Assignment 5.1

Name: P. Umesh Reddy   Roll.no: 2403A510F9

Batch.no: 06   course: AI Assisted coding

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.

```
import requests

API_KEY = "your_insecure_api_key_here"  # ⚠ Hardcoded API key
CITY = "London"

def get_weather(city_name):
    """Fetches current weather data for a specified city."""
    base_url = "http://api.openweathermap.org/data/2.5/weather"
    params = {
        "q": city_name,
        "appid": API_KEY,
        "units": "metric"
    }
    try:
        response = requests.get(base_url, params=params)
        response.raise_for_status()  # Raises an HTTPError for bad
            responses (4xx or 5xx)
        return response.json()
    except requests.exceptions.HTTPError as err:
        return f"HTTP Error: {err}"
    except requests.exceptions.RequestException as err:
```

```
        return f"Error: {err}"

weather_data = get_weather(CITY)
print(weather_data)
Secure Version using Environment Variables
This version uses the os module to access an environment variable
    named WEATHER_API_KEY. It also uses the python-dotenv library,
    which is a popular and easy way to manage environment variables
    for local development.

Secure Code
Python

# Secure version using environment variables

import os
import requests
from dotenv import load_dotenv

# Load environment variables from a .env file
load_dotenv()
```

```
# Access the API key from an environment variable
API_KEY = os.getenv("WEATHER_API_KEY")

if not API_KEY:
    raise RuntimeError("WEATHER_API_KEY environment variable is not
        set.")

CITY = "London"

def get_weather(city_name):
    """Fetches current weather data for a specified city."""
    base_url = "http://api.openweathermap.org/data/2.5/weather"
    params = {
        "q": city_name,
        "appid": API_KEY,
        "units": "metric"
    }
    try:
        response = requests.get(base_url, params=params)
        response.raise_for_status()  # Raises an HTTPError for bad
```

```
            responses (4xx or 5xx)
        return response.json()
    except requests.exceptions.HTTPError as err:
        return f"HTTP Error: {err}"
    except requests.exceptions.RequestException as err:
        return f"Error: {err}"

weather_data = get_weather(CITY)
print(weather_data)
```

Explanation

1. The `os` Module 📂

Python's built-in `os` module provides a way to interact with the operating system, including accessing environment variables. The `os.getenv()` function retrieves the value of a specified environment variable. If the variable isn't found, it returns `None`, which is a safer default than an error.

2. The `.env` File 📝

To use this code locally, you'll need to create a file named `.env` in the root directory of your project. This file is a simple text file that stores your key-value pairs. You should add this file to your `.gitignore` file to ensure it's never committed to version control.

Example `.env` file:

```
WEATHER_API_KEY=your_actual_api_key_here
```

3. `python-dotenv` 📦

The `python-dotenv` library reads the key-value pairs from the `.env` file and adds them to your system's environment variables for the duration of the script. You can install it with pip: `pip install python-dotenv`. This library simplifies the process and makes local development seamless.

How It Works Together 🤝

- `load_dotenv()` reads the `.env` file.
- `os.getenv("WEATHER_API_KEY")` then looks up the `WEATHER_API_KEY` from the loaded environment variables.
- The rest of the code functions as before, but the API key is never visible in the script itself. This method is considered a security best practice for managing secrets in software development.

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)

This code is a basic, but highly insecure, way to save user data. It writes the user's name, email, and password directly to a file without any protection.

Python

```python
# WARNING: This code is INSECURE.
# It stores sensitive data, including passwords, in plain text.

def save_user(username, email, password):
    """Saves user data to a file in plain text."""
    with open("users.txt", "a") as f:
        f.write(f"Username: {username}\n")
        f.write(f"Email: {email}\n")
        f.write(f"Password: {password}\n")
        f.write("-" * 20 + "\n")
    print("User data saved (insecurely).")
```

```
# Example usage (DO NOT USE IN PRODUCTION)
save_user("alice", "alice@example.com", "password123")
```

Revised Version with Encrypted Password Storage

This version uses the `bcrypt` library, a modern and highly recommended password hashing algorithm. It's designed to be slow and resistant to brute-force attacks, making it a secure choice. The other user data (name and email) is less sensitive and can be stored in plain text, but the password is a hash.

