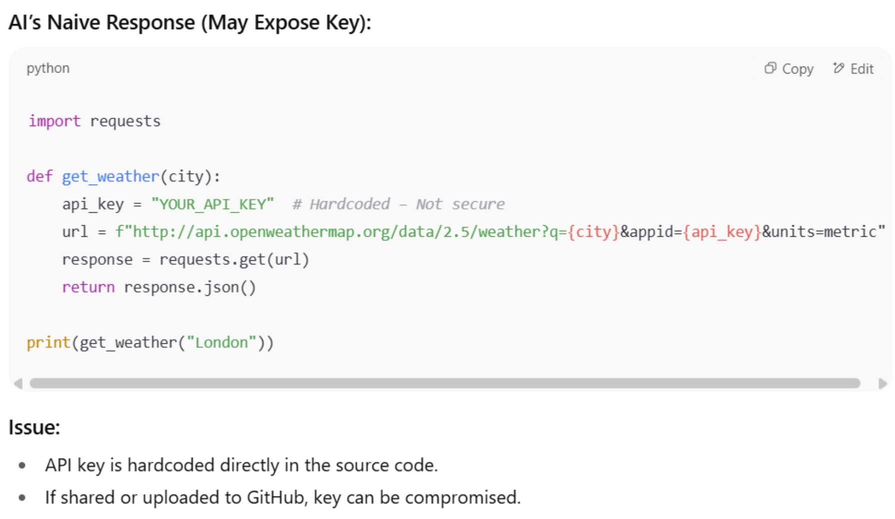
ASSIGNMENT-5.1

Name: G.Sudeeksha **Hall-Ticket No: 2403a510E0**

Batch No: 05 **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."
Output:
• Original AI code (check if keys are hardcoded).
• Secure version using environment variables

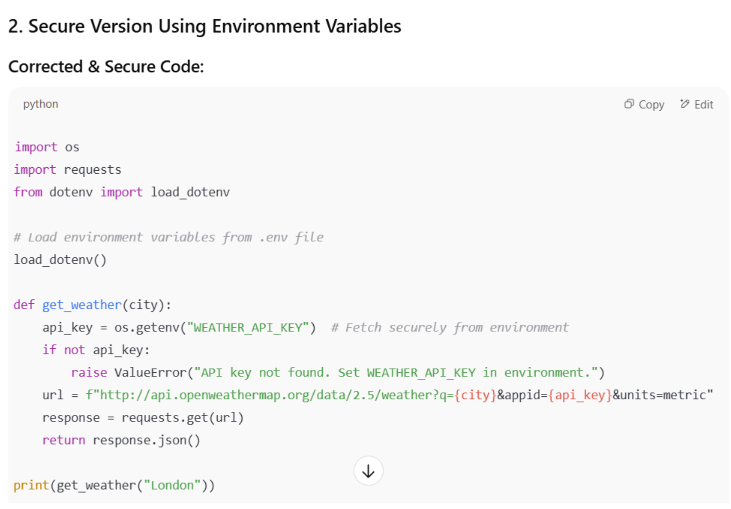


AI's Naive Response (May Expose Key):

```python
import requests

def get_weather(city):
    api_key = "YOUR_API_KEY"  # Hardcoded – Not secure
    url = f"http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&units=metric"
    response = requests.get(url)
    return response.json()

print(get_weather("London"))
```

Issue:

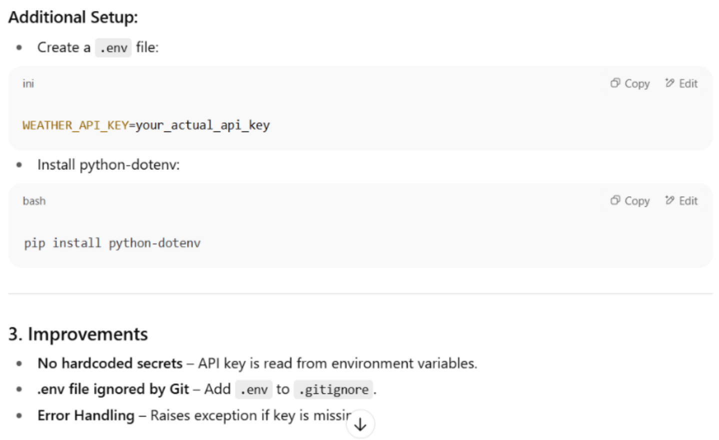
- API key is hardcoded directly in the source code.
- If shared or uploaded to GitHub, key can be compromised.



2. Secure Version Using Environment Variables

Corrected & Secure Code:

```python
import os
import requests
from dotenv import load_dotenv

# Load environment variables from .env file
load_dotenv()

def get_weather(city):
    api_key = os.getenv("WEATHER_API_KEY")  # Fetch securely from environment
    if not api_key:
        raise ValueError("API key not found. Set WEATHER_API_KEY in environment.")
    url = f"http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&units=metric"
    response = requests.get(url)
    return response.json()

print(get_weather("London"))
```



Additional Setup:

- Create a `.env` file:

```ini
WEATHER_API_KEY=your_actual_api_key
```

- Install python-dotenv:

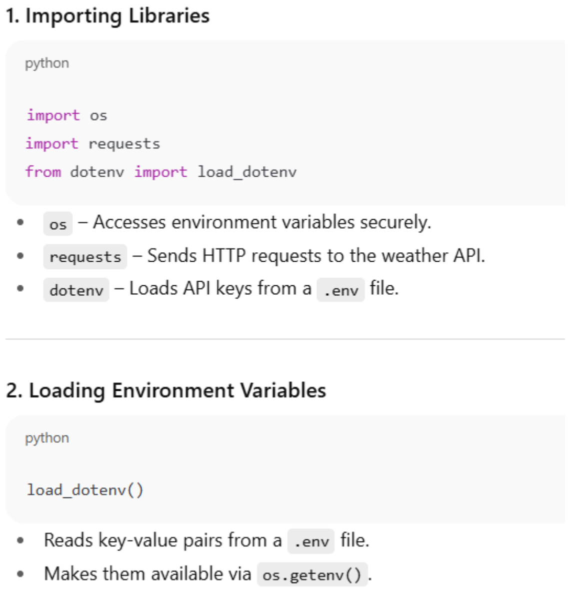
```bash
pip install python-dotenv
```

---

3. Improvements

- **No hardcoded secrets** – API key is read from environment variables.
- **.env file ignored by Git** – Add `.env` to `.gitignore`.
- **Error Handling** – Raises exception if key is missir

Explanation:



1. Importing Libraries

```python
import os
import requests
from dotenv import load_dotenv
```

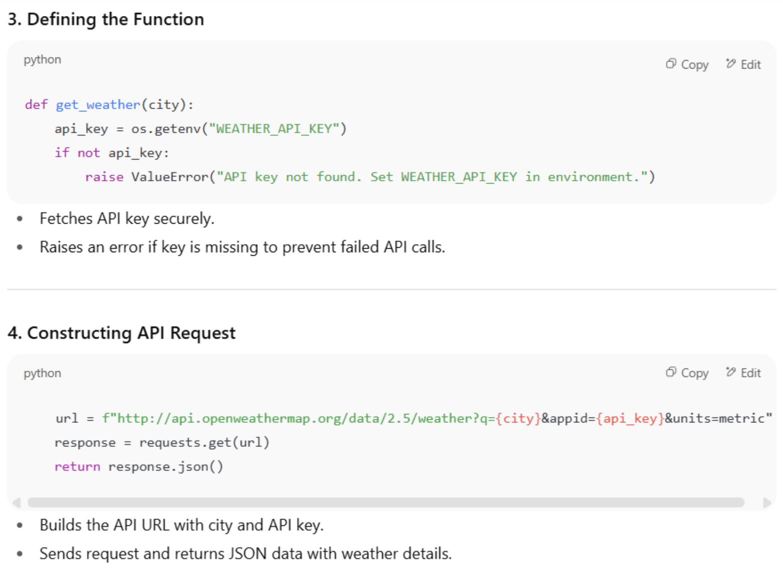
- `os` – Accesses environment variables securely.
- `requests` – Sends HTTP requests to the weather API.
- `dotenv` – Loads API keys from a `.env` file.

---

2. Loading Environment Variables

```python
load_dotenv()
```

- Reads key-value pairs from a `.env` file.
- Makes them available via `os.getenv()`.



