Assignment-2.5

Name: Kedala Shivacharan

Ht.No:2303A52160

Batch-34

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.

Original Code:

Prompt: Program to calculate sum of odd and even numbers in a list

```python
def sum_odd_even(numbers):
    sum_odd = 0
    sum_even = 0

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

    return sum_odd, sum_even
if __name__ == "__main__":
    user_input = input("Enter numbers separated by spaces: ")
    try:
        numbers = list(map(int, user_input.split()))
        sum_odd, sum_even = sum_odd_even(numbers)
        print(f"\nList of numbers: {numbers}")
        print(f"Sum of odd numbers: {sum_odd}")
        print(f"Sum of even numbers: {sum_even}")

    except ValueError:
        print("Error: Please enter valid integers separated by spaces.")
```

Optimized Code:

```python
def sum_odd_even(numbers):
    sum_even = sum(n for n in numbers if n % 2 == 0)
    sum_odd = sum(n for n in numbers if n % 2 != 0)
    return sum_odd, sum_even
if __name__ == "__main__":
    try:
        numbers = list(map(int, input("Enter numbers separated by spaces: ").split()))
        sum_odd, sum_even = sum_odd_even(numbers)

        print(f"List: {numbers}")
        print(f"Sum of odd numbers: {sum_odd}")
        print(f"Sum of even numbers: {sum_even}")
    except ValueError:
        print("Error: Please enter valid integers separated by spaces.")
```

Output:

```
Enter numbers separated by spaces: 1 2 3 4 5 6 7 8

List of numbers: [1, 2, 3, 4, 5, 6, 7, 8]
Sum of odd numbers: 16
Sum of even numbers: 20
PS C:\Users\Shailaja\OneDrive\Desktop\STUFF\SEM VI\AAC>
```

Justification:

The refactored code is better because it produces the same output with fewer lines and cleaner logic. It uses Python's built-in sum() function with generator expressions, which improves readability and reduces manual variable handling. Both versions have the same time complexity, but the refactored version is easier to understand and maintain.

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.

Prompt:

Refactor a Python function that calculates the area of different shapes (circle, rectangle, triangle).
Optimize the code for readability and maintainability by avoiding repeated conditional logic.
Use clean structure, meaningful variable names, and Python best practices.
Handle invalid shape input gracefully.
Return only the optimized code and include brief inline comments explaining the logic.

Code:

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

[9]

```python
import math
SHAPES = {
    "circle": (1, lambda r: math.pi * r * r),
    "rectangle": (2, lambda l, w: l * w),
    "triangle": (2, lambda b, h: 0.5 * b * h),
    "square": (1, lambda s: s * s),
}
def area(shape, *values):
    shape = shape.lower()
    if shape not in SHAPES:
        return "Unsupported shape"
    expected_params, formula = SHAPES[shape]
    if len(values) != expected_params:
        return f"{shape.capitalize()} requires {expected_params} value(s)"
    return formula(*values)

if __name__ == "__main__":
    shape = input("Enter shape: ")

    try:
        values = list(map(float, input("Enter dimensions (space-separated): ").split()))
        result = area(shape, *values)
        print("Area:", result)
    except ValueError:
        print("Invalid numeric input")
```

Output:

```
Enter shape: circle
Enter dimensions (space-separated): 4
Area: 50.26548245743669
```

Explanation:

The math module is imported to access mathematical constants such as math.pi, which is required for calculating the area of a circle.

Helper functions like _calculate_circle_area, _calculate_rectangle_area, and _calculate_triangle_area are defined to compute the area of specific shapes. Each helper function accepts **kwargs to receive shape-specific dimensions, retrieves the required values using .get(), and validates that the inputs are positive numeric values. If validation fails, a ValueError is raised. If validation passes, the appropriate area formula is applied and the result is returned.

A dictionary named _shape_calculators maps shape names to their corresponding helper functions. This enables dynamic dispatch and avoids long if/elif conditional chains, making the code cleaner and easier to extend.

The main area function takes a shape name and keyword arguments for dimensions. The shape name is normalized to lowercase to ensure case-insensitive input handling. The function retrieves the appropriate calculator function from _shape_calculators. If the shape is unsupported, a descriptive ValueError is raised listing valid shapes. Otherwise, the selected helper function is called with the provided arguments, and the computed area is returned.

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.

  Prompt: Write a Python program to calculate the sum of odd and even numbers in a list.

  Code:

```python
def calculate_odd_even_sums(numbers):
    sum_odd = 0
    sum_even = 0

    for num in numbers:
        if num % 2 == 0:  # Even number
            sum_even += num
        else:  # Odd number
            sum_odd += num

    return sum_odd, sum_even


def main():
    # Example 1: Using a predefined list
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    print("List of numbers:", numbers)

    sum_odd, sum_even = calculate_odd_even_sums(numbers)

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

    # Example 2: Taking user input
    print("\n" + "="*50)
    print("Enter numbers (separated by spaces):")
    user_input = input().strip()
```

```python
    if user_input:
        try:
            user_numbers = [int(x) for x in user_input.split()]
            sum_odd_user, sum_even_user = calculate_odd_even_sums(user_n

            print(f"\nList of numbers: {user_numbers}")
            print(f"Sum of odd numbers: {sum_odd_user}")
            print(f"Sum of even numbers: {sum_even_user}")
        except ValueError:
            print("Error: Please enter valid integers separated by space
    else:
        print("No input provided.")


if __name__ == "__main__":
    main()
```

Prompt2: Write an optimized Python function to calculate the sum of odd and even numbers using clean and readable code.

