

NAME :B.YASHWANTH
ROLL NO: 2503A51L42
BATCH:20

TASK1

- **PROMPT:** "Generate code to fetch weather data securely without exposing API keys in the code."
- **CODE GENERATED BY THE AI AND ITS OUTPUT IN THE BELOW PICTURE:**

```
1 import os
2 import requests
3 from dotenv import load_dotenv
4
5 load_dotenv()
6
7 def get_weather(city):
8     api_key = os.getenv('WEATHER_API_KEY')
9     if not api_key:
10         raise ValueError("API key not found. Set the WEATHER_API_KEY environment variable.")
11     url = f"https://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&units=metric"
12     response = requests.get(url)
13     response.raise_for_status()
14     return response.json()
15
16 if __name__ == "__main__":
17     city = input("Enter city name: ")
18     try:
19         weather = get_weather(city)
20         print(f"Weather in {city}: {weather['weather'][0]['description']}, temperature: {weather['temp']}°C")
21     except Exception as e:
22         print(f"Error fetching weather data: {e}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

[notice] To update, run: C:\Users\ramsa\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFoundation.Python.3.11_qbz5n2kz8v96qpython.exe -m pip install --upgrade pip

PS C:\Users\ramsa> & C:\Users\ramsa\AppData\Local\Microsoft\WindowsApps\python3.11.exe "C:/Users/ramsa/OneDrive/Documents/ai assistant lab/ai_lab.py"

Enter city name: warangal

Error fetching weather data: API key not found. Set the WEATHER_API_KEY environment variable.

PS C:\Users\ramsa> & C:\Users\ramsa\AppData\Local\Microsoft\WindowsApps\python3.11.exe "C:/Users/ramsa/OneDrive/Documents/ai assistant lab/ai_lab.py"

Enter city name: warangal

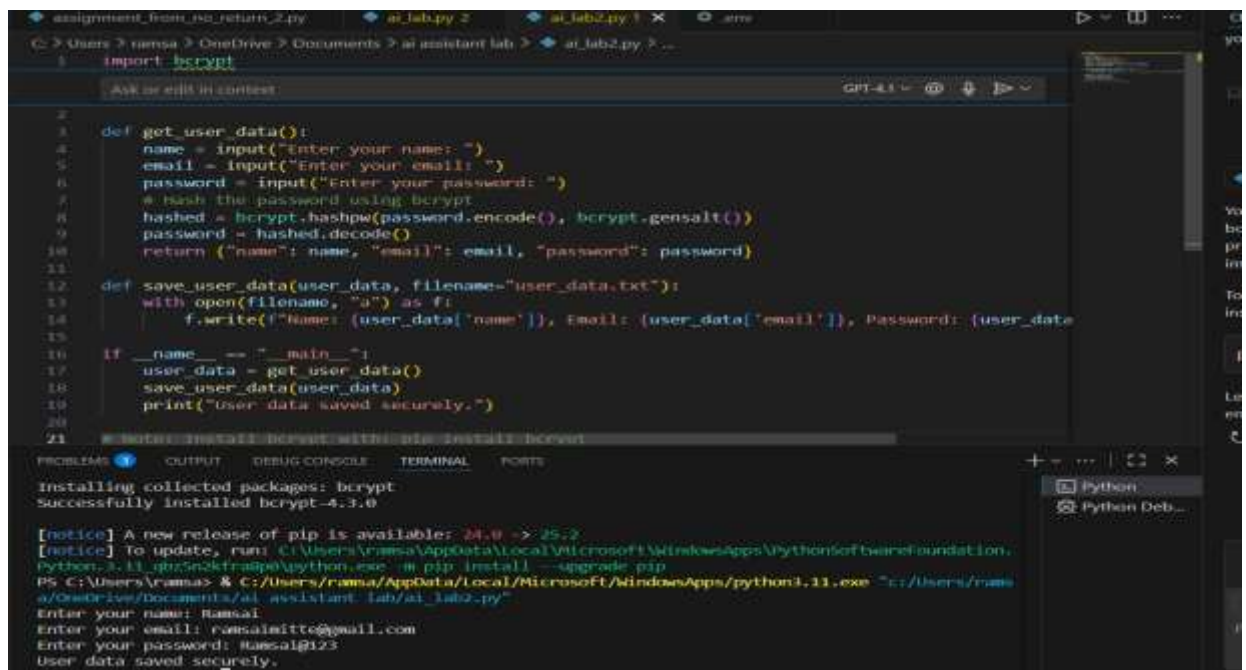
Error fetching weather data: 401 client error: unauthorized for url: https://api.openweathermap.org/data/2.5/weather?q=warangal&appid=3645651d31b39d24f1695b42f3c93757&units=metric

1 WEATHER_API_KEY="3645651d31b39d24f1695b42f3c93757"

....

TASK 2

- **Task:** Use an AI tool to generate a Python script that stores user data (name, email, password) in a file.
- **PROMPT:** To generate a Python script that stores user data (name, email, password) in a file.



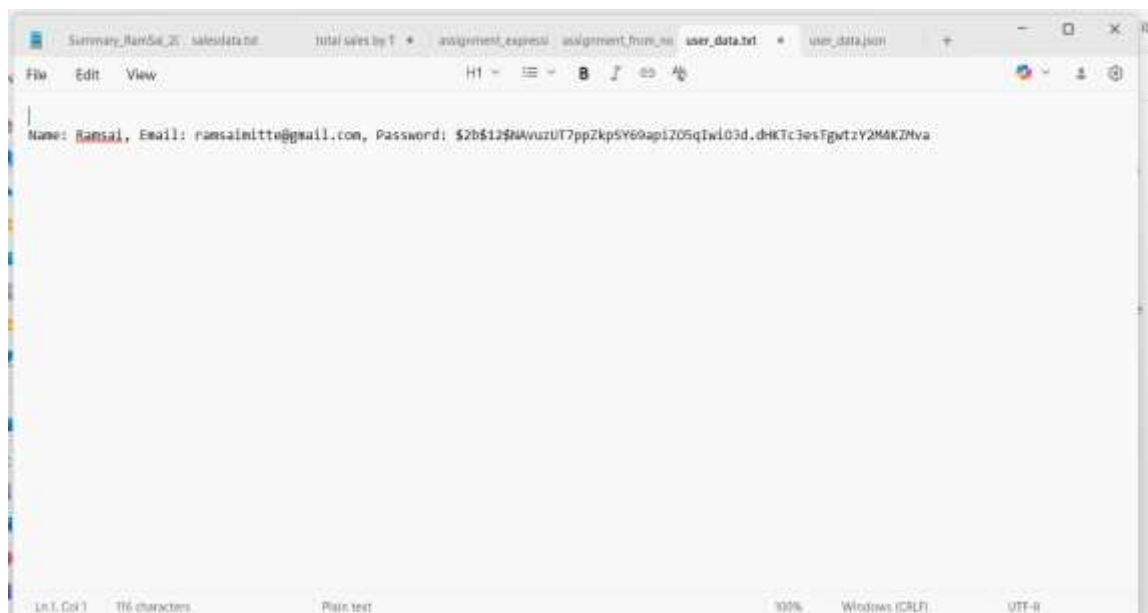
```
1 import bcrypt
2
3 def get_user_data():
4     name = input("Enter your name: ")
5     email = input("Enter your email: ")
6     password = input("Enter your password: ")
7     # hash the password using bcrypt
8     hashed = bcrypt.hashpw(password.encode(), bcrypt.gensalt())
9     password = hashed.decode()
10    return {"name": name, "email": email, "password": password}
11
12 def save_user_data(user_data, filename="user_data.txt"):
13    with open(filename, "a") as f:
14        f.write(f'Name: {user_data["name"]}, Email: {user_data["email"]}, Password: {user_data["password"]}\n')
15
16 if __name__ == "__main__":
17    user_data = get_user_data()
18    save_user_data(user_data)
19    print("User data saved securely.")
20
21 # Notes: Install bcrypt with: pip install bcrypt
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Installing collected packages: bcrypt
Successfully installed bcrypt-4.3.0

