for role_name, count in role_counts.items():
            print(f"  {role_name} ({count} employees):")
            print(f"    Average Compensation: ${role_compensation[role_name] / count:,.2f}")
            print(f"    Average Performance Rating: {role_performance[role_name] / count:.2f}/5")
    else:
        print("  No role-specific data.")


# --- Sample Employee Data (inclusive and job-focused) ---
workforce = [
    {"name": "Pat Lee", "role": "Software Engineer", "compensation": 90000, "performance_rating": 4, "years_of_experience": 3},
```

OUTPUT:

```
--- Inclusive Employee Data Processing ---

  Pat Lee, Role: Software Engineer, Compensation: $90,000.00, Performance: 4/5, Experience: 3 years
  Casey Smith, Role: Data Analyst, Compensation: $75,000.00, Performance: 3/5, Experience: 2 years
  Jamie Brown, Role: Software Engineer, Compensation: $110,000.00, Performance: 5/5, Experience: 6 years
  Alex Green, Role: Project Manager, Compensation: $105,000.00, Performance: 4/5, Experience: 8 years
  Sam White, Role: Data Analyst, Compensation: $80,000.00, Performance: 4/5, Experience: 4 years

--- Overall Workforce Statistics ---
  Total Employees: 5
  Average Compensation: $92,000.00
  Average Performance Rating: 4.00/5
  Average Years of Experience: 4.60 years

--- Statistics by Role ---
  Software Engineer (2 employees):
    Average Compensation: $100,000.00
    Average Performance Rating: 4.50/5
  Data Analyst (2 employees):
    Average Compensation: $77,500.00
    Average Performance Rating: 3.50/5
  Project Manager (1 employees):
    Average Compensation: $105,000.00
    Average Performance Rating: 4.00/5
```

JUSTIFICATION:

The non-inclusive version relies on gender in program logic, which can introduce bias and unfair assumptions. The improved version removes identity-based conditions and focuses only on job-related attributes. This makes the code more ethical, professional, and suitable for real-world systems. Overall, the revised design promotes inclusiveness, equity, and responsible software development.