AI_ASSITED_CODING_LAB_ASSIGNMENT_3.3

Name : MERUGU RAKSHITH

Halticket no : 2403a52007

Batch no : CSAIB02

Task Description#1

- Try 3 different prompts to generate a factorial function.

prompt : generate a python code of Trying 3 different prompts to generate a factorial function.

```python
# Prompt 1: Using a loop

def factorial_loop(n):
  """Calculates the factorial of a non-negative integer using a loop."""
  if n < 0:
    return "Factorial is not defined for negative numbers"
  elif n == 0:
    return 1
  else:
    result = 1
    for i in range(1, n + 1):
      result *= i
    return result

print(f"Factorial of 5 using loop: {factorial_loop(5)}")

# Prompt 2: Using recursion

def factorial_recursion(n):
  """Calculates the factorial of a non-negative integer using recursion."""
  if n < 0:
    return "Factorial is not defined for negative numbers"
  elif n == 0:
    return 1
  else:
    return n * factorial_recursion(n - 1)
```

```
print(f"Factorial of 5 using recursion: {factorial_recursion(5)}")

# Prompt 3: Using math.factorial (built-in function)

import math

def factorial_builtin(n):
  """Calculates the factorial of a non-negative integer using
math.factorial."""
  if n < 0:
    return "Factorial is not defined for negative numbers"
  else:
    return math.factorial(n)

print(f"Factorial of 5 using built-in function: {factorial_builtin(5)}")
```

```
Factorial of 5 using loop: 120
Factorial of 5 using recursion: 120
Factorial of 5 using built-in function: 120
```

descrption : This code shows three different ways to calculate the factorial of a non-negative integer in Python.

Using a loop (factorial_loop): This function calculates the factorial iteratively. It initializes a result to 1 and then multiplies it by each integer from 1 up to the input number n. It also includes checks for negative numbers and the base case of 0. Using recursion (factorial_recursion): This function calculates the factorial recursively. It defines the factorial of 0 as 1 (the base case) and for any positive number n, it returns n multiplied by the factorial of n-1. It also handles negative inputs. Using math.factorial (factorial_builtin): This function uses the built-in factorial function from Python's math module. This is generally the most efficient and concise way to calculate factorials for non-negative integers. It also includes a check for negative inputs.

Task Description#2

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

prompt : generate a python code of providing a clear example input-output prompt to generate a sorting function.

1. List item
1. List item

```
# Prompt with Input-Output Example

# Input: A list of unsorted numbers
input_list = [5, 2, 8, 1, 9, 4]

# Expected Output: The sorted list
# output_list = [1, 2, 4, 5, 8, 9]
```

```
def bubble_sort(arr):
  """Sorts a list of numbers using the bubble sort algorithm."""
  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

# Generate the sorted output using the function
sorted_list = bubble_sort(input_list.copy()) # Use a copy to keep the
original input_list unchanged

print(f"Input list: {input_list}")
print(f"Sorted list: {sorted_list}")
```

```
Input list: [5, 2, 8, 1, 9, 4]
Sorted list: [1, 2, 4, 5, 8, 9]
```

descrption : This code provides an example of a sorting function using the bubble sort algorithm, along with a clear input and the resulting output.

Input and Expected Output: The code starts by defining input_list which is an unsorted list of numbers. It also shows the Expected Output as a comment, illustrating what the sorted list should look like. bubble_sort function: This function takes a list arr as input and sorts it in ascending order using the bubble sort algorithm. It iterates through the list multiple times. In each pass, it compares adjacent elements and swaps them if they are in the wrong order. This process continues until the list is sorted. Generating and Printing Output: The code then calls the bubble_sort function with a copy of the input_list to get the sorted_list. Finally, it prints both the original Input list and the Sorted list to show the result of the sorting operation.

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

prompt : generate a python code to calculate power bill

Task

Generate python code to calculate power bill based on a fixed rate of $0.15 per kWh.

Initial code (vague prompt)

Subtask:

Generate a basic Python code snippet based on the initial vague prompt, likely assuming a simple fixed rate calculation.

