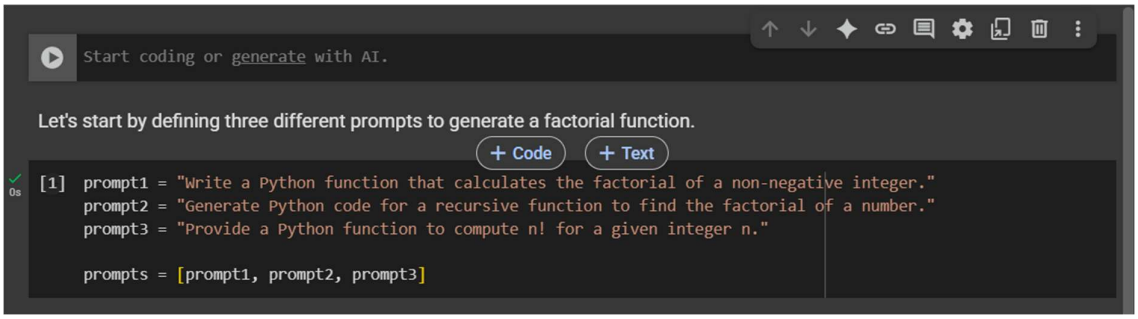
AI ASSISTED CODING

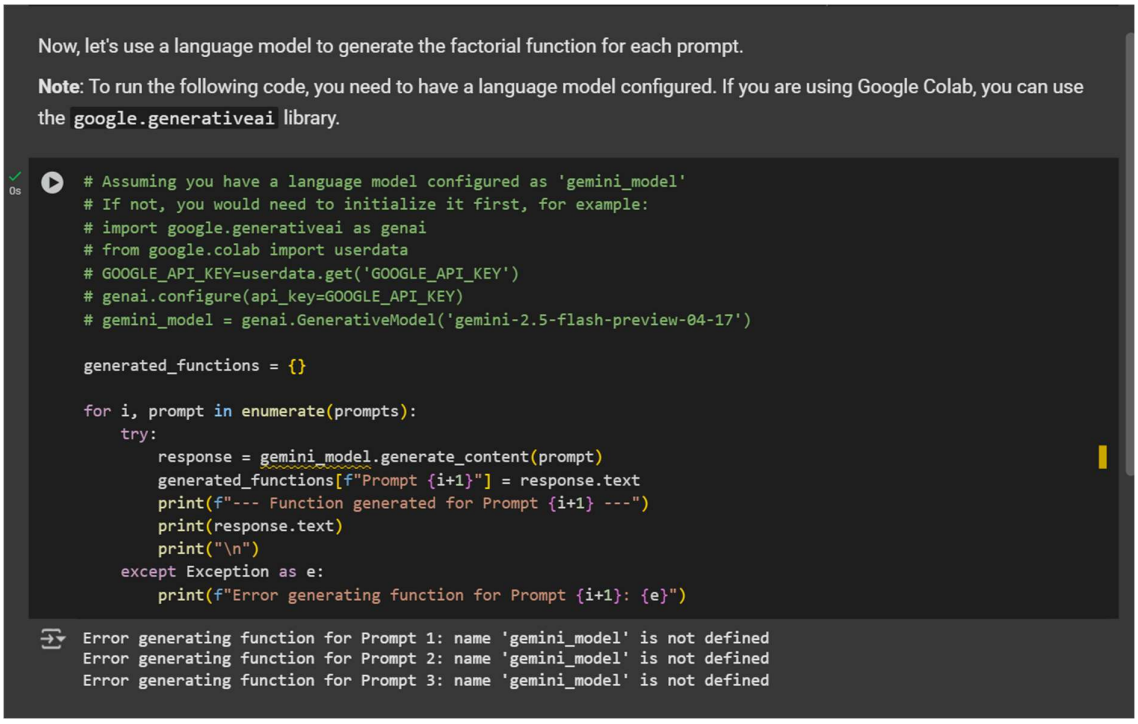
NAME:PULI.ALA

ASSIGNMENT 3.3

TASK DISCRIPTION#1

1)Try 3 different prompts to generate a factorial function





CODE EXPLANATION:

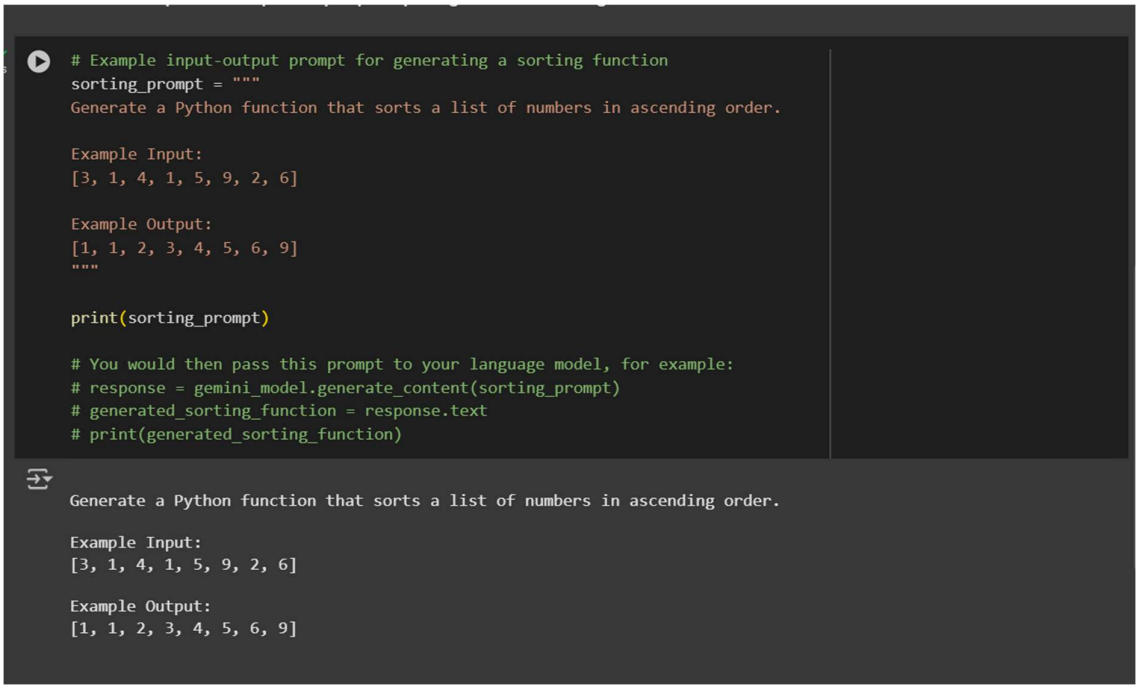
Here's a breakdown of the code cells:

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.

TASK DISCRIPTION#2

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



```
# Example input-output prompt for generating a sorting function
sorting_prompt = """
Generate a Python function that sorts a list of numbers in ascending order.

Example Input:
[3, 1, 4, 1, 5, 9, 2, 6]

Example Output:
[1, 1, 2, 3, 4, 5, 6, 9]
"""

print(sorting_prompt)

# You would then pass this prompt to your language model, for example:
# response = gemini_model.generate_content(sorting_prompt)
# generated_sorting_function = response.text
# print(generated_sorting_function)
```

```
Generate a Python function that sorts a list of numbers in ascending order.

Example Input:
[3, 1, 4, 1, 5, 9, 2, 6]

Example Output:
[1, 1, 2, 3, 4, 5, 6, 9]
```