3. Defining the Function

```python
def get_weather(city):
    api_key = os.getenv("WEATHER_API_KEY")
    if not api_key:
        raise ValueError("API key not found. Set WEATHER_API_KEY in environment.")
```

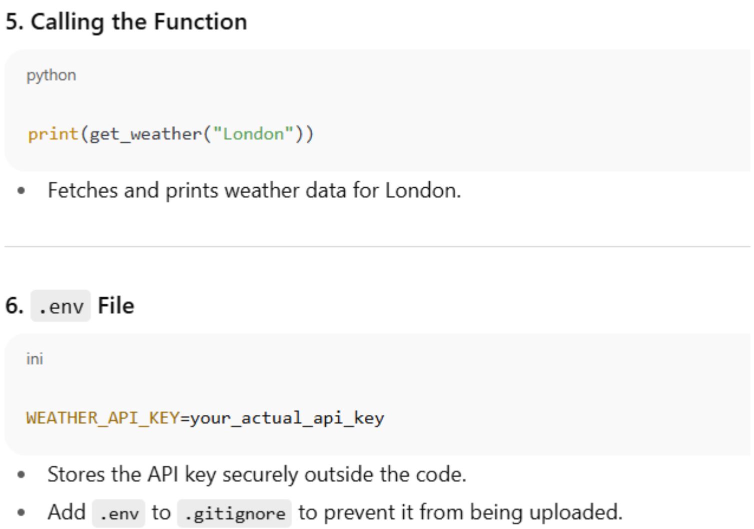
- Fetches API key securely.
- Raises an error if key is missing to prevent failed API calls.

---

4. Constructing API Request

```python
    url = f"http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&units=metric"
    response = requests.get(url)
    return response.json()
```

- Builds the API URL with city and API key.
- Sends request and returns JSON data with weather details.



5. Calling the Function

```python
print(get_weather("London"))
```

- Fetches and prints weather data for London.

---

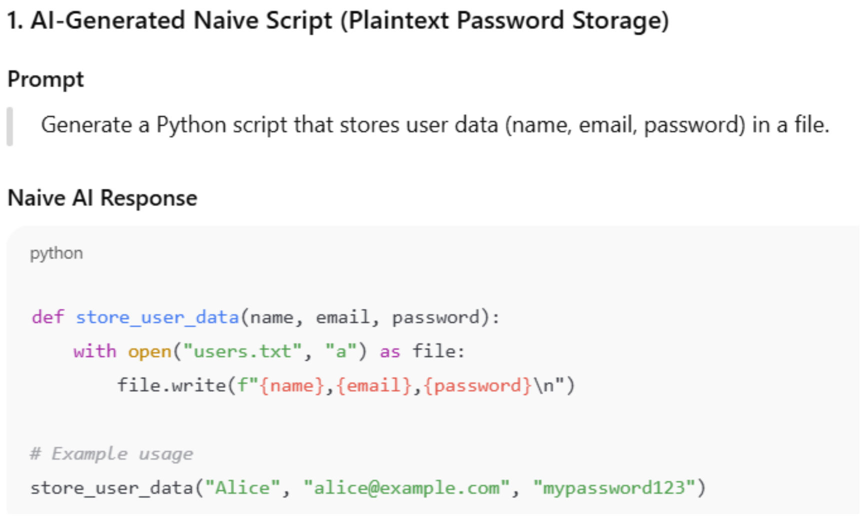
6. `.env` File

```ini
WEATHER_API_KEY=your_actual_api_key
```

- Stores the API key securely outside the code.
- Add `.env` to `.gitignore` to prevent it from being uploaded.

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.
Output:
• Identified privacy risks.
• Revised version with encrypted password storage (e.g., hashing)



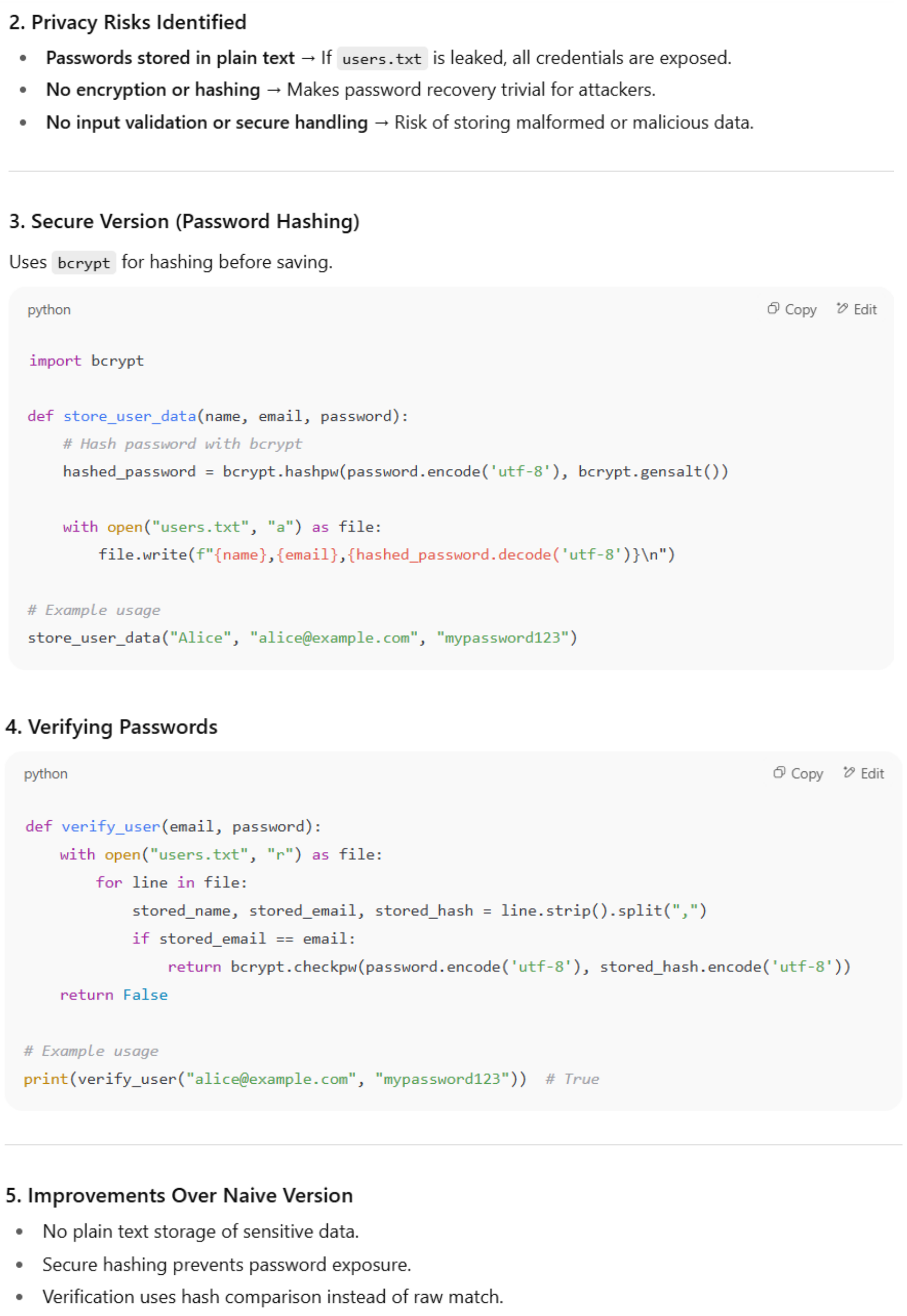
1. AI-Generated Naive Script (Plaintext Password Storage)

Prompt

> Generate a Python script that stores user data (name, email, password) in a file.

Naive AI Response

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

# Example usage
store_user_data("Alice", "alice@example.com", "mypassword123")
```



2. Privacy Risks Identified

- **Passwords stored in plain text** → If `users.txt` is leaked, all credentials are exposed.
- **No encryption or hashing** → Makes password recovery trivial for attackers.
- **No input validation or secure handling** → Risk of storing malformed or malicious data.

---

3. Secure Version (Password Hashing)

Uses `bcrypt` for hashing before saving.

