AI ASSISTED CODING WEEK-5.1

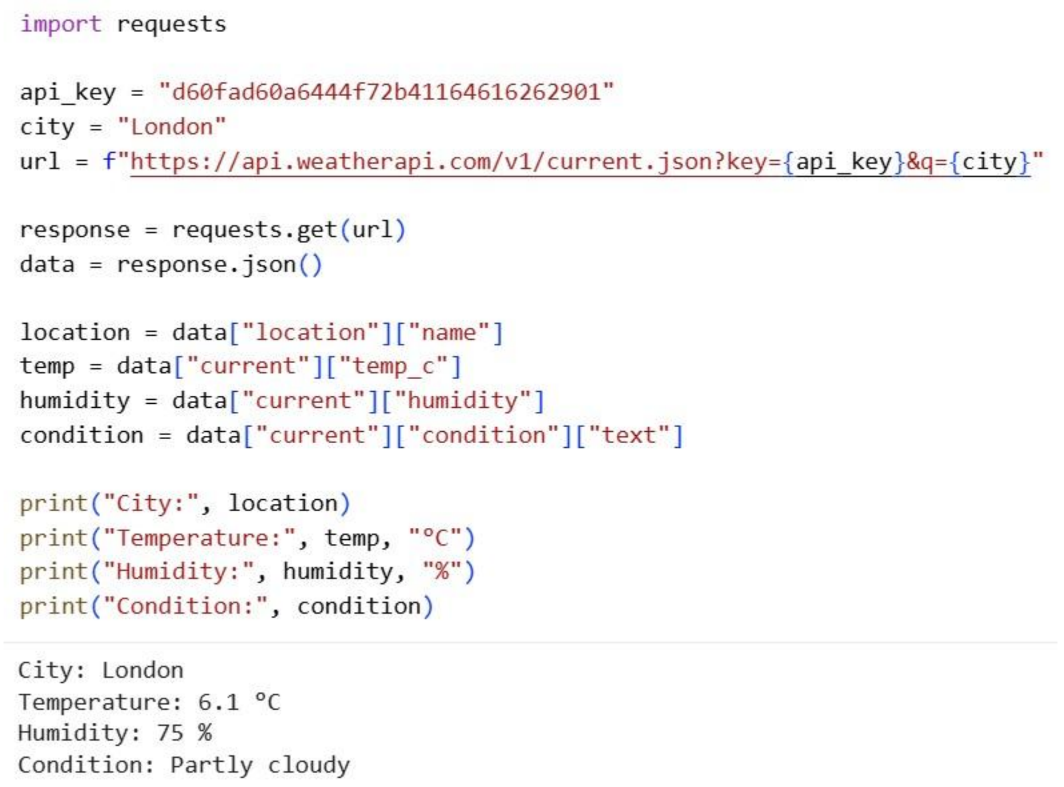
2303A510H2

B-27

Task Description #1 (Privacy in API Usage)

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

Original AI Generated Code (Insecure):



```python
import requests

api_key = "d60fad60a6444f72b41164616262901"
city = "London"
url = f"https://api.weatherapi.com/v1/current.json?key={api_key}&q={city}"

response = requests.get(url)
data = response.json()

location = data["location"]["name"]
temp = data["current"]["temp_c"]
humidity = data["current"]["humidity"]
condition = data["current"]["condition"]["text"]

print("City:", location)
print("Temperature:", temp, "°C")
print("Humidity:", humidity, "%")
print("Condition:", condition)
```

