

# AI ASSISTANT CODING

## LAB-02

Name : R.Deekshith Goud

Roll No : 2303A52104

Batch No : 33

## Lab 2: Exploring Additional AI Coding Tools beyond Copilot – Google Gemini (Colab) and Cursor AI

### Task – 1 : Refactoring Odd/Even Logic (List Version)

Scenario: You are improving legacy code.

The given legacy program calculates the sum of odd and even numbers in a list. The code needs to be improved for better readability and efficiency using AI tools.

#### Prompt Used :

Write a program to calculate the sum of odd and even numbers in a list, Refactor this Python code to improve readability and efficiency.

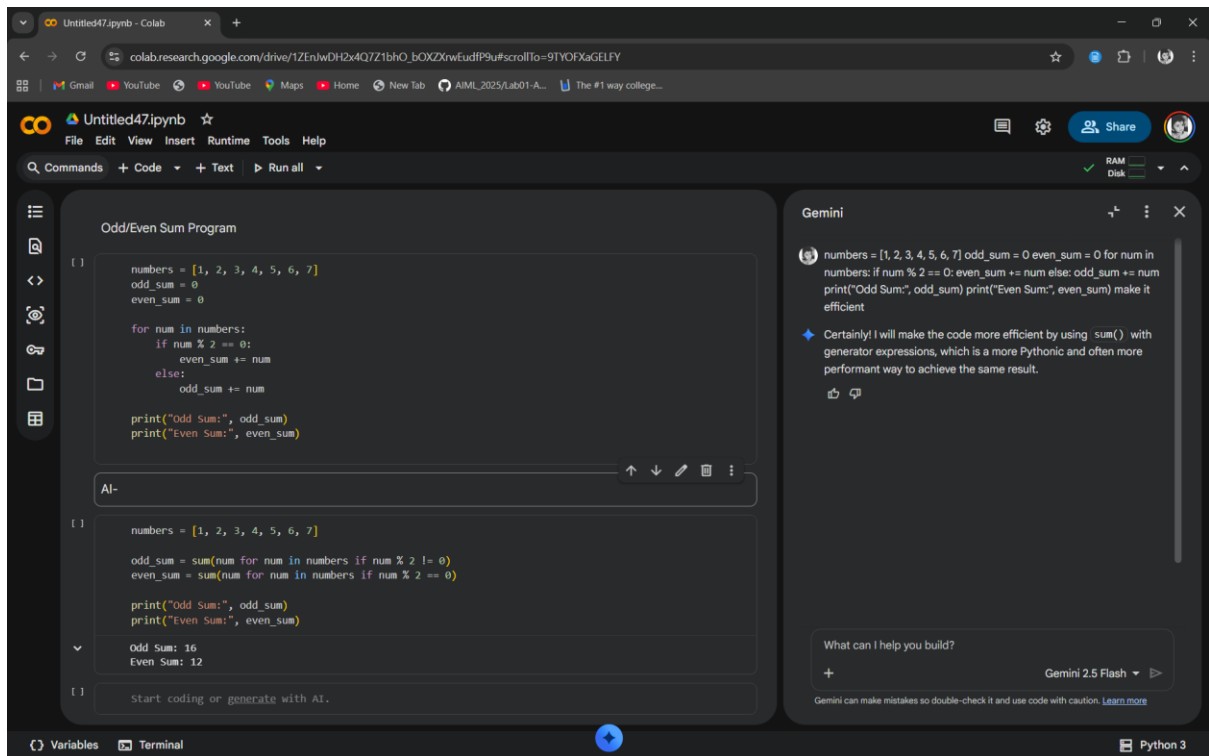
#### Original Code :

```
def sum_odd_even(numbers: list[int]) -> tuple[int, int]:  
    even_sum = 0  
  
    odd_sum = 0  
  
    for num in numbers:  
        if num % 2 == 0:  
            even_sum += num  
        else:  
            odd_sum += num  
  
    return even_sum, odd_sum  
  
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
even_sum, odd_sum = sum_odd_even(my_list)  
  
print(f"List: {my_list}")
```

