AI-Assisted-Coding Week-5.1

2303A51878

B-28

Task Description 1 – Privacy in API Usage

Objective

To demonstrate how weather data can be fetched using a Python program with and without secure API key handling, and to highlight best practices for protecting sensitive credentials.

1) Original AI-Generated Code (Insecure – API Key Hardcoded)

Problem

The API key is written directly in the source code.
This is unsafe because:

- Anyone accessing the code can see the key
- If uploaded to GitHub, the key can be stolen
- Violates secure coding and privacy principles

Insecure Python Code

```
import requests

API_KEY = "YOUR_API_KEY_HERE"

CITY = "Delhi"

URL=f"https://api.openweathermap.org/data/2.5/weather?q={CITY}&appid={API_KEY}"

response = requests.get(URL)

data = response.json()

print("City:", data["name"])

print("Temperature:", data["main"]["temp"])

print("Weather:", data["weather"][0]["description"])
```

Security Issue Identified

- API key is exposed in plain text
- High risk of credential leakage

2) Secure Version Using Environment Variables (Recommended)

Solution

Store the API key in an environment variable and access it securely in the code.

Step 1: Set Environment Variable

Windows (Command Prompt)

```
setx WEATHER_API_KEY "your_api_key_here"
```

Linux / macOS

```
export WEATHER_API_KEY="your_api_key_here"
```

Step 2: Secure Python Code

```
import os

import requests

API_KEY = os.getenv("WEATHER_API_KEY")

if not API_KEY:

    raise ValueError("API key not found. Please set WEATHER_API_KEY environment variable.")

CITY = "Delhi"

URL=f"https://api.openweathermap.org/data/2.5/weather?q={CITY}&appid={API_KEY}"

response = requests.get(URL)

data = response.json()

print("City:", data["name"])

print("Temperature:", data["main"]["temp"])

print("Weather:", data["weather"][0]["description"])
```

Why This Approach Is Secure

- API key is not visible in the source code
- Safe to upload code to GitHub
- Follows industry best practices
- Protects user privacy and credentials

Task Description 2 – Privacy & Security in File Handling

Objective

To analyze how an AI-generated Python script stores user data in a file and to identify privacy risks when sensitive data (passwords) are stored insecurely. Then, to provide a **secure revised version** using **password hashing**.

Part 1: Original AI-Generated Code (Insecure – Plain Text Storage)

Description

The AI-generated script stores user details (name, email, password) directly into a file without any protection.

Insecure Python Code

```
name = input("Enter name: ")

email = input("Enter email: ")

password = input("Enter password: ")

with open("users.txt", "a") as file:

    file.write(f"Name: {name}, Email: {email}, Password: {password}\n")

print("User data saved successfully.")
```

Sample Input

Enter name: Rahul

Enter email: rahul@gmail.com

Enter password: rahul123

Program Output (Console)

User data saved successfully.

File Output (users.txt)

Name: Rahul, Email: rahul@gmail.com, Password: rahul123

Privacy & Security Risks Identified

Plain Text Password Storage

- Passwords are stored exactly as entered.
- Anyone with file access can read them.

No Encryption or Hashing

- Violates basic security principles.
- Unsafe for real-world applications.

High Risk if File Is Leaked

- Enables identity theft.
- Leads to account compromise.

Part 2: Secure Revised Version (Password Hashing Implemented)

Solution

Instead of storing passwords directly:

- Use **hashing** to convert the password into a non-reversible format.
- Store **only the hashed password**.

Secure Python Code Using Hashing

```
import hashlib
name = input("Enter name: ")
email = input("Enter email: ")
password = input("Enter password: ")
hashed_password = hashlib.sha256(password.encode()).hexdigest()
with open("users_secure.txt", "a") as file:
    file.write(f"Name: {name}, Email: {email}, PasswordHash: {hashed_password}\n")
print("User data saved securely.")
```

Sample Input

Enter name: Rahul

Enter email: rahul@gmail.com

Enter password: rahul123

Program Output (Console)

User data saved securely.

