

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:

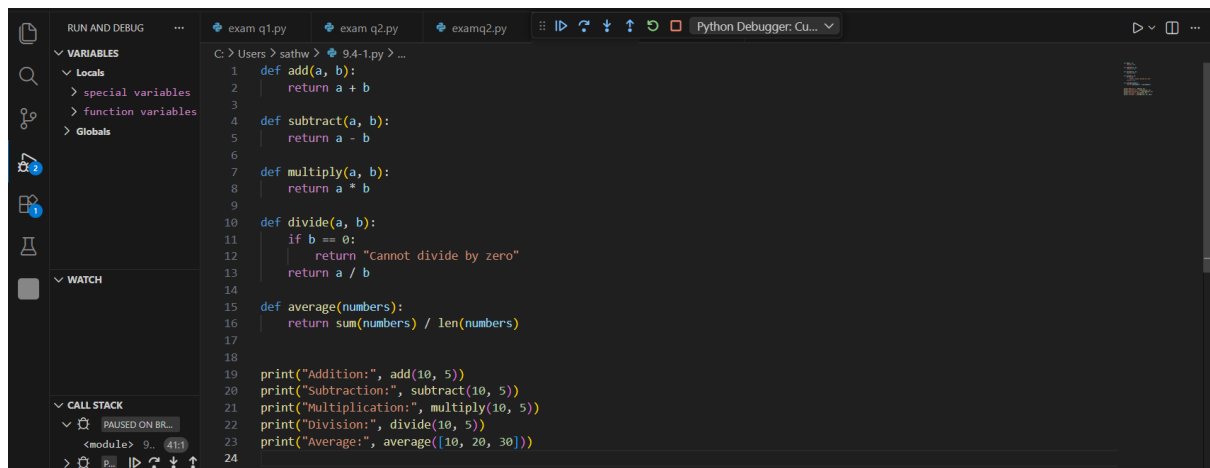
- Ask the AI to automatically generate Google-style function docstrings for each function
- Each docstring should include:
 - o A brief description of the function
 - o Parameters with data types
 - o Return values
 - o At least one example usage (if applicable)

Experiment with different prompting styles (zero-shot or context-based)

to observe quality differences.

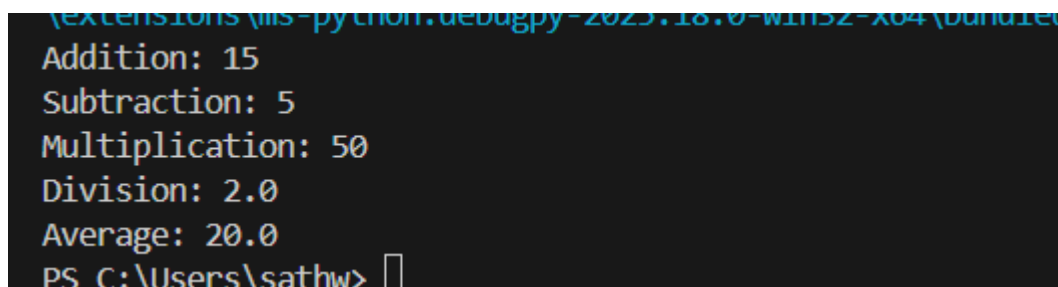
Expected Outcome

- A Python script with well-structured Google-style docstrings
- Docstrings that clearly explain function behavior and usage
- Improved readability and usability of the codebase



```
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]))
24
```

