

AI ASSISTED CODING

ASSIGNMENT-9.4

HTNO: 2303A510B2

NAME: K AKHIL

BATCH: 14

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

```

lab9.4.py > ...
1  def add_numbers(a, b):
2      return a + b
3  def is_even(n):
4      return n % 2 == 0
5  def find_max(numbers):
6      max_val = numbers[0]
7      for num in numbers:
8          if num > max_val:
9              max_val = num
10     return max_val
11 def greet(name):
12     return "Hello, " + name + "!"
13 if __name__ == "__main__":
14     # Example usage of the functions
15     print(add_numbers(5, 3)) # Output: 8

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

PS C:\Users\akhil\OneDrive\Documents\AI(vs)> & c:/users/akhil/appdata/local/programs/python/python313/python.exe "c:/users/akhil/onedrive/documents/ai(vs)/lab9.4.py"
● 8
○ PS C:\Users\akhil\OneDrive\Documents\AI(vs)>

#prompt:

#Add Google-style docstrings to all functions in this Python code.

#Include:

#- Function description

#- Parameters with types

#- Return values

#- Example usage

The screenshot shows a code editor interface with a dark theme. At the top, there's a file navigation bar with a file icon, 'lab9.4.py', and a folder icon, 'add_numbers'. Below this is a large code block containing Python code with Google-style docstrings. The code defines a function 'add_numbers(a, b)' with detailed documentation for its args, returns, and example usage. At the bottom of the code block, there's a line 'return a + b'. Below the code editor, there's a navigation bar with tabs: PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is underlined, indicating it's active), PORTS, and POSTMAN CONSOLE. The terminal window below shows a command prompt 'PS C:\Users\akhil\OneDrive\Documents\AI(vs)> & C:/Users/akhil/AppData/Local/Programs/Python/Python3.8\python.exe lab9.4.py' followed by the output '8'.

```
16 #Add Google-style docstrings to all functions in this Python code.
17 #Include:
18 #- Function description
19 #- Parameters with types
20 #- Return values
21 #- Example usage
22 def add_numbers(a, b):
23     """Add two numbers together.
24
25     Args:
26         a (int): The first number.
27         b (int): The second number.
28
29     Returns:
30         int: The sum of a and b.
31
32     Example:
33         >>> add_numbers(5, 3)
34         8
35     """
36     return a + b
```

TERMINAL

```
PS C:\Users\akhil\OneDrive\Documents\AI(vs)> & C:/Users/akhil/AppData/Local/Programs/Python/Python3.8\python.exe lab9.4.py
8
○ PS C:\Users\akhil\OneDrive\Documents\AI(vs)>
```

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)

```
lab9.4.2.py > ...
1 def find_primes(n):
2     primes = []
3     for num in range(2, n + 1):
4         is_prime = True
5         for i in range(2, int(num ** 0.5) + 1):
6             if num % i == 0:
7                 is_prime = False
8                 break
9             if is_prime:
10                 primes.append(num)
11     return primes
12
13 print(find_primes(20))
14
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```
PS C:\Users\akhil\OneDrive\Documents\AI(vs)> & c:/Users/akhil/AppData/Local/Programs/Python/Python313/pyt
4.2.py"
● [2, 3, 5, 7, 11, 13, 17, 19]
○ PS C:\Users\akhil\OneDrive\Documents\AI(vs)>
```

AI-Generated Inline Comments (Improved Version)

#prompt:

#insert inline comments only for complex or non-obvious logic

#Avoid commenting on trivial or self-explanatory syntax

#Explain why the logic is used.

#Return only the improved code.

```

❸ lab9.4.2.py > ...
1  def find_primes(n):
2      primes = []
3      for num in range(2, n + 1):
4          is_prime = True
5          for i in range(2, int(num ** 0.5) + 1):
6              if num % i == 0:
7                  is_prime = False
8                  break
9          if is_prime:
10              primes.append(num)
11      return primes
12
13 print(find_primes(20))
14 #Insert inline comments only for complex or non-obvious logic
15 #Avoid commenting on trivial or self-explanatory syntax
16 #Explain why the logic is used.
17 #Return only the improved code.
18 def find_primes(n):
19     primes = []
20     for num in range(2, n + 1):
21         is_prime = True
22         # Check if num is divisible by any number from 2 to the square root of num
23         for i in range(2, int(num ** 0.5) + 1):
24             if num % i == 0:
25                 is_prime = False
26                 break # No need to check further if num is not prime
27         if is_prime:
28             primes.append(num) # Add num to the list of primes if it is prime
29     return primes
30 print(find_primes(20))

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```

PS C:\Users\akhil\OneDrive\Documents\AI(vs)> & C:/Users/akhil/AppData/Local/Programs/Python/Python313/python.exe "c:/[2, 3, 5, 7, 11, 13, 17, 19]
[2, 3, 5, 7, 11, 13, 17, 19]
PS C:\Users\akhil\OneDrive\Documents\AI(vs)>

```

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

