AI ASSISTED CODING

Lab Assignment-9.4

Name : T.Shylasri

H.T.NO:2303A51876

Batch-14(LAB-9)

Date:11-02-2026

ASSIGNMENT-9.4

Lab 9 – Documentation Generation: Automatic Documentation and Code Comments.

Lab Objectives

• To use AI-assisted coding tools for generating Python documentation and code comments.

• To apply zero-shot, few-shot, and context-based prompt engineering for documentation creation.

• To practice generating and refining docstrings, inline comments, and module-level documentation.

• To compare outputs from different prompting styles for quality analysis.

Lab Outcomes

• Generate structured code documentation using AI tools

• Apply appropriate documentation styles to different code contexts

• Improve code readability through selective commenting

• Convert informal developer comments into professional documentation

• Analyze and refine AI-generated documentation

Task 1: Auto-Generating Function Documentation in a Shared Codebase

Scenario

You have joined a development team where several utility functions are already implemented, but the code lacks proper documentation. New team members are struggling to understand how these functions should be used.

Task Description

You are given a Python script containing multiple functions without any docstrings.

Using an AI-assisted coding tool:

• Ask the AI to automatically generate Google-style function docstrings for each function

• Each docstring should include:

- A brief description of the function
- Parameters with data types
- Return values
- At least one example usage (if applicable)

Experiment with different prompting styles (zero-shot or context-based) to observe quality differences.

Expected Outcome

• A Python script with well-structured Google-style docstrings

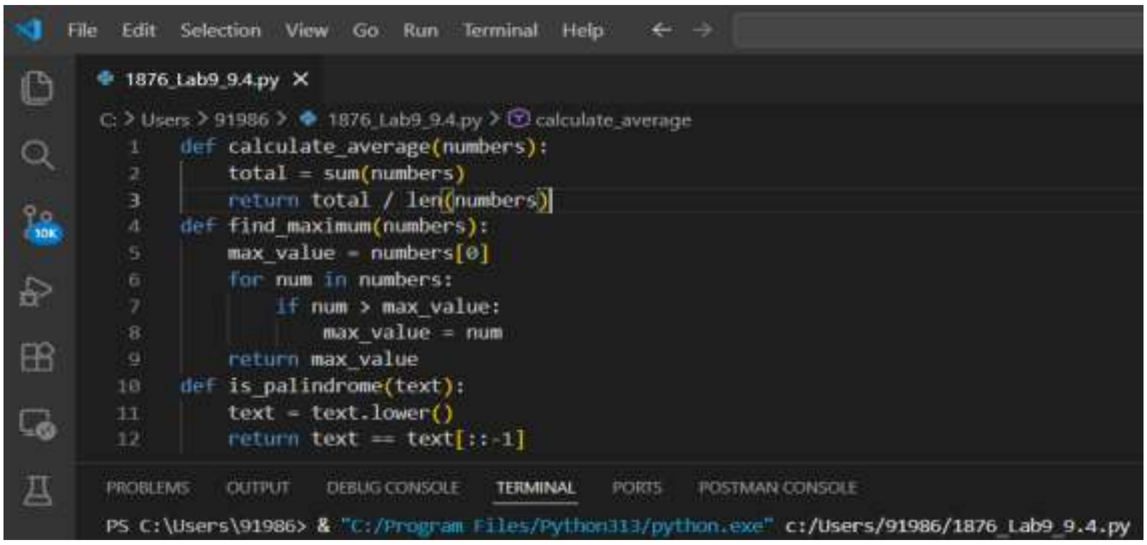
• Docstrings that clearly explain function behavior and usage

• Improved readability and usability of the codebase

Objective

To use AI-assisted coding tools to automatically generate structured Google-style docstrings for existing Python utility functions and compare zero-shot and context-based prompting styles.

Undocumented Code (UN DOC – Before AI)


File Edit Selection View Go Run Terminal Help
1876_Lab9_9.4.py
C: > Users > 91986 > 1876_Lab9_9.4.py > calculate_average
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```python
def calculate_average(numbers):
    total = sum(numbers)
    return total / len(numbers)
def find_maximum(numbers):
    max_value = numbers[0]
    for num in numbers:
        if num > max_value:
            max_value = num
    return max_value
def is_palindrome(text):
    text = text.lower()
    return text == text[::-1]
```

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/1876_Lab9_9.4.py
```

Zero-Shot Prompt Used

#Generate Google-style docstrings for the following Python functions.

#Include:

#- A brief description

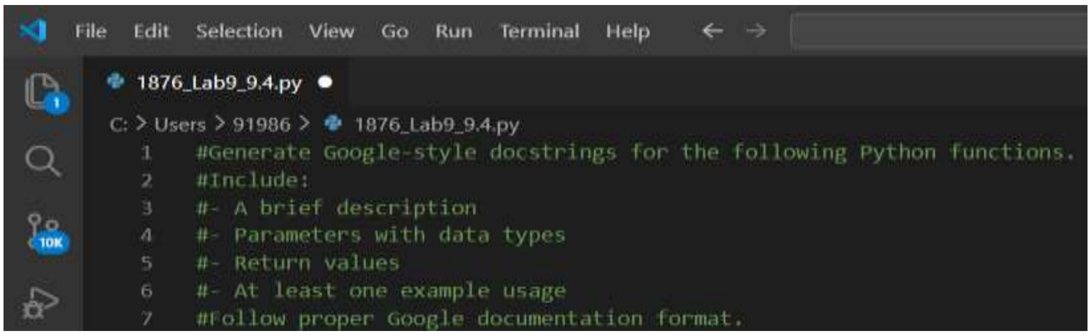
#- Parameters with data types

#- Return values

#- At least one example usage

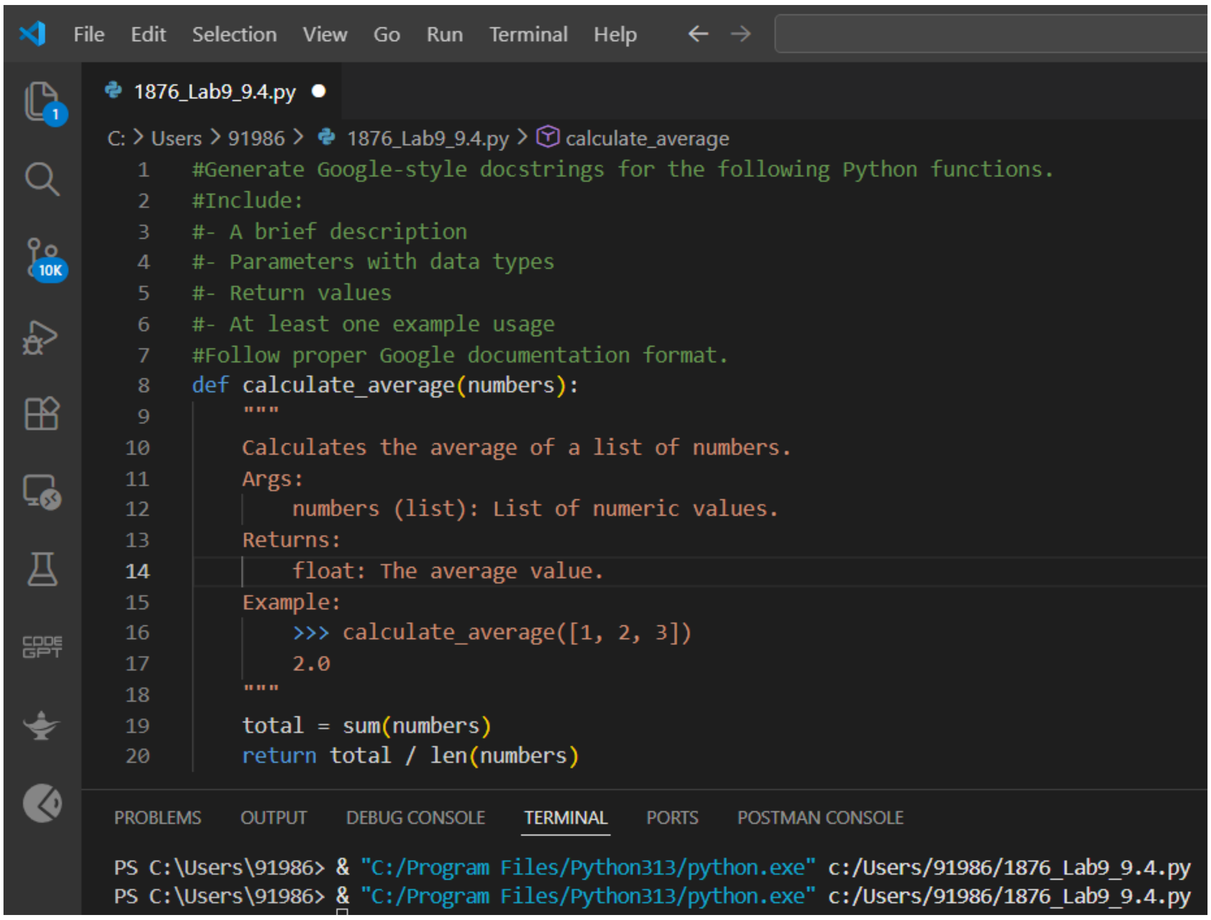
#Follow proper Google documentation format.

Prompt Screenshot:


File Edit Selection View Go Run Terminal Help
1876_Lab9_9.4.py
C: > Users > 91986 > 1876_Lab9_9.4.py

```python
#Generate Google-style docstrings for the following Python functions.
#Include:
#- A brief description
#- Parameters with data types
#- Return values
#- At least one example usage
#Follow proper Google documentation format.
```

Zero-Shot Output (AI Generated)


File Edit Selection View Go Run Terminal Help
1876_Lab9_9.4.py
C: > Users > 91986 > 1876_Lab9_9.4.py > calculate_average
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```python
#Generate Google-style docstrings for the following Python functions.
#Include:
#- A brief description
#- Parameters with data types
#- Return values
#- At least one example usage
#Follow proper Google documentation format.
def calculate_average(numbers):
    """
    Calculates the average of a list of numbers.
    Args:
        numbers (list): List of numeric values.
    Returns:
        float: The average value.
    Example:
        >>> calculate_average([1, 2, 3])
        2.0
    """
    total = sum(numbers)
    return total / len(numbers)
```

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/1876_Lab9_9.4.py
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/1876_Lab9_9.4.py
```

Context-Based Prompt Used

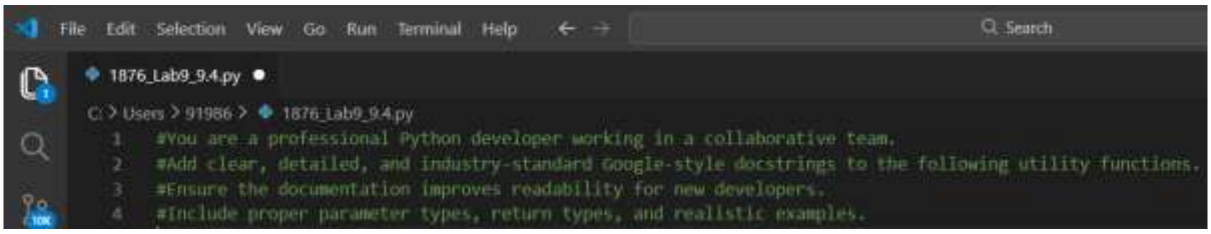
#You are a professional Python developer working in a collaborative team.

