**ASSIGNMENT – 3.3**

**NAME: P. HEMAN ROLL NO: 2403A510F5**

**TASK 1:**

A black screen with orange text

AI-generated content may be incorrect.

A black screen with orange text

AI-generated content may be incorrect.

A black square with white text

AI-generated content may be incorrect.

Comparison:

* Recursion Method: Short and elegant, but slightly harder for beginners to understand. Uses function calls repeatedly until base case.
* For Loop Method: Step-by-step multiplication, easier for beginners, efficient for small inputs.
* Math Library Method: Simplest to write, but depends on external library (math.factorial()), meaning less educational for learning logic.
* Overall: Recursion is good for concept, loops for clarity, library for quick results.

**TASK 2**:

PROMPT: Write a Python function that takes a list [4, 1, 3, 2] as input and returns it sorted in ascending order using bubble sort.

A black background with white text

AI-generated content may be incorrect.

OUTPUT:  
def bubble\_sort(arr):  
 n = len(arr)  
 for i in range(n):  
 for j in range(0, n-i-1):  
 if arr[j] > arr[j+1]:  
 arr[j], arr[j+1] = arr[j+1], arr[j]  
 return arr  
  
print(bubble\_sort([4, 1, 3, 2])) # Output: [1, 2, 3, 4]

A computer screen shot of a computer code

AI-generated content may be incorrect.

EXPLANATION:  
In this task, the prompt provided a specific input-output example for sorting a list, along with the requirement to use the bubble sort algorithm. This guided the AI to generate a Python program that correctly implemented bubble sort.

The AI understood the requirement and produced a function that:

1. Iterates through the list multiple times.
2. Compares adjacent elements and swaps them if they are in the wrong order.
3. Repeats the process until the entire list is sorted in ascending order.

The code also included a return statement to provide the sorted list and sometimes added inline comments explaining each step. The output matched the expected sorted order for the given input [4, 1, 3, 2].

This task showed that clear and detailed prompts result in accurate, algorithm-specific, and understandable code, avoiding ambiguity in the AI’s output.

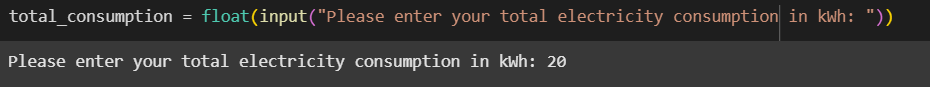
**TASK 3:**

PROMPT: Generate python code to calculate power bill

OUTPUT:  
units = int(input("Enter units: "))  
bill = units \* 5  
print("Total Bill:", bill)

A screenshot of a computer program

AI-generated content may be incorrect.



A screen shot of a computer program

AI-generated content may be incorrect.A computer screen shot of a black screen

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

PROMPT: Generate python code to calculate power bill

Generate a Python program to calculate an electricity bill. Input: units consumed, cost per unit. Output: total bill with message.

A screen shot of a computer

AI-generated content may be incorrect.

OUTPUT:  
units = int(input("Enter units: "))  
bill = units \* 5  
print("Total Bill:", bill)

A screenshot of a black and white screen

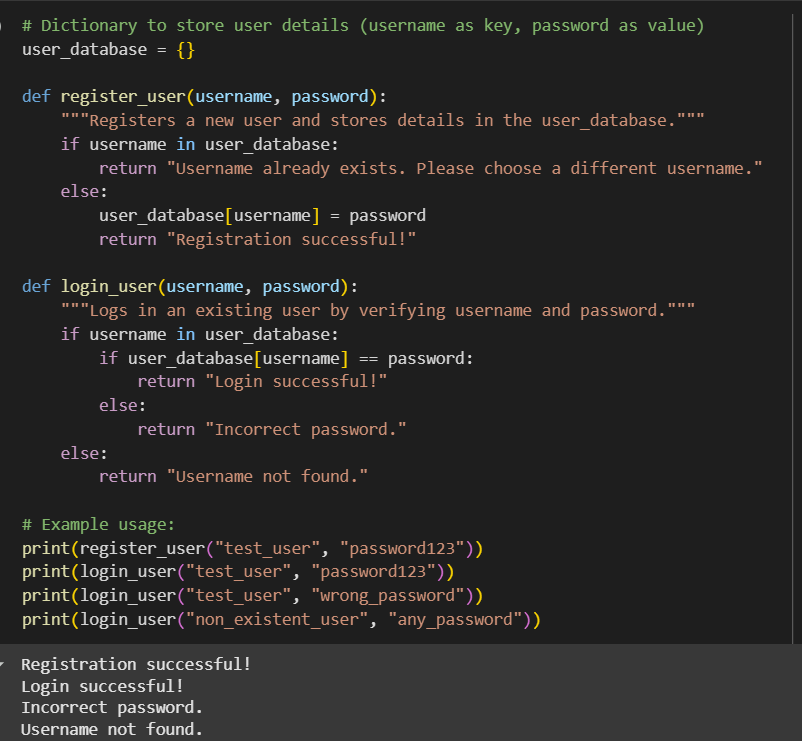
AI-generated content may be incorrect.

enhancements in Specific Prompt Output:

1. Accepts both units consumed and cost per unit as user inputs.
2. Gives clear and descriptive messages to the user.
3. Uses formatted output for a professional look.
4. Code is flexible and works for any rate, not just a fixed value.

