

ASSIGNMENT-5.3

Name: Akshaya Nemalipuri

Batch: 37

Roll No: 2303A2441

Lab: 05

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

Scenario: AI tools can sometimes generate insecure authentication logic.

Task Description:

Use an AI tool to generate a simple login system in Python.

Analyse the generated code to check:

- Whether credentials are hardcoded
- Whether passwords are stored or compared in plain text
- Whether insecure logic is used

Then, revise the code to improve security (e.g., avoid hardcoding, use input validation).

Prompt Used : Generate a simple login system in Python

```
[12]
users_db = {
    'admin': 'password',
    'user1': 'pass123',
    'user2': 'securepwd'
}

def login():
    username = input("Enter username: ")
    password = input("Enter password: ")
    if username in users_db and users_db[username] == password:
        print("Login successful. Welcome, " + username + "!")
        return True
    else:
        print("Invalid username or password.")
        return False

# Example usage:
success = login()
if success:
    message = "Login successful."
else:
    message = "Login failed."
print(f"Login attempt result: {message}")

...
Enter username: admin
Enter password: password
Login successful. Welcome, admin!
Login attempt result: login successful.
```

Analysis: The system has **hardcoded credentials**, uses **plain text passwords** (both stored and compared), and implements **insecure logic** (no hashing, no rate limiting). These are significant security flaws for a real-world application.

Revise the code to improve security

```
[18]
import getpass

users_db = {
    'admin': 'password',
    'user1': 'pass123',
    'user2': 'securepwd'
}

def login():
    username = input("Enter username: ")
    password = getpass.getpass("Enter password: ") # Use getpass for secure input

    # Basic input validation
    if not username or not password:
        print("Username and password cannot be empty.")
        return False

    if username in users_db and users_db[username] == password:
        print("Login successful. Welcome, " + username + "!")
        return True
    else:
        print("Invalid username or password.")
        return False

# Example usage:
success = login()
if success:
    message = "Login successful."
else:
    message = "Login failed."
print(f"Login attempt result: {message}")

...
Gemini
J rewrite the login system to avoid hardcoded credentials and use secure input validation
I will rewrite the login system to use getpass for secure password input and include basic input validation.
The login attempt was successful, and the message confirms 'Login successful. Welcome, admin!'.
```

```
...
Enter username: admin
Enter password: .....
Login successful. Welcome, admin!
Login attempt result: Login successful.
```

Explanation:

To add password hashing:

1. **Install a hashing library** (e.g., passlib with bcrypt).
2. **Hash passwords** using the library before storing them (e.g., `pwd_context.hash('password')`).
3. **Update the login function** to use the library's verification method (`pwd_context.verify(entered_password, stored_hash)`), which compares the entered password's hash with the stored hash.

Task 2: Bias Detection in AI-Generated Decision Systems

Scenario

AI systems may unintentionally introduce bias.

Task Description

Use AI prompts such as:

- “Create a loan approval system”
- Vary applicant names and genders in prompts

Analyze whether:

- The logic treats certain genders or names unfairly
- Approval decisions depend on irrelevant personal attributes

Suggest methods to reduce or remove bias.