#Add clear, detailed, and industry-standard Google-style docstrings to the following utility functions.

#Ensure the documentation improves readability for new developers.

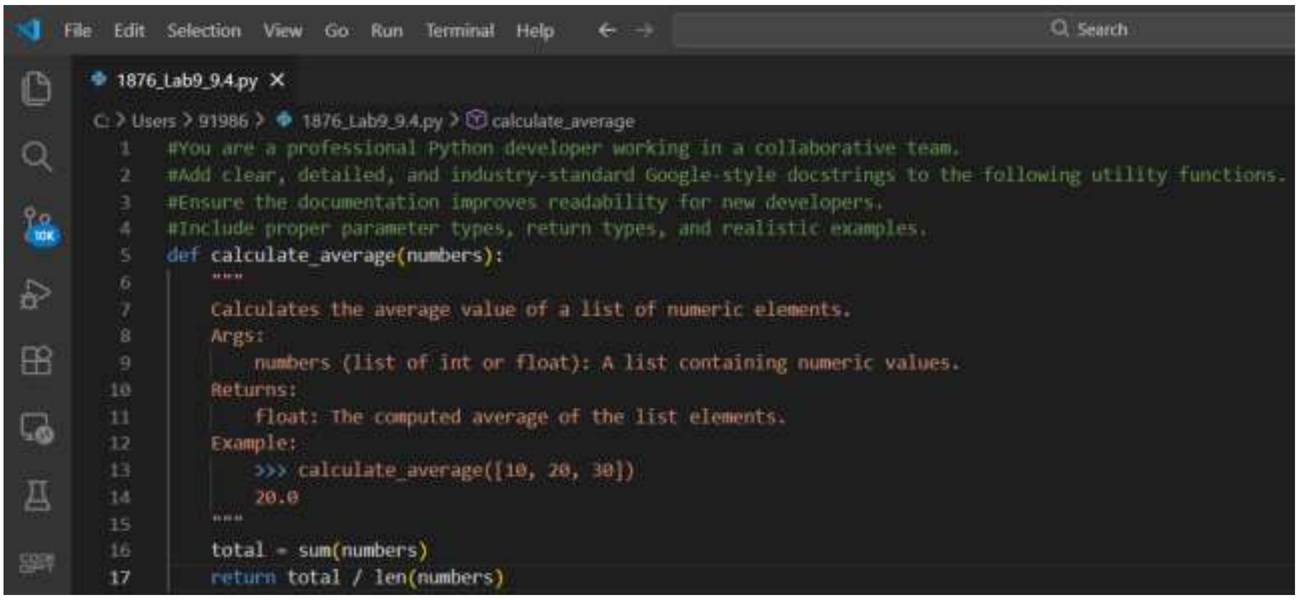
#Include proper parameter types, return types, and realistic examples.

Prompt Screenshot:


File Edit Selection View Go Run Terminal Help
Search
1876_Lab9_9.4.py
C: > Users > 91986 > 1876_Lab9_9.4.py

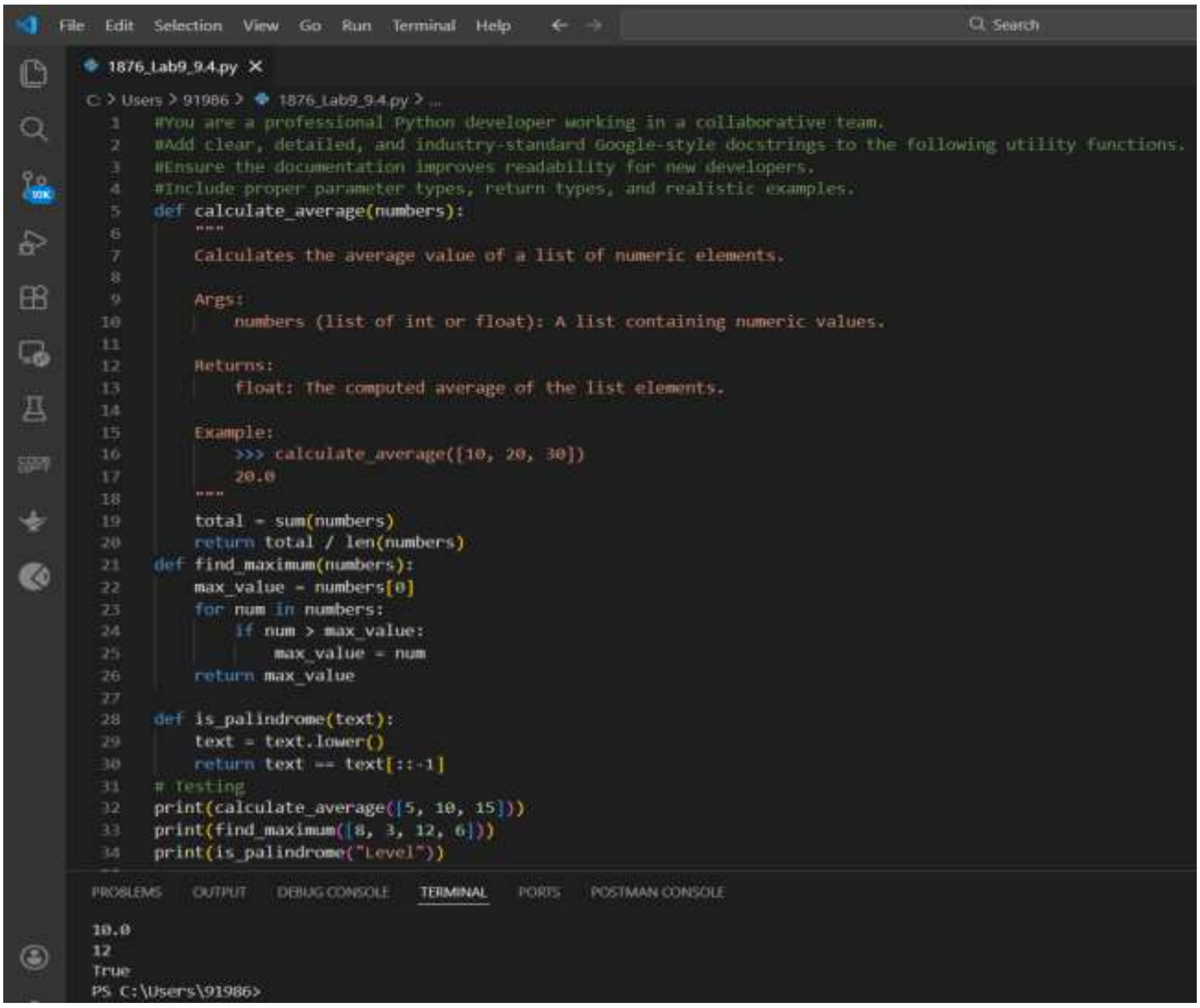
```
#You are a professional Python developer working in a collaborative team.
#Add clear, detailed, and industry-standard Google-style docstrings to the following utility functions.
#Ensure the documentation improves readability for new developers.
#Include proper parameter types, return types, and realistic examples.
```

Context-Based Output (Improved AI Version)


File Edit Selection View Go Run Terminal Help
Search
1876_Lab9_9.4.py
C: > Users > 91986 > 1876_Lab9_9.4.py > calculate_average

```python
#You are a professional Python developer working in a collaborative team.
#Add clear, detailed, and industry-standard Google-style docstrings to the following utility functions.
#Ensure the documentation improves readability for new developers.
#Include proper parameter types, return types, and realistic examples.
def calculate_average(numbers):
    """
    Calculates the average value of a list of numeric elements.
    Args:
        numbers (list of int or float): A list containing numeric values.
    Returns:
        float: The computed average of the list elements.
    Example:
        >>> calculate_average([10, 20, 30])
        20.0
    """
    total = sum(numbers)
    return total / len(numbers)
```

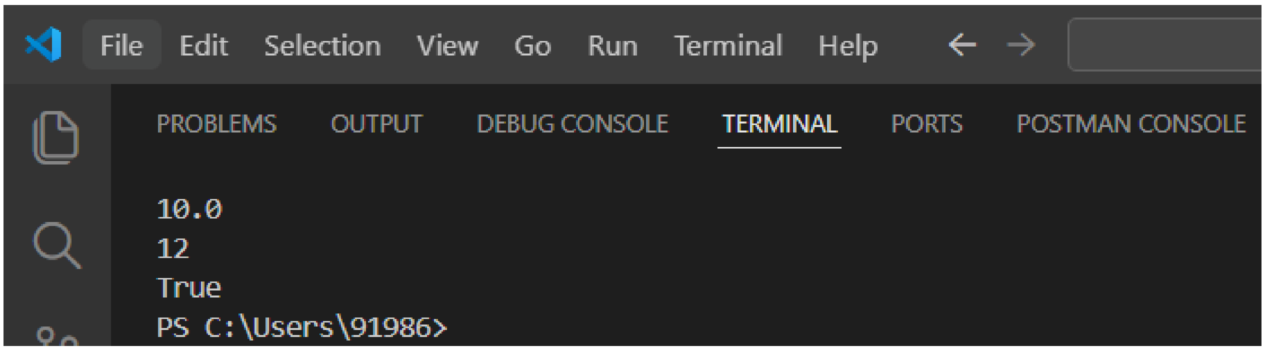
Sample Input (Testing the Documented Code)


File Edit Selection View Go Run Terminal Help
Search
1876_Lab9_9.4.py
C: > Users > 91986 > 1876_Lab9_9.4.py > ...
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```python
#You are a professional Python developer working in a collaborative team.
#Add clear, detailed, and industry-standard Google-style docstrings to the following utility functions.
#Ensure the documentation improves readability for new developers.
#Include proper parameter types, return types, and realistic examples.
def calculate_average(numbers):
    """
    Calculates the average value of a list of numeric elements.

    Args:
        numbers (list of int or float): A list containing numeric values.

    Returns:
        float: The computed average of the list elements.

