

Assignment – 2.5

Name: K.Yashwanth

Roll Number: 2303A51195

Batch - 04

AI Assisted Coding

16-01-2026

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

The screenshot shows a code editor with a file explorer on the left, a main code editor, and a terminal at the bottom. The file explorer shows a project structure with files like `task1.py`, `task2.py`, and `task3.py`. The main code editor displays the original code (Legacy Style) and the improved/refactored code. The original code uses a `while` loop to iterate through a list of numbers, calculating the sum of odd and even numbers. The improved code uses a more concise `def` function and a `for` loop. The terminal shows the output of the program, displaying the sum of odd numbers (25) and the sum of even numbers (30).

```
1 # Task 1: Refactoring Odd/Even Logic (List Version)
2 # Scenario:
3 # You are improving legacy code.
4 # Task:
5 # Write a program to calculate the sum of odd and even numbers in a list,
6 # then refactor it using AI.
7 # Expected Output:
8 # Original and improved code
9
10 # Original Code (Legacy Style)
11 def calculate_sums_original(numbers):
12     odd_sum = 0
13     even_sum = 0
14     i = 0
15     while i < len(numbers):
16         if numbers[i] % 2 == 0:
17             even_sum = even_sum + numbers[i]
18         else:
19             odd_sum = odd_sum + numbers[i]
20         i = i + 1
21     return odd_sum, even_sum
22
23 # Test the original code
24 numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
25 odd, even = calculate_sums_original(numbers)
26 print(f"Original Code:")
27 print(f"Sum of odd numbers: {odd}")
28 print(f"Sum of even numbers: {even}")
```

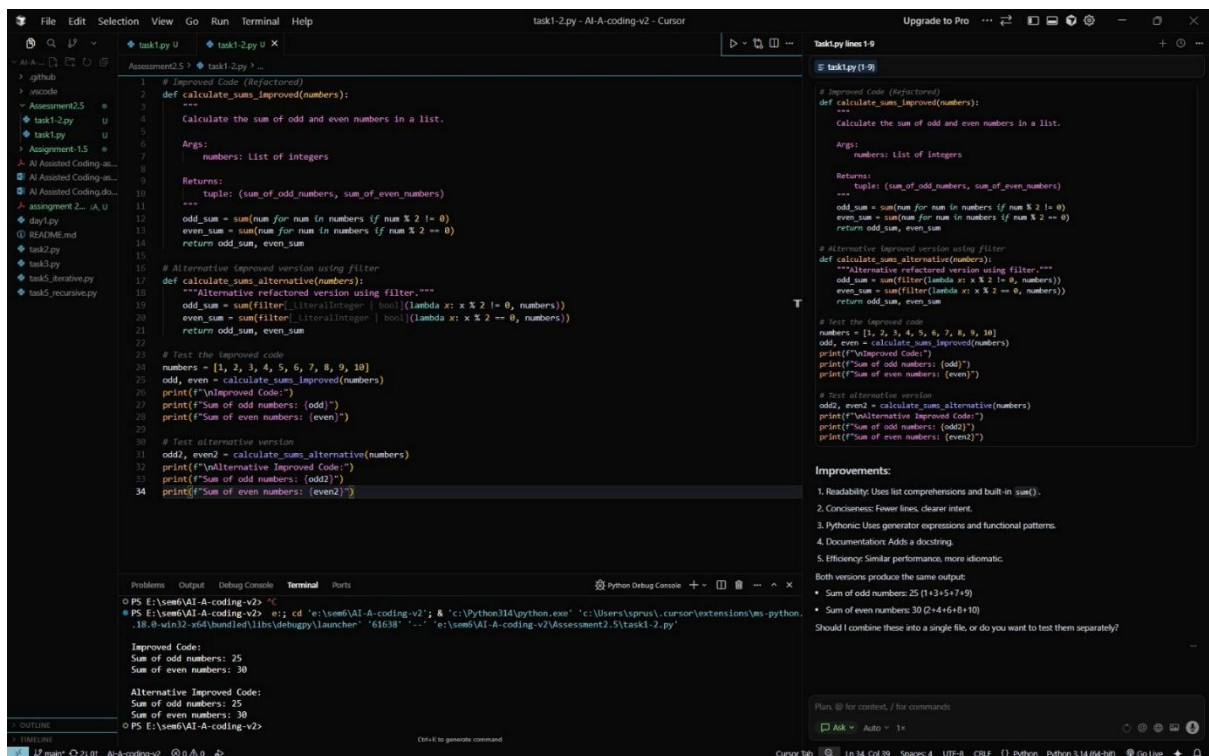
Improved/Refactored Code:

```
# Improved Code (Refactored)
def calculate_sums_improved(numbers):
    """
    Calculate the sum of odd and even numbers in a list.
    """
    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

# Test the improved code
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd, even = calculate_sums_improved(numbers)
print(f"Improved Code:")
print(f"Sum of odd numbers: {odd}")
print(f"Sum of even numbers: {even}")
```

