

AI ASSISTED CODING

LAB– 9.2

NAME : CH.RAKSHAVARDHAN

ENROLL.NO:2403A52034

BATCH:03

Task-1:

(Documentation – Google-Style Docstrings for Python Functions)

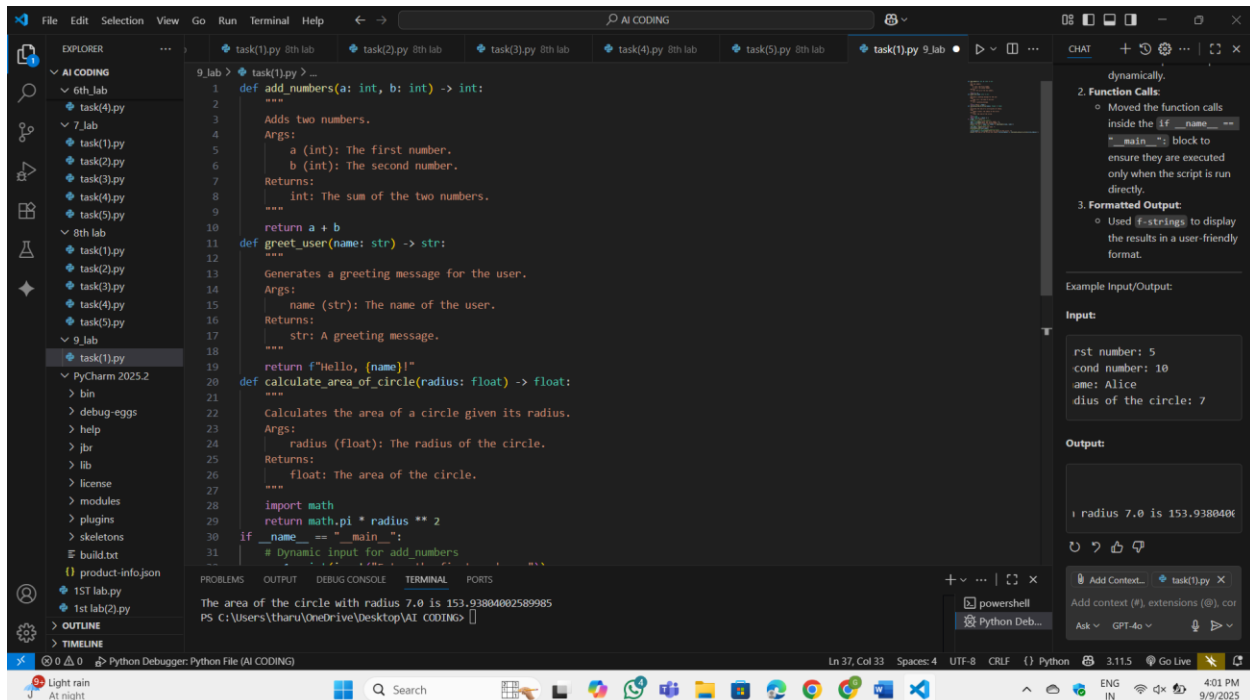
- Task: Use AI to add Google-style docstrings to all functions in a given Python script.
- Instructions:
 - o Prompt AI to generate docstrings without providing any input-output examples.
 - o Ensure each docstring includes:
 - Function description
 - Parameters with type hints
 - Return values with type hints
 - Example usage
 - o Review the generated docstrings for accuracy and formatting.
- Expected Output #1:
 - o A Python script with all functions documented using correctly formatted Google-style docstrings

Prompt:

Add Google-style docstrings to all functions in a given Python script.

Ensure each docstring includes: Function description, Parameters with type hints, Return values with type hints

Code:



The screenshot shows a VS Code editor with a Python file named `task(1).py` open. The code defines three functions: `add_numbers`, `greet_user`, and `calculate_area_of_circle`. Each function has a Google-style docstring with a description, arguments, and returns. The `add_numbers` function takes two integers and returns their sum. The `greet_user` function takes a string name and returns a greeting message. The `calculate_area_of_circle` function takes a float radius and returns the area of the circle. The code also includes a `__main__` block that calls `add_numbers` and `calculate_area_of_circle` with sample inputs.

```
1 def add_numbers(a: int, b: int) -> int:
2     """
3     Adds two numbers.
4     Args:
5         a (int): The first number.
6         b (int): The second number.
7     Returns:
8         int: The sum of the two numbers.
9     """
10    return a + b
11
12 def greet_user(name: str) -> str:
13    """
14    Generates a greeting message for the user.
15    Args:
16        name (str): The name of the user.
17    Returns:
18        str: A greeting message.
19    """
20    return f"Hello, {name}!"
21
22 def calculate_area_of_circle(radius: float) -> float:
23    """
24    Calculates the area of a circle given its radius.
25    Args:
26        radius (float): The radius of the circle.
27    Returns:
28        float: The area of the circle.
29    """
30    import math
31    return math.pi * radius ** 2
32
33 if __name__ == "__main__":
34     # Dynamic input for add_numbers
35     a = input("Enter first number: ")
36     b = input("Enter second number: ")
37     sum = add_numbers(int(a), int(b))
38     print(f"Sum: {sum}")
39
40     radius = input("Enter radius: ")
41     area = calculate_area_of_circle(float(radius))
42     print(f"Area: {area}")
```

The chat window on the right shows the following content:

dynamically.

