Assignment-5.1

2303A51018

Batch-28

Task Description #1 (Privacy in API Usage)

Task: Use an AI tool to generate a Python program that connects to a weather API.

Prompt:

"Generate code to fetch weather data securely without exposing API keys in the code."

Expected Output:

• Original AI code (check if keys are hardcoded).

• Secure version using environment variables.

Code

```
import requests

# Insecure: API key is hardcoded
API_KEY = "YOUR_API_KEY_HERE"
CITY = "London"
URL = f"http://api.openweathermap.org/data/2.5/weather?q={CITY}&appid={API_KEY}"

response = requests.get(URL)

if response.status_code == 200:
    data = response.json()
    print(f"Weather in {CITY}: {data['weather'][0]['description']}")
else:
```

print("Failed to fetch weather data")

Output

export WEATHER_API_KEY="your_real_api_key_here"

- **How it works:**
  - `os.getenv()` fetches the value of an **environment variable**.
  - The API key is now stored **outside the code**, in your system's environment.
  - If the environment variable isn't set, the program raises an error:

Task Description #2 (Privacy & Security in File Handling)

Task: Use an AI tool to generate a Python script that stores user data

(name, email, password) in a file.

Analyze: Check if the AI stores sensitive data in plain text or without

encryption.

Expected Output:

• Identified privacy risks.

• Revised version with encrypted password storage (e.g., hashing).

Prompt

import bcrypt

# Collect user data

name = input("Enter your name: ")

email = input("Enter your email: ")

password = input("Enter your password: ")

# Hash the password

hashed_password = bcrypt.hashpw(password.encode(), bcrypt.gensalt())

```
# Store data in file

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

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


print("User data saved securely!")
```

Output

Enter your name: Alice

Enter your email: alice@example.com

Enter your password: mypassword123

Explanation

## Optional Security Enhancements:

- Limit access to the file with OS-level permissions (`chmod 600` on Linux).
- Validate email format before storing.
- Encrypt the entire file for extra security (e.g., using `cryptography.fernet`).
- Avoid printing sensitive info to the console.

Task Description #3 (Transparency in Algorithm Design)

Objective: Use AI to generate an Armstrong number checking function

with comments and explanations.

Instructions:

1. Ask AI to explain the code line-by-line.

2. Compare the explanation with code functionality.

Expected Output:

• Transparent, commented code.

• Correct, easy-to-understand explanation.

Prompt

Code

def is_armstrong(number): """ Check if a number is an Armstrong number. An Armstrong number is equal to the sum of its own digits each raised to the power of the number of digits.

```
Example: 153 -> 1^3 + 5^3 + 3^3 = 153 (Armstrong)
"""

# Convert the number to string to iterate over digits
digits = str(number)

# Count the number of digits
num_digits = len(digits)

# Initialize sum to 0
sum_of_powers = 0

# Loop through each digit and add its power to sum
for digit in digits:
    sum_of_powers += int(digit) ** num_digits  # raise digit to power
of num_digits

# Check if the sum equals the original number
if sum_of_powers == number:
    return True
else:
    return False
```

output

153 is an Armstrong number

Explanation

| Line / Section | Explanation |
| --- | --- |

| | |
|---|---|
| `def is_armstrong(number):` | Defines a function named `is_armstrong` that takes a number as input. |
| `digits = str(number)` | Converts the number into a string so we can loop over each digit easily. |
| `num_digits = len(digits)` | Counts how many digits the number has, which is used as the exponent. |
| `sum_of_powers = 0` | Initializes a variable to store the sum of each digit raised to the power of `num_digits`. |
| `for digit in digits:` | Loops through each character (digit) in the number. |
| `sum_of_powers += int(digit) ** num_digits` | Converts the character back to integer, raises it to the power of the number of digits, and adds it to `sum_of_powers`. |
| `if sum_of_powers == number:` | Compares the calculated sum with the original number. |
| `return True / False` | Returns True if it is an Armstrong number, otherwise False. |
| `num = 153 ... print(...)` | Example usage: checks if 153 is an Armstrong number and prints the result. |

## Key Points for Transparency

1. **Comments explain every step**—from converting digits to summing powers.
2. **Docstring** explains the purpose and definition of Armstrong numbers.
3. **Variable names** are meaningful (`digits`, `num_digits`, `sum_of_powers`).
4. **Function is modular**—easy to test with any number.

Task Description #4 (Transparency in Algorithm Comparison)

Task: Use AI to implement two sorting algorithms (e.g., QuickSort and

BubbleSort).