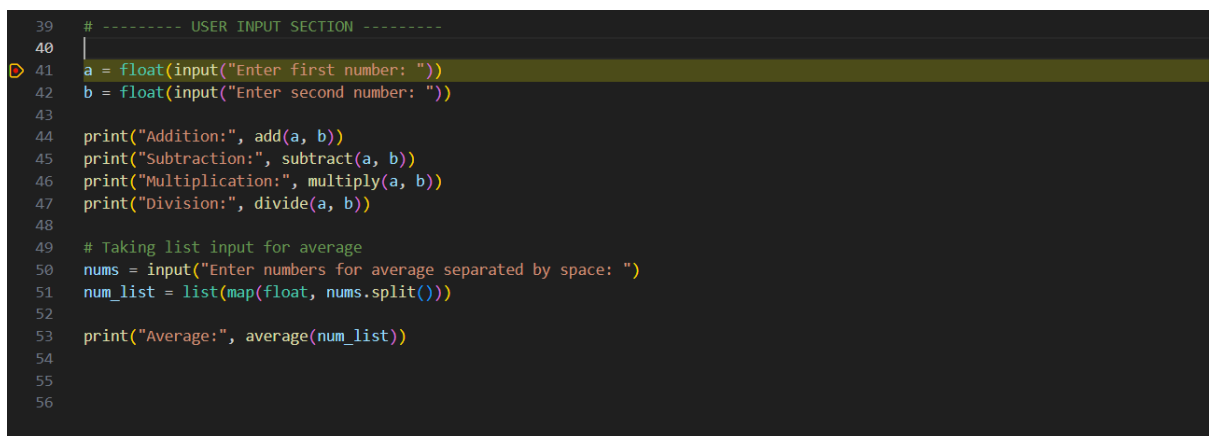
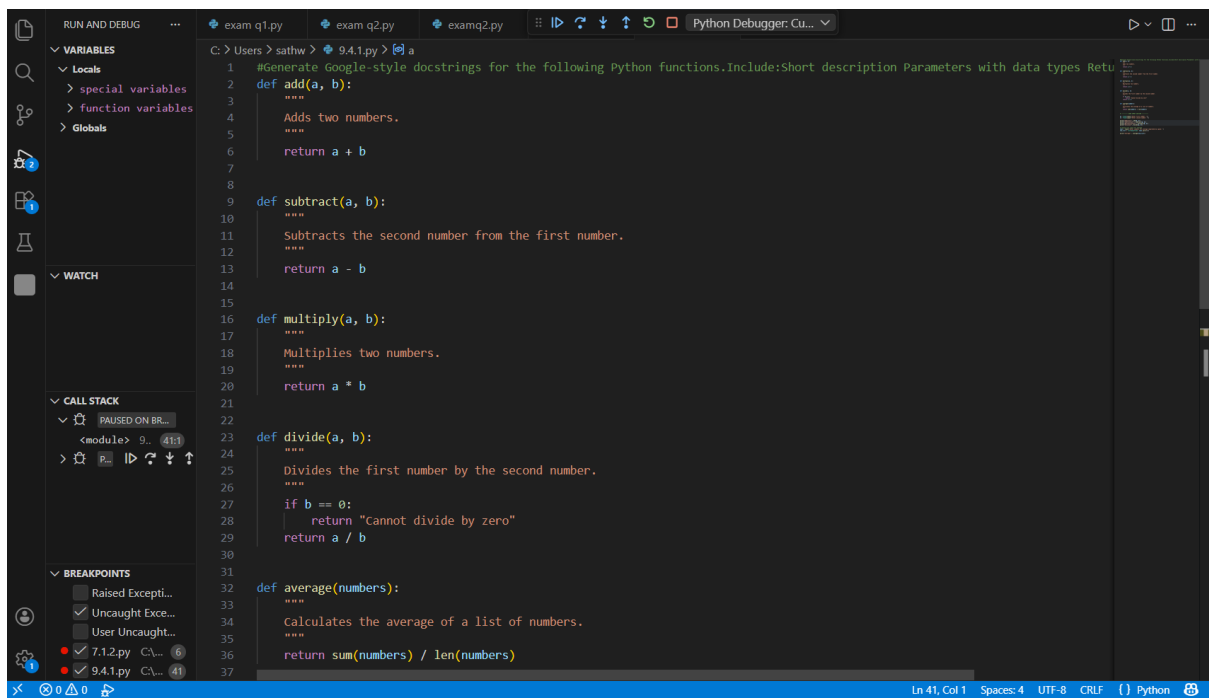
#OUTPUT:



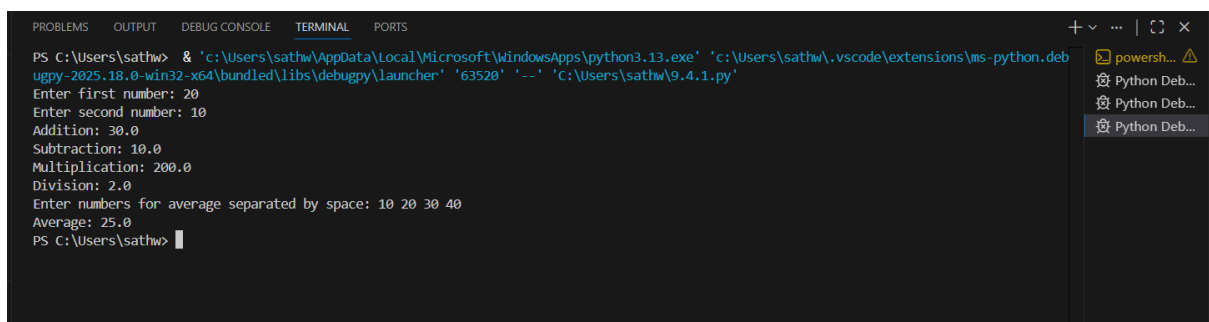
```
extensions\ms-python.debugpy-2023.18.0-win32-x64\bin\python.exe
Addition: 15
Subtraction: 5
Multiplication: 50
Division: 2.0
Average: 20.0
PS C:\Users\sathw>
```

#prompt:

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



#Output



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.

