Assignment-9.4

Name: Sushma Purella

H. No:2303A52188

Batch:44

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

Ask the AI to automatically generate Google-style function

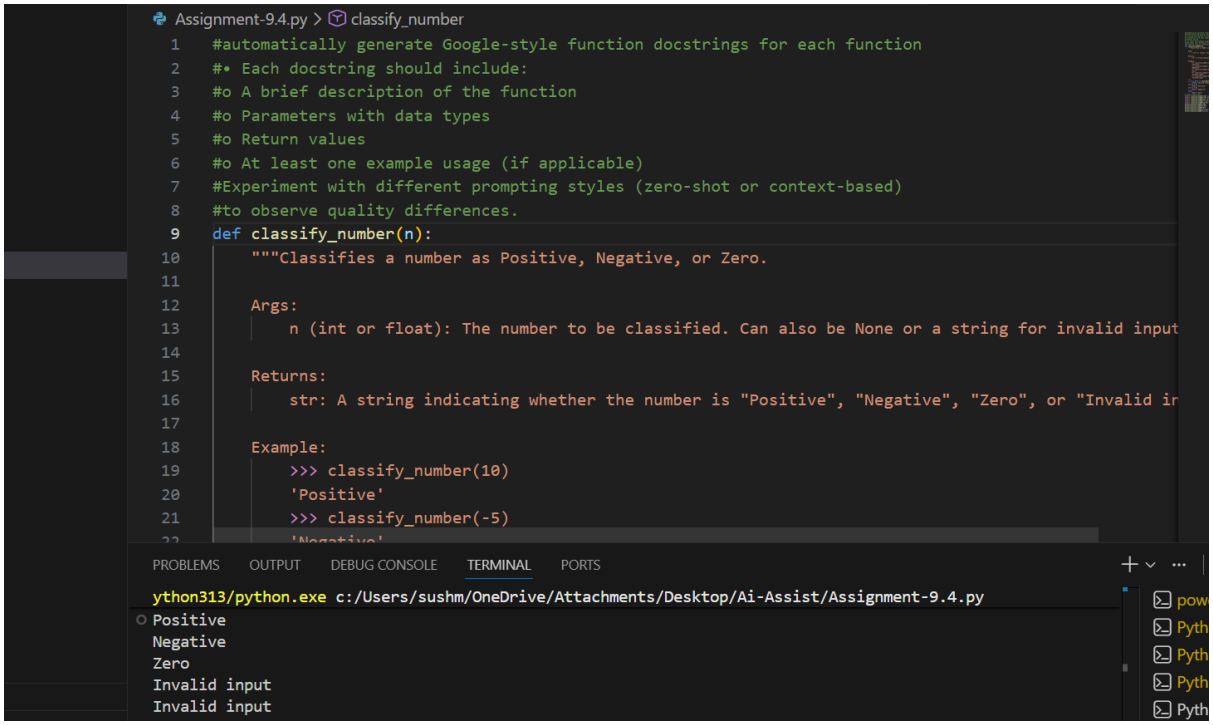
docstrings for each function

