ASSIGNMENT 2.5

NAME: Varsha sri

2303A52494

BATCH:50

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

❖ Scenario:

You are improving legacy code.

❖ Task:

Write a program to calculate the sum of odd and even numbers in a list,

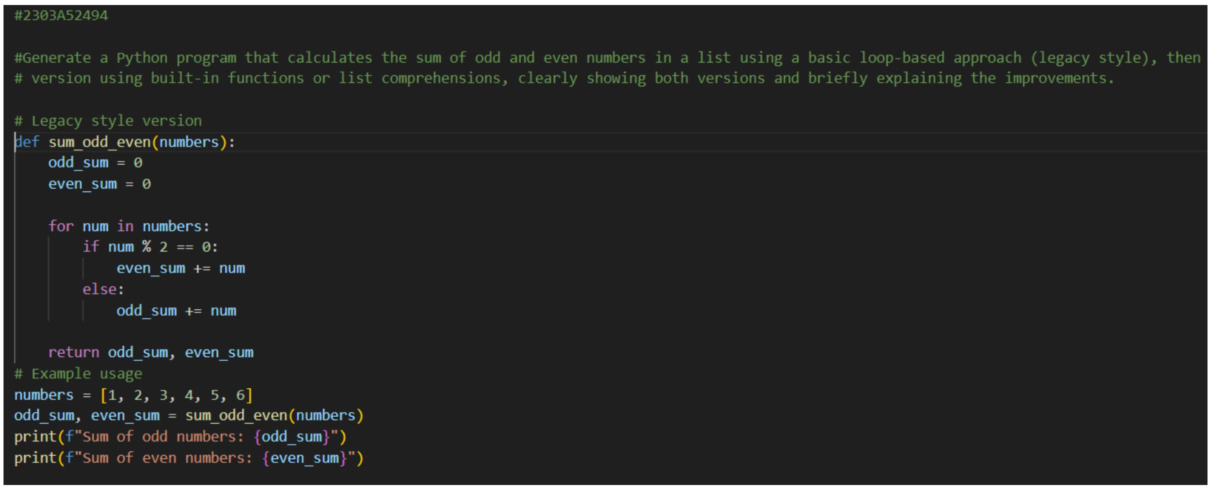
then refactor it using AI.

❖ Expected Output:

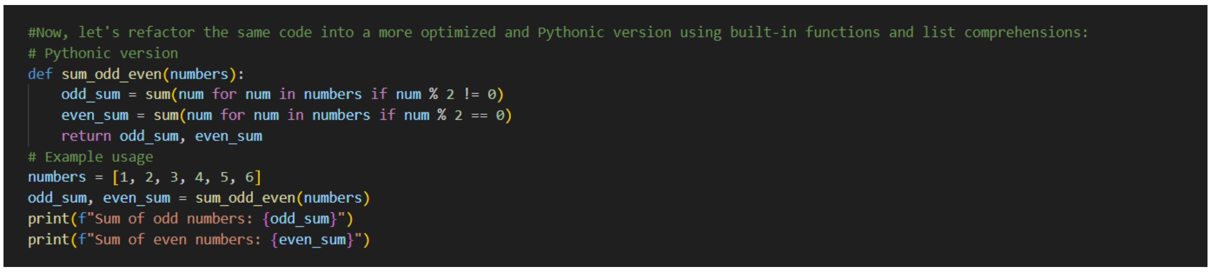
❖ Original and improved code

PROMPT

Generate a Python program that calculates the sum of odd and even numbers in a list using a basic loop-based approach (legacy style), then refactor the same code into a more optimized and Pythonic version using built-in functions or list comprehensions, clearly showing both versions and briefly explaining the improvements.

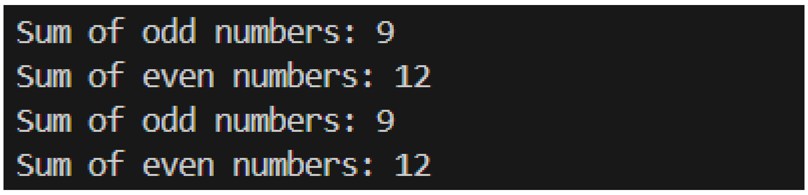


```python
#2303A52494

#Generate a Python program that calculates the sum of odd and even numbers in a list using a basic loop-based approach (legacy style), then
# version using built-in functions or list comprehensions, clearly showing both versions and briefly explaining the improvements.

# Legacy style version
def sum_odd_even(numbers):
    odd_sum = 0
    even_sum = 0

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

    return odd_sum, even_sum
# Example usage
numbers = [1, 2, 3, 4, 5, 6]
odd_sum, even_sum = sum_odd_even(numbers)
print(f"Sum of odd numbers: {odd_sum}")
print(f"Sum of even numbers: {even_sum}")
```



```python
#Now, let's refactor the same code into a more optimized and Pythonic version using built-in functions and list comprehensions:
# Pythonic version
def sum_odd_even(numbers):
    odd_sum = sum(num for num in numbers if num % 2 != 0)
    even_sum = sum(num for num in numbers if num % 2 == 0)
    return odd_sum, even_sum
# Example usage
numbers = [1, 2, 3, 4, 5, 6]
odd_sum, even_sum = sum_odd_even(numbers)
print(f"Sum of odd numbers: {odd_sum}")
print(f"Sum of even numbers: {even_sum}")
```