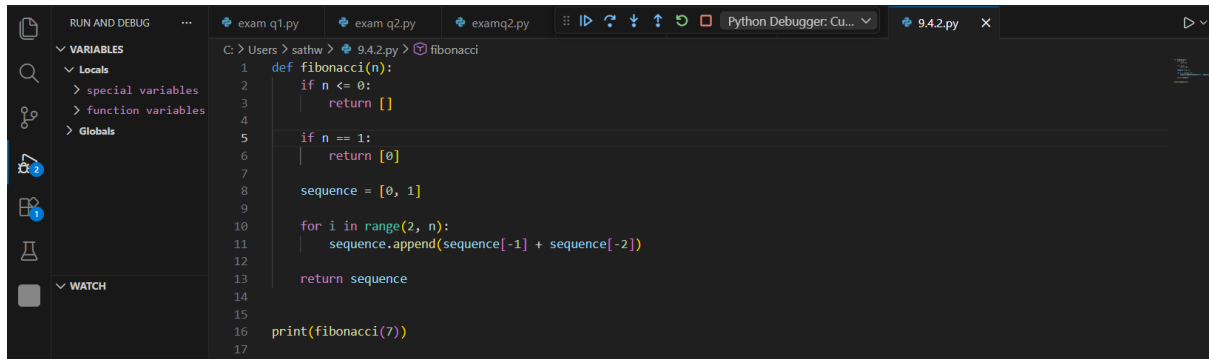
Expected Outcome

- A Python script with concise, meaningful inline comments
- Comments that explain why the logic exists, not what Python

syntax does

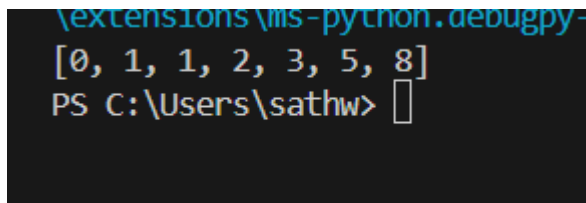
- Noticeable improvement in code readability

Original Code (Without Comments)



```
C:\Users> sathw > 9.4.2.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
16    print(fibonacci(7))
17
```

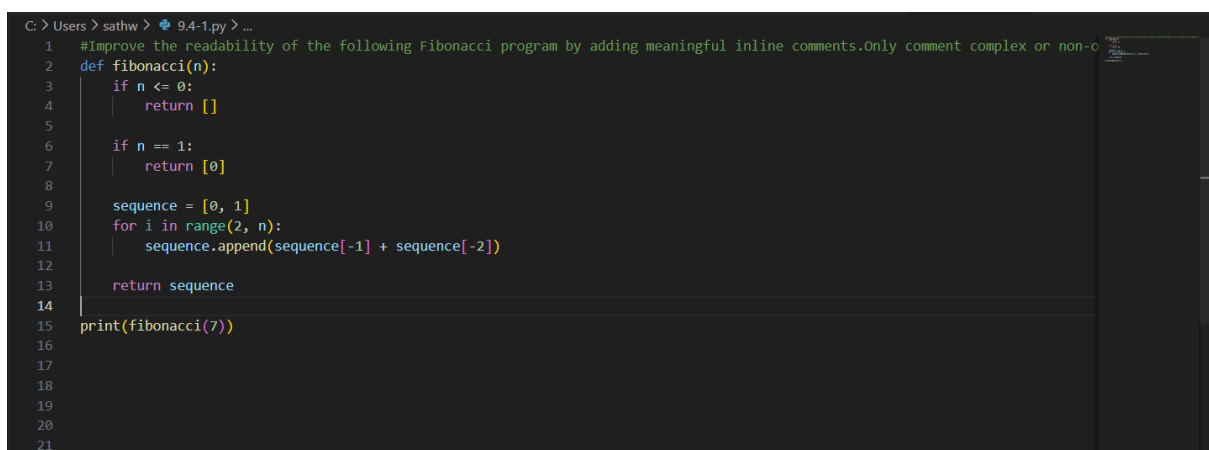
#OUTPUT:



```
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.



```
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:

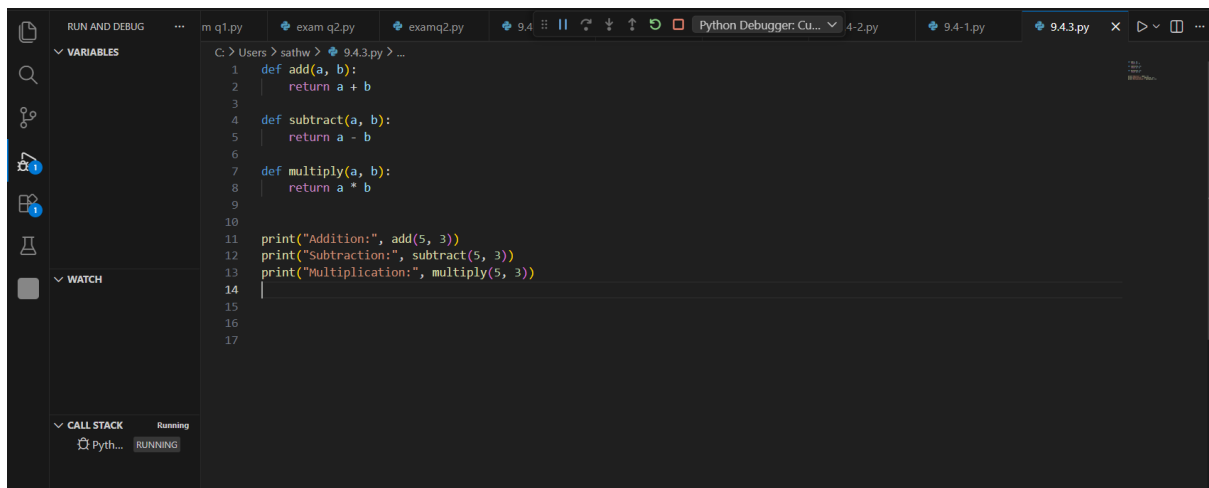
- The purpose of the module
- Required libraries or dependencies
- A brief description of key functions and classes
- A short example of how the module can be used

