

AI ASSISTANT CODING ASSIGNMENT-2.5

Name: A.Sathwik

Enrollment no: 2303A52158

Batch: 34

Semester: VI

Branch: CSE(AIML)

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

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.

❖ Expected Output:

❖ Original and improved code:

Original Code:

```
def calculate_odd_even_sums_original(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

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd, even = calculate_odd_even_sums_original(my_list)
print(f"Original Code:\nOdd sum: {odd}, Even sum: {even}")
```

Refactored Code:

```
def calculate_odd_even_sums_refactored(numbers):
    odd_numbers = [num for num in numbers if num % 2 != 0]
    even_numbers = [num for num in numbers if num % 2 == 0]
    return sum(odd_numbers), sum(even_numbers)

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_refactored, even_refactored = calculate_odd_even_sums_refactored(my_list)
print(f"Refactored Code:\nOdd sum: {odd_refactored}, Even sum: {even_refactored}")
```

The screenshot shows the Gemini AI interface with two tabs open. Tab [1] displays the original Python code:

```
def calculate_odd_even_sums_original(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

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd, even = calculate_odd_even_sums_original(my_list)
print(f"Original code:\nOdd sum: {odd}, Even sum: {even}")
```

Tab [2] displays the refactored Python code:

```
def calculate_odd_even_sums_refactored(numbers):
    odd_numbers = [num for num in numbers if num % 2 != 0]
    even_numbers = [num for num in numbers if num % 2 == 0]
    return sum(odd_numbers), sum(even_numbers)

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_refactored, even_refactored = calculate_odd_even_sums_refactored(my_list)
print(f"Refactored Code:\nOdd sum: {odd_refactored}, Even sum: {even_refactored}")
```

Both tabs include explanatory text and code execution results.

The screenshot shows the Gemini AI interface with two tabs open. Tab [2] displays the refactored Python code:

```
def calculate_odd_even_sums_refactored(numbers):
    odd_numbers = [num for num in numbers if num % 2 != 0]
    even_numbers = [num for num in numbers if num % 2 == 0]
    return sum(odd_numbers), sum(even_numbers)

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_refactored, even_refactored = calculate_odd_even_sums_refactored(my_list)
print(f"Refactored Code:\nOdd sum: {odd_refactored}, Even sum: {even_refactored}")
```

Tab [1] includes a "Refactored Code Explanation" section with explanatory text.

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.

❖ Expected Output:

> Code & Explanation

```
import math

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

    Args:
        shape (str): The type of the shape ('circle', 'rectangle', 'triangle').
        **kwargs: Keyword arguments for shape dimensions.

    Returns:
        float: The calculated area of the shape.
        str: An error message if the shape is not recognized or dimensions are missing.
    """

    if shape == 'circle':
        if 'radius' in kwargs:
            return math.pi * (kwargs['radius'] ** 2)
        else:
            return "Error: Missing 'radius' for circle."
    elif shape == 'rectangle':
        if 'length' in kwargs and 'width' in kwargs:
            return kwargs['length'] * kwargs['width']
        else:
            return "Error: Missing 'length' or 'width' for rectangle."
    elif shape == 'triangle':
        if 'base' in kwargs and 'height' in kwargs:
            return 0.5 * kwargs['base'] * kwargs['height']
        else:
            return "Error: Missing 'base' or 'height' for triangle."
    else:
        return "Error: Unknown shape type."

# Example usage:
print(f"Circle area with radius 5: {calculate_area('circle', radius=5)}")
print(f"Rectangle area with length 4 and width 6: {calculate_area('rectangle', length=4, width=6)}")
print(f"Triangle area with base 10 and height 3: {calculate_area('triangle', base=10, height=3)}")
```

```
print(f"Invalid shape: {calculate_area('square', side=5)}")  
print(f"Missing dimension: {calculate_area('circle')}")
```

The screenshot shows a Jupyter Notebook interface with a dark theme. The code cell contains a Python function named `calculate_area` that calculates the area of different geometric shapes based on keyword arguments. The function handles three types of shapes: circle, rectangle, and triangle. It includes error handling for missing dimensions and unrecognized shapes. The code is annotated with docstrings explaining the arguments and return values.

```
import math  
  
def calculate_area(shape, **kwargs):  
    """  
    Calculates the area of different geometric shapes.  
  
    Args:  
        shape (str): The type of the shape ('circle', 'rectangle', 'triangle').  
        **kwargs: Keyword arguments for shape dimensions.  
  
    Returns:  
        float: The calculated area of the shape.  
        str: An error message if the shape is not recognized or dimensions are missing.  
    """  
  
    if shape == 'circle':  
        if 'radius' in kwargs:  
            return math.pi * (kwargs['radius'] ** 2)  
        else:  
            return "Error: Missing 'radius' for circle."  
    elif shape == 'rectangle':  
        if 'length' in kwargs and 'width' in kwargs:  
            return kwargs['length'] * kwargs['width']  
        else:  
            return "Error: Missing 'length' or 'width' for rectangle."  
    elif shape == 'triangle':  
        if 'base' in kwargs and 'height' in kwargs:  
            return 0.5 * kwargs['base'] * kwargs['height']  
        else:  
            return "Error: Missing 'base' or 'height' for triangle."  
    else:  
        return "Error: Unknown shape type."  
  
    # Example usage:  
    print(f"Circle area with radius 5: {calculate_area('circle', radius=5)}")  
    print(f"Rectangle area with length 4 and width 6: {calculate_area('rectangle', length=4, width=6)}")  
    print(f"Triangle area with base 10 and height 3: {calculate_area('triangle', base=10, height=3)}")  
    print(f"Invalid shape: {calculate_area('square', side=5)}")  
    print(f"Missing dimension: {calculate_area('circle')}")
```

