

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}")
```

The screenshot shows a Google Colab notebook titled "Untitled47.ipynb". The left panel displays two versions of a Python script. The first version is the original code:

```

numbers = [1, 2, 3, 4, 5, 6, 7]
odd_sum = 0
even_sum = 0

for num in numbers:
    if num % 2 == 0:
        even_sum += num
    else:
        odd_sum += num

print("Odd Sum:", odd_sum)
print("Even Sum:", even_sum)

```

The second version is the AI-refactored code:

```

numbers = [1, 2, 3, 4, 5, 6, 7]

odd_sum = sum(num for num in numbers if num % 2 != 0)
even_sum = sum(num for num in numbers if num % 2 == 0)

print("Odd Sum:", odd_sum)
print("Even Sum:", even_sum)

```

The right panel shows a "Gemini" AI interface with a sidebar suggesting improvements and a main area where the AI provides feedback and generates code. The feedback includes:

- A note about efficiency: "numbers: if num % 2 == 0: even_sum += num else: odd_sum += num print("Odd Sum:", odd_sum) print("Even Sum:", even_sum) make it efficient"
- A tip: "◆ Certainly! I will make the code more efficient by using sum() with generator expressions, which is a more Pythonic and often more performant way to achieve the same result."

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 shows a Google Colab notebook titled "Untitled48.ipynb - Colab". The code editor contains the refactored `calculate_area` function and example usage. The Gemini sidebar is open, providing suggestions for making the code more efficient and refactoring it to use a dictionary-based dispatch.

```

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}")

```

Gemini

make it 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 helper functions: 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 can make mistakes so double-check it and use code with caution. Learn more

The screenshot shows a Google Colab notebook titled "Untitled48.ipynb". The code in the notebook is as follows:

```

def calculate_area(shape, **kwargs):
    if shape == 'circle':
        length = kwargs.get('radius')
        width = kwargs.get('width')
        return length * width
    else:
        raise ValueError("For 'rectangle', 'length' and 'width' are required.")
    elif shape == '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("Area of circle (radius=5): (calculate_area('circle', radius=5), 25f)")
print("Area of rectangle (length=4, width=6): (calculate_area('rectangle', length=4, width=6), 24f)")
print("Area of triangle (base=7, height=3): (calculate_area('triangle', base=7, height=3), 21f)")

# Example of handling an error (uncomment to test):
# try:
#     print(calculate_area('square', side=5))
# except ValueError as e:
#     print(f"Error: {e}")

```

To the right of the code, the Gemini AI interface is open with the following plan:

Gemini

give code explanation for this

◆ The plan I provided outlines steps to refactor the `calculate_area` function. Currently, the function uses a series of `if/elif` statements to determine the shape and calculate its area. While functional, this approach can become less efficient and harder to maintain as more shapes are added, due to repetitive string comparisons and a linear search through conditions.

The proposed plan aims to improve this:

- Refactoring to a Dictionary-Based Approach:** Instead of `if/elif`, we'll use a dictionary where keys are shape names (e.g., 'circle', 'rectangle') and values are functions that calculate the area for that specific shape. This allows for direct lookup, which is generally faster than sequential `if/elif` checks, especially with many conditions.
- Implementing Shape Handlers:** For each shape, we'll create a dedicated small function or use a `lambda` expression. These handlers will contain the specific area calculation logic and ensure that all necessary arguments (like `radius` for a circle or `length` and `width` for a rectangle) are present. This makes the code modular and easier to extend.
- Updating Error Handling:** The revised function will still need to handle cases where an unsupported shape is requested or required dimensions are missing for a given shape. The plan includes adapting the error handling to work seamlessly with the new dictionary-based structure.
- Testing:** After the refactoring, the plan suggests running the existing example usage to ensure that the function still produces the correct results for all supported shapes and correctly raises errors when invalid inputs are provided.

In essence, this change moves from a conditional branching structure to a data-driven approach, which is often considered more Pythonic and efficient for dispatching actions based on input values.

- Explain the Refactoring Plan: Provide a detailed explanation of the refactoring plan, outlining why a dictionary-based approach is more efficient and how it addresses the potential issues of the current `if/elif` structure. This explanation should cover the benefits of modularity, extensibility, and improved performance.
- Final Task: Confirm that the explanation of the refactoring plan is clear and addresses all user questions.

Accept & auto-run | Run step-by-step | Cancel

What can I help you build?

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

The screenshot shows a Microsoft Visual Studio Code interface. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and a search bar. The left sidebar has sections for NO FOLDER OPENED, Open Folder, Clone Repository, and Learn more about Git and source control. The main editor area contains Python code for calculating factorial, with three different prompts shown in comments. The bottom status bar displays Cursor Tab, Line 43 Col 1, Spaces: 4, UTE-8, CRLF, Python 3.13.0 64-bit (Microsoft Store).

```
#1. Write a Python program to calculate factorial of a number
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
print(factorial(5))
#test case 1:
print(factorial(5))
#output:
120

#2. Write an optimized Python function to calculate factorial with comments and test cases
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
print(factorial(5))
#test case 1:
print(factorial(5))
#output:
120

#3. Generate a beginner-friendly factorial program with error handling
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
print(factorial(5))
#test case 1:
print(factorial(5))
#output:
120
```

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.