2303A51415
Batch - 03

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.

CODE:

```
def sum_odd_even(numbers):
    """
    Calculate the sum of odd and even numbers in a list.

    Args:
        numbers: List of integers

    Returns:
        tuple: (sum_odd, sum_even)
    """
    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


# Main program
if __name__ == "__main__":
    # Example list
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

    odd_sum, even_sum = sum_odd_even(numbers)

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

OUTPUT:

List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Sum of odd numbers: 25
Sum of even numbers: 30
Total sum: 55

```python
from typing import List, Tuple


def sum_odd_even(numbers: List[int]) -> Tuple[int, int]:
    """
    Calculate the sum of odd and even numbers in a list using list comprehensions.

    Args:
        numbers: List of integers

    Returns:
        tuple: (sum_odd, sum_even)
    """
    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 sum_odd_even_alternative(numbers: List[int]) -> Tuple[int, int]:
    """
    Alternative refactored version using a single pass through the list.
    More memory efficient for large lists.

    Args:
        numbers: List of integers

    Returns:
        tuple: (sum_odd, sum_even)
    """
    return sum(num for num in numbers if num % 2 != 0), \
           sum(num for num in numbers if num % 2 == 0)


# Main program
if __name__ == "__main__":
    # Example list
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

    odd_sum, even_sum = sum_odd_even(numbers)

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

```
    print(f"Total sum: {odd_sum + even_sum}")
```

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.

CODE:

```
import math

def calculate_area(shape_type, **kwargs):
    """
    Calculates the area of different shapes.

    Args:
        shape_type (str): The type of the shape ('circle', 'square',
'triangle').
        **kwargs: Keyword arguments for the shape's dimensions.
                  For 'circle': 'radius'
                  For 'square': 'side'
                  For 'triangle': 'base', 'height'

    Returns:
        float: The calculated area.
        str: An error message if the shape_type is invalid or
dimensions are missing.
    """
    if shape_type == 'circle':
        if 'radius' in kwargs:
            return math.pi * (kwargs['radius'] ** 2)
        else:
            return "Error: For a circle, 'radius' is required."
    elif shape_type == 'square':
        if 'side' in kwargs:
            return kwargs['side'] ** 2
        else:
            return "Error: For a square, 'side' is required."
    elif shape_type == 'triangle':
        if 'base' in kwargs and 'height' in kwargs:
```

```
            return 0.5 * kwargs['base'] * kwargs['height']
        else:
            return "Error: For a triangle, 'base' and 'height' are
required."
    else:
        return "Error: Invalid shape_type. Choose from 'circle',
'square', 'triangle'."

# Example Usage:
print(f"Area of circle with radius 5: {calculate_area('circle',
radius=5)}")
print(f"Area of square with side 4: {calculate_area('square',
side=4)}")
print(f"Area of triangle with base 6 and height 3:
{calculate_area('triangle', base=6, height=3)}")
print(f"Invalid shape type: {calculate_area('rectangle', length=5,
width=10)}")
print(f"Missing dimension for circle: {calculate_area('circle')}")
```

OUTPUT:

```
Area of circle with radius 5: 78.53981633974483
Area of square with side 4: 16
Area of triangle with base 6 and height 3: 9.0
Invalid shape type: Error: Invalid shape_type. Choose from 'circle',
'square', 'triangle'.
Missing dimension for circle: Error: For a circle, 'radius' is
required.
```

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

Prompt : give me a function to check if the given string is palindrome or not

def is_palindrome(text):

"""

Check if a given string is a palindrome.