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

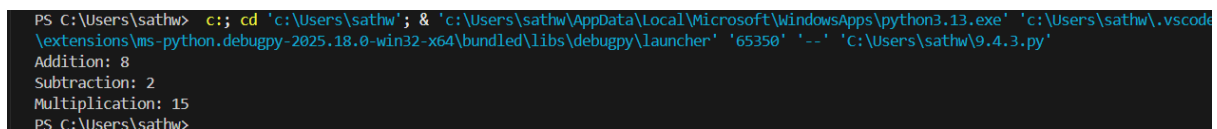
#CODE AND INPUT



The screenshot shows the Visual Studio Code editor with a Python file named 9.4.3.py. The code defines three functions: add, subtract, and multiply. It then prints the results of these functions with arguments 5 and 3. The interface includes a sidebar with icons for Explorer, Search, Run and Debug, and Extensions. The bottom status bar indicates the Python Debugger is running.

```
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:



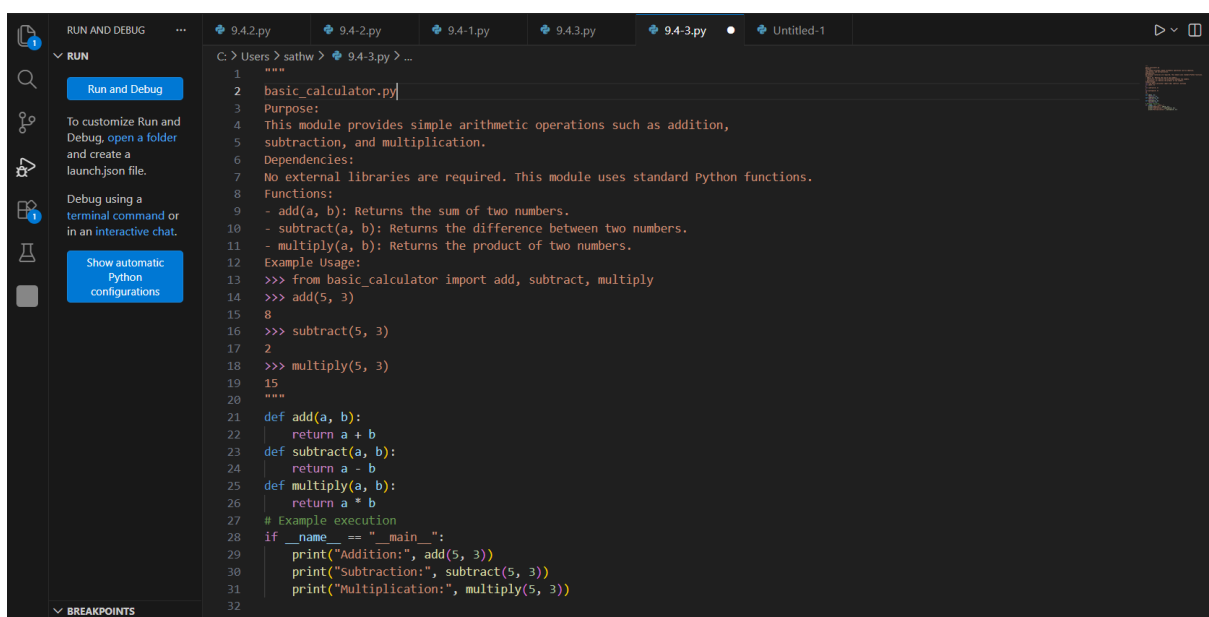
The screenshot shows a terminal window with the command prompt. The command executed is 'cd 'c:\Users\sathw'; & 'c:\Users\sathw\AppData\Local\Microsoft\WindowsApps\python3.13.exe' 'c:\Users\sathw\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundle\libs\debugpy\launcher' '65350' '--' 'C:\Users\sathw\9.4.3.py'. The output shows the results of the calculator functions: Addition: 8, Subtraction: 2, and Multiplication: 15.