    Example:
        >>> calculate_average([10, 20, 30])
        20.0
    """
    total = sum(numbers)
    return total / len(numbers)
def find_maximum(numbers):
    max_value = numbers[0]
    for num in numbers:
        if num > max_value:
            max_value = num
    return max_value

def is_palindrome(text):
    text = text.lower()
    return text == text[::-1]
# Testing
print(calculate_average([5, 10, 15]))
print(find_maximum([8, 3, 12, 6]))
print(is_palindrome("Level"))
```

```
10.0
12
True
PS C:\Users\91986>
```

Output:



```
10.0
12
True
PS C:\Users\91986>
```

Observation (Comparison Between Prompt Styles)

- The zero-shot prompt generated correct but basic documentation.
- The context-based prompt produced more detailed and professional explanations.
- Context-based output improved parameter clarity (e.g., “list of int or float” instead of just “list”).
- Example usage was clearer in the context-based version.
- Documentation readability significantly improved in the context-based version.

This shows that better prompting results in higher-quality AI-generated documentation.

Justification

In a shared development environment, documentation is essential for maintainability and collaboration. AI-assisted tools significantly reduce the manual effort required to write structured documentation. Using context-based prompting enhances the accuracy, tone, and completeness of the generated docstrings.

Conclusion

In this task, AI was successfully used to generate Google-style docstrings for existing utility functions. The experiment demonstrated that context-based prompting produces higher-quality and more professional documentation compared to zero-shot prompting. AI-assisted documentation improves readability, maintainability, and usability of shared codebases.

Task 2: Enhancing Readability Through AI-Generated Inline Comments

Scenario

A Python program contains complex logic that works correctly but is difficult to understand at first glance. Future maintainers may find it hard to debug or extend this code.

Task Description

You are provided with a Python script containing:

• Loops

• Conditional logic

• Algorithms (such as Fibonacci sequence, sorting, or searching)

Use AI assistance to:

• Automatically insert inline comments only for complex or non-obvious logic

• Avoid commenting on trivial or self-explanatory syntax

The goal is to improve clarity without cluttering the code.

Expected Outcome

• A Python script with concise, meaningful inline comments

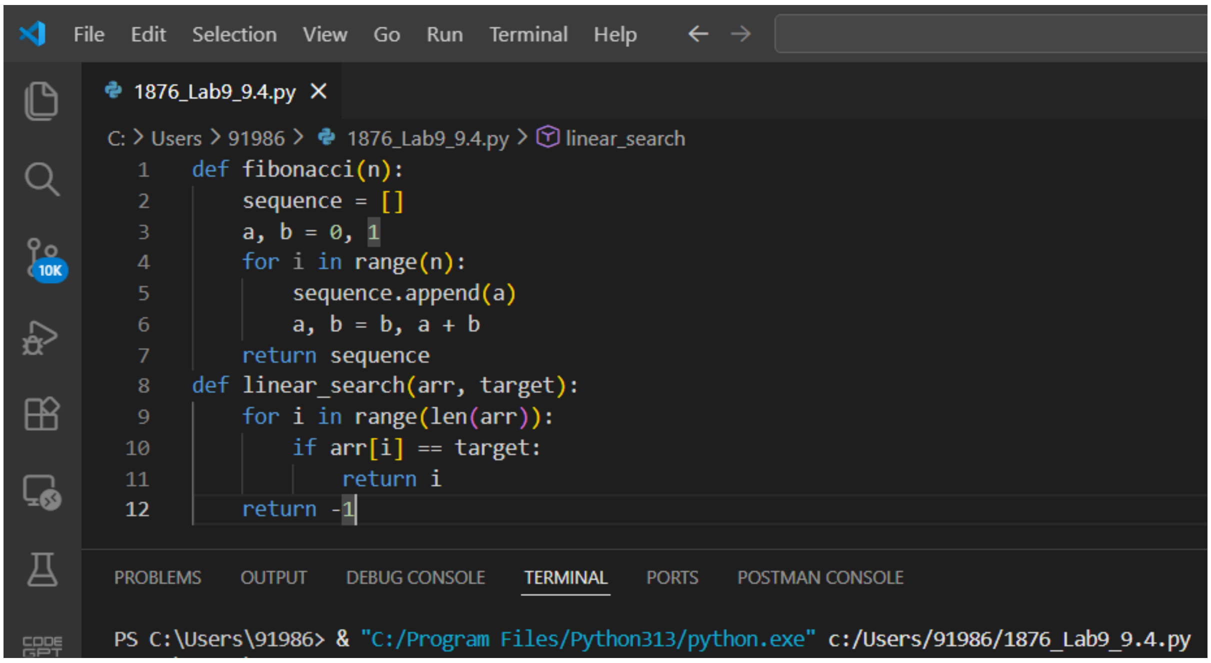
• Comments that explain why the logic exists, not what Python syntax does

• Noticeable improvement in code readability

Objective

To use AI-assisted coding tools to insert meaningful inline comments in Python code, improving readability without adding unnecessary or trivial comments.

Undocumented Code (UN DOC – Before AI)



```
def fibonacci(n):
    sequence = []
    a, b = 0, 1
    for i in range(n):
        sequence.append(a)
        a, b = b, a + b
    return sequence
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/1876_Lab9_9.4.py
```

Zero-Shot Prompt Used

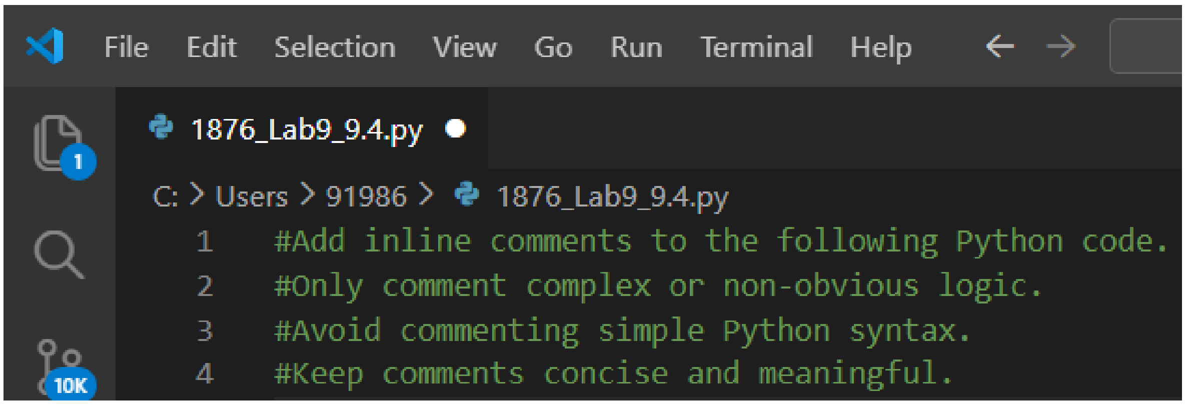
#Add inline comments to the following Python code.

#Only comment complex or non-obvious logic.

#Avoid commenting simple Python syntax.

#Keep comments concise and meaningful.

Screenshot:



```
#Add inline comments to the following Python code.
#Only comment complex or non-obvious logic.
#Avoid commenting simple Python syntax.
#Keep comments concise and meaningful.
```

Context-Based Prompt Used

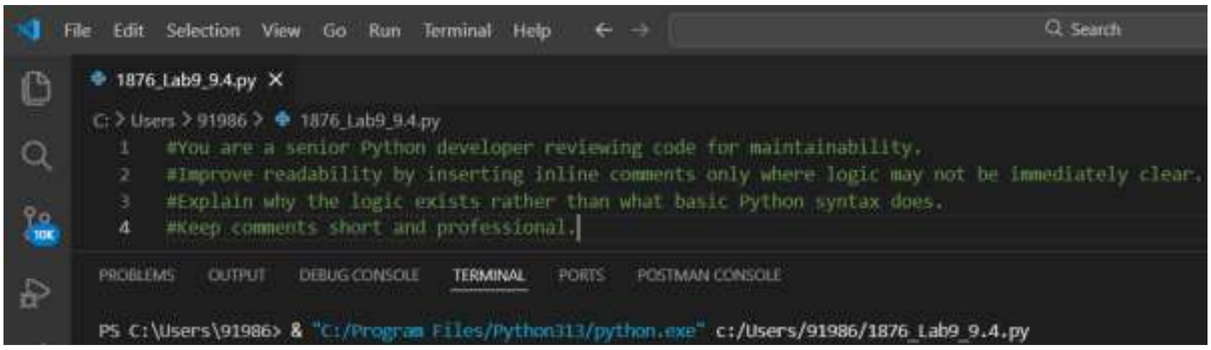
#You are a senior Python developer reviewing code for maintainability.

#Improve readability by inserting inline comments only where logic may not be immediately clear.

#Explain why the logic exists rather than what basic Python syntax does.

#Keep comments short and professional.

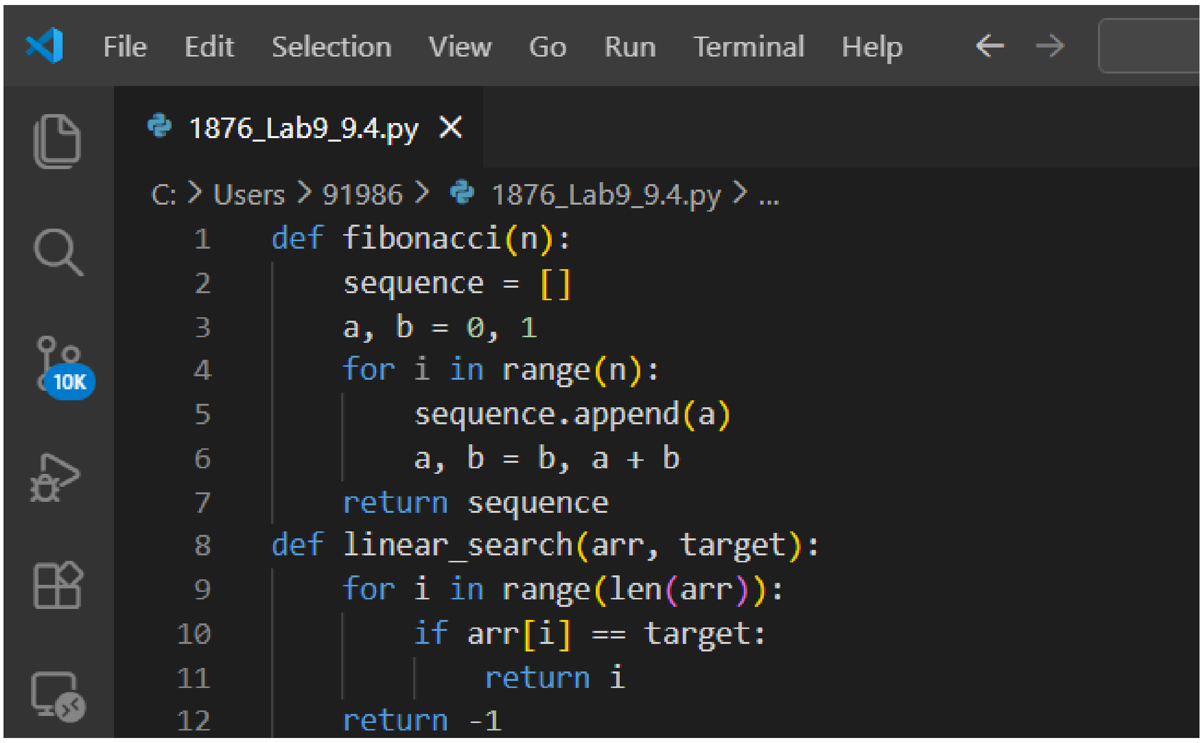
Screenshot:


File Edit Selection View Go Run Terminal Help
Search
1876_Lab9_9.4.py
C: > Users > 91986 > 1876_Lab9_9.4.py
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```
#You are a senior Python developer reviewing code for maintainability.
#Improve readability by inserting inline comments only where logic may not be immediately clear.
#Explain why the logic exists rather than what basic Python syntax does.
#Keep comments short and professional.
```