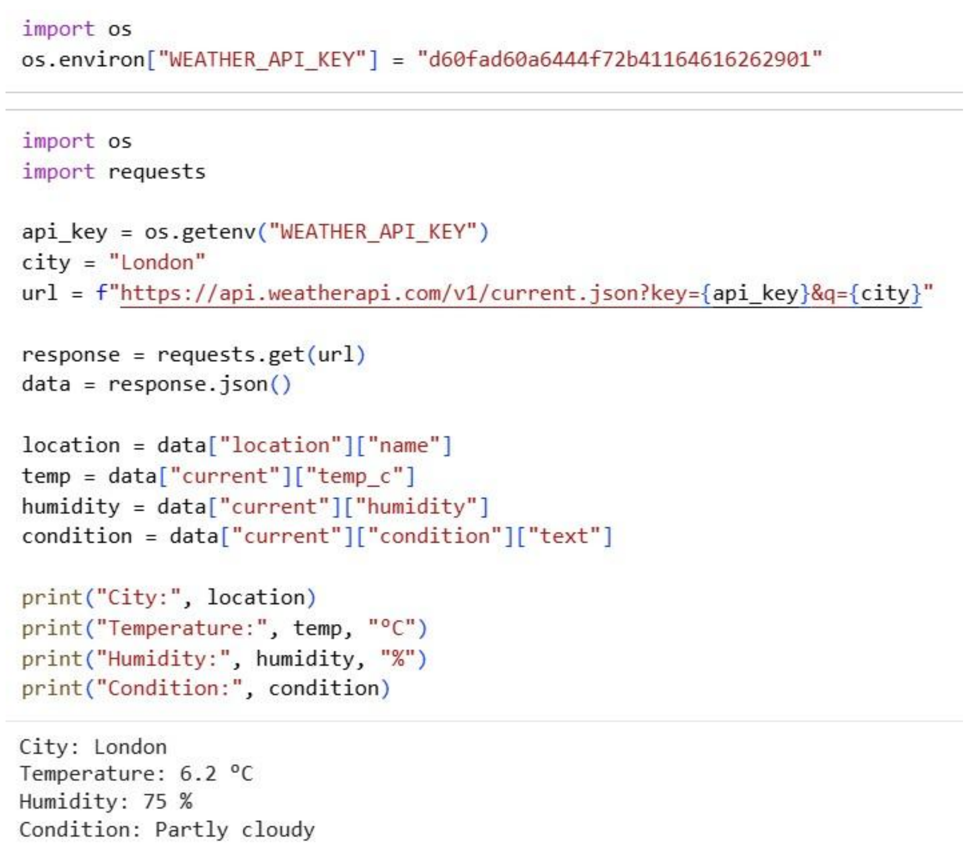
```
City: London
Temperature: 6.1 °C
Humidity: 75 %
Condition: Partly cloudy
```

This version hardcodes the API key directly in the source code, which is a privacy and security risk.

Why this is a problem

- API key is visible to anyone who accesses the code
- Risk of key leakage if pushed to GitHub or shared
- Violates basic API security practices

Secure Version Using Environment Variables:



```python
import os
os.environ["WEATHER_API_KEY"] = "d60fad60a6444f72b41164616262901"
```

```python
import os
import requests

api_key = os.getenv("WEATHER_API_KEY")
city = "London"
url = f"https://api.weatherapi.com/v1/current.json?key={api_key}&q={city}"

response = requests.get(url)
data = response.json()

location = data["location"]["name"]
temp = data["current"]["temp_c"]
humidity = data["current"]["humidity"]
condition = data["current"]["condition"]["text"]

print("City:", location)
print("Temperature:", temp, "°C")
print("Humidity:", humidity, "%")
print("Condition:", condition)
```

```
City: London
Temperature: 6.2 °C
Humidity: 75 %
Condition: Partly cloudy
```

This version keeps the API key outside the code and loads it securely from environment variables.

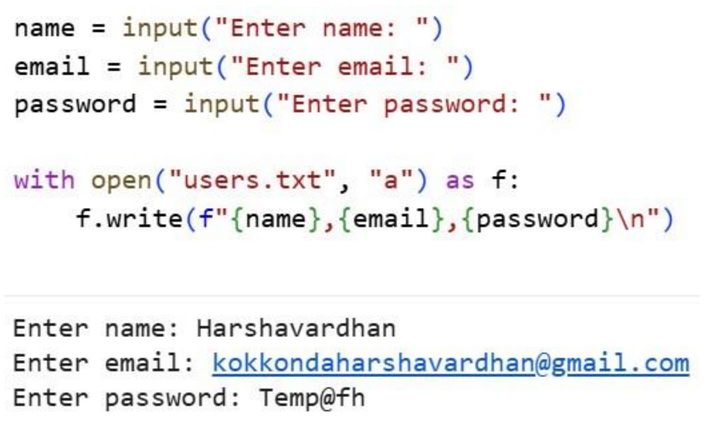
How this improves privacy

- API key is never written in the source code
- Safe to share and push to repositories
- Follows industry best practices

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.