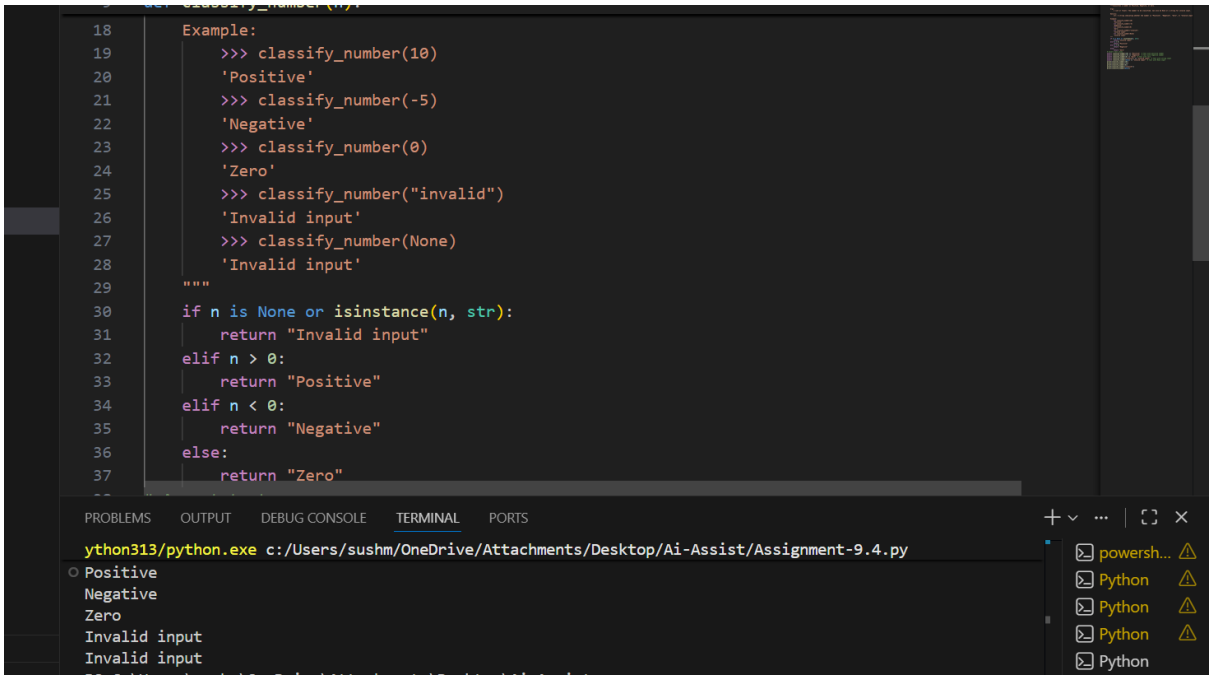
- Each docstring should include:
- A brief description of the function
- Parameters with data types
- Return values
- At least one example usage

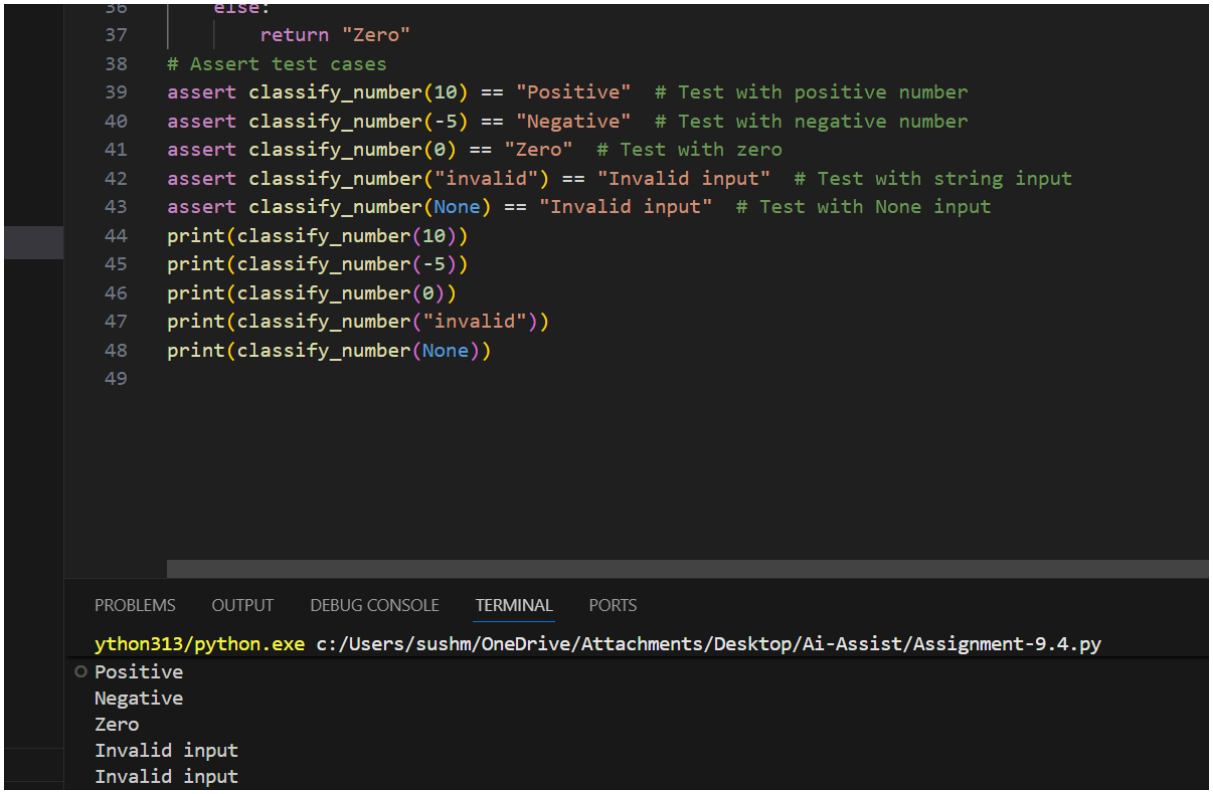
Prompt:

"Generate Google-style docstrings for the following Python function."

Code:


Assignment-9.4.py > classify_number
#automatically generate Google-style function docstrings for each function
#• Each docstring should include:
#o A brief description of the function
#o Parameters with data types
#o Return values
#o At least one example usage (if applicable)
#Experiment with different prompting styles (zero-shot or context-based)
#to observe quality differences.
def classify_number(n):
"""Classifies a number as Positive, Negative, or Zero.
Args:
n (int or float): The number to be classified. Can also be None or a string for invalid input
Returns:
str: A string indicating whether the number is "Positive", "Negative", "Zero", or "Invalid ir
Example:
>>> classify_number(10)
'Positive'
>>> classify_number(-5)
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
ython313/python.exe c:/Users/sushm/OneDrive/Attachments/Desktop/Ai-Assist/Assignment-9.4.py
Positive
Negative
Zero
Invalid input
Invalid input


Example:
>>> classify_number(10)
'Positive'
>>> classify_number(-5)
'Negative'
>>> classify_number(0)
'Zero'
>>> classify_number("invalid")
'Invalid input'
>>> classify_number(None)
'Invalid input'
"""
if n is None or isinstance(n, str):
return "Invalid input"
elif n > 0:
return "Positive"
elif n < 0:
return "Negative"
else:
return "Zero"
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
ython313/python.exe c:/Users/sushm/OneDrive/Attachments/Desktop/Ai-Assist/Assignment-9.4.py
Positive
Negative
Zero
Invalid input
Invalid input
powersh...
Python
Python
Python
Python



Observation:

- The AI generated a basic docstring.
- Description and parameters were included.
- Examples were sometimes missing or minimal.
- Formatting was correct but less detailed.

Task 2: Enhancing Readability Through AI-Generated Inline Comments

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

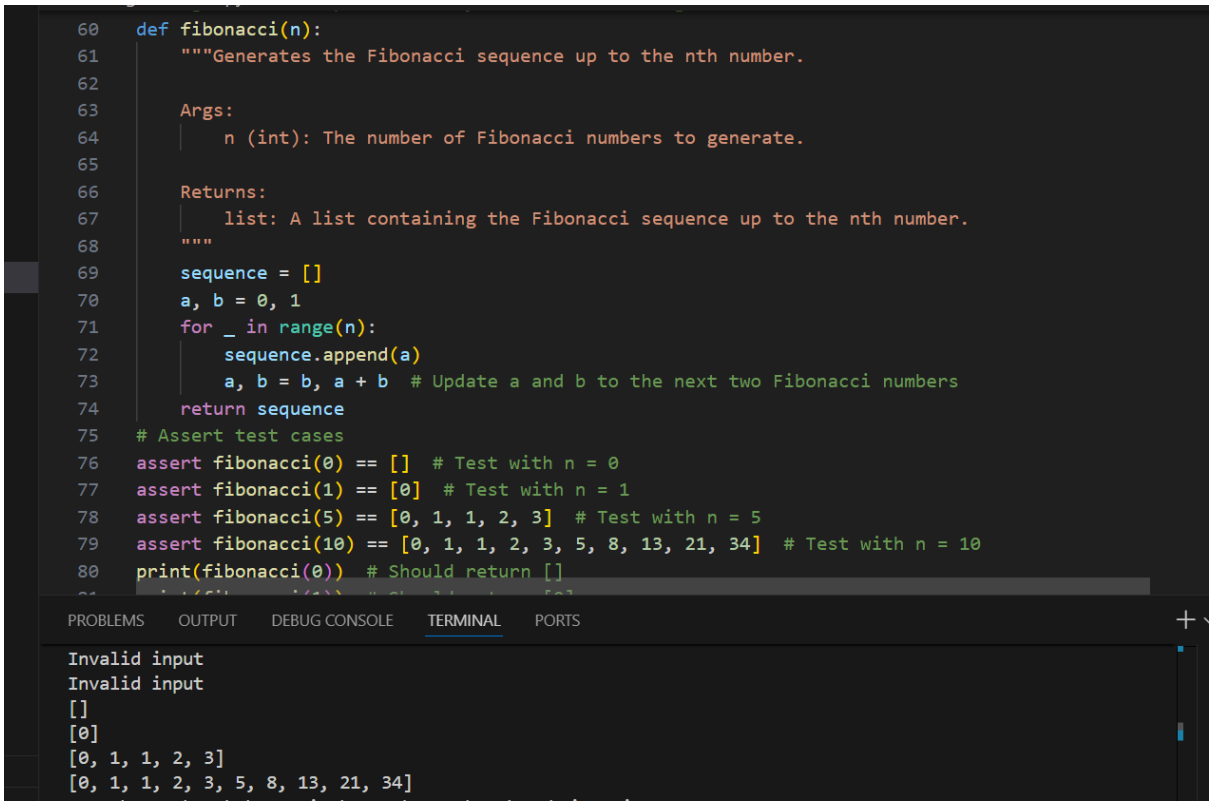
• Avoid commenting on trivial or self-explanatory syntax