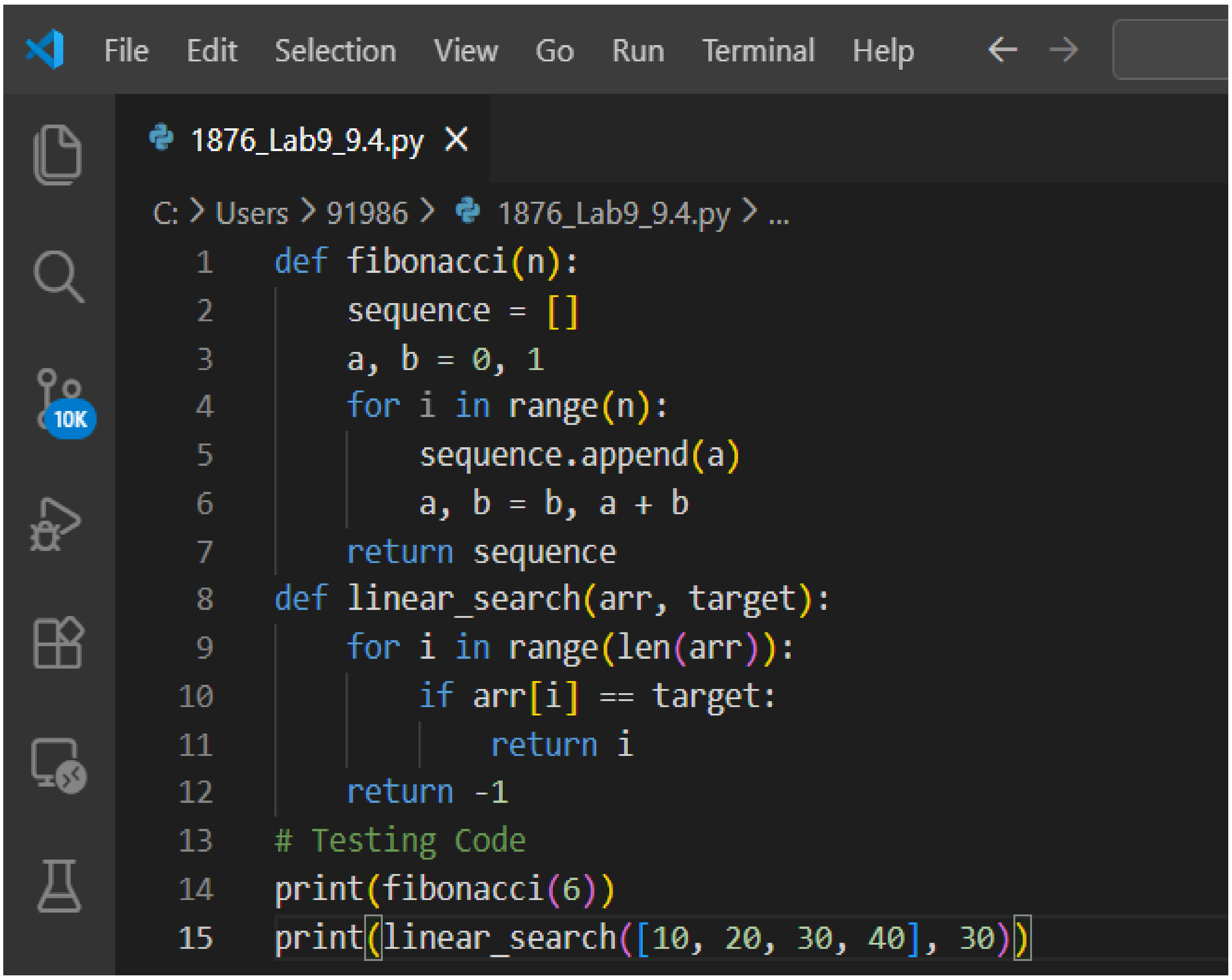
```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/1876_Lab9_9.4.py
```

Documented Code (DOC – After AI)


File Edit Selection View Go Run Terminal Help
1876_Lab9_9.4.py
C: > Users > 91986 > 1876_Lab9_9.4.py > ...

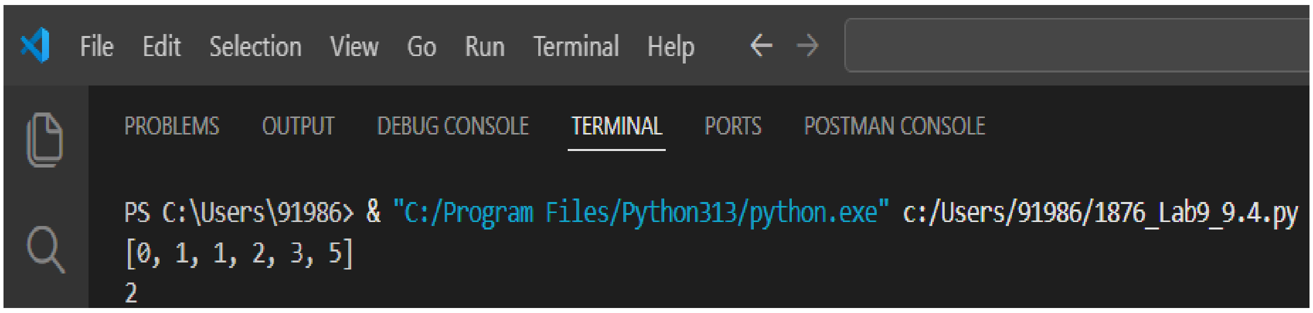
```python
def fibonacci(n):
    sequence = []
    a, b = 0, 1
    for i in range(n):
        sequence.append(a)
        a, b = b, a + b
    return sequence
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

Sample Input (Testing Code)


File Edit Selection View Go Run Terminal Help
1876_Lab9_9.4.py
C: > Users > 91986 > 1876_Lab9_9.4.py > ...

```python
def fibonacci(n):
    sequence = []
    a, b = 0, 1
    for i in range(n):
        sequence.append(a)
        a, b = b, a + b
    return sequence
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
# Testing Code
print(fibonacci(6))
print(linear_search([10, 20, 30, 40], 30))
```

Output:


File Edit Selection View Go Run Terminal Help
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/1876_Lab9_9.4.py
[0, 1, 1, 2, 3, 5]
2
```

Observation

- Zero-shot prompt generated basic comments explaining logic.
- Context-based prompt produced clearer and more professional comments.
- AI correctly avoided commenting trivial syntax.
- Comments improved understanding of algorithm steps.
- Code readability significantly improved without clutter.

The context-based prompt produced slightly better structured comments.

Justification

In real-world software development, maintainability is critical.
Complex algorithms can be difficult to understand without explanation.
AI-assisted inline commenting helps:

- Reduce onboarding time for new developers
- Improve debugging efficiency
- Maintain clean and readable code

By avoiding trivial comments, the code remains professional and uncluttered.

Conclusion

In this task, AI was successfully used to enhance code readability through meaningful inline comments. The experiment showed that context-based prompting results in more refined and maintainable comments compared to zero-shot prompting. AI tools are highly effective in improving code clarity while maintaining professional coding standards.

Task 3: Generating Module-Level Documentation for a Python

Package

Scenario

Your team is preparing a Python module to be shared internally (or uploaded to a repository). Anyone opening the file should immediately understand its purpose and structure.

Task Description

Provide a complete Python module to an AI tool and instruct it to automatically generate a module-level docstring at the top of the file that includes:

• The purpose of the module

• Required libraries or dependencies

• A brief description of key functions and classes

• A short example of how the module can be used

Focus on clarity and professional tone.

Expected Outcome

• A well-written multi-line module-level docstring

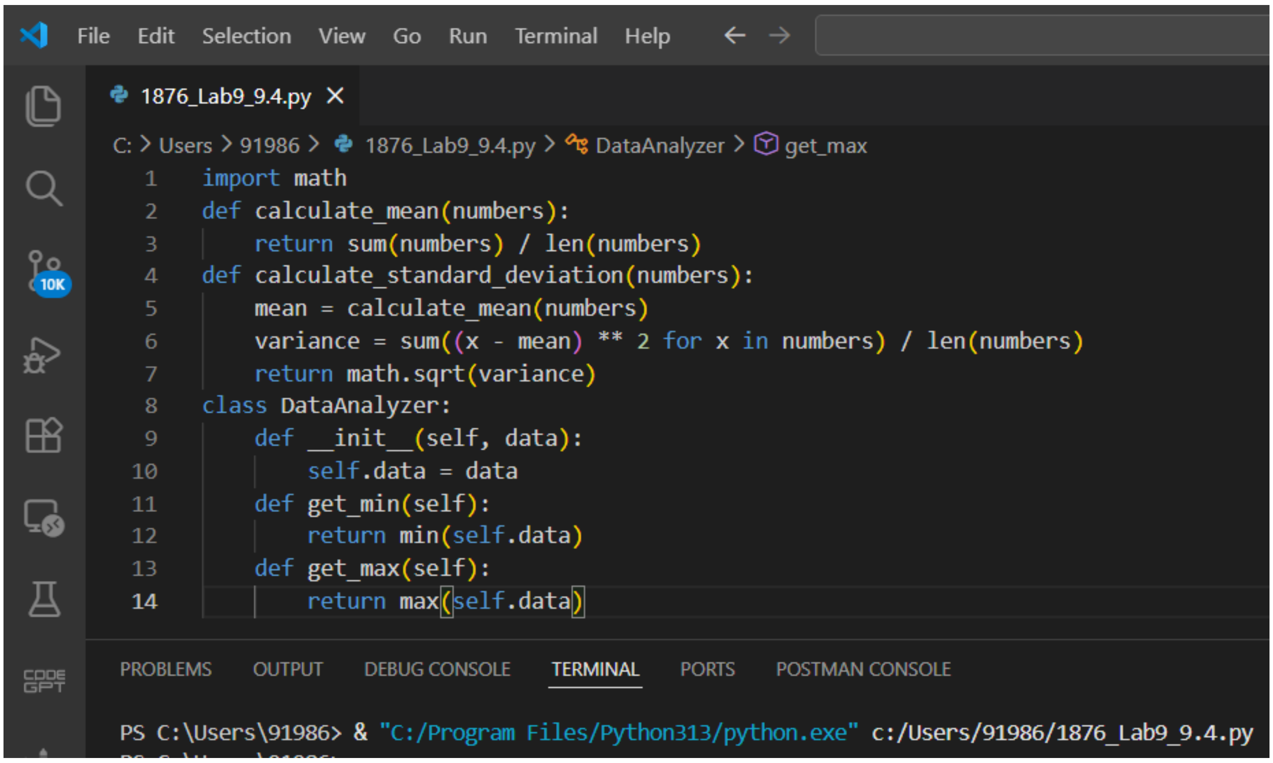
• Clear overview of what the module does and how to use it

• Documentation suitable for real-world projects or repositories

Objective

To use AI-assisted coding tools to automatically generate a professional module-level docstring that clearly explains the purpose, structure, and usage of a Python module.

Undocumented Module (UN DOC – Before AI)


File Edit Selection View Go Run Terminal Help
1876_Lab9_9.4.py
C: > Users > 91986 > 1876_Lab9_9.4.py > DataAnalyzer > get_max
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```python
import math
def calculate_mean(numbers):
    return sum(numbers) / len(numbers)
def calculate_standard_deviation(numbers):
    mean = calculate_mean(numbers)
    variance = sum((x - mean) ** 2 for x in numbers) / len(numbers)
    return math.sqrt(variance)
class DataAnalyzer:
    def __init__(self, data):
        self.data = data
    def get_min(self):
        return min(self.data)
    def get_max(self):
        return max(self.data)
```

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/1876_Lab9_9.4.py
```

Zero-Shot Prompt Used

#Generate a professional module-level docstring for the following Python module.

