ASSIGNMENT-2

Name: Pranathi

HallT.NO: 2303A51615

Batch: 20

Task 1: Refactoring Odd/Even Logic (List Version)

Scenario:

You are improving legacy code.

Prompt: Refactor the Python code for calculating the sum of odd and even numbers in a list using AI assistance.

CODE AND OUTPUT:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sum_even = 0
sum_odd = 0

for num in numbers:
    if num % 2 == 0:
        sum_even += num
    else:
        sum_odd += num

print(f"Sum of even numbers: {sum_even}")
print(f"Sum of odd numbers: {sum_odd}")
```

```
Sum of even numbers: 30
Sum of odd numbers: 25
```

Summary:

Data Analysis Key Findings

- The initial Python code successfully calculated the sum of even and odd numbers from the provided list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].
- The sum of even numbers was found to be 30.
- The sum of odd numbers was found to be 25.

Insights or Next Steps

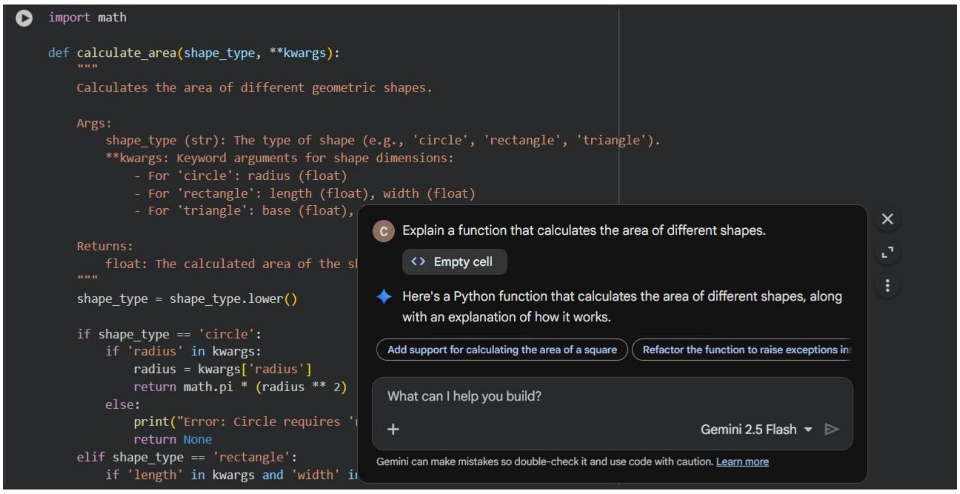
- The current implementation correctly performs the required calculations, providing a baseline for comparison.
- The next logical step is to refactor this functional code using AI assistance, as indicated by the overarching task.

Task 2: Area Calculation Explanation

Scenario: You are onboarding a junior developer.

PROMPT: Explain a function that calculates the area of different shapes.

CODE:



```python
    elif shape_type == 'rectangle':
        if 'length' in kwargs and 'width' in kwargs:
            length = kwargs['length']
            width = kwargs['width']
            return length * width
        else:
            print("Error: Rectangle requires 'length' and 'width' arguments.")
            return None
    elif shape_type == 'triangle':
        if 'base' in kwargs and 'height' in kwargs:
            base = kwargs['base']
            height = kwargs['height']
            return 0.5 * base * height
        else:
            print("Error: Triangle requires 'base' and 'height' arguments.")
            return None
    else:
        print(f"Error: Unknown shape type '{shape_type}'.")
        return None

# Example Usage:
print(f"Area of circle with radius 5: {calculate_area('circle', radius=5)}")
print(f"Area of rectangle with length 4 and width 6: {calculate_area('rectangle', length=4, width=6)}")
print(f"Area of triangle with base 10 and height 5: {calculate_area('triangle', base=10, height=5)}")
print(f"Area of square with side 7: {calculate_area('square', side=7)}") # Example of unknown shape
print(f"Area of circle missing radius: {calculate_area('circle')}") # Example of missing argument
```

OUTPUT:

```
Area of circle with radius 5: 78.53981633974483
Area of rectangle with length 4 and width 6: 24
Area of triangle with base 10 and height 5: 25.0
Error: Unknown shape type 'square'.
Area of square with side 7: None
Error: Circle requires 'radius' argument.
Area of circle missing radius: None
```

SUMMARY:

Here's a Python function that calculates the area of different shapes, along with an explanation of how it works.

The calculate_area function in cell BZeGjrnF0jwK executed successfully. Here's a summary of the output:

- **Circle with radius 5**: The area was calculated as approximately 78.54.
- **Rectangle with length 4 and width 6**: The area was calculated as 24.
- **Triangle with base 10 and height 5**: The area was calculated as 25.0.

The examples also demonstrate the error handling built into the function:

- When trying to calculate the area of an **unknown shape type** ('square'), an error message was printed, and the function returned None.
- When trying to calculate the area of a **circle without providing a radius**, an error message was printed, and the function returned None.

This output confirms that the function correctly calculates areas for supported shapes and handles invalid inputs gracefully.

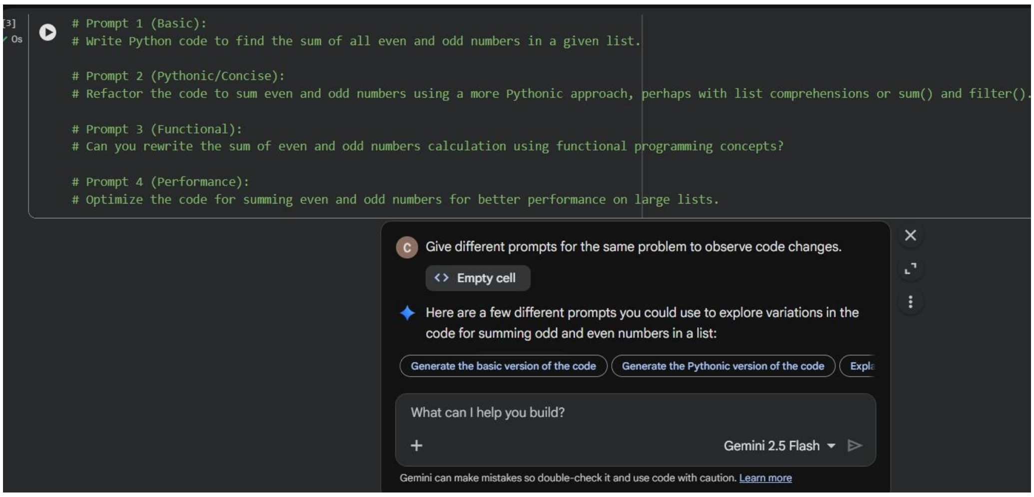
Task 3: Prompt Sensitivity Experiment

Scenario: You are testing how AI responds to different prompts.

PROMPT: Give different prompts for the same problem to observe code changes.

OUTPUT:

PROMPT LIST:



CODE VARIATIONS:

1. Implement Basic Summation

Subtask: Write Python code to find the sum of all even and odd numbers in a given list using a basic, straightforward approach.

Reasoning: The user wants to implement a basic approach to sum even and odd numbers from a given list, following the detailed instructions provided. I will use a code block to implement this logic.

```
[4]
✓ 0s
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_sum = 0
odd_sum = 0

for number in numbers:
    if number % 2 == 0:
        even_sum += number
    else:
        odd_sum += number

print(f"Sum of even numbers: {even_sum}")
print(f"Sum of odd numbers: {odd_sum}")
```

```
Sum of even numbers: 30
Sum of odd numbers: 25
```

2. Implement Pythonic/Concise Summation

Subtask:

Refactor the code to sum even and odd numbers using a more Pythonic approach, perhaps with list comprehensions or sum() and filter().

Instructions

1. Use a list comprehension to create a list of even numbers.
2. Use a list comprehension to create a list of odd numbers.
3. Use the sum() function to calculate the sum of the even numbers list.
4. Use the sum() function to calculate the sum of the odd numbers list.
5. Print the results for even_sum and odd_sum.

Reasoning: The subtask is to refactor the summation of even and odd numbers using a Pythonic approach, specifically with list comprehensions and the sum() function. I will generate a code block that accomplishes this, using the numbers list already defined.