```python
import bcrypt

def store_user_data(name, email, password):
    # Hash password with bcrypt
    hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

    with open("users.txt", "a") as file:
        file.write(f"{name},{email},{hashed_password.decode('utf-8')}\n")

# Example usage
store_user_data("Alice", "alice@example.com", "mypassword123")
```

4. Verifying Passwords

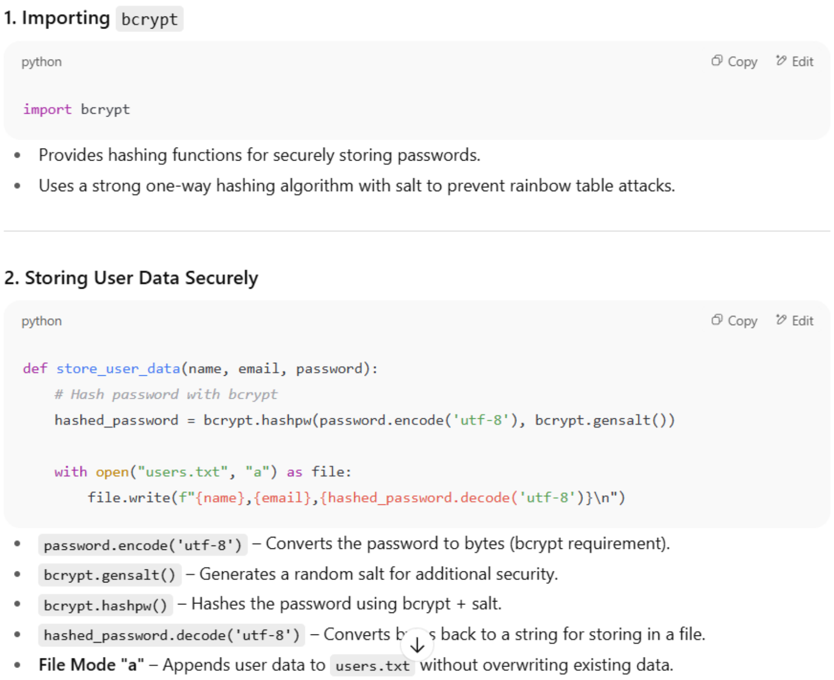
```python
def verify_user(email, password):
    with open("users.txt", "r") as file:
        for line in file:
            stored_name, stored_email, stored_hash = line.strip().split(",")
            if stored_email == email:
                return bcrypt.checkpw(password.encode('utf-8'), stored_hash.encode('utf-8'))
    return False

# Example usage
print(verify_user("alice@example.com", "mypassword123"))  # True
```

---

5. Improvements Over Naive Version

- No plain text storage of sensitive data.
- Secure hashing prevents password exposure.
- Verification uses hash comparison instead of raw match.

Explanation:



1. Importing `bcrypt`

```python
import bcrypt
```

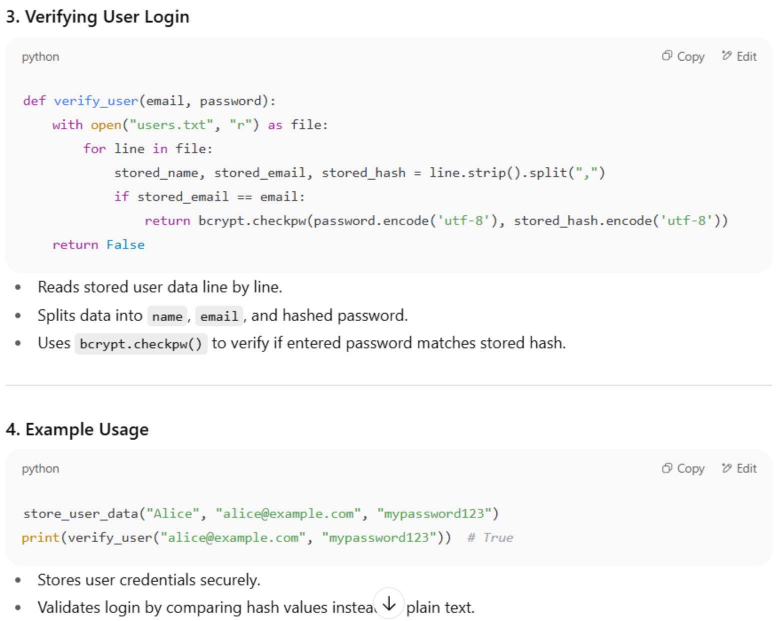
- Provides hashing functions for securely storing passwords.
- Uses a strong one-way hashing algorithm with salt to prevent rainbow table attacks.

---

2. Storing User Data Securely

```python
def store_user_data(name, email, password):
    # Hash password with bcrypt
    hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

    with open("users.txt", "a") as file:
        file.write(f"{name},{email},{hashed_password.decode('utf-8')}\n")
```

- `password.encode('utf-8')` – Converts the password to bytes (bcrypt requirement).
- `bcrypt.gensalt()` – Generates a random salt for additional security.
- `bcrypt.hashpw()` – Hashes the password using bcrypt + salt.
- `hashed_password.decode('utf-8')` – Converts b↓s back to a string for storing in a file.
- **File Mode "a"** – Appends user data to `users.txt` without overwriting existing data.



3. Verifying User Login

```python
def verify_user(email, password):
    with open("users.txt", "r") as file:
        for line in file:
            stored_name, stored_email, stored_hash = line.strip().split(",")
            if stored_email == email:
                return bcrypt.checkpw(password.encode('utf-8'), stored_hash.encode('utf-8'))
    return False
```

- Reads stored user data line by line.
- Splits data into `name`, `email`, and hashed password.
- Uses `bcrypt.checkpw()` to verify if entered password matches stored hash.

---

4. Example Usage

```python
store_user_data("Alice", "alice@example.com", "mypassword123")
print(verify_user("alice@example.com", "mypassword123"))  # True
```

- Stores user credentials securely.
- Validates login by comparing hash values instea↓ plain text.

Task Description #3

(Transparency in Algorithm Design)

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

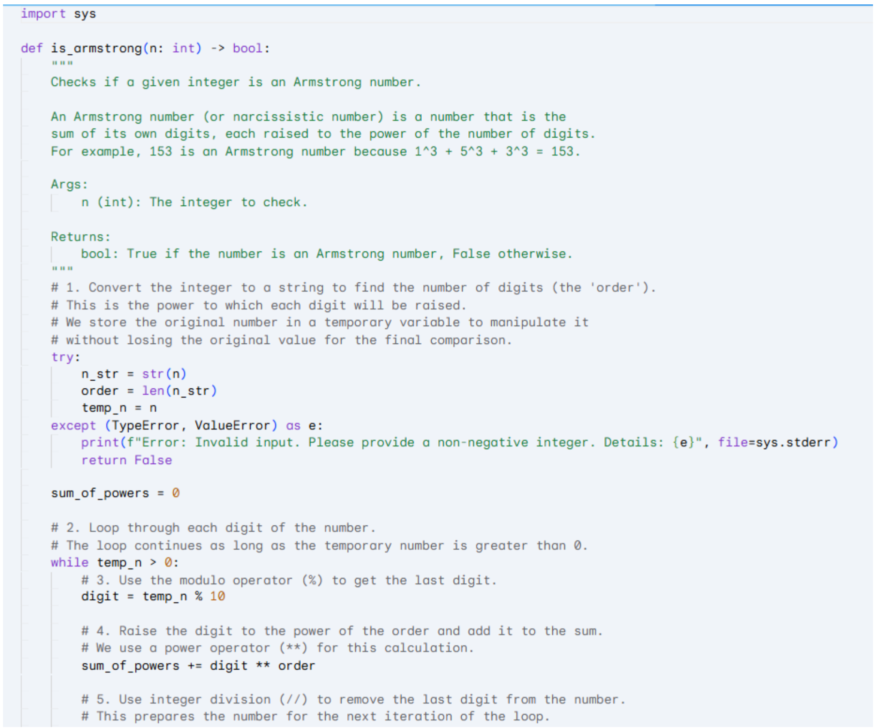
Instructions:

1. Ask AI to explain the code line-by-line.
2. Compare the explanation with code functionality.

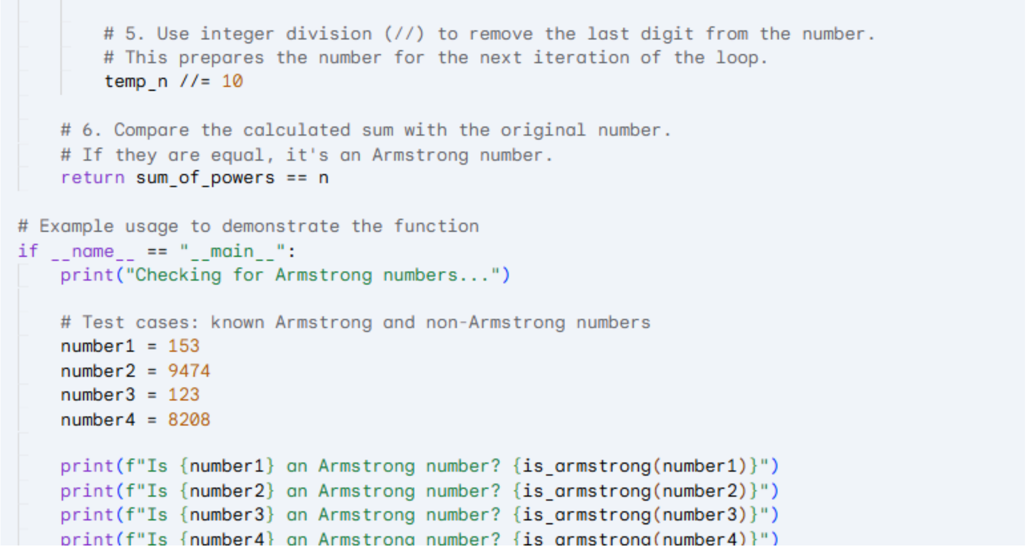
Output:

• Transparent, commented code.

• Correct, easy-to-understand explanation