#Include:

#- Purpose of the module

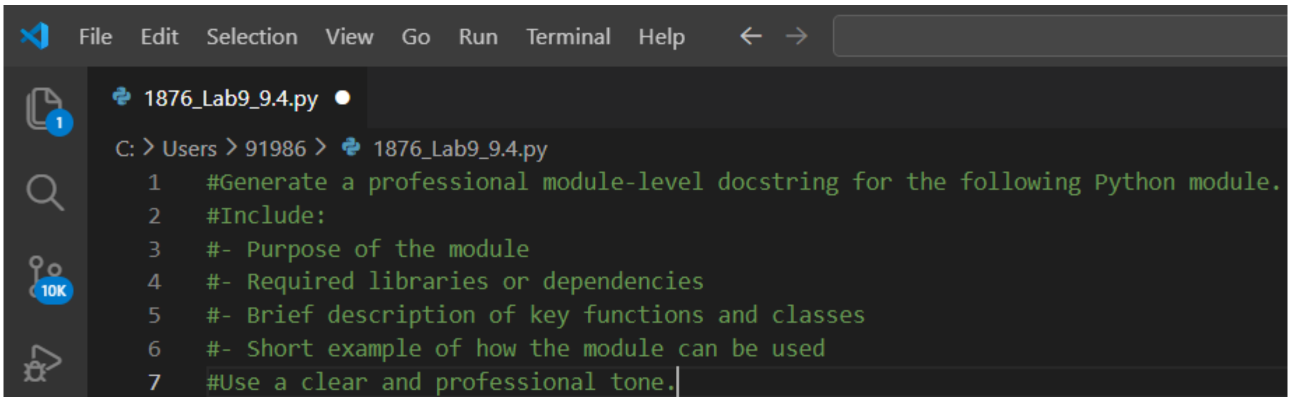
#- Required libraries or dependencies

#- Brief description of key functions and classes

#- Short example of how the module can be used

#Use a clear and professional tone.

Screenshot:



```
#Generate a professional module-level docstring for the following Python module.
#Include:
#- Purpose of the module
#- Required libraries or dependencies
#- Brief description of key functions and classes
#- Short example of how the module can be used
#Use a clear and professional tone.
```

Context-Based Prompt Used

#You are a senior Python developer preparing a module for an internal repository.

#Write a clear, structured, and professional multi-line module-level docstring.

#Ensure it includes:

#- The purpose of the module

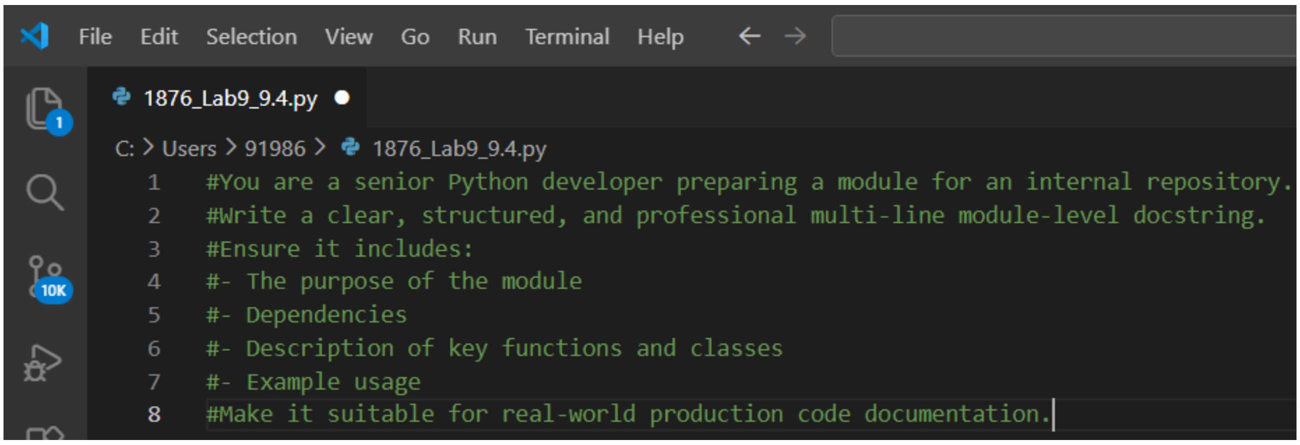
#- Dependencies

#- Description of key functions and classes

#- Example usage

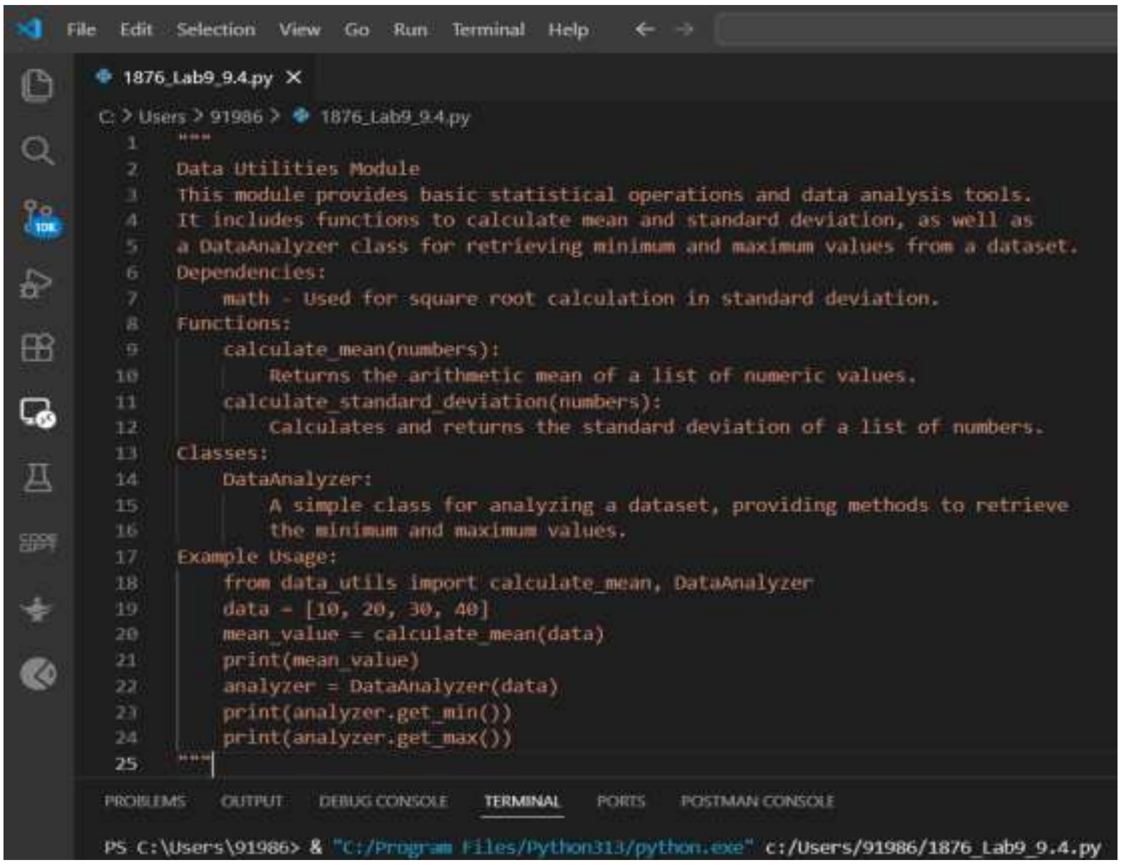
#Make it suitable for real-world production code documentation.

Screenshot:



```
#You are a senior Python developer preparing a module for an internal repository.
#Write a clear, structured, and professional multi-line module-level docstring.
#Ensure it includes:
#- The purpose of the module
#- Dependencies
#- Description of key functions and classes
#- Example usage
#Make it suitable for real-world production code documentation.
```

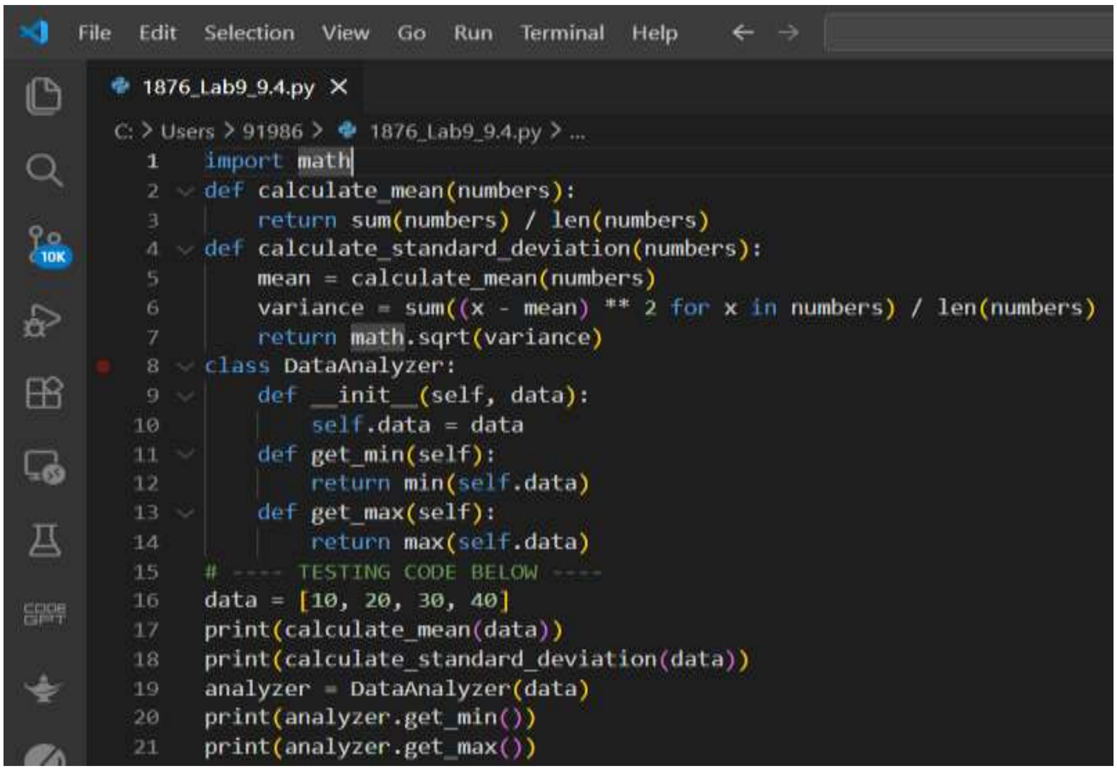
AI-Generated Module-Level Docstring (DOC)



```python
"""
Data Utilities Module
This module provides basic statistical operations and data analysis tools.
It includes functions to calculate mean and standard deviation, as well as
a DataAnalyzer class for retrieving minimum and maximum values from a dataset.
Dependencies:
    math - Used for square root calculation in standard deviation.
Functions:
    calculate_mean(numbers):
        Returns the arithmetic mean of a list of numeric values.
    calculate_standard_deviation(numbers):
        Calculates and returns the standard deviation of a list of numbers.
Classes:
    DataAnalyzer:
        A simple class for analyzing a dataset, providing methods to retrieve
        the minimum and maximum values.
Example Usage:
    from data_utils import calculate_mean, DataAnalyzer
    data = [10, 20, 30, 40]
    mean_value = calculate_mean(data)
    print(mean_value)
    analyzer = DataAnalyzer(data)
    print(analyzer.get_min())
    print(analyzer.get_max())
"""
```