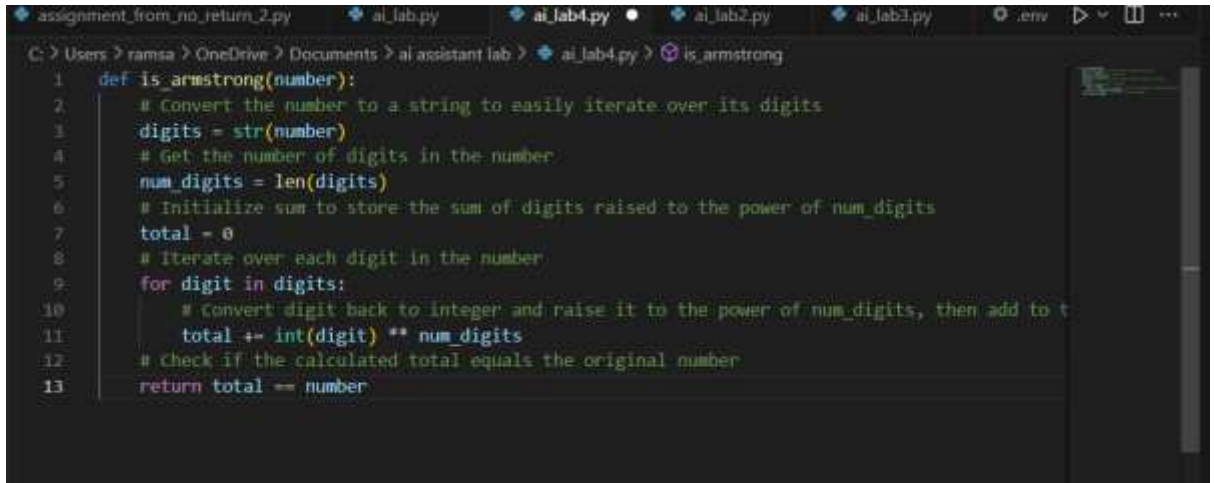
[notice] A new release of pip is available: 24.0 -> 25.2
[notice] To update, run: C:\Users\rama-a\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFoundation.Python.3.11_qbz5n2kfrahpq\python.exe -m pip install --upgrade pip
PS C:\Users\rama-a> & C:\Users\rama-a\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:/Users/rama-a/OneDrive/Documents/ai assistant lab/ai_lab2.py"
Enter your name: ramsalmitte@gmail.com
Enter your email: ramsalmitte@gmail.com
Enter your password: ramsal123
User data saved securely.
PS C:\Users\rama-a>

