

LAB 9 – Documentation Generation

Course: AI Assisted Coding

Week 5

Name: G. Karthikeya

BATCH-14

2303A510D2

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:

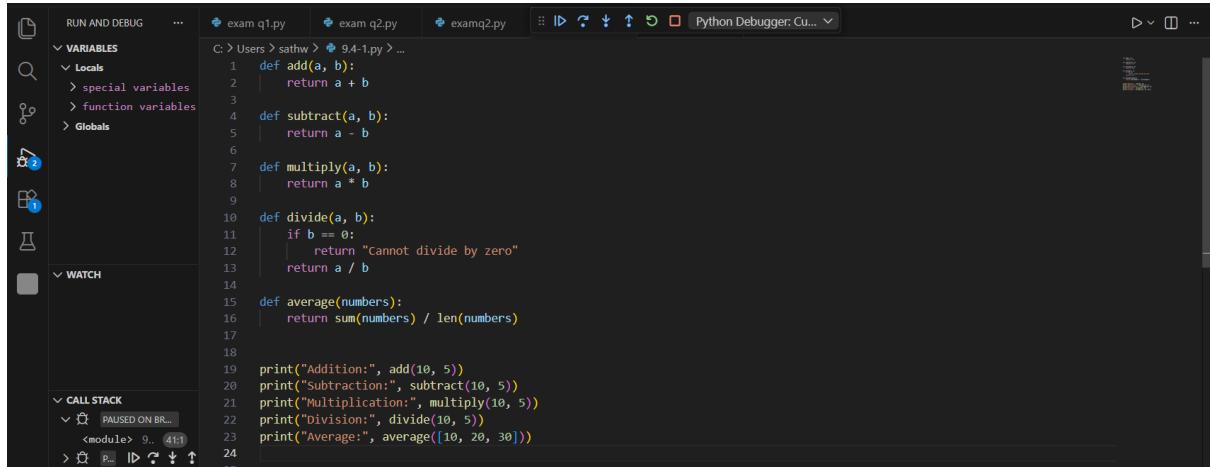
- Ask the AI to automatically generate Google-style function docstrings for each function
 - Each docstring should include:
 - A brief description of the function
 - Parameters with data types
 - Return values
 - At least one example usage (if applicable)

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

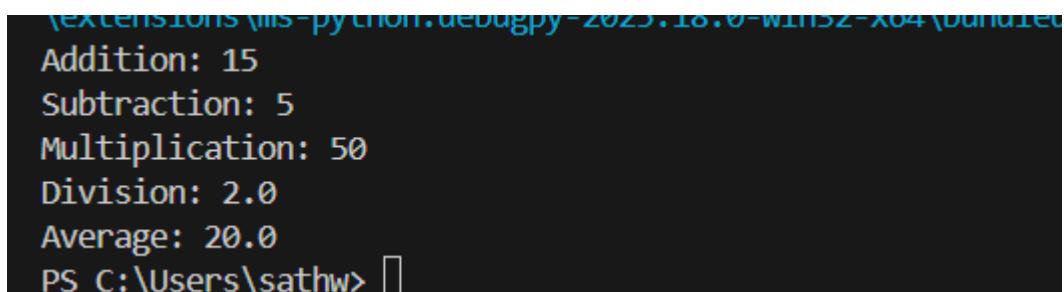


The screenshot shows the Python Debugger interface in Visual Studio Code. The code editor displays a script named `examq1.py` with the following content:

```
C:\Users> sathw > 9.4-1.py > ...
1 def add(a, b):
2     return a + b
3
4 def subtract(a, b):
5     return a - b
6
7 def multiply(a, b):
8     return a * b
9
10 def divide(a, b):
11     if b == 0:
12         return "Cannot divide by zero"
13     return a / b
14
15 def average(numbers):
16     return sum(numbers) / len(numbers)
17
18
19 print("Addition:", add(10, 5))
20 print("Subtraction:", subtract(10, 5))
21 print("Multiplication:", multiply(10, 5))
22 print("Division:", divide(10, 5))
23 print("Average:", average([10, 20, 30]))
```

The left sidebar shows the debugger's state with sections for VARIABLES, WATCH, and CALL STACK. The CALL STACK section indicates the script is PAUSED ON BREAK at line 11, column 11. The status bar at the bottom shows the path `C:\Users\... 41:1`.