File Output (users_secure.txt)

Name: Rahul, Email: rahul@gmail.com, PasswordHash: 4e2a8a3d5f9c7a0c9c1f64b2c5d58b91f2f9b7a65a1e9d3d3b5a8f4c1a7b

Why Hashing Is Secure

- Original password cannot be retrieved from hash
- Protects users even if the file is leaked
- Industry-accepted security practice
- Meets privacy and ethical coding standards

Task Description 3 – Transparency in Algorithm Design

Objective

To use an AI-generated Python program to check whether a given number is an **Armstrong number**, and to provide a **clear line-by-line explanation** demonstrating algorithm transparency.

What Is an Armstrong Number?

An **Armstrong number** (also called a Narcissistic number) is a number where:

> Sum of each digit raised to the power of number of digits = Original number

Example

- $153 \rightarrow 1^3 + 5^3 + 3^3 = 153$ ✅
- $123 \rightarrow 1^3 + 2^3 + 3^3 = 36$ ❌

AI-Generated Armstrong Number Program (Transparent & Commented)

Python Code

```python
def is_armstrong(number):
    # Convert the number to string to easily access each digit
    num_str = str(number)

    # Count the total number of digits in the number
    num_digits = len(num_str)

    # Initialize a variable to store the sum of powered digits
    total = 0

    # Loop through each digit in the number
    for digit in num_str:
        # Convert digit back to integer and raise it to num_digits power
        total += int(digit) ** num_digits

    # Check if the calculated sum equals the original number
    if total == number:
        return True
    else:
        return False

# Take input from the user
num = int(input("Enter a number: "))

# Check and display result
if is_armstrong(num):
    print(num, "is an Armstrong number")
else:
    print(num, "is not an Armstrong number")
```

Sample Output

Input

Enter a number: 153

Output

153 is an Armstrong number

Another Sample

Input

Enter a number: 123

Output

123 is not an Armstrong number

Line-by-Line Explanation (Transparency Check)

Code Line	Explanation
def is_armstrong(number):	Defines a function to check Armstrong condition
num_str = str(number)	Converts number to string to iterate over digits
num_digits = len(num_str)	Counts total digits in the number
total = 0	Initializes sum of powered digits
for digit in num_str:	Loops through each digit
int(digit) ** num_digits	Raises digit to power of digit count
total += …	Adds result to total sum
if total == number:	Compares computed sum with original number
return True / False	Returns result of check
input()	Accepts user input
print()	Displays result clearly

Verification: Explanation vs Code Functionality

Explanation Accuracy

- Each line explanation **matches exactly** what the code performs.
- No hidden logic or unexplained operations.

Algorithm Transparency

- Variables have meaningful names.
- Logic is easy to trace.
- Comments clearly describe each step.

Task Description 4 – Transparency in Algorithm Comparison

Objective

To implement **QuickSort** and **BubbleSort** using Python and explain **step-by-step how each algorithm works**, highlighting their **differences in logic and efficiency**.

Algorithm 1: Bubble Sort (Simple & Transparent)

Python Code with Comments

```
def bubble_sort(arr):
    n = len(arr)

    # Loop through the entire list multiple times
    for i in range(n):
        # Compare adjacent elements in each pass
        for j in range(0, n - i - 1):
            # Swap if elements are in the wrong order
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

    return arr
```

How Bubble Sort Works (Simple Explanation)

Bubble Sort repeatedly compares **adjacent elements** and swaps them if they are in the wrong order. With each pass, the **largest element "bubbles up" to the end** of the list. This process continues until the list is fully sorted.

Bubble Sort is easy to understand and implement, but it is **slow for large inputs** because it compares elements many times, even if the list is almost sorted.

Algorithm 2: Quick Sort (Efficient & Divide-and-Conquer)

Python Code with Comments

```
def quick_sort(arr):
    # Base case: lists with 0 or 1 element are already sorted
    if len(arr) <= 1:
        return arr

    # Choose a pivot element
    pivot = arr[len(arr) // 2]

    # Elements smaller than pivot
    left = [x for x in arr if x < pivot]

    # Elements equal to pivot
    middle = [x for x in arr if x == pivot]

    # Elements greater than pivot
    right = [x for x in arr if x > pivot]

    # Recursively sort left and right parts
    return quick_sort(left) + middle + quick_sort(right)
```

How Quick Sort Works (Simple Explanation)

Quick Sort follows a **divide-and-conquer** approach. It selects a **pivot element** and divides the list into three parts: elements smaller than the pivot, equal to the pivot, and greater than the pivot. The smaller and greater parts are sorted recursively, and all parts are then combined to produce the final sorted list.