- **The file saved like= output:**



TASK 3

- **Task:** Use AI to generate an Armstrong number checking function with comments and explanations.
- **The output code:**

A screenshot of a code editor window with a dark theme. The editor shows a Python function named 'is_armstrong' with line numbers 1 through 13. The code includes comments explaining each step: converting the number to a string, getting the number of digits, initializing a sum, iterating over each digit, converting the digit back to an integer, raising it to the power of the number of digits, and checking if the sum equals the original number. The function returns 'total == number'.

```
1 def is_armstrong(number):
2     # Convert the number to a string to easily iterate over its digits
3     digits = str(number)
4     # Get the number of digits in the number
5     num_digits = len(digits)
6     # Initialize sum to store the sum of digits raised to the power of num_digits
7     total = 0
8     # Iterate over each digit in the number
9     for digit in digits:
10        # Convert digit back to integer and raise it to the power of num_digits, then add to total
11        total += int(digit) ** num_digits
12    # Check if the calculated total equals the original number
13    return total == number
```

Explanation of the code:

Line-by-line explanation:

1. `def is_armstrong(number):`
Defines a function named `is_armstrong` that takes an integer number as input.
2. `digits = str(number)`
Converts the number to a string so we can easily access each digit.
3. `num_digits = len(digits)`
Calculates how many digits are in the number.
4. `total = 0`
Initializes a variable to keep the sum of each digit raised to the power of `num_digits`.
5. `for digit in digits:`
Loops through each digit in the string representation of the number.
6. `total += int(digit) ** num_digits`
Converts the digit back to an integer, raises it to the power of `num_digits`, and adds it to total.
7. `return total == number`