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

## AI-Refactored Code :

```
def sum_odd_even(numbers: list[int]) -> tuple[int, int]:  
    """  
  
    Calculates the sum of even and odd numbers in a list.  
  
    Args:  
        numbers: A list of integers.  
  
    Returns:  
        A tuple containing (even_sum, odd_sum).  
    """  
  
    even_sum = sum(num for num in numbers if num % 2 == 0)  
    odd_sum = sum(num for num in numbers if num % 2 != 0)  
    return even_sum, odd_sum  
  
# Example usage:  
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
even_sum, odd_sum = sum_odd_even(my_list)  
  
print(f"List: {my_list}")  
print(f"Sum of even numbers: {even_sum}")  
print(f"Sum of odd numbers: {odd_sum}")
```



## Explanation :

The refactored code reduces the number of variables and uses Python's built-in sum() function. It improves readability, reduces code length, and enhances maintainability. The logic is clearer and easier to modify.

## Task 2: Area Calculation Explanation (Gemini)

### Scenario

This task helps onboard a junior developer by explaining a function that calculates areas of different shapes.

## Prompt Used :

Write a Python function to calculate area of circle, rectangle, and triangle and explain the code

```
import math
```

```
def calculate_area(shape: str, **kwargs) -> float:
```

```
    """
```

```
    Calculates the area of different geometric shapes.
```

```
    Args:
```

```
        shape (str): The type of shape ('circle', 'rectangle', 'triangle').
```

```
        **kwargs: Keyword arguments for the dimensions of the shape:
```

```
            - For 'circle': 'radius' (float)
```

```
            - For 'rectangle': 'length' (float), 'width' (float)
```

```
            - For 'triangle': 'base' (float), 'height' (float)
```

```
    Returns:
```

```
        float: The calculated area of the shape.
```

```
    Raises:
```

```
        ValueError: If an unsupported shape is provided or required dimensions are missing.
```

```
    """
```

```
    if shape.lower() == 'circle':
```

```
        if 'radius' in kwargs:
```

```
            radius = kwargs['radius']
```

```
            return math.pi * radius**2
```

```
        else:
```

```
            raise ValueError("For 'circle', 'radius' is required.")
```

```
    elif shape.lower() == 'rectangle':
```

```
        if 'length' in kwargs and 'width' in kwargs:
```

```
            length = kwargs['length']
```

```
            width = kwargs['width']
```

```
            return length * width
```

```
        else:
```

```
            raise ValueError("For 'rectangle', 'length' and 'width' are required.")
```

```
    elif shape.lower() == 'triangle':
```

if 'base' in kwargs and 'height' in kwargs:

```
    base = kwargs['base']
```

```
    height = kwargs['height']
```

```
    return 0.5 * base * height
```

else:

```
    raise ValueError("For 'triangle', 'base' and 'height' are required.")
```

else:

```
    raise ValueError("Unsupported shape. Choose from 'circle', 'rectangle', or 'triangle'.")
```

# Example usage:

```
print(f"Area of circle (radius=5): {calculate_area('circle', radius=5):.2f}")
```

```
print(f"Area of rectangle (length=4, width=6): {calculate_area('rectangle', length=4, width=6):.2f}")
```

```
print(f"Area of triangle (base=7, height=3): {calculate_area('triangle', base=7, height=3):.2f}")
```

# Example of handling an error (uncomment to test):

# try:

```
# print(calculate_area('square', side=5))
```

# except ValueError as e:

```
# print(f"Error: {e}")
```

The screenshot displays a Google Colab notebook environment. The main code cell contains a Python function `calculate_area` that takes a shape name and keyword arguments. It uses `if/elif/else` statements to handle different shapes: circle (radius), rectangle (length, width), and triangle (base, height). It includes error handling with `ValueError` for unsupported shapes or missing dimensions. Example usage and error handling code are also present.

On the right, the Gemini AI chat interface is open. It shows a plan for refactoring the `calculate_area` function to use a dictionary-based dispatch approach for better readability and extensibility. The plan includes steps for refactoring, implementing shape handlers, updating error handling, testing, and a final task to confirm the refactoring.

