AIAC-9.3

Name: Vempati Siri

Htno no:2303A51309

Batch no:05

Task-1

Task 1: Basic Docstring Generation

Scenario

You are developing a utility function that processes numerical lists and must be properly documented for future maintenance.

Requirements

• Write a Python function to return the sum of even numbers and sum of odd numbers in a given list

• Manually add a Google Style docstring to the function

• Use an AI-assisted tool (Copilot / Cursor AI) to generate a function-level docstring

• Compare the AI-generated docstring with the manually written docstring

• Analyze clarity, correctness, and completeness

Expected Output

• Python function with manual Google-style docstring

• AI-generated docstring for the same function

• Comparison explaining differences between manual and AI-generated documentation

• Improved understanding of AI-generated function-level documentation

Code:

```
# Task 1: Basic Docstring Generation
def sum_even_odd_manual(numbers):
    """
    Calculates the sum of even and odd numbers in a list.
    Args:
        numbers (list of int): A list containing integer values.
    Returns:
        tuple: A tuple containing two integers:
            - Sum of even numbers
            - Sum of odd numbers

    Raises:
        TypeError: If the input is not a list of integers.
    Example:
        >>> sum_even_odd_manual([1, 2, 3, 4])
        (6, 4)
    """
    if not all(isinstance(num, int) for num in numbers):
        raise TypeError("All elements must be integers.")
    even_sum = sum(num for num in numbers if num % 2 == 0)
    odd_sum = sum(num for num in numbers if num % 2 != 0)
    return even_sum, odd_sum
#AI Docstring Function Output
def sum_even_odd_ai(numbers):
    """
    Returns the sum of even and odd numbers from the provided list.
    Parameters:
        numbers (list): List of integers.
    Returns:
        tuple: (even_sum, odd_sum)
    """
    even_sum = sum(num for num in numbers if num % 2 == 0)
    odd_sum = sum(num for num in numbers if num % 2 != 0)
    return even_sum, odd_sum
user_input = list(map(int, input("Enter numbers separated by space: ").split()))
```

```
print("\nManual Docstring Function Output:", sum_even_odd_manual(user_input))
print("AI Docstring Function Output:", sum_even_odd_ai(user_input))
```

# Comparison & Analysis

DOCSTRING COMPARISON

1. Clarity:

Manual docstring clearly explains arguments, return values, exceptions,
and includes an example. AI docstring is shorter but understandable.

2. Correctness:

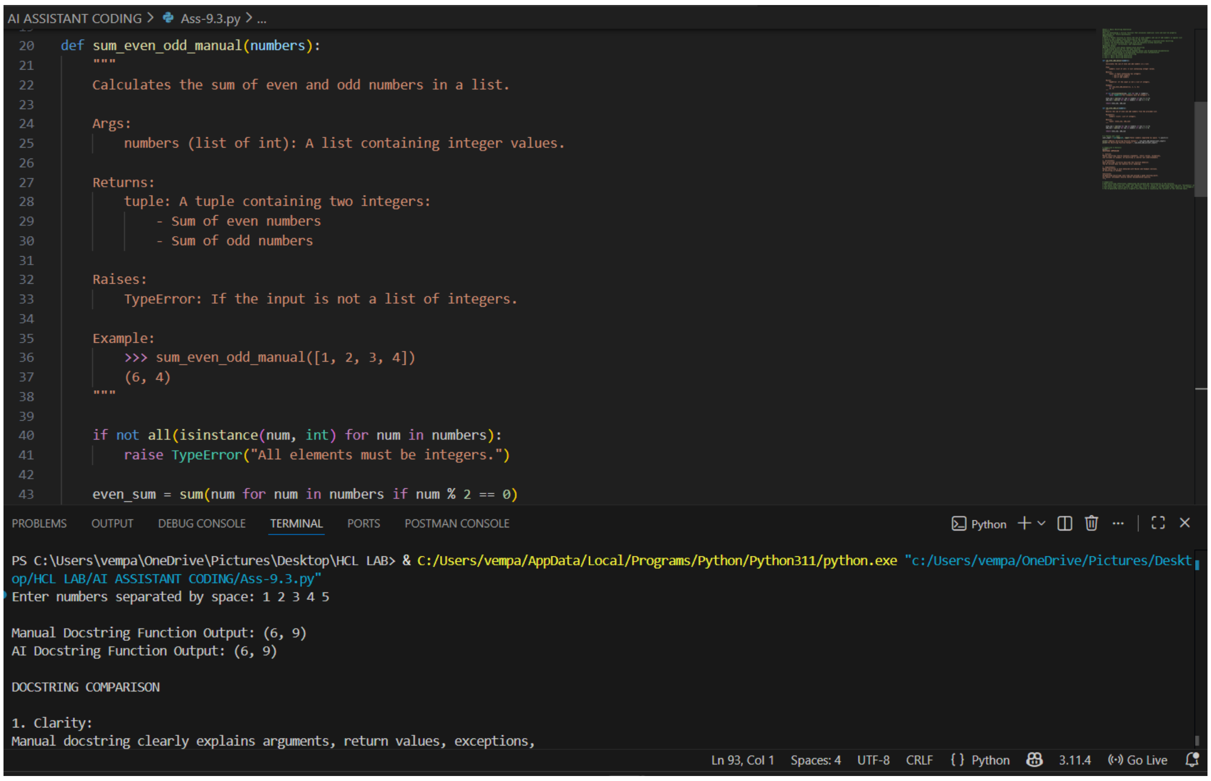
Both docstrings correctly describe the function behavior.
The AI version does not mention error handling.