2. Function Calls:

- Moved the function calls inside the `if __name__ == "__main__":` block to ensure they are executed only when the script is run directly.

3. Formatted Output:

- Used `f-strings` to display the results in a user-friendly format.

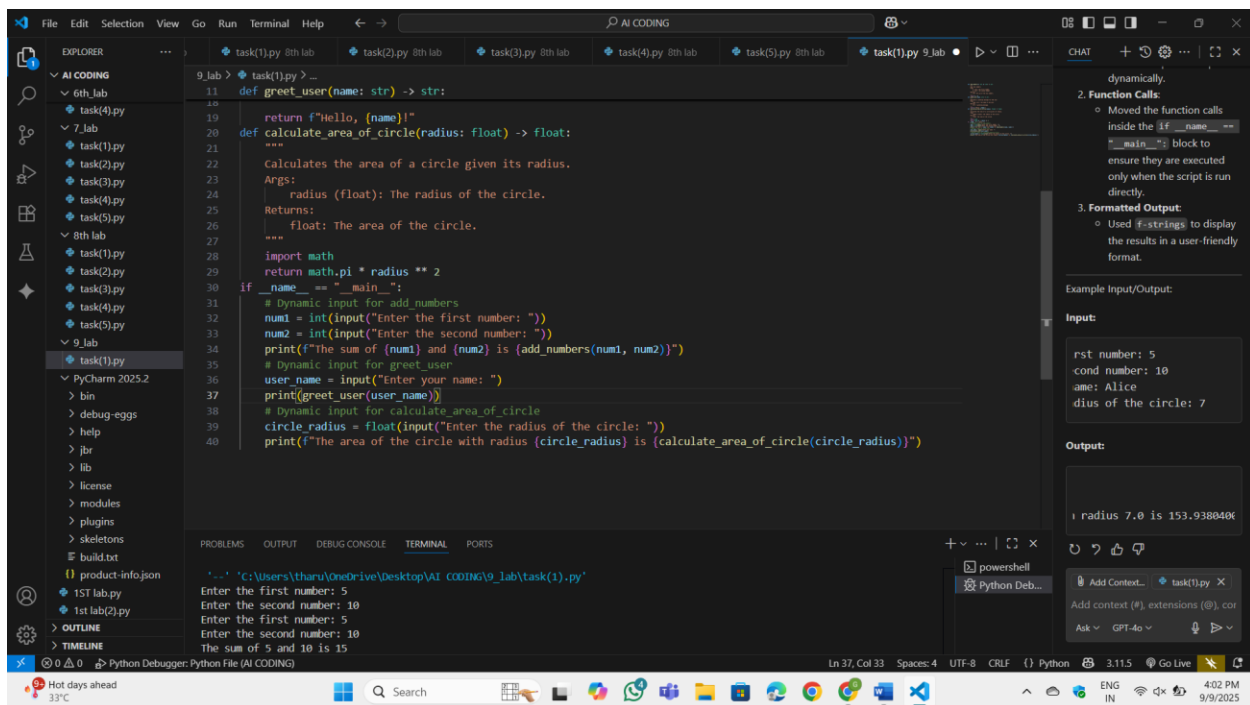
Example Input/Output:

Input:

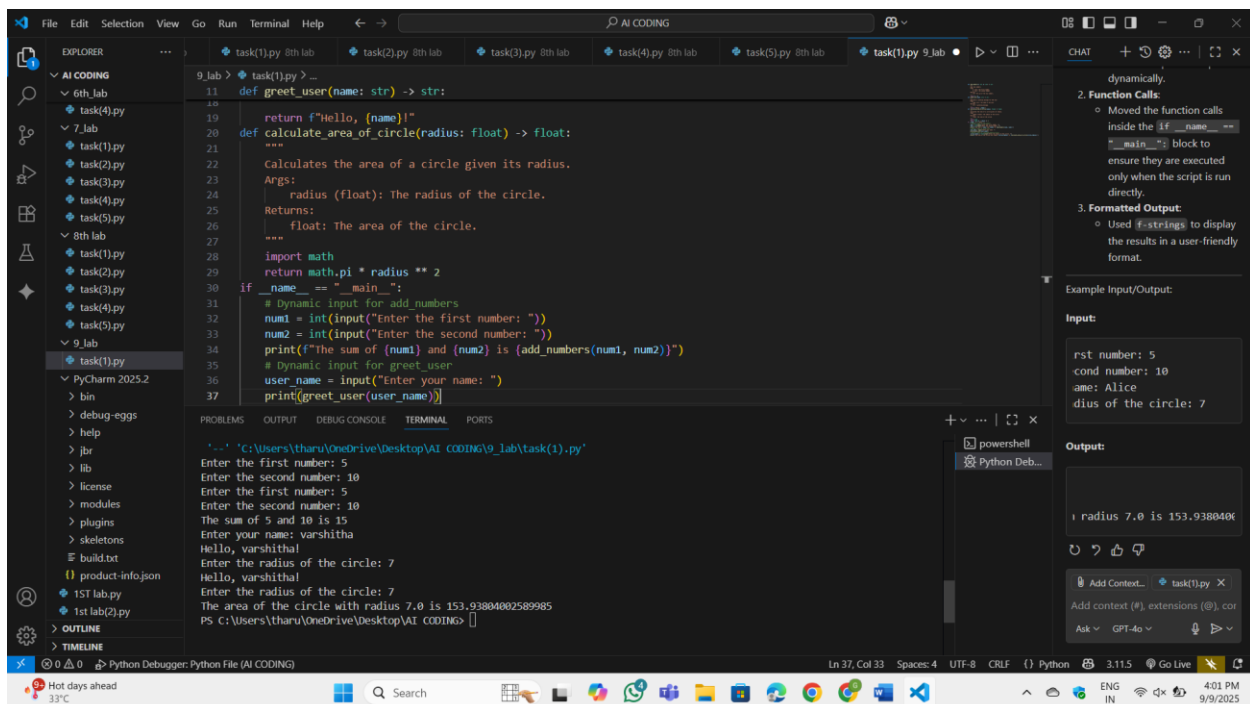
```
rst number: 5
cond number: 10
ame: Alice
dius of the circle: 7
```

