----------AI ASSISSTANT CODING ----------

Lab 2.5

BILLA SPURTHI

2303A51802

Batch 29

TASK 1

Ask Gemini Write a program to calculate the sum of odd and even numbers in a list,then refactor it using AI.

```
def calculate_odd_even_sums(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:
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_sum, even_sum = calculate_odd_even_sums(my_list)
print(f"Original List: {my_list}")
print(f"Sum of odd numbers: {odd_sum}")
print(f"Sum of even numbers: {even_sum}")
```

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

Explanation of the initial program:

This program defines a function `calculate_odd_even_sums` that takes a list of numbers as input. It initializes two variables, `odd_sum` and `even_sum`, to zero. Then, it iterates through each number in the list. If a number is even (i.e., its remainder when divided by 2 is 0), it's added to `even_sum`; otherwise, it's added to `odd_sum`. Finally, it returns both sums.

Refactoring the code using AI:

Now, let's refactor the code to make it more concise and potentially more Pythonic. We can use list comprehensions or the `sum()` function with `filter()` to achieve this in a more compact way.

```
def calculate_odd_even_sums_refactored(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 with refactored code:
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_sum_refactored, even_sum_refactored = calculate_odd_even_sums_refactored(my_list)
print(f"Refactored List: {my_list}")
print(f"Refactored Sum of odd numbers: {odd_sum_refactored}")
print(f"Refactored Sum of even numbers: {even_sum_refactored}")
```

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

Explanation of the refactored program:

In the refactored version, we use generator expressions within the `sum()` function.

- `sum(num for num in numbers if num % 2 != 0)` calculates the sum of all odd numbers.
- `sum(num for num in numbers if num % 2 == 0)` calculates the sum of all even numbers.

This approach is more compact and often considered more 'Pythonic' as it leverages built-in functions and generator expressions for cleaner code.

TASK 2

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

```
Start coding or generate with AI.
```

```
import math

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

    Args:
        shape (str): The type of shape (e.g., 'circle', 'square', 'rectangle', 'triangle').
        **kwargs: Keyword arguments specific to the shape:
            - circle: radius (float)
            - square: side (float)
            - rectangle: length (float), width (float)
            - triangle: base (float), height (float)

    Returns:
        float: The calculated area of the shape.
        str: An error message if the shape or arguments are invalid.
    """
    if shape == 'circle':
        radius = kwargs.get('radius')
        if radius is not None and radius >= 0:
            return math.pi * radius**2
        else:
            return "Error: Circle requires a non-negative 'radius'."
    elif shape == 'square':
        side = kwargs.get('side')
        if side is not None and side >= 0:
            return side**2
        else:
            return "Error: Square requires a non-negative 'side'."
    elif shape == 'rectangle':
        length = kwargs.get('length')
        width = kwargs.get('width')
        if length is not None and width is not None and length >= 0 and width >= 0:
            return length * width
        else:
            return "Error: Rectangle requires non-negative 'length' and 'width'."
    elif shape == 'triangle':
        base = kwargs.get('base')
        height = kwargs.get('height')
        if base is not None and height is not None and base >= 0 and height >= 0:
            return 0.5 * base * height
        else:
            return "Error: Triangle requires non-negative 'base' and 'height'."
    else:
        return "Error: Invalid shape specified."

# Example usage:
print(f"Area of circle (radius=5): {calculate_area('circle', radius=5)}")
print(f"Area of square (side=4): {calculate_area('square', side=4)}")
print(f"Area of rectangle (length=6, width=7): {calculate_area('rectangle', length=6, width=7)}")
print(f"Area of triangle (base=10, height=5): {calculate_area('triangle', base=10, height=5)}")
print(f"Area with invalid shape: {calculate_area('oval', major_axis=5, minor_axis=3)}")
print(f"Area of circle with missing radius: {calculate_area('circle')}")
```

```
Area of circle (radius=5): 78.53981633974483
Area of square (side=4): 16
Area of rectangle (length=6, width=7): 42
Area of triangle (base=10, height=5): 25.0
Area with invalid shape: Error: Invalid shape specified.
Area of circle with missing radius: Error: Circle requires a non-negative 'radius'.
```

explaination of `calculate_area` function:

Certainly! To recap, the calculate_area function serves to calculate the area for various geometric shapes. It uses conditional logic to handle different shape inputs such as 'circle', 'square', 'rectangle', and 'triangle'. Each shape requires specific keyword arguments (e.g., radius for a circle, side for a square, length and width for a rectangle, and base and height for a triangle).

The function also incorporates error handling to ensure that all necessary parameters are provided and are valid (e.g., non-negative values). If a shape is not recognized or if required parameters are missing or invalid, it returns a descriptive error message instead of calculating an area.

```
##  Task 4: Tool Comparison Reflection (Short Reflection)

Based on my work in this topic, I compared Gemini, GitHub Copilot, and Cursor AI in terms of usability and code quality.

Gemini is easy to use and gives good explanations, especially for understanding concepts. It is helpful when we want steps,

GitHub Copilot works best inside coding editors and is very fast in giving code suggestions while typing. It produces good q

Cursor AI is the most useful for this topic because it supports prompt-based coding, refactoring, and generating multiple va

Overall Recommendation:
For writing and improving programs quickly with refactoring, Cursor AI is the best choice, while Copilot is best for fast co
```