Prompt:

"Generate Python code for QuickSort and BubbleSort, and include

comments explaining step-by-step how each works and where they differ."

Expected Output:

• Code for both algorithms.

• Transparent, comparative explanation of their logic and efficiency.

Code

```
def bubble_sort(arr):
  """
  Bubble Sort repeatedly swaps adjacent elements if they are in the wrong order.
  It is simple but inefficient for large lists.
  """
  n = len(arr)

  # Traverse through all array elements
  for i in range(n):
    # Last i elements are already in place, no need to check them
    for j in range(0, n - i - 1):
      # Swap if the element found is greater than the next element
      if arr[j] > arr[j + 1]:
        arr[j], arr[j + 1] = arr[j + 1], arr[j]  # swap

# Example usage
arr1 = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr1)
print("BubbleSort Result:", arr1)
```

Output

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

**Step-by-step explanation:**

1. Outer loop goes through each element.
2. Inner loop compares adjacent elements.
3. If a pair is in the wrong order, swap them.
4. After each outer iteration, the largest unsorted element "bubbles up" to its correct position.
5. Time Complexity: **$O(n^2)$** in worst case, **$O(n)$** in best case (already sorted).
6. Space Complexity: **$O(1)$** (in-place).

- **Understanding the Algorithms:**
- **BubbleSort** is simple and intuitive: it repeatedly swaps adjacent elements until the list is sorted.
- **QuickSort** uses a divide-and-conquer approach: it partitions the list around a pivot and recursively sorts sublists.
- **Efficiency and Practical Use:**
- BubbleSort has **$O(n^2)$** time complexity and is inefficient for large datasets.
- QuickSort has **$O(n \log n)$** average time complexity, making it suitable for large datasets.
- **Memory Usage and Implementation Differences:**
- BubbleSort is **in-place** and uses minimal extra memory.
- QuickSort requires extra memory for recursion and partitioned lists (unless implemented in-place).
- **Stability:**
- BubbleSort is **stable**, preserving the relative order of equal elements.
- QuickSort is **not inherently stable**, unless modified.

Task Description #5 (Transparency in AI Recommendations)

Task: Use AI to create a product recommendation system.

Prompt:

"Generate a recommendation system that also provides reasons for each

suggestion."

Expected Output:

• Code with explainable recommendations.

• Evaluation of whether explanations are understandable.

Code

# Sample dataset: users and products they liked

user_preferences = { "Alice": ["Laptop", "Headphones", "Smartphone"], "Bob": ["Laptop", "Smartwatch", "Camera"], "Carol": ["Camera", "Headphones", "Smartphone"], "David": ["Smartwatch", "Laptop", "Headphones"] }

def recommend_products(target_user): """ Recommend products to the target_user based on what similar users liked. Provides explanations for each recommendation. """ if target_user not in user_preferences: return [], "User not found in dataset."

```
target_products = set(user_preferences[target_user])
recommendation_scores = {}
explanations = {}

# Compare with other users
for user, products in user_preferences.items():
    if user == target_user:
        continue
    overlap = target_products.intersection(products)  # products in
common
    score = len(overlap)

    for product in products:
        if product not in target_products:
            # Increase score based on similarity
            recommendation_scores[product] =
recommendation_scores.get(product, 0) + score
            # Add explanation
            explanations[product] = explanations.get(product, []) +
[f"Also liked by {user}, who shares {score} similar products with
you."]

# Sort recommendations by score
sorted_recommendations = sorted(recommendation_scores.items(),
key=lambda x: x[1], reverse=True)
```

```
# Prepare readable output
final_recommendations = []
for product, score in sorted_recommendations:
    explanation_text = " ".join(explanations[product])
    final_recommendations.append((product, explanation_text))

return final_recommendations
```

Output

Recommendations for Alice:

- Smartwatch: Also liked by Bob, who shares 1 similar products with you. Also liked by David, who shares 2 similar products with you.

- Camera: Also liked by Bob, who shares 1 similar products with you. Also liked by Carol, who shares 2 similar products with you.

Explanation

- **Transparent Recommendations:**
- The system explicitly states which **users influenced the suggestion** and **how many shared preferences** contributed to the score.
- **Understandability:**
- Simple natural-language explanations make it clear **why a product is suggested**.
- Example: "Also liked by Bob, who shares 1 similar products with you."
- **Limitations / Improvements:**
- Currently works for **small datasets**; large datasets may need **similarity metrics** (cosine similarity, Pearson correlation).
- Explanations could include **categories, ratings, or trends** for even better clarity.
- Could integrate **weighting by similarity** or **popularity** to make recommendations more accurate.

.