

AI Assisted Coding

Assignment 2.5

Name: R.Bharadwaj

Hall ticket no: 2303A51610

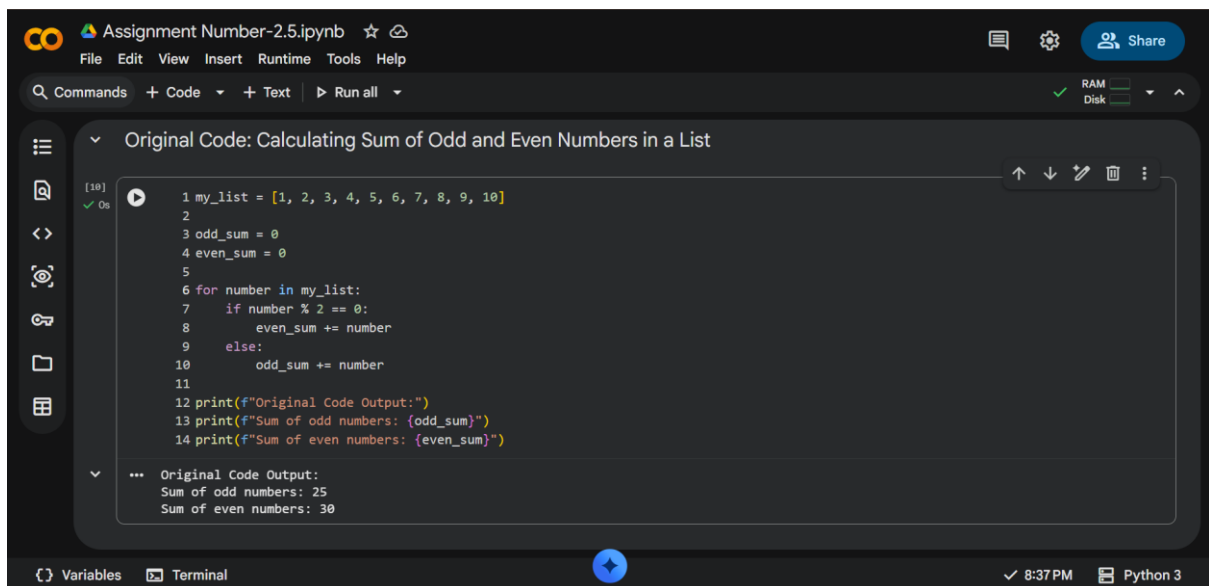
Batch no: 19

Task 1:

Prompt:

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

Code & Output:

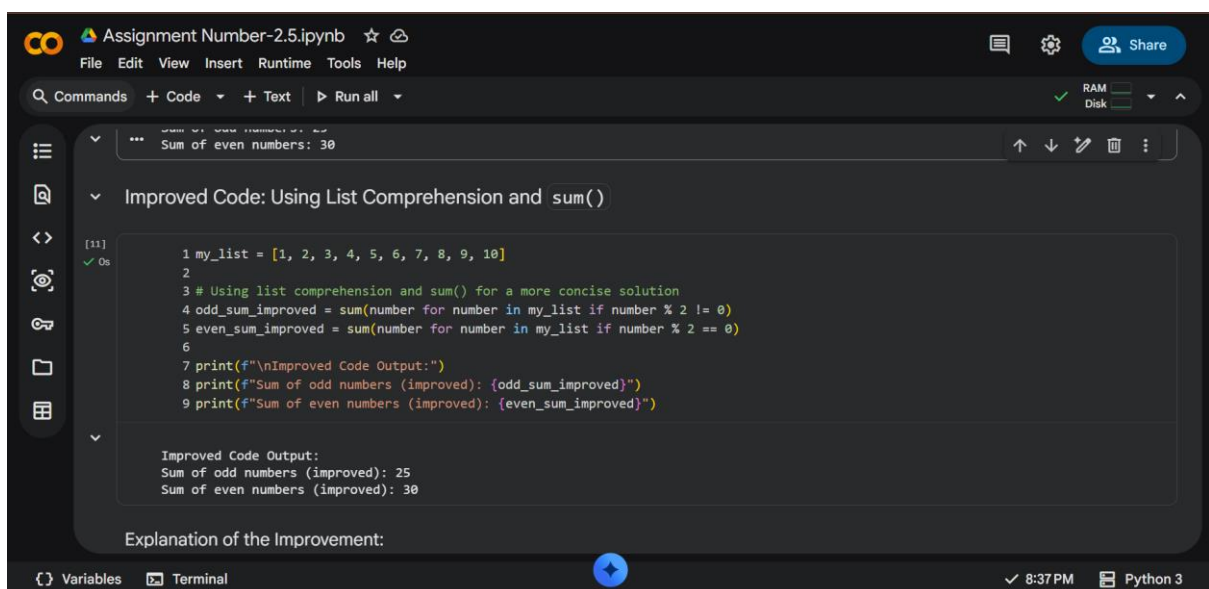


The screenshot shows a Jupyter Notebook titled "Assignment Number-2.5.ipynb". The code cell contains the following Python code:

```
1 my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2
3 odd_sum = 0
4 even_sum = 0
5
6 for number in my_list:
7     if number % 2 == 0:
8         even_sum += number
9     else:
10        odd_sum += number
11
12 print(f"Original Code Output:")
13 print(f"Sum of odd numbers: {odd_sum}")
14 print(f"Sum of even numbers: {even_sum}")
```

The output cell shows the following output:

```
... Original Code Output:
Sum of odd numbers: 25
Sum of even numbers: 30
```



The screenshot shows a Jupyter Notebook titled "Assignment Number-2.5.ipynb". The code cell contains the following Python code:

```
1 my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2
3 # Using list comprehension and sum() for a more concise solution
4 odd_sum_improved = sum(number for number in my_list if number % 2 != 0)
5 even_sum_improved = sum(number for number in my_list if number % 2 == 0)
6
7 print(f"\nImproved Code Output:")
8 print(f"Sum of odd numbers (improved): {odd_sum_improved}")
9 print(f"Sum of even numbers (improved): {even_sum_improved}")
```

The output cell shows the following output:

```
Improved Code Output:
Sum of odd numbers (improved): 25
Sum of even numbers (improved): 30
```

Explanation of the Improvement:

Explanation:

The original code uses a traditional for loop with if/else conditions to iterate through the list and accumulate sums. While functional, it is more verbose.

The improved code leverages Python's built-in functionalities:

- List Comprehension (Generator Expression): Instead of creating intermediate lists, a generator expression (number for number in my_list if condition) is used directly within sum(). This is memory-efficient as it generates values on the fly.
- sum() function: The sum() function is a highly optimized built-in that efficiently calculates the total of all items in an iterable.

This refactored approach is more *Pythonic*, concise, and often more readable for experienced Python developers, especially for simpler aggregation tasks.

Task 2:

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

Prompt:

Explain a function that calculates the area of different shapes.

Code:

```
import math
```

```
def calculate_area(shape, **kwargs):
```

```
    """
```

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

```
    Args:
```

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

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

```
            - 'radius' for circle
```

```
            - 'length', 'width' for rectangle
```

```
            - 'base', 'height' for triangle
```

```
    Returns:
```

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

```
        str: An error message if the shape is unknown or dimensions are missing.
```

```
    """
```

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

```
        radius = kwargs.get('radius')
```

```
        if radius is not None:
```

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

```

    else:
        return "Error: Circle requires 'radius'."
elif shape.lower() == 'rectangle':
    length = kwargs.get('length')
    width = kwargs.get('width')
    if length is not None and width is not None:
        return length * width
    else:
        return "Error: Rectangle requires 'length' and 'width'."
elif shape.lower() == 'triangle':
    base = kwargs.get('base')
    height = kwargs.get('height')
    if base is not None and height is not None:
        return 0.5 * base * height
    else:
        return "Error: Triangle requires 'base' and 'height'."
else:
    return f"Error: Unknown shape '{shape}'."

# --- Example Usage ---
print("--- Area Calculations ---")

# Circle
circle_area = calculate_area('circle', radius=5)
print(f"Area of a circle with radius 5: {circle_area:.2f}")

# Rectangle
rectangle_area = calculate_area('rectangle', length=4, width=6)
print(f"Area of a rectangle with length 4 and width 6: {rectangle_area}")

# Triangle
triangle_area = calculate_area('triangle', base=10, height=3)
print(f"Area of a triangle with base 10 and height 3: {triangle_area}")

# Invalid shape
invalid_shape_area = calculate_area('square', side=5)
print(f"Invalid shape calculation: {invalid_shape_area}")

