AI ASSISTED CODING

NAME: N.KEERTHI REDDY

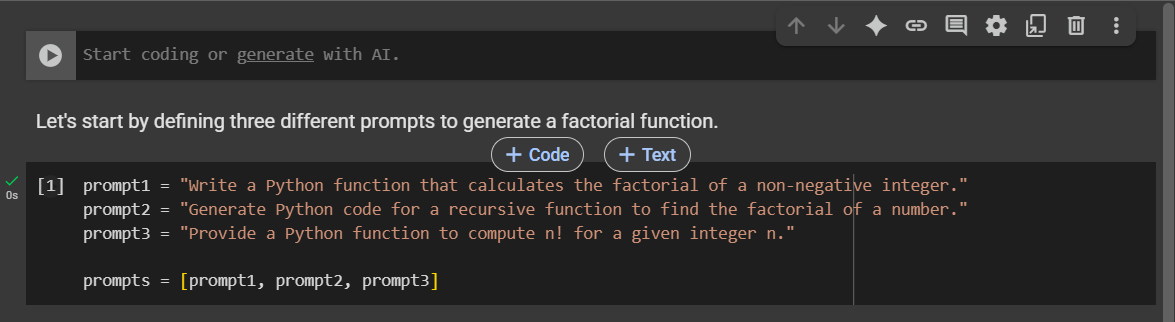
ENROLL NUMBER: 2403A52376

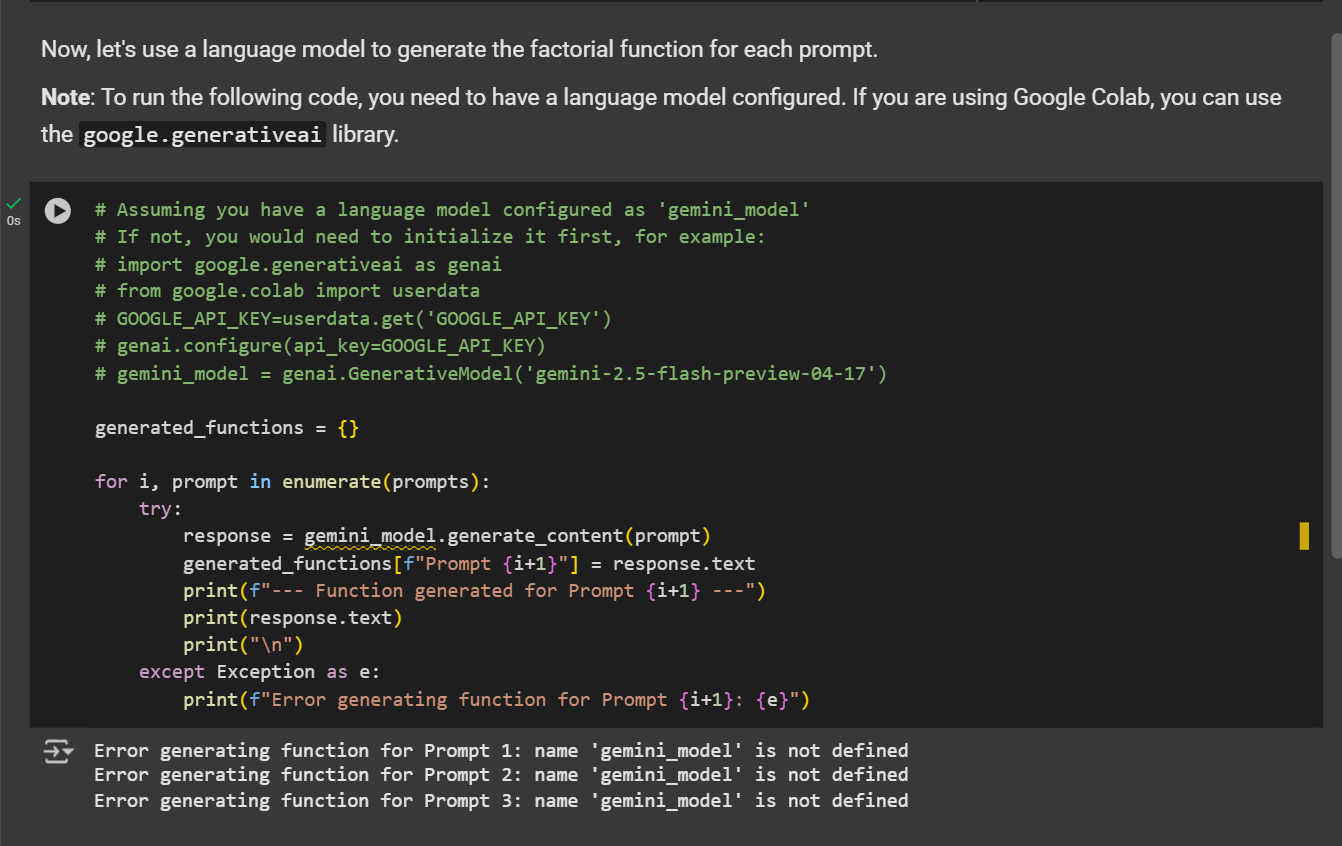
BATCH NUMBER :14

ASSIGNMENT 3.3

TASK DISCRIPTION#1

1)Try 3 different prompts to generate a factorial function





CODE EXPLANATION:

Here's a breakdown of the code cells:

1. **Code cell dc151576**:
   * This cell defines three different string variables: prompt1, prompt2, and prompt3. Each string contains a slightly different prompt asking for a Python function to calculate the factorial of a number.
   * These three prompts are then stored in a list called prompts. This list will be used later to iterate through each prompt.
2. **Code cell e02776d6**:
   * This cell is designed to use a language model (presumably named gemini\_model) to generate Python code for a factorial function based on each prompt defined in the previous cell.
   * It initializes an empty dictionary called generated\_functions. This dictionary will store the generated code, with the keys indicating which prompt was used ("Prompt 1", "Prompt 2", etc.).
   * It then loops through the prompts list. For each prompt:
     + It calls gemini\_model.generate\_content(prompt) to send the prompt to the language model and get a response.
     + The generated text from the response (response.text) is stored in the generated\_functions dictionary.
     + It prints a separator and the generated function code to the console so you can see the output as it's generated.
   * There's a try...except block to catch any errors that might occur during the generation process, such as if the gemini\_model is not defined or if there's an issue with the API call.

In essence, the code sets up different ways of asking for a factorial function and then uses a language model to see how the generated code varies based on the prompt.

