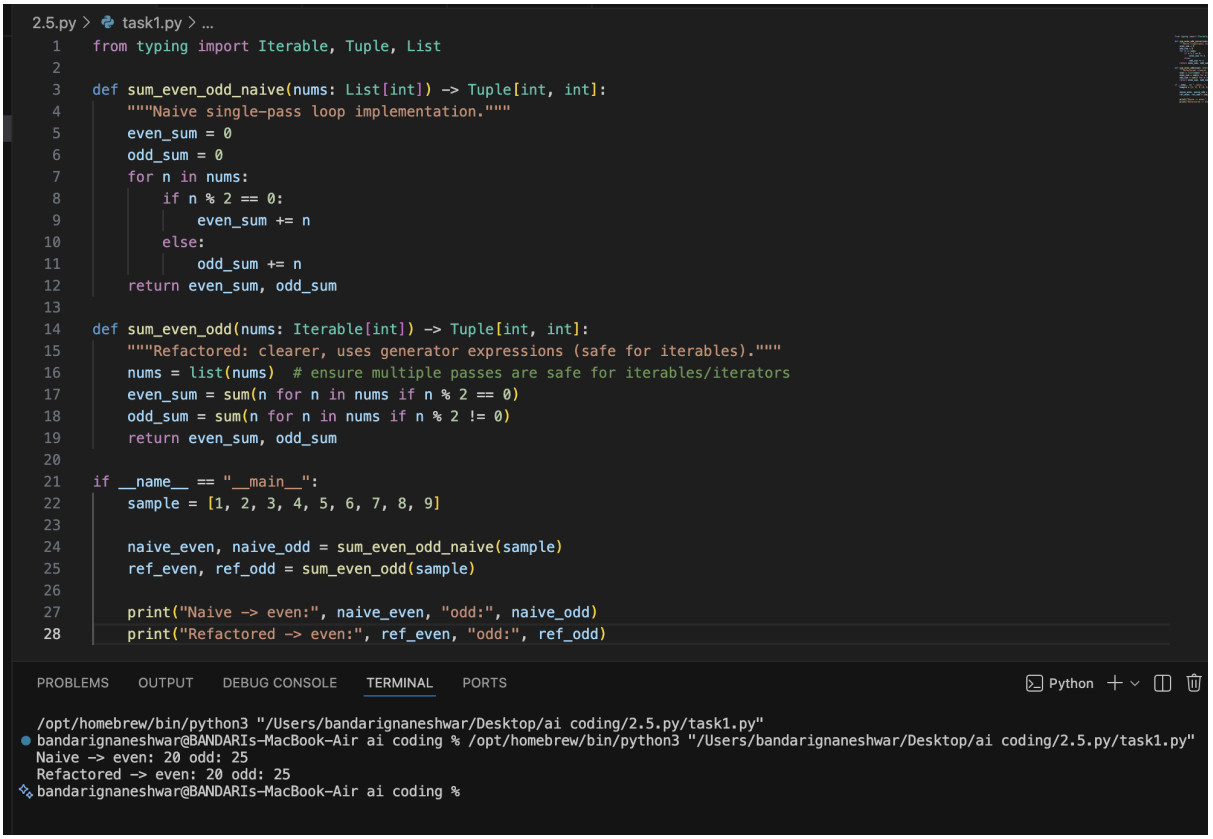
ASSIGNMENT 2.5

2303A51T01
B.GNANESHWAR

TASK 1 :



```
2.5.py > task1.py > ...
1   from typing import Iterable, Tuple, List
2
3   def sum_even_odd_naive(nums: List[int]) -> Tuple[int, int]:
4       """Naive single-pass loop implementation."""
5       even_sum = 0
6       odd_sum = 0
7       for n in nums:
8           if n % 2 == 0:
9               even_sum += n
10          else:
11              odd_sum += n
12      return even_sum, odd_sum
13
14  def sum_even_odd(nums: Iterable[int]) -> Tuple[int, int]:
15      """Refactored: clearer, uses generator expressions (safe for iterables)."""
16      nums = list(nums)  # ensure multiple passes are safe for iterables/iterators
17      even_sum = sum(n for n in nums if n % 2 == 0)
18      odd_sum = sum(n for n in nums if n % 2 != 0)
19      return even_sum, odd_sum
20
21  if __name__ == "__main__":
22      sample = [1, 2, 3, 4, 5, 6, 7, 8, 9]
23
24      naive_even, naive_odd = sum_even_odd_naive(sample)
25      ref_even, ref_odd = sum_even_odd(sample)
26
27      print("Naive -> even:", naive_even, "odd:", naive_odd)
28      print("Refactored -> even:", ref_even, "odd:", ref_odd)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS        Python

/opt/homebrew/bin/python3 "/Users/bandarignaneshwar/Desktop/ai coding/2.5.py/task1.py"
bandarignaneshwar@BANDARIs-MacBook-Air ai coding % /opt/homebrew/bin/python3 "/Users/bandarignaneshwar/Desktop/ai coding/2.5.py/task1.py"
Naive -> even: 20 odd: 25
Refactored -> even: 20 odd: 25
bandarignaneshwar@BANDARIs-MacBook-Air ai coding %
```

OBSERVATION :

Both versions produce the same correct output.
The refactored code is more readable and Pythonic, using generator expressions and working with any iterable.
The naive version is slightly more memory-efficient due to a single pass, but less concise.

