AI ASSISTED CODING

LAB– 9.2

NAME:M.Suchith

ENROLL.NO:2403A52068

BATCH:04

Task-1:

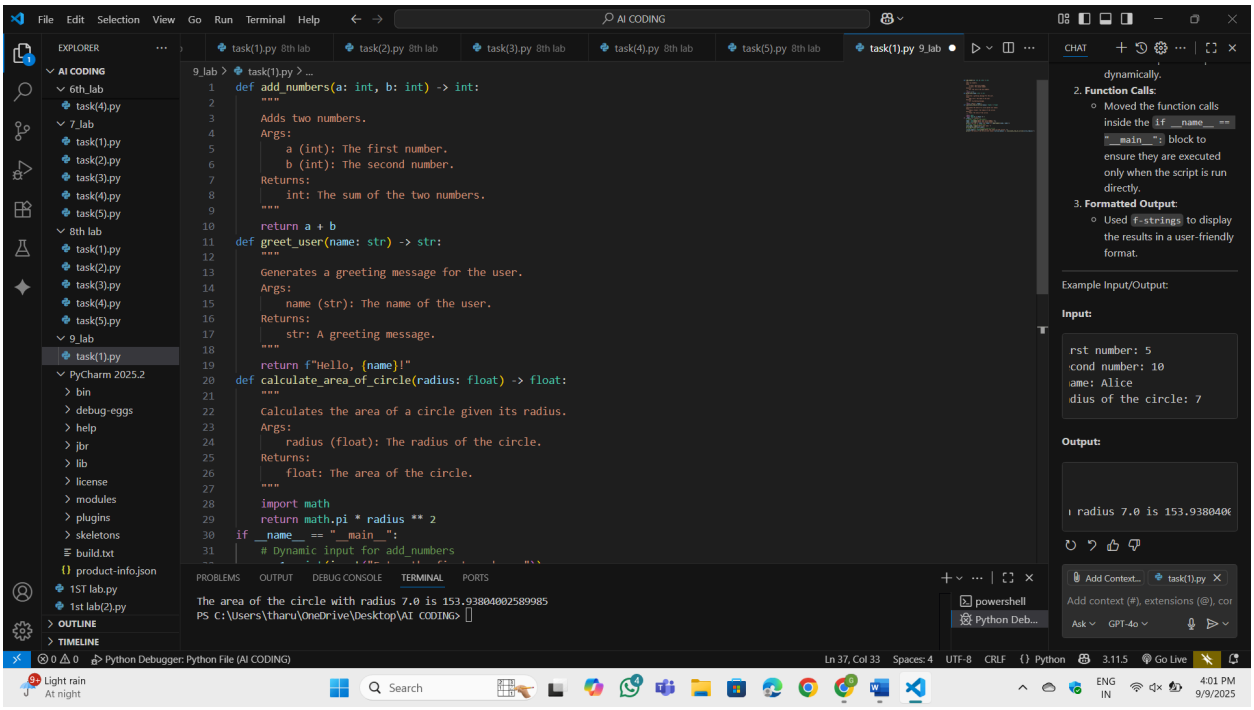
(Documentation – Google-Style Docstrings for
Python Functions)
• Task: Use AI to add Google-style docstrings to all functions in a given Python script.
• Instructions:
o Prompt AI to generate docstrings without providing any input-output examples.
o Ensure each docstring includes:
▪ Function description
▪ 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

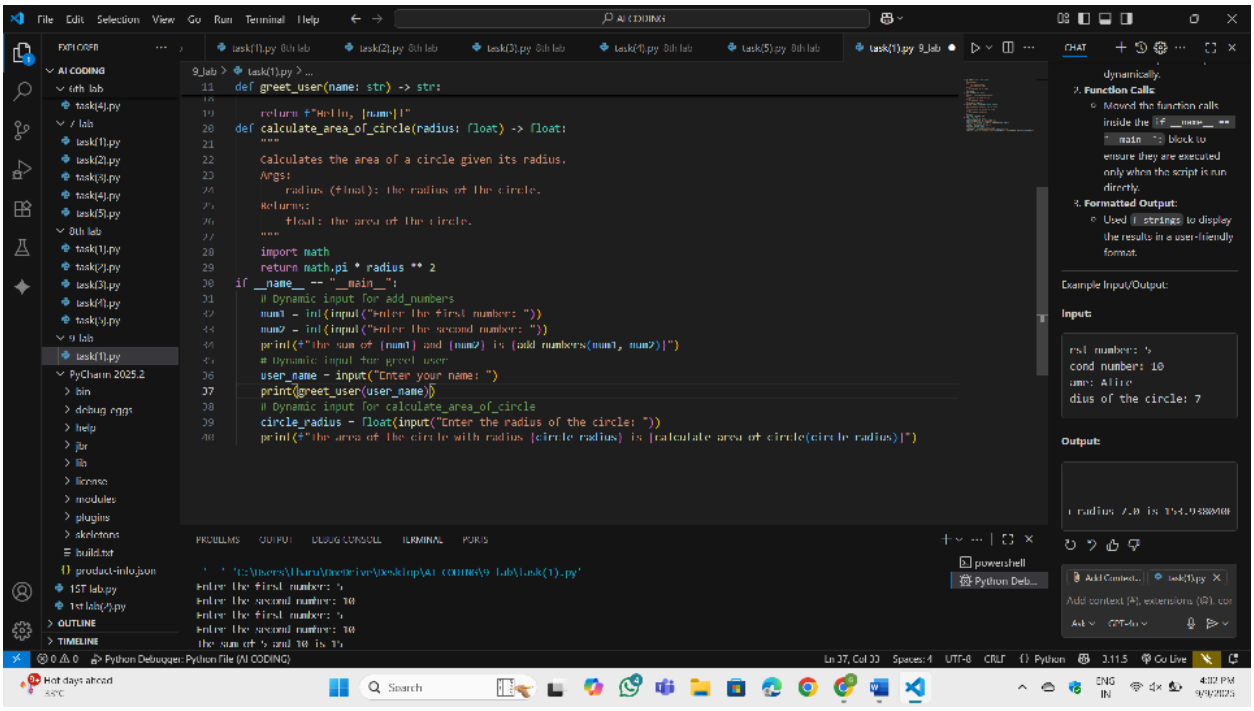
Prompt:

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

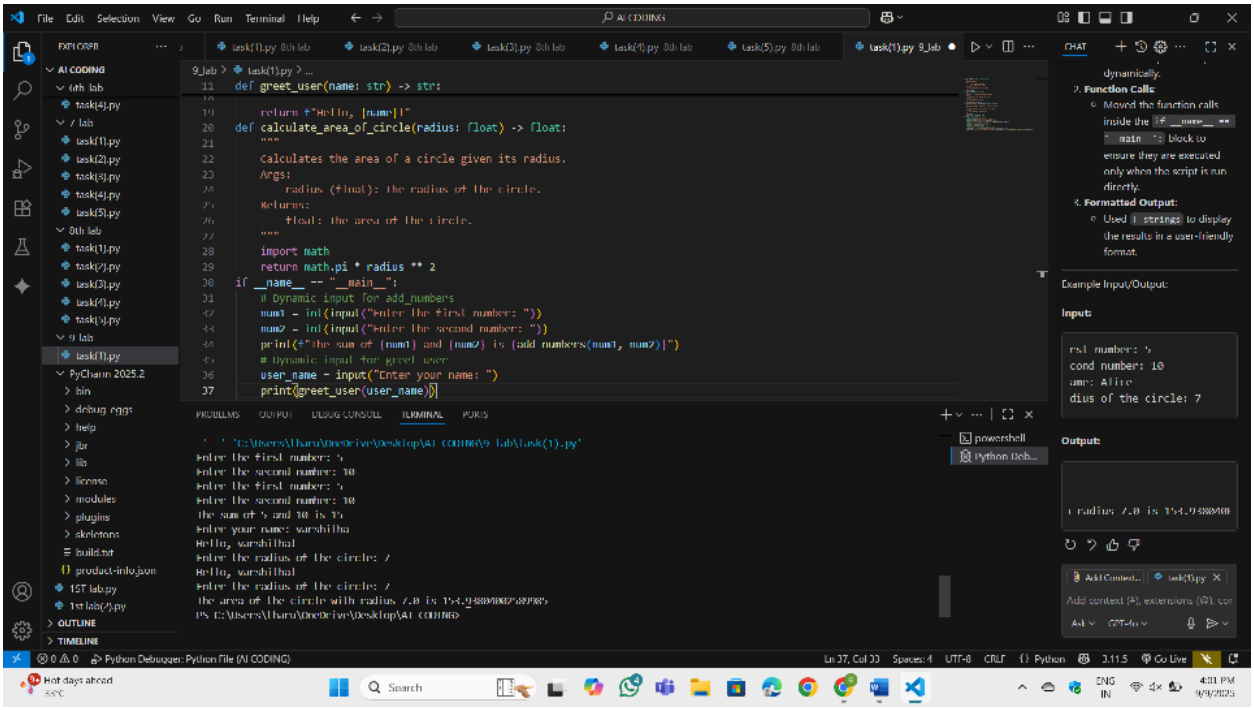
Ensure each docstring includes: Function description, Parameters with type hints, Return values with type hints

Code:




def greet_user(name: str) -> str:
return f"Hello, {name}!"
def calculate_area_of_circle(radius: float) -> float:
"""
Calculates the area of a circle given its radius.
Args:
radius (float): The radius of the circle.
Returns:
float: The area of the circle.
"""
import math
return math.pi * radius ** 2
if __name__ == "__main__":
# Dynamic input for add_numbers
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))
print(f"The sum of {num1} and {num2} is {add_numbers(num1, num2)}")
# Dynamic input for greet_user
user_name = input("Enter your name: ")
print(greet_user(user_name))
# Dynamic input for calculate_area_of_circle
circle_radius = float(input("Enter the radius of the circle: "))
print(f"The area of the circle with radius {circle_radius} is {calculate_area_of_circle(circle_radius)}")
Enter the first number: 5
Enter the second number: 10
Enter the first number: 5
Enter the second number: 10
The sum of 5 and 10 is 15

OUTPUT:


def greet_user(name: str) -> str:
return f"Hello, {name}!"
def calculate_area_of_circle(radius: float) -> float:
"""
Calculates the area of a circle given its radius.
Args:
radius (float): The radius of the circle.
Returns:
float: The area of the circle.
"""
import math
return math.pi * radius ** 2
if __name__ == "__main__":
# Dynamic input for add_numbers
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))
print(f"The sum of {num1} and {num2} is {add_numbers(num1, num2)}")
# Dynamic input for greet_user
user_name = input("Enter your name: ")
print(greet_user(user_name))
Enter the first number: 5
Enter the second number: 10
Enter the first number: 5
Enter the second number: 10
The sum of 5 and 10 is 15
Enter your name: varshitha
Hello, varshitha!
Enter the radius of the circle: 7
Hello, varshitha!
Enter the radius of the circle: 7
The area of the circle with radius 7.0 is 153.93804002589985
PS C:\Users\Tharu\OneDrive\Desktop\AI CODING>

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.