Output:

```
r radius 7.0 is 153.9380400
```



OUTPUT:



Observation:

Added input() prompts for each function to allow the user to provide input dynamically. Moved the function calls inside the if `__name__ == "__main__":` block to ensure they are executed only when the script is run directly. Used f-strings to display the results in a user-friendly format.

Task-2:

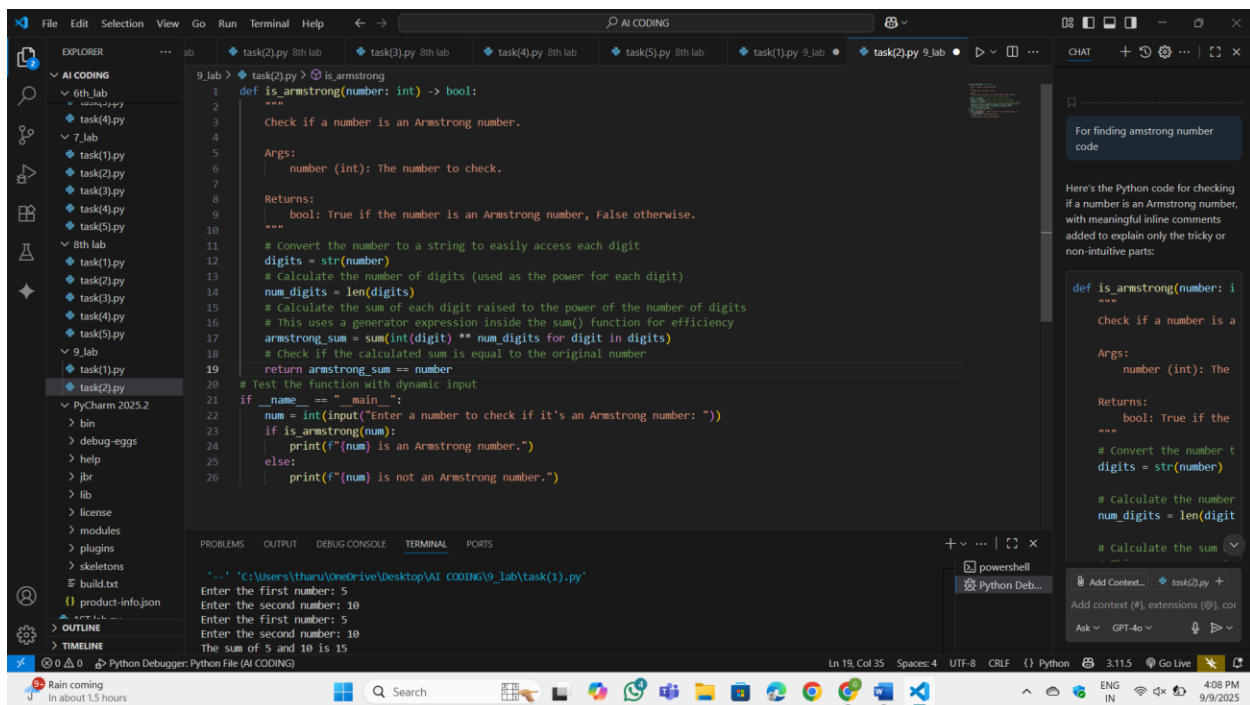
(Documentation – Inline Comments for Complex Logic)

- Task: Use AI to add meaningful inline comments to a Python program explaining only complex logic parts.
- Instructions:
 - o Provide a Python script without comments to the AI.
 - o Instruct AI to skip obvious syntax explanations and focus only on tricky or non-intuitive code sections.
 - o Verify that comments improve code readability and maintainability.
- Expected Output #2:
 - o Python code with concise, context-aware inline comments for complex logic blocks