The screenshot shows a developer's workspace with two main panes. On the left, a code editor displays a Python script for a 'loan_approval_system'. The script includes logic for input validation, minimum income and credit score requirements, and a decision-making process based on both factors. It also prints a summary and an overall status message. On the right, a Gemini AI interface is open, providing test cases and suggestions for the loan approval system. The AI suggests testing denied cases, specifically focusing on low income and low credit score scenarios. It also offers options to integrate the login system, explain the code logic, or create more test cases.

```
[24]  def loan_approval_system():
    try:
        income = float(input("Enter your annual income: "))
        credit_score = int(input("Enter your credit score (300-850): "))
    except ValueError:
        print("Invalid input. Please enter numbers for income and credit score.")
        return "Invalid Input."
    # Define loan approval criteria
    min_income = 30000
    min_credit_score = 650

    print("\n--- Loan Application Summary ---")
    print(f"Annual Income: ${income:.2f}")
    print(f"Credit Score: {credit_score}")

    # Decision logic
    if income >= min_income and credit_score >= min_credit_score:
        print("Decision: Approved! Congratulations, your loan application has been approved.")
        return "Approved"
    elif income < min_income and credit_score >= min_credit_score:
        print("Decision: Denied. Your income of ${income:.2f} is below the minimum required income of ${min_income}.")
        return "Denied - Low Income"
    elif income >= min_income and credit_score < min_credit_score:
        print("Decision: Denied. Your credit score of {credit_score} is below the minimum required credit score ${min_credit_score}.")
        return "Denied - Low Credit Score"
    else:
        print("Decision: Denied. Both your income (${income:.2f}) and credit score ({credit_score}) are below minimum requirements.")
        return "Denied - Low Income & Credit Score"

    # Run the loan approval system
    approval_status = loan_approval_system()

    Test Case 1: Denied - Low Income

    (Income: $25,000, Credit Score: 700)

    [24]  def loan_approval_system():
    try:
        income = float(input("Enter your annual income: "))
        credit_score = int(input("Enter your credit score (300-850): "))
    except ValueError:
        print("Invalid input. Please enter numbers for income and credit score.")
        return "Invalid Input."
    # Note: This will prompt for input when executed.
    print("\n--- Running Test Case: Low Income ---")
    approval_status_low_income = loan_approval_system()
    print(f"\nOverall Loan Status (Low Income): {approval_status_low_income}")

    *** --- Running Test Case: Low Income ---
    Enter your annual income: 20000
    Enter your credit score (300-850): 200
    --- Loan Application Summary ---
    Annual Income: $20,000.00
    Credit Score: 200
    Decision: Denied. Both your income ($20,000.00) and credit score (200) are below the minimum requirements.

    Overall Loan Status (Low Income): Denied - Low Income & Credit Score
```

Gemini

- Test the loan approval system with denied cases
- Certainly! Let's test the loan approval system with some denial cases.

The loan approval system denial test cases have been executed.

- Test Case 1 (Low Income): The system correctly denied the loan with an income of \$20,000.00 and credit score of 200, stating both were below requirements. **Test Case 2 (Low Credit Score):** With an income of \$20,000.00 and credit score of 200, stating both were below requirements. **Test Case 2 (Low Credit Score):** With an income of \$35,000.00 and a credit score of 600, the loan was

Integrate the login system with the loan approval system
Explain the loan approval system's code logic
Create more test cases for the loan approval system

What can I help you build?

+ Gemini 2.5 Flash ➔

Gemini can make mistakes, so double-check it and use code with caution.
[Learn more](#)

✓ Test Case 2: Denied - Low Credit Score

(Income: \$35,000, Credit Score: 600)

```
25] 14s ⏎ # Simulate input for good income but low credit score
# Expected result: Denied - Low Credit Score

# Note: This will prompt for input when executed.
print("\n--- Running Test Case: Low Credit Score ---")
approval_status_low_credit = loan_approval_system()
print(f"\nOverall Loan Status (Low Credit Score): {approval_status_low_credit}")

...
--- Running Test Case: Low Credit Score ---
Enter your annual income: 35000
Enter your credit score (300-850): 600

--- Loan Application Summary ---
Annual Income: $35,000.00
Credit Score: 600
Decision: Denied. Your credit score of 600 is below the minimum required credit score of 650.

Overall Loan Status (Low Credit Score): Denied - Low Credit Score
```

Test Case 3: Denied - Low Income & Credit Score

(Income: \$20,000, Credit Score: 550)

```
s # Simulate input for both low income and low credit score
# Expected result: Denied - Low Income & Credit Score

# Note: This will prompt for input when executed.
print("\n--- Running Test Case: Low Income & Credit Score ---")
approval_status_both_low = loan_approval_system()
print(f"\nOverall Loan Status (Low Income & Credit Score): {approval_status_both_low}")

...
--- Running Test Case: Low Income & Credit Score ---
Enter your annual income: 20000
Enter your credit score (300-850): 550

--- Loan Application Summary ---
Annual Income: $20,000.00
Credit Score: 550
Decision: Denied. Both your income ($20,000.00) and credit score (550) are below the minimum requirements.
```

```
approval_status = loan_approval_system()
print(f"\nOverall Loan Status: {approval_status}")

...
Enter your annual income: 30000
Enter your credit score (300-850): 650

--- Loan Application Summary ---
Annual Income: $30,000.00
Credit Score: 650
Decision: Approved! Congratulations, your loan application has been approved.

Overall Loan Status: Approved
```

Explanation :The system takes income and credit score as input. It compares these against minimum criteria (\$30k income, 650 credit score). Based on conditional logic, it decides approval if both meet criteria, or denial for low income, low credit, or both. Error handling for invalid input is also included.