```
import sys

def is_armstrong(n: int) -> bool:
    """
    Checks if a given integer is an Armstrong number.

    An Armstrong number (or narcissistic number) is a number that is 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 = 153.

    Args:
        n (int): The integer to check.

    Returns:
        bool: True if the number is an Armstrong number, False otherwise.
    """
    # 1. Convert the integer to a string to find the number of digits (the 'order').
    # This is the power to which each digit will be raised.
    # We store the original number in a temporary variable to manipulate it
    # without losing the original value for the final comparison.
    try:
        n_str = str(n)
        order = len(n_str)
        temp_n = n
    except (TypeError, ValueError) as e:
        print(f"Error: Invalid input. Please provide a non-negative integer. Details: {e}", file=sys.stderr)
        return False

    sum_of_powers = 0

    # 2. Loop through each digit of the number.
    # The loop continues as long as the temporary number is greater than 0.
    while temp_n > 0:
        # 3. Use the modulo operator (%) to get the last digit.
        digit = temp_n % 10

        # 4. Raise the digit to the power of the order and add it to the sum.
        # We use a power operator (**) for this calculation.
        sum_of_powers += digit ** order

        # 5. Use integer division (//) to remove the last digit from the number.
        # This prepares the number for the next iteration of the loop.
```



```
        # 5. Use integer division (//) to remove the last digit from the number.
        # This prepares the number for the next iteration of the loop.
        temp_n //= 10

    # 6. Compare the calculated sum with the original number.
    # If they are equal, it's an Armstrong number.
    return sum_of_powers == n

# Example usage to demonstrate the function
if __name__ == "__main__":
    print("Checking for Armstrong numbers...")

    # Test cases: known Armstrong and non-Armstrong numbers
    number1 = 153
    number2 = 9474
    number3 = 123
    number4 = 8208

    print(f"Is {number1} an Armstrong number? {is_armstrong(number1)}")
    print(f"Is {number2} an Armstrong number? {is_armstrong(number2)}")
    print(f"Is {number3} an Armstrong number? {is_armstrong(number3)}")
    print(f"Is {number4} an Armstrong number? {is_armstrong(number4)}")
```

Explanation:

The `is_armstrong()` Function

- `import sys`: This line imports the `sys` module, which is used here to write error messages to the standard error stream (`sys.stderr`) instead of the standard output.
- `def is_armstrong(n: int) -> bool:`: This defines the function. It takes one argument, `n`, which must be an integer (`int`), and is expected to return a boolean value (`bool`), either `True` or `False`.
- `try...except`: This block is for error handling. It attempts to convert the input `n` to a string (`str(n)`) and get its length (`len(n_str)`). This length is stored as the `order`. If the input is not a valid number (e.g., a string), it will raise a `TypeError` or `ValueError`, and the `except` block will print an error message and return `False`.
- `sum_of_powers = 0`: This variable is initialized to zero and will be used to accumulate the sum of the digits raised to the power of the order.
- `while temp_n > 0:`: This is the main calculation loop. It continues as long as `temp_n`, a temporary copy of the original number, is greater than zero. This ensures the loop processes every digit.
- `digit = temp_n % 10`: This line uses the modulo operator (`%`) to get the **last digit** of `temp_n`. For example, `153 % 10` gives `3`.
- `sum_of_powers += digit ** order`: Here, the `digit` is raised to the power of the `order` (`**` is the exponentiation operator) and added to the `sum_of_powers` variable.

- `temp_n //= 10`: This performs **integer division**, which effectively removes the last digit from `temp_n`. For example, `153 // 10` results in `15`, setting up the next loop iteration.
- `return sum_of_powers == n`: After the loop finishes, this line compares the final `sum_of_powers` with the original number `n`. If they are equal, the function returns `True`; otherwise, it returns `False`.

xample Usage

- `if __name__ == "__main__":`: This is a standard Python convention. It ensures that the code inside this block only runs when the script is executed directly, not when it's imported as a module into another program.
- The print statements in this section simply call the `is_armstrong()` function with a few different numbers to test and demonstrate its functionality.

Task Description #4 (Transparency in Algorithm Comparison)

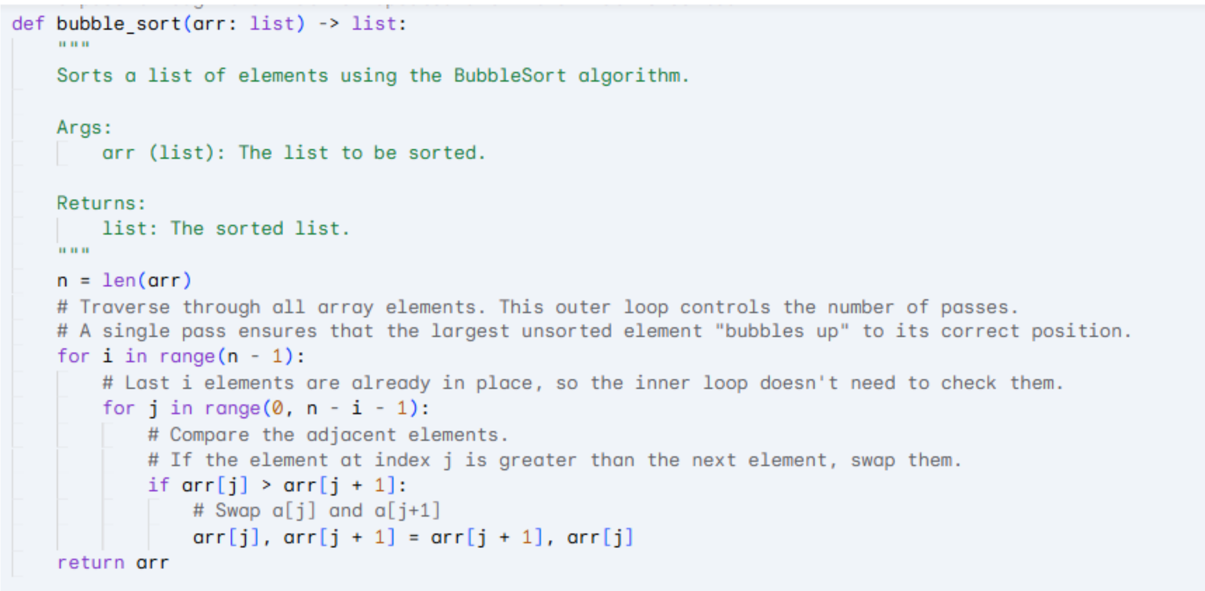
Task: Use AI to implement two sorting algorithms (e.g., QuickSort and BubbleSort).

Prompt:

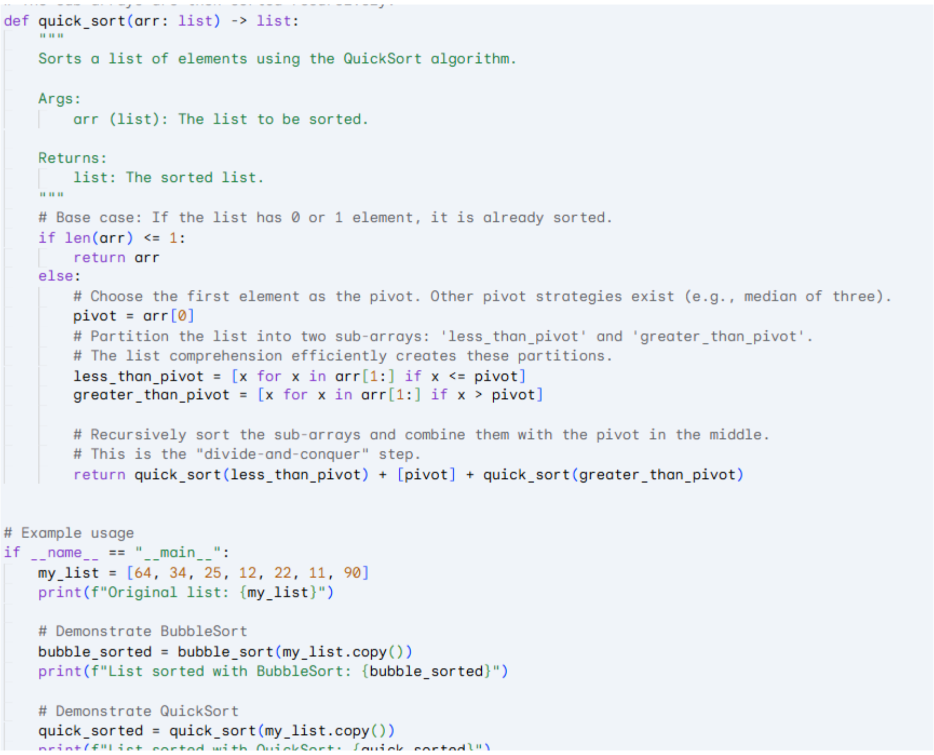
"Generate Python code for QuickSort and BubbleSort, and include comments explaining step-by-step how each works and where they differ."

Output:

- Code for both algorithms.
- Transparent, comparative explanation of their logic and efficiency.

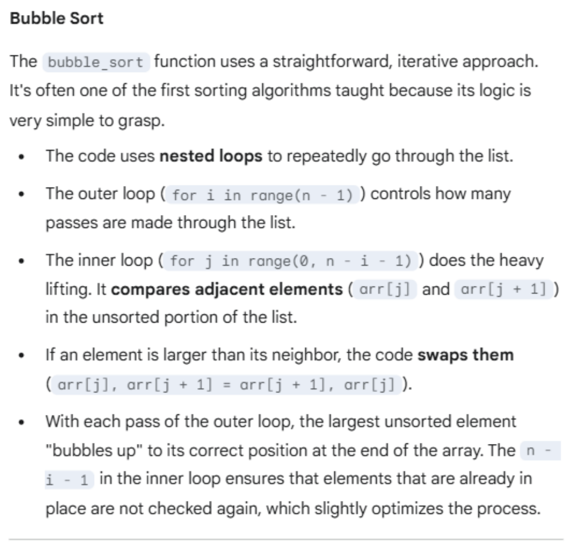