```
PS C:\Users\sathw> c:: 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\bundle\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:



The screenshot shows the Visual Studio Code editor with a Python file named 9.4.3.py. The code includes a module-level docstring that describes the purpose, dependencies, and functions of the module. It also includes an example usage section. The code defines three functions: add, subtract, and multiply. The interface includes a sidebar with icons for Explorer, Search, Run and Debug, and Extensions. The bottom status bar indicates the Python Debugger is running.

```
C:\Users\sathw> 9.4.3.py > ...
1  """
2  basic_calculator.py
3  Purpose:
4  This module provides simple arithmetic operations such as addition,
5  subtraction, and multiplication.
6  Dependencies:
7  No external libraries are required. This module uses standard Python functions.
8  Functions:
9  - add(a, b): Returns the sum of two numbers.
10 - subtract(a, b): Returns the difference between two numbers.
11 - multiply(a, b): Returns the product of two numbers.
12 Example Usage:
13 >>> from basic_calculator import add, subtract, multiply
14 >>> add(5, 3)
15 8
16 >>> subtract(5, 3)
17 2
18 >>> multiply(5, 3)
19 15
20 """
21 def add(a, b):
22     return a + b
23 def subtract(a, b):
24     return a - b
25 def multiply(a, b):
26     return a * b
27 # Example execution
28 if __name__ == "__main__":
29     print("Addition:", add(5, 3))
30     print("Subtraction:", subtract(5, 3))
31     print("Multiplication:", multiply(5, 3))
32
33
```

#OUTPUT:

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.

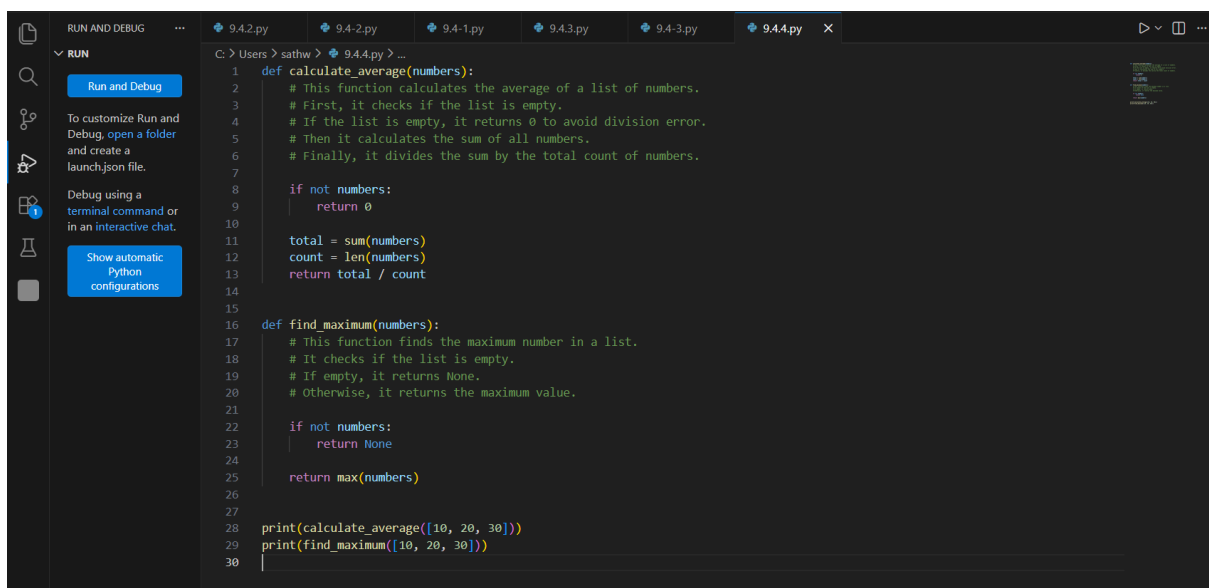
Use AI to:

- 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

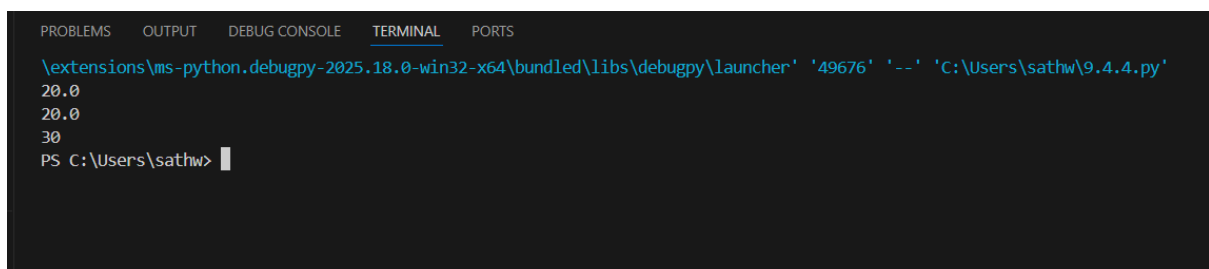
- Functions with clean, standardized docstrings
- Reduced clutter inside function bodies
- Improved consistency across the codebase

