

## 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 :

```
2.5.py > task2.py > ...
1  import math
2
3  def calculate_area(shape, dim1, dim2=0):
4      shape = shape.lower()
5
6      if shape == "square":
7          return dim1 ** 2
8      elif shape == "rectangle":
9          return dim1 * dim2
10     elif shape == "circle":
11         return math.pi * (dim1 ** 2)
12     elif shape == "triangle":
13         return 0.5 * dim1 * dim2
14     else:
15         return "Shape not recognized"
16
17 # Example usage:
18 print(f"Circle area: {calculate_area('circle', 5):.2f}")
19 print(f"Rectangle area: {calculate_area('rectangle', 10, 5)}")
20 print(f"Triangle area: {calculate_area('triangle', 4, 3)}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python + - [ ] [X]

```
/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 :

```
18 # Solution 2: Functional approach
19 def sum_evens_functional(numbers):
20     return sum(filter(lambda x: x % 2 == 0, numbers))
21
22 # Solution 3: List comprehension approach
23 def sum_evens_comprehension(numbers):
24     return sum([num for num in numbers if num % 2 == 0])
25
26 # Test cases
27 if __name__ == "__main__":
28     test_data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
29
30     print(f"Input: {test_data}")
31     print(f"Imperative: {sum_evens_imperative(test_data)}")
32     print(f"Functional: {sum_evens_functional(test_data)}")
33     print(f"Comprehension: {sum_evens_comprehension(test_data)}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python + - [ ] [X]

```
/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 :

```
2.5.py > task4.py > ...
16         "documentation": "Good",
17         "best_for": "Python, JavaScript, TypeScript",
18         "code_style": "Follows conventions well"
19     },
20 },
21 "Google Gemini": {
22     "usability": {
23         "integration": "Good - web-based interface",
24         "learning_curve": "Low - conversational",
25         "response_time": "Moderate",
26         "ease_of_use": 8
27     },
28     "code_quality": {
29         "accuracy": "Very High",
30         "documentation": "Excellent",
31         "best_for": "General programming, explanations",
32         "code_style": "Clear and well-documented"
33     }
34 },
35 "Cursor AI": {
36     "usability": {
37         "integration": "Excellent - custom IDE experience",
38         "learning_curve": "Low - familiar VS Code base",
39         "response_time": "Very Fast",
40         "ease_of_use": 9
41     },
42     "code_quality": {
43         "accuracy": "High",
44         "documentation": "Good",
45         "best_for": "Real-time coding, refactoring",
46         "code_style": "Context-aware suggestions"
47     }
48 }
49 }
50
51 def print_comparison():
52     for tool, details in comparison_data.items():
53         print(f"\n{tool.upper()}")
54         print("=" * 40)
55         for category, info in details.items():
56             print(f"\n{category.capitalize()}:")
57             for key, value in info.items():
58                 print(f"    {key}: {value}")
59
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python + v
bandarignaneshwar@BANDARIs-MacBook-Air ai coding % /opt/homebrew/bin/python3 "/Users/bandarignaneshwar/Desktop/ai coding/2.5.py/task4.
response_time: Fast
ease_of_use: 9

Code_quality:
accuracy: High
documentation: Good
best_for: Python, JavaScript, TypeScript
code_style: Follows conventions well

GOOGLE GEMINI
=====

Usability:
integration: Good - web-based interface
learning_curve: Low - conversational
response_time: Moderate
ease_of_use: 8

Code_quality:
accuracy: Very High
documentation: Excellent
best_for: General programming, explanations
code_style: Clear and well-documented

CURSOR AI
=====

Usability:
integration: Excellent - custom IDE experience
learning_curve: Low - familiar VS Code base
response_time: Very Fast
ease_of_use: 9

Code_quality:
accuracy: High
documentation: Good
best_for: Real-time coding, refactoring
code_style: Context-aware suggestions
bandarignaneshwar@BANDARIs-MacBook-Air ai coding %
```

## 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.**