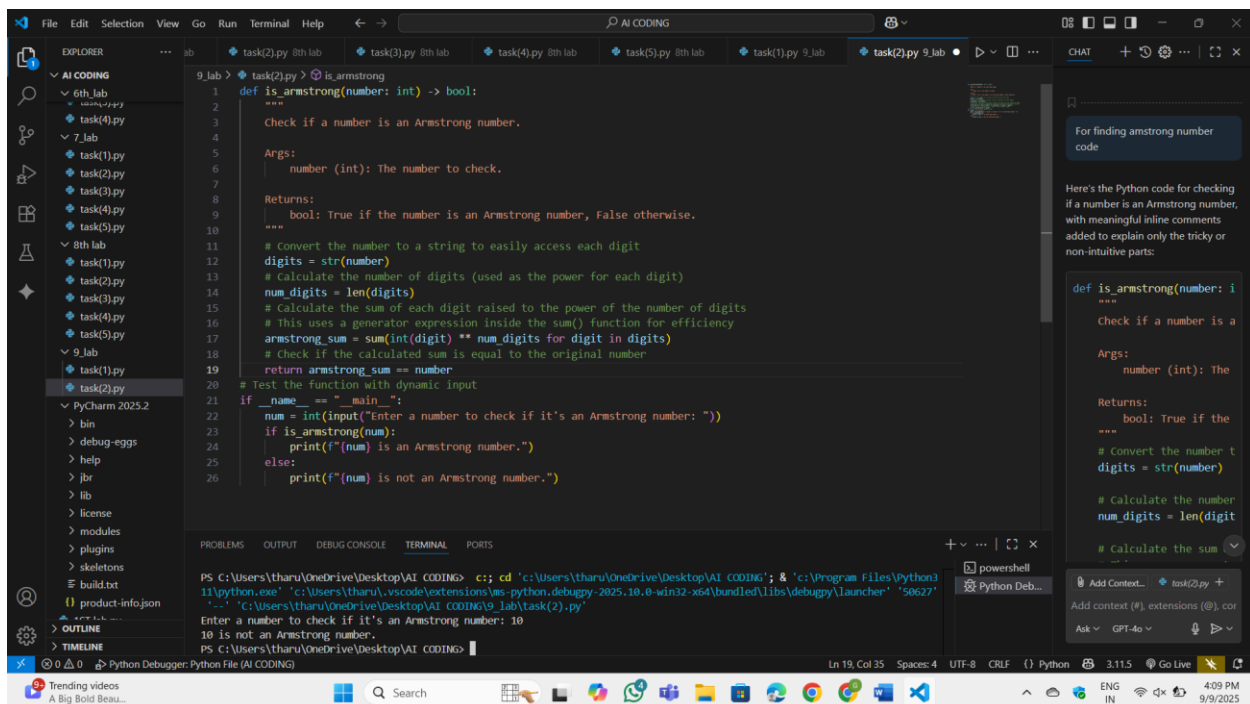
Prompt :

For the code I provided focus only on tricky or non-intuitive code sections. And add meaningful inline comments to a Python program explaining only complex logic parts

Code:



OUTPUT:



Observation:

`digits = str(number)`: Converts the number to a string to allow iteration over its digits. `num_digits = len(digits)`: Calculates the number of digits in the number, which determines the power to which each digit is raised. `sum(int(digit) ** num_digits for digit in digits)`: Uses a generator expression to calculate the sum of each digit raised to the power of `num_digits`. This avoids creating an intermediate list, making the code more memory-efficient. `return armstrong_sum == number`: Compares the calculated sum to the original number to determine if it's an Armstrong number

Task-3:

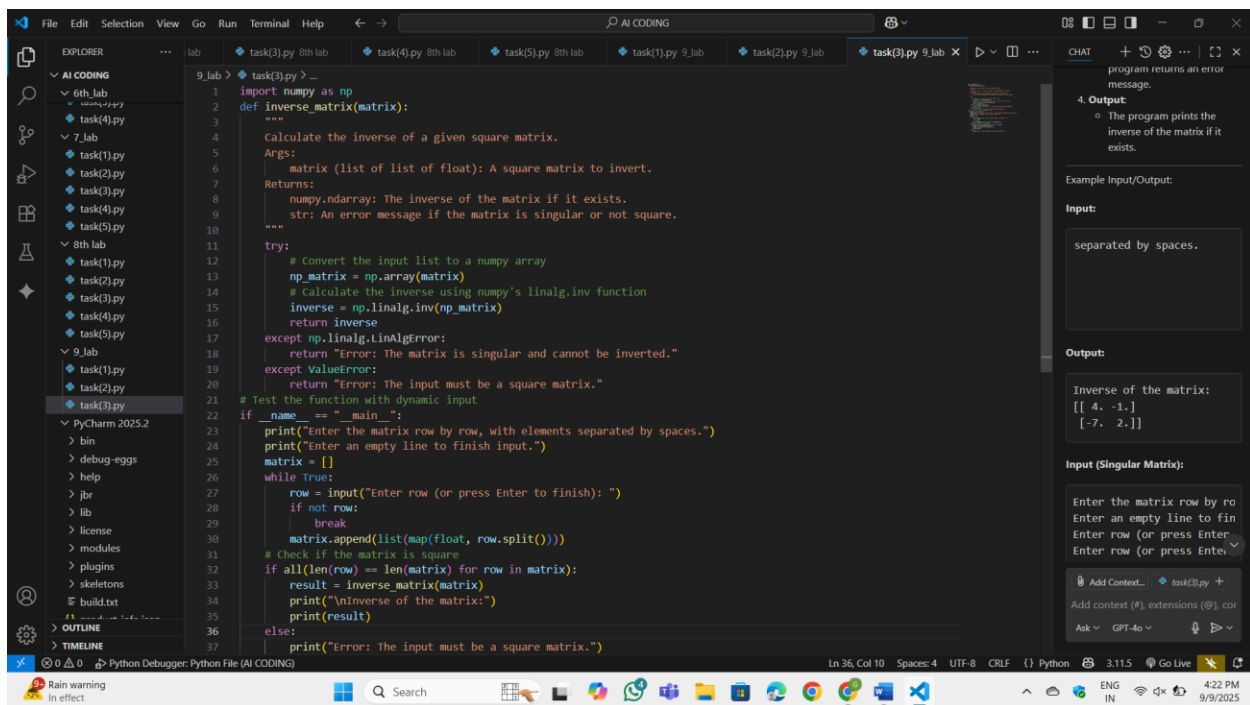
(Documentation – Module-Level Documentation)