#CODE AND INPUT:



```
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:

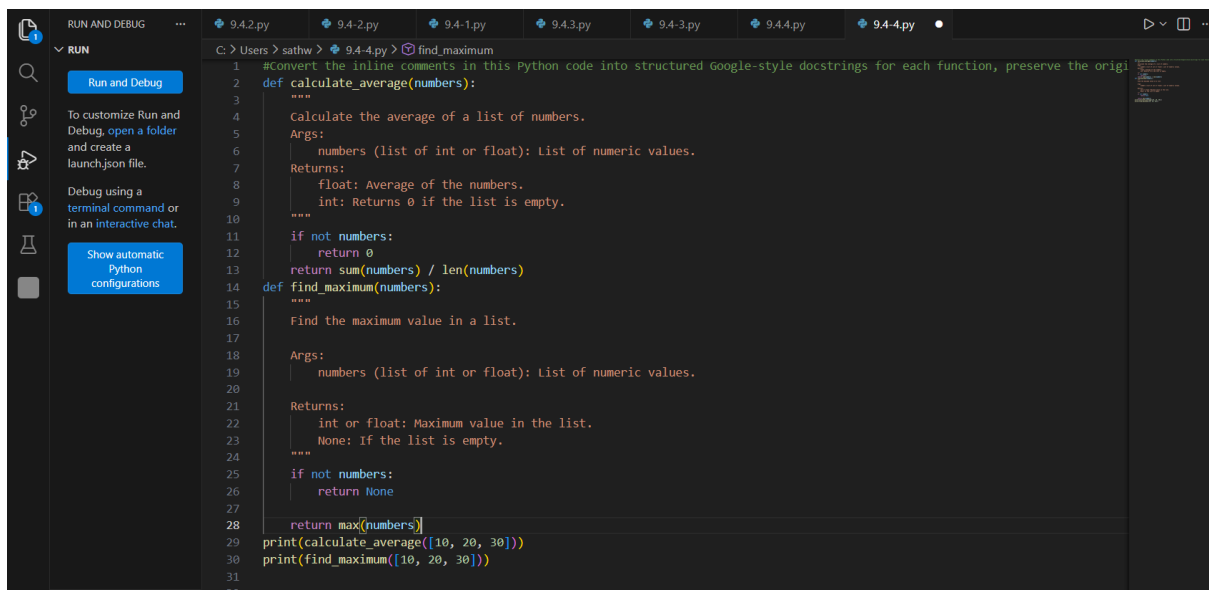


```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '49676' '--' 'C:\Users\sathw\9.4.4.py'
20.0
20.0
30
PS C:\Users\sathw>
```

Converted Version:

#PROMPT:

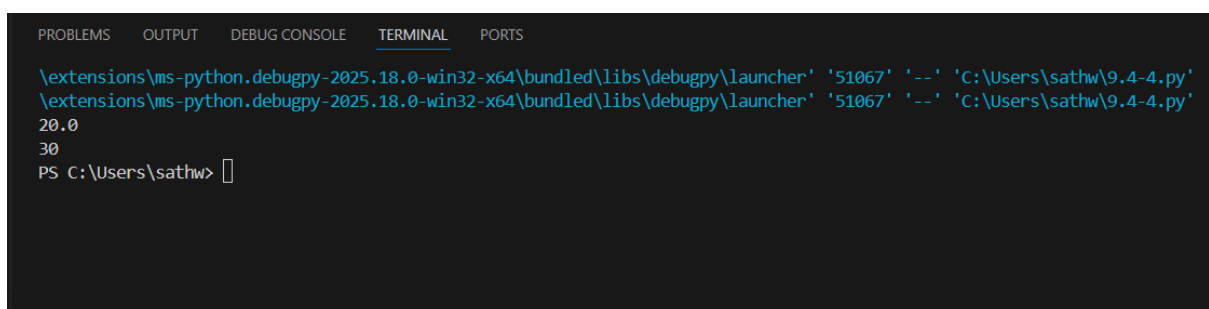
Convert the inline comments in this Python code into structured Google-style docstrings for each function, preserve the original meaning, remove redundant comments inside the function body, and do not change the program logic.



The screenshot shows a VS Code editor with a Python file named 9.4-4.py. The code defines two functions: calculate_average and find_maximum. The calculate_average function has a docstring that describes its purpose, arguments, and return values. The find_maximum function also has a docstring. The code includes inline comments that are being converted into structured docstrings. The output of the code is shown in the terminal window at the bottom.