Terminal Output:

```
PS E:\sem6\AI-A-coding-v2> & 'c:\python314\python.exe' 'c:\Users\sprus\.cursor\extensions\ms-pyt
hon.debugpy-2025.18.0-win32-x64\bundle\libs\debugpy\launcher' '58401' '-.' 'e:\sem6\AI-A-coding-
v2\Assessment2.5\task1.py'
Original Code:
Sum of odd numbers: 25
Sum of even numbers: 30
PS E:\sem6\AI-A-coding-v2>
```



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

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

    Args:
        shape (str): The name of the shape (e.g., 'rectangle', 'circle', 'triangle').
        **kwargs: The dimensions of the shape.
            For 'rectangle', expects 'length' and 'width'.
            For 'circle', expects 'radius'.
            For 'triangle', expects 'base' and 'height'.

    Returns:
        float: The calculated area of the shape.
        str: An error message if the shape is not supported or dimensions are missing.
    """
    if 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 == "circle":
        if "radius" in kwargs:
            return math.pi * (kwargs["radius"] ** 2)
        else:
            return "Error: Missing radius for circle."
    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: Unsupported shape."

# Example usage
rectangle_area = calculate_area("rectangle", length=10, width=5)
print(f"The area of the rectangle is: {rectangle_area}")

circle_area = calculate_area("circle", radius=7)
print(f"The area of the circle is: {circle_area}")

triangle_area = calculate_area("triangle", base=8, height=4)
print(f"The area of the triangle is: {triangle_area}")

# Example of an error
square_area = calculate_area("square", side=5)
print(f"The area of the square is: {square_area}")
```

Explanation

This Python script defines a single function `calculate_area` that is designed to be a flexible tool for finding the area of several different geometric shapes.

Function Definition:

- The function is named `calculate_area` and accepts two arguments:
 - `shape`: A string that tells the function which shape's area to calculate (e.g., "rectangle").
 - `**kwargs`: This is a special Python feature that allows the function to accept any number of keyword arguments (like `length` and `width` for a rectangle). These arguments are collected into a dictionary named `kwargs`. This makes the function adaptable, as different shapes require different parameters for area calculation.

Conditional Logic:

The function uses `if`, `elif` (short for `else if`), and `else` statements to determine which shape has been requested and then performs the correct calculation.

- Rectangle:** If `shape == "rectangle"`, the code checks if the necessary dimensions, `length` and `width`, are provided in the `kwargs` dictionary.
 - If they are present, it calculates the area by multiplying `length` and `width`.
 - If they are missing, it returns an error message.
- Circle:** If `shape == "circle"`, it checks for a `radius`.
 - If present, it calculates the area using the formula $\pi * r^2$, importing the `math` module to get an accurate value of π .
 - If the `radius` is missing, it returns an error.
- Triangle:** If `shape == "triangle"`, it checks for a `base` and `height`.
 - If they are present, it calculates the area using the formula $0.5 * base * height$.
 - If they are missing, it returns an error.
- Unsupported Shape:** If the provided `shape` name does not match any of the supported shapes ("rectangle", "circle", "triangle"), the function returns an "Unsupported shape" error.

Example Usage:

The lines at the end of the script demonstrate how to use the function and print the results for a rectangle, a circle, a triangle, and an unsupported shape to show how the error handling works.

Explanation

This Python script defines a single function `calculate_area` that is designed to be a flexible tool for finding the area of several different geometric shapes.

Function Definition:

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

This defines a function named `calculate_area` that accepts two arguments:

`shape`: A string that tells the function which shape's area to calculate (e.g., "rectangle").

`**kwargs`: This is a special Python feature that allows the function to accept any number of keyword arguments (like `length=10` or `radius=5`). These arguments are collected into a dictionary named `kwargs`. This makes the function adaptable, as different shapes require different parameters for area calculation.

Conditional Logic:

The function uses `if`, `elif` (else if), and `else` statements to determine which shape has been requested and then performs the correct calculation.