Installation

First, you'll need to install the `bcrypt` library:

Bash

```bash
pip install bcrypt
```

Secure Code

Python

```python
import bcrypt

def save_user_securely(username, email, password):
    """
```

```
    Saves user data with a securely hashed password.

    The password is a one-way hash, so it cannot be decrypted.
    This protects against data breaches.
    """
    # 1. Convert the password string to bytes, which bcrypt requires
    password_bytes = password.encode('utf-8')

    # 2. Generate a salt and hash the password
    # bcrypt automatically generates a salt and incorporates it into the hash
    hashed_password = bcrypt.hashpw(password_bytes, bcrypt.gensalt())

    # 3. Decode the hash to a string for storage
    hashed_password_str = hashed_password.decode('utf-8')

    # Save the data (excluding the plain text password)
```

Python

```python
    with open("users_secure.txt", "a") as f:
        f.write(f"Username: {username}\n")
        f.write(f"Email: {email}\n")
        f.write(f"Hashed Password: {hashed_password_str}\n")
        f.write("-" * 20 + "\n")
    print("User data saved securely.")

# Example usage
save_user_securely("bob", "bob@example.com", "strong_password!123")
```

Explanation

- `bcrypt.gensalt()`: This function generates a **salt**, which is a random string of data. The salt is combined with the password before it's hashed. This is crucial because it ensures that even if two users have the same password, their hashes will be different. This prevents passwords) to crack multiple accounts at once.

- `bcrypt.hashpw()`: This function takes the password (as bytes) and the salt, then performs the hashing. The result is a secure hash that includes the algorithm, cost factor, and the salt itself, all encoded into a single string. This is what you save to the file.
- **One-Way Function**: The beauty of this process is that the hash cannot be reversed. When a user tries to log in, you retrieve their stored hash, and then use `bcrypt.checkpw()` to verify their password. This function re-hashes the provided password with the stored hash's salt and compares the two results. Since the original password is never stored, a breach of the user file doesn't expose any actual passwords.

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

Armstrong Number Checker 🔢

An **Armstrong number** (also known as a narcissistic number) is a number that is equal to the sum of its own digits, each raised to the power of the number of digits. For example, 153 is an Armstrong number because $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$.

Code

Python

```python
def is_armstrong(number):
    """
    Checks if a number is an Armstrong number.

    An Armstrong number is one that equals the sum of its own digits,
    each raised to the power of the total number of digits.
```

```python
    Args:
        number (int): The integer to check.

    Returns:
        bool: True if it's an Armstrong number, False otherwise.
    """
    # 1. Convert the number to a string to easily find the number of digits.
    #    The length of the string is the power to which each digit will be raised.
    num_str = str(number)
    num_digits = len(num_str)

    # 2. Initialize a variable to store the sum of the powered digits.
    sum_of_powers = 0

    # 3. Iterate through each digit of the number.
    #    We convert the string back to an integer for the calculation.
```

```python
    for digit_char in num_str:
        digit = int(digit_char)

        # 4. Calculate the power of the digit and add it to the sum.
        #    We use the exponentiation operator (**) for this.
        sum_of_powers += digit ** num_digits

    # 5. Compare the final sum with the original number.
    #    If they are equal, the number is an Armstrong number.
    return sum_of_powers == number

# Example usage:
print(f"Is 153 an Armstrong number? {is_armstrong(153)}")
print(f"Is 1634 an Armstrong number? {is_armstrong(1634)}")
print(f"Is 123 an Armstrong number? {is_armstrong(123)}")
```

Line-by-Line Explanation

1. `def is_armstrong(number):`
    - This line defines a function named `is_armstrong` that takes one argument, `number`. Functions are reusable blocks of code.
2. `num_str = str(number)`
    - The input `number` (which is an integer) is converted into a string and stored in the variable `num_str`. This conversion is necessary so we can easily determine the number of digits and iterate through each one.
3. `num_digits = len(num_str)`
    - The `len()` function is used to find the length of the string `num_str`. This gives us the total count of digits in the original number. For 153, `num_digits` will be 3.

4. `sum_of_powers = 0`
    - A variable named `sum_of_powers` is initialized to zero. This variable will accumulate the sum of each digit raised to the power of `num_digits`.
5. `for digit_char in num_str:`
    - This is a `for` loop that iterates through each character in the string `num_str`. In the case of "153", the loop will run three times, with `digit_char` taking the values '1', '5', and '3' in sequence.
6. `digit = int(digit_char)`
    - Inside the loop, each character (`digit_char`) is converted back into an integer and stored in the `digit` variable. This allows us to perform mathematical operations on it.