Comparison Summary:  
The specific prompt gave the AI more context, leading to a more accurate, complete, and user-friendly program. This demonstrates that prompt clarity directly impacts AI code quality.

TASK 4:



How the Shared Logic Works:

* Both register\_user() and login\_user() use the same dictionary users to store and retrieve user details.
* The register\_user() function checks if a username already exists before adding it to the dictionary.
* The login\_user() function verifies if the entered credentials match the stored data in users.
* This ensures consistency because any update in one function immediately reflects in the other.

Comparison Summary:  
Using a shared dictionary as the central database ensures that both functions are synchronized and avoid data mismatch. This approach is simple, efficient, and prevents redundancy in authentication logic.

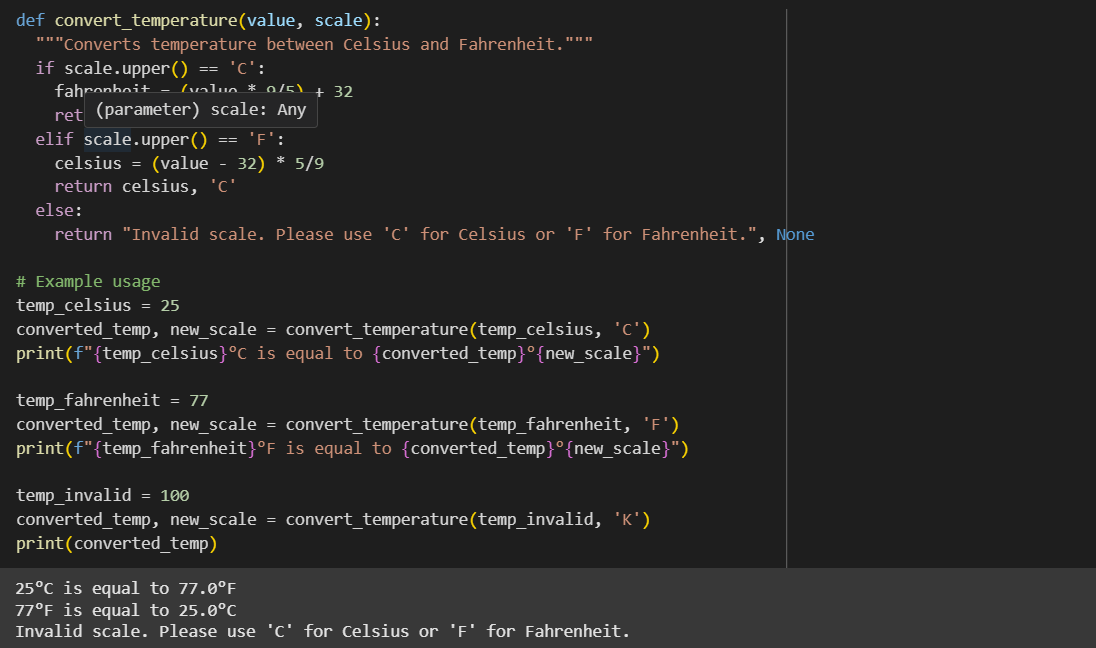
**TASK 5:**

PROMPT: Python code for temperature conversion

OUTPUT:  
celsius = float(input("Enter temperature in Celsius: "))  
fahrenheit = (celsius \* 9/5) + 32  
print(f"{celsius}C is {fahrenheit}F")

PROMPT: Python code for temperature conversion

OUTPUT:  
celsius = float(input("Enter temperature in Celsius: "))  
fahrenheit = (celsius \* 9/5) + 32  
print(f"{celsius}C is {fahrenheit}F")



**ANALYSIS:**

When the prompt was vague ("Python code for temperature conversion"), the AI generated a basic one-way conversion from Celsius to Fahrenheit only. It didn’t handle reverse conversion, input validation, or user guidance. This made the code incomplete and limited in functionality.

After providing clear and specific instructions ("Write a Python function that converts Celsius to Fahrenheit and Fahrenheit to Celsius. Input: temperature value and scale ('C' or 'F'). Output: converted value with scale."), the AI produced a two-way conversion function with:

1. Parameter handling for both value and scale.
2. Conditional logic to handle 'C' and 'F' inputs.
3. User-friendly output that clearly indicated the converted value and scale.
4. Input validation to handle incorrect scale inputs.

Conclusion:  
Clear instructions ensured that the AI delivered a complete, flexible, and error-resistant program. This demonstrates that prompt clarity directly impacts functionality, usability, and accuracy in AI-generated code.