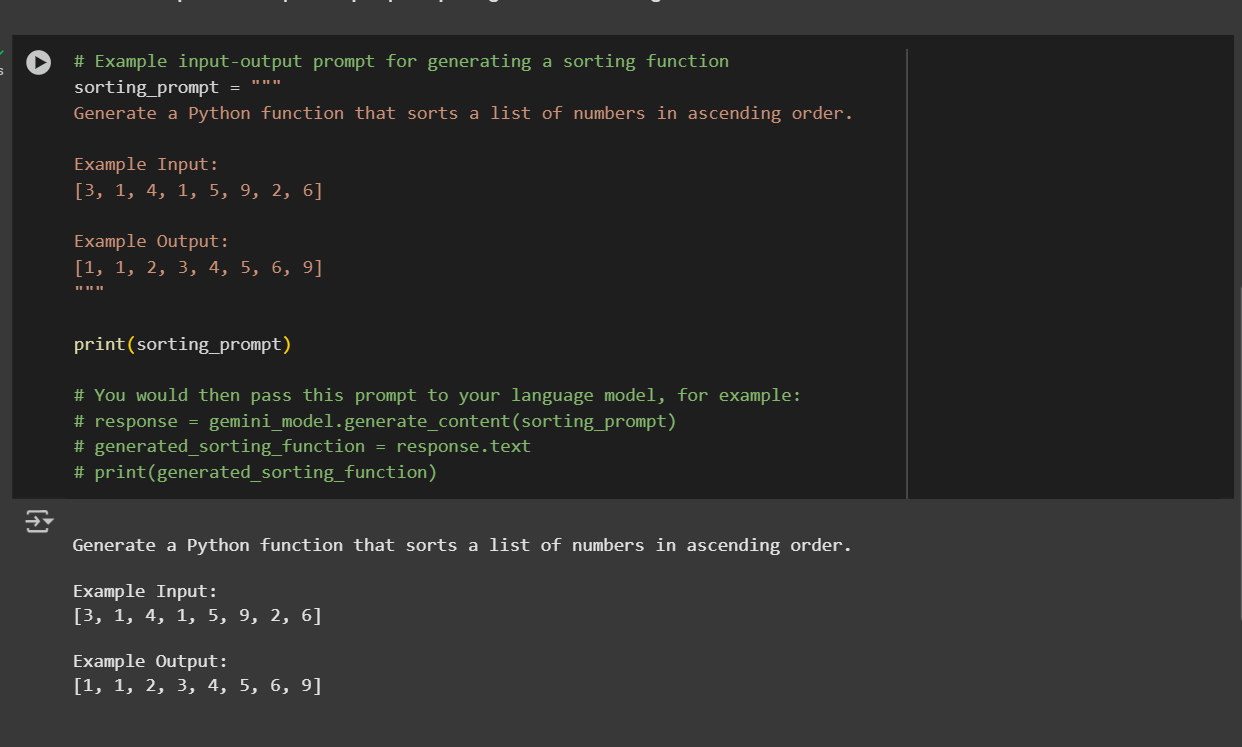
0 / 2000

Gemini can make mistakes so double-check it and use code with caution.

[Learn more](http://g.co/legal/generative-code)

TASK DISCRIPTION#2

2) Provide a clear example input-output prompt to generate a sorting function



CODE EXPLANATION:

1. **Code cell dc151576**: This cell defines three different prompts for generating a factorial function and stores them in a list.
2. **Code cell e02776d6**: This cell attempts to use a language model (presumably gemini\_model) to generate factorial functions based on the prompts defined in the previous cell. It stores the generated functions in a dictionary and prints them. (Note: The output indicates that gemini\_model was not defined when this cell was run).
3. **Code cell a626d757**: This is the most recently added cell. It defines a multi-line string variable called sorting\_prompt. This string contains a clear example of an input-output prompt designed to guide a language model in generating a Python function that sorts a list of numbers in ascending order. It includes:
   * A clear instruction on what the function should do.
   * An "Example Input" showing a sample list.
   * An "Example Output" showing the expected sorted list for the given input.
   * The cell then prints this sorting\_prompt to the console.
   * It also includes commented-out lines showing how you would typically pass this prompt to a language model (gemini\_model) to get the generated sorting function code.

In summary, the latest code cell provides a template for how to structure prompts for language models when you want them to generate code based on specific input-output examples.

0 / 2000

TASK DISCRIPTION#3

3) Start with the vague prompt “Generate python code to calculate power bill” and improve it step-by-step

**Step 1: Clarify the Problem**

Ask questions to remove ambiguity:

* What inputs are required? (e.g., usage in kWh?)
* Are there tiered rates or a flat rate?
* Should taxes or fees be included?
* Is it for residential or commercial use?

➡️ Let's assume:

* User inputs energy usage in **kWh**.
* A **tiered rate system** is used.
* There is a **fixed monthly service charge**.
* **Tax is applied** as a percentage of the subtotal.

**🔹 Step 2: Define Example Requirements**

Here’s a concrete example of what we want:

* First 500 kWh: $0.12 per kWh
* Next 500 kWh (501–1000): $0.15 per kWh
* Above 1000 kWh: $0.20 per kWh
* Fixed monthly service charge: $5.00
* Tax: 8% on subtotal (usage + service charge)