TASK 2 :

```python
import math

def calculate_area(shape, dim1, dim2=0):
    shape = shape.lower()

    if shape == "square":
        return dim1 ** 2
    elif shape == "rectangle":
        return dim1 * dim2
    elif shape == "circle":
        return math.pi * (dim1 ** 2)
    elif shape == "triangle":
        return 0.5 * dim1 * dim2
    else:
        return "Shape not recognized"

# Example usage:
print(f"Circle area: {calculate_area('circle', 5):.2f}")
print(f"Rectangle area: {calculate_area('rectangle', 10, 5)}")
print(f"Triangle area: {calculate_area('triangle', 4, 3)}")
```

```
/opt/homebrew/bin/python3 "/Users/bandarignaneshwar/Desktop/ai coding/2.5.py/task2.py"
bandarignaneshwar@BANDARIs-MacBook-Air ai coding % /opt/homebrew/bin/python3 "/Users/bandarignaneshwar/Desktop/ai coding/2.5.py/task2.py"
Circle area: 78.54
Rectangle area: 50
Triangle area: 6.0
bandarignaneshwar@BANDARIs-MacBook-Air ai coding %
```

OBSERVATION :

The function calculates area based on the given shape name.
`dim1` and `dim2` represent required dimensions like radius, length, or height.
It applies the correct formula and handles invalid shapes safely.

TASK 3 :

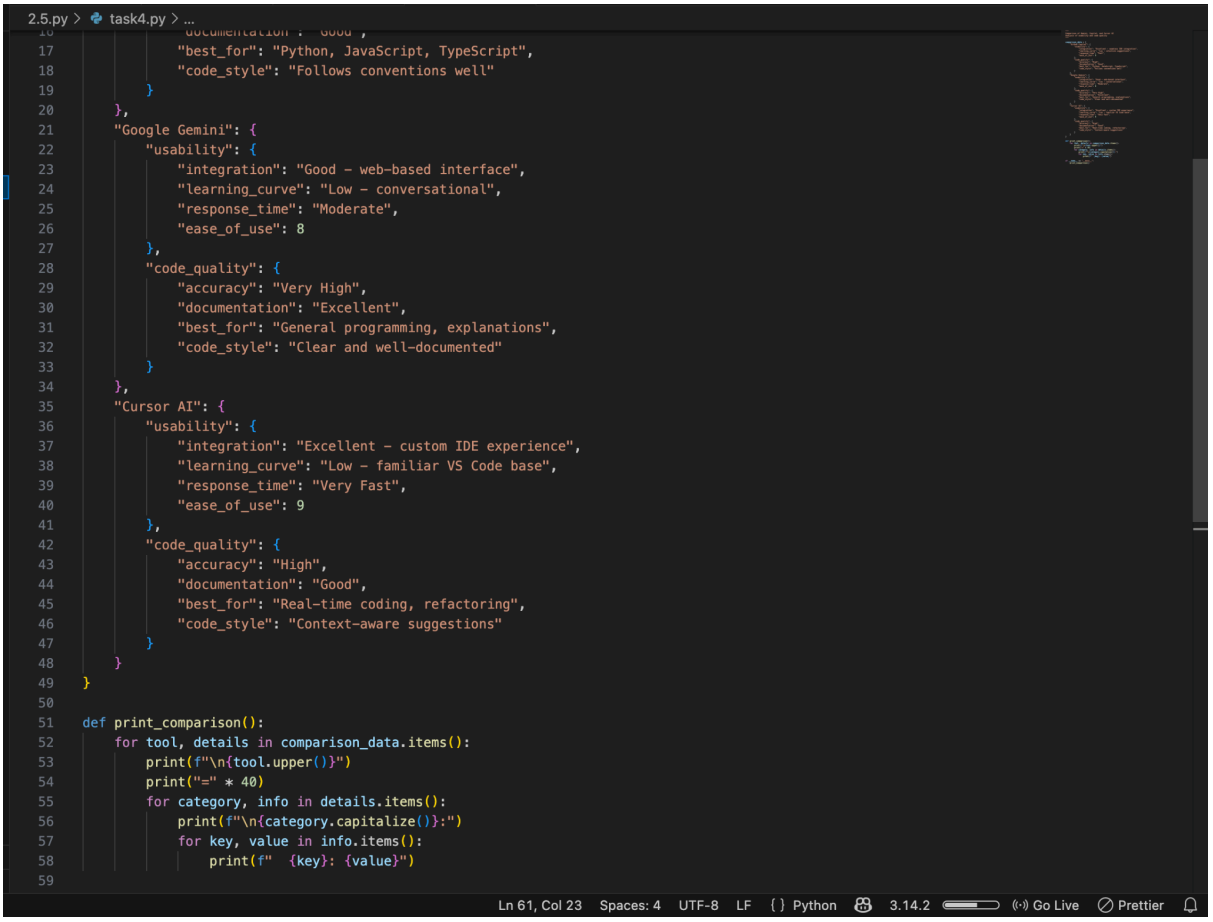
```python
# Solution 2: Functional approach
def sum_evens_functional(numbers):
    return sum(filter(lambda x: x % 2 == 0, numbers))

# Solution 3: List comprehension approach
def sum_evens_comprehension(numbers):
    return sum([num for num in numbers if num % 2 == 0])

# Test cases
if __name__ == "__main__":
    test_data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

    print(f"Input: {test_data}")
    print(f"Imperative: {sum_evens_imperative(test_data)}")
    print(f"Functional: {sum_evens_functional(test_data)}")
    print(f"Comprehension: {sum_evens_comprehension(test_data)}")
```