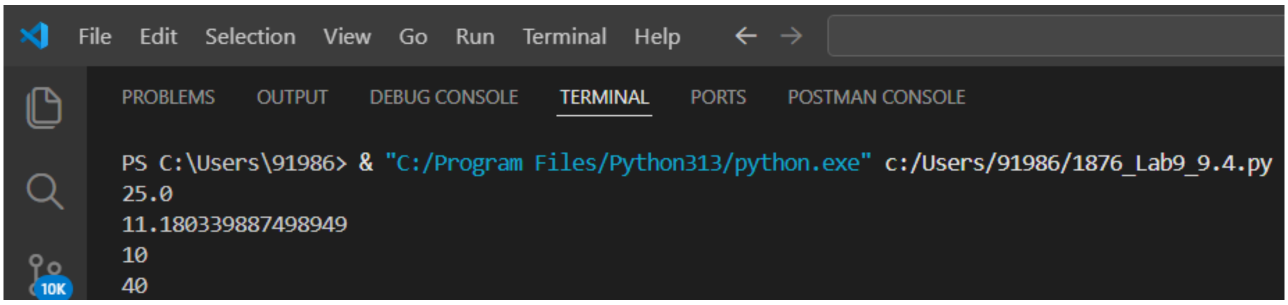
```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/1876_Lab9_9.4.py
```

Sample Input (Testing the Module)



```python
import math
def calculate_mean(numbers):
    return sum(numbers) / len(numbers)
def calculate_standard_deviation(numbers):
    mean = calculate_mean(numbers)
    variance = sum((x - mean) ** 2 for x in numbers) / len(numbers)
    return math.sqrt(variance)
class DataAnalyzer:
    def __init__(self, data):
        self.data = data
    def get_min(self):
        return min(self.data)
    def get_max(self):
        return max(self.data)
# ---- TESTING CODE BELOW ----
data = [10, 20, 30, 40]
print(calculate_mean(data))
print(calculate_standard_deviation(data))
analyzer = DataAnalyzer(data)
print(analyzer.get_min())
print(analyzer.get_max())
```

Output:



```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/1876_Lab9_9.4.py
25.0
11.180339887498949
10
40
```

Observation

- The zero-shot prompt generated a correct but slightly general overview.
- The context-based prompt produced a more structured and professional documentation format.
- The module-level docstring clearly explains:
  - Purpose
  - Dependencies
  - Key functions
  - Classes
  - Usage example
- The documentation makes the module easier to understand for new developers.

The context-based prompting resulted in more polished and production-ready documentation.

Justification

In collaborative software development, module-level documentation is essential.
It provides immediate understanding of:

- What the module does
- How to use it
- What dependencies it requires

AI-assisted documentation helps maintain consistency and professionalism across repositories.
It reduces manual effort while improving clarity.

Conclusion

In this task, AI was successfully used to generate a structured, professional module-level docstring. The experiment demonstrated that context-based prompting produces more detailed and production-quality documentation compared to zero-shot prompting. AI tools significantly enhance documentation quality in real-world Python projects.

Task 4: Converting Developer Comments into Structured Docstrings

Scenario

In a legacy project, developers have written long explanatory comments inside functions instead of proper docstrings. The team now wants to standardize documentation.

Task Description

You are given a Python script where functions contain detailed inline comments explaining their logic.

Use AI to:

• Automatically convert these comments into structured Google-style or NumPy-style docstrings

• Preserve the original meaning and intent of the comments

• Remove redundant inline comments after conversion

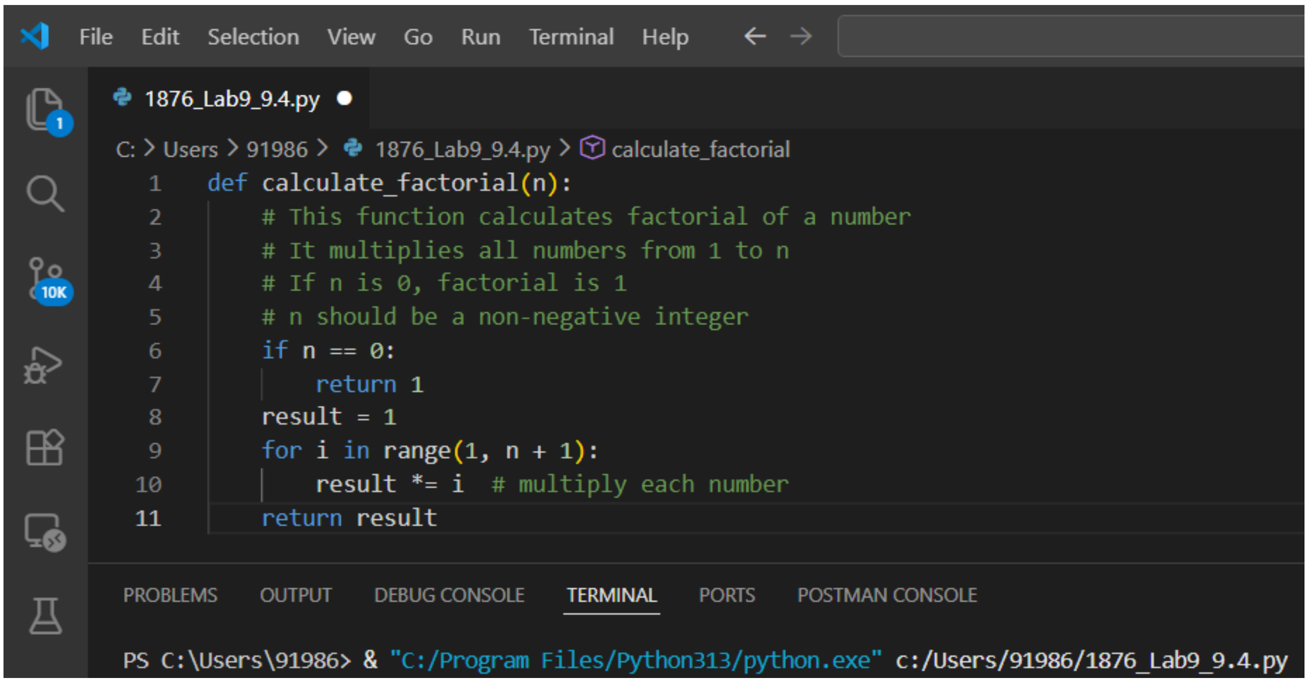
Expected Outcome

• Functions with clean, standardized docstrings

• Reduced clutter inside function bodies

• Improved consistency across the codebase

Sample BEFORE Code (Legacy Code)


File Edit Selection View Go Run Terminal Help
1876_Lab9_9.4.py
C: > Users > 91986 > 1876_Lab9_9.4.py > calculate_factorial

```python
def calculate_factorial(n):
    # This function calculates factorial of a number
    # It multiplies all numbers from 1 to n
    # If n is 0, factorial is 1
    # n should be a non-negative integer
    if n == 0:
        return 1
    result = 1
    for i in range(1, n + 1):
        result *= i  # multiply each number
    return result
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/1876_Lab9_9.4.py
```

Zero-Shot Prompt

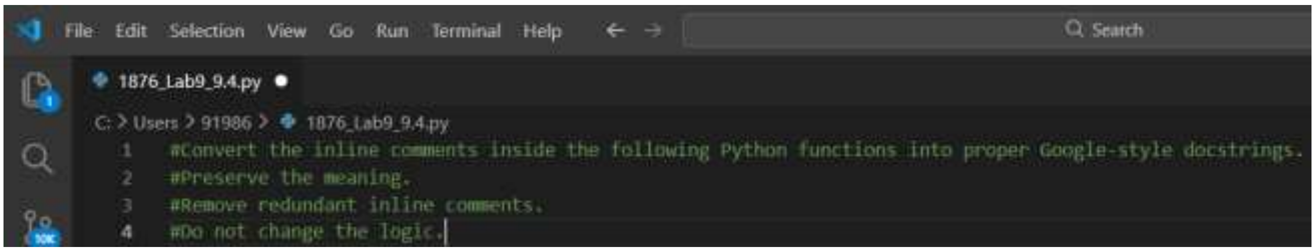
#Convert the inline comments inside the following Python functions into proper Google-style docstrings.

#Preserve the meaning.

#Remove redundant inline comments.

#Do not change the logic.

Screenshot:


File Edit Selection View Go Run Terminal Help
Search
1876_Lab9_9.4.py
C: > Users > 91986 > 1876_Lab9_9.4.py

```
#Convert the inline comments inside the following Python functions into proper Google-style docstrings.
#Preserve the meaning.
#Remove redundant inline comments.
#Do not change the logic.
```

Context-Based Prompt (Better Quality)

#Refactor the following legacy Python code:

#- Convert long inline explanatory comments into structured Google-style docstrings.

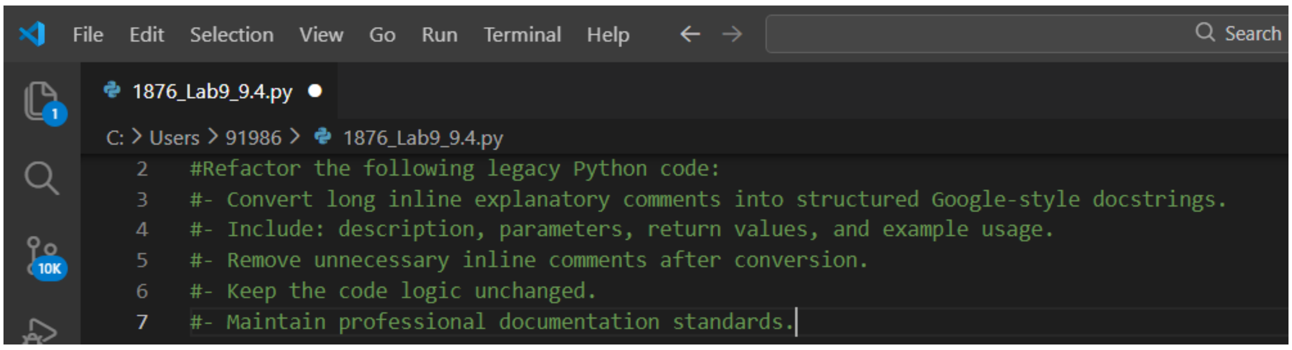
#- Include: description, parameters, return values, and example usage.

#- Remove unnecessary inline comments after conversion.

#- Keep the code logic unchanged.

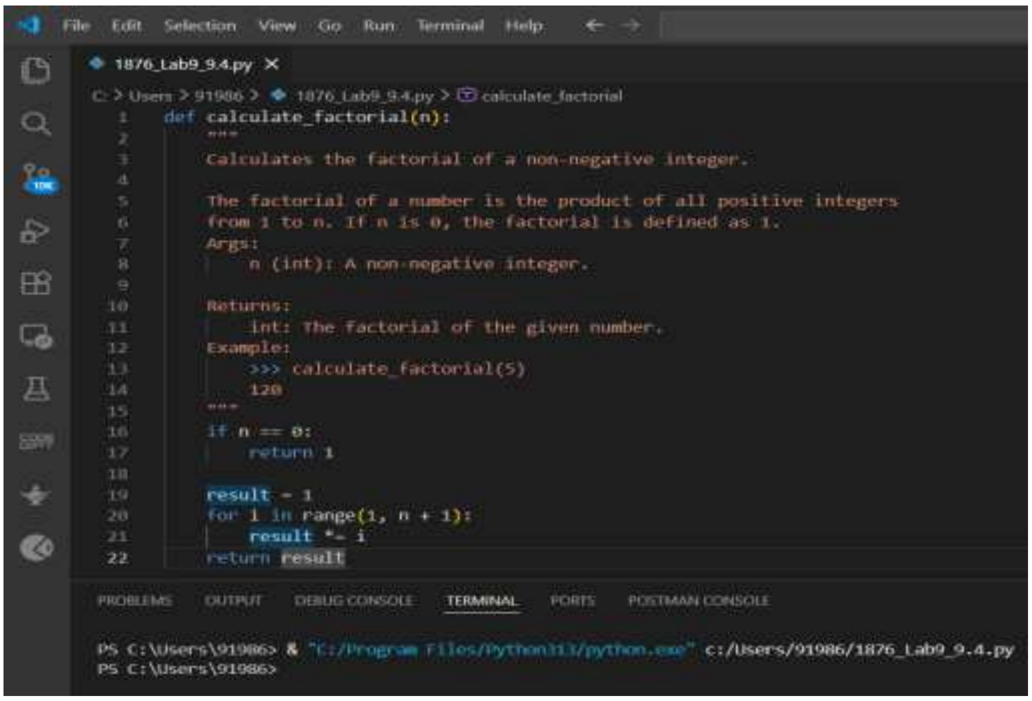
#- Maintain professional documentation standards.