3. Completeness:

Manual docstring is more detailed with Raises and Example sections.
AI docstring is minimal.

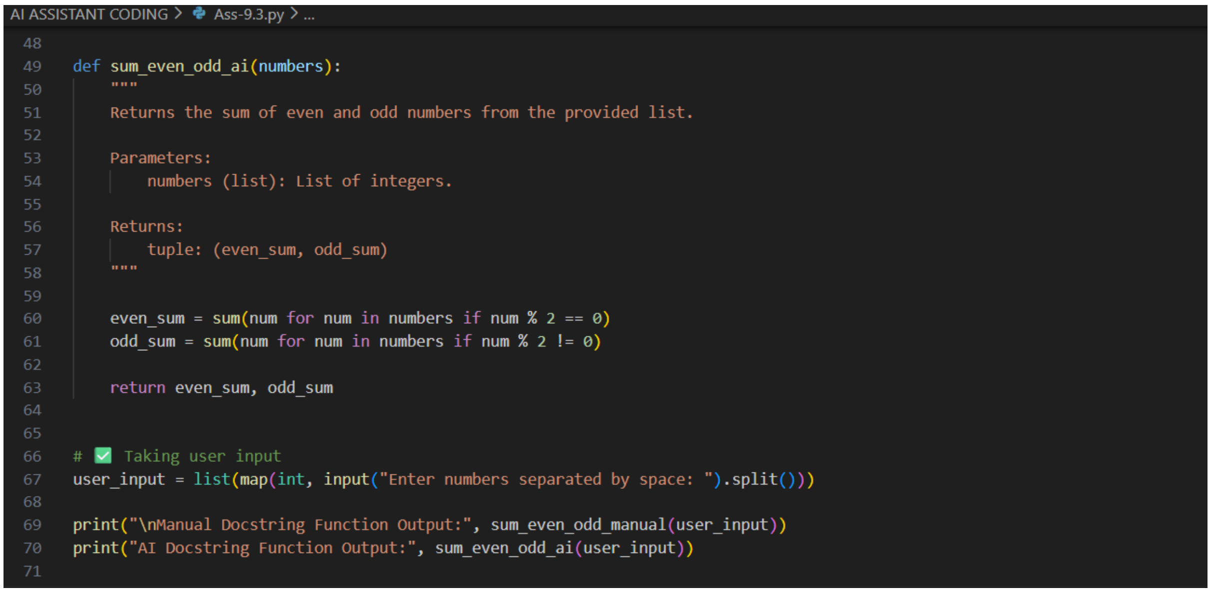
Conclusion:

AI-generated docstrings save time and provide a good starting point,
but manual refinement ensures better documentation quality.


