Assignment 3.3

Name: P.Umesh Reddy   Roll.no: 2403A510F9

Batch.no: 06   course: AI Assisted coding

Task Description#1

• Try 3 different prompts to generate a factorial function.

Expected Output#1

• Comparison of AI-generated code styles

Prompt 1 — Beginner-friendly & recursive

Prompt:

"Write a simple, beginner-friendly Python function `factorial(n)` using recursion. Include a clear docstring, basic input validation for non-negative integers, and examples in the docstring."

Likely output:

python   Copy   Edit

```python
def factorial(n: int) -> int:
    """
    Return n! (n factorial) using a simple recursive approach.

    Args:
        n (int): Non-negative integer.

    Returns:
        int: The factorial of n.
```

```
Examples:
    >>> factorial(0)
    1
    >>> factorial(5)
    120
"""
if not isinstance(n, int):
    raise ValueError("n must be an integer")
if n < 0:
    raise ValueError("n must be non-negative")
if n in (0, 1):
    return 1
return n * factorial(n - 1)
```

Explanation of the Three Prompts and Code Styles

🔹 Prompt 1 — Beginner-friendly & Recursive

- **Why this style?** The wording "simple, beginner-friendly" and "using recursion" nudges the AI to generate a function that mirrors the **mathematical definition** of factorial:

$$n! = n \times (n-1)! \quad \text{with base case } 0! = 1$$

- **Characteristics of the code:**
  - Recursive calls (`factorial(n-1)`)
  - Clear docstring with examples
  - Input validation written step-by-step
- **Best for:** Teaching programming fundamentals, recursion, and mathematical concepts.
- **Drawback:** Not efficient for large inputs (Python's recursion limit ~1000).

Prompt 2 — Performance-leaning, iterative, type-hinted

Prompt:

"Implement a production-quality `factorial(n: int) -> int` in Python without recursion. Use an iterative approach (or `math.prod`) for speed and to avoid recursion depth limits. Add precise type hints and raise `ValueError` for invalid input."

Likely output:

python Copy Edit

```python
from math import prod

def factorial(n: int) -> int:
    """
    Compute n! iteratively for performance and stability.

    Args:
        n: Non-negative integer.

    Returns:
        The factorial of n.
```

```python
    Raises:
        ValueError: If n is negative or not an integer.
    """
    if not isinstance(n, int):
        raise ValueError("n must be an integer")
    if n < 0:
        raise ValueError("n must be non-negative")
    if n < 2:
        return 1
    return prod(range(2, n + 1))
```

🔹 **Prompt 2 — Performance-leaning, Iterative**

- **Why this style?** The instruction "production-quality, no recursion, use `math.prod`" directs the AI toward **robust and scalable code** suitable for real-world use.
- **Characteristics of the code:**
  - Iterative approach (or `math.prod`, which is C-optimized)
  - Explicit type hints (`n: int -> int`)
  - Handles large `n` efficiently without stack overflow
- **Best for:** Real applications where performance and reliability matter.
- **Drawback:** Less "elegant" compared to recursion from a mathematical teaching standpoint.

Prompt 3 — Functional/one-liner style with `reduce`

Prompt:
"Write a concise, functional-style Python factorial using `functools.reduce` in as few lines as reasonable. Validate input and keep the implementation compact."

Likely output:

python Copy Edit

```python
from functools import reduce
import operator as op

def factorial(n: int) -> int:
    if not isinstance(n, int) or n < 0:
        raise ValueError("n must be a non-negative integer")
    return 1 if n < 2 else reduce(op.mul, range(2, n + 1), 1)
```

🔹 **Prompt 3 — Functional/One-liner**

- **Why this style?** The phrasing "concise, functional-style, use `reduce`" nudges AI to compress logic into a **compact expression.**
- **Characteristics of the code:**
  - Uses `functools.reduce` with `operator.mul` to multiply sequence elements
  - Very compact (basically a one-liner return)
  - Validation handled in a single condition
- **Best for:** Developers who prefer functional programming or concise code.
- **Drawback:** Slightly harder for beginners to read; more "Python tricks" than fundamentals.