**🔹 Step 3: Design the Inputs and Outputs**

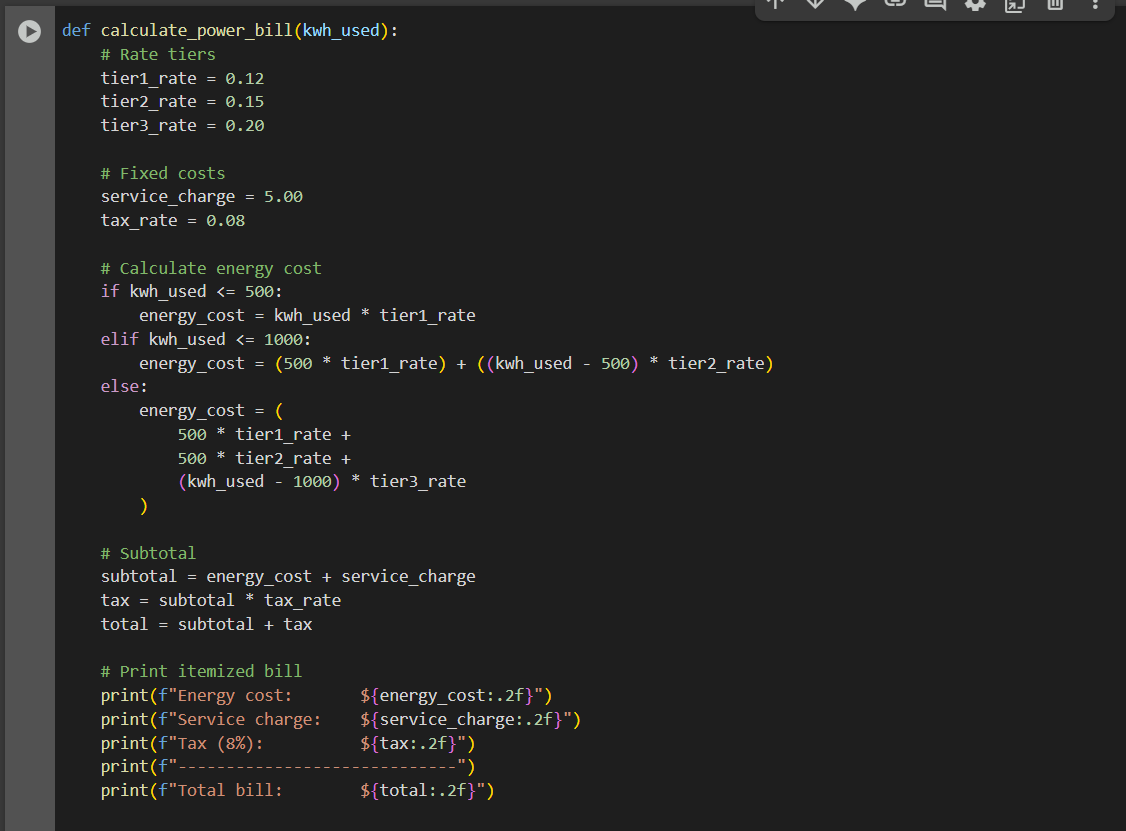
* **Input**: Total energy used in kWh (float or int)
* **Output**: Print itemized bill:
  + Energy cost
  + Service charge
  + Tax
  + Total

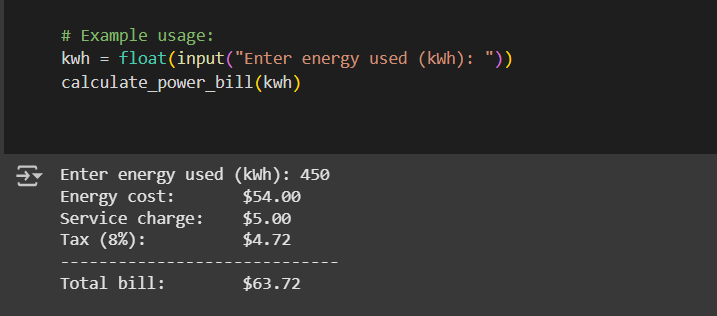
**🔹 Step 4: Write Final Prompt**

Generate Python code to calculate a monthly electricity bill using the following rules:

* First 500 kWh are billed at $0.12 per kWh
* Next 500 kWh (501–1000) at $0.15 per kWh
* Any usage over 1000 kWh at $0.20 per kWh
* Add a fixed service charge of $5.00
* Apply an 8% tax on the subtotal (usage cost + service charge)  
  The program should take energy usage (kWh) as input and print an itemized bill.

CODE:





CODE EXPLANATION:

**Function Definition**

python

CopyEdit

def calculate\_power\_bill(kwh\_used):

This defines a function named calculate\_power\_bill that takes one argument:

* kwh\_used: the total number of kilowatt-hours (kWh) used in a month.