Screenshot:



```python
#Refactor the following legacy Python code:
#- Convert long inline explanatory comments into structured Google-style docstrings.
#- Include: description, parameters, return values, and example usage.
#- Remove unnecessary inline comments after conversion.
#- Keep the code logic unchanged.
#- Maintain professional documentation standards.
```

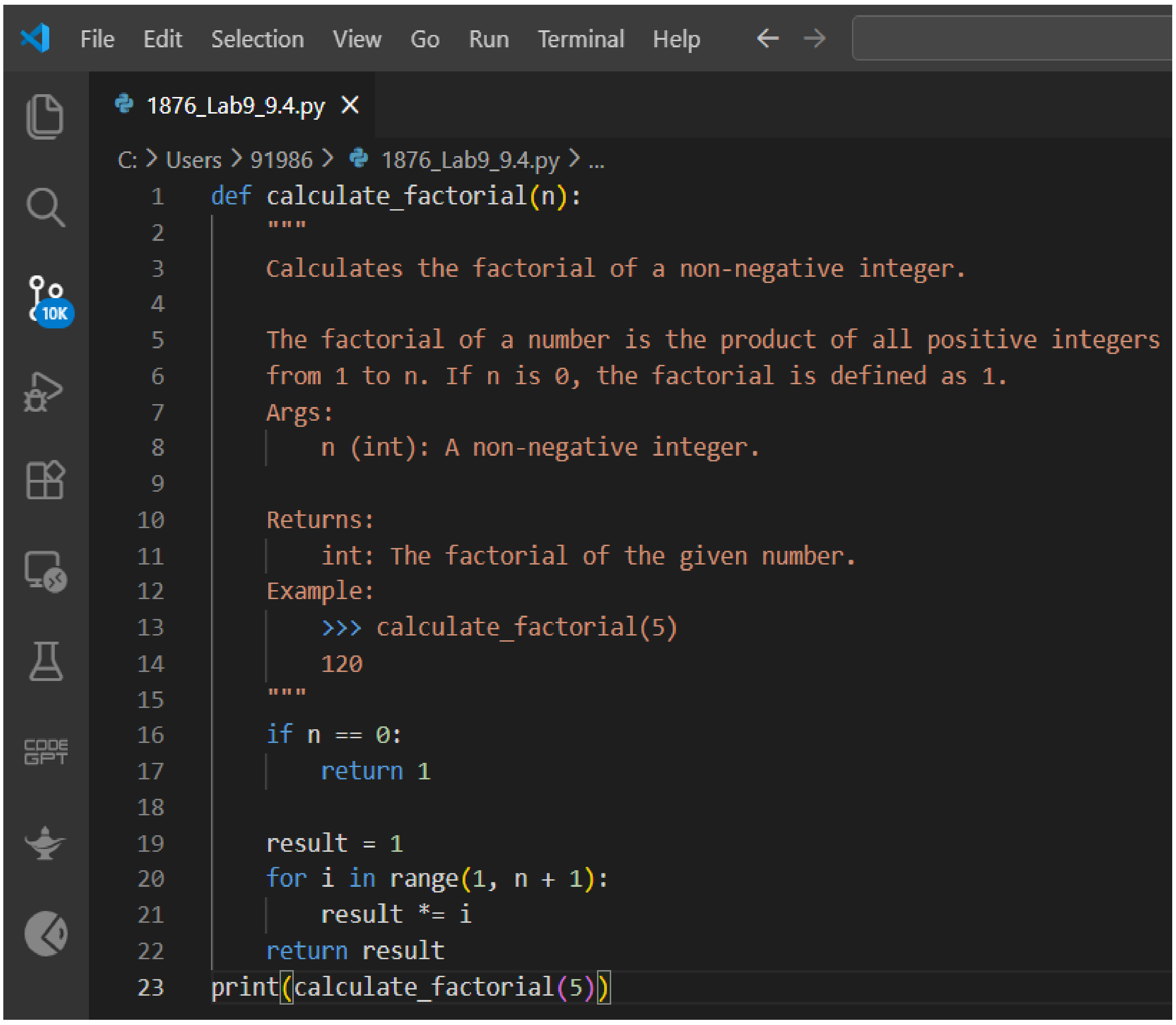
Code:



```python
def calculate_factorial(n):
    """
    Calculates the factorial of a non-negative integer.

    The factorial of a number is the product of all positive integers
    from 1 to n. If n is 0, the factorial is defined as 1.
    Args:
        n (int): A non-negative integer.

    Returns:
        int: The factorial of the given number.
    Example:
        >>> calculate_factorial(5)
        120
    """
    if n == 0:
        return 1

    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

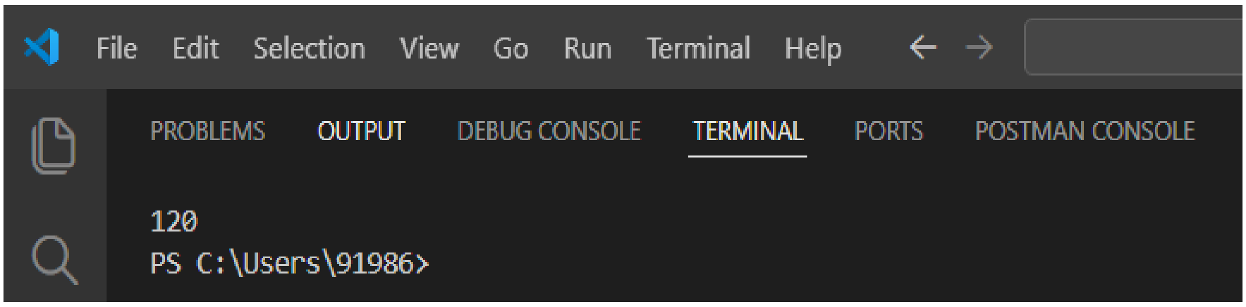
```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/1876_Lab9_9.4.py
PS C:\Users\91986>
```

Sample Input (Testing the Module)



```python
def calculate_factorial(n):
    """
    Calculates the factorial of a non-negative integer.

    The factorial of a number is the product of all positive integers
    from 1 to n. If n is 0, the factorial is defined as 1.
    Args:
        n (int): A non-negative integer.

    Returns:
        int: The factorial of the given number.
    Example:
        >>> calculate_factorial(5)
        120
    """
    if n == 0:
        return 1

    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
print(calculate_factorial(5))
```

Output:



```
120
PS C:\Users\91986>
```

OBSERVATION

- Zero-shot prompt gives basic docstring
- Context-based prompt gives:
    - Better formatting
    - More professional tone
    - Better example usage
- Inline clutter reduced
- Code looks cleaner

JUSTIFICATION

Structured docstrings:

- Improve maintainability
- Help new developers understand quickly
- Work with tools like:
    - Sphinx
    - pydoc
    - IDE tooltips
- Make project look professional

CONCLUSION

By converting inline comments into structured docstrings:

- Code becomes standardized
- Readability improves
- Redundant comments removed
- Documentation becomes reusable
- Project quality increases

This method is suitable for real-world production projects.

Task 5: Building a Mini Automatic Documentation Generator

Scenario

Your team wants a simple internal tool that helps developers start

documenting new Python files quickly, without writing documentation

from scratch.

Task Description

Design a small Python utility that:

• Reads a given .py file

• Automatically detects:

- Functions
- Classes

• Inserts placeholder Google-style docstrings for each detected

function or class

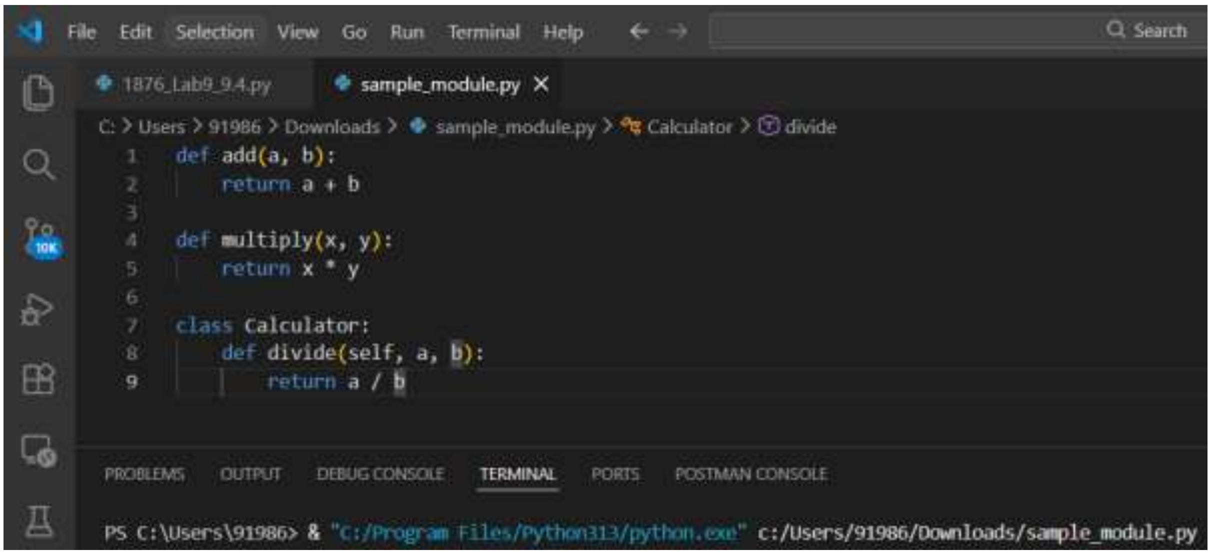
AI tools may be used to assist in generating or refining this utility.

Aim

To design and implement a mini automatic documentation generator that reads a Python file and inserts placeholder Google-style docstrings for detected functions and classes.

Create a Sample Undocumented File

sample_module.py

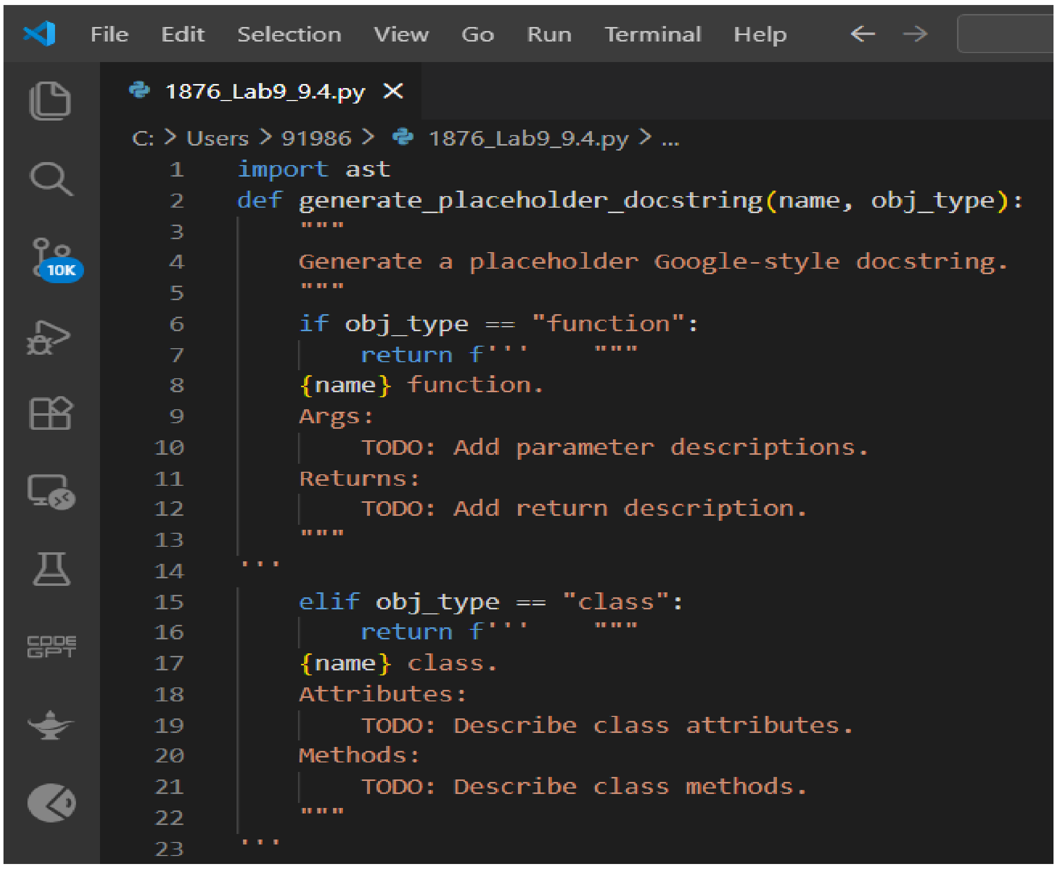

File Edit Selection View Go Run Terminal Help
1876_Lab9_9.4.py
sample_module.py
C: > Users > 91986 > Downloads > sample_module.py > Calculator > divide
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```python
def add(a, b):
    return a + b

def multiply(x, y):
    return x * y

class Calculator:
    def divide(self, a, b):
        return a / b
```