#OUTPUT:



```
PS C:\Users\sathw> 
Addition: 15
Subtraction: 5
Multiplication: 50
Division: 2.0
Average: 20.0
PS C:\Users\sathw>
```

#prompt:

Generate Google-style docstrings for the following Python functions.
Include: Short description Parameters with data types Return type
Example usage.

The screenshot shows the VS Code interface with the Python extension. The code editor displays a script with several mathematical functions: add, subtract, multiply, divide, and average. The 'RUN AND DEBUG' sidebar on the left shows variables, watch, call stack, and breakpoints. Breakpoints are set at lines 6, 13, 20, 31, and 36. The status bar at the bottom indicates the file is 9.4.1.py, line 41, column 1, with 4 spaces, using UTF-8 encoding.

```

1 #Generate Google-style docstrings for the following Python functions. Include:Short description Parameters with data types Return value
2 def add(a, b):
3     """
4     Adds two numbers.
5     """
6     return a + b
7
8 def subtract(a, b):
9     """
10    Subtracts the second number from the first number.
11    """
12    return a - b
13
14 def multiply(a, b):
15     """
16     Multiplies two numbers.
17     """
18    return a * b
19
20 def divide(a, b):
21     """
22     Divides the first number by the second number.
23     """
24    if b == 0:
25        return "Cannot divide by zero"
26    return a / b
27
28 def average(numbers):
29     """
30     calculates the average of a list of numbers.
31     """
32    return sum(numbers) / len(numbers)

```

This screenshot shows the code editor with user input being entered into the script. The line `a = float(input("Enter first number: "))` is highlighted in yellow, indicating it is currently being executed. The rest of the script is shown below, with line numbers 39 through 56.

```

39 # ----- USER INPUT SECTION -----
40
41 a = float(input("Enter first number: "))
42 b = float(input("Enter second number: "))
43
44 print("Addition:", add(a, b))
45 print("Subtraction:", subtract(a, b))
46 print("Multiplication:", multiply(a, b))
47 print("Division:", divide(a, b))
48
49 # Taking list input for average
50 nums = input("Enter numbers for average separated by space: ")
51 num_list = list(map(float, nums.split()))
52
53 print("Average:", average(num_list))
54
55
56

```

#Output

The terminal window shows the execution of the script and its output. The user enters two numbers, 20 and 10, and the script prints their addition, subtraction, multiplication, division, and average.

```

PS C:\Users\sathw> & 'c:\Users\sathw\AppData\Local\Microsoft\WindowsApps\python3.13.exe' 'c:\Users\sathw\.vscode\extensions\ms-python.python\2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '63520' '--' 'c:\Users\sathw\9.4.1.py'
Enter first number: 20
Enter second number: 10
Addition: 30.0
Subtraction: 10.0
Multiplication: 200.0
Division: 2.0
Enter numbers for average separated by space: 10 20 30 40
Average: 25.0
PS C:\Users\sathw>

```

EXPLANATION:

In this task, we created mathematical functions like addition, subtraction, multiplication, division, and average.

Initially, the functions had no documentation.

We used an AI tool to automatically generate **Google-style docstrings**.

The docstrings include:

- Function description
- Parameters
- Return values
- Example usage

This improves code readability and makes the program easier for other developers to understand and maintain.

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

Original Code (Without Comments)

The screenshot shows the Python Debugger interface in VS Code. The code editor displays the following Python script:

```
C: > Users > sathw > 9.4.py > fibonacci
1 def fibonacci(n):
2     if n <= 0:
3         return []
4
5     if n == 1:
6         return [0]
7
8     sequence = [0, 1]
9
10    for i in range(2, n):
11        sequence.append(sequence[-1] + sequence[-2])
12
13    return sequence
14
15 print(fibonacci(7))
16
17
18
19
20
21
```

#OUTPUT:

```
\extensions\ms-python.debugpy-
[0, 1, 1, 2, 3, 5, 8]
PS C:\Users\sathw> 
```

AI-Generated Inline Comments (Improved Version)

#prompt: Improve the readability of the following Fibonacci program by adding meaningful inline comments. Only comment complex or non-obvious logic. Do not explain basic Python syntax.