**💰 Define Rates and Charges**

python

CopyEdit

tier1\_rate = 0.12

tier2\_rate = 0.15

tier3\_rate = 0.20

service\_charge = 5.00

tax\_rate = 0.08

These are the constants used in the billing:

* **Tier 1**: First 500 kWh at $0.12 per kWh
* **Tier 2**: Next 500 kWh (501–1000) at $0.15 per kWh
* **Tier 3**: Anything over 1000 kWh at $0.20 per kWh
* **Service Charge**: A fixed fee of $5.00 added to every bill
* **Tax Rate**: 8% tax applied to the subtotal (usage + service)

**⚙️ Calculate Energy Cost**

python

CopyEdit

if kwh\_used <= 500:

energy\_cost = kwh\_used \* tier1\_rate

If usage is 500 kWh or less, the entire usage is billed at the Tier 1 rate.

python

CopyEdit

elif kwh\_used <= 1000:

energy\_cost = (500 \* tier1\_rate) + ((kwh\_used - 500) \* tier2\_rate)

If usage is between 501 and 1000 kWh:

* First 500 kWh are billed at Tier 1
* The rest (up to 500 kWh more) are billed at Tier 2

python

CopyEdit

else:

energy\_cost = (

500 \* tier1\_rate +

500 \* tier2\_rate +

(kwh\_used - 1000) \* tier3\_rate

)

If usage exceeds 1000 kWh:

* First 500 → Tier 1
* Next 500 → Tier 2
* Remaining → Tier 3

**🧮 Calculate Subtotal, Tax, and Total**

python

CopyEdit

subtotal = energy\_cost + service\_charge

tax = subtotal \* tax\_rate

total = subtotal + tax

* **Subtotal** is the sum of energy cost and the fixed service charge
* **Tax** is 8% of the subtotal
* **Total** is the final bill after tax

**🧾 Print the Itemized Bill**

python

CopyEdit

print(f"Energy cost: ${energy\_cost:.2f}")

print(f"Service charge: ${service\_charge:.2f}")

print(f"Tax (8%): ${tax:.2f}")

print(f"-----------------------------")

print(f"Total bill: ${total:.2f}")

This prints a neatly formatted bill:

* Each component (energy, service, tax)
* A separator line
* Final total

.2f ensures all dollar values show **two decimal places**, like $23.45

**🧑‍💻 Example Usage**

python

CopyEdit

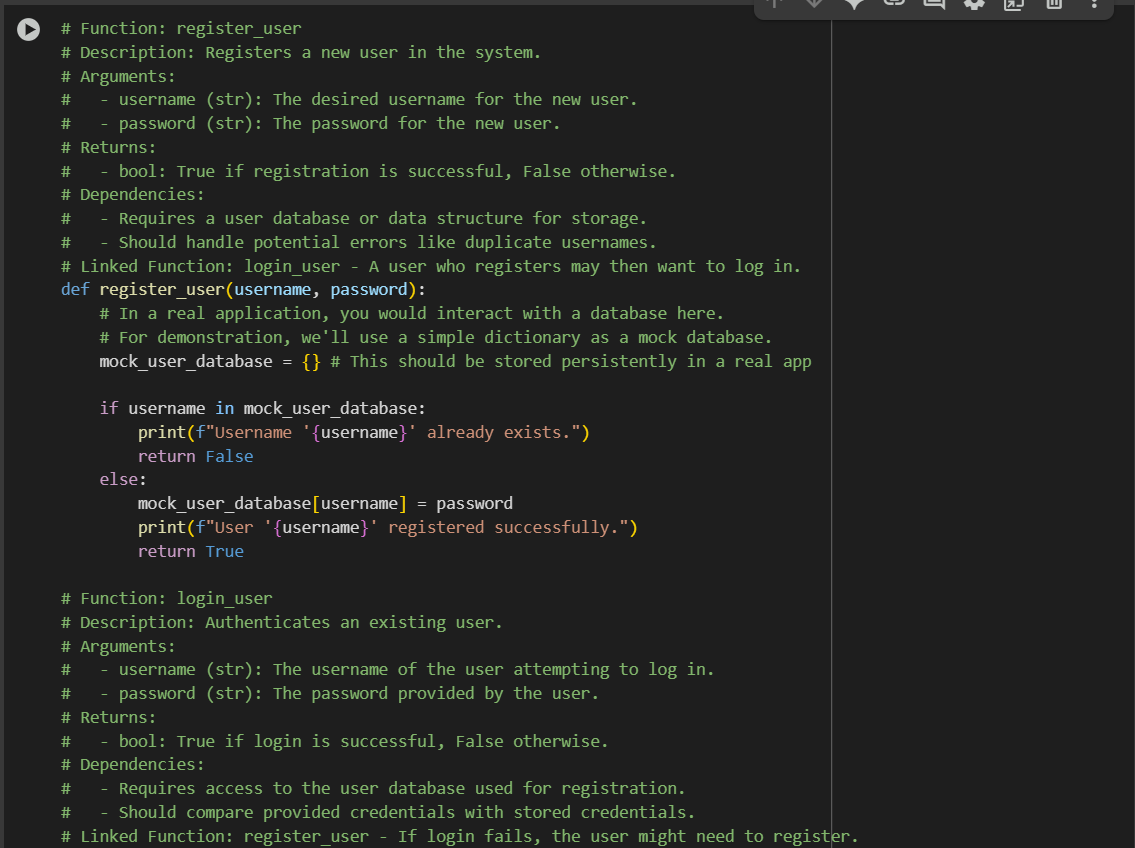
kwh = float(input("Enter energy used (kWh): "))