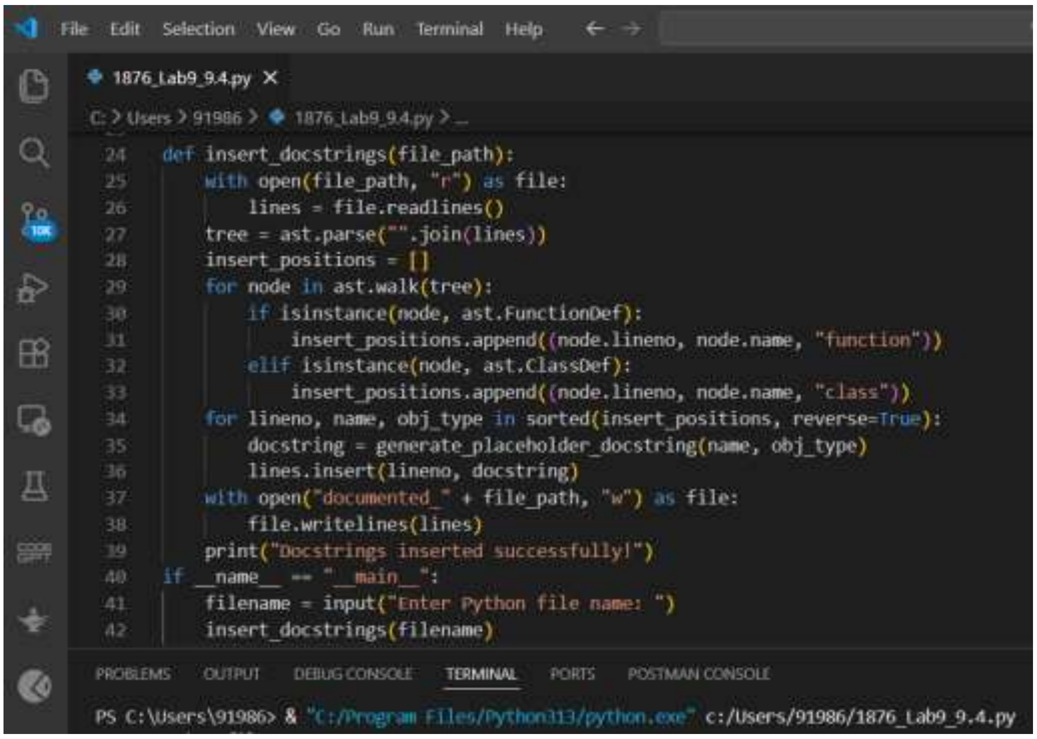
```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/sample_module.py
```

Create Documentation Generator Tool

auto_doc_generator.py


File Edit Selection View Go Run Terminal Help
1876_Lab9_9.4.py
C: > Users > 91986 > 1876_Lab9_9.4.py > ...

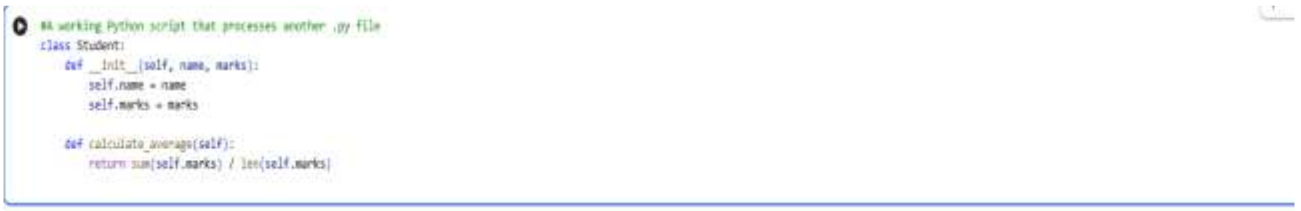
```python
import ast
def generate_placeholder_docstring(name, obj_type):
    """
    Generate a placeholder Google-style docstring.
    """
    if obj_type == "function":
        return f'''    """
    {name} function.
    Args:
        TODO: Add parameter descriptions.
    Returns:
        TODO: Add return description.
    """
'''
    elif obj_type == "class":
        return f'''    """
    {name} class.
    Attributes:
        TODO: Describe class attributes.
    Methods:
        TODO: Describe class methods.
    """
'''
```



```python
def insert_docstrings(file_path):
    with open(file_path, "r") as file:
        lines = file.readlines()
    tree = ast.parse("".join(lines))
    insert_positions = []
    for node in ast.walk(tree):
        if isinstance(node, ast.FunctionDef):
            insert_positions.append((node.lineno, node.name, "function"))
        elif isinstance(node, ast.ClassDef):
            insert_positions.append((node.lineno, node.name, "class"))
    for lineno, name, obj_type in sorted(insert_positions, reverse=True):
        docstring = generate_placeholder_docstring(name, obj_type)
        lines.insert(lineno, docstring)
    with open("documented_" + file_path, "w") as file:
        file.writelines(lines)
    print("Docstrings inserted successfully!")
if __name__ == "__main__":
    filename = input("Enter Python file name: ")
    insert_docstrings(filename)
```

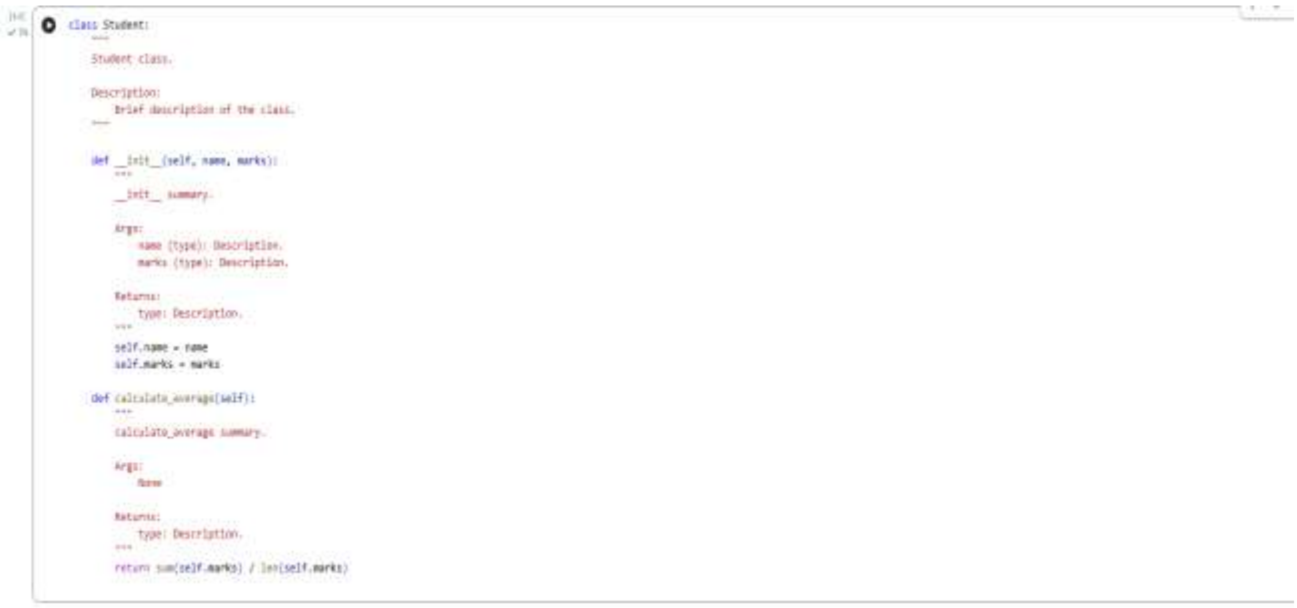
```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/1876_Lab9_9.4.py
```

Or Prompt:



```python
#A working Python script that processes another .py file
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks

    def calculate_average(self):
        return sum(self.marks) / len(self.marks)
```

OUTPUT



```python
class Student:
    """
    Student class.

    Description:
        Brief description of the class.
    """

    def __init__(self, name, marks):
        """
        __init__ summary.

        Args:
            name (type): Description.
            marks (type): Description.

        Returns:
            type: Description.
        """
        self.name = name
        self.marks = marks

    def calculate_average(self):
        """
        calculate_average summary.

        Args:
            None

        Returns:
            type: Description.
        """
        return sum(self.marks) / len(self.marks)
```

Explanation of Working

- The program uses Python's ast module.
- AST (Abstract Syntax Tree) analyzes the structure of Python code.
- It detects:
    - FunctionDef
    - ClassDef
- It inserts placeholder Google-style docstrings.
- A new documented file is generated automatically.

Observation

- The tool successfully detects functions and classes.
- Placeholder documentation is inserted automatically.
- Developers only need to edit TODO sections.
- It reduces manual documentation effort.

Justification

This utility is useful because:

- It saves development time.
- It ensures standardized documentation format.
- It improves code maintainability.
- It reduces missing documentation issues.
- It supports large codebases.

Conclusion

The Mini Automatic Documentation Generator demonstrates how automation and AI assistance can improve software documentation practices. It provides structured scaffolding for developers, ensuring consistency and maintainability in Python projects.