The screenshot shows the Python Debugger interface in VS Code. The code editor displays the following Python script with inline comments:

```
C: > Users > sathw > 9.4-1.py > ...
1 #Improve the readability of the following Fibonacci program by adding meaningful inline comments.Only comment complex or non-o...
2 def fibonacci(n):
3     if n <= 0:
4         return []
5
6     if n == 1:
7         return [0]
8
9     sequence = [0, 1]
10    for i in range(2, n):
11        sequence.append(sequence[-1] + sequence[-2])
12
13    return sequence
14
15 print(fibonacci(7))
16
17
18
19
20
21
```

#OUTPUT:

```
\extensions\ms-python.debugpy
[0, 1, 1, 2, 3, 5, 8]
PS C:\Users\sathw> 
```

EXPLANATION

In this task, AI was used to add meaningful inline comments to a Python program containing loops and an algorithm (Fibonacci).

Only complex logic was commented, and basic syntax was not explained.

This improved code readability and made the program easier to understand and maintain.

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:

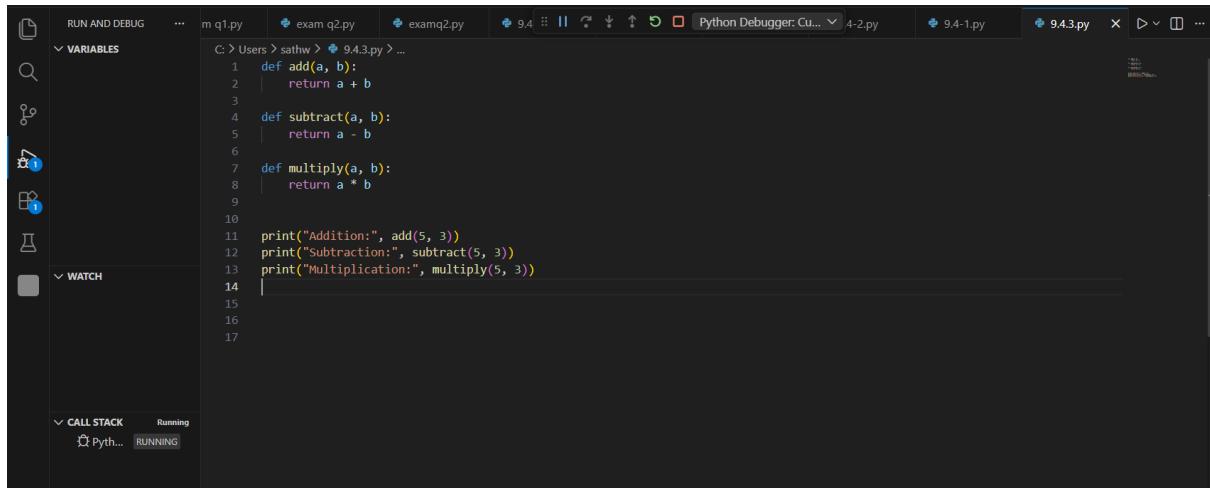
- The purpose of the module
- Required libraries or dependencies
- A brief description of key functions and classes
- A short example of how the module can be used

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

#CODE AND INPUT



```
C:\Users>sathw> 9.4.3.py > ...
1 def add(a, b):
2     return a + b
3
4 def subtract(a, b):
5     return a - b
6
7 def multiply(a, b):
8     return a * b
9
10
11 print("Addition:", add(5, 3))
12 print("Subtraction:", subtract(5, 3))
13 print("Multiplication:", multiply(5, 3))
14
15
16
17
```