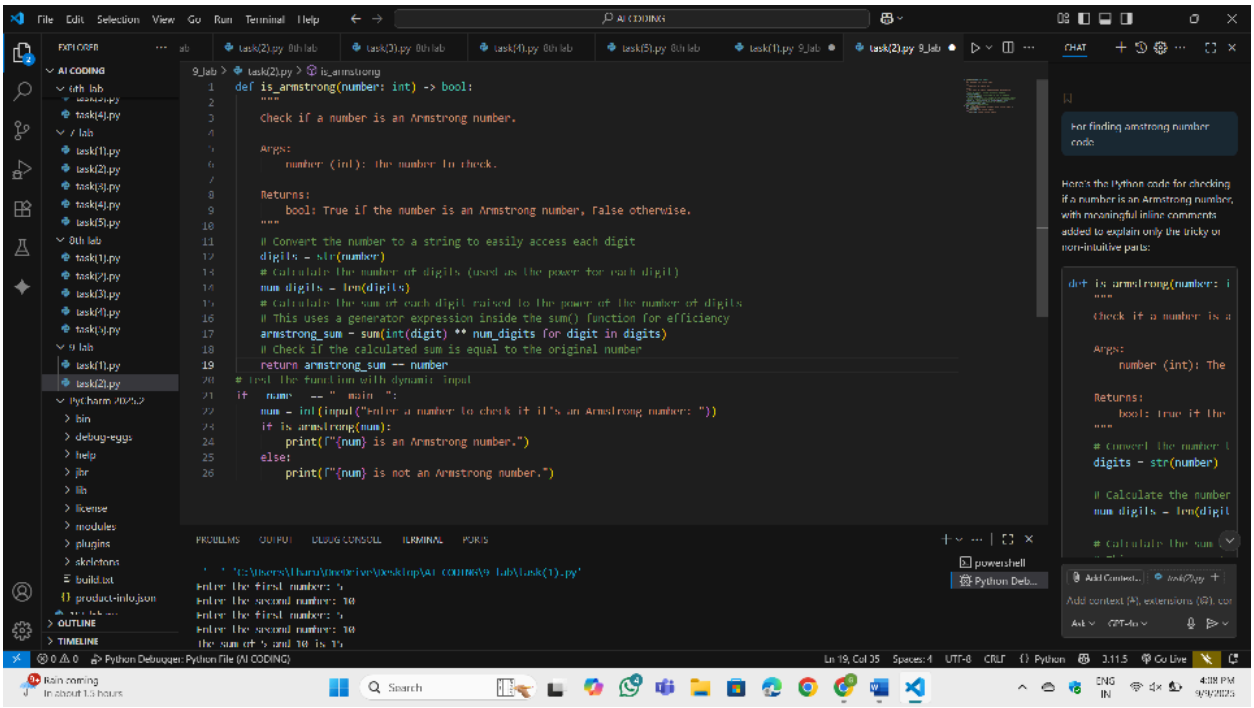
Task-2:

(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

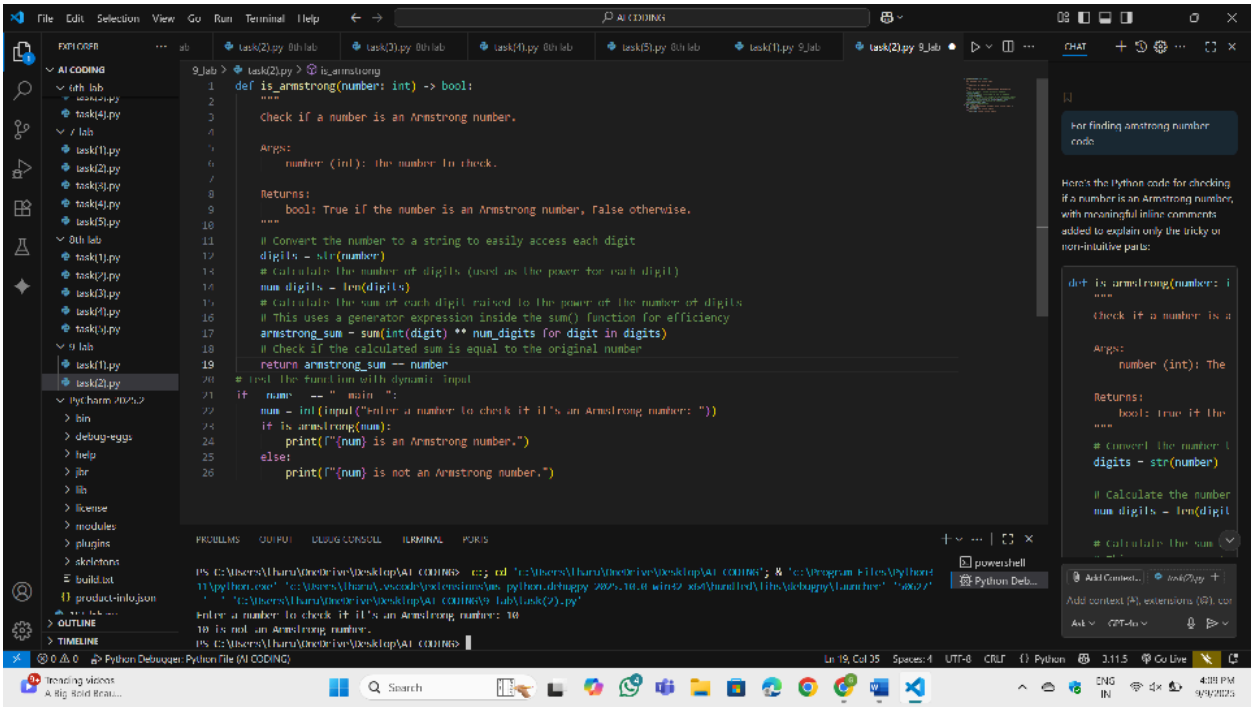
Prompt :

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:


def is_armstrong(number: int) -> bool:
"""
Check if a number is an Armstrong number.
Args:
number (int): the number to check.
Returns:
bool: True if the number is an Armstrong number, False otherwise.
"""
# Convert the number to a string to easily access each digit
digits = str(number)
# Calculate the number of digits (used as the power for each digit)
num_digits = len(digits)
# Calculate the sum of each digit raised to the power of the number of digits
# This uses a generator expression inside the sum() function for efficiency
armstrong_sum = sum(int(digit) ** num_digits for digit in digits)
# Check if the calculated sum is equal to the original number
return armstrong_sum == number
# Test the function with dynamic input
if __name__ == "__main__":
num = int(input("Enter a number to check if it's an Armstrong number: "))
if is_armstrong(num):
print(f"{num} is an Armstrong number.")
else:
print(f"{num} is not an Armstrong number.")
Enter the first number: 5
Enter the second number: 10
Enter the first number: 5
Enter the second number: 10
The sum of 5 and 10 is 15