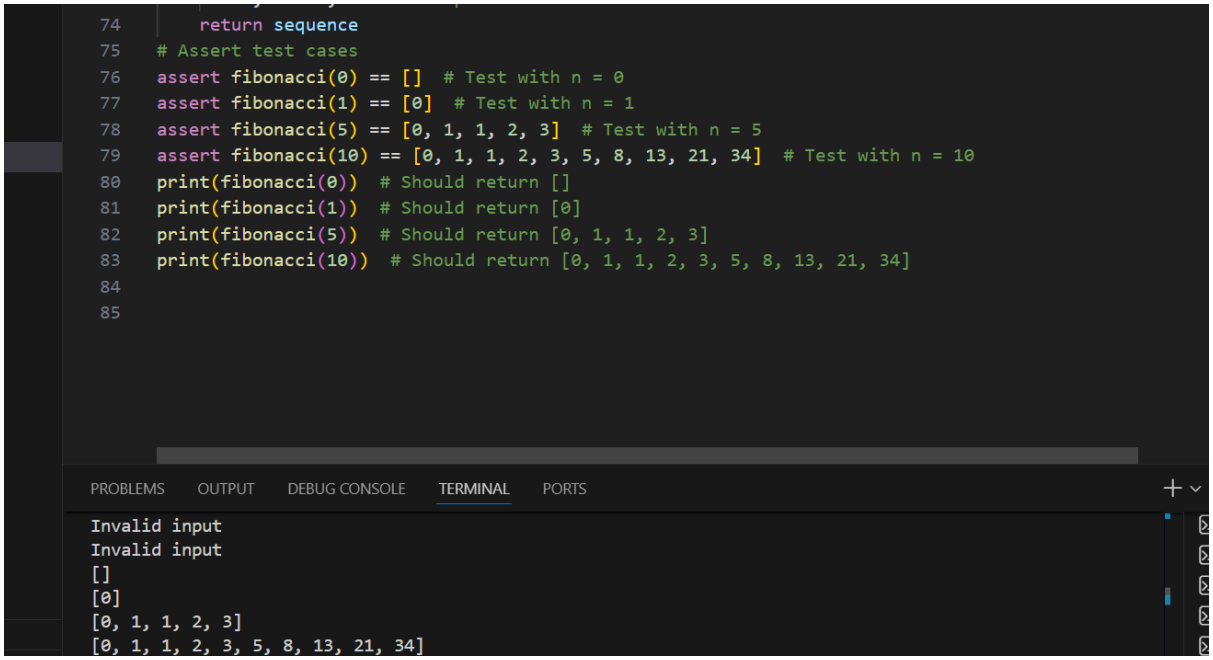
The goal is to improve clarity without cluttering the code.