7. `sum_of_powers += digit ** num_digits`
    - This is the core calculation. `digit` is raised to the power of `num_digits` using the exponentiation operator `**`. The result is then added to the running total stored in `sum_of_powers`.
8. `return sum_of_powers == number`
    - After the loop completes, the function compares the final `sum_of_powers` with the `original number` that was passed into the function. It returns a boolean value (`True` or `False`) based on whether the two values are equal. This is the final result of the check.

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

```
# BubbleSort implementation
def bubble_sort(arr):
    # Get the length of the array
    n = len(arr)
    # Loop through all array elements
    for i in range(n):
        # Flag to check if any swaps were made in the inner loop
        swapped = False
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Compare adjacent elements
            if arr[j] > arr[j + 1]:
                # Swap them if they are in the wrong order
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # If no two elements were swapped by the inner loop, then the
            array is sorted
        if not swapped:
            break
    return arr
```

```python
# QuickSort implementation
def quick_sort(arr):
    # Call the helper function to perform the quick sort
    quick_sort_helper(arr, 0, len(arr) - 1)
    return arr

def quick_sort_helper(arr, low, high):
    # Check if there are elements to sort
    if low < high:
        # Partition the array and get the pivot index
        pivot_index = partition(arr, low, high)
        # Recursively sort the sub-array before the pivot
        quick_sort_helper(arr, low, pivot_index - 1)
        # Recursively sort the sub-array after the pivot
        quick_sort_helper(arr, pivot_index + 1, high)

def partition(arr, low, high):
    # Choose the rightmost element as the pivot
    pivot = arr[high]
    # Index of the smaller element
    i = low - 1
```

```python
# Loop through the array from low to high-1
for j in range(low, high):
    # If the current element is smaller than the pivot
    if arr[j] < pivot:
        # Increment the index of the smaller element
        i = i + 1
        # Swap the elements
        arr[i], arr[j] = arr[j], arr[i]
# Place the pivot element at its correct sorted position
arr[i + 1], arr[high] = arr[high], arr[i + 1]
# Return the index of the pivot
return i + 1
```

Key Differences in Logic and Efficiency

Feature	BubbleSort	QuickSort
Logic	**Comparison-based** with an iterative approach. It repeatedly passes through the list, swapping adjacent elements.	**Divide and Conquer.** It partitions the array and recursively sorts sub-arrays.
Efficiency	**Time complexity of $O(n^2)$.** Inefficient for large datasets due to its brute-force approach of repeated passes.	**Average-case time complexity of $O(n \log n)$.** Highly efficient and one of the fastest sorting algorithms.
Mechanism	It sorts by **repeatedly swapping adjacent elements**, with larger elements "bubbling up" to their correct position.	It sorts by **partitioning the array** around a pivot and then recursively applying the same process to the sub-arrays.
Space Complexity	$O(1)$, as it sorts in-place.	$O(\log n)$ on average due to the recursive call stack.
Best-Case	$O(n)$ if the array is already sorted, due to the optimization of breaking the loop if no swaps occur.	$O(n \log n)$
Worst-Case	$O(n^2)$ when the array is in reverse order.	$O(n^2)$ when the pivot selection consistently leads to imbalanced partitions. This is a rare occurrence with a good pivot strategy.