Comparison of AI-generated code styles

Aspect	Prompt 1 (Recursive)	Prompt 2 (Iterative/ `math.prod`)	Prompt 3 (Functional/ `reduce`)
Readability	Very clear for learners; mirrors math definition	Clean and explicit; easy to follow	Compact and clever, slightly denser
Error handling	Strong (type + range checks)	Strong (type + range checks)	Strong but terse (combined check)
Performance	Good for small n; recursion overhead	Fast; no recursion; uses C-optimized `prod`	Fast; Python-level loop via `reduce`
Scalability	Limited by recursion depth (~1000)	Scales to very large n (Python big ints)	Scales well; similar to iterative
Pythonic style	Classic teaching example	"Practical Python" / production-leaning	Functional elegance / golfy
Dependencies	None	`math.prod` (std lib)	`functools` , `operator` (std lib)
Use when...	Explaining factorial conceptually	You want robust, fast, safe default	You prefer concise functional style

Task Description#2

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

Expected Output#2

• Functional sorting code from AI

```python
def sort_list(arr):
    """
    Sorts a list of integers in ascending order using the Merge Sort algorithm.

    Merge Sort is a 'divide and conquer' algorithm that recursively divides
    the list into halves until it has single-element lists, and then merges
    those smaller lists back together in sorted order.

    Args:
        arr (list): The list of integers to be sorted.

    Returns:
        list: A new list with the integers sorted in ascending order.
    """
    # Base case: If the list has 0 or 1 element, it's already sorted.
    if len(arr) <= 1:
        return arr

    # Find the middle point of the list to divide it into two halves.
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Recursively sort both halves.
    left_sorted = sort_list(left_half)
    right_sorted = sort_list(right_half)
```

```python
    # Merge the two sorted halves back into a single sorted list.
    return merge(left_sorted, right_sorted)

def merge(left, right):
    """
    Merges two sorted lists into a single sorted list.

    This function is a key part of the Merge Sort algorithm. It compares
    the elements of the two sorted lists and appends the smaller one to
    the new merged list until one of the lists is empty.

    Args:
        left (list): The first sorted list.
        right (list): The second sorted list.

    Returns:
        list: The new, merged and sorted list.
    """
    merged_list = []
    i = 0  # Index for the left list
    j = 0  # Index for the right list
```

```python
    # Compare elements from both lists and append the smaller one.
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged_list.append(left[i])
            i += 1
        else:
            merged_list.append(right[j])
            j += 1

    # Append any remaining elements from the left list.
    while i < len(left):
        merged_list.append(left[i])
        i += 1

    # Append any remaining elements from the right list.
    while j < len(right):
        merged_list.append(right[j])
        j += 1

    return merged_list
```

```
        j += 1

    return merged_list

# Example usage:
my_list = [38, 27, 43, 3, 9, 82, 10]
sorted_list = sort_list(my_list)
print(f"Original list: {my_list}")
print(f"Sorted list: {sorted_list}")
```

Prompt

"Generate a Python function named `sort_list` that takes a list of integers as input and return a new list with the integers sorted in ascending order. Use the **Merge Sort** algorithm, and includ comments explaining how the algorithm works."

Functional Merge Sort Code

Here is the functional Python code for the Merge Sort algorithm, generated in response to the prompt.

Explanation of the Code

The generated code successfully implements the Merge Sort algorithm. The code is split into two functions to clearly separate the two main phases of the algorithm: **dividing** and **merging**.

- `sort_list` **function**: This is the recursive part of the algorithm. It continually divides the input list into two halves until it is left with lists containing a single element. A list with one element is considered sorted, which serves as the **base case** for the recursion.
- `merge` **function**: This function takes the smaller, sorted lists and **merges** them back together. It works by comparing the first element of each list and adding the smaller one to a new list. This process is repeated until all elements from both lists are in the new, sorted list.

The comments within the code are transparent and accurately describe each step of the process, making it easy to understand the algorithm's logic.

Task Description#3

• Start with the vague prompt "Generate python code to calculate power bill" and improve it step-by-step

Expected Output#3

• Enhanced AI output with clearer prompts

```
def calculate_power_bill(kilowatt_hours, rate_per_kwh):
    # Calculate the total bill by multiplying the consumed kilowatt_hours by the rate per kWh.
    total_bill = kilowatt_hours * rate_per_kwh
    return total_bill

# Example usage:
kwh_used = 150
rate = 0.12  # $0.12 per kWh
bill_amount = calculate_power_bill(kwh_used, rate)
print(f"Your power bill is: ${bill_amount:.2f}")
```

Vague Prompt and Initial Output

Prompt: "Generate python code to calculate power bill"

This is a very vague prompt. A power bill calculation can vary significantly based on location, rate structures (flat rate, tiered, time-of-use), and additional fees. Without more information, the AI can only make assumptions.