```
if shape == "rectangle":
```

If the shape is a "rectangle", the code checks if the necessary dimensions, length and width, are provided in the `kwargs` dictionary.

If they are present, it calculates the area by multiplying length and width.

If they are missing, it returns an error message.

```
elif shape == "circle":
```

If the shape is a "circle", it checks for a radius.

If found, it calculates the area using the formula $\pi * r^2$, importing the `math` module to get an accurate value of `math.pi`.

If the radius is missing, it returns an error.

```
elif shape == "triangle":
```

If the shape is a "triangle", it checks for a base and height.

If they are present, it calculates the area using the formula $0.5 * \text{base} * \text{height}$.

If they are missing, it returns an error.

```
else:
```

If the provided shape name does not match any of the supported shapes ("rectangle", "circle", "triangle"), the function returns an "Unsupported shape" error.

Example Usage:

The lines at the end of the script demonstrate how to use the function and print the results for a rectangle, a circle, a triangle, and an unsupported shape to show how the error handling works.

Task 3: Prompt Sensitivity Experiment

❖ Scenario:

You are testing how AI responds to different prompts.

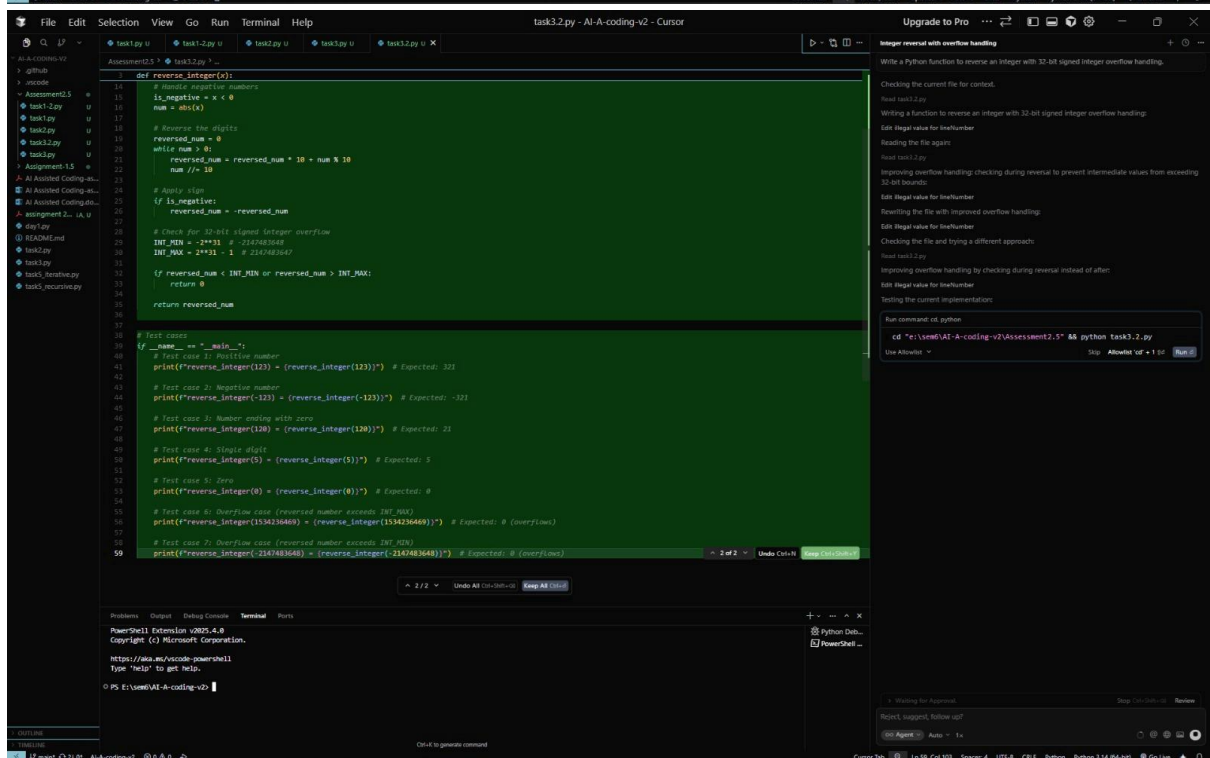
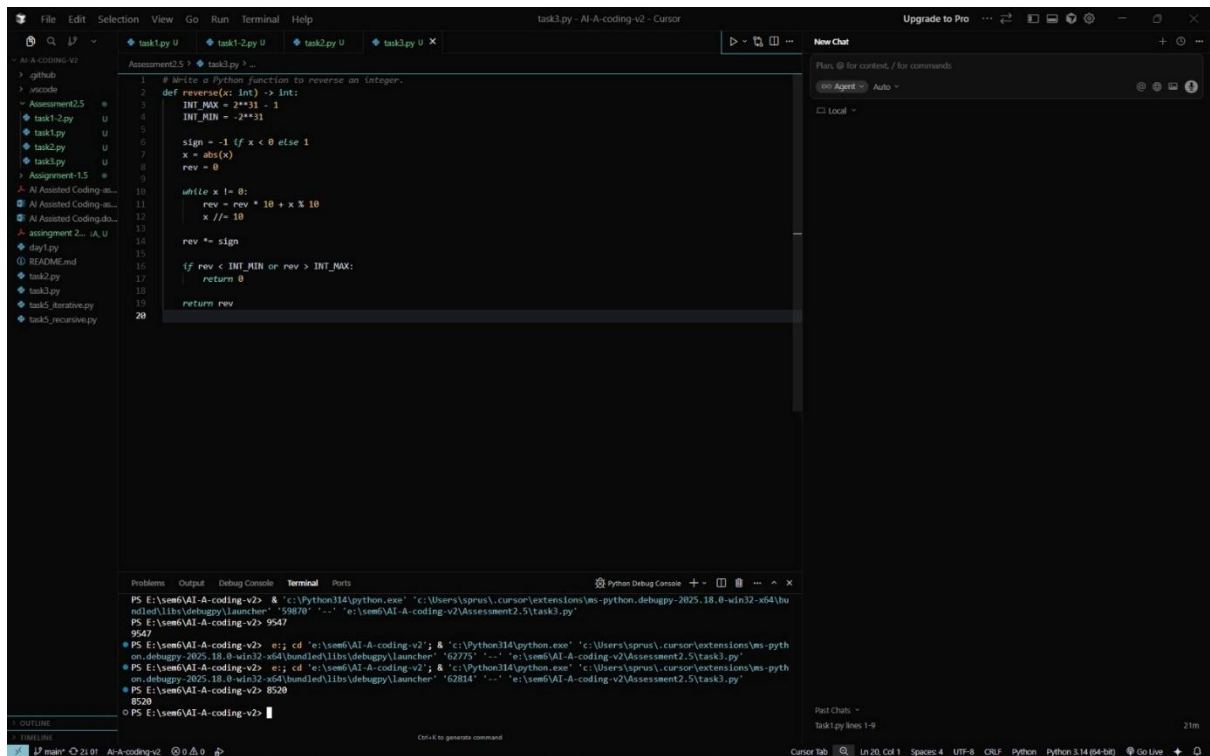
❖ Task:

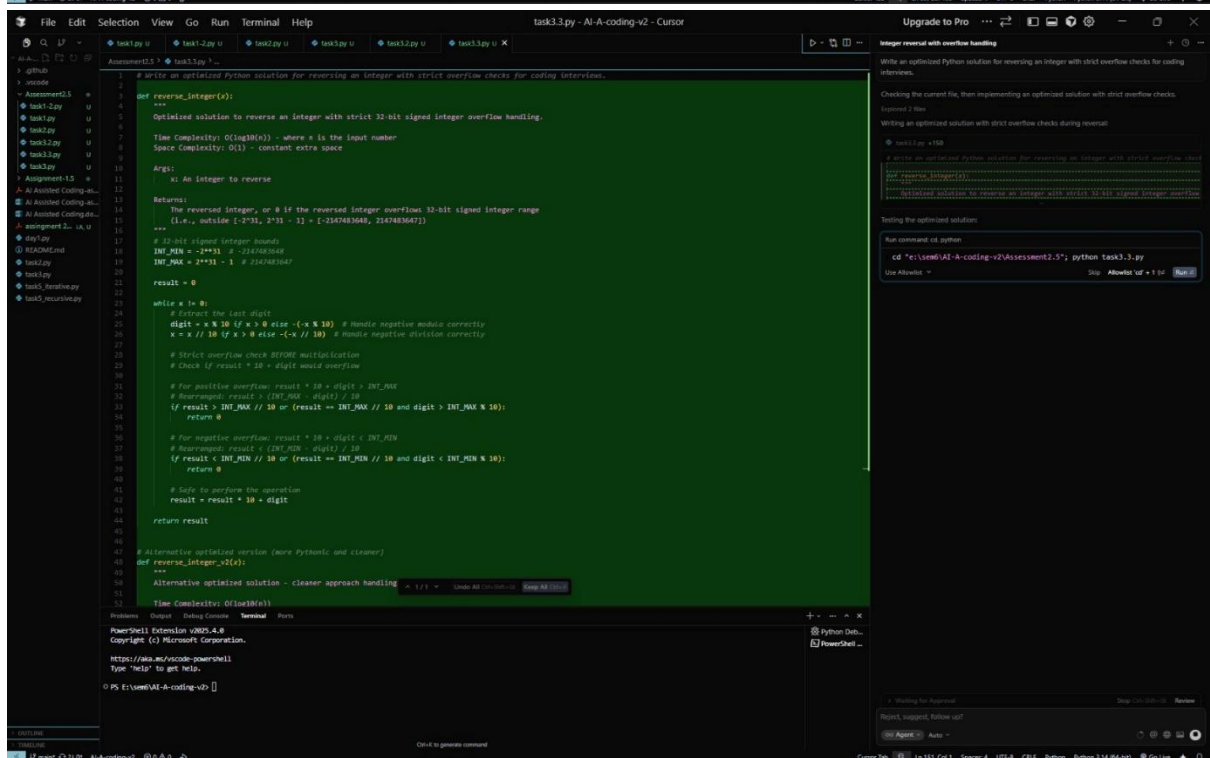