AI-Generated Script (Insecure):

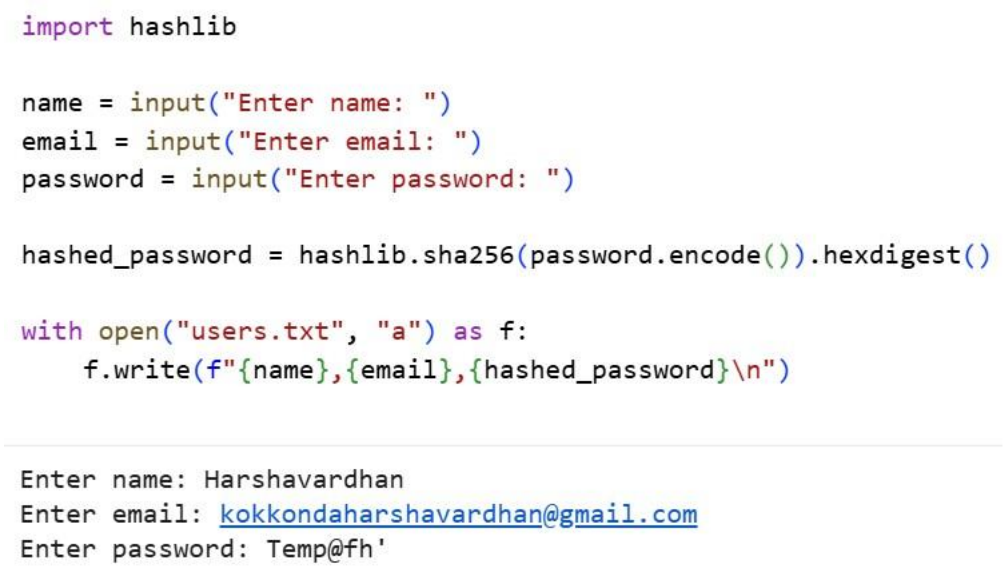


```python
name = input("Enter name: ")
email = input("Enter email: ")
password = input("Enter password: ")

with open("users.txt", "a") as f:
    f.write(f"{name},{email},{password}\n")
```

```
Enter name: Harshavardhan
Enter email: kokkondaharshavardhan@gmail.com
Enter password: Temp@fh
```

This is a **typical AI output** that directly stores user data in a file.

Revised Secure Version (Password Hashing):



```python
import hashlib

name = input("Enter name: ")
email = input("Enter email: ")
password = input("Enter password: ")

hashed_password = hashlib.sha256(password.encode()).hexdigest()

with open("users.txt", "a") as f:
    f.write(f"{name},{email},{hashed_password}\n")
```