```
lab9.4.3.py > ...
1  def find_primes(n):
2      primes = []
3      for num in range(2, n + 1):
4          is_prime = True
5          for i in range(2, int(num ** 0.5) + 1):
6              if num % i == 0:
7                  is_prime = False
8                  break
9          if is_prime:
10              primes.append(num)
11      return primes
12 n = int(input("Enter a number: "))
13 result = find_primes(n)
14 print("Prime numbers up to", n, "are:", result)
15 
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

PS C:\Users\akhil\OneDrive\Documents\AI(vs)> & C:/Users/akhil/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/akhil/OneDrive/I
● Enter a number: 20
Prime numbers up to 20 are: [2, 3, 5, 7, 11, 13, 17, 19]
○ PS C:\Users\akhil\OneDrive\Documents\AI(vs)>

AI Generates Module-Level Docstring:

PROMPT:

```
#Generate a professional module-level docstring for this Python file.
```

```
#Include:
```

```
#- Purpose of the module
```

```
#- Dependencies
```

```
#- Description of functions
```

```
#- Example usage
```

```
#Return the updated code with the docstring at the top.
```

```
#CODE AND INPUT:
```

```
15  #Generate a professional module-level docstring for this Python file.
16  #Include:
17  #- Purpose of the module
18  #- Dependencies
19  #- Description of functions
20  #- Example usage
21  #Return the updated code with the docstring at the top.
22  """
23  This module provides a function to find all prime numbers up to a given number.
24  Dependencies:
25  - None
26  Functions:
27  - find_primes(n): Returns a list of prime numbers up to n.
28  Example usage:
29  >>> find_primes(20)
30  [2, 3, 5, 7, 11, 13, 17, 19]
31  """
32 def find_primes(n):
33     primes = []
34     for num in range(2, n + 1):
35         is_prime = True
36         for i in range(2, int(num ** 0.5) + 1):
37             if num % i == 0:
38                 is_prime = False
39                 break
40             if is_prime:
41                 primes.append(num)
42     return primes
43 n = int(input("Enter a number: "))
44 result = find_primes(n)
45 print("Prime numbers up to", n, "are:", result)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```
PS C:\Users\akhil\OneDrive\Documents\AI(vs)> & C:/Users/akhil/AppData/Local/Programs/Python/Python313>
> Enter a number: 20
> Prime numbers up to 20 are: [2, 3, 5, 7, 11, 13, 17, 19]
```

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:

```
lab9.4.py > ...
1  def calculate_average(numbers):
2      # This function calculates the average of a list of numbers.
3      # It first checks if the list is empty.
4      # If the list is empty, it returns 0 to avoid division by zero.
5      # Otherwise, it sums all numbers and divides by the count.
6
7      if not numbers:
8          return 0
9
10     total = sum(numbers)
11     count = len(numbers)
12     return total / count
13
14 # Example usage: in user input format
15 if __name__ == "__main__":
16     input_str = input("Enter a list of numbers: ")
17     try:
18         numbers = list(map(float, input_str.split()))
19         average = calculate_average(numbers)
20         print(f"The average is: {average}")
21     except ValueError:
22         print("Please enter valid numbers.")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```
PS C:\Users\akhil\OneDrive\Documents\AI(vs)> & C:/Users/akhil/AppData/Local/Programs/Python/Python313/python lab9.4.py
> Enter a list of numbers: 1 2 4 5 2 6 9
The average is: 10.83333333333334
PS C:\Users\akhil\OneDrive\Documents\AI(vs)> & C:/Users/akhil/AppData/Local/Programs/Python/Python313/python lab9.4.py
> Enter a list of numbers: 25 48 3 26
The average is: 25.5
PS C:\Users\akhil\OneDrive\Documents\AI(vs)>
```

Converted Version:

#PROMPT:

#Convert the inline explanatory comments inside functions into

#structured Google-style docstrings.

#Requirements:

Preserve the meaning of the comments

Remove redundant inline comments

Keep the function logic unchanged

Return only the updated code

```
lab9.4.4.py > ...
22  def calculate_average(numbers):
23  #Convert the inline explanatory comments inside functions into
24  #structured Google-style docstrings.
25  #Requirements:
26  # Preserve the meaning of the comments
27  # Remove redundant inline comments
28  # Keep the function logic unchanged
29  # Return only the updated code
30  """Calculate the average of a list of numbers.
31
32  Args:
33      numbers (list of float): A list of numbers to calculate the average from.
34
35  Returns:
36      float: The average of the numbers. Returns 0 if the list is empty.
37
38  Example:
39      >>> calculate_average([1, 2, 3, 4])
40      2.5
41  """
42  if not numbers:
43      return 0
44
45  total = sum(numbers)
46  count = len(numbers)
47  return total / count
48 if __name__ == "__main__":
49     input_str = input("Enter a list of numbers: ")
50     try:
51         numbers = list(map(float, input_str.split()))
52         average = calculate_average(numbers)
53         print(f"The average is: {average}")
54     except ValueError:
55         print("Please enter valid numbers.")

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    POSTMAN CONSOLE
PS C:\Users\akhil\OneDrive\Documents\AI(vs)> & C:/Users/akhil/AppData/Local/Programs/Python/Python31
> Enter a list of numbers: 25 24 30 10
The average is: 22.25
> Enter a list of numbers: 10 20 30 40 50
The average is: 30.0
> PS C:\Users\akhil\OneDrive\Documents\AI(vs)>
```

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.