**Gemini**

make in more efficient

Great! I've created a plan to make the `calculate_area` function more efficient by using a dictionary-based approach instead of multiple `if/elif` statements. This will improve readability and extensibility. Take a look and let me know if you'd like any adjustments.

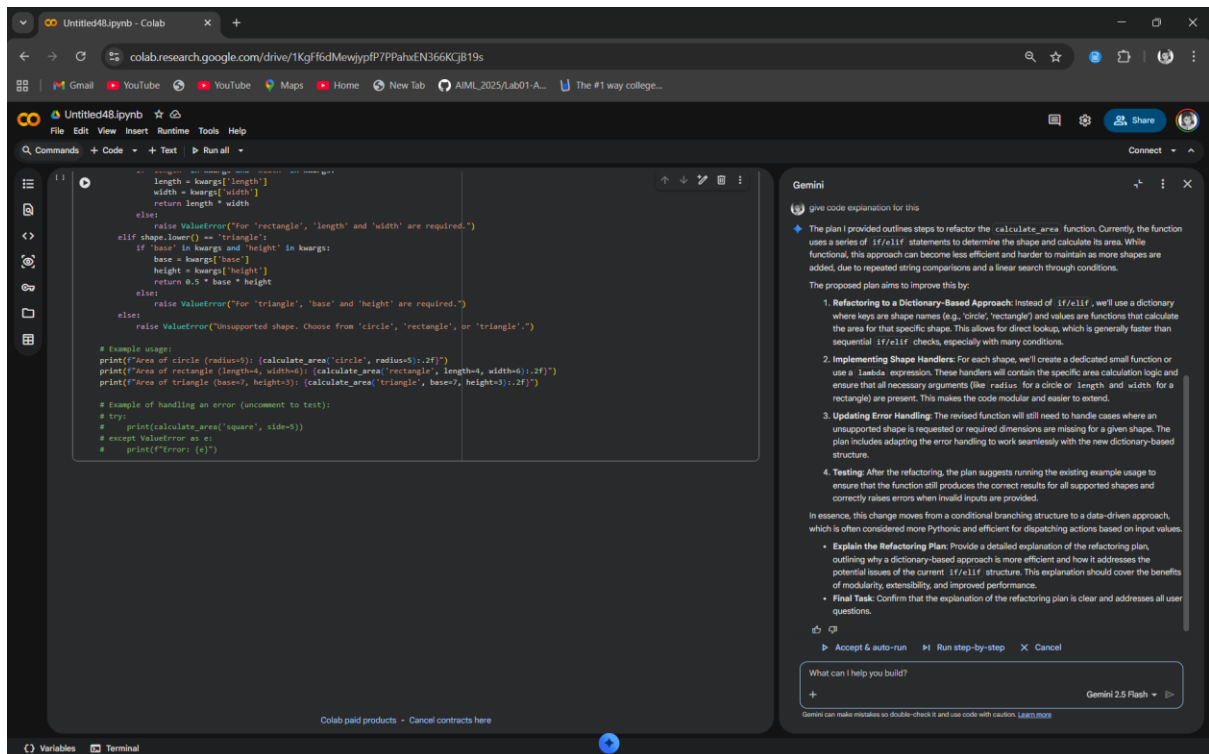
- **Refactor `calculate_area`:** Refactor the `calculate_area` function to use a dictionary-based dispatch for shape calculation. This will replace the `if/elif` structure, making the function more efficient and extensible by avoiding multiple string comparisons and allowing for easier addition of new shapes.
- **Implement Shape Handlers:** Define individual helper functions or lambda expressions for each shape ('circle', 'rectangle', 'triangle') that encapsulate their specific area calculation logic and argument validation. These will be stored in the dispatch dictionary.
- **Update Error Handling:** Modify the error handling to integrate with the dictionary-based approach, ensuring that unsupported shapes and missing required arguments are still caught and raise appropriate `ValueError` exceptions.
- **Test Refactored Function:** Run the example usage provided in the original code to verify that the refactored `calculate_area` function produces the same correct outputs for 'circle', 'rectangle', and 'triangle' and handles errors as expected.
- **Final Task:** Confirm the `calculate_area` function has been made more efficient and robust, and provide the updated code.