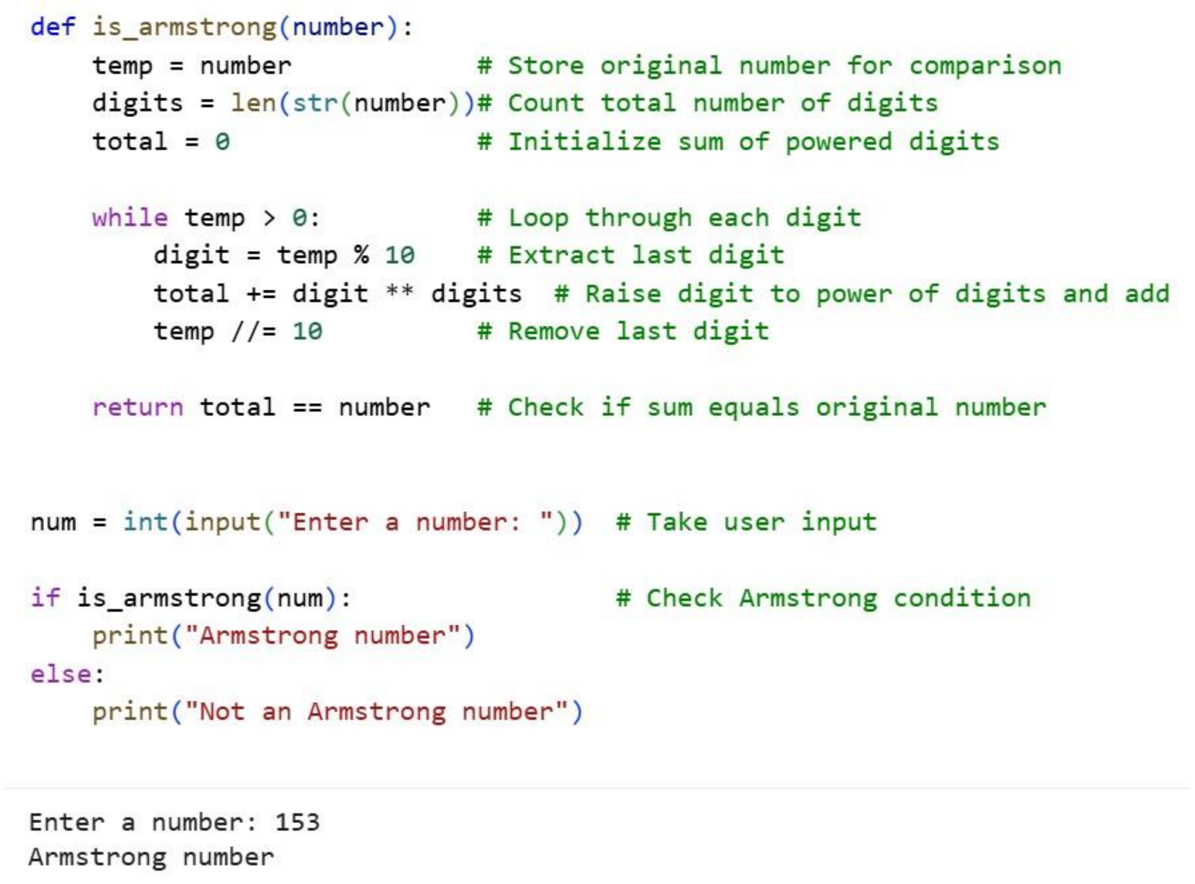
```
Enter name: Harshavardhan
Enter email: kokkondaharshavardhan@gmail.com
Enter password: Temp@fh'
```

Instead of storing the actual password, we store a **hashed value**.
Even if the file is leaked, original passwords cannot be recovered easily.

Task Description #3 (Transparency in Algorithm Design)

Objective: Use AI to generate an Armstrong number checking function with comments and explanations.

AI-Generated Armstrong Number Program (With Comments):



```python
def is_armstrong(number):
    temp = number           # Store original number for comparison
    digits = len(str(number))# Count total number of digits
    total = 0               # Initialize sum of powered digits

    while temp > 0:         # Loop through each digit
        digit = temp % 10   # Extract last digit
        total += digit ** digits  # Raise digit to power of digits and add
        temp //= 10         # Remove last digit

    return total == number  # Check if sum equals original number


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

if is_armstrong(num):                 # Check Armstrong condition
    print("Armstrong number")
else:
    print("Not an Armstrong number")
```

```
Enter a number: 153
Armstrong number
```

Line-by-Line Explanation (Transparent):

def is_armstrong(number):

Defines a function to check whether a given number is an Armstrong number.

temp = number

Stores the original number in a temporary variable so it is not modified.

digits = len(str(number))

Counts how many digits are in the number. This is required for the Armstrong

calculation. **total = 0**

Initializes a variable to store the sum of powered digits.

while temp > 0:

Loops through each digit of the number.

digit = temp % 10

Extracts the last digit of the number.

total += digit ** digits

Raises the digit to the power of total digits and adds it to the sum.

temp //= 10

Removes the last digit from the number.

return total == number

Compares the calculated sum with the original number and returns True or

False. **num = int(input("Enter a number: "))** Takes user input.

if is_armstrong(num):