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.

```python
# A simple explainable product recommendation system using content-based filtering.
# This system recommends products based on shared features, making the reasons transparent.

class ProductRecommender:
    """
    A class to generate product recommendations with explanations.
    The system uses a simple content-based filtering approach.
    """
    def __init__(self, products):
        """
        Initializes the recommender with a list of products.

        Args:
            products (list of dict): A list of dictionaries, where each dict
                                     represents a product with its features.
                                     Example: {'name': 'Echo Dot', 'category': 'Electronics', 'tags': ['Smart Home', 'Voice Assistant']}
        """
        self.products = products

    def recommend(self, user_history, num_recommendations=3):
        """
        Generates a list of recommended products with explanations.

        Args:
            user_history (list of dict): A list of products the user has liked or purchased.
            num_recommendations (int): The number of top recommendations to return.

        Returns:
            list of dict: A list of dictionaries, each containing a 'product' and
                          an 'explanation' for the recommendation.
        """
```

```python
# Step 1: Gather all unique features from the user's history.
user_features = set()
for product in user_history:
    if 'category' in product:
        user_features.add(f"category: {product['category']}")
    if 'tags' in product:
        user_features.update(f"tag: {tag}" for tag in product['tags'])

# If no user history is provided, return an empty list
if not user_features:
    return []

recommendations = []

# Step 2: Iterate through all available products.
for product in self.products:
    # Step 3: Skip products the user has already interacted with.
    if product in user_history:
        continue

    # Step 4: Compare product features with user features.
    score = 0
    reasons = []

    # Check for shared category
    if 'category' in product and f"category: {product['category']}" in user_features:
        score += 2  # Give category a higher weight
        reasons.append(f"a product in the '{product['category']}' category")

    # Check for shared tags
    if 'tags' in product:
```

```python
                shared_tags = [tag for tag in product['tags'] if f"tag: {tag}" in user_features]
                score += len(shared_tags)
                if shared_tags:
                    reasons.append(f"products with the following tags: {', '.join(shared_tags)}")

            # Step 5: If the product has a non-zero score, add it to the recommendations list.
            if score > 0:
                # Create a clear explanation string
                explanation = f"Because you liked {' and '.join(reasons)}."
                recommendations.append({
                    'product': product,
                    'score': score,
                    'explanation': explanation
                })

        # Step 6: Sort recommendations by score in descending order.
        recommendations.sort(key=lambda x: x['score'], reverse=True)

        # Step 7: Return the top N recommendations.
        return recommendations[:num_recommendations]

# --- Example Usage ---

# Define a list of products
all_products = [
    {'name': 'Echo Dot', 'category': 'Electronics', 'tags': ['Smart Home', 'Voice Assistant']},
    {'name': 'Google Nest', 'category': 'Electronics', 'tags': ['Smart Home', 'Voice Assistant']},
    {'name': 'Smart Light Bulb', 'category': 'Electronics', 'tags': ['Smart Home', 'Lighting']},
    {'name': 'Laptop', 'category': 'Electronics', 'tags': ['Computers']},
    {'name': 'Gaming Mouse', 'category': 'Accessories', 'tags': ['PC Gaming']},
    {'name': 'Bluetooth Speaker', 'category': 'Audio', 'tags': ['Portable', 'Electronics']},
```

```python
    {'name': 'Cookbook', 'category': 'Books', 'tags': ['Cooking', 'Recipes']},
    {'name': 'Headphones', 'category': 'Audio', 'tags': ['Portable', 'Noise Cancelling']}
]

# Create an instance of the recommender
recommender = ProductRecommender(all_products)

# Define a user's past purchases
user_history_1 = [
    {'name': 'Echo Dot', 'category': 'Electronics', 'tags': ['Smart Home', 'Voice Assistant']},
    {'name': 'Smart Light Bulb', 'category': 'Electronics', 'tags': ['Smart Home', 'Lighting']}
]

# Get recommendations for the user
recommendations = recommender.recommend(user_history_1)

print("--- Recommendations for User 1 ---")
for rec in recommendations:
    print(f"Product: {rec['product']['name']}")
    print(f"Explanation: {rec['explanation']}\n")

# Example for a different user
user_history_2 = [
    {'name': 'Gaming Mouse', 'category': 'Accessories', 'tags': ['PC Gaming']}
]

recommendations_2 = recommender.recommend(user_history_2)
```

```
print("--- Recommendations for User 2 ---")
for rec in recommendations_2:
    print(f"Product: {rec['product']['name']}")
    print(f"Explanation: {rec['explanation']}\n")
```

Evaluation of Explanation Understandability

The core strength of this system's explanations is that they are **directly tied to the data and logic** used to make the recommendation. The system is a simple example of a **content-based filtering model**, meaning it recommends products that are similar to what the user has already liked. This approach naturally lends itself to clear explanations.

When the code recommends "Google Nest," the reason given is, "Because you liked products in the 'Electronics' category and with the 'Smart Home' tag." This explanation is easy for a user to understand because it points to concrete, relatable features. It's a simple, logical `if-then` statement: "If you like X, then you might like Y, because Y is like X." This contrasts with more complex machine learning models, which are often considered "black boxes" because their internal logic is too complex to easily translate into a human-readable explanation.