AI ASSISTANT CODING > Ass-9.3.py > ...
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
Python
Ln 93, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.11.4 Go Live

```python
def sum_even_odd_manual(numbers):
    """
    Calculates the sum of even and odd numbers in a list.

    Args:
        numbers (list of int): A list containing integer values.

    Returns:
        tuple: A tuple containing two integers:
            - Sum of even numbers
            - Sum of odd numbers

    Raises:
        TypeError: If the input is not a list of integers.

    Example:
        >>> sum_even_odd_manual([1, 2, 3, 4])
        (6, 4)
    """

    if not all(isinstance(num, int) for num in numbers):
        raise TypeError("All elements must be integers.")

    even_sum = sum(num for num in numbers if num % 2 == 0)
```

```
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB> & C:/Users/vempa/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/vempa/OneDrive/Pictures/Desktop/HCL LAB/AI ASSISTANT CODING/Ass-9.3.py"
Enter numbers separated by space: 1 2 3 4 5

Manual Docstring Function Output: (6, 9)
AI Docstring Function Output: (6, 9)

DOCSTRING COMPARISON

1. Clarity:
Manual docstring clearly explains arguments, return values, exceptions,
```



```python

def sum_even_odd_ai(numbers):
    """
    Returns the sum of even and odd numbers from the provided list.

    Parameters:
        numbers (list): List of integers.

    Returns:
        tuple: (even_sum, odd_sum)
    """

    even_sum = sum(num for num in numbers if num % 2 == 0)
    odd_sum = sum(num for num in numbers if num % 2 != 0)

    return even_sum, odd_sum


# ✅ Taking user input
user_input = list(map(int, input("Enter numbers separated by space: ").split()))

print("\nManual Docstring Function Output:", sum_even_odd_manual(user_input))
print("AI Docstring Function Output:", sum_even_odd_ai(user_input))

```

Task 2: Automatic Inline Comments

Scenario

You are developing a student management module that must be easy to understand for new developers.

Requirements

• Write a Python program for an sru_student class with the following:

– Attributes: name, roll_no, hostel_status

– Methods: fee_update() and display_details()

• Manually write inline comments for each line or logical block

• Use an AI-assisted tool to automatically add inline comments

• Compare manual comments with AI-generated comments

• Identify missing, redundant, or incorrect AI comments

Expected Output

• Python class with manually written inline comments

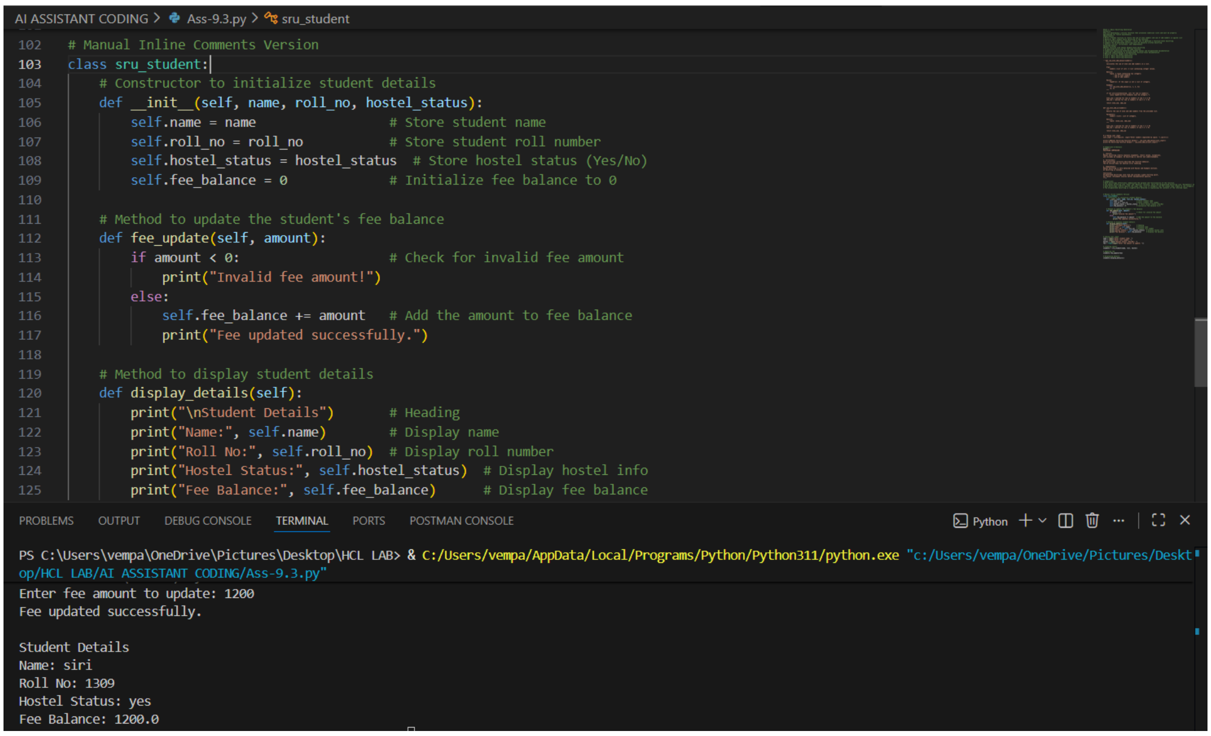
• AI-generated inline comments added to the same code