Initial Output (Based on Vague Prompt):

The initial code would likely be a simple function that assumes a single, fixed rate per kilowatt-hour (kWh). It would also probably require a hardcoded rate and perhaps a simple input for kWh consumed.

Step 1: Clarifying the Prompt

Enhanced Prompt: "Generate a Python function to calculate a power bill. The function should take the `kilowatt_hours` consumed and a `rate_per_kwh` as input and return the total bill amount. Add a comment explaining the calculation."

By adding specific parameters and a return value, we have created a much clearer prompt. The AI now has a precise set of instructions for the function's signature and its purpose.

Step 2: Adding Complexity

Enhanced Prompt: "Update the Python function to calculate a power bill. The function should now account for a tiered rate system. The first 100 kilowatt-hours are charged at a lower rate ($0.10/kWh), and any usage above 100 kilowatt-hours is charged at a higher rate ($0.15/kWh). The function should still take `kilowatt_hours` as input and return the total bill amount. Provide comments for each tier of calculation."

This prompt introduces a specific business logic (tiered pricing), which is a very common feature of utility billing. This is a significant improvement over the basic prompt.

Step 3: Adding a Final, Complex Requirement

Final Enhanced Prompt: "Generate a Python script to calculate a power bill. The script should define a function `calculate_bill` that takes `kilowatt_hours` as input. This function must apply a tiered rate system: $0.10/kWh for the first 50 kWh, $0.12/kWh for the next 150 kWh, and $0.15/kWh for all usage above 200 kWh. The script should also include a fixed monthly service fee of $5.00. The script must ask the user for their kilowatt-hour usage and print a formatted bill summary, showing the usage, service fee, and total amount."

This final prompt is comprehensive. It specifies the function name, its parameters, the exact tiered rate structure, a fixed fee, and the desired output format, including user interaction. This level of detail ensures the generated code will be highly specific and useful.

Task Description#4

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

Expected Output#4

- Consistent functions with shared logic

```python
# A dictionary to simulate a user database.
# This is a shared data structure that both functions will interact with.
users = {}

def register_user(username, password):
    """
    Registers a new user by adding them to the shared 'users' dictionary.

    Args:
        username (str): The username for the new account.
        password (str): The password for the new account.

    Returns:
        str: A message indicating success or failure.
    """
    # 1. Check if the username already exists in the shared 'users' dictionary.
    if username in users:
        # 1.1. If it exists, return a specific error message.
        return "Error: This username is already taken."
    else:
        # 2. If the username does not exist, add the new user
        #    to the 'users' dictionary.
        users[username] = password
```

```python
        # 2.1. Return a success message.
        return f"Success: User '{username}' registered successfully."


def login_user(username, password):
    """
    Logs in a user by verifying their credentials against the shared 'users' dictionary.

    Args:
        username (str): The username to log in with.
        password (str): The password to verify.

    Returns:
        str: A message indicating a successful or failed login.
    """
    # 1. Check if the username exists in the same 'users' dictionary.
    if username not in users:
        # 1.1. If the username is not found, return an error message.
        return "Error: Username not found."

    # 2. If the username is found, check if the provided password matches
    #    the password stored in the 'users' dictionary for that username.
    stored_password = users[username]
    if password == stored_password:
```

```python
        # 2.1. If the passwords match, return a success message.
        return f"Success: Welcome, {username}!"
    else:
        # 2.2. If the passwords do not match, return an error message.
        return "Error: Incorrect password."

# Example Usage:
print(register_user("john.doe", "password123"))
print(login_user("john.doe", "password123"))
print(login_user("john.doe", "wrong_password"))
print(register_user("john.doe", "new_password"))
```

Explanation of Structured Comments and Shared Logic

The structured comments serve as a clear, step-by-step guide for the AI, ensuring consistency across both functions.

1. Shared Data Model: The first crucial element is the comment defining the `users` dictionary as a **"shared data structure that both functions will interact with."** This single comment establishes the central point of truth for both `register_user()` and `login_user()`. It tells the AI that any user-related data must be read from or written to this dictionary.