```

```
# Missing dimensions
```

```
missing_dim_area = calculate_area('circle')
```

```
print(f"Missing dimension calculation: {missing_dim_area}")
```

Output:

```
--- Area Calculations ---
```

```
Area of a circle with radius 5: 78.54
```

```
Area of a rectangle with length 4 and width 6: 24
```

```
Area of a triangle with base 10 and height 3: 15.0
```

```
Invalid shape calculation: Error: Unknown shape 'square'.
```

```
Missing dimension calculation: Error: Circle requires 'radius'.
```

Explanation:

This `calculate_area` function is a versatile tool for finding the area of different shapes like circles, rectangles, and triangles. It uses a single function to handle multiple tasks, which is a good practice for organizing code.

Key Ideas:

1. **import math:** We import the `math` library to use `math.pi` (the value of π) for circle calculations.
2. **def calculate_area(shape, **kwargs):** This defines the function:
 - `shape`: This tells the function *which* shape to calculate (e.g., 'circle').
 - `**kwargs`: This is a flexible way to pass *extra information* like `radius=5`, `length=4`, etc. It gathers these into a dictionary.
3. **if/elif/else for Different Shapes:** The function checks the shape you provide:
 - If it's a **circle**, it looks for `radius` and uses $\pi * \text{radius}^2$.
 - If it's a **rectangle**, it looks for `length` and `width` and uses `length * width`.
 - If it's a **triangle**, it looks for `base` and `height` and uses `0.5 * base * height`.
4. **Error Handling:** If you forget to provide a necessary dimension (like `radius` for a circle) or provide an unknown shape, the function returns a helpful error message instead of crashing. This makes the function user-friendly.

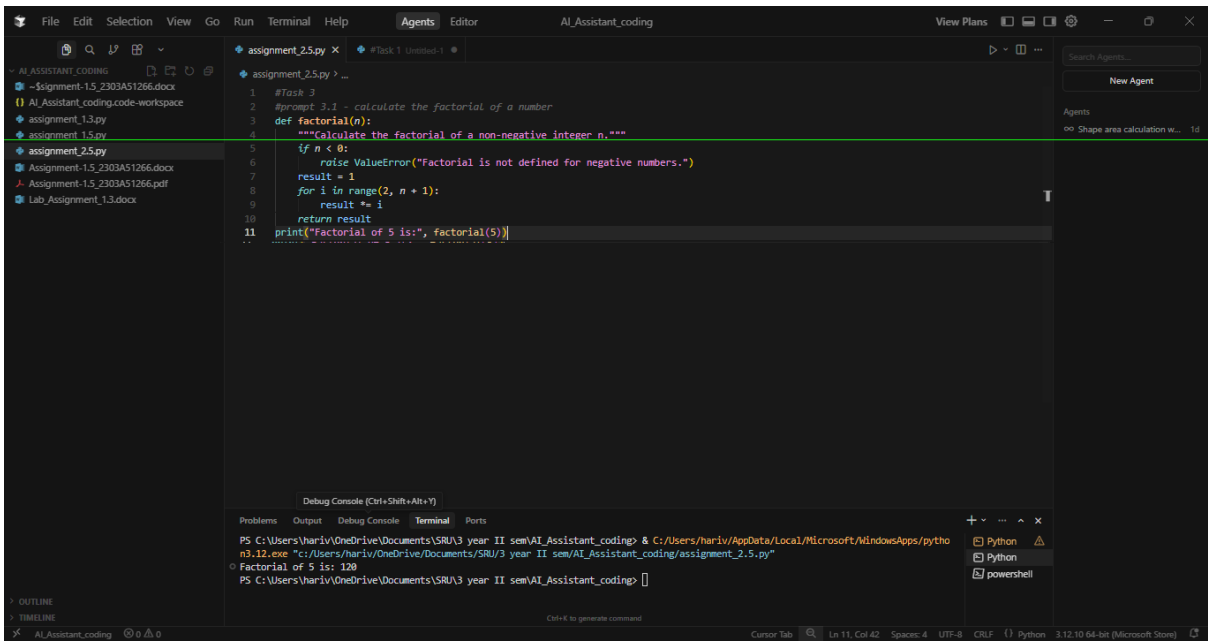
Task 3:

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

Prompt 3.1:

Calculate the factorial of a number

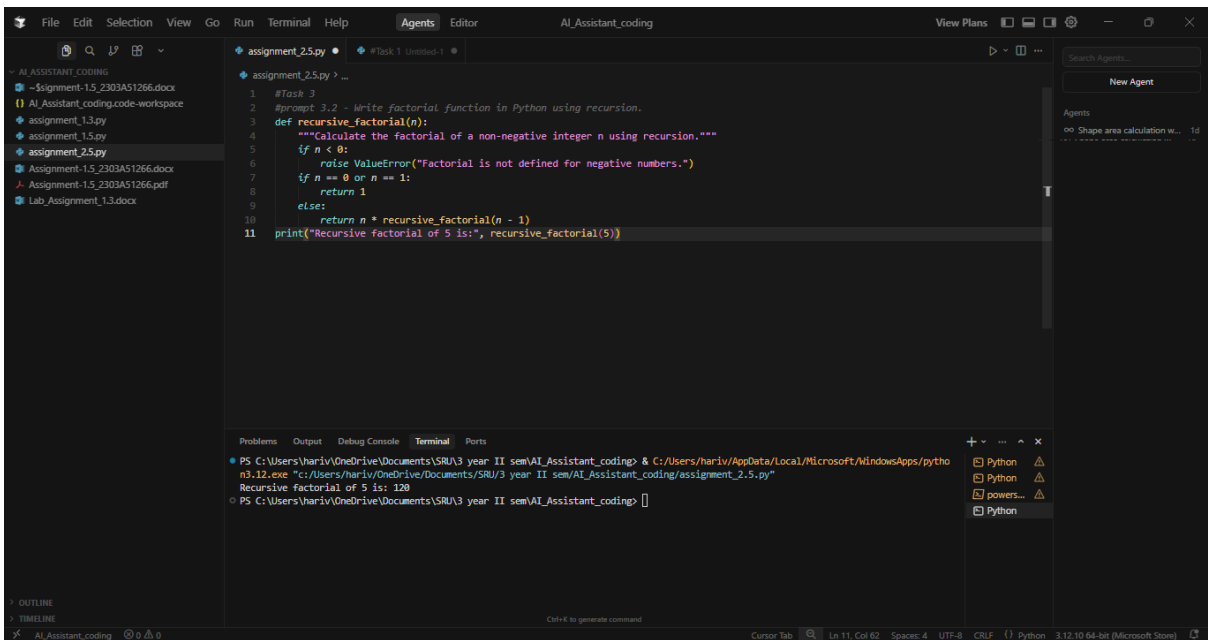
Code& Output:



Prompt 3.2 :