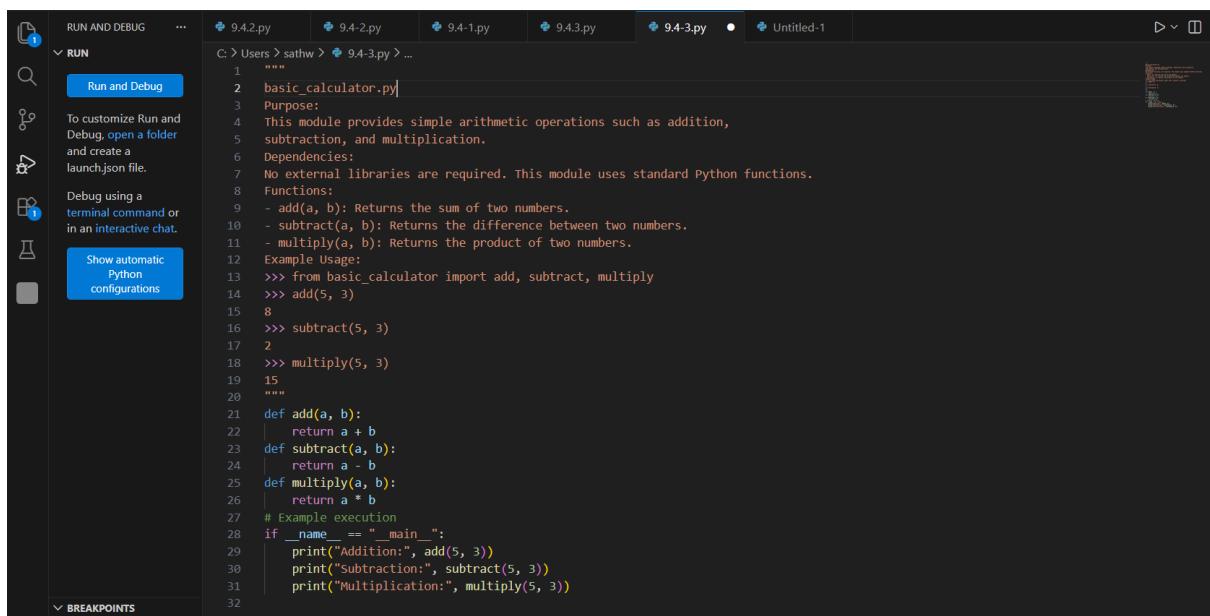
#OUTPUT:

```
PS C:\Users\sathw> cd 'c:\Users\sathw'; & 'c:\Users\sathw\AppData\Local\Microsoft\WindowsApps\python3.13.exe' 'c:\Users\sathw\vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '65350' '--' 'C:\Users\sathw\9.4.3.py'
Addition: 8
Subtraction: 2
Multiplication: 15
PS C:\Users\sathw>
```

AI Generates Module-Level Docstring:

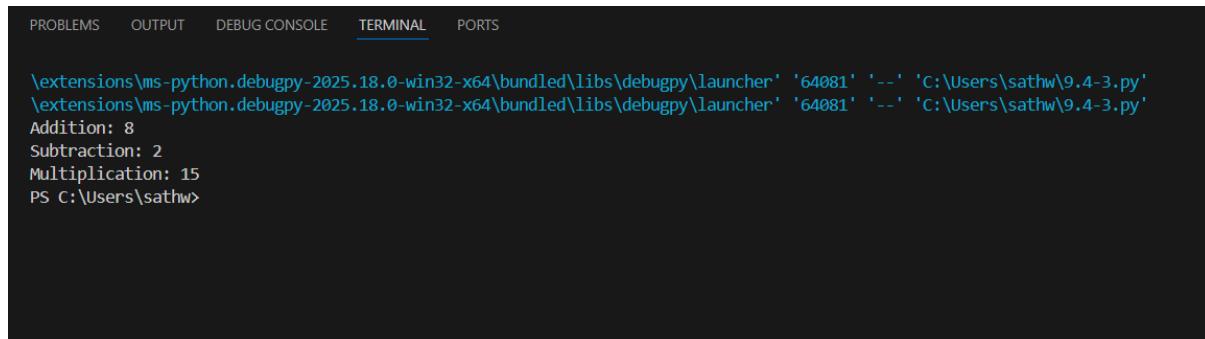
Generate a professional module-level docstring for this Python file that explains the module purpose, dependencies, key functions, and includes a short example usage, without changing the existing code.