Show me the refactored `calculate_area` function    Implement helper functions for each shape calculation    Test the refactored `calculate_area` with provided examples

What can I help you build?

+    Gemini 2.5 Flash    >

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)



## Explanation :

The function accepts the shape name and required dimensions as input. Conditional statements decide which formula to apply. For a circle, radius is used, while rectangle and triangle require two values. The function returns the calculated area based on the selected shape.

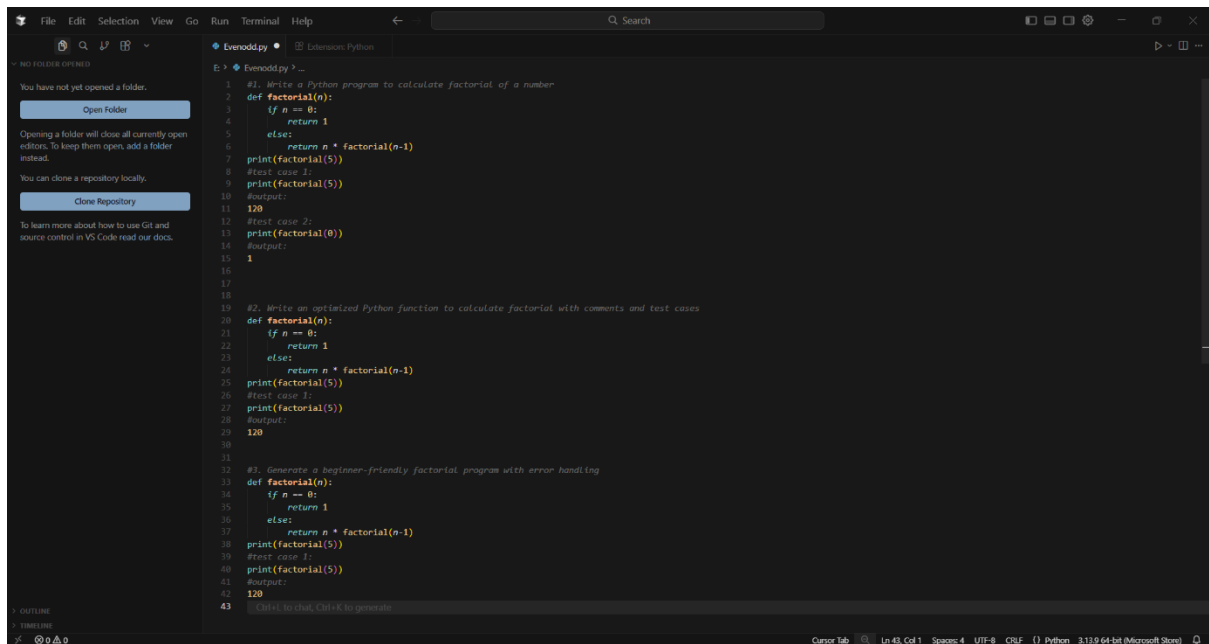
## Task 3: Prompt Sensitivity Experiment (Cursor AI)

Use Cursor AI with different prompts for the same problem and observe code changes.

The factorial problem was used to test how different prompts affect AI-generated code.

## Prompts Used :

1. Write a Python program to calculate factorial
2. Write an optimized Python function to calculate factorial with comments
3. Generate a beginner-friendly factorial program with error handling



## Observation

The AI produced different code styles depending on the prompt. Optimized prompts improved performance and readability, while beginner-friendly prompts added validation and safety checks.

## Task 4: Tool Comparison Reflection

Comparison of Gemini, Copilot, and Cursor AI

### Reflection :

Google Gemini is best suited for explanations and learning support. GitHub Copilot provides real-time inline suggestions, improving developer productivity. Cursor AI excels in experimentation, refactoring, and prompt-based exploration. Each tool serves a different purpose, and choosing the right one depends on whether the goal is learning, development speed, or code analysis.