TASK 4

- **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:**
 - 1) code for QuickSort:

```
# QuickSort: Divide-and-conquer, recursive, efficient for large lists
def quicksort(arr):
    """
    Sorts the list using QuickSort algorithm.
    Steps:
    1. If the list has 0 or 1 elements, it's already sorted.
    2. Choose a 'pivot' element from the list.
    3. Partition the list into two sublists:
       - Elements less than the pivot
       - Elements greater than or equal to the pivot
    4. Recursively sort the sublists.
    5. Combine the sorted sublists and the pivot.
    """
    if len(arr) <= 1:
        return arr # Base case: already sorted
    pivot = arr[len(arr) // 2] # Choose middle element as pivot
    left = [x for x in arr if x < pivot] # Elements less than pivot
    middle = [x for x in arr if x == pivot] # Elements equal to pivot
    right = [x for x in arr if x > pivot] # Elements greater than pivot
    # Recursively sort left and right, then combine
    return quicksort(left) + middle + quicksort(right)
```

- 2) BubbleSort:

```
# BubbleSort: Simple, iterative, inefficient for large lists
def bubblesort(arr):
    """
    Sorts the list using BubbleSort algorithm.
    Steps:
    1. Repeat for each element in the list:
       a. Compare each pair of adjacent elements.
       b. Swap them if they are in the wrong order.
    2. Continue until no swaps are needed (list is sorted).
    """
    n = len(arr)
    for i in range(n):
        # Track if any swaps are made in this pass
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                # Swap if elements are in wrong order
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break # List is sorted
    return arr
```

- comparative explanation of their logic and efficiency.

```
# Comparative Explanation of QuickSort vs BubbleSort
# -----
# QuickSort:
# - Logic: Uses divide-and-conquer. Selects a pivot, partitions the list into elements less than, equal to, and greater than the pi
# - Efficiency: Average and best case time complexity is  $O(n \log n)$ . Worst case is  $O(n^2)$  (rare, e.g., sorted input with poor pivot
# - Space: Uses extra space for recursion and partitioning (not in-place in this version).
# - Use Case: Preferred for large datasets due to speed and efficiency.
#
# BubbleSort:
# - Logic: Repeatedly compares adjacent elements and swaps them if out of order. Each pass moves the largest unsorted element to it
# - Efficiency: Time complexity is  $O(n^2)$  in all cases. Very slow for large lists.
# - Space: In-place sorting, no extra space needed.
# - Use Case: Mainly for educational purposes or very small datasets.
#
# Summary:
# QuickSort is much faster and more efficient for large lists, while BubbleSort is simple but inefficient. QuickSort uses recursion
```

TASK 5

Task: Use AI to create a product recommendation system.

Prompt:

"Generate a recommendation system that also provides reasons for each suggestion."

Output:

```
1 # Simple Explainable Recommendation System
2 # This example recommends books based on user interests and explains each suggestion.
3
4 def recommend_books(user_interests, books):
5     recommendations = []
6     for book in books:
7         # Check if any user interest matches the book's genre
8         matched_genres = set(user_interests).intersection(set(book['genres']))
9         if matched_genres:
10             # Build an explanation for the recommendation
11             explanation = f"Recommended because it matches your interest in: {', '.join(matched_genres)}."
12             recommendations.append({
13                 'title': book['title'],
14                 'explanation': explanation
15             })
16     return recommendations
17
18 # Example usage
19 if __name__ == "__main__":
20     user_interests = ['science fiction', 'history']
21     books = [
22         {'title': 'Dune', 'genres': ['science fiction', 'adventure']},
23         {'title': 'Sapiens', 'genres': ['history', 'non-fiction']},
24         {'title': 'Pride and Prejudice', 'genres': ['romance', 'classic']}
25     ]
26     recs = recommend_books(user_interests, books)
27     for rec in recs:
28         print(f"Suggestion: {rec['title']}")
29         print(f"Reason: {rec['explanation']}")
30
31 # Evaluation:
32 # The explanations are clear and directly reference the user's interests, making them understandable and transparent.
```

```
ab5.py"
Suggestion: Dune
Reason: Recommended because it matches your interest in: science fiction.
Suggestion: Sapiens
Reason: Recommended because it matches your interest in: history.
PS C:\Users\ramsa> 
```