#CODE AND INPUT:



```
'''basic calculator.py
Purpose:
This module provides simple arithmetic operations such as addition,
subtraction, and multiplication.
Dependencies:
No external libraries are required. This module uses standard Python functions.
Functions:
- add(a, b): Returns the sum of two numbers.
- subtract(a, b): Returns the difference between two numbers.
- multiply(a, b): Returns the product of two numbers.
Example Usage:
>>> from basic_calculator import add, subtract, multiply
>>> add(5, 3)
8
>>> subtract(5, 3)
2
>>> multiply(5, 3)
15
'''
def add(a, b):
    return a + b
def subtract(a, b):
    return a - b
def multiply(a, b):
    return a * b
# Example execution
if __name__ == '__main__':
    print("Addition:", add(5, 3))
    print("Subtraction:", subtract(5, 3))
    print("Multiplication:", multiply(5, 3))
```

#OUTPUT:



A screenshot of a terminal window from a code editor. The tabs at the top are PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is underlined), and PORTS. The terminal output shows the following Python code execution:

```
\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '64081' '--' 'C:\Users\sathw\9.4-3.py'
\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '64081' '--' 'C:\Users\sathw\9.4-3.py'
Addition: 8
Subtraction: 2
Multiplication: 15
PS C:\Users\sathw>
```

Task 3 is about **adding proper documentation at the top of a Python file**.

The goal is:

- To explain **what the module does**
- To mention **required libraries (if any)**
- To describe **main functions**
- To show **how to use it**

So when someone opens the file, they immediately understand its purpose without reading all the code.

☞ In short:

Task 3 improves professionalism and clarity by adding a clear module-level docstring.

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.

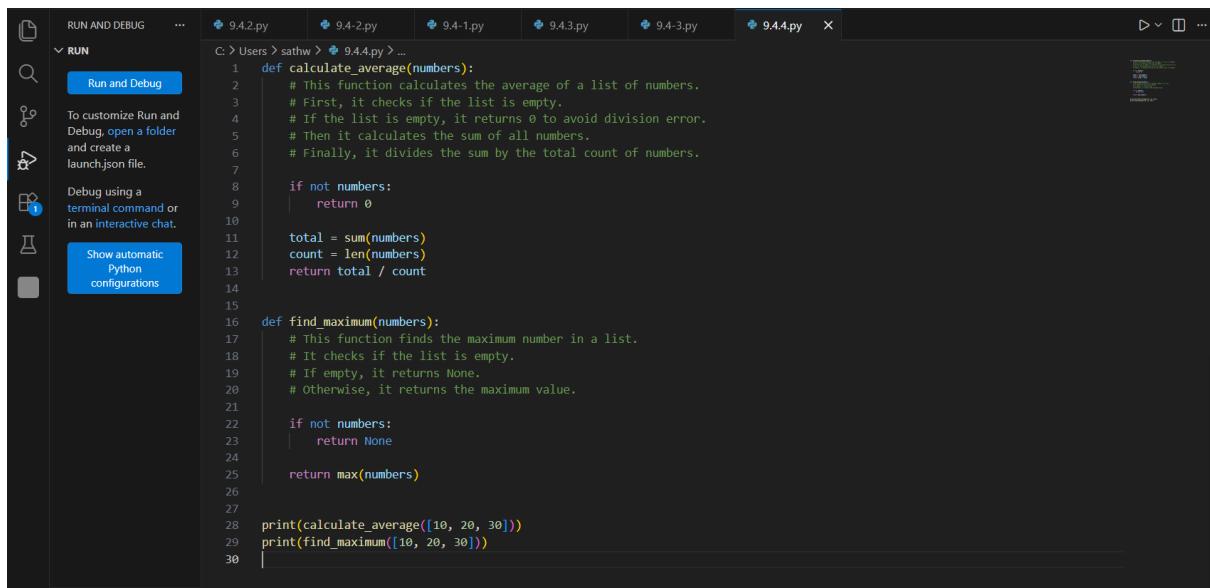
Use AI to:

- 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

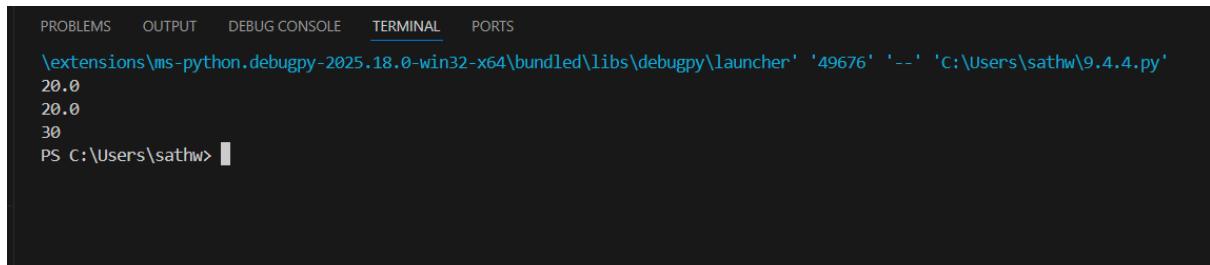
#CODE AND INPUT:



The screenshot shows a code editor interface with a dark theme. On the left, there's a sidebar with icons for file operations, search, and run/debug. The main area displays a Python script with two functions: `calculate_average` and `find_maximum`. The code is well-commented, explaining its purpose and logic.

```
C: > Users > sathw > 9.4.4.py > ...
1 def calculate_average(numbers):
2     # This function calculates the average of a list of numbers.
3     # First, it checks if the list is empty.
4     # If the list is empty, it returns 0 to avoid division error.
5     # Then it calculates the sum of all numbers.
6     # Finally, it divides the sum by the total count of numbers.
7
8     if not numbers:
9         return 0
10
11    total = sum(numbers)
12    count = len(numbers)
13    return total / count
14
15
16 def find_maximum(numbers):
17     # This function finds the maximum number in a list.
18     # It checks if the list is empty.
19     # If empty, it returns None.
20     # Otherwise, it returns the maximum value.
21
22     if not numbers:
23         return None
24
25     return max(numbers)
26
27
28 print(calculate_average([10, 20, 30]))
29 print(find_maximum([10, 20, 30]))
30
```

#OUTPUT:



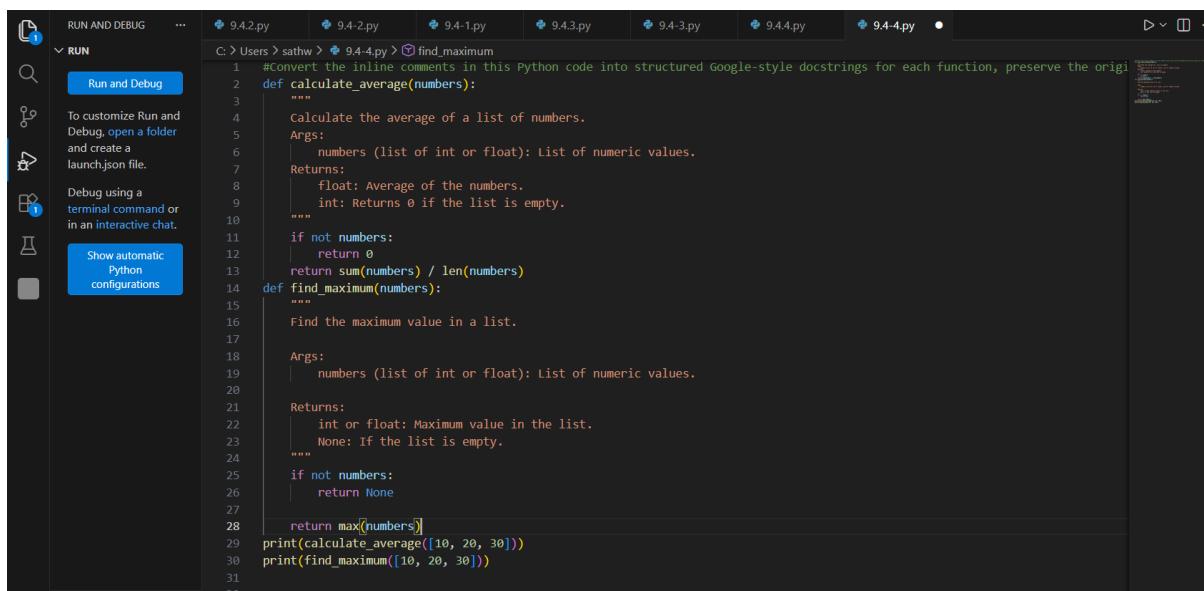
The screenshot shows a terminal window with a dark theme. It displays the command `python 9.4.4.py` being run, followed by the output of the two printed statements from the script. The output shows the average of 20.0 and the maximum of 30.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '49676' '--' 'C:\Users\sathw\9.4.4.py'
20.0
20.0
30
PS C:\Users\sathw>
```

Converted Version:

#PROMPT:

Convert the inline comments in this Python code into structured Google-style docstrings for each function, preserve the original meaning, remove redundant comments inside the function body, and do not change the program logic.



The screenshot shows the Visual Studio Code interface with the Python extension installed. The left sidebar has 'RUN AND DEBUG' selected under 'RUN'. A tooltip for 'Run and Debug' says: 'To customize Run and Debug, open a folder and create a launch.json file.' Below it, another tooltip says: 'Debug using a terminal command or in an interactive chat.' A button labeled 'Show automatic Python configurations' is also visible. The main editor area contains the following Python code with docstrings:

```
C:\> Users > sathw > 9.4-4.py > find_maximum
#Convert the inline comments in this Python code into structured Google-style docstrings for each function, preserve the original meaning, remove redundant comments inside the function body, and do not change the program logic.

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

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

    Returns:
        float: Average of the numbers.
        int: Returns 0 if the list is empty.
    """
    if not numbers:
        return 0
    return sum(numbers) / len(numbers)

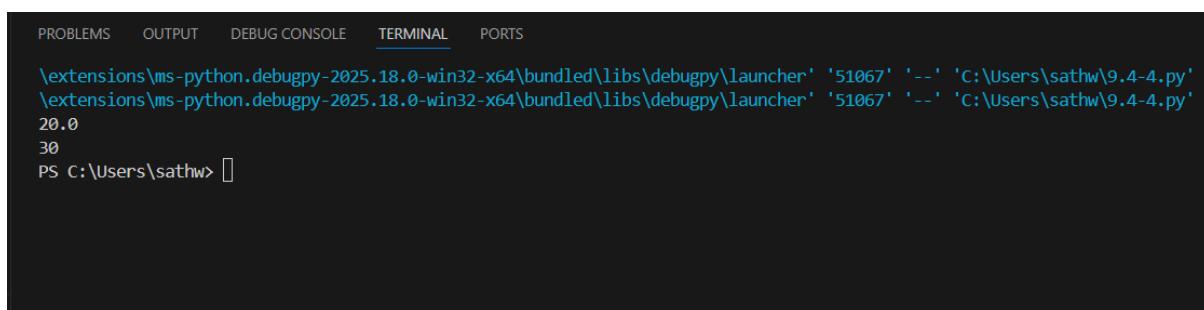
def find_maximum(numbers):
    """
    Find the maximum value in a list.

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

    Returns:
        int or float: Maximum value in the list.
        None: If the list is empty.
    """
    if not numbers:
        return None
    return max(numbers)

print(calculate_average([10, 20, 30]))
print(find_maximum([10, 20, 30]))
```

#OUTPUT:



The screenshot shows the VS Code terminal tab. The output shows the execution of the converted Python code. The code defines two functions: `calculate_average` and `find_maximum`. Both functions use Google-style docstrings. When run, the code prints the average and maximum of the list [10, 20, 30].

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '51067' '--' 'C:\Users\sathw\9.4-4.py'
extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '51067' '--' 'C:\Users\sathw\9.4-4.py'
20.0
30
PS C:\Users\sathw> 
```

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.