Args:

    text: String to check (case-insensitive, ignores spaces and punctuation)

Returns:

    bool: True if the string is a palindrome, False otherwise

Examples:

    >>> is_palindrome("racecar")

    True

    >>> is_palindrome("hello")

    False

    >>> is_palindrome("A man a plan a canal Panama")

    True

"""

Convert to lowercase and remove spaces for comparison

cleaned_text = ''.join(text.lower().split())

Check if the cleaned string equals its reverse

return cleaned_text == cleaned_text[::-1]

Prompt : Give me a function to check if the given string is palindrome or not use strict checks.

```
def is_palindrome_strict(text):
    """
    Strict palindrome checker - only checks if the exact string is a palindrome.
    Case-sensitive and includes spaces.

    Args:
        text: String to check (case-sensitive, includes spaces)

    Returns:
        bool: True if the string is a palindrome, False otherwise

    Examples:
        >>> is_palindrome_strict("racecar")
        True
        >>> is_palindrome_strict("Racecar")
        False
        >>> is_palindrome_strict("a man a plan")
        False
    """
    return text == text[::-1]
```

Prompt : Give me a function to check if the given string is palindrome or not use strict checks keep it advanced ignores back spaces,punctuation and case and only alphanumeric characters.

Code :

```
def is_palindrome_advanced(text):
    """
    Advanced palindrome checker that ignores spaces, punctuation, and case.
    Only checks alphanumeric characters.

    Args:
        text: String to check

    Returns:
        bool: True if the string is a palindrome, False otherwise

    Examples:
        >>> is_palindrome_advanced("A man, a plan, a canal: Panama")
        True
        >>> is_palindrome_advanced("race a car")
        False
```

```
    """
    # Remove all non-alphanumeric characters and convert to lowercase
    cleaned = ''.join(char.lower() for char in text if char.isalnum())

    return cleaned == cleaned[::-1]
```

Prompt: Give me a function to check if the given palindrome or not using two pointer technique(memory efficient) Ignore cases and spaces.

Code :

```
def is_palindrome_two_pointer(text):
    """
    Palindrome checker using two-pointer technique (more memory efficient).
    Ignores case and spaces.

    Args:
        text: String to check

    Returns:
        bool: True if the string is a palindrome, False otherwise
    """
    # Clean the text: lowercase and remove spaces
    cleaned = ''.join(text.lower().split())
```

```
    # Two-pointer approach
    left = 0
    right = len(cleaned) - 1

    while left < right:
        if cleaned[left] != cleaned[right]:
            return False
        left += 1
        right -= 1

    return True
```

Outputs :

1. Basic Palindrome Checker (ignores case and spaces):

'racecar' -> True

'hello' -> False

'level' -> True

'madam' -> True

'A man a plan a canal Panama' -> True

'race a car' -> False

'A man, a plan, a canal: Panama' -> False

'12321' -> True

'12345' -> False

'' -> True

'a' -> True

'Aa' -> True

2. Strict Palindrome Checker (case-sensitive, includes spaces):

'racecar' -> True

'hello' -> False

'level' -> True

'madam' -> True

'A man a plan a canal Panama' -> False

'race a car' -> False

3. Advanced Palindrome Checker (ignores punctuation):

'racecar' -> True

'hello' -> False

'level' -> True

'madam' -> True

'A man a plan a canal Panama' -> True

'race a car' -> False

'A man, a plan, a canal: Panama' -> True

'12321' -> True

'12345' -> False

'' -> True

'a' -> True

'Aa' -> True

4. Two-Pointer Palindrome Checker (memory efficient):

'racecar' -> True

'hello' -> False

'level' -> True

'madam' -> True

'A man a plan a canal Panama' -> True

'race a car' -> False

'A man, a plan, a canal: Panama' -> False

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

Written reflection :

I prefer using Copilot while working in VS Code and Gemini when collaborating with others, as both tools are extremely helpful during development. The auto-complete functionality in these tools significantly speeds up the coding process and reduces repetitive effort. The models they currently use are highly efficient and capable of generating clean, structured code that aligns well with our requirements. Another advantage is that they offer multiple approaches to solving the same problem, which helps in choosing the most optimal or readable solution. These tools also adapt well to different coding styles and project needs. Overall, I would strongly recommend these two AI tools for coding, as they are among the best options available today, especially for vibe coding where productivity and flow matter the most.