from typing import List, Tuple Untitled-1 ●

```python
def calculate_odd_even_sums(numbers: List[int]) -> Tuple[int, int]:
    sum_odd = sum_even = 0
    for number in numbers:
        if number % 2 == 0:
            sum_even += number
        else:
            sum_odd += number
    return sum_odd, sum_even
def calculate_odd_even_sums_functional(numbers: List[int]) -> Tuple[int, int]:
    sum_odd = sum(num for num in numbers if num % 2 != 0)
    sum_even = sum(num for num in numbers if num % 2 == 0)
    return sum_odd, sum_even
def calculate_odd_even_sums_bitwise(numbers: List[int]) -> Tuple[int, int]:
    sum_odd = sum_even = 0
    for number in numbers:
        if number & 1:
            sum_odd += number
        else:
            sum_even += number
    return sum_odd, sum_even
if __name__ == "__main__":
    test_lists = [
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        [0, 1, 2, 3],
        [5, 7, 9],
        [2, 4, 6, 8],
        [],
        [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5],
    ]
    print("Testing calculate_odd_even_sums:\n")
    for numbers in test_lists:
        sum_odd, sum_even = calculate_odd_even_sums(numbers)
        print(f"Numbers: {numbers}")
        print(f"  Sum of odd:  {sum_odd}")
        print(f"  Sum of even: {sum_even}")
        print()
```

Prompt3: Refactor the Python code to calculate odd and even sums using Python best practices and minimal lines of code.

Code:

```python
from typing import List, Tuple


def calculate_odd_even_sums(numbers: List[int]) -> Tuple[int, int]:
    sum_odd = sum(num for num in numbers if num % 2)
    sum_even = sum(num for num in numbers if not num % 2)
    return sum_odd, sum_even


if __name__ == "__main__":
    test_lists = [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [0, 1, 2, 3], [5, 7, 9], [2, 4,

    for numbers in test_lists:
        sum_odd, sum_even = calculate_odd_even_sums(numbers)
        print(f"Numbers: {numbers} -> Odd: {sum_odd}, Even: {sum_even}")
```

Code 1 uses a procedural approach with explicit loops, conditionals, and user input handling. It is easy to understand and suitable for beginners but is longer and less optimized.

Code 2 provides multiple implementations (iterative, functional, and bitwise) using type hints. It focuses on optimization and comparison of different techniques rather than simplicity.

Code 3 uses a single, Pythonic function with generator expressions. It is concise, readable, and follows best practices with minimal code.

Key Differences

- Code 1 prioritizes clarity and learning.
- Code 2 emphasizes optimization and multiple approaches.
- Code 3 focuses on minimal, clean, and maintainable code.

Best Code

Code 3 is the best choice because it achieves the same functionality with the least code, high readability, and better maintainability.

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.

Prompt: Gemini

Explain and refactor a Python program that calculates the sum of odd and even numbers in a list. Provide clean, readable code and explain how the logic works step by step in simple terms. Focus on clarity and beginner-friendly explanation.

Commands + Code + Text ▷ Run all

[10] ✓ 0s

```python
def sum_odd_even(numbers):
  odd_sum = 0
  even_sum = 0

  for num in numbers:
    if not isinstance(num, int):
      print(f"Warning: Skipping non-integer value '{num}'.")
      continue
    if num % 2 == 0:
      even_sum += num
    else:
      odd_sum += num
  return odd_sum, even_sum

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11.5, 'a']
odd_total, even_total = sum_odd_even(my_list)

print(f"Original list: {my_list}")
print(f"Sum of odd numbers: {odd_total}")
print(f"Sum of even numbers: {even_total}")

another_list = [20, 35, 40, 55, 60]
odd_total_2, even_total_2 = sum_odd_even(another_list)

print(f"\nOriginal list: {another_list}")
print(f"Sum of odd numbers: {odd_total_2}")
print(f"Sum of even numbers: {even_total_2}")
```

Prompt: Copilot

 # Write an optimized Python function to calculate the sum of odd and even numbers in a list

 # Use clean logic and minimal lines of code

```python
#
def sum_odd_even(numbers):
    return (
        sum(n for n in numbers if n % 2),
        sum(n for n in numbers if n % 2 == 0)
    )
if __name__ == "__main__":
    try:
        numbers = list(map(int, input("Enter numbers separated by spaces:
        sum_odd, sum_even = sum_odd_even(numbers)

        print(f"List: {numbers}")
        print(f"Sum of odd numbers: {sum_odd}")
        print(f"Sum of even numbers: {sum_even}")
    except ValueError:
        print("Error: Please enter valid integers separated by spaces.")
```

Prompt: Cursor AI

Refactor the given Python code that calculates odd and even sums.

Optimize for readability, minimal lines, and Python best practices.

Remove unnecessary logic and return only the improved code

```python
from typing import List, Tuple


def calculate_odd_even_sums(numbers: List[int]) -> Tuple[int, int]:
    return sum(n for n in numbers if n % 2), sum(n for n in numbers if not n % 2)


if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    sum_odd, sum_even = calculate_odd_even_sums(numbers)
    print(f"Sum of odd: {sum_odd}, Sum of even: {sum_even}")
```

Reflection:

The **first version** is defensive and verbose. It validates input types, skips invalid values, and prints warnings. This improves robustness but increases code length and reduces clarity. It is suitable for learning and real-world data handling, not for optimization.

The **second version** is concise and optimized. It uses generator expressions and user input handling with minimal logic. The focus is on clean computation rather than validation. This is a typical result when the prompt emphasizes optimization and minimal code.

The **third version** is the most Pythonic and minimal. It uses type hints and pure functional logic with no extra control flow. It is easy to read, maintain, and ideal for demonstrations or production code where inputs are trusted.

Conclusion:

The third version is the best in terms of readability, maintainability, and code quality. The first prioritizes safety, while the second balances usability and optimization. This clearly shows how different prompts lead to different AI-generated solutions.