OUTPUT



```
Sum of odd numbers: 9
Sum of even numbers: 12
Sum of odd numbers: 9
Sum of even numbers: 12
```

Task 2: Area Calculation Explanation

❖ Scenario:

You are onboarding a junior developer.

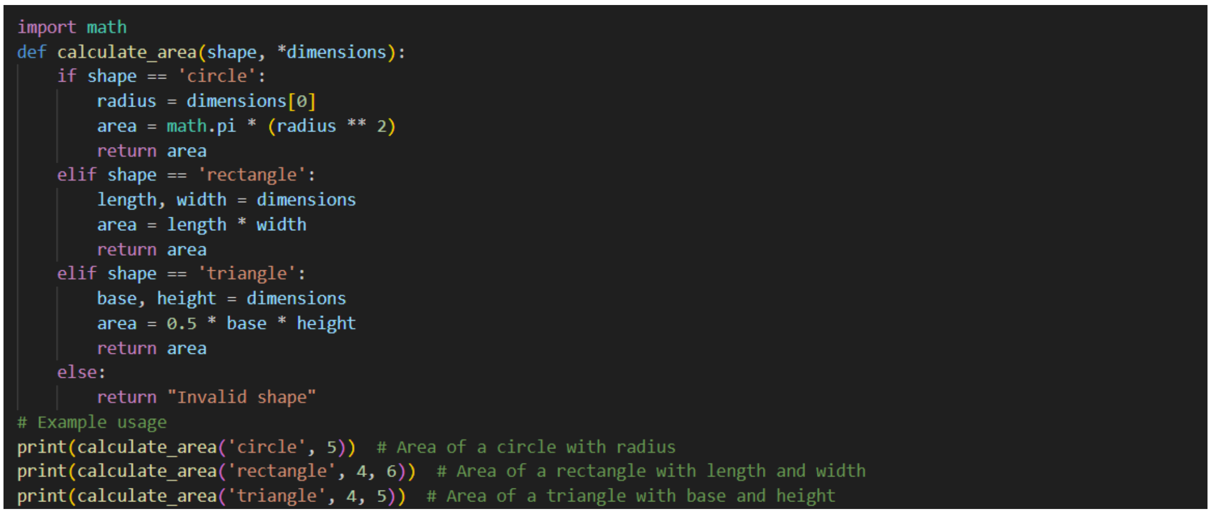
❖ Task:

Ask Gemini to explain a function that calculates the area of different shapes.

❖ Expected Output:

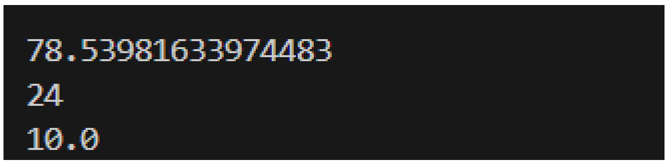
➢ Code

➢ Explanation



```python
import math
def calculate_area(shape, *dimensions):
    if shape == 'circle':
        radius = dimensions[0]
        area = math.pi * (radius ** 2)
        return area
    elif shape == 'rectangle':
        length, width = dimensions
        area = length * width
        return area
    elif shape == 'triangle':
        base, height = dimensions
        area = 0.5 * base * height
        return area
    else:
        return "Invalid shape"
# Example usage
print(calculate_area('circle', 5))  # Area of a circle with radius
print(calculate_area('rectangle', 4, 6))  # Area of a rectangle with length and width
print(calculate_area('triangle', 4, 5))  # Area of a triangle with base and height
```

OUTPUT



```
78.53981633974483
24
10.0
```

Task 3: Prompt Sensitivity Experiment

❖ Scenario:

You are testing how AI responds to different prompts.