The simplicity of the model is its greatest asset for transparency. While it might not be as accurate or nuanced as a complex recommendation engine used by a major e-commerce site, its transparency is unmatched. The user doesn't have to guess *why* they're seeing a particular suggestion. The explanation provides a clear path of reasoning. The explanations are not just a label but a breakdown of the specific features that led to the match, which empowers the user and builds trust in the system.

Task Description #6 (Transparent Code Generation)
Task: Ask AI to generate a Python function for calculating factorial using recursion.
Prompt:
"Generate a recursive factorial function with comments that explain each line and a final summary of the algorithm's flow."
Expected Output:
• Fully commented code.
• Clear documentation of how recursion works.

```python
def factorial(n):
    """
    Calculates the factorial of a non-negative integer using recursion.

    Args:
        n (int): The number to calculate the factorial of. Must be a non-negative integer.

    Returns:
        int: The factorial of the number n.
    """
    # Base Case: This is the stopping condition for the recursion.
    # The factorial of 0 or 1 is 1. Without this, the function would call itself
    # indefinitely, leading to an infinite loop and a "RecursionError".
    if n == 0 or n == 1:
        return 1
    # Recursive Step: This is where the function calls itself.
    # The factorial of n is defined as n multiplied by the factorial of (n-1).
    # The function breaks down the problem into a smaller, simpler sub-problem.
    else:
        return n * factorial(n - 1)
```

```python
# Example Usage:
number = 5
result = factorial(number)
print(f"The factorial of {number} is {result}")

# Let's trace the execution for factorial(5):
# factorial(5) returns 5 * factorial(4)
#   factorial(4) returns 4 * factorial(3)
#     factorial(3) returns 3 * factorial(2)
#       factorial(2) returns 2 * factorial(1)
#         factorial(1) returns 1  (This is the base case!)
# The calls then return back up the chain:
#       ...2 * 1 = 2
#     ...3 * 2 = 6
#   ...4 * 6 = 24
# ...5 * 24 = 120
```

Algorithm Flow Summary

The `factorial()` function follows a clear recursive pattern:

1. **Base Case Check:** The function first checks if the input `n` is `0` or `1`. This is the **most critical part of the algorithm.** Since the factorial of $0!$ and $1!$ is 1, this condition provides a definite answer without any further recursion. It's the point where the recursion "unwinds" and the function starts returning values.
2. **Recursive Step:** If the input is not the base case, the function calls itself with a smaller argument, `n - 1`. It then multiplies the original number `n` by the result of this recursive call. This continues until the base case is reached.

3. **Unwinding the Recursion:** Once the base case (`factorial(1)`) returns the value `1`, the call stack starts to "unwind." Each previous function call receives the result from the one below it, performs its multiplication, and returns its own result to the call above. This process continues all the way back up to the initial call, at which point the final factorial value is returned. The code comments illustrate this unwinding process for `factorial(5)`, showing how the problem is broken down and then solved piece by piece.

Task Description #7 (Inclusiveness in Customer Support)
Code Snippet:

```
def support_reply(name, gender):
    if gender.lower() == "male":
        prefix = "Mr."
    else:
        prefix = "Mrs."
    return f"Dear {prefix} {name}, we have resolved your i
```

Task:
Regenerate the code so that support messages use neutral language (e.g.,
“Dear {name}”) and optionally accept preferred titles.
Expected Output:
• Neutral, user-friendly support responses.

```python
def support_reply(name, title=None):
    """
    Generates a neutral support reply message.

    Args:
        name (str): The customer's name.
        title (str, optional): Preferred title (e.g., Dr., Prof., Mr.,
            Ms.). Defaults to None.

    Returns:
        str: A polite, user-friendly support message.
    """
    if title:
        greeting = f"Dear {title} {name}"
    else:
        greeting = f"Dear {name}"

    return f"{greeting}, we have resolved your issue. Thank you for
        reaching out to our support team."
```

Explanation:

- **Neutral by default:** If no title is provided, the message is simply `"Dear {name}"`.
- **Optional title:** If the user wants to be addressed as `"Dr. Smith"` or `"Prof. Lee"`, they can provide it in the `title` argument.
- **Polite closing:** Maintains a professional, user-friendly tone.

Example Usage:

python Copy Edit

```python
print(support_reply("Alex"))
# Dear Alex, we have resolved your issue. Thank you for reaching out to our support team.

print(support_reply("Jordan", title="Dr."))
# Dear Dr. Jordan, we have resolved your issue. Thank you for reaching out to our support team.
```