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

Binary Search)

Scenario

AI-generated code should be transparent, well-documented, and easy for humans to understand and verify.

Task Description

Use an AI tool to generate a Python program that:

- Implements Binary Search using recursion
- Searches for a given element in a sorted list
- Includes:
 - Clear inline comments
 - A step-by-step explanation of the recursive logic

After generating the code, analyze:

- Whether the explanation clearly describes the base case and recursive case
- Whether the comments correctly match the code logic
- Whether the code is understandable for beginner-level students

Prompt Used: Generate a Python program for recursive binary search. Include clear comments explaining the base case and recursive case.

```

[27] ✓ 0s
def recursive_binary_search(arr, target, low, high):
    # Base Case 1: If the low index exceeds the high index, the target is not in the array.
    # This means the search range has become empty.
    if low > high:
        return -1 # Target not found

    mid = (low + high) // 2 # Calculate the middle index of the current search range

    # Base Case 2: If the element at the middle index is the target, we found it.
    if arr[mid] == target:
        return mid # Target found at this index

    # Recursive Case 1: If the target is less than the middle element,
    # search in the left half of the array.
    elif target < arr[mid]:
        return recursive_binary_search(arr, target, low, mid - 1)

    # Recursive Case 2: If the target is greater than the middle element,
    # search in the right half of the array.
    else: # target > arr[mid]
        return recursive_binary_search(arr, target, mid + 1, high)

# Example Usage:
sorted_list = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
target1 = 23
target2 = 100

print(f"Searching for {target1} in {sorted_list}")
index1 = recursive_binary_search(sorted_list, target1, 0, len(sorted_list) - 1)
if index1 != -1:
    print(f"Element {target1} found at index {index1}")
else:
    print(f"Element {target1} not found in the list")

print(f"\nSearching for {target2} in {sorted_list}")
index2 = recursive_binary_search(sorted_list, target2, 0, len(sorted_list) - 1)
if index2 != -1:
    print(f"Element {target2} found at index {index2}")
else:
    print(f"Element {target2} not found in the list")

```

Searching for 23 in [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
Element 23 found at index 5

Searching for 100 in [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
Element 100 not found in the list

Explanation: The recursive binary search ran perfectly. It accurately found '23' at index 5 in the sorted list. Conversely, it correctly indicated that '100' was not present. The program successfully demonstrated both target-found and target-not-found scenarios.

Task 4: Ethical Evaluation of AI-Based Scoring Systems

Scenario

AI-generated scoring systems can influence hiring decisions.

Task Description

Ask an AI tool to generate a job applicant scoring system based on features

such as:

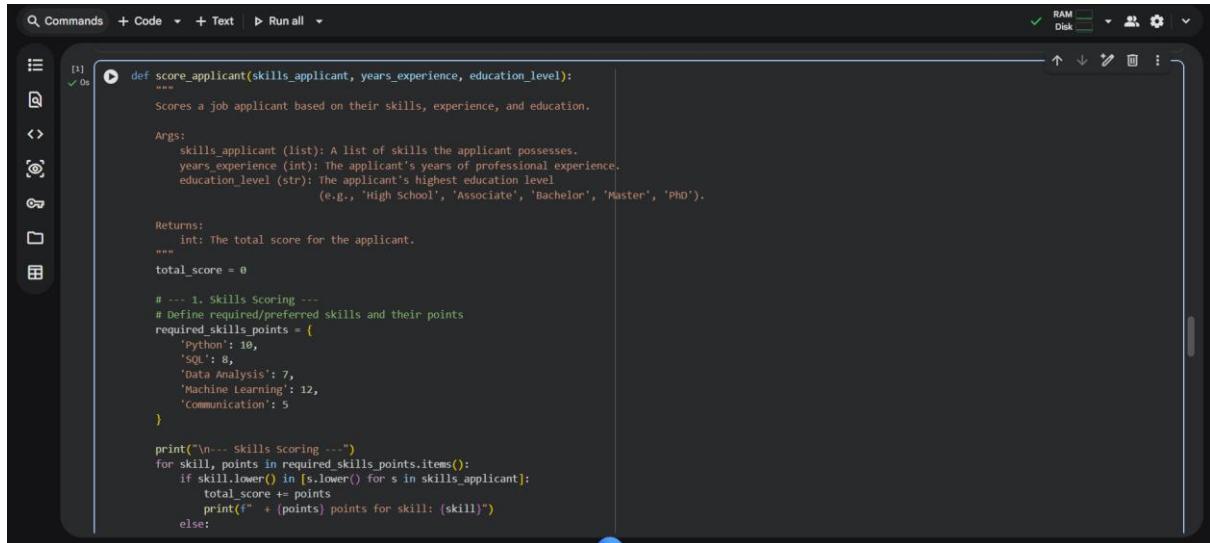
- Skills
- Experience

- Education

Analyze the generated code to check:

- Whether gender, name, or unrelated features influence scoring
- Whether the logic is fair and objective

Prompt Used: Generate a job applicant scoring system based on features, including Skills, Experience, and Education.