Prompt:

Insert meaningful inline comments in the following Python code. Comment only complex or non-obvious logic and avoid explaining basic Python syntax

Code:





Observation

- The AI added comments only where the logic was not immediately obvious.
- The tuple update in the Fibonacci algorithm was clearly explained.
- Trivial lines like variable initialization and loop syntax were not commented.
- Code readability improved without adding unnecessary clutter.

Task 3: Generating Module-Level Documentation for a Python Package

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

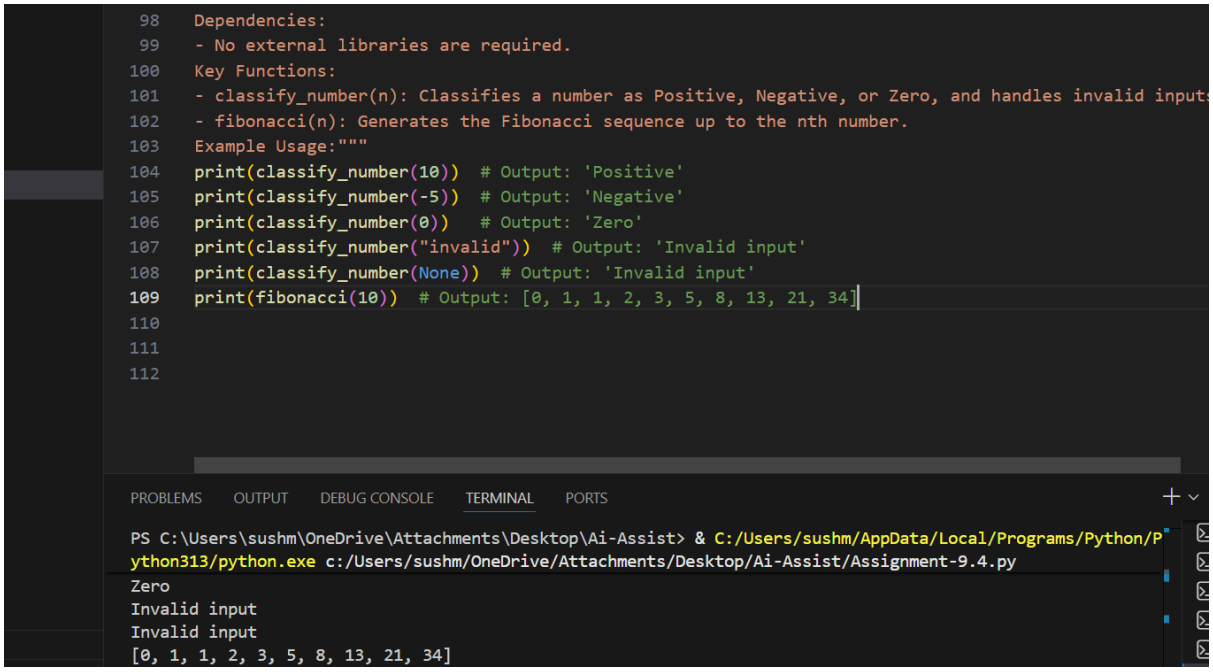
• A brief description of key functions and classes

• A short example of how the module can be used

Prompt:

Generate a professional module-level docstring for this Python module. Include the purpose, dependencies, key functions, and example usage

Code:



Observation

- The AI generated a structured multi-line docstring at the top of the file.
- The documentation clearly explained the module's purpose and functionality.
- Dependencies and key functions were summarized concisely.
- Example usage improved clarity for new users of the module.

- The tone was professional and suitable for real-world repositories.