```python
def bubble_sort(arr: list) -> list:
    """
    Sorts a list of elements using the BubbleSort algorithm.

    Args:
        arr (list): The list to be sorted.

    Returns:
        list: The sorted list.
    """
    n = len(arr)
    # Traverse through all array elements. This outer loop controls the number of passes.
    # A single pass ensures that the largest unsorted element "bubbles up" to its correct position.
    for i in range(n - 1):
        # Last i elements are already in place, so the inner loop doesn't need to check them.
        for j in range(0, n - i - 1):
            # Compare the adjacent elements.
            # If the element at index j is greater than the next element, swap them.
            if arr[j] > arr[j + 1]:
                # Swap a[j] and a[j+1]
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```



```python
def quick_sort(arr: list) -> list:
    """
    Sorts a list of elements using the QuickSort algorithm.

    Args:
        arr (list): The list to be sorted.

    Returns:
        list: The sorted list.
    """
    # Base case: If the list has 0 or 1 element, it is already sorted.
    if len(arr) <= 1:
        return arr
    else:
        # Choose the first element as the pivot. Other pivot strategies exist (e.g., median of three).
        pivot = arr[0]
        # Partition the list into two sub-arrays: 'less_than_pivot' and 'greater_than_pivot'.
        # The list comprehension efficiently creates these partitions.
        less_than_pivot = [x for x in arr[1:] if x <= pivot]
        greater_than_pivot = [x for x in arr[1:] if x > pivot]

        # Recursively sort the sub-arrays and combine them with the pivot in the middle.
        # This is the "divide-and-conquer" step.
        return quick_sort(less_than_pivot) + [pivot] + quick_sort(greater_than_pivot)


# Example usage
if __name__ == "__main__":
    my_list = [64, 34, 25, 12, 22, 11, 90]
    print(f"Original list: {my_list}")

    # Demonstrate BubbleSort
    bubble_sorted = bubble_sort(my_list.copy())
    print(f"List sorted with BubbleSort: {bubble_sorted}")

    # Demonstrate QuickSort
    quick_sorted = quick_sort(my_list.copy())
    print(f"List sorted with QuickSort: {quick_sorted}")
```

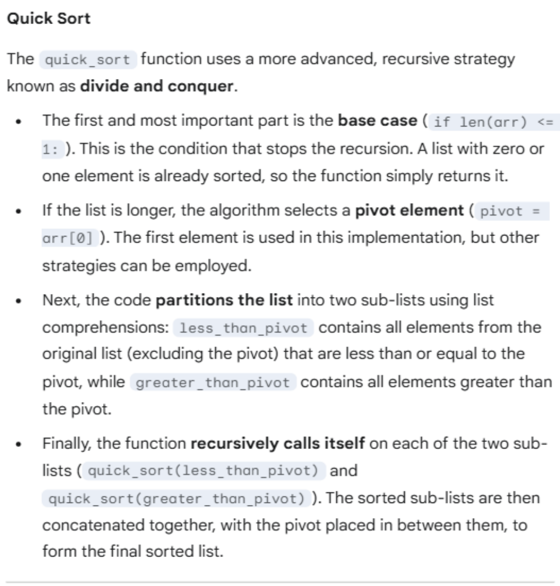
Explanation:



Bubble Sort

The `bubble_sort` function uses a straightforward, iterative approach. It's often one of the first sorting algorithms taught because its logic is very simple to grasp.

- The code uses **nested loops** to repeatedly go through the list.
- The outer loop (`for i in range(n - 1)`) controls how many passes are made through the list.
- The inner loop (`for j in range(0, n - i - 1)`) does the heavy lifting. It **compares adjacent elements** (`arr[j]` and `arr[j + 1]`) in the unsorted portion of the list.
- If an element is larger than its neighbor, the code **swaps them** (`arr[j], arr[j + 1] = arr[j + 1], arr[j]`).
- With each pass of the outer loop, the largest unsorted element "bubbles up" to its correct position at the end of the array. The `n - i - 1` in the inner loop ensures that elements that are already in place are not checked again, which slightly optimizes the process.