Reasoning: Generate a basic Python code snippet to calculate the power bill based on energy consumption and a fixed rate, as requested by the subtask.

```
def calculate_power_bill(kwh_consumed):
  """Calculates the power bill based on a fixed rate."""
  RATE_PER_KWH = 0.15  # Assumed fixed rate
  bill_amount = kwh_consumed * RATE_PER_KWH
  return bill_amount

# Example usage:
energy_used = 100  # in kWh
bill = calculate_power_bill(energy_used)
print(f"Energy consumed: {energy_used} kWh")
print(f"Estimated power bill: ${bill:.2f}")
```

```
Energy consumed: 100 kWh
Estimated power bill: $15.00
```

Refine prompt (adding detail)

Subtask:

Suggest ways to improve the prompt by adding details about the billing structure (e.g., "Generate python code to calculate power bill based on a fixed rate of $0.15 per kWh").

Reasoning: Describe the vagueness of the initial prompt and provide an improved prompt with a specific rate, explaining why the improvement is beneficial.

```
# 1. Describe the vagueness of the initial prompt.
print("The initial prompt 'Generate python code to calculate power bill' is
vague because it doesn't specify key details needed for the calculation.")
print("It doesn't mention:")
print("- The rate per kilowatt-hour (kWh).")
print("- If there are different rates for different consumption tiers.")
print("- Any fixed charges or taxes.")
print("- How the input (energy consumption) should be provided.")
print("- The desired format of the output.")
print("\nWithout these details, the AI has to make assumptions, which might
not align with the user's actual requirements.")

# 2. Provide an example of a more detailed prompt.
print("Example of a more detailed prompt:")
print("Generate python code to calculate power bill based on a fixed rate of
$0.15 per kWh. The function should take the total kWh consumed as input and
return the calculated bill amount as a floating-point number.")

# 3. Explain why adding this level of detail improves the prompt.
print("\nAdding this level of detail improves the prompt because it:")
print("- Provides a specific rate ($0.15 per kWh), eliminating ambiguity
about the calculation method.")
```

```
print("- Clearly defines the input (total kWh consumed) and the expected
output (floating-point bill amount).")
print("- Guides the AI to generate code that directly addresses the user's
specific scenario.")
print("- Reduces the need for the AI to guess or make assumptions, leading to
more accurate and relevant code.")
```

```
The initial prompt 'Generate python code to calculate power bill' is vague
because it doesn't specify key details needed for the calculation.
It doesn't mention:
- The rate per kilowatt-hour (kWh).
- If there are different rates for different consumption tiers.
- Any fixed charges or taxes.
- How the input (energy consumption) should be provided.
- The desired format of the output.

Without these details, the AI has to make assumptions, which might not align
with the user's actual requirements.
Example of a more detailed prompt:
Generate python code to calculate power bill based on a fixed rate of $0.15
per kWh. The function should take the total kWh consumed as input and return
the calculated bill amount as a floating-point number.

Adding this level of detail improves the prompt because it:
- Provides a specific rate ($0.15 per kWh), eliminating ambiguity about the
calculation method.
- Clearly defines the input (total kWh consumed) and the expected output
(floating-point bill amount).
- Guides the AI to generate code that directly addresses the user's specific
scenario.
- Reduces the need for the AI to guess or make assumptions, leading to more
accurate and relevant code.
```

Generate code (refined prompt)

Subtask:

Generate Python code that incorporates the details from the refined prompt.