- Task: Use AI to create a module-level docstring summarizing the purpose, dependencies, and main functions/classes of a Python file.
- Instructions:
 - o Supply the entire Python file to AI.
 - o Instruct AI to write a single multi-line docstring at the top of the file.
 - o Ensure the docstring clearly describes functionality and usage without rewriting the entire code

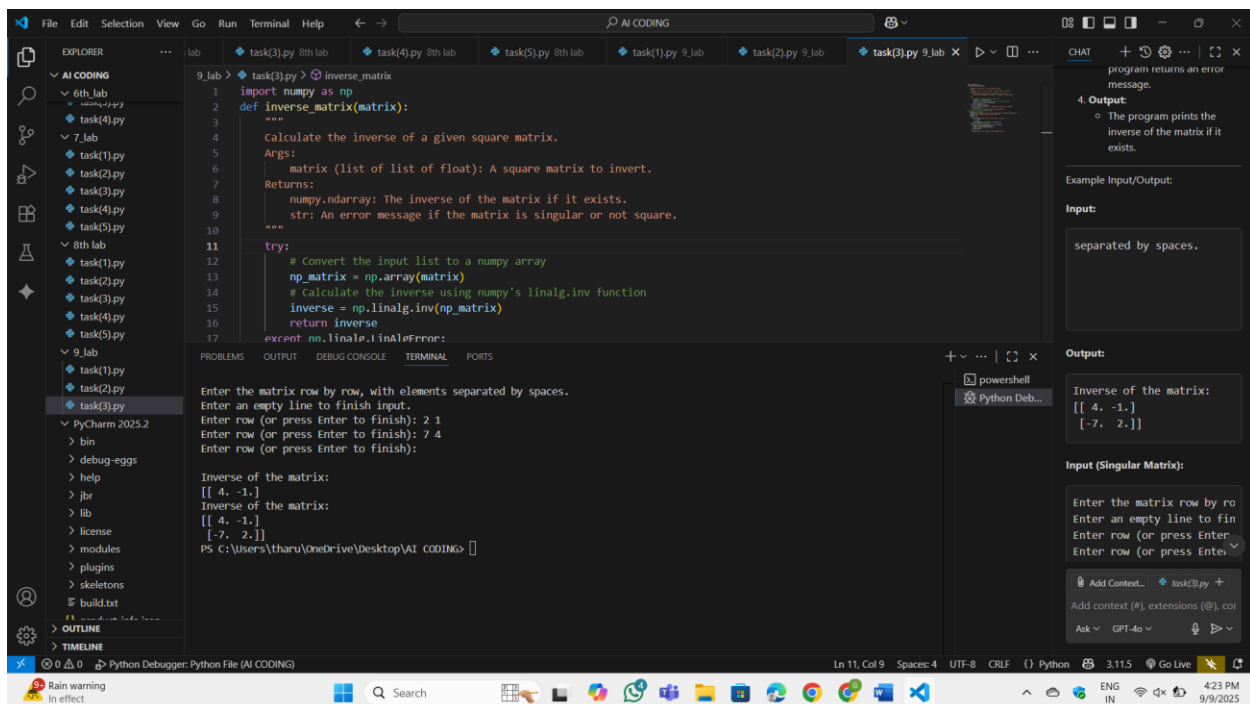
Prompt:

For the given code describes functionality and usage without rewriting the entire code

Code:



OUTPUT:



OBSERVATION:

The numpy library is used for matrix operations.

The `np.linalg.inv()` function calculates the inverse of a matrix. The user inputs the matrix row by row. The input is converted into a list of lists, where each inner list represents a row. If the matrix is not square, the program returns an error message. If the matrix is singular (determinant = 0), the program returns an error message. The program prints the inverse of the matrix if it exists.

Task-4:

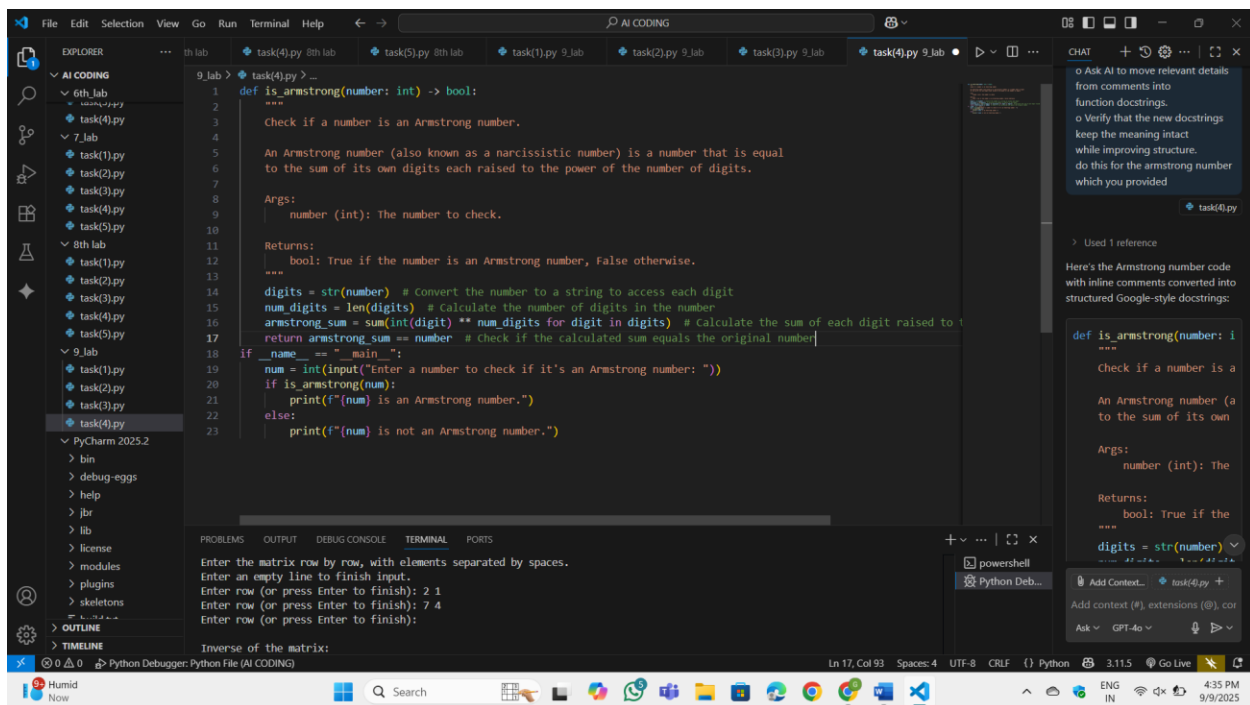
(Documentation – Convert Comments to Structured Docstrings)

- Task: Use AI to transform existing inline comments into structured function docstrings following Google style.
- Instructions:
 - o Provide AI with Python code containing inline comments.
 - o Ask AI to move relevant details from comments into function docstrings.
 - o Verify that the new docstrings keep the meaning intact while improving structure.
- Expected Output #4:
 - o Python code with comments replaced by clear, standardized docstrings

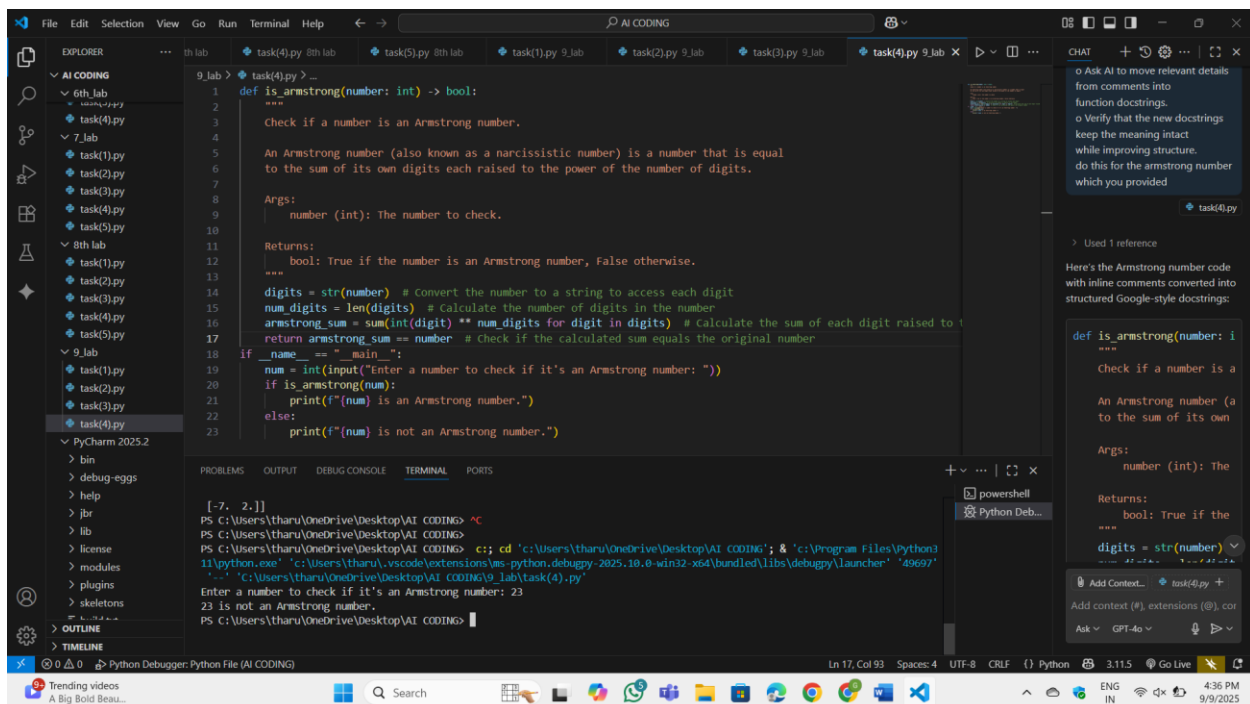
Prompt:

convert Comments to Structured Docstrings for the code which I provided.