Quick Sort is much **faster than Bubble Sort for large datasets**, although its logic is more complex.

Transparent Comparison of Both Algorithms

Logic Difference

Bubble Sort works by **repeatedly swapping adjacent elements**, while Quick Sort works by **dividing the list into smaller parts using a pivot**.

Efficiency Difference

Bubble Sort compares elements many times, leading to poor performance for large lists. Quick Sort reduces the problem size in each step, making it significantly faster in most cases.

Ease of Understanding

Bubble Sort is easier for beginners to understand. Quick Sort requires understanding recursion and partitioning.

Time Complexity (Easy to Remember)

- **Bubble Sort:**
  Worst Case → **$O(n^2)$**
  Best Case → **$O(n)$** (if optimized)
- **Quick Sort:**
  Average Case → **$O(n \log n)$**
  Worst Case → **$O(n^2)$** (rare, depends on pivot)

Sample Input

arr = [64, 34, 25, 12, 22, 11, 90]

Sample Output

Bubble Sort Result: [11, 12, 22, 25, 34, 64, 90]

Quick Sort Result: [11, 12, 22, 25, 34, 64, 90]

Task Description 5 – Transparency in AI Recommendations

Objective

To design a **product recommendation system** that not only suggests products to users but also **clearly explains why each product is recommended**, ensuring transparency in AI decision-making.

Explainable Product Recommendation System (Python)

Python Code with Explanations Built-In

def recommend_products(user_preferences, products):

```
    recommendations = []

    # Go through each product in the product list
    for product in products:
        reasons = []

        # Check category preference
        if product["category"] in user_preferences["categories"]:
            reasons.append(f"matches your interest in {product['category']}")

        # Check budget preference
        if product["price"] <= user_preferences["max_price"]:
            reasons.append(f"is within your budget of ₹{user_preferences['max_price']}")

        # If at least one reason exists, recommend the product
        if reasons:
            recommendations.append({
                "product": product["name"],
                "reason": " and ".join(reasons)
            })

    return recommendations


# Sample user preferences
user_preferences = {
    "categories": ["Electronics", "Books"],
    "max_price": 1500
}

# Sample product database
products = [
```

```python
    {"name": "Wireless Mouse", "category": "Electronics", "price": 999},
    {"name": "Python Programming Book", "category": "Books", "price": 799},
    {"name": "Office Chair", "category": "Furniture", "price": 4500},
    {"name": "USB-C Cable", "category": "Electronics", "price": 299}
]

# Generate recommendations
results = recommend_products(user_preferences, products)

# Display recommendations with reasons
for item in results:
    print(f"Recommended Product: {item['product']}")
    print(f"Reason: This product {item['reason']}.\n")
```

Sample Output

Recommended Product: Wireless Mouse

Reason: This product matches your interest in Electronics and is within your budget of ₹1500.

Recommended Product: Python Programming Book

Reason: This product matches your interest in Books and is within your budget of ₹1500.

Recommended Product: USB-C Cable

Reason: This product matches your interest in Electronics and is within your budget of ₹1500.

How Transparency Is Achieved

The recommendation system evaluates each product based on **user-defined preferences**, such as product category and budget. Instead of giving a recommendation silently, the system explicitly records **the reasons** behind each suggestion. These reasons are shown to the user in plain language, making the decision process easy to understand.

Evaluation of Explanation Quality

Are the explanations understandable?

Yes. The explanations:

- Use **simple, human-readable language**
- Clearly link recommendations to **user preferences**
- Avoid technical or hidden logic

Do they match system behavior?

Yes. Each recommendation is justified by:

- Category match
- Budget suitability

No product is recommended without a clear reason.

Transparency Assessment

The AI system is **transparent and explainable**, as users can see exactly *why* each product was suggested.