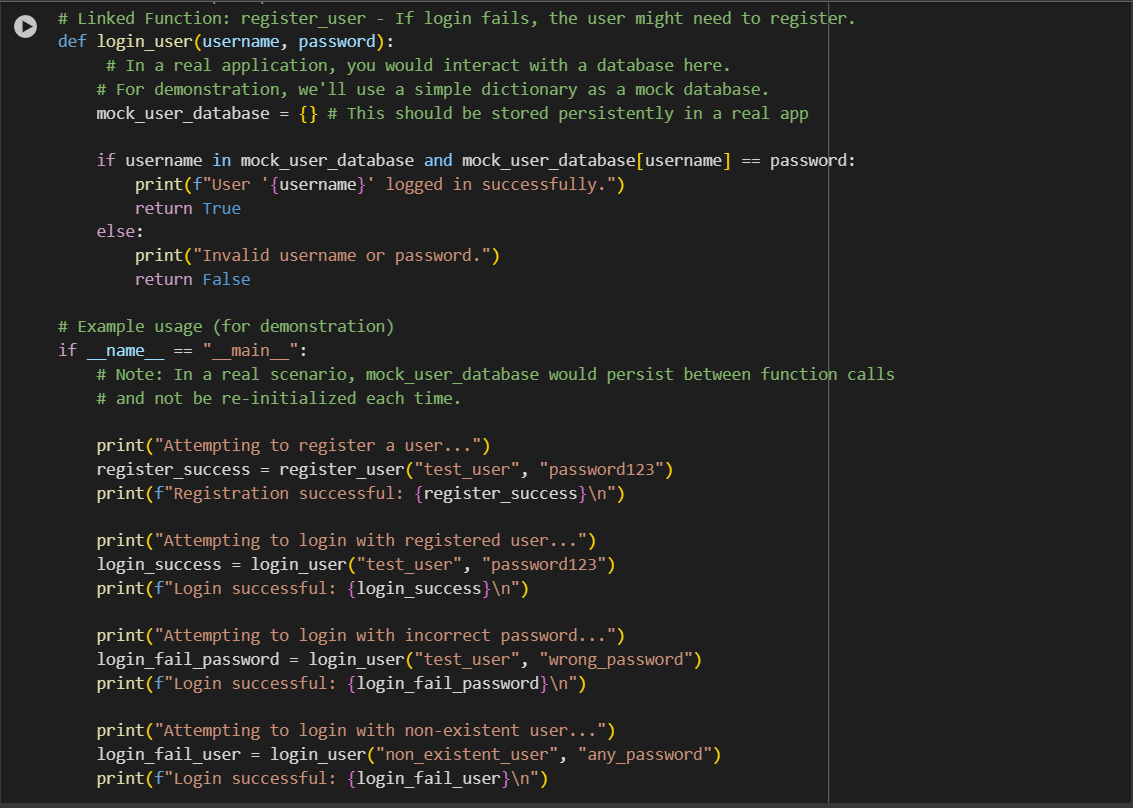
calculate\_power\_bill(kwh)

This asks the user to enter their energy usage and passes that number to the function.

TASK DISCRIPTION#4

4) Write structured comments to help AI generate two linked functions (e.g., login\_user() and register\_user())







CODE EXPLANATION:

Certainly! I can explain the Python code I generated for the register\_user and login\_user functions.