Code:



OUTPUT:



OBSERVATION:

Removed inline comments from the code. Added a detailed docstring to the `is_armstrong` function. The docstring explains: **Purpose:** What the function does. **Args:** The input parameter and its type. **Returns:** The return value and its type. **Logic:** A brief explanation of the Armstrong number concept. The docstring improves the structure and readability of the code while keeping the meaning intact

Task-5:

(Documentation – Review and Correct Docstrings)

- Task: Use AI to identify and correct inaccuracies in existing docstrings.
- Instructions:
 - o Provide Python code with outdated or incorrect docstrings.
 - o Instruct AI to rewrite each docstring to match the current code behavior.
 - o Ensure corrections follow Google-style formatting.
- Expected Output #5:
 - o Python file with updated, accurate, and standardized docstrings

Prompt:

Identify and correct inaccuracies in existing docstrings.

Code:

The screenshot shows the VS Code editor with a file named `TASK 5-1(9.2).py` open. The code defines a function `square` that takes an integer `num` and returns its cube. The function has a docstring with a description, arguments, and return values. The terminal window at the bottom shows the following commands and output:

```
Microsoft Windows [Version 10.0.26100.5074]
(c) Microsoft Corporation. All rights reserved.

C:\Users\nalla\OneDrive\Desktop\AI assisted coding>C:\Users\nalla\anaconda3\Scripts\activate

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding>conda activate base

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding>cmd /C "c:\Users\nalla\anaconda3\python.exe c:\Users\nalla\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundle\libs\debugpy\launcher 62271 -- "C:\Users\nalla\OneDrive\Desktop\AI assisted coding\TASK 5-1(9.2).py" "
```

The screenshot shows the VS Code editor with a file named `task5-2(9.2).py` open. The code defines a function `square` that takes an integer `num` and returns its square. The function has a docstring with a description, arguments, return values, and an example. The terminal window at the bottom shows the following commands and output:

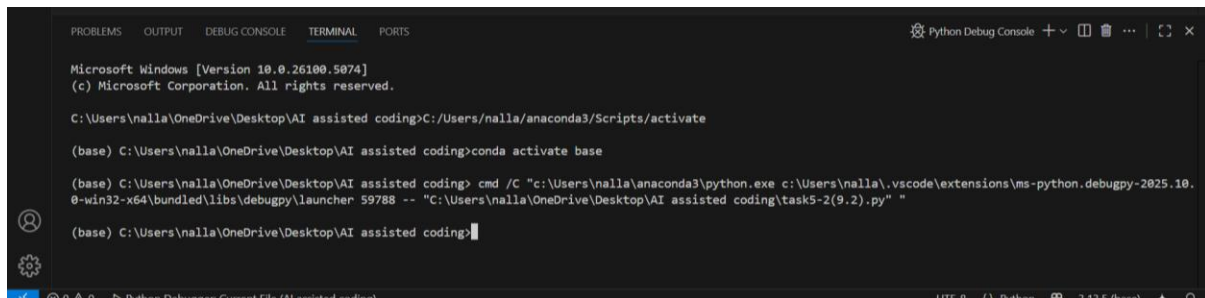
```
Microsoft Windows [Version 10.0.26100.5074]
(c) Microsoft Corporation. All rights reserved.

C:\Users\nalla\OneDrive\Desktop\AI assisted coding>C:\Users\nalla\anaconda3\Scripts\activate

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding>conda activate base

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding>cmd /C "c:\Users\nalla\anaconda3\python.exe c:\Users\nalla\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundle\libs\debugpy\launcher 59788 -- "C:\Users\nalla\OneDrive\Desktop\AI assisted coding\task5-2(9.2).py" "
```

OUTPUT:



```
Microsoft Windows [Version 10.0.26100.5874]
(c) Microsoft Corporation. All rights reserved.

C:\Users\nalla\OneDrive\Desktop\AI assisted coding>C:\Users\nalla\anaconda3\Scripts\activate

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding>conda activate base

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding>cmd /C "c:\Users\nalla\anaconda3\python.exe c:\Users\nalla\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundle\libs\debugpy\launcher 59788 -- "C:\Users\nalla\OneDrive\Desktop\AI assisted coding\task5-2(9.2).py" "
```

OBSERVATION:

The main issue is docstring drift—the code changes but the documentation doesn’t. Correcting the docstrings to Google style makes the functions clearer, accurate, and easier to maintain

Task-6:

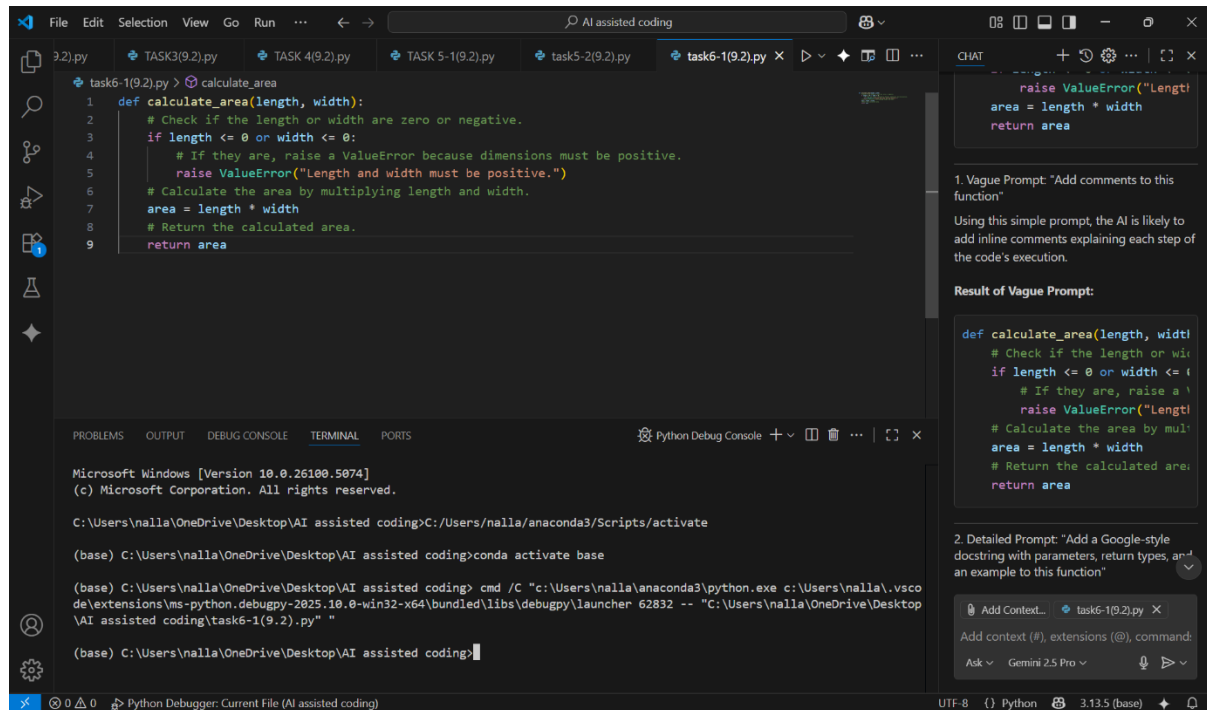
(Documentation – Prompt Comparison Experiment)

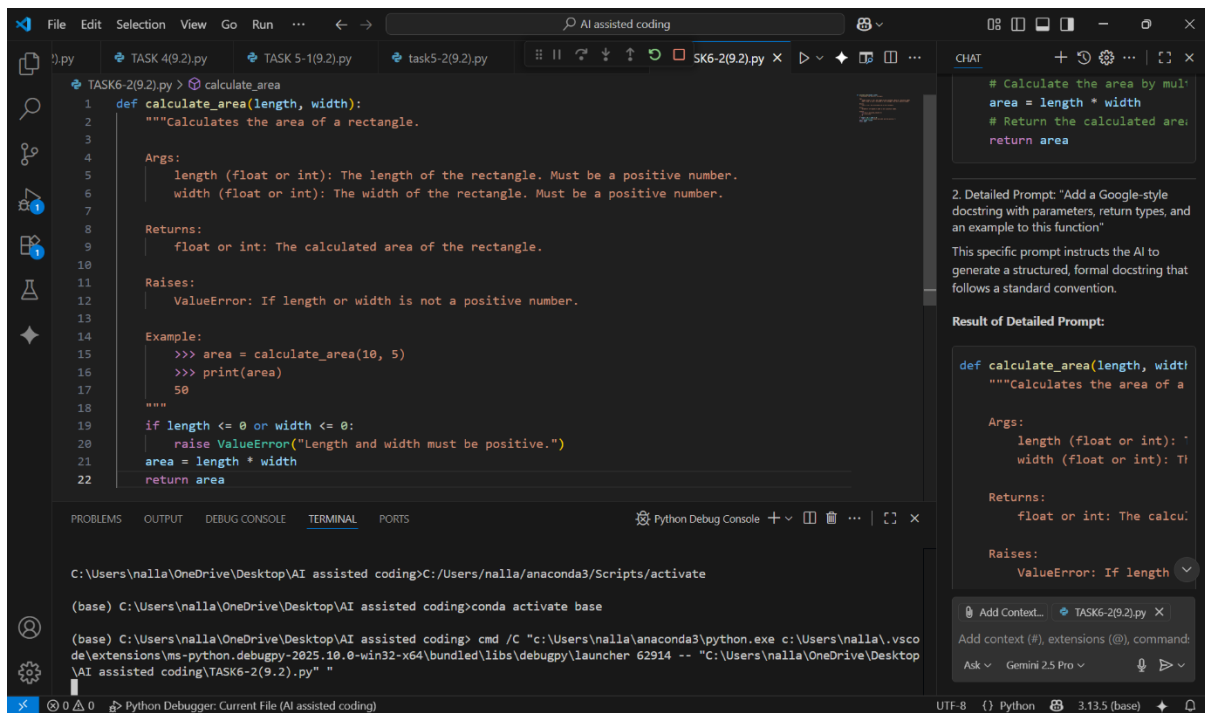
- Task: Compare documentation output from a vague prompt and a detailed prompt for the same Python function.
- Instructions:
 - o Create two prompts: one simple (“Add comments to this function”) and one detailed (“Add Google-style docstrings with parameters, return types, and examples”).
 - o Use AI to process the same Python function with both prompts.
 - o Analyze and record differences in quality, accuracy, and completeness.
- Expected Output #6:
 - o A comparison table showing the results from both prompts with observations

Prompt:

Compare documentation output from a vague prompt and a detailed prompt for the same Python function. Create two prompts: one simple (“Add comments to this function”) and one detailed (“Add Google-style docstrings with parameters, return types, and examples”).

Code:





Observation:

A detailed and specific prompt yields a vastly superior documentation result. It moves beyond simple line-by-line explanations to create structured, comprehensive, and professional documentation that significantly improves code maintainability and usability.