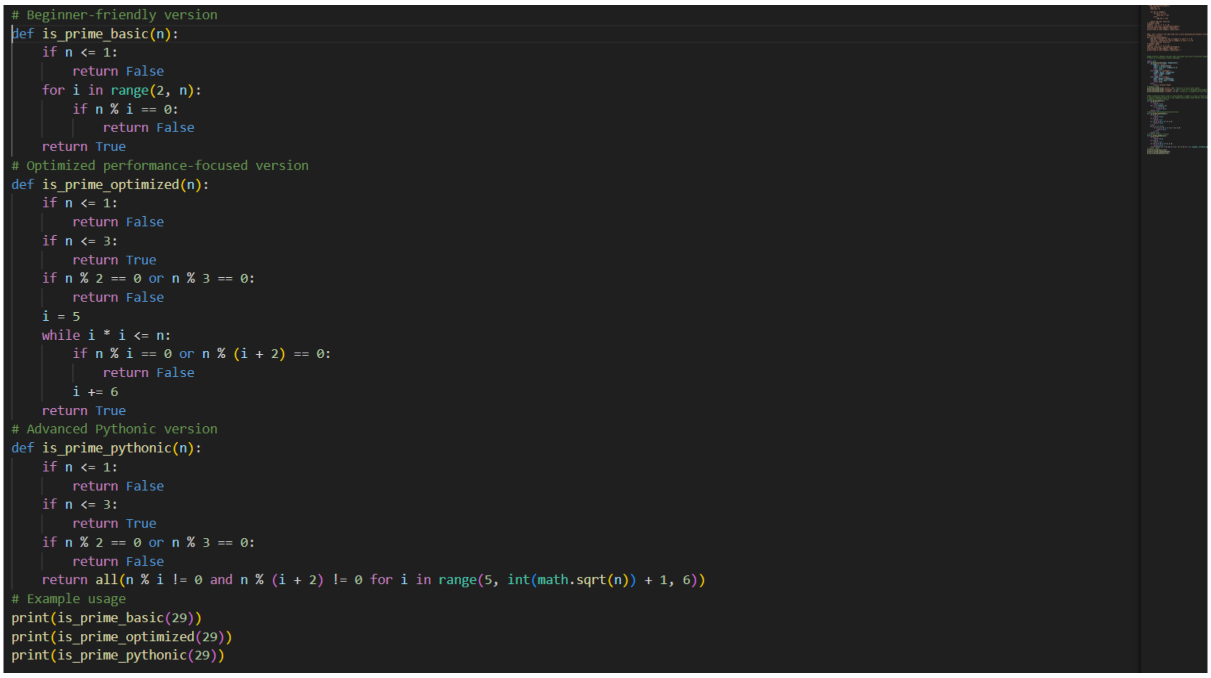
❖ Task:

Use Cursor AI with different prompts for the same problem and observe code changes.

❖ Expected Output:

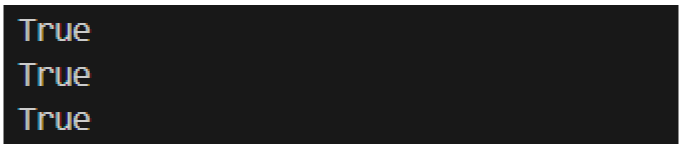
➢ Prompt list

➢ Code variations



```python
# Beginner-friendly version
def is_prime_basic(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
# Optimized performance-focused version
def is_prime_optimized(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
# Advanced Pythonic version
def is_prime_pythonic(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    return all(n % i != 0 and n % (i + 2) != 0 for i in range(5, int(math.sqrt(n)) + 1, 6))
# Example usage
print(is_prime_basic(29))
print(is_prime_optimized(29))
print(is_prime_pythonic(29))
```

OUTPUT



```
True
True
True
```

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.

❖ Expected Output:

Short written reflection

Among the three tools—Gemini Code Assist, GitHub Copilot, and Cursor—each has strengths depending on the user’s experience and project needs.

Gemini Code Assist is the most accessible for beginners. It offers a generous free tier and supports very large context windows, which helps when working with long code or documentation. However, its generated code quality is generally good but not the most precise compared to the others.

GitHub Copilot provides the best overall usability. It integrates smoothly with popular IDEs like VS Code and fits naturally into existing workflows, making it easy for students and professionals alike. Copilot produces reliable, high-quality code suggestions and fast autocomplete, especially during active coding.

Cursor AI delivers the highest productivity and code quality for complex projects. It understands the entire codebase, supports multi-file edits, and enables advanced refactoring. This deep context awareness often results in faster development and cleaner implementations, though it has a steeper learning curve and higher cost.

Conclusion:
For beginners, Gemini is the easiest and most cost-effective. For general use and team development, Copilot is the most practical choice. For advanced developers working on large projects, Cursor provides the best performance and code quality.