Write factorial function in python using recursion

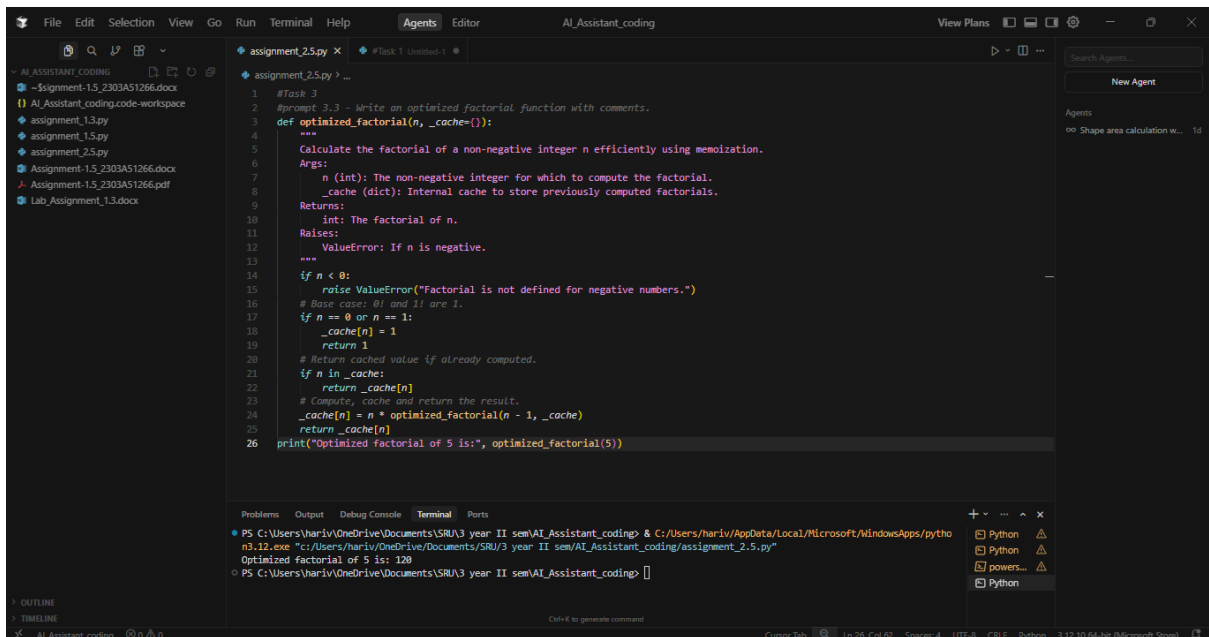
Code & Output:



Prompt 3.3:

Write an optional factorial function with comments.

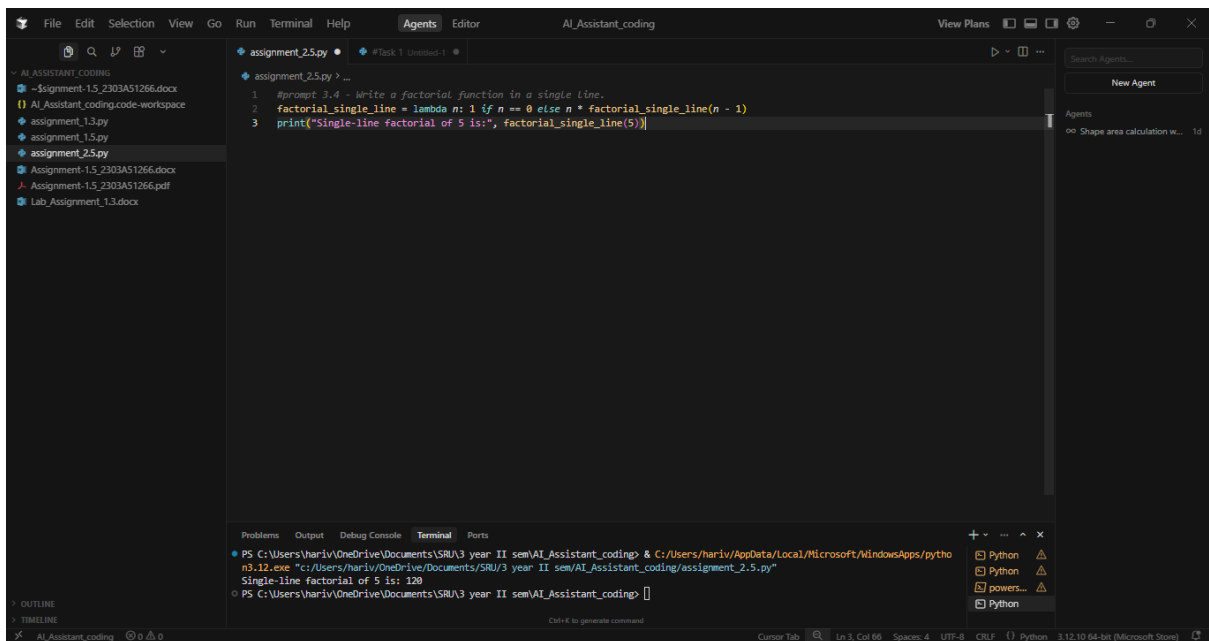
Code & Output:



Prompt 3.4:

Write a factorial function in a single line.