Reasoning: Generate the Python code based on the refined prompt, including a function definition, rate, calculation, return value, example usage, and formatted output.

```
def calculate_power_bill_refined(kwh_consumed):
  """
  Calculates the power bill based on a fixed rate.

  Args:
    kwh_consumed: The total kilowatt-hours consumed (float or int).

  Returns:
```

```python
        The calculated bill amount as a floating-point number.
        Returns a string message if input is invalid.
    """
    if not isinstance(kwh_consumed, (int, float)) or kwh_consumed < 0:
        return "Invalid input: kWh consumed must be a non-negative number."

    RATE_PER_KWH = 0.15  # Fixed rate in USD per kWh
    bill_amount = kwh_consumed * RATE_PER_KWH
    return bill_amount

# Example usage with a sample kWh value
sample_kwh = 250
calculated_bill = calculate_power_bill_refined(sample_kwh)

# Print the calculated bill amount in a user-friendly format
if isinstance(calculated_bill, (int, float)):
    print(f"With {sample_kwh} kWh consumed, the estimated power bill is: ${calculated_bill:.2f}")
else:
    print(calculated_bill)

# Example with invalid input
invalid_kwh = -100
invalid_bill = calculate_power_bill_refined(invalid_kwh)
print(invalid_bill)

invalid_kwh_type = "abc"
invalid_bill_type = calculate_power_bill_refined(invalid_kwh_type)
print(invalid_bill_type)
```

```
With 250 kWh consumed, the estimated power bill is: $37.50
Invalid input: kWh consumed must be a non-negative number.
Invalid input: kWh consumed must be a non-negative number.
```

Further refine prompt (adding complexity)

Subtask:

Suggest further refinements to the prompt to include more complex billing scenarios (e.g., tiered rates, taxes, fees).

Reasoning: Describe the limitations of the current prompt and suggest refinements with examples for more complex billing scenarios.

```python
# 1. Describe how the current prompt is still limited.
print("While the refined prompt ('Generate python code to calculate power bill based on a fixed rate of $0.15 per kWh') is better than the initial vague one, it's still limited.")
print("It only covers a very basic scenario: a single, fixed rate for all energy consumption.")
print("Real-world power bills are often more complex and can include:")
print("- Tiered rates: Different rates apply to different blocks of energy consumption.")
print("- Fixed monthly fees: A constant charge regardless of energy usage.")
print("- Taxes: A percentage of the bill or a fixed amount added on top.")
print("- Other fees: Delivery charges, surcharges, etc.")
print("\nThe current prompt is insufficient to generate code that accounts for these common complexities.")

# 2. Provide examples of prompts that incorporate these complexities.

print("\nExamples of refined prompts for more complex scenarios:")

print("\nPrompt Example A (Tiered Rates):")
print("Generate python code to calculate power bill based on tiered rates. The first 100 kWh are billed at $0.10/kWh, and any consumption above 100 kWh is billed at $0.20/kWh. The function should take total kWh consumed as input and return the total bill amount.")

print("\nPrompt Example B (Fixed Fee + Taxes):")
print("Generate python code to calculate power bill. Include a fixed monthly service fee of $10.00 and add a 5% tax on the total consumption charge. The consumption is billed at a fixed rate of $0.15/kWh. The function should take total kWh consumed as input and return the total bill amount including the fixed fee and tax.")

print("\nPrompt Example C (Tiered Rates + Fixed Fee + Taxes):")
print("Generate python code to calculate power bill with tiered rates, a fixed fee, and taxes. The first 50 kWh are billed at $0.08/kWh, the next 150 kWh at $0.12/kWh, and any consumption above 200 kWh at $0.18/kWh. There is a fixed monthly fee of $5.00. A 7% tax is applied to the sum of the tiered consumption charges and the fixed fee. The function should take total kWh consumed as input and return the final bill amount.")


# 3. Explain how these detailed prompts lead to more sophisticated code.
print("\nThese more detailed prompts lead to more sophisticated and realistic code because they require the code to:")
print("- Implement conditional logic to apply different rates based on consumption thresholds (tiered rates).")
print("- Include multiple components in the calculation (consumption charge, fixed fee, taxes).")
print("- Potentially handle the order of calculations (e.g., calculate consumption charge, add fixed fee, then apply tax).")
print("- Manage multiple variables representing different rates, fees, and tax percentages.")
print("This moves the generated code beyond a simple multiplication to a more structured function that reflects actual billing structures.")
```

While the refined prompt ('Generate python code to calculate power bill based on a fixed rate of $0.15 per kWh') is better than the initial vague one, it's still limited.
It only covers a very basic scenario: a single, fixed rate for all energy consumption.
Real-world power bills are often more complex and can include:
- Tiered rates: Different rates apply to different blocks of energy consumption.
- Fixed monthly fees: A constant charge regardless of energy usage.
- Taxes: A percentage of the bill or a fixed amount added on top.
- Other fees: Delivery charges, surcharges, etc.