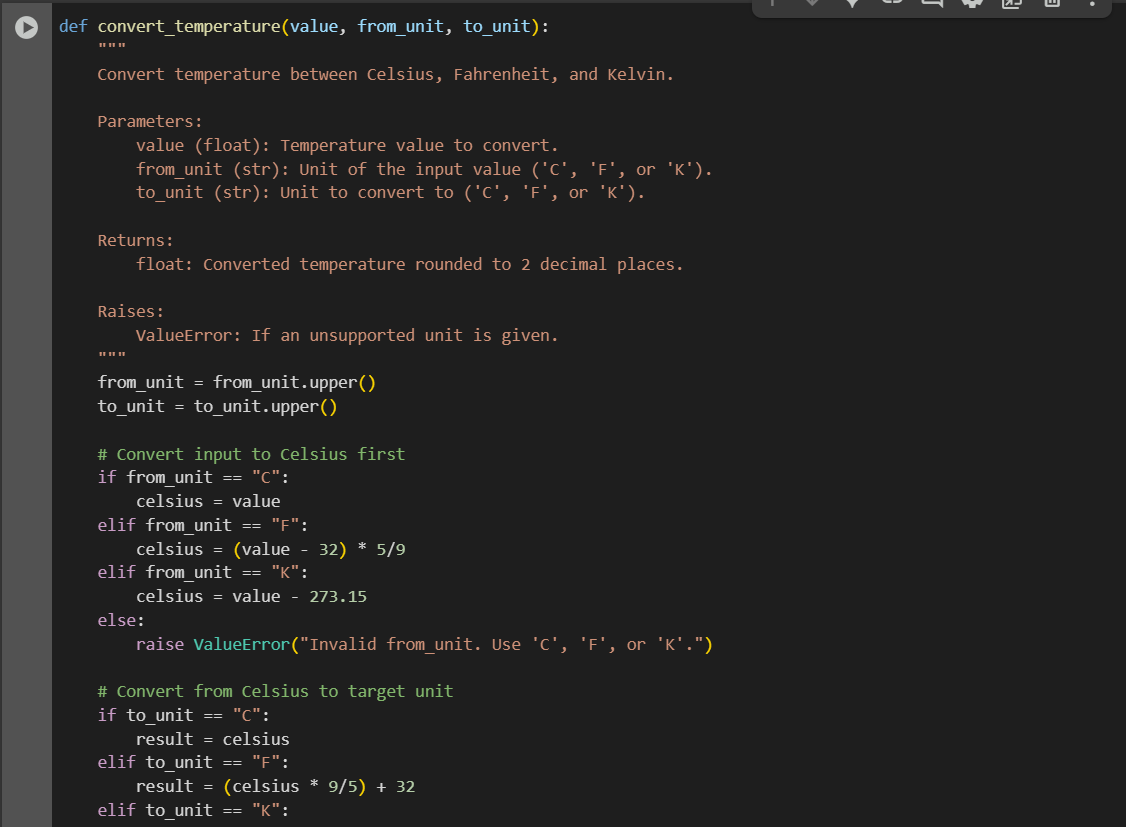
Here's a breakdown of the code:

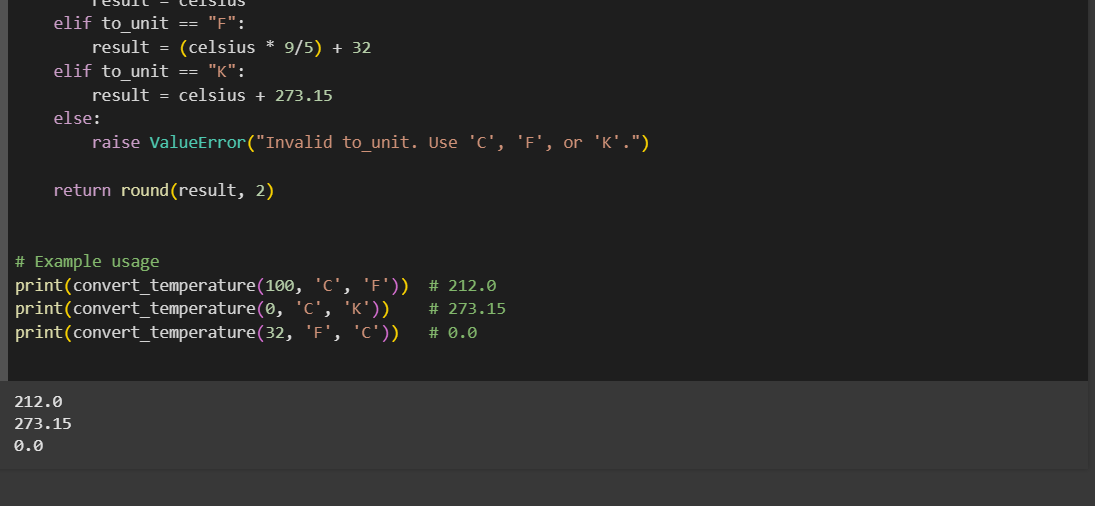
1. **register\_user(username, password) function:**
   * This function is designed to register a new user.
   * It takes two arguments: username and password.
   * Inside the function, a dictionary called mock\_user\_database is created. **Important:** In a real application, this would be a persistent database (like SQL, NoSQL, etc.), not a temporary dictionary that gets reset every time the function is called.
   * It checks if the provided username already exists as a key in the mock\_user\_database.
   * If the username exists, it prints a message indicating that the username is taken and returns False.
   * If the username does not exist, it adds the username as a key and the password as the value to the mock\_user\_database, prints a success message, and returns True.
2. **login\_user(username, password) function:**
   * This function is designed to authenticate an existing user.
   * It also takes username and password as arguments.
   * Like the register\_user function, it uses a mock\_user\_database dictionary. Again, in a real application, this would access the persistent database.
   * It checks if the provided username exists as a key in the mock\_user\_database AND if the provided password matches the value associated with that username in the dictionary.
   * If both conditions are true (username exists and password matches), it prints a success message and returns True.
   * If either condition is false (username doesn't exist or password doesn't match), it prints an "Invalid username or password" message and returns False.
3. **if \_\_name\_\_ == "\_\_main\_\_": block:**
   * This block of code is for demonstration purposes and will only run when the script is executed directly (not when imported as a module).
   * It shows example usage of the register\_user and login\_user functions.
   * It attempts to register a user, then attempts to log in with the correct credentials, an incorrect password, and a non-existent user to show the different outcomes.

**Key takeaway:** The current implementation uses a simple dictionary for the user database, which means any registered users are lost when the program finishes running. For a real application, you would need to replace the dictionary with a persistent storage solution.

TASK DISCRIPTION#5

5) Analyzing Prompt Specificity: Improving Temperature Conversion Function with  
Clear Instructions





CODE EXPLANATION:

**Step 1 — Normalize Units**

from\_unit = from\_unit.upper()

to\_unit = to\_unit.upper()

* Converts both units to uppercase so it doesn’t matter if the user enters 'c' or 'C'.
* Example: "c" → "C"

**Step 2 — Convert Input to Celsius First**

if from\_unit == "C":

celsius = value

elif from\_unit == "F":

celsius = (value - 32) \* 5/9

elif from\_unit == "K":

celsius = value - 273.15

else:

raise ValueError("Invalid from\_unit. Use 'C', 'F', or 'K'.")

* No matter what the original unit is, we first convert it to **Celsius**.
* **Why?** → It’s easier to go from Celsius to any other unit.
* If the unit is invalid, we **stop the program** with an error.

**Step 3 — Convert Celsius to Target Unit**

if to\_unit == "C":

result = celsius

elif to\_unit == "F":

result = (celsius \* 9/5) + 32

elif to\_unit == "K":

result = celsius + 273.15

else:

raise ValueError("Invalid to\_unit. Use 'C', 'F', or 'K'.")

* Now we take the **Celsius value** and convert it to the requested **target unit**.
* Again, invalid inputs trigger an error.

**Step 4 — Round and Return**

return round(result, 2)

* Rounds the final temperature to **2 decimal places**.
* Returns the value to wherever the function was called.

**Example Usage**

print(convert\_temperature(100, 'C', 'F')) # 212.0

print(convert\_temperature(0, 'C', 'K')) # 273.15

print(convert\_temperature(32, 'F', 'C')) # 0.0

* Converts **100°C → Fahrenheit** (212°F).
* Converts **0°C → Kelvin** (273.15K).
* Converts **32°F → Celsius** (0°C)