OUTPUT:


def is_armstrong(number: int) -> bool:
"""
Check if a number is an Armstrong number.
Args:
number (int): the number to check.
Returns:
bool: True if the number is an Armstrong number, False otherwise.
"""
# Convert the number to a string to easily access each digit
digits = str(number)
# Calculate the number of digits (used as the power for each digit)
num_digits = len(digits)
# Calculate the sum of each digit raised to the power of the number of digits
# This uses a generator expression inside the sum() function for efficiency
armstrong_sum = sum(int(digit) ** num_digits for digit in digits)
# Check if the calculated sum is equal to the original number
return armstrong_sum == number
# Test the function with dynamic input
if __name__ == "__main__":
num = int(input("Enter a number to check if it's an Armstrong number: "))
if is_armstrong(num):
print(f"{num} is an Armstrong number.")
else:
print(f"{num} is not an Armstrong number.")
Enter a number to check if it's an Armstrong number: 10
10 is not an Armstrong number.

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:

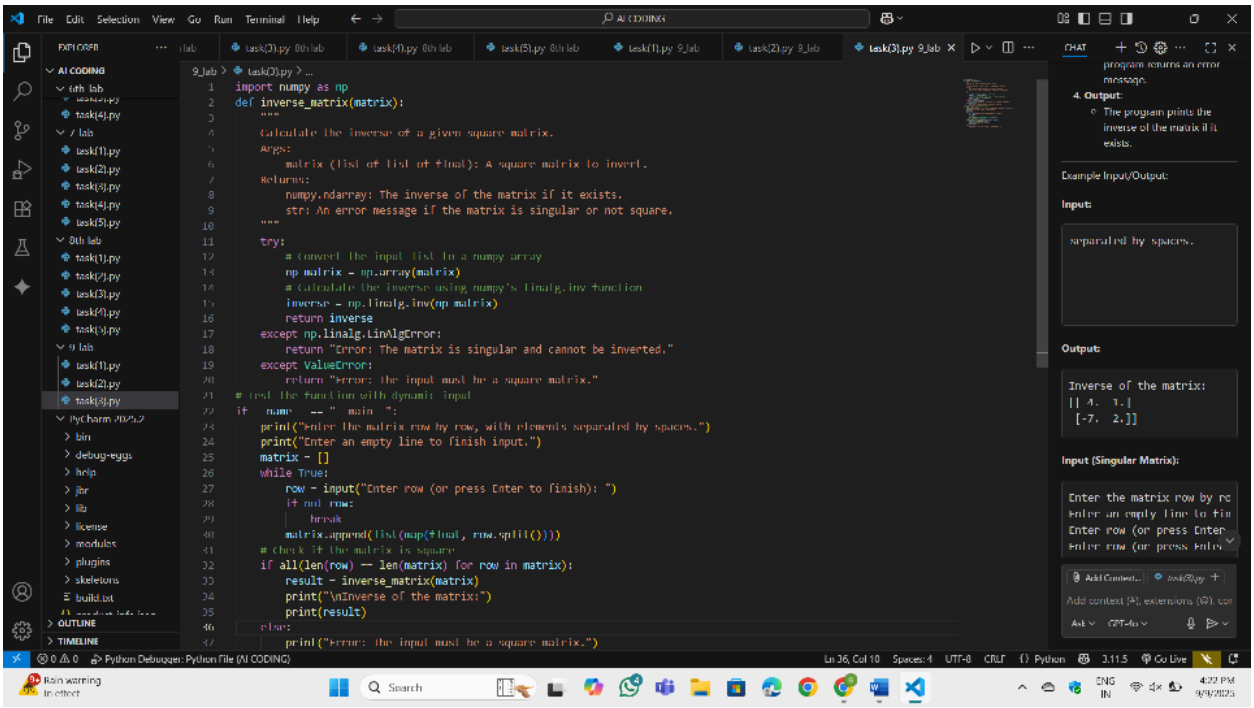
(Documentation – Module-Level Documentation)
• Task: Use AI to create a module-level docstring summarizing the purpose, dependencies, and main functions/classes of a Python file.

• 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

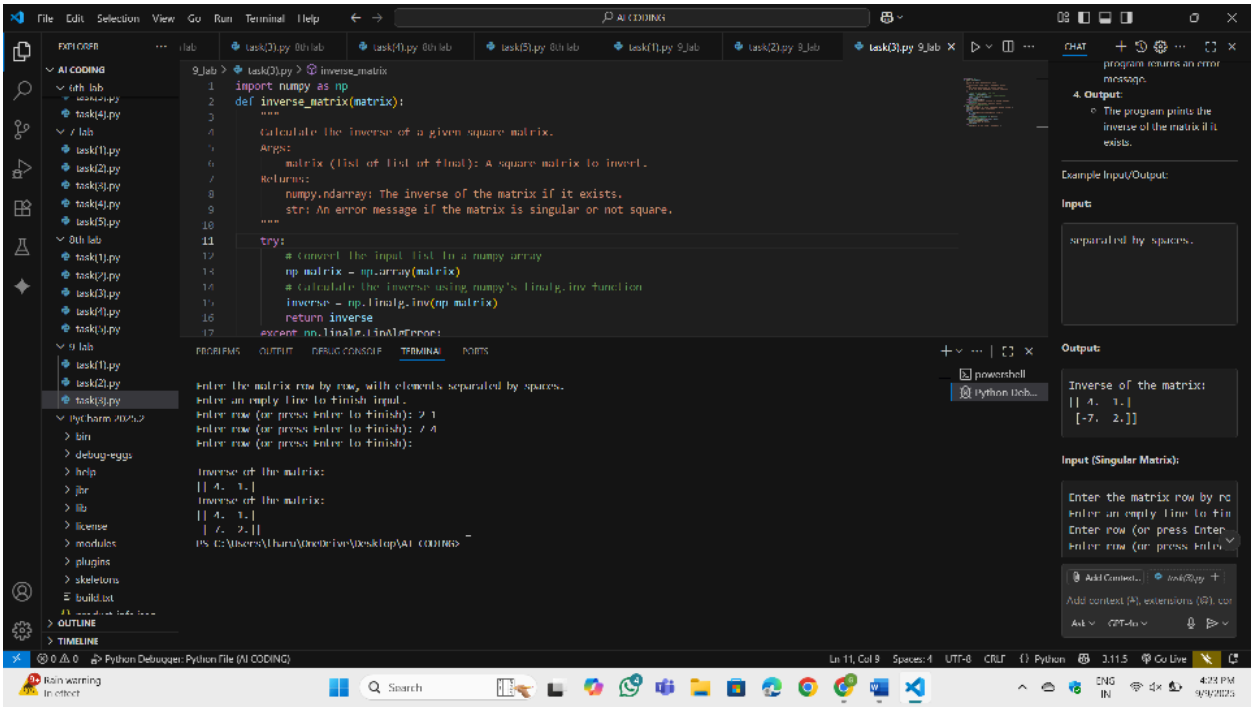
Prompt:

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

Code:


import numpy as np
def inverse_matrix(matrix):
"""
Calculate the inverse of a given square matrix.
Args:
matrix (list of list of float): A square matrix to invert.
Returns:
numpy.ndarray: The inverse of the matrix if it exists.
str: An error message if the matrix is singular or not square.
"""
try:
# Convert the input list to a numpy array
np_matrix = np.array(matrix)
# Calculate the inverse using numpy's linalg.inv function
inverse = np.linalg.inv(np_matrix)
return inverse
except np.linalg.LinAlgError:
return "Error: The matrix is singular and cannot be inverted."
except ValueError:
return "Error: The input must be a square matrix."
# Test the function with dynamic input
if __name__ == "__main__":
print("Enter the matrix row by row, with elements separated by spaces.")
print("Enter an empty line to finish input.")
matrix = []
while True:
row = input("Enter row (or press Enter to finish): ")
if not row:
break
matrix.append(list(map(float, row.split())))
# Check if the matrix is square
if all(len(row) == len(matrix) for row in matrix):
result = inverse_matrix(matrix)
print("\nInverse of the matrix:")
print(result)
else:
print("Error: The input must be a square matrix.")
Inverse of the matrix:
[[4. 1.]
[-7. 2.]]

OUTPUT:


Enter the matrix row by row, with elements separated by spaces.
Enter an empty line to finish input.
Enter row (or press Enter to finish): 2 1
Enter row (or press Enter to finish): 7 4
Enter row (or press Enter to finish):
Inverse of the matrix:
[[4. 1.]
[-7. 2.]]
Inverse of the matrix:
[[4. 1.]
[-7. 2.]]
PS C:\Users\Dharu\OneDrive\Desktop\AI CODING>

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.

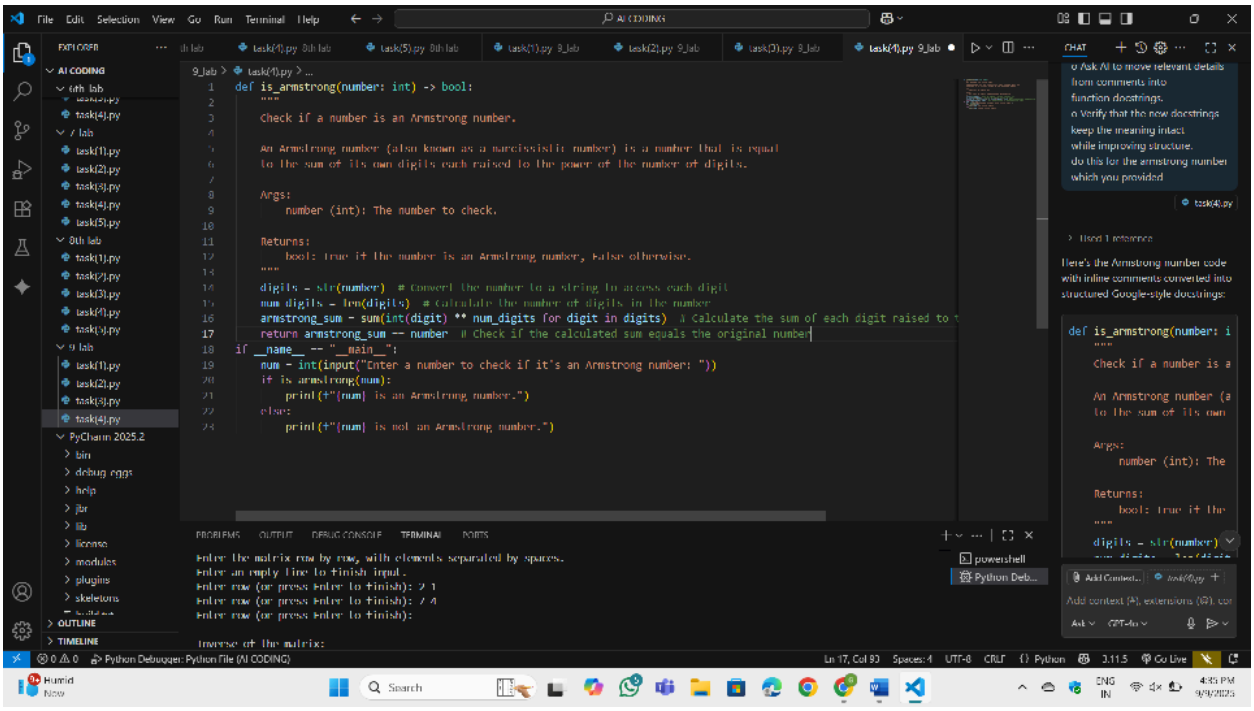
Task-4:

(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

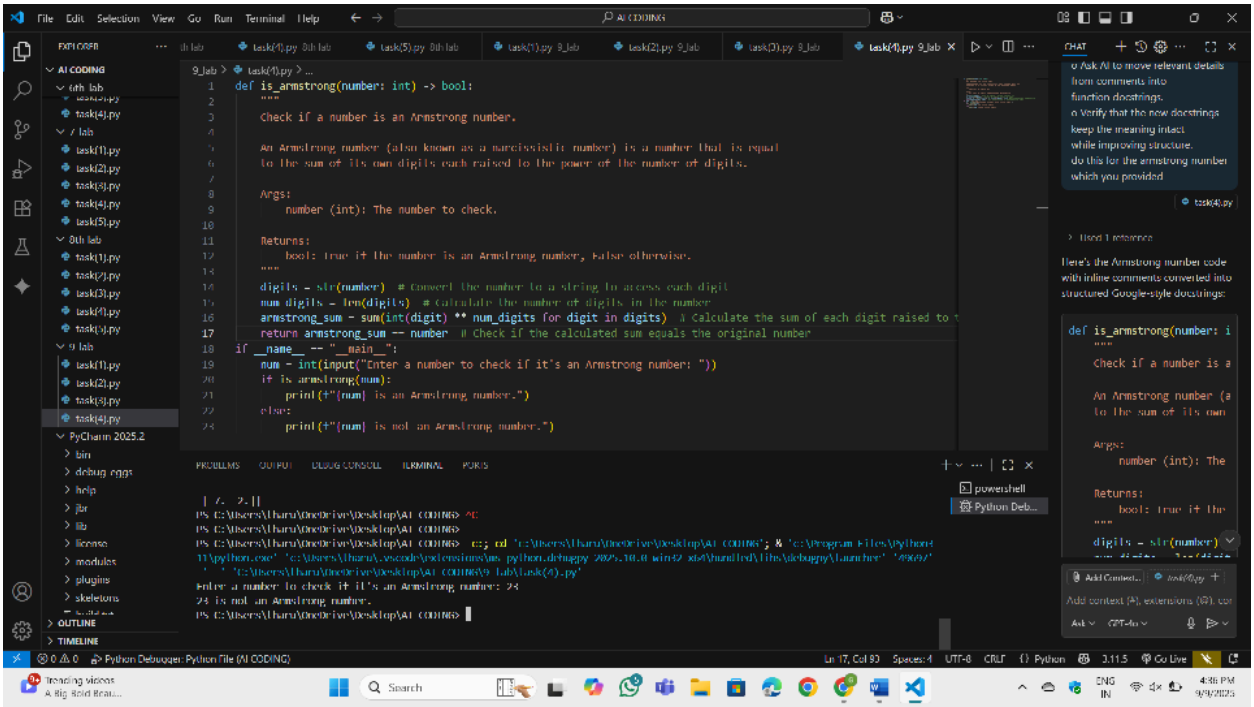
Prompt:

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

Code:


def is_armstrong(number: int) -> bool:
Check if a number is an Armstrong number.
An Armstrong number (also known as a narcissistic number) is a number that is equal
to the sum of its own digits each raised to the power of the number of digits.
Args:
number (int): The number to check.
Returns:
bool: True if the number is an Armstrong number, False otherwise.
digits = str(number) # Convert the number to a string to access each digit
num_digits = len(digits) # Calculate the number of digits in the number
armstrong_sum = sum(int(digit) ** num_digits for digit in digits) # Calculate the sum of each digit raised to
return armstrong_sum == number # Check if the calculated sum equals the original number
if __name__ == "__main__":
num = int(input("Enter a number to check if it's an Armstrong number: "))
if is_armstrong(num):
print(f"{num} is an Armstrong number.")
else:
print(f"{num} is not an Armstrong number.")
Enter the matrix row by row, with elements separated by spaces.
Enter an empty line to finish input.
Enter row (or press Enter to finish): 2 1
Enter row (or press Enter to finish): 7 4
Enter row (or press Enter to finish):
Inverse of the matrix:

OUTPUT:


def is_armstrong(number: int) -> bool:
Check if a number is an Armstrong number.
An Armstrong number (also known as a narcissistic number) is a number that is equal
to the sum of its own digits each raised to the power of the number of digits.
Args:
number (int): The number to check.
Returns:
bool: True if the number is an Armstrong number, False otherwise.
digits = str(number) # Convert the number to a string to access each digit
num_digits = len(digits) # Calculate the number of digits in the number
armstrong_sum = sum(int(digit) ** num_digits for digit in digits) # Calculate the sum of each digit raised to
return armstrong_sum == number # Check if the calculated sum equals the original number
if __name__ == "__main__":
num = int(input("Enter a number to check if it's an Armstrong number: "))
if is_armstrong(num):
print(f"{num} is an Armstrong number.")
else:
print(f"{num} is not an Armstrong number.")
Enter a number to check if it's an Armstrong number: 23
23 is not an Armstrong number.