The current prompt is insufficient to generate code that accounts for these common complexities.

Examples of refined prompts for more complex scenarios:

Prompt Example A (Tiered Rates):
Generate python code to calculate power bill based on tiered rates. The first 100 kWh are billed at $0.10/kWh, and any consumption above 100 kWh is billed at $0.20/kWh. The function should take total kWh consumed as input and return the total bill amount.

Prompt Example B (Fixed Fee + Taxes):
Generate python code to calculate power bill. Include a fixed monthly service fee of $10.00 and add a 5% tax on the total consumption charge. The consumption is billed at a fixed rate of $0.15/kWh. The function should take total kWh consumed as input and return the total bill amount including the fixed fee and tax.

Prompt Example C (Tiered Rates + Fixed Fee + Taxes):
Generate python code to calculate power bill with tiered rates, a fixed fee, and taxes. The first 50 kWh are billed at $0.08/kWh, the next 150 kWh at $0.12/kWh, and any consumption above 200 kWh at $0.18/kWh. There is a fixed monthly fee of $5.00. A 7% tax is applied to the sum of the tiered consumption charges and the fixed fee. The function should take total kWh consumed as input and return the final bill amount.

These more detailed prompts lead to more sophisticated and realistic code because they require the code to:
- Implement conditional logic to apply different rates based on consumption thresholds (tiered rates).
- Include multiple components in the calculation (consumption charge, fixed fee, taxes).
- Potentially handle the order of calculations (e.g., calculate consumption charge, add fixed fee, then apply tax).
- Manage multiple variables representing different rates, fees, and tax percentages.

This moves the generated code beyond a simple multiplication to a more structured function that reflects actual billing structures.

Generate code (more complex prompt)

Subtask:

Generate Python code that handles the more complex billing rules.

Reasoning: Implement a Python function to calculate the power bill with tiered rates, a fixed fee, and taxes, including input validation and example usage, covering steps 1 through 8 of the subtask.

```
def calculate_power_bill_complex(kwh_consumed):
  """
  Calculates the power bill based on tiered rates, a fixed fee, and taxes.

  Tiered Rates:
  - First 50 kWh: $0.08/kWh
  - Next 150 kWh (51-200 kWh): $0.12/kWh
  - Above 200 kWh: $0.18/kWh

  Fixed Fee: $5.00

  Tax: 7% on the sum of consumption charges and fixed fee.

  Args:
    kwh_consumed: The total kilowatt-hours consumed (float or int).

  Returns:
    The calculated total bill amount as a floating-point number.
    Returns a string message if input is invalid.
  """
  # 3. Include input validation
  if not isinstance(kwh_consumed, (int, float)) or kwh_consumed < 0:
      return "Invalid input: kWh consumed must be a non-negative number."

  # 2. Implement complex billing scenarios (tiered rates, fixed fees, taxes)
  consumption_charge = 0

  # Tier 1
  if kwh_consumed <= 50:
    consumption_charge = kwh_consumed * 0.08
  # Tier 2
  elif kwh_consumed <= 200:
    consumption_charge = (50 * 0.08) + ((kwh_consumed - 50) * 0.12)
  # Tier 3
  else:
    consumption_charge = (50 * 0.08) + (150 * 0.12) + ((kwh_consumed - 200) *
0.18)
```

```
  # Add fixed fee
  FIXED_FEE = 5.00
  subtotal = consumption_charge + FIXED_FEE

  # Add tax
  TAX_RATE = 0.07
  tax_amount = subtotal * TAX_RATE
  total_bill = subtotal + tax_amount

  # 5. Return the calculated total bill amount
  return total_bill

# 6. Add example usage with a sample kWh value
sample_kwh_complex = 250
calculated_bill_complex = calculate_power_bill_complex(sample_kwh_complex)

# 7. Print the calculated bill amount in a user-friendly format
if isinstance(calculated_bill_complex, (int, float)):
    print(f"With {sample_kwh_complex} kWh consumed under complex rules, the
total power bill is: ${calculated_bill_complex:.2f}")
else:
    print(calculated_bill_complex)

# 8. Include an example with invalid input
invalid_kwh_complex = -50
invalid_bill_complex = calculate_power_bill_complex(invalid_kwh_complex)
print(invalid_bill_complex)

invalid_kwh_complex_type = "xyz"
invalid_bill_complex_type =
calculate_power_bill_complex(invalid_kwh_complex_type)
print(invalid_bill_complex_type)
```

```
With 250 kWh consumed under complex rules, the total power bill is: $38.52
Invalid input: kWh consumed must be a non-negative number.
Invalid input: kWh consumed must be a non-negative number.
```