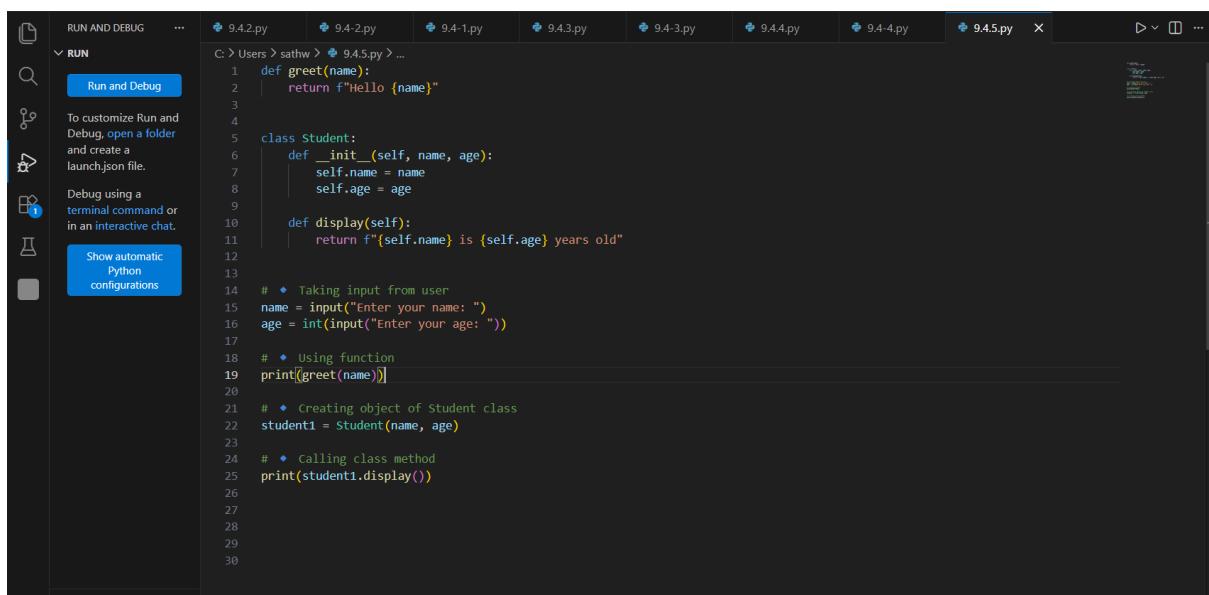
Task Description

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.

#CODE AND INPUT:



```
C:\Users\sathw> 9.4.5.py > ...
1 def greet(name):
2     return f"Hello {name}"
3
4
5 class Student:
6     def __init__(self, name, age):
7         self.name = name
8         self.age = age
9
10    def display(self):
11        return f"{self.name} is {self.age} years old"
12
13
14 # • Taking input from user
15 name = input("Enter your name: ")
16 age = int(input("Enter your age: "))
17
18 # • Using function
19 print(greet(name))
20
21 # • Creating object of Student class
22 student1 = Student(name, age)
23
24 # • Calling class method
25 print(student1.display())
26
27
28
29
30
```

#OUTPUT:

```
Enter your name: sathwik
Enter your age: 21
Hello sathwik
sathwik is 21 years old
PS C:\Users\sathw> []
```

#PROMPT:

Generate a professional Google-style documentation for this Python program, improve code readability if needed, and ensure proper input validation and structured main() usage without changing the core functionality.

```
C:\> Users > sathw > 9.4-5.py > ...
1 #Generate a professional Google-style documentation for this Python program, improve code readability if needed, and ensure pr...
2 def greet(name):
3     """Return a greeting message for the given name."""
4     return f"Hello, {name}! Welcome."
5
6 class Student:
7     """Represents a student with name and age."""
8
9     def __init__(self, name, age):
10        self.name = name
11        self.age = age
12
13    def display(self):
14        """Return formatted student details."""
15        return f"{self.name} is {self.age} years old."
16
17 def main():
18     """Main function to take user input and display results."""
19     name = input("Enter your name: ").strip()
20     # Age validation
21     while True:
22         try:
23             age = int(input("Enter your age: "))
24             if age <= 0:
25                 print("Age must be positive.")
26                 continue
27             break
28         except ValueError:
29             print("Please enter a valid number.")
30
31     print(greet(name))
32     student = Student(name, age)
33     print(student.display())
34
35 if __name__ == "__main__":
36     main()
```

#OUTPUT:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Enter your name: sathwik Reddy
Enter your name: sathwik Reddy
Enter your age: 21
Enter your age: 21
Hello, sathwik Reddy! Welcome.
Hello, sathwik Reddy! Welcome.
sathwik Reddy is 21 years old.
PS C:\Users\sathw> 
```

#EXPLANATION:

This program:

- Takes name and age from the user

- Validates age to ensure it is a positive number
- Uses a greet() function to display a welcome message
- Uses a Student class to store and display student details
- Uses a main() function to organize the program properly

☞ In short:

It is a well-structured, validated, and professional version of a simple greeting and student information program.