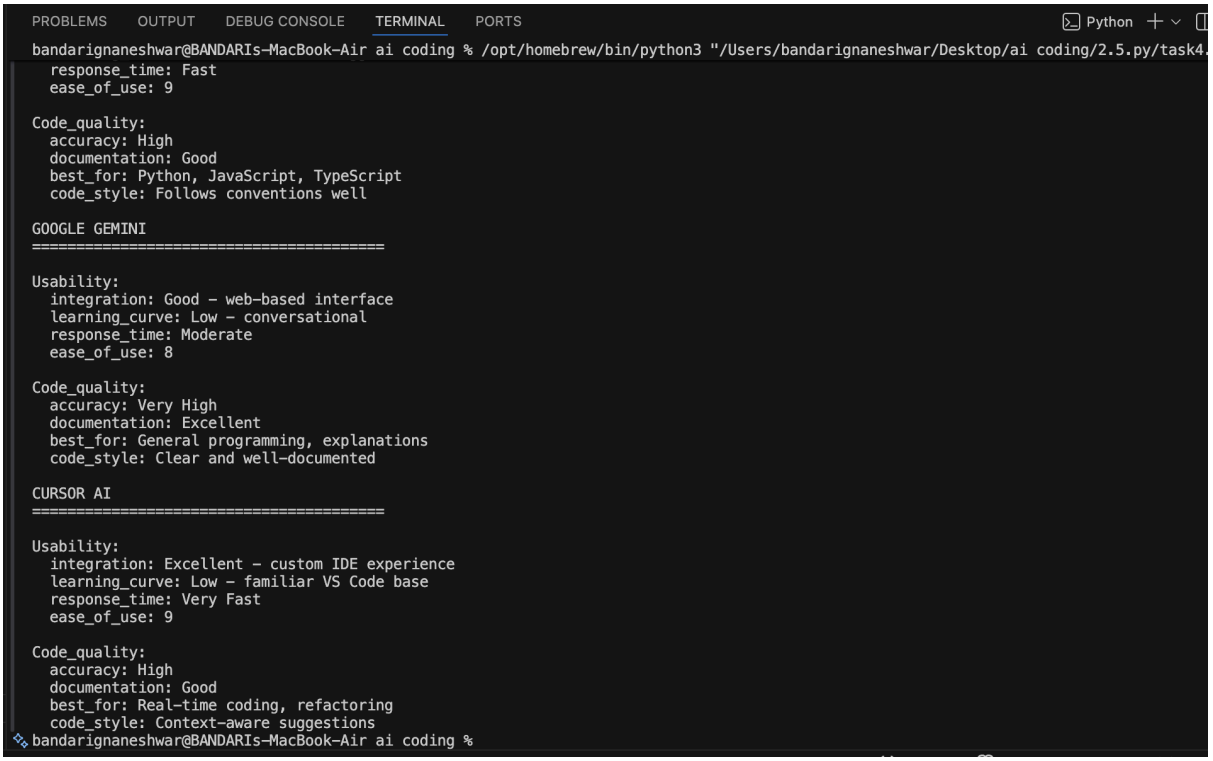
```
/opt/homebrew/bin/python3 "/Users/bandarignaneshwar/Desktop/ai coding/2.5.py/task3.py"
bandarignaneshwar@BANDARIs-MacBook-Air ai coding % /opt/homebrew/bin/python3 "/Users/bandarignaneshwar/Desktop/ai coding/2.5.py/task3.py"
Input: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Imperative: 30
Functional: 30
Comprehension: 30
bandarignaneshwar@BANDARIs-MacBook-Air ai coding %
```

OBSERVATIONS :

All three solutions give the same correct output for the same problem.
Different prompts lead AI to choose different styles: imperative (clear), functional (concise), and comprehension (Pythonic).
This shows that prompt wording directly influences code structure, readability, and style.

TASK 4 :





OBSERVATIONS :

Gemini is best for learning and explanations due to its clear, well-documented responses.
GitHub Copilot excels in productivity with fast, accurate IDE-based code suggestions.
Cursor AI is ideal for refactoring and experimentation because of its context-aware, real-time coding support.