Task 4: Converting Developer Comments into Structured Docstrings

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

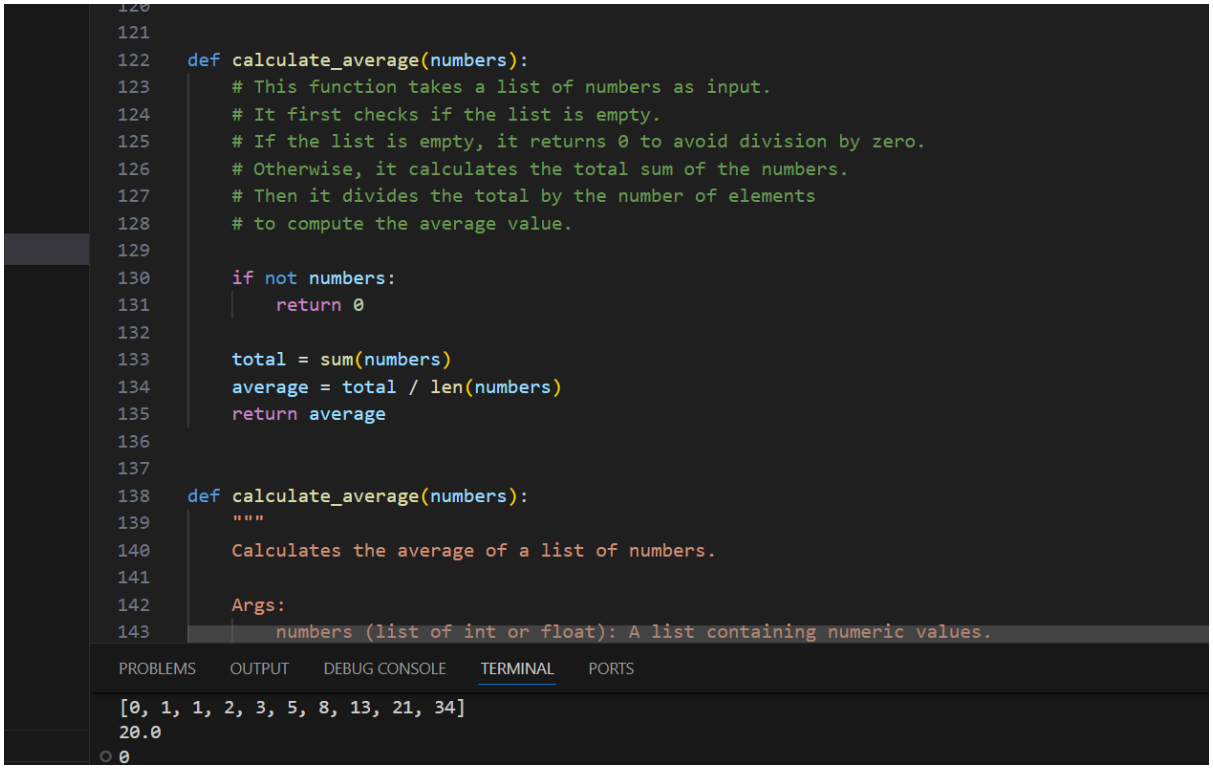
• Preserve the original meaning and intent of the comments

• Remove redundant inline comments after conversion

Prompt:

Convert the following inline explanatory comments into a structured Google-style docstring. Preserve the original meaning and remove redundant inline comments