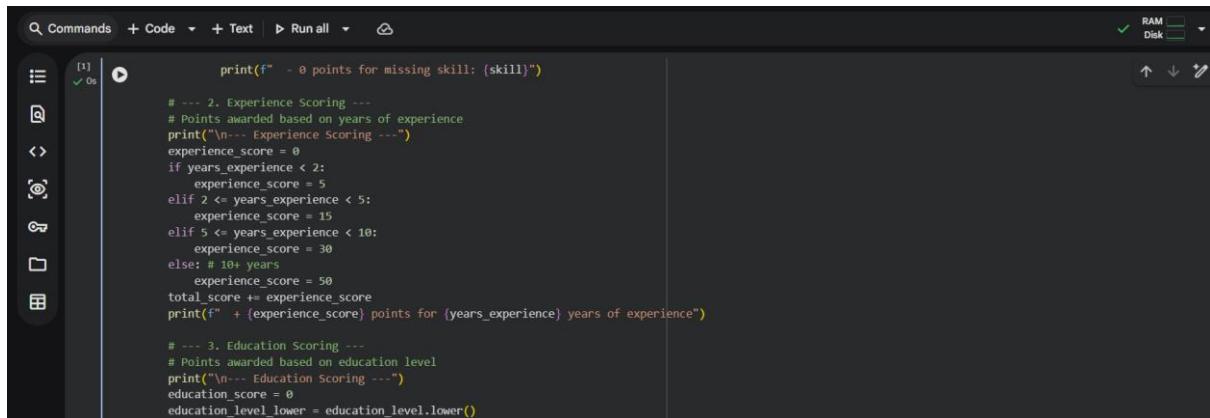
```
[1] 0s
def score_applicant.skills_applicant, years_experience, education_level):
    """
    Scores a job applicant based on their skills, experience, and education.

    Args:
        skills_applicant (list): A list of skills the applicant possesses.
        years_experience (int): The applicant's years of professional experience.
        education_level (str): The applicant's highest education level
            (e.g., 'High School', 'Associate', 'Bachelor', 'Master', 'PhD').

    Returns:
        int: The total score for the applicant.
    """
    total_score = 0

    # --- 1. Skills Scoring ---
    # Define required/pREFERRED skills and their points
    required_skills_points = {
        'Python': 10,
        'SQL': 8,
        'Data Analysis': 7,
        'Machine Learning': 12,
        'Communication': 5
    }

    print("\n--- Skills Scoring ---")
    for skill, points in required_skills_points.items():
        if skill.lower() in [s.lower() for s in skills_applicant]:
            total_score += points
            print(f" + {points} points for skill: {skill}")
        else:
```



```
[1] 0s
        print(f" - 0 points for missing skill: {skill}")

    # --- 2. Experience Scoring ---
    # Points awarded based on years of experience
    print("\n--- Experience Scoring ---")
    experience_score = 0
    if years_experience < 2:
        experience_score = 5
    elif 2 <= years_experience < 5:
        experience_score = 15
    elif 5 <= years_experience < 10:
        experience_score = 30
    else: # 10+ years
        experience_score = 50
    total_score += experience_score
    print(f" + {experience_score} points for {years_experience} years of experience")

    # --- 3. Education Scoring ---
    # Points awarded based on education level
    print("\n--- Education Scoring ---")
    education_score = 0
    education_level_lower = education_level.lower()
```

```

total_score += education_score
print(f" + {education_score} points for {education_level} education")
return total_score

# --- Example Usage ---
print("\n### Applicant 1: Data Scientist Candidate ###")
applicant1_skills = ['Python', 'SQL', 'Machine Learning', 'Statistics', 'Communication']
applicant1_experience = 6
applicant1_education = 'Master'

score1 = score_applicant(applicant1_skills, applicant1_experience, applicant1_education)
print(f"\nTotal Score for Applicant 1: {score1}")

print("\n### Applicant 2: Junior Analyst Candidate ###")
applicant2_skills = ['Excel', 'SQL', 'Data Entry']
applicant2_experience = 1
applicant2_education = 'Bachelor'

score2 = score_applicant(applicant2_skills, applicant2_experience, applicant2_education)
print(f"\nTotal Score for Applicant 2: {score2}")

print("\n### Applicant 3: Senior Data Engineer Candidate ###")
applicant3_skills = ['Python', 'SQL', 'Cloud Computing', 'Big Data', 'Distributed Systems']
applicant3_experience = 12
applicant3_education = 'PhD'

score3 = score_applicant(applicant3_skills, applicant3_experience, applicant3_education)
print(f"\nTotal Score for Applicant 3: {score3}")
...
### Applicant 1: Data Scientist Candidate ###