• Comparative analysis of manual vs AI comments

• Critical discussion on strengths and limitations of AI-generated comments

Code:

```
# Manual Inline Comments Version


class sru_student:
    # Constructor to initialize student details
    def __init__(self, name, roll_no, hostel_status):
        self.name = name                # Store student name
        self.roll_no = roll_no          # Store student roll number
        self.hostel_status = hostel_status  # Store hostel status (Yes/No)
        self.fee_balance = 0            # Initialize fee balance to 0
    # Method to update the student's fee balance
    def fee_update(self, amount):
        if amount < 0:              # Check for invalid fee amount
            print("Invalid fee amount!")
        else:
            self.fee_balance += amount   # Add the amount to fee balance
            print("Fee updated successfully.")
    # Method to display student details
    def display_details(self):
        print("\nStudent Details")       # Heading
        print("Name:", self.name)        # Display name
        print("Roll No:", self.roll_no)  # Display roll number
        print("Hostel Status:", self.hostel_status)  # Display hostel info
        print("Fee Balance:", self.fee_balance)      # Display fee balance
# Taking user input
name = input("Enter student name: ")
roll = input("Enter roll number: ")
hostel = input("Hostel Status (Yes/No): ")
fee = float(input("Enter fee amount to update: "))
# Creating object
student1 = sru_student(name, roll, hostel)

# Updating fee
student1.fee_update(fee)
# Displaying details
student1.display_details()
```