2. Consistent Validation Logic: The comments then guide the validation process for each function in a logical, parallel manner.

- The `register_user()` comments instruct the AI to check for `username in users`.
- The `login_user()` comments instruct the AI to check for `username not in users`. These inverse checks, though separate, are directly linked by their reference to the same data source. This ensures that a user can't be registered if their name exists, and a user can't log in if their name doesn't exist, maintaining the integrity of the system.

3. Specificity in Error Handling: The comments also dictate the specific return messages for each condition. For example, `login_user()` must return **"Error: Username not found,"** and `register_user()` must return **"Error: This username is already taken."** This level of detail ensures the functions produce predictable and user-friendly output that differentiates between specific failure modes. The structured comments provided a blueprint for the AI to follow, resulting in two well-connected and functionally consistent functions.

Task Description#5

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

Expected Output#5

• Code quality difference analysis for various prompts

```python
def celsius_to_fahrenheit(celsius: float) -> float:
    """
    Converts a temperature from Celsius to Fahrenheit.

    This function uses the standard formula to convert a given Celsius temperature
    into its Fahrenheit equivalent.

    Args:
        celsius (float): The temperature in degrees Celsius.

    Returns:
        float: The converted temperature in degrees Fahrenheit.

    Raises:
        TypeError: If the input 'celsius' is not a numeric type.
    """
    # Validate the input to ensure it is a numeric type (int or float).
    if not isinstance(celsius, (int, float)):
        raise TypeError("Input must be a numeric value (int or float).")
```

```python
    # Apply the conversion formula: F = (C * 9/5) + 32.
    fahrenheit = (celsius * 9/5) + 32

    return fahrenheit

# Example usage with valid input:
c_temp = 25
f_temp = celsius_to_fahrenheit(c_temp)
print(f"{c_temp}°C is {f_temp}°F")

# Example with invalid input to demonstrate error handling:
try:
    celsius_to_fahrenheit("twenty-five")
except TypeError as e:
    print(f"Caught an expected error: {e}")
```

1. Vague Prompt

Prompt: "Generate a function to convert temperature."

A vague prompt like this leaves almost all decisions to the AI. It doesn't specify the units, the programming language, or the desired output format.

Expected Output:

The AI would likely default to a simple Celsius-to-Fahrenheit conversion, as it's a very common example. The resulting code would be functional but basic, lacking comments, documentation, and error handling. It would look something like this:

Python

```python
def convert_temp(c):
    f = (c * 9/5) + 32
    return f
```

Code Quality Analysis: This code works, but its quality is poor. The function and variable names (`convert_temp`, `c`, `f`) are not descriptive. There are no comments to explain the formula, no docstrings to describe its purpose, and it will fail if the user inputs a non-numeric value (like 'abc'). It is an example of a **brittle** function—it works for a narrow, expected input but breaks easily.

2. Specific Prompt

Prompt: "Generate a Python function named `celsius_to_fahrenheit` that converts a temperature from Celsius to Fahrenheit. The function should accept one argument, `celsius`, and include a docstring that explains its purpose, arguments, and return value. The function must also validate that the input is a numeric type and raise a `TypeError` with an informative message if it is not."

This prompt is a significant improvement. It is now a **complete specification** that dictates the function's name, its parameters, its behavior, and its documentation requirements. It also introduces a critical instruction for **error handling**.

Code Quality Analysis:

This code is far superior to the initial version. The differences in quality can be analyzed across several key areas:

- **Readability and Documentation**: The function has a clear, descriptive name (`celsius_to_fahrenheit`) and a comprehensive **docstring.** The docstring explains the function's purpose, its arguments, what it returns, and what errors it might raise. This makes the code **self-documenting**, allowing other developers (or you, in the future) to understand its purpose without reading the implementation details.
- **Robustness and Error Handling**: The `if` statement checks the type of the input. This is a crucial addition that prevents a **runtime crash** if the user provides an unexpected value. Instead of crashing, the function raises a `TypeError` with a helpful message, allowing the program to handle the error gracefully. This makes the function much more reliable in real-world applications.

- **Correctness and Maintainability**: The code is not only correct in its calculation but is also formatted to **PEP 8 standards** (like `fahrenheit = ...`). The use of clear variable names and comments on the validation and calculation steps makes the code easier to maintain and debug.

In summary, the specific prompt yielded a function that is not only functional but also **robust,**