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:

```python
def calculate_power_bill(kwh_used):
    # Rate tiers
    tier1_rate = 0.12
    tier2_rate = 0.15
    tier3_rate = 0.20

    # Fixed costs
    service_charge = 5.00
    tax_rate = 0.08

    # Calculate energy cost
    if kwh_used <= 500:
        energy_cost = kwh_used * tier1_rate
    elif kwh_used <= 1000:
        energy_cost = (500 * tier1_rate) + ((kwh_used - 500) * tier2_rate)
    else:
        energy_cost = (
            500 * tier1_rate +
            500 * tier2_rate +
            (kwh_used - 1000) * tier3_rate
        )

    # Subtotal
    subtotal = energy_cost + service_charge
    tax = subtotal * tax_rate
    total = subtotal + tax

    # Print itemized bill
    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}")
```

```python
# Example usage:
kwh = float(input("Enter energy used (kWh): "))
calculate_power_bill(kwh)
```

```
Enter energy used (kWh): 450
Energy cost:       $54.00
Service charge:    $5.00
Tax (8%):          $4.72
----------------------------
Total bill:        $63.72
```

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())

```python
# Function: register_user
# Description: Registers a new user in the system.
# Arguments:
#   - username (str): The desired username for the new user.
#   - password (str): The password for the new user.
# Returns:
#   - bool: True if registration is successful, False otherwise.
# Dependencies:
#   - Requires a user database or data structure for storage.
#   - Should handle potential errors like duplicate usernames.
# Linked Function: login_user - A user who registers may then want to log in.
def register_user(username, password):
    # In a real application, you would interact with a database here.
    # For demonstration, we'll use a simple dictionary as a mock database.
    mock_user_database = {} # This should be stored persistently in a real app

    if username in mock_user_database:
        print(f"Username '{username}' already exists.")
        return False
    else:
        mock_user_database[username] = password
        print(f"User '{username}' registered successfully.")
        return True

# Function: login_user
# Description: Authenticates an existing user.
# Arguments:
#   - username (str): The username of the user attempting to log in.
#   - password (str): The password provided by the user.
# Returns:
#   - bool: True if login is successful, False otherwise.
# Dependencies:
#   - Requires access to the user database used for registration.
#   - Should compare provided credentials with stored credentials.
# Linked Function: register_user - If login fails, the user might need to register.
```

```python
# Linked Function: register_user - If login fails, the user might need to register.
def login_user(username, password):
    # In a real application, you would interact with a database here.
    # For demonstration, we'll use a simple dictionary as a mock database.
    mock_user_database = {} # This should be stored persistently in a real app

    if username in mock_user_database and mock_user_database[username] == password:
        print(f"User '{username}' logged in successfully.")
        return True
    else:
        print("Invalid username or password.")
        return False

# Example usage (for demonstration)
if __name__ == "__main__":
    # Note: In a real scenario, mock_user_database would persist between function calls
    # and not be re-initialized each time.

    print("Attempting to register a user...")
    register_success = register_user("test_user", "password123")
    print(f"Registration successful: {register_success}\n")

    print("Attempting to login with registered user...")
    login_success = login_user("test_user", "password123")
    print(f"Login successful: {login_success}\n")

    print("Attempting to login with incorrect password...")
    login_fail_password = login_user("test_user", "wrong_password")
    print(f"Login successful: {login_fail_password}\n")

    print("Attempting to login with non-existent user...")
    login_fail_user = login_user("non_existent_user", "any_password")
    print(f"Login successful: {login_fail_user}\n")
```

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

```python
def convert_temperature(value, from_unit, to_unit):
    """
    Convert temperature between Celsius, Fahrenheit, and Kelvin.

    Parameters:
        value (float): Temperature value to convert.
        from_unit (str): Unit of the input value ('C', 'F', or 'K').
        to_unit (str): Unit to convert to ('C', 'F', or 'K').

    Returns:
        float: Converted temperature rounded to 2 decimal places.

    Raises:
        ValueError: If an unsupported unit is given.
    """
    from_unit = from_unit.upper()
    to_unit = to_unit.upper()

    # 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'.")

    # Convert from Celsius to target unit
    if to_unit == "C":
        result = celsius
    elif to_unit == "F":
        result = (celsius * 9/5) + 32
    elif to_unit == "K":
```

```python
        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'.")

    return round(result, 2)


# 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
```

```
212.0
273.15
0.0
```

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)