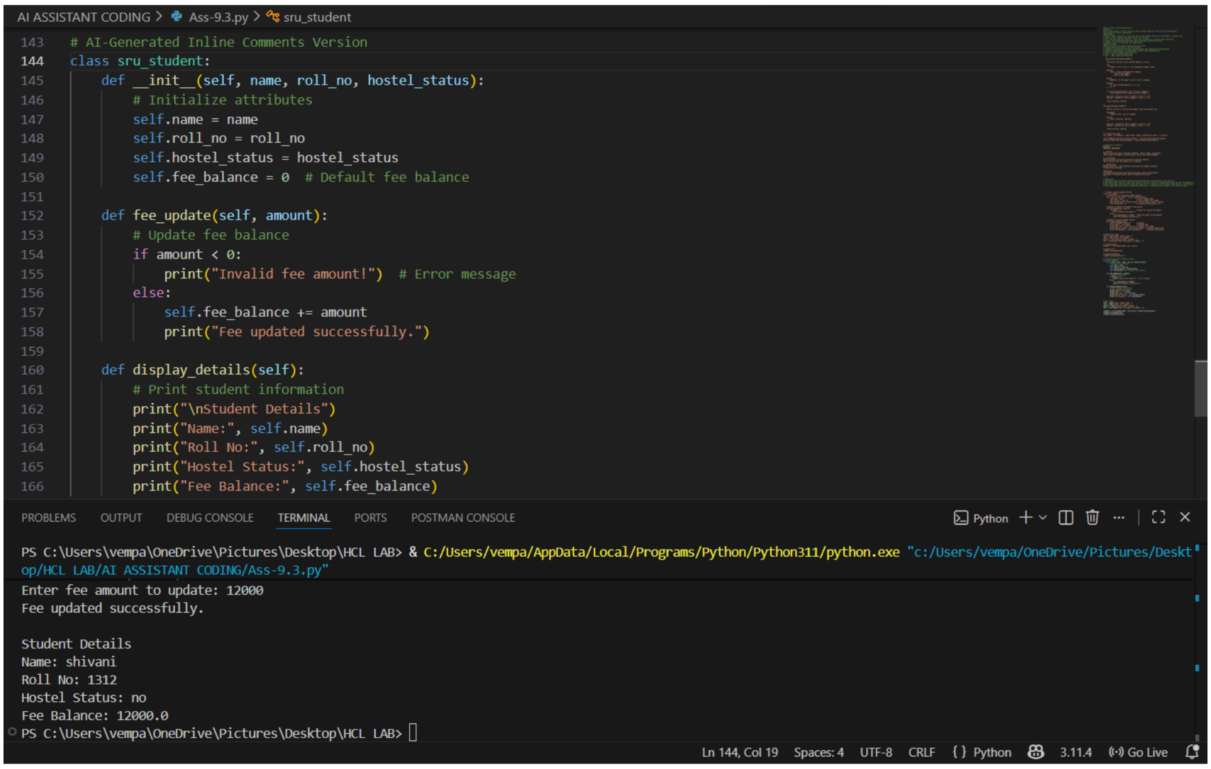


```
# Manual Inline Comments Version
class sru_student:
    # Constructor to initialize student details
    def __init__(self, name, roll_no, hostel_status):
        self.name = name                      # Store student name
        self.roll_no = roll_no                # Store student roll number
        self.hostel_status = hostel_status   # Store hostel status (Yes/No)
        self.fee_balance = 0                  # Initialize fee balance to 0

    # Method to update the student's fee balance
    def fee_update(self, amount):
        if amount < 0:                        # Check for invalid fee amount
            print("Invalid fee amount!")
        else:
            self.fee_balance += amount      # Add the amount to fee balance
            print("Fee updated successfully.")

    # Method to display student details
    def display_details(self):
        print("\nStudent Details")          # Heading
        print("Name:", self.name)           # Display name
        print("Roll No:", self.roll_no)   # Display roll number
        print("Hostel Status:", self.hostel_status)  # Display hostel info
        print("Fee Balance:", self.fee_balance)      # Display fee balance
```

```
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB> & C:/Users/vempa/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/vempa/OneDrive/Pictures/Deskt
op/HCL LAB/AI ASSISTANT CODING/Ass-9.3.py"
Enter fee amount to update: 1200
Fee updated successfully.

Student Details
Name: siri
Roll No: 1309
Hostel Status: yes
Fee Balance: 1200.0
```

```
# AI-Generated Inline Comments Version
class sru_student:
    def __init__(self, name, roll_no, hostel_status):
        # Initialize attributes
        self.name = name
        self.roll_no = roll_no
        self.hostel_status = hostel_status
        self.fee_balance = 0  # Default fee balance
    def fee_update(self, amount):
        # Update fee balance
        if amount < 0:
            print("Invalid fee amount!")  # Error message
        else:
            self.fee_balance += amount
            print("Fee updated successfully.")
    def display_details(self):
        # Print student information
        print("\nStudent Details")
        print("Name:", self.name)
        print("Roll No:", self.roll_no)
        print("Hostel Status:", self.hostel_status)
        print("Fee Balance:", self.fee_balance)
# User input
name = input("Enter student name: ")
roll = input("Enter roll number: ")
hostel = input("Hostel Status (Yes/No): ")
fee = float(input("Enter fee amount to update: "))
student1 = sru_student(name, roll_no=roll, hostel_status=hostel)
student1.fee_update(fee)
```

student1.display_details()


AI ASSISTANT CODING > Ass-9.3.py > sru_student
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
Python
Ln 144, Col 19 Spaces: 4 UTF-8 CRLF { } Python 3.11.4 Go Live

```python
# AI-Generated Inline Comments Version
class sru_student:
    def __init__(self, name, roll_no, hostel_status):
        # Initialize attributes
        self.name = name
        self.roll_no = roll_no
        self.hostel_status = hostel_status
        self.fee_balance = 0  # Default fee balance

    def fee_update(self, amount):
        # Update fee balance
        if amount < 0:
            print("Invalid fee amount!")  # Error message
        else:
            self.fee_balance += amount
            print("Fee updated successfully.")

    def display_details(self):
        # Print student information
        print("\nStudent Details")
        print("Name:", self.name)
        print("Roll No:", self.roll_no)
        print("Hostel Status:", self.hostel_status)
        print("Fee Balance:", self.fee_balance)
```

```
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB> & C:/Users/vempa/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/vempa/OneDrive/Pictures/Desktop/HCL LAB/AI ASSISTANT CODING/Ass-9.3.py"
Enter fee amount to update: 12000
Fee updated successfully.

Student Details
Name: shivani
Roll No: 1312
Hostel Status: no
Fee Balance: 12000.0
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB> 
```

Comparative Analysis of Manual vs AI Comments

- **Clarity:** Manual comments are more detailed and easier for beginners, while AI comments are shorter and more concise.
- **Completeness:** Manual comments explain logic and purpose better; AI may miss important details.
- **Accuracy:** Manual comments are usually more reliable since they reflect developer intent. AI comments can sometimes be too generic.
- **Redundancy:** Manual comments may over-explain obvious code, whereas AI avoids unnecessary comments.
- **Maintainability:** AI helps quickly regenerate comments when code changes.