```
1 #convert the inline comments in this Python code into structured Google-style docstrings for each function, preserve the origi
2 def calculate_average(numbers):
3     """
4     Calculate the average of a list of numbers.
5     Args:
6         numbers (list of int or float): List of numeric values.
7     Returns:
8         float: Average of the numbers.
9         int: Returns 0 if the list is empty.
10    """
11    if not numbers:
12        return 0
13    return sum(numbers) / len(numbers)
14 def find_maximum(numbers):
15     """
16     Find the maximum value in a list.
17     Args:
18         numbers (list of int or float): List of numeric values.
19     Returns:
20         int or float: Maximum value in the list.
21         None: If the list is empty.
22    """
23    if not numbers:
24        return None
25    return max(numbers)
26 print(calculate_average([10, 20, 30]))
27 print(find_maximum([10, 20, 30]))
28
```

#OUTPUT:



The screenshot shows a terminal window with the output of the Python code. The output is: 20.0 and 30. The prompt is PS C:\Users\sathw>.

```
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:

o Functions

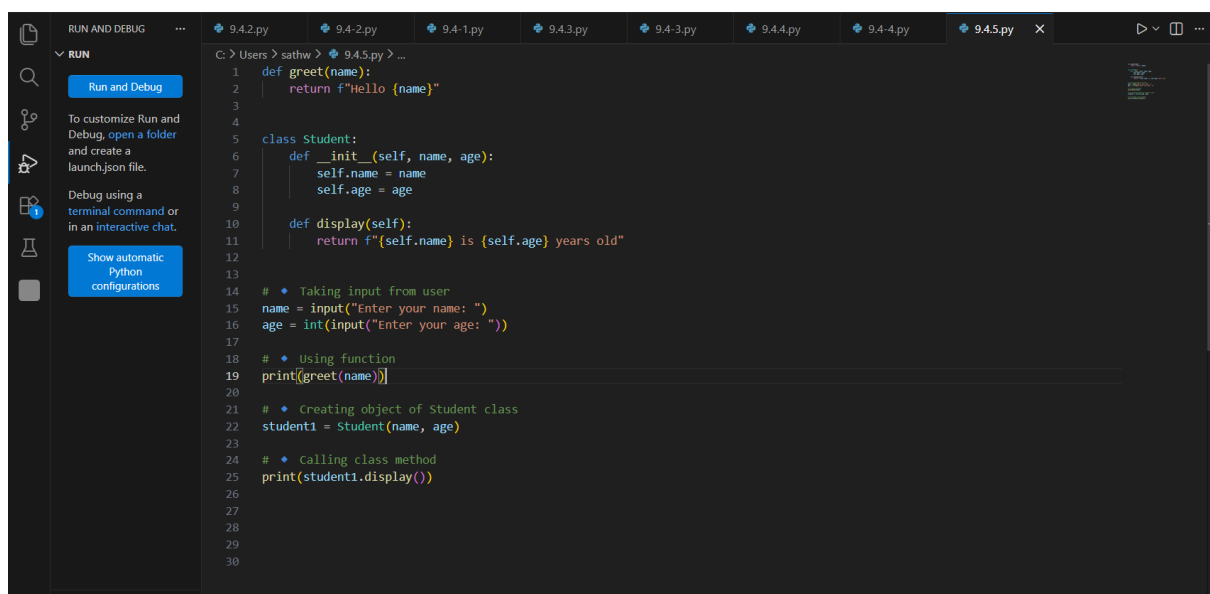
o Classes

- Inserts placeholder Google-style docstrings for each detected

function or class

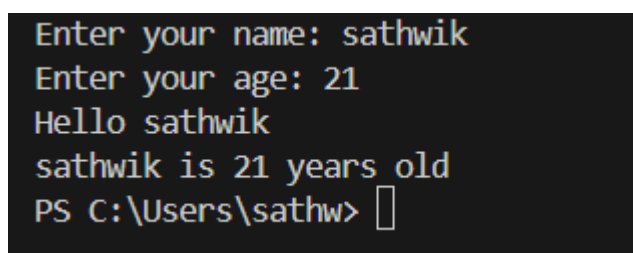
AI tools may be used to assist in generating or refining this utility.

#CODE AND INPUT:

A screenshot of a Python IDE with a dark theme. The left sidebar shows the 'RUN AND DEBUG' panel with a 'Run and Debug' button and instructions. The main editor area shows a Python file named '9.4.5.py' with the following code:

```
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
```