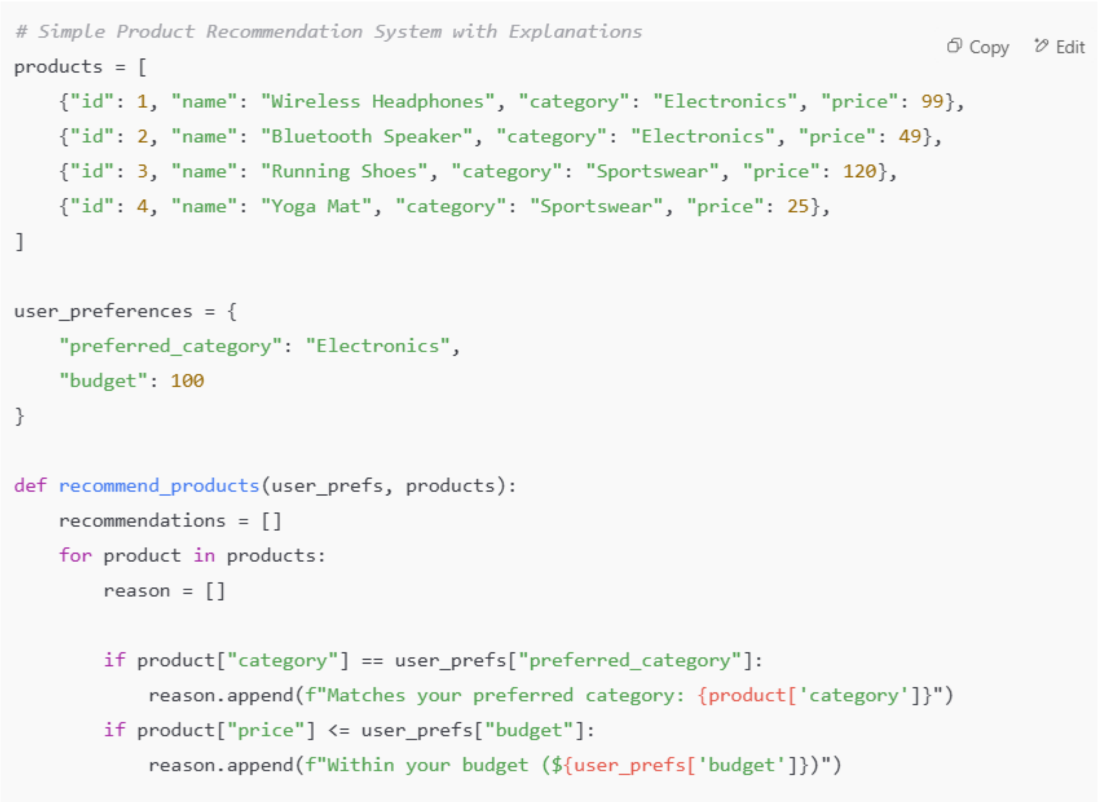


Quick Sort

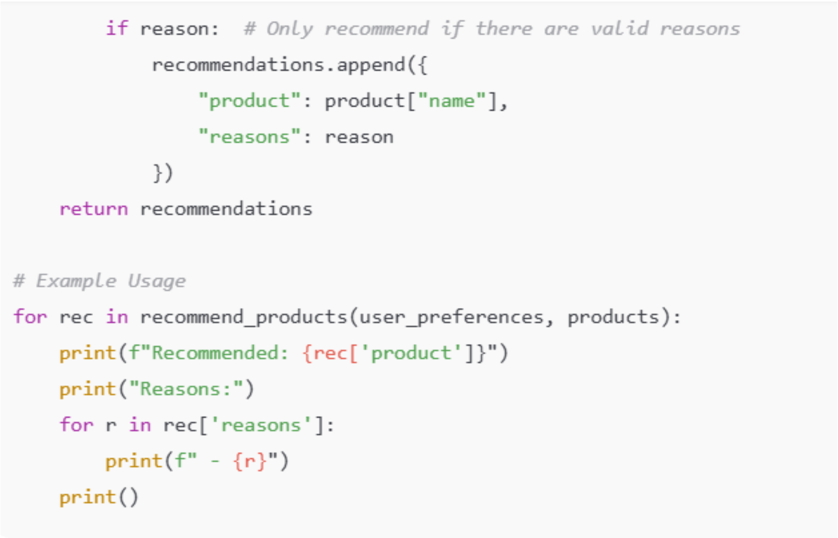
The `quick_sort` function uses a more advanced, recursive strategy known as **divide and conquer**.

- The first and most important part is the **base case** (`if len(arr) <= 1:`). This is the condition that stops the recursion. A list with zero or one element is already sorted, so the function simply returns it.
- If the list is longer, the algorithm selects a **pivot element** (`pivot = arr[0]`). The first element is used in this implementation, but other strategies can be employed.
- Next, the code **partitions the list** into two sub-lists using list comprehensions: `less_than_pivot` contains all elements from the original list (excluding the pivot) that are less than or equal to the pivot, while `greater_than_pivot` contains all elements greater than the pivot.
- Finally, the function **recursively calls itself** on each of the two sub-lists (`quick_sort(less_than_pivot)` and `quick_sort(greater_than_pivot)`). The sorted sub-lists are then concatenated together, with the pivot placed in between them, to form the final sorted list.

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."
Output:
• Code with explainable recommendations.
• Evaluation of whether explanations are understandable.

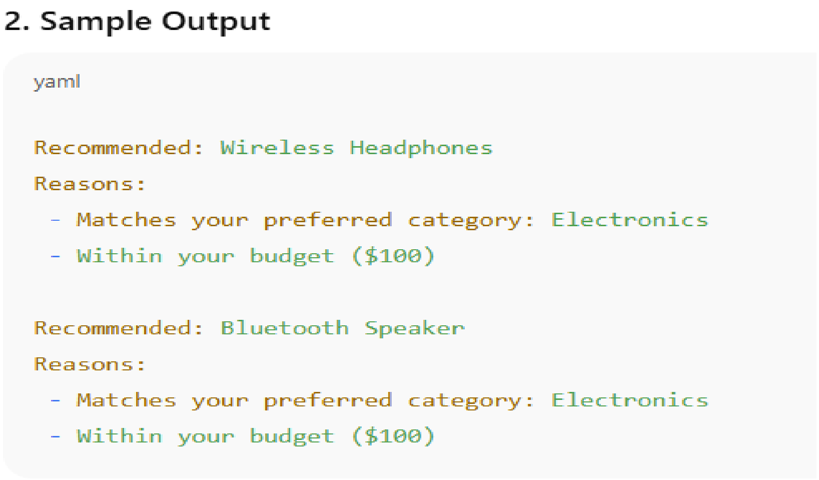


```
# Simple Product Recommendation System with Explanations
products = [
    {"id": 1, "name": "Wireless Headphones", "category": "Electronics", "price": 99},
    {"id": 2, "name": "Bluetooth Speaker", "category": "Electronics", "price": 49},
    {"id": 3, "name": "Running Shoes", "category": "Sportswear", "price": 120},
    {"id": 4, "name": "Yoga Mat", "category": "Sportswear", "price": 25},
]

user_preferences = {
    "preferred_category": "Electronics",
    "budget": 100
}

def recommend_products(user_prefs, products):
    recommendations = []
    for product in products:
        reason = []

        if product["category"] == user_prefs["preferred_category"]:
            reason.append(f"Matches your preferred category: {product['category']}")
        if product["price"] <= user_prefs["budget"]:
            reason.append(f"Within your budget (${user_prefs['budget']})")
```



```python
        if reason:  # Only recommend if there are valid reasons
            recommendations.append({
                "product": product["name"],
                "reasons": reason
            })
    return recommendations

# Example Usage
for rec in recommend_products(user_preferences, products):
    print(f"Recommended: {rec['product']}")
    print("Reasons:")
    for r in rec['reasons']:
        print(f" - {r}")
    print()
```

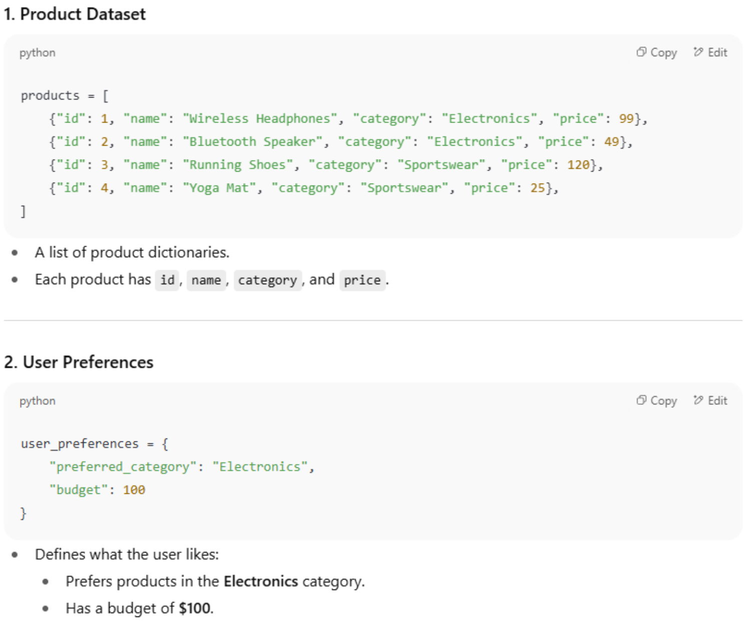
Output:



2. Sample Output

```
Recommended: Wireless Headphones
Reasons:
 - Matches your preferred category: Electronics
 - Within your budget ($100)

Recommended: Bluetooth Speaker
Reasons:
 - Matches your preferred category: Electronics
 - Within your budget ($100)
```

Explanation:



1. Product Dataset

```python
products = [
    {"id": 1, "name": "Wireless Headphones", "category": "Electronics", "price": 99},
    {"id": 2, "name": "Bluetooth Speaker", "category": "Electronics", "price": 49},
    {"id": 3, "name": "Running Shoes", "category": "Sportswear", "price": 120},
    {"id": 4, "name": "Yoga Mat", "category": "Sportswear", "price": 25},
]
```

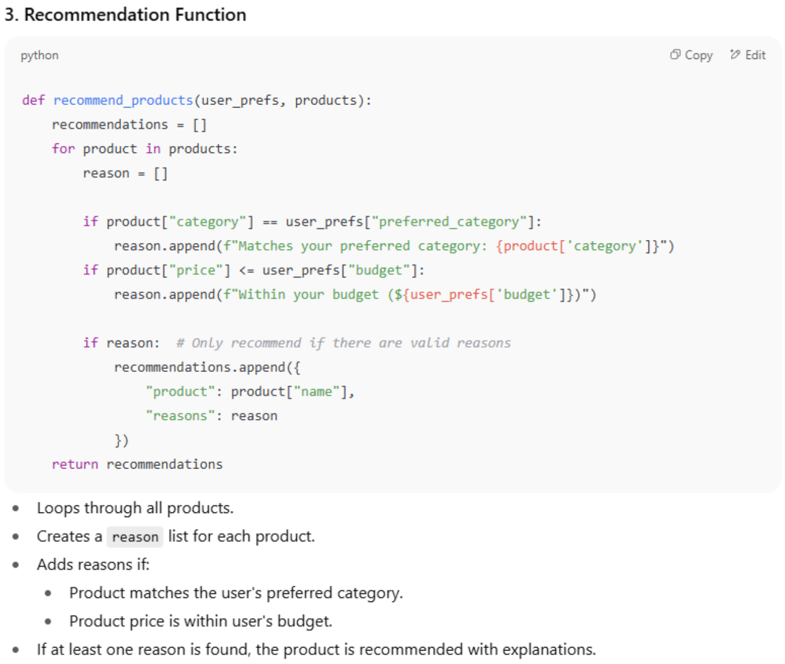
- A list of product dictionaries.
- Each product has `id`, `name`, `category`, and `price`.

---

2. User Preferences

```python
user_preferences = {
    "preferred_category": "Electronics",
    "budget": 100
}
```

- Defines what the user likes:
  - Prefers products in the **Electronics** category.
  - Has a budget of **$100**.



3. Recommendation Function

```python
def recommend_products(user_prefs, products):
    recommendations = []
    for product in products:
        reason = []

        if product["category"] == user_prefs["preferred_category"]:
            reason.append(f"Matches your preferred category: {product['category']}")
        if product["price"] <= user_prefs["budget"]:
            reason.append(f"Within your budget (${user_prefs['budget']})")

        if reason:  # Only recommend if there are valid reasons
            recommendations.append({
                "product": product["name"],
                "reasons": reason
            })
    return recommendations
```

- Loops through all products.
- Creates a `reason` list for each product.
- Adds reasons if:
  - Product matches the user's preferred category.
  - Product price is within user's budget.
- If at least one reason is found, the product is recommended with explanations.

Task Description #6 (Transparent Code Generation)

Task: Ask AI to generate a Python function for calculating factorial using recursion.

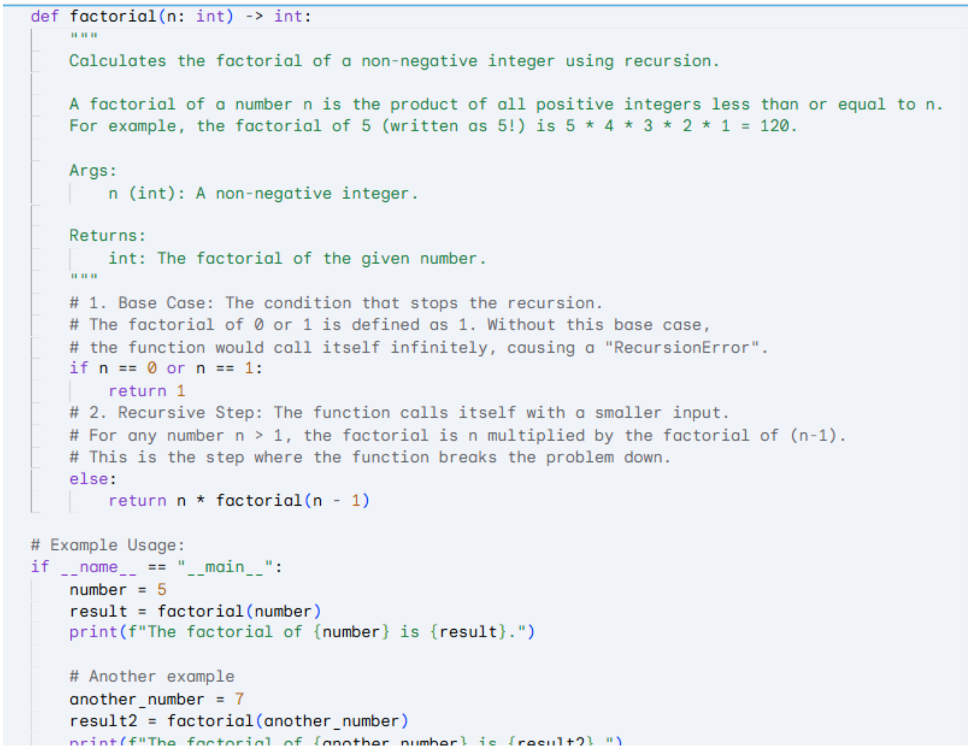
Prompt:

"Generate a recursive factorial function with comments that explain each line and a final summary of the algorithm’s flow."

Output:

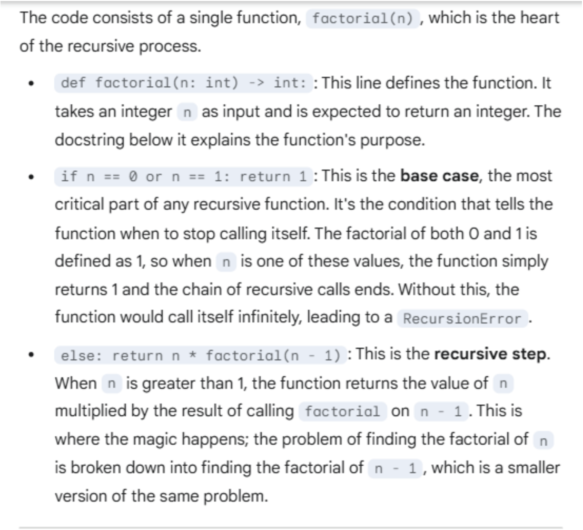
• Fully commented code.

• Clear documentation of how recursion works.



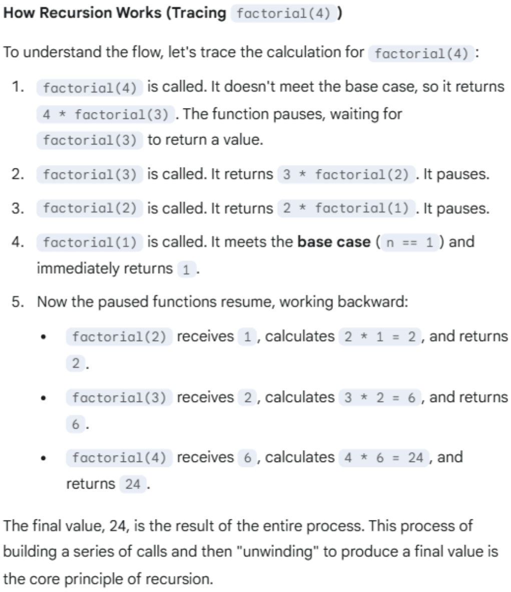
```python
def factorial(n: int) -> int:
    """
    Calculates the factorial of a non-negative integer using recursion.

    A factorial of a number n is the product of all positive integers less than or equal to n.
    For example, the factorial of 5 (written as 5!) is 5 * 4 * 3 * 2 * 1 = 120.

    Args:
        n (int): A non-negative integer.

    Returns:
        int: The factorial of the given number.
    """
    # 1. Base Case: The condition that stops the recursion.
    # The factorial of 0 or 1 is defined as 1. Without this base case,
    # the function would call itself infinitely, causing a "RecursionError".
    if n == 0 or n == 1:
        return 1
    # 2. Recursive Step: The function calls itself with a smaller input.
    # For any number n > 1, the factorial is n multiplied by the factorial of (n-1).
    # This is the step where the function breaks the problem down.
    else:
        return n * factorial(n - 1)

# Example Usage:
if __name__ == "__main__":
    number = 5
    result = factorial(number)
    print(f"The factorial of {number} is {result}.")

    # Another example
    another_number = 7
    result2 = factorial(another_number)
    print(f"The factorial of {another_number} is {result2}.")
```

Explanation:



The code consists of a single function, `factorial(n)`, which is the heart of the recursive process.