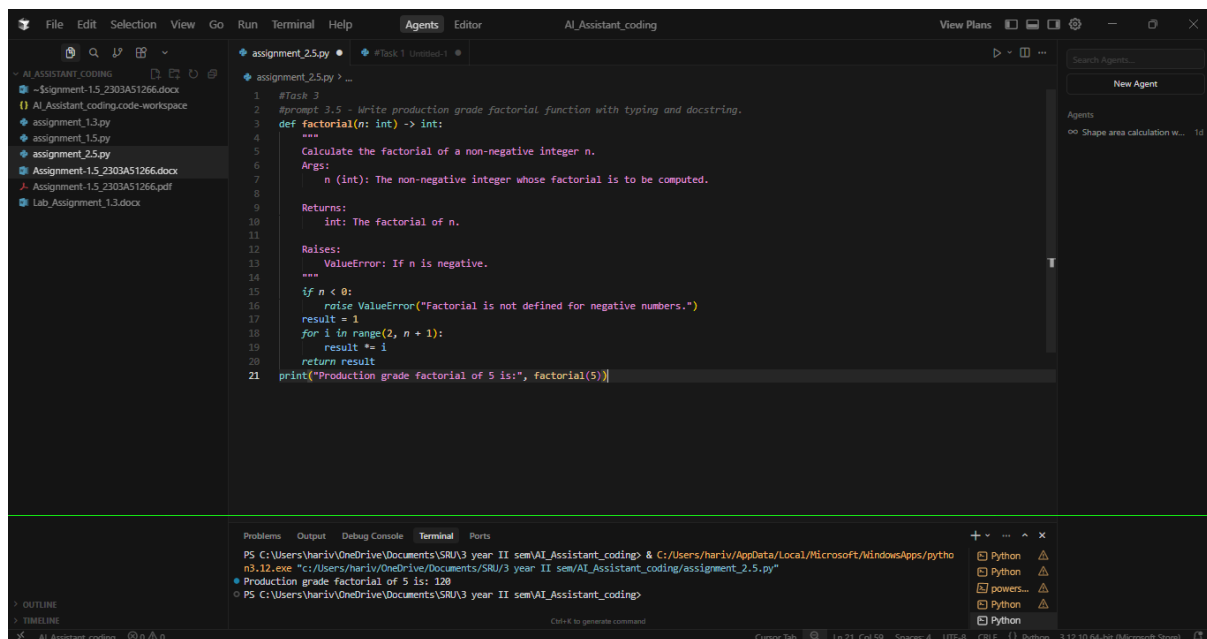
Code & Output:



Prompt 3.5:

Write production grade factorial function with typing and docstring.

Code & Output:



Explanation:

In this task, I asked Cursor to generate the factorial program using different prompts. Even though the problem was the same, the tool produced different versions of the code depending on how I asked. First, it gave a simple loop-based factorial, then a recursive version when I mentioned recursion, and later an optimized one when I asked for better performance. It also handled a single-line lambda version and finally a production-style version with typing and docstrings. This shows that the tool understands prompt details and adjusts the code style, structure, and features based on how clearly the prompt is written.

Task 4:

Compare Gemini, Copilot, and Cursor AI for usability and code quality.

Short reflection:

Gemini, Copilot, and Cursor AI each help with coding in different ways, especially when it comes to usability and code quality. Gemini is very good at explaining ideas and showing clear, readable examples. It helps you understand how code works, but it usually sticks to simple and safe patterns and does not easily adjust to different styles or advanced needs. This makes it better for learning than for serious development work.

GitHub Copilot is designed to make coding faster inside an editor. It completes lines and suggests blocks of code based on what you start typing. This improves speed, but it does not really think about whether the logic is correct or well designed. If you make a mistake, Copilot will continue building on that mistake.

Cursor AI is the most flexible and powerful of the three. It changes the way it writes code based on how the problem is asked. The same task can result in different styles such as simple, optimized, or production-level code. This shows that Cursor understands the intent behind the prompt and produces better structured and more professional code.

Because of this, Cursor AI is the best overall choice. It combines good usability with high-quality, adaptable code, making it the most suitable tool for real-world software development.