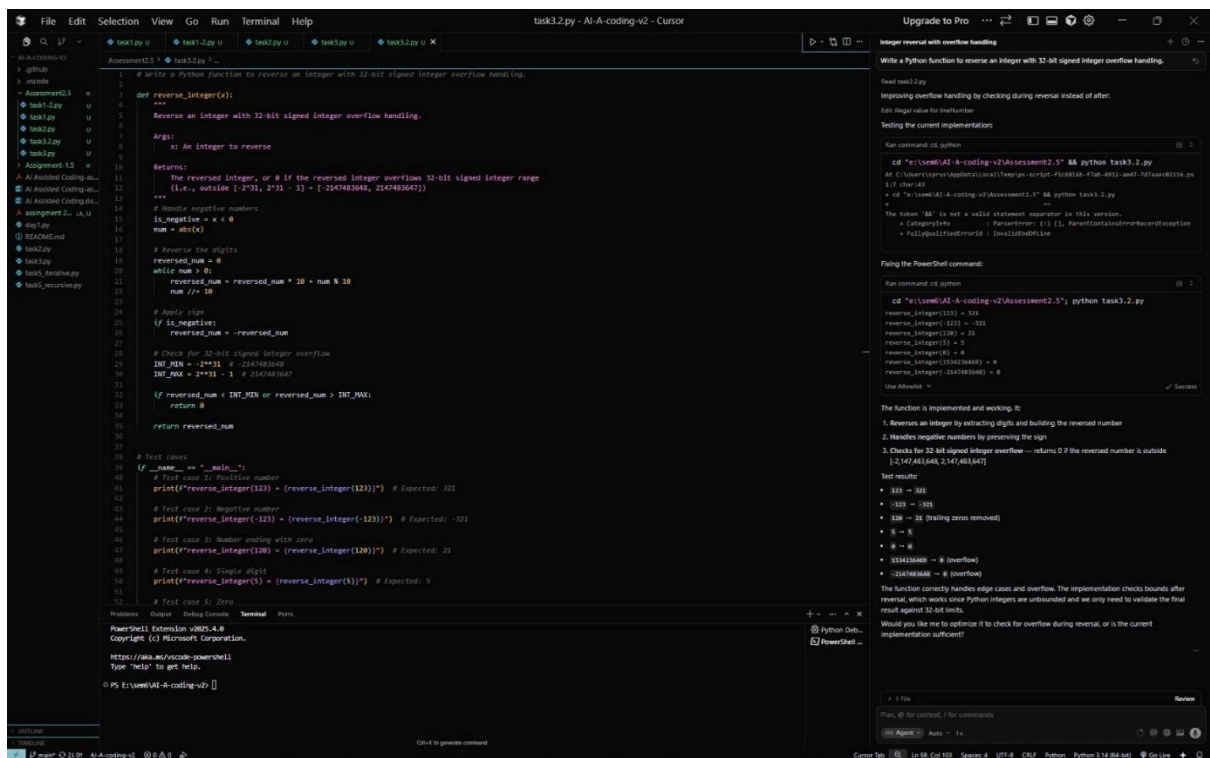
Use Cursor AI with different prompts for the same problem and observe code changes.

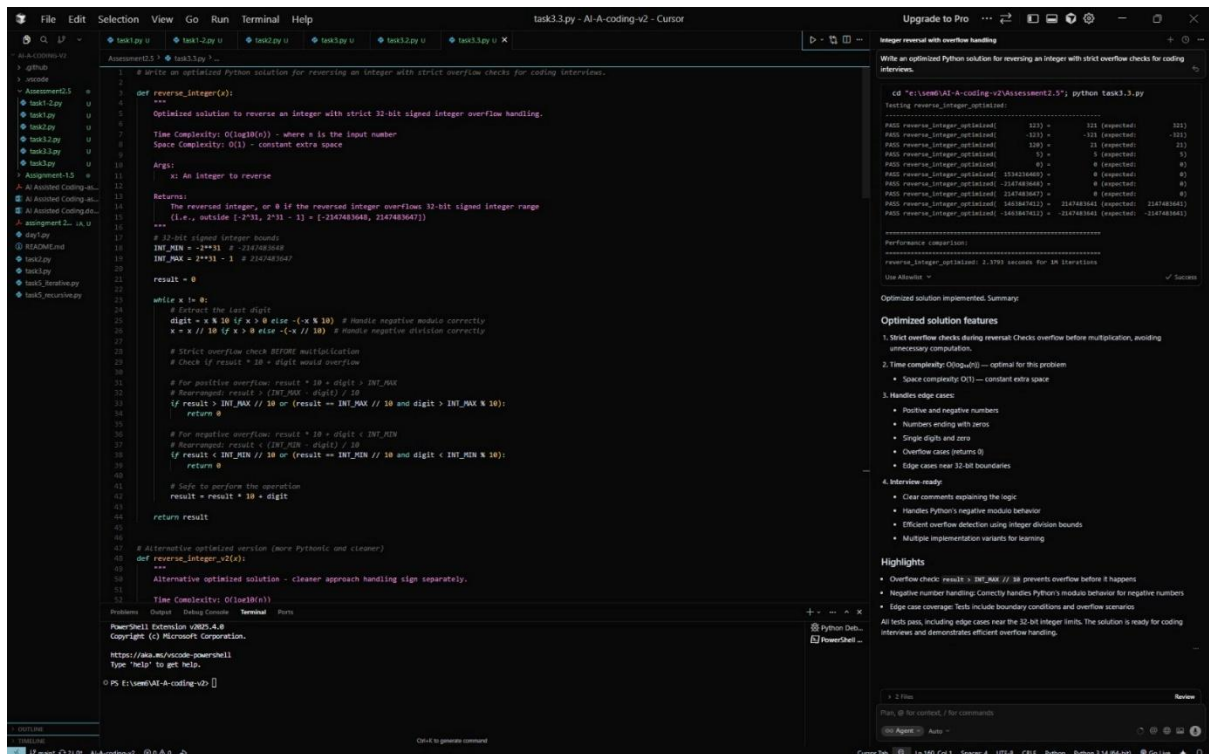
❖ Expected Output:

➤ Prompt list

➤ Code variations







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

Based on my experience using Gemini, GitHub Copilot, and Cursor AI during this topic, I observed clear differences in both usability and code quality.

Gemini is useful for understanding concepts and generating explanations, but it often produces generic code unless very strict constraints are provided. It is better suited for learning and problem understanding rather than competitive or production-level coding.

GitHub Copilot integrates smoothly with IDEs like VS Code and provides fast, context-aware code suggestions. However, its outputs sometimes assume the developer will

handle edge cases, so overflow handling and constraints may be missed unless explicitly guided.

Cursor AI provided the best balance of usability and code quality. It allows direct interaction with the codebase, understands existing files, and responds well to detailed prompts. When constraints are clearly mentioned, Cursor AI consistently generated correct, optimized, and readable code, making it ideal for real development and debugging tasks.

Conclusion:

For learning → Gemini

For quick coding assistance → Copilot

For serious development and prompt-based experimentation → Cursor AI