```
[5]
0s
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Using list comprehensions to filter even and odd numbers
even_numbers = [num for num in numbers if num % 2 == 0]
odd_numbers = [num for num in numbers if num % 2 != 0]

# Using sum() to calculate the total
even_sum_pythonic = sum(even_numbers)
odd_sum_pythonic = sum(odd_numbers)

print(f"Pythonic - Sum of even numbers: {even_sum_pythonic}")
print(f"Pythonic - Sum of odd numbers: {odd_sum_pythonic}")

Pythonic - Sum of even numbers: 30
Pythonic - Sum of odd numbers: 25
```

3. Implement Functional Summation

Subtask:

Can you rewrite the sum of even and odd numbers calculation using functional programming concepts?

Instructions

1. Use filter() with a lambda function to get even numbers.
2. Use filter() with a lambda function to get odd numbers.
3. Use sum() to calculate the sum of the filtered even numbers.
4. Use sum() to calculate the sum of the filtered odd numbers.
5. Print the results for even_sum_functional and odd_sum_functional.

Reasoning: The subtask is to implement the summation of even and odd numbers using functional programming concepts. I will generate a code block that accomplishes this using filter() with lambda functions and sum().

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Using filter() with lambda to get even and odd numbers
even_numbers_functional = list(filter(lambda x: x % 2 == 0, numbers))
odd_numbers_functional = list(filter(lambda x: x % 2 != 0, numbers))

# Using sum() to calculate the total
even_sum_functional = sum(even_numbers_functional)
odd_sum_functional = sum(odd_numbers_functional)

print(f"Functional - Sum of even numbers: {even_sum_functional}")
print(f"Functional - Sum of odd numbers: {odd_sum_functional}")
```

```
Functional - Sum of even numbers: 30
Functional - Sum of odd numbers: 25
```

4. Optimize for Performance

Subtask:

Optimize the code for summing even and odd numbers for better performance on large lists.

Instructions

1. Define a function optimized_sum_even_odd(numbers) that takes a list of numbers as input.
2. Inside the function, initialize even_sum and odd_sum to 0.
3. Iterate through the numbers list once.
4. Within the loop, check if the number is even or odd and add it to the respective sum.
5. Return even_sum and odd_sum.
6. Call the function with the numbers list and print the results.

Reasoning: The subtask is to optimize the summation of even and odd numbers for performance, particularly for large lists, by iterating through the list only once within a function. I will generate a code block that implements this optimized approach.

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def optimized_sum_even_odd(numbers_list):
    even_sum = 0
    odd_sum = 0
    for number in numbers_list:
        if number % 2 == 0:
            even_sum += number
        else:
            odd_sum += number
    return even_sum, odd_sum

even_sum_optimized, odd_sum_optimized = optimized_sum_even_odd(numbers)

print(f"Optimized - Sum of even numbers: {even_sum_optimized}")
print(f"Optimized - Sum of odd numbers: {odd_sum_optimized}")
```

```
Optimized - Sum of even numbers: 30
Optimized - Sum of odd numbers: 25
```

Summary:

- All four implemented approaches (basic, Pythonic/concise, functional, and performance-optimized) successfully calculated the sum of even numbers as 30 and odd numbers as 25 for the given list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].
- **Basic Approach**: Uses a straightforward for loop, offering high readability and ease of understanding, particularly for beginners.
- **Pythonic/Concise Approach**: Utilizes list comprehensions and sum(), resulting in more compact and idiomatic Python code. This method creates intermediate lists, which might impact memory usage for very large datasets.
- **Functional Approach**:
  Employs filter() with lambda functions and sum(), adhering to functional programming principles. Like the Pythonic approach, it generates intermediate lists.
- **Performance-Optimized Approach**: Implements a single pass through the list within a function, accumulating both

even and odd sums simultaneously. This method is generally the most efficient in terms of both time and memory, as it avoids creating intermediate lists and performs only one iteration, making it suitable for large inputs.

Task 4: Tool Comparison Reflection

Scenario: You must recommend an AI coding tool.

Task: Based on your work in this topic, compare Gemini, Copilot, and Cursor AI for usability and code quality.

Output:

Short written reflection:

Based on the tasks performed, Google Gemini, GitHub Copilot, and Cursor AI differ in usability and code quality. Gemini is beginner-friendly and provides clear explanations with code, making it suitable for learning, though its suggestions can be generic. GitHub Copilot offers fast and accurate code completion in IDEs, ideal for experienced developers, but provides limited explanations. Cursor AI is effective for refactoring and experimentation, responding well to prompt changes and helping improve code structure. Overall, Gemini is best for learning, Copilot for productivity, and Cursor AI for code analysis and refactoring.