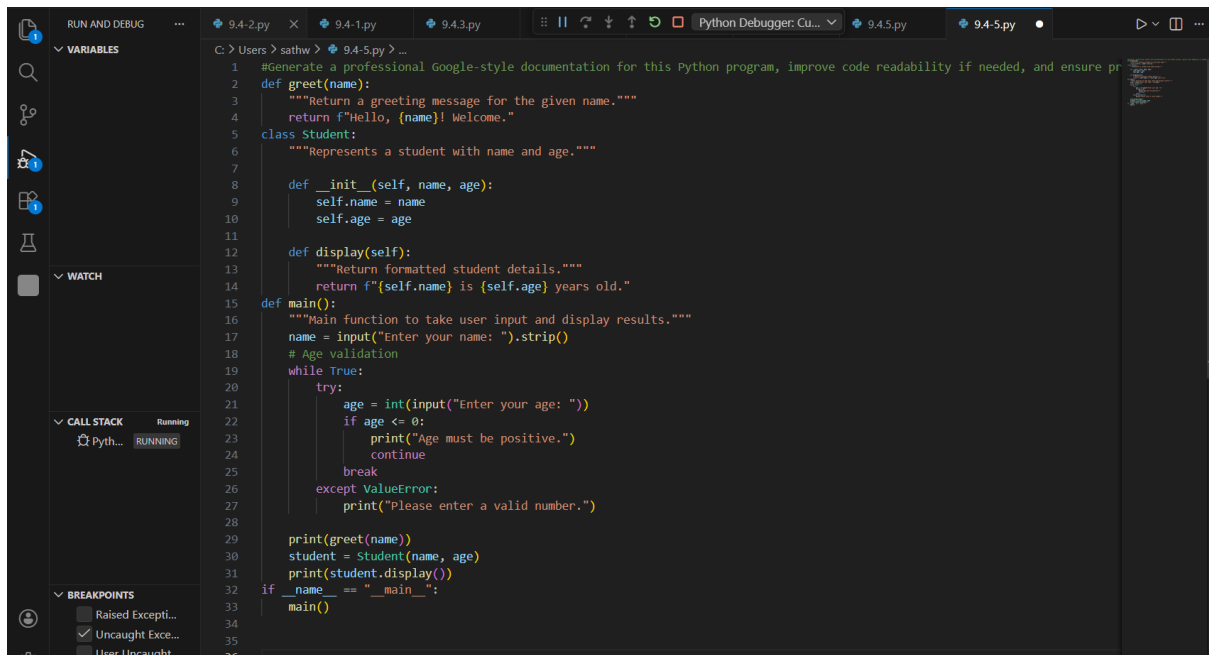
#OUTPUT:

A screenshot of a terminal window showing the output of the Python utility. The text is as follows:

```
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.

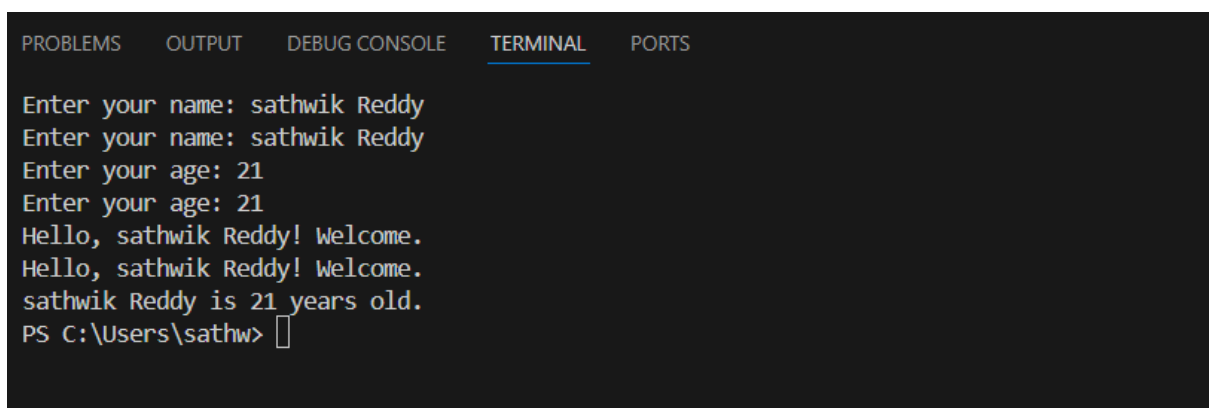


The screenshot shows a Python IDE with a dark theme. The main editor window displays the following code:

```
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 class Student:
6     """Represents a student with name and age."""
7
8     def __init__(self, name, age):
9         self.name = name
10        self.age = age
11
12    def display(self):
13        """Return formatted student details."""
14        return f"{self.name} is {self.age} years old."
15
16 def main():
17     """Main function to take user input and display results."""
18     name = input("Enter your name: ").strip()
19     # Age validation
20     while True:
21         try:
22             age = int(input("Enter your age: "))
23             if age <= 0:
24                 print("Age must be positive.")
25                 continue
26             break
27         except ValueError:
28             print("Please enter a valid number.")
29
30     print(greet(name))
31     student = Student(name, age)
32     print(student.display())
33
34 if __name__ == "__main__":
35     main()
```

The left sidebar shows the 'RUN AND DEBUG' panel with 'VARIABLES', 'WATCH', 'CALL STACK', and 'BREAKPOINTS' sections. The 'CALL STACK' section shows 'Python... RUNNING'. The 'BREAKPOINTS' section shows 'Uncaught Exce...' checked.

#OUTPUT:



The screenshot shows a terminal window with the following output:

```
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.