Summary:

Data Analysis Key Findings

- The initial Python code successfully calculated the power bill based on a fixed rate of $0.15 per kWh.
- Refining the prompt by adding details like the specific rate, input type, and output format led to more accurate and relevant code generation.
- Further refining the prompt to include complex scenarios like tiered rates, fixed fees, and taxes required implementing conditional logic and multiple calculation components in the Python code.

- The final Python code successfully handled tiered rates (first 50 kWh at $0.08/kWh, next 150 kWh at $0.12/kWh, above 200 kWh at $0.18/kWh), a fixed fee ($5.00), and a 7% tax on the subtotal for a consumption of 250 kWh, resulting in a total bill of $38.52.
- Input validation was successfully added to the code to handle non-negative numeric inputs and provide informative messages for invalid types.

Insights or Next Steps

- Providing detailed requirements in the prompt is crucial for generating complex and accurate code solutions.
- The generated code can be extended to handle additional complexities such as different tax rules, peak/off-peak rates, or discounts.

Task Description#4

• Write structured comments to help AI generate two linked functions (e.g., login_user() and register_user()).

prompt : generate a python code that structured comments to help AI generate two linked functions (e.g., login_user() and register_user()).

```
# Function 1: register_user
# Purpose: To register a new user with a username and password.
# Arguments:
# - username (str): The desired username for the new user.
# - password (str): The desired password for the new user.
# Returns:
# - bool: True if registration is successful, False otherwise (e.g., username
already exists).
# Dependencies:
# - Requires a mechanism to store user data (e.g., a dictionary, database).
# - Should check if the username already exists before registering.
# - Should store the username and a hashed version of the password securely.

def register_user(username, password, user_database):
  """Registers a new user if the username is not already taken."""
  if username in user_database:
    print(f"Error: Username '{username}' already exists.")
    return False
  # In a real application, password should be hashed for security
  user_database[username] = password # Storing plain password for
demonstration
  print(f"User '{username}' registered successfully.")
  return True

# Function 2: login_user
# Purpose: To authenticate an existing user based on their username and
password.
# Arguments:
```

```python
# - 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 (username and password match), False otherwise.
# Dependencies:
# - Requires access to the stored user data (used by register_user).
# - Should compare the provided password with the stored (hashed) password.

def login_user(username, password, user_database):
  """Authenticates a user based on username and password."""
  if username not in user_database:
    print(f"Error: Username '{username}' not found.")
    return False
  # In a real application, provided password should be hashed and compared to stored hash
  if user_database[username] == password: # Comparing plain password for demonstration
    print(f"User '{username}' logged in successfully.")
    return True
  else:
    print("Error: Incorrect password.")
    return False

# Example Usage:
user_database = {} # Simulate a simple user database using a dictionary

# Registering users
register_user("alice", "secure_password_123", user_database)
register_user("bob", "another_secure_pwd", user_database)
register_user("alice", "attempt_duplicate_reg", user_database) # Attempt to register duplicate

print("\n--- Attempting Logins ---")

# Logging in users
login_user("alice", "secure_password_123", user_database) # Successful login
login_user("bob", "wrong_password", user_database)     # Incorrect password
login_user("charlie", "any_password", user_database)   # Non-existent user
```

```
User 'alice' registered successfully.
User 'bob' registered successfully.
Error: Username 'alice' already exists.

--- Attempting Logins ---
User 'alice' logged in successfully.
Error: Incorrect password.
Error: Username 'charlie' not found.

False
```

descrption : This code defines two Python functions, register_user and login_user, which are designed to simulate basic user registration and login functionality using a simple dictionary as a user database.