Calls the function to check if the number is an Armstrong

number. **print("Armstrong number") else:**

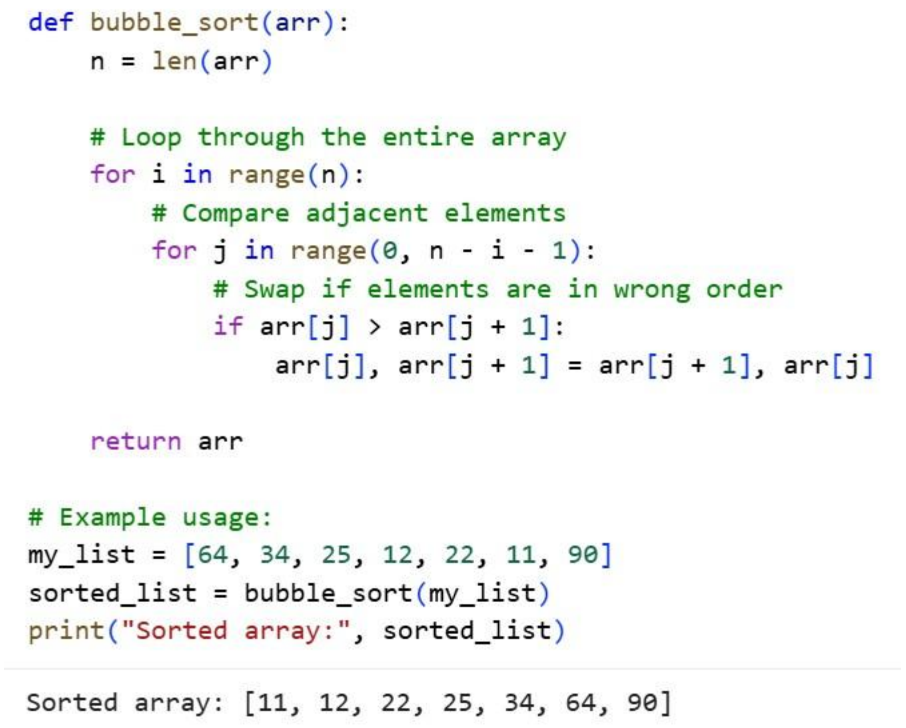
print("Not an Armstrong number")

Prints the result based on the function output.

Task Description #4 (Transparency in Algorithm Comparison)

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

Bubble Sort (With Step-by-Step Comments):



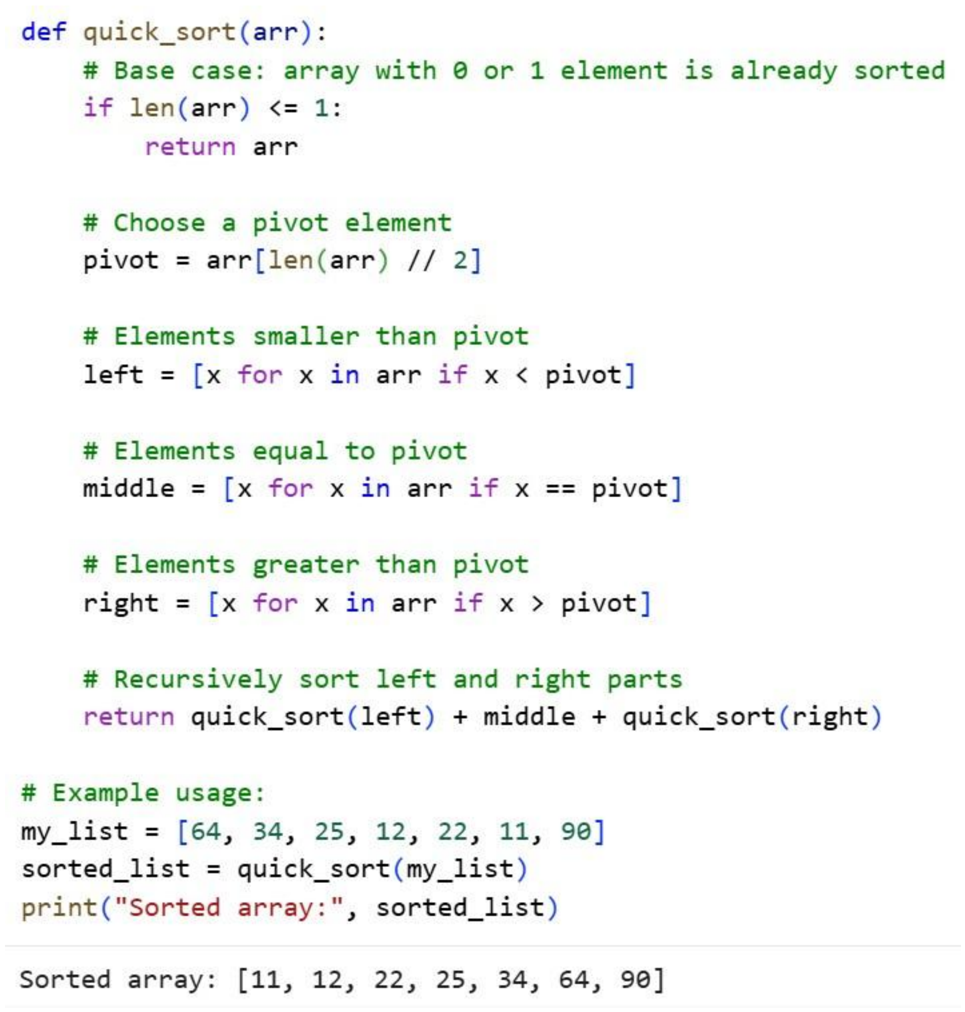
```
def bubble_sort(arr):
    n = len(arr)

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

    return arr

# Example usage:
my_list = [64, 34, 25, 12, 22, 11, 90]
sorted_list = bubble_sort(my_list)
print("Sorted array:", sorted_list)
```

```
Sorted array: [11, 12, 22, 25, 34, 64, 90]
```