Overall: Manual comments provide depth, while AI comments improve efficiency

Strengths of AI-Generated Comments

- Saves time and increases productivity
- Maintains consistent commenting style
- Useful for documenting large codebases
- Provides a quick starting point

Limitations of AI-Generated Comments

- May lack project context
- Can generate vague or generic explanations
- Might misinterpret complex logic
- Still requires human review

Conclusion: AI comments are best used as a starting point, but manual refinement ensures high-quality documentation.

Task-03

Module-Level and Function-Level Documentation

Scenario

You are building a small calculator module that will be shared across multiple projects and requires structured documentation.

Requirements

• Write a Python script containing 3–4 functions (e.g., add, subtract, multiply, divide)

• Manually write NumPy Style docstrings for each function

• Use AI assistance to generate:

– A module-level docstring

– Individual function-level docstrings

• Compare AI-generated docstrings with manually written ones

• Evaluate documentation structure, accuracy, and readability

Expected Output

• Python script with manual NumPy-style docstrings

• AI-generated module-level and function-level documentation

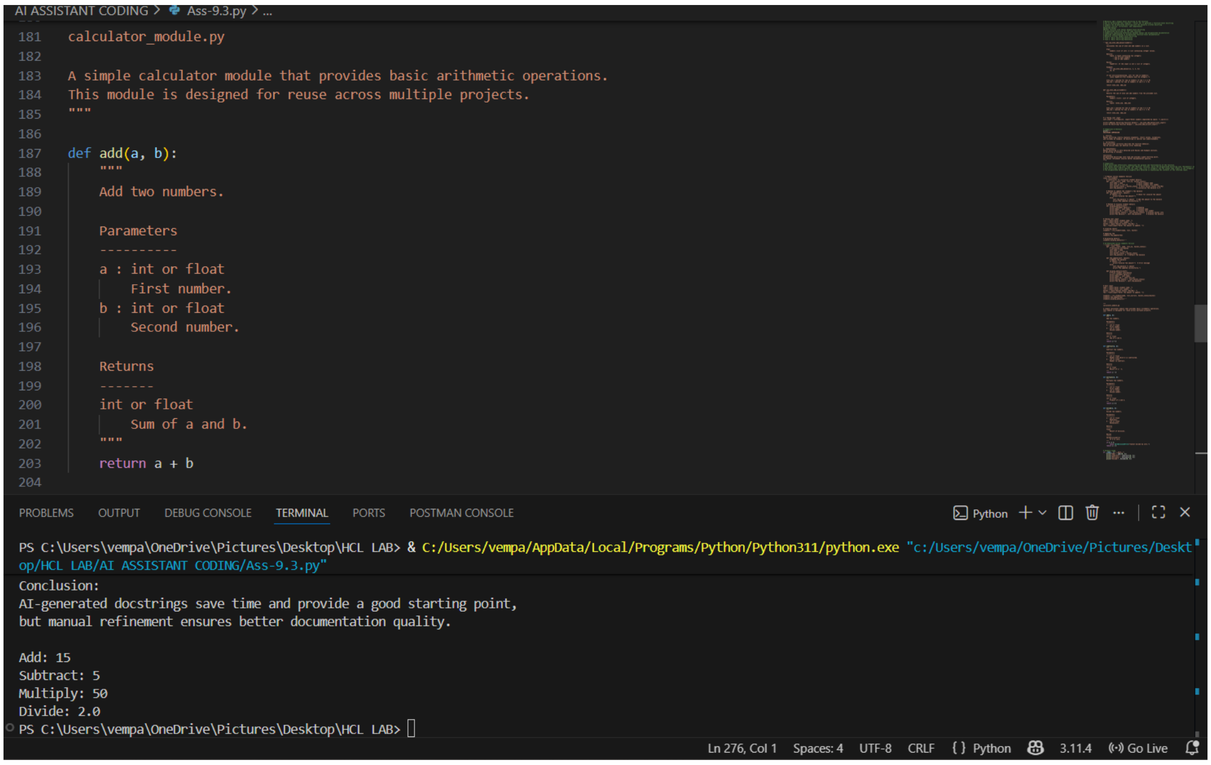
• Comparison between AI-generated and manual documentation