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

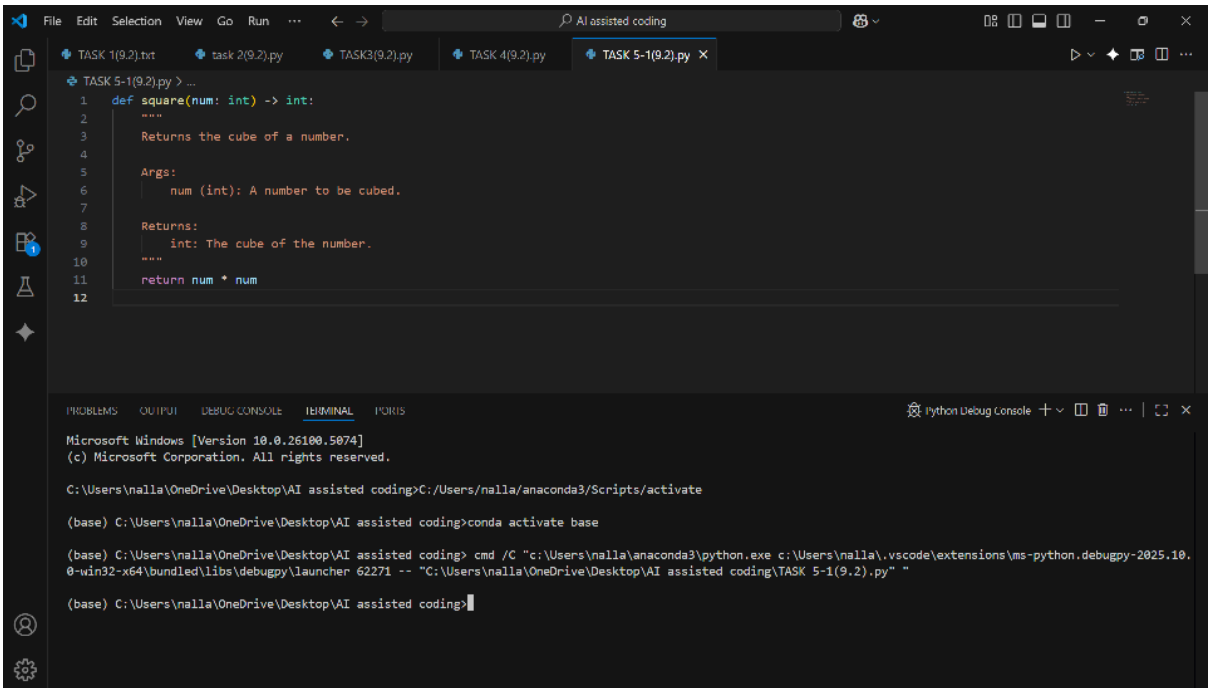
Task-5:

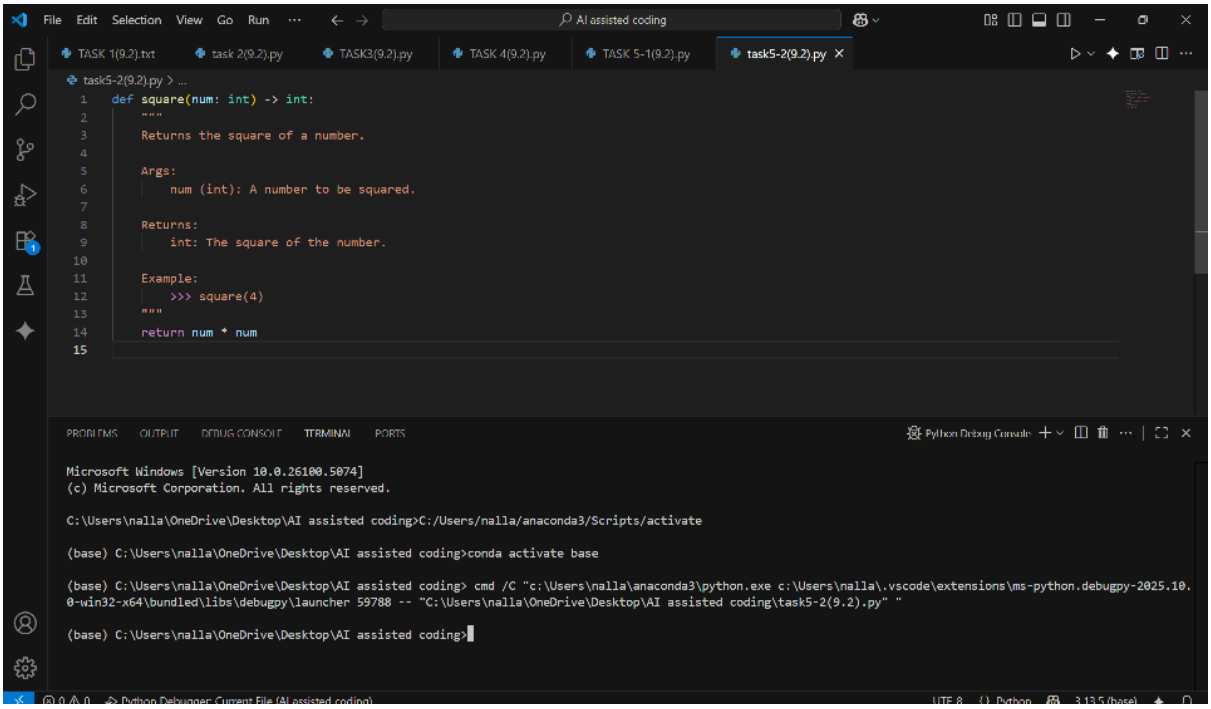
(Documentation – Review and Correct
Docstrings)
• Task: Use AI to identify and correct inaccuracies in existing docstrings.
• Instructions:
o Provide Python code with outdated or incorrect docstrings.
o Instruct AI to rewrite each docstring to match the current code behavior.
o Ensure corrections follow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

Prompt:

Identify and correct inaccuracies in existing docstrings.

Code:


File Edit Selection View Go Run
AI assisted coding
TASK 1(9.2).txt
task 2(9.2).py
TASK3(9.2).py
TASK 4(9.2).py
TASK 5-1(9.2).py
TASK 5-1(9.2).py > ...
def square(num: int) -> int:
"""
Returns the cube of a number.
Args:
num (int): A number to be cubed.
Returns:
int: The cube of the number.
"""
return num * num
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python Debug Console
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\bundled\libs\debugpy\launcher 62271 -- "C:\Users\nalla\OneDrive\Desktop\AI assisted coding\TASK 5-1(9.2).py" "
(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding>


File Edit Selection View Go Run
AI assisted coding
TASK 1(9.2).txt
task 2(9.2).py
TASK3(9.2).py
TASK 4(9.2).py
TASK 5-1(9.2).py
task5-2(9.2).py
task5-2(9.2).py > ...
def square(num: int) -> int:
"""
Returns the square of a number.
Args:
num (int): A number to be squared.
Returns:
int: The square of the number.
Example:
>>> square(4)
"""
return num * num
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python Debug Console
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\bundled\libs\debugpy\launcher 59788 -- "C:\Users\nalla\OneDrive\Desktop\AI assisted coding\task5-2(9.2).py" "
(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding>

OUTPUT:

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:

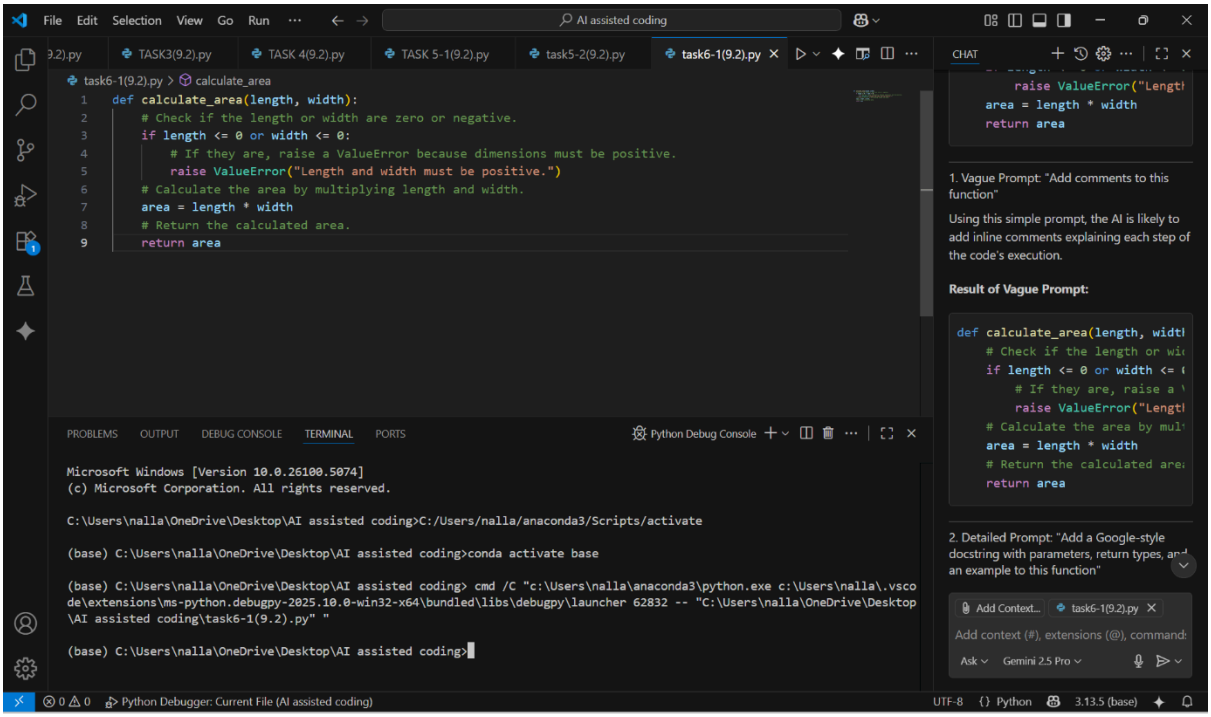
(Documentation – Prompt Comparison
Experiment)

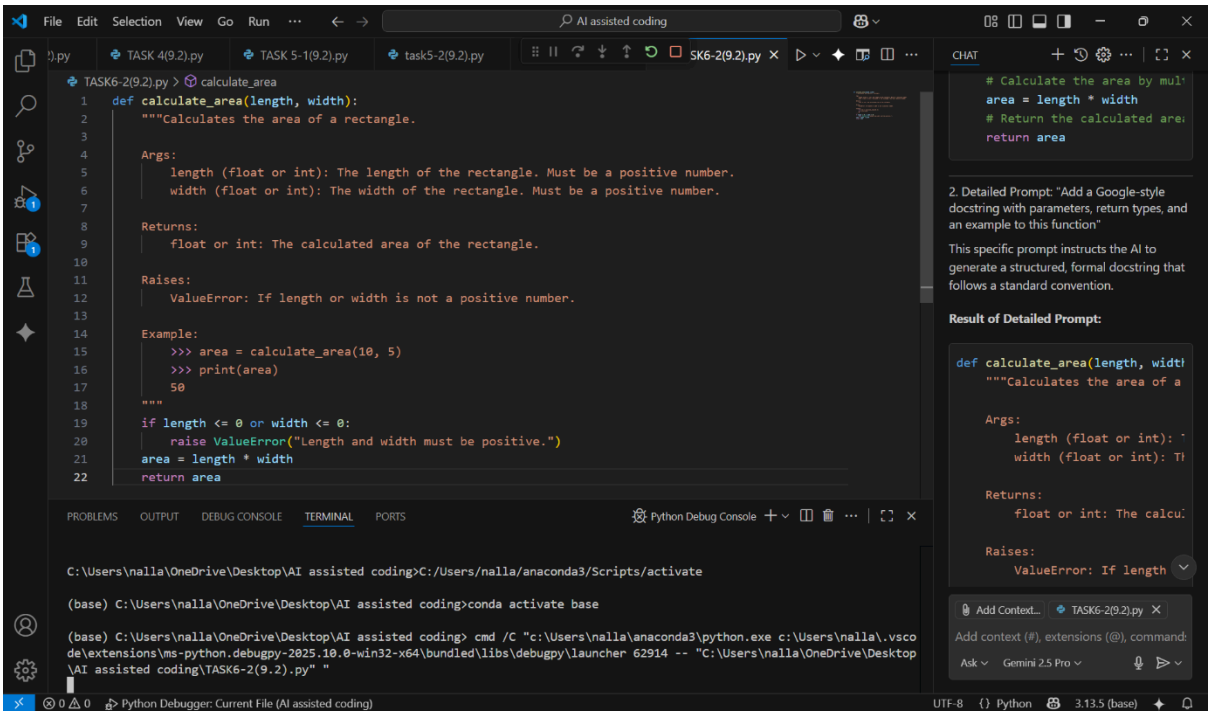
• Task: Compare documentation output from a vague prompt and a
detailed prompt for the same Python function.
• Instructions:
o Create two prompts: one simple ("Add comments to this function") and one detailed ("Add Google-style docstrings with parameters, return types, and examples").
o Use AI to process 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.