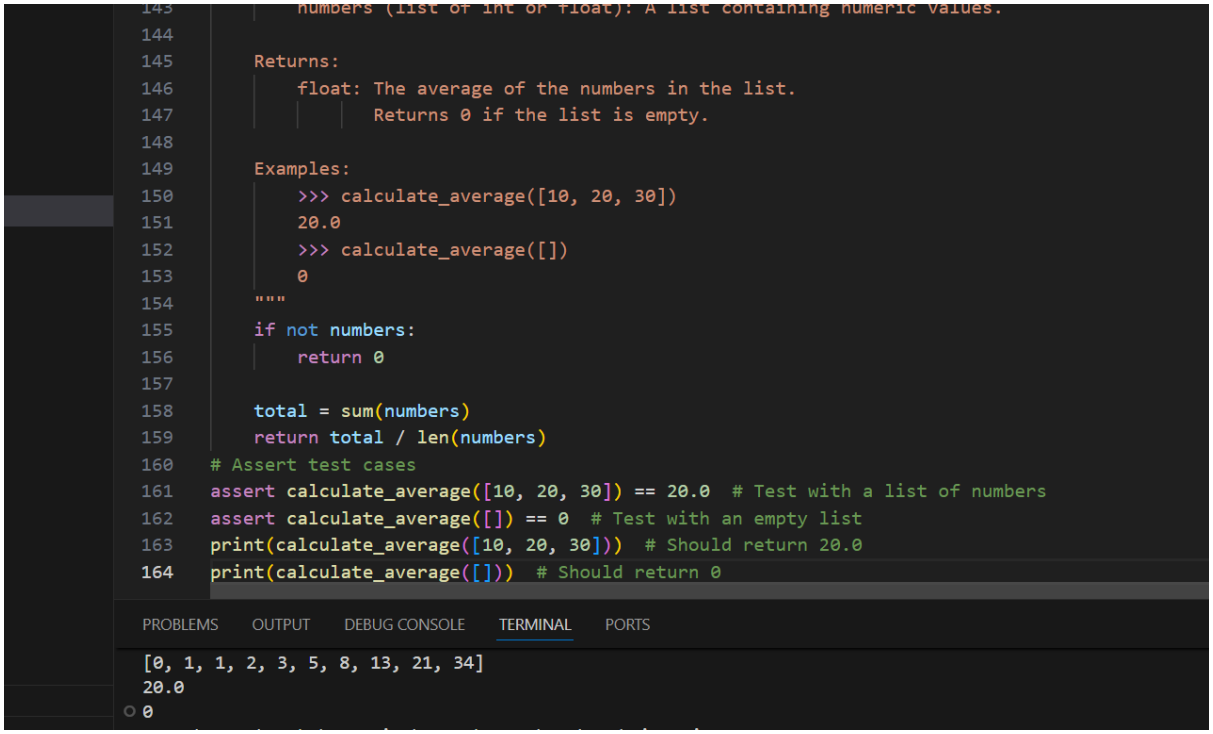
Code:



```
def calculate_average(numbers):
    # This function takes a list of numbers as input.
    # It first checks if the list is empty.
    # If the list is empty, it returns 0 to avoid division by zero.
    # Otherwise, it calculates the total sum of the numbers.
    # Then it divides the total by the number of elements
    # to compute the average value.

    if not numbers:
        return 0

    total = sum(numbers)
    average = total / len(numbers)
    return average


def calculate_average(numbers):
    """
    Calculates the average of a list of numbers.

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

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
20.0
0
```



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

    Returns:
        float: The average of the numbers in the list.
               Returns 0 if the list is empty.

    Examples:
        >>> calculate_average([10, 20, 30])
        20.0
        >>> calculate_average([])
        0
    """
    if not numbers:
        return 0

    total = sum(numbers)
    return total / len(numbers)
# Assert test cases
assert calculate_average([10, 20, 30]) == 20.0  # Test with a list of numbers
assert calculate_average([]) == 0  # Test with an empty list
print(calculate_average([10, 20, 30]))  # Should return 20.0
print(calculate_average([]))  # Should return 0
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
20.0
0
```

Observation:

- The AI successfully converted descriptive inline comments into a structured docstring.
- The original meaning of the logic was preserved.

- Redundant inline comments were removed.
- The function body became cleaner and more readable.
- Documentation format became consistent with professional standards.

Task 5: Building a Mini Automatic Documentation Generator

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

Prompt:

Help me build a Python utility that reads a .py file, detects functions and classes using AST, and inserts placeholder Google-style docstrings where missing

Code:

```python

#Design a small Python utility that:
#• Reads a given .py file
#• Automatically detects:
#o Functions
#o Classes
#• Inserts placeholder Google-style docstrings for each detected function or class
import ast
def insert_docstrings(file_path):
    with open(file_path, 'r') as file:
        source = file.read()
    tree = ast.parse(source)
    for node in ast.walk(tree):
        if isinstance(node, (ast.FunctionDef, ast.ClassDef)):
            if not ast.get_docstring(node):
                docstring = f"TODO: Add Google-style docstring for {node.name}."
                node.body.insert(
                    0,
                    ast.Expr(value=ast.Constant(value=docstring))
                )
    new_source = ast.unparse(tree)
    with open(file_path, 'w') as file:
        file.write(new_source)

```

Observation:

- AI suggested using the ast module to parse and modify the syntax tree.
- Functions and classes were automatically detected.
- Placeholder docstrings were inserted only where missing.
- The tool successfully scaffolded documentation without affecting existing docstrings.
- Demonstrates automation of documentation setup.