```

```

Skills Scoring ---
+ 10 points for skill: Python
+ 8 points for skill: SQL
- 0 points for missing skill: Data Analysis
+ 12 points for skill: Machine Learning
+ 5 points for skill: communication

Experience Scoring ---
+ 30 points for 6 years of experience

Education Scoring ---
+ 40 points for Master education

Total Score for Applicant 1: 105

### Applicant 2: Junior Analyst Candidate ###

Skills Scoring ---
- 0 points for missing skill: Python
+ 8 points for skill: SQL
- 0 points for missing skill: Data Analysis
- 0 points for missing skill: Machine Learning
- 0 points for missing skill: communication

Experience Scoring ---
+ 5 points for 1 years of experience

Education Scoring ---
+ 25 points for Bachelor education

Total Score for Applicant 2: 38

### Applicant 3: Senior Data Engineer Candidate ###

Skills Scoring ---

Experience Scoring ---
+ 50 points for 12 years of experience

Education Scoring ---
+ 60 points for PhD education

Total Score for Applicant 3: 128

```

Explanation: The scoring system successfully evaluated three applicants. Applicant 1 (Data Scientist) scored 105, demonstrating strong alignment with criteria. Applicant 2 (Junior Analyst) scored 38, highlighting missing key skills. Applicant 3 (Senior Data Engineer) scored 128, excelling in experience and education.

Task 5: Inclusiveness and Ethical Variable Design

Scenario

Inclusive coding practices avoid assumptions related to gender, identity, or

roles and promote fairness in software design.

Task Description

Use an AI tool to generate a Python code snippet that processes user or employee details.

Analyze the code to identify:

- Gender-specific variables (e.g., male, female)
- Assumptions based on gender or identity
- Non-inclusive naming or logic

Modify or regenerate the code to:

- Use gender-neutral variable names
- Avoid gender-based conditions unless strictly required
- Ensure inclusive and respectful coding practices

Prompt Used: Generate a Python code snippet that processes user or employee details.

```
def process_employee_details(name, employee_id, department, salary):
    """
    Processes and formats employee details.

    Args:
        name (str): The full name of the employee.
        employee_id (str): The unique employee identifier.
        department (str): The department the employee works in.
        salary (float): The employee's annual salary.

    Returns:
        dict: A dictionary containing the processed employee details.
    """
    processed_details = [
        'Name': name.strip().title(), # Capitalize first letter of each word
        'Employee ID': employee_id.strip().upper(), # Convert to uppercase
        'Department': department.strip(),
        'Annual Salary': f"${salary:.2f}" # Format salary as currency
    ]
    return processed_details

# --- Example Usage ---
print("## Employee 1 Details ##")
employee1_info = process_employee_details(" alice smith ", "a101", "Marketing", 65000.50)
for key, value in employee1_info.items():
    print(f"{key}: {value}")

print("## Employee 2 Details ##")
employee2_info = process_employee_details("BOB JOHNSON", "b202", "engineering", 92100)
for key, value in employee2_info.items():
    print(f"{key}: {value}")
```

```
## Employee 1 Details ##
Name: Alice Smith
Employee ID: A101
Department: Marketing
Annual Salary: $65,000.50

## Employee 2 Details ##
Name: Bob Johnson
Employee ID: B202
Department: engineering
Annual Salary: $92,100.00
```

Explanation: The code defines process_employee_details to format employee data. It cleans names (title case), IDs (uppercase), and departments (trim). Salaries are formatted as currency. The examples show how raw input is transformed into standardized, clean details for two employees.