Here's a breakdown:

register_user(username, password, user_database): Purpose: This function attempts to register a new user. How it works: It checks if the username already exists as a key in the user_database dictionary. If the username exists, it prints an error and returns False. If the username does not exist, it adds the username as a key and the password as its value to the user_database. It prints a success message and returns True. Note: In a real application, the password should be securely hashed before storing it, not stored in plain text as shown here for demonstration. login_user(username, password, user_database): Purpose: This function attempts to authenticate an existing user. How it works: It checks if the username exists as a key in the user_database dictionary. If the username does not exist, it prints an error and returns False. If the username exists, it compares the provided password with the password stored in the user_database for that username. If the passwords match, it prints a success message and returns True. If the passwords do not match, it prints an error and returns False. Note: Similar to registration, in a real application, the provided password should be hashed and compared to the stored hashed password. Example Usage: A simple dictionary user_database is initialized to simulate storing user data. The code then demonstrates registering two users (alice and bob) and attempting to register a duplicate user (alice again). Finally, it shows examples of attempting to log in with correct credentials, incorrect credentials, and a non-existent user. The print statements show the outcome of each operation.

Task Description#5

- Analyzing Prompt Specificity: Improving Temperature Conversion Function with Clear Instructions

prompt : generate a python code that improves temperature Conversion Function with Clear Instructions

```python
# Function: celsius_to_fahrenheit
# Purpose: To convert a temperature from Celsius to Fahrenheit.
# Arguments:
# - celsius (float): The temperature in degrees Celsius.
# Returns:
# - float: The equivalent temperature in degrees Fahrenheit.
# Formula: (Celsius * 9/5) + 32

def celsius_to_fahrenheit(celsius):
  """Converts Celsius to Fahrenheit."""
  # Check if input is a number
  if not isinstance(celsius, (int, float)):
    return "Invalid input: Please provide a numeric value for Celsius."
  fahrenheit = (celsius * 9/5) + 32
  return fahrenheit
```

```python
# Example Usage:
temp_celsius = 25
temp_fahrenheit = celsius_to_fahrenheit(temp_celsius)
print(f"{temp_celsius}°C is equal to {temp_fahrenheit}°F")

temp_celsius_invalid = "abc"
temp_fahrenheit_invalid = celsius_to_fahrenheit(temp_celsius_invalid)
print(temp_fahrenheit_invalid)
```

```
25°C is equal to 77.0°F
Invalid input: Please provide a numeric value for Celsius.
```

descrption : Comments: The code starts with structured comments (#) that act as clear instructions or documentation for the function:

Function: celsius_to_fahrenheit: States the name of the function.

Purpose: To convert a temperature from Celsius to Fahrenheit.: Explains what the function does.

Arguments:: Details the expected input.

- celsius (float): The temperature in degrees Celsius.: Specifies the name and expected data type of the argument.

Returns:: Describes the output.

- float: The equivalent temperature in degrees Fahrenheit.: Specifies the expected data type of the return value.

Formula: (Celsius * 9/5) + 32: Provides the mathematical formula used for the conversion.

Function Definition (def celsius_to_fahrenheit(celsius):): This defines the function named celsius_to_fahrenheit that takes one argument, celsius. Docstring ("""Converts Celsius to Fahrenheit."""): This is a brief description of the function's purpose, useful for documentation. Input Validation (if not isinstance(celsius, (int, float)): ...): This line checks if the input celsius is not an integer or a float. If it's not a number, it returns an "Invalid input" message. This makes the function more robust. Conversion Calculation (fahrenheit = (celsius * 9/5) + 32): If the input is valid, this line applies the standard formula to convert the Celsius temperature to Fahrenheit. Return Value (return fahrenheit): The function returns the calculated Fahrenheit temperature. Example Usage: The code then shows how to use the function: It sets temp_celsius to 25 and calls the function to get the Fahrenheit equivalent, printing the result. It also demonstrates the input validation by calling the function with a non-numeric input ("abc") and printing the error message returned by the function.