How Bubble Sort Works

- Repeatedly compares **adjacent elements**
- Swaps them if they are in the wrong order
- Largest elements “bubble up” to the end
- After each pass, one element is in its correct position

Quick Sort (With Step-by-Step Comments):



```python
def quick_sort(arr):
    # Base case: array with 0 or 1 element is 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)

# Example usage:
my_list = [64, 34, 25, 12, 22, 11, 90]
sorted_list = quick_sort(my_list)
print("Sorted array:", sorted_list)
```

```
Sorted array: [11, 12, 22, 25, 34, 64, 90]
```

How Quick Sort Works

- Selects a **pivot element**
- Divides the array into smaller elements and larger elements
- Recursively sorts subarrays
- Combines the sorted parts to get the final result

Transparency & Efficiency Analysis:

- Bubble Sort is **easy to understand** but inefficient
- Quick Sort is **more complex** but significantly faster

- Bubble Sort shows algorithm behavior clearly
- Quick Sort demonstrates optimization through recursion and partitioning
- The comments explain every step, ensuring transparency

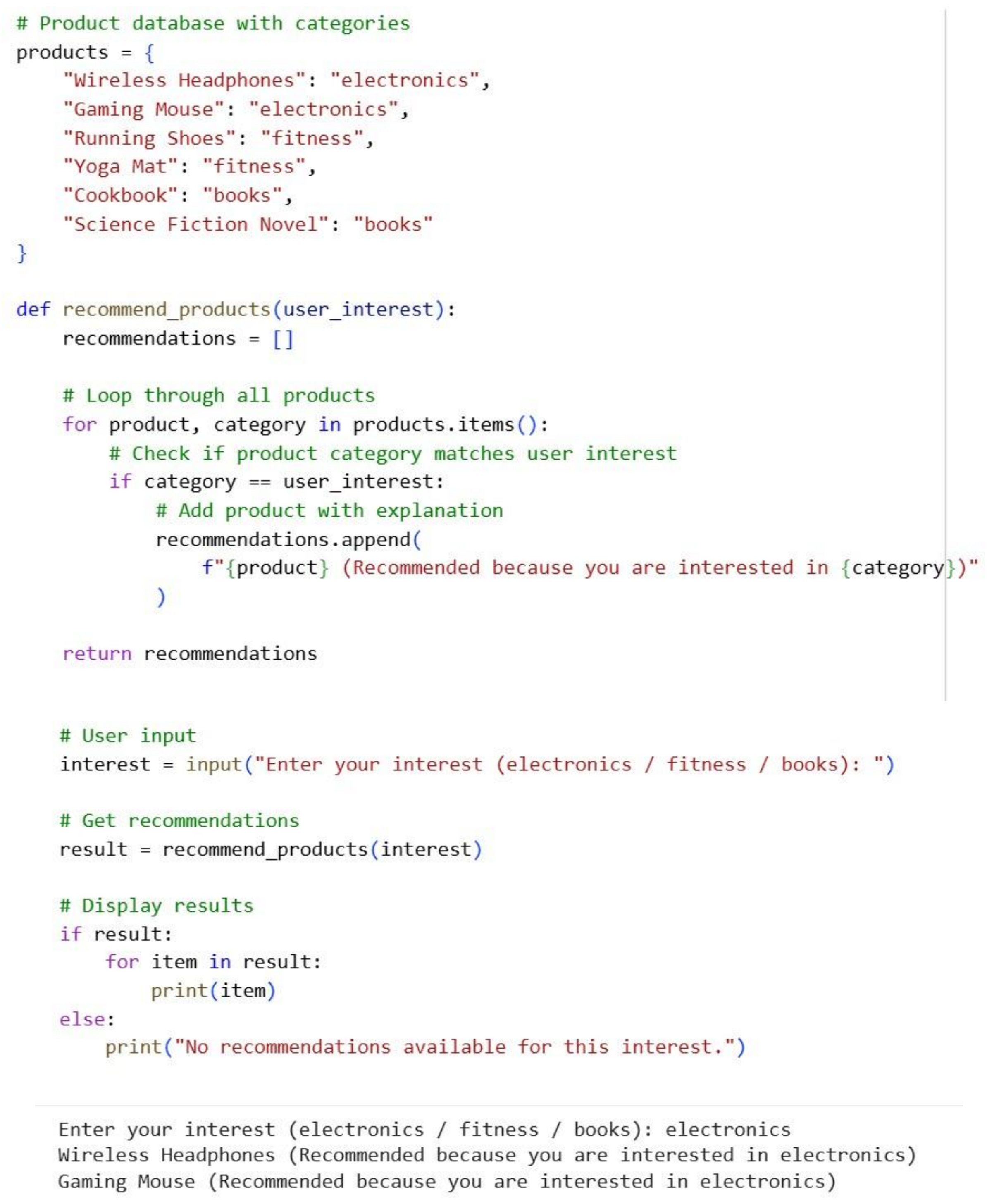
Comparison: Bubble Sort vs Quick Sort:

Feature	**Bubble Sort**	**Quick Sort**
Approach	Repeated swap	Divide and conquer
Compari	Adjacent eleme	Based on pivot
Speed	Very slow for lar	Very fast on average
Best Cas	O(n)	O(n log n)
Average (	O(n^2)	O(n log n)
Worst Ca	O(n^2)	O(n^2)
Practical	Small datasets,	Large datasets, real systems

Task Description #5 (Transparency in AI Recommendations)

Task: Use AI to create a product recommendation system.

AI-Generated Recommendation System (Explainable):



```python
# Product database with categories
products = {
    "Wireless Headphones": "electronics",
    "Gaming Mouse": "electronics",
    "Running Shoes": "fitness",
    "Yoga Mat": "fitness",