The screenshot shows the execution results of the code in the previous screenshot. The output pane displays the results of the `print` statements, including the calculated areas for a circle, rectangle, and triangle, and error messages for invalid shapes and missing dimensions.

```
... Circle area with radius 5: 78.53981633974483  
Rectangle area with length 4 and width 6: 24  
Triangle area with base 10 and height 3: 15.0  
Invalid shape: Error: Unknown shape type.  
Missing dimension: Error: Missing 'radius' for circle.
```

The screenshot shows a Jupyter Notebook interface with the title "Lab Assignment 2.5". The main content area displays the following code and its explanatory text:

```
Explanation of the calculate_area function:

Hello junior developer! Let's break down this calculate_area function, which is designed to compute the area of various geometric shapes. It's a great example of a flexible function using conditional logic.

1. import math: This line imports Python's built-in math module. We need it specifically for math.pi, which gives us the value of Pi, essential for calculating a circle's area.

2. def calculate_area(shape, **kwargs): : This is our function definition. Let's look at its parts:
    ○ shape : This is a regular argument that expects a string (like 'circle', 'rectangle', 'triangle') to tell the function which shape's area we want to calculate.
    ○ **kwargs : This is a very powerful Python feature! It stands for "keyword arguments" and allows the function to accept an arbitrary number of keyword-value pairs (like radius=5, length=4, width=6). These arguments are collected into a dictionary named kwargs inside the function. This makes our function very flexible, as different shapes require different dimensions.

3. Docstring ( """Calculates the area...""") : This block of text explains what the function does, its arguments (Args), and what it returns (Returns). It's crucial for making your code understandable to others (and your future self!).

4. if shape == 'circle': : This is the first conditional check. If the shape argument is exactly 'circle':
    ○ if 'radius' in kwargs: : We then check if a 'radius' was provided in the kwargs dictionary. Circles need a radius!
    ○ return math.pi * (kwargs['radius'] ** 2) : If radius is present, we calculate the area using the formula  $\pi * r^2$  and return the result.
    ○ else: return "Error: Missing 'radius' for circle." : If radius is missing, we return an informative error message.

5. elif shape == 'rectangle': : If it's not a circle, we check if it's a 'rectangle'.
    ○ if 'length' in kwargs and 'width' in kwargs: : Rectangles need both 'length' and 'width'. We check for both.
    ○ return kwargs['length'] * kwargs['width'] : If both are present, we calculate the area  $length * width$ .
    ○ else: return "Error: Missing 'length' or 'width' for rectangle." : Otherwise, an error message is returned.

6. elif shape == 'triangle': : Similar logic for a 'triangle', checking for 'base' and 'height' and using the formula  $0.5 * base * height$ .
7. else: return "Error: Unknown shape type." : Finally, if the shape provided doesn't match any of our defined cases ('circle', 'rectangle', 'triangle'), we return an error indicating an unknown shape.

This function demonstrates how to create reusable code that handles different scenarios and gracefully reports errors when necessary. Good luck with your coding journey!
```

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

❖ 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:

- 1) Write a function to compute the factorial of a non-negative integer using a loop.
- 2) Generate Python code for a recursive function to find the factorial of number.
- 3) Provide a Python one-liner to calculate the factorial, perhaps using a built-in function or a concise expression.

➤ Code variation:

```
def factorial_loop(n):
    if not isinstance(n, int) or n < 0:
        return "Input must be a non-negative integer"
    if n == 0:
        result = 1
    else:
        result = 1
        for i in range(1, n + 1):
            result *= i
    return result

print("Factorial of 5 (loop):", factorial_loop(5))
print("Factorial of 0 (loop):", factorial_loop(0))

Factorial of 5 (loop): 120
Factorial of 0 (loop): 1

def factorial_recursive(n):
    if not isinstance(n, int) or n < 0:
        return "Input must be a non-negative integer"
    if n == 0:
        return 1
    else:
        return n * factorial_recursive(n - 1)

print("Factorial of 5 (recursive):", factorial_recursive(5))
print("Factorial of 0 (recursive):", factorial_recursive(0))

Factorial of 5 (recursive): 120
Factorial of 0 (recursive): 1
```

```
[12] import math

def factorial_oneliner(n):
    if not isinstance(n, int) or n < 0:
        return "Input must be a non-negative integer"
    return math.factorial(n)

print(f"Factorial of 5 (one-liner): {factorial_oneliner(5)}")
print(f"Factorial of 0 (one-liner): {factorial_oneliner(0)}")

... Factorial of 5 (one-liner): 120
Factorial of 0 (one-liner): 1
```

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

❖ 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:

Cursor AI:

Cursor AI is a prompt-driven AI coding tool that enables code generation, refactoring, and explanation within a single environment. It is highly sensitive to prompt wording, producing noticeable changes in code structure, optimization, and documentation. Cursor AI excels in refactoring legacy code and improving readability and maintainability. Its interactive editor integration reduces context switching and improves productivity.