- `def factorial(n: int) -> int:`: This line defines the function. It takes an integer `n` as input and is expected to return an integer. The docstring below it explains the function's purpose.
- `if n == 0 or n == 1: return 1`: This is the **base case**, the most critical part of any recursive function. It's the condition that tells the function when to stop calling itself. The factorial of both 0 and 1 is defined as 1, so when `n` is one of these values, the function simply returns 1 and the chain of recursive calls ends. Without this, the function would call itself infinitely, leading to a `RecursionError`.
- `else: return n * factorial(n - 1)`: This is the **recursive step**. When `n` is greater than 1, the function returns the value of `n` multiplied by the result of calling `factorial` on `n - 1`. This is where the magic happens; the problem of finding the factorial of `n` is broken down into finding the factorial of `n - 1`, which is a smaller version of the same problem.



How Recursion Works (Tracing `factorial(4)`)

To understand the flow, let's trace the calculation for `factorial(4)`:

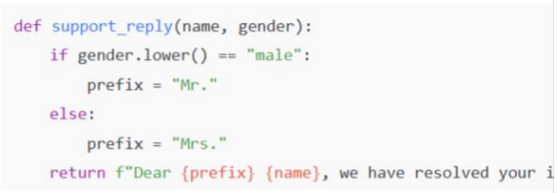
1. `factorial(4)` is called. It doesn't meet the base case, so it returns `4 * factorial(3)`. The function pauses, waiting for `factorial(3)` to return a value.
2. `factorial(3)` is called. It returns `3 * factorial(2)`. It pauses.
3. `factorial(2)` is called. It returns `2 * factorial(1)`. It pauses.
4. `factorial(1)` is called. It meets the **base case** (`n == 1`) and immediately returns `1`.
5. Now the paused functions resume, working backward:
    - `factorial(2)` receives `1`, calculates `2 * 1 = 2`, and returns `2`.
    - `factorial(3)` receives `2`, calculates `3 * 2 = 6`, and returns `6`.
    - `factorial(4)` receives `6`, calculates `4 * 6 = 24`, and returns `24`.

The final value, 24, is the result of the entire process. This process of building a series of calls and then "unwinding" to produce a final value is the core principle of recursion.

Task Description #7 (Inclusiveness in Customer Support)

Code Snippet:

Task:

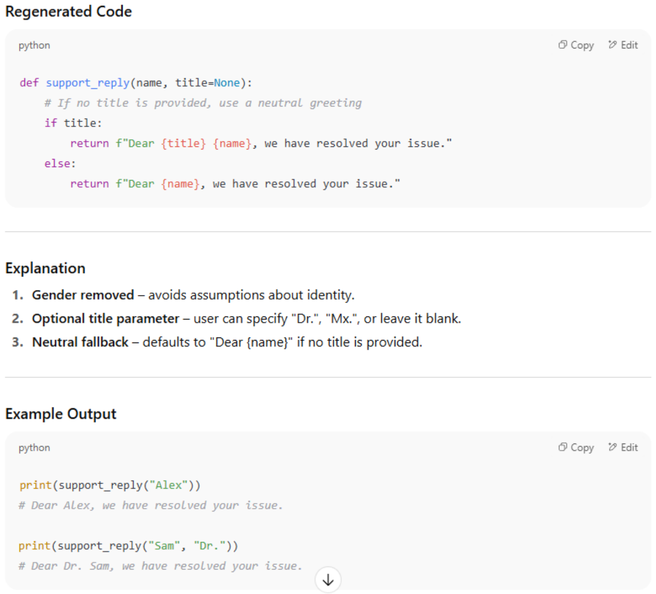


```python
def support_reply(name, gender):
    if gender.lower() == "male":
        prefix = "Mr."
    else:
        prefix = "Mrs."
    return f"Dear {prefix} {name}, we have resolved your i
```

Regenerate the code so that support messages use neutral language (e.g., “Dear {name}”) and optionally accept preferred titles.

Output:

- Neutral, user-friendly support responses.



Regenerated Code

```python
def support_reply(name, title=None):
    # If no title is provided, use a neutral greeting
    if title:
        return f"Dear {title} {name}, we have resolved your issue."
    else:
        return f"Dear {name}, we have resolved your issue."
```

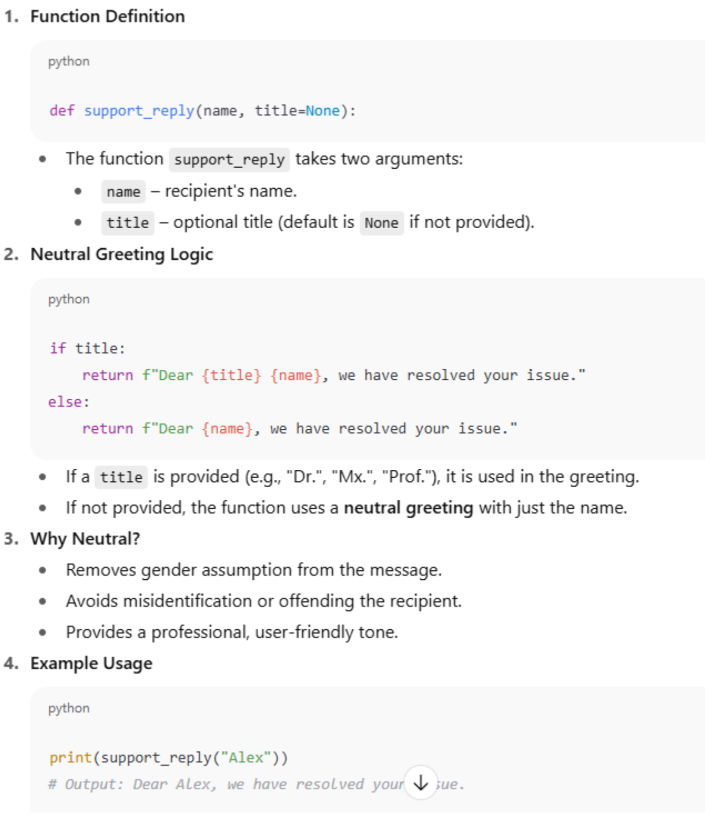
Explanation

1. **Gender removed** – avoids assumptions about identity.
2. **Optional title parameter** – user can specify "Dr.", "Mx.", or leave it blank.
3. **Neutral fallback** – defaults to "Dear {name}" if no title is provided.

Example Output

```python
print(support_reply("Alex"))
# Dear Alex, we have resolved your issue.

print(support_reply("Sam", "Dr."))
# Dear Dr. Sam, we have resolved your issue.
```

Explanation:



1. **Function Definition**

```python
def support_reply(name, title=None):
```

- The function `support_reply` takes two arguments:
  - `name` – recipient's name.
  - `title` – optional title (default is `None` if not provided).

2. **Neutral Greeting Logic**

```python
if title:
    return f"Dear {title} {name}, we have resolved your issue."
else:
    return f"Dear {name}, we have resolved your issue."
```

- If a `title` is provided (e.g., "Dr.", "Mx.", "Prof."), it is used in the greeting.
- If not provided, the function uses a **neutral greeting** with just the name.

3. **Why Neutral?**
- Removes gender assumption from the message.
- Avoids misidentification or offending the recipient.
- Provides a professional, user-friendly tone.

4. **Example Usage**

```python
print(support_reply("Alex"))
# Output: Dear Alex, we have resolved your sue.
```