    "Cookbook": "books",
    "Science Fiction Novel": "books"
}

def recommend_products(user_interest):
    recommendations = []

    # Loop through all products
    for product, category in products.items():
        # Check if product category matches user interest
        if category == user_interest:
            # Add product with explanation
            recommendations.append(
                f"{product} (Recommended because you are interested in {category})"
            )

    return recommendations


    # User input
    interest = input("Enter your interest (electronics / fitness / books): ")

    # Get recommendations
    result = recommend_products(interest)

    # Display results
    if result:
        for item in result:
            print(item)
    else:
        print("No recommendations available for this interest.")
```

```
Enter your interest (electronics / fitness / books): electronics
Wireless Headphones (Recommended because you are interested in electronics)
Gaming Mouse (Recommended because you are interested in electronics)
```

How the Recommendation Works (Transparent Explanation):

- Products are grouped by **category** •   User provides an **interest category**
- The system:

  - Matches products with the same category
  - Recommends only relevant items
  - Clearly states the **reason** for each recommendation
- No hidden logic or complex scoring is used Every recommendation includes:

“Recommended because you are interested in X”

This makes the system **explainable and transparent**.

Transparency Summary:

- The recommendation system provides suggestions based on user interests.
- Each recommendation includes a clear explanation for transparency.
- The logic is straightforward and understandable.
- Users can easily verify how recommendations are generated.
- This approach avoids opaque AI decision-making.