• Clear understanding of structured documentation for multi-function scripts

Code:

```
"""
calculator_module.py
A simple calculator module that provides basic arithmetic operations.
```

```
This module is designed for reuse across multiple projects.
"""
def add(a, b):
    """
    Add two numbers.
    Parameters
    ----------
    a : int or float
        First number.
    b : int or float
        Second number.
    Returns
    -------
    int or float
        Sum of a and b.
    """
    return a + b
def subtract(a, b):
    """
    Subtract two numbers.
    Parameters
    a : int or float
        Number from which b is subtracted.
    b : int or float
        Number to subtract.

    Returns
    int or float
        Result of a - b.
    """
    return a - b
def multiply(a, b):
    """
    Multiply two numbers.
    Parameters
    a : int or float
        First number.
    b : int or float
        Second number.
    Returns
    -------
    int or float
        Product of a and b.
    """
    return a * b
def divide(a, b):
    """
    Divide two numbers.
    Parameters
    ----------
    a : int or float
        Numerator.
    b : int or float
        Denominator.

    Returns
    float
        Result of division.
    Raises
    ZeroDivisionError
        If b is zero.
    """
    if b == 0:
        raise ZeroDivisionError("Cannot divide by zero.")
    return a / b
# Example usage
if __name__ == "__main__":
    print("Add:", add(10, 5))
    print("Subtract:", subtract(10, 5))
    print("Multiply:", multiply(10, 5))
    print("Divide:", divide(10, 5))
```


AI ASSISTANT CODING > Ass-9.3.py > ...
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
Python
Ln 276, Col 1 Spaces: 4 UTF-8 CRLF { } Python 3.11.4 Go Live

```
calculator_module.py

A simple calculator module that provides basic arithmetic operations.
This module is designed for reuse across multiple projects.
"""

def add(a, b):
    """
    Add two numbers.

    Parameters
    ----------
    a : int or float
        First number.
    b : int or float
        Second number.

    Returns
    -------
    int or float
        Sum of a and b.
    """
    return a + b

```

```
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB> & C:/Users/vempa/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/vempa/OneDrive/Pictures/Deskt
op/HCL LAB/AI ASSISTANT CODING/Ass-9.3.py"
Conclusion:
AI-generated docstrings save time and provide a good starting point,
but manual refinement ensures better documentation quality.

Add: 15
Subtract: 5
Multiply: 50
Divide: 2.0
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB> 
```

Calculator Module

This module provides basic arithmetic operations including addition,

subtraction, multiplication, and division. The functions are designed

to be simple, reusable, and suitable for integration into larger applications that require mathematical calculations.

AI-Generated Function-Level Documentation

```
def add(a, b):
    """Returns the sum of two numbers.
    Parameters:
        a (int or float): First number.
        b (int or float): Second number.
    Returns:
        int or float: Result of the addition.
    """
def subtract(a, b):
    """
    Returns the difference between two numbers.
    Parameters:
        a (int or float): Minuend.
        b (int or float): Subtrahend.
    Returns:
        int or float: Result of the subtraction.
    """
def multiply(a, b):
    """
    Returns the product of two numbers.
    Parameters:
        a (int or float): First value.
        b (int or float): Second value.
    Returns:
        int or float: Multiplication result.
    """
```

```
def divide(a, b):
    """
    Divides one number by another.
    Parameters:
        a (int or float): Numerator.
        b (int or float): Denominator.
    Returns:
        float: Division result.
    Raises:
        ZeroDivisionError: If the denominator is zero."""
```

Comparison: AI vs Manual Documentation

- **Structure:** Manual docstrings are more standardized (e.g., NumPy style).
- **Accuracy:** Manual documentation often includes edge cases; AI may miss them.
- **Readability:** AI is shorter and quicker to read.
- **Completeness:** Manual is more detailed.

Overall: AI is fast, but manual documentation ensures higher quality.

Understanding Structured Documentation

Structured documentation:

Improves readability
Helps new developers understand code quickly
Supports reuse across projects
Makes maintenance easier

Best Practice: Generate docstrings with AI, then refine them manually for clarity and correctness.