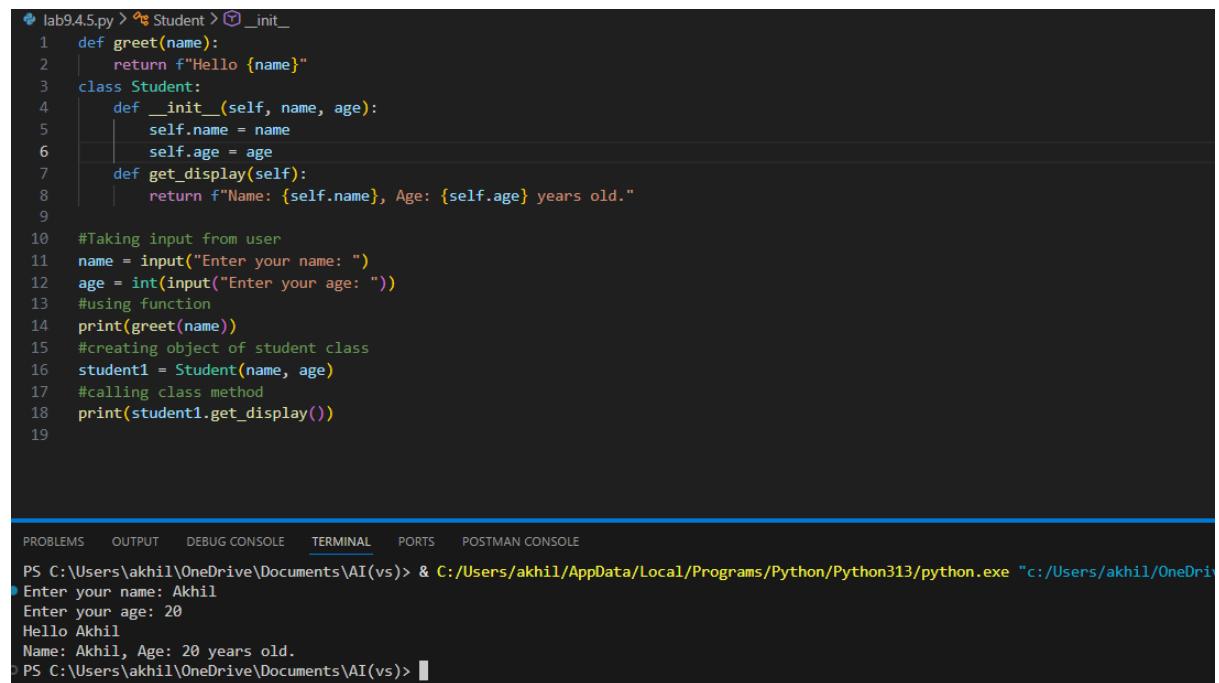
Task Description

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:



```
lab9.4.5.py > Student > _init_
1  def greet(name):
2      return f"Hello {name}"
3  class Student:
4      def __init__(self, name, age):
5          self.name = name
6          self.age = age
7      def get_display(self):
8          return f"Name: {self.name}, Age: {self.age} years old."
9
10 #Taking input from user
11 name = input("Enter your name: ")
12 age = int(input("Enter your age: "))
13 #using function
14 print(greet(name))
15 #creating object of student class
16 student1 = Student(name, age)
17 #calling class method
18 print(student1.get_display())
19
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```
PS C:\Users\akhil\OneDrive\Documents\AI(vs)> & C:/Users/akhil/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/akhil/OneDrive/.../lab9.4.5.py"
Enter your name: Akhil
Enter your age: 20
Hello Akhil
Name: Akhil, Age: 20 years old.
PS C:\Users\akhil\OneDrive\Documents\AI(vs)>
```

#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.

```
lab9.4.5.py > ...
2 #Generate a professional Google-style documentation for this Python program, improve code readability if needed,
3 #and ensure proper input validation and structured main() usage without changing the core functionality.
4 """This module defines a simple program that greets the user and displays their name and age using a Student class.
5 Functions:
6 - greet(name): Returns a greeting message for the given name.
7 Classes:
8 - Student: A class representing a student with attributes name and age, and a method to display this information.
9 Example usage:
10 >>> main()
11 Enter your name: Alice
12 Enter your age: 30
13 Hello Alice
14 Name: Alice, Age: 30 years old.
15 """
16 def greet(name):
17     """Return a greeting message for the given name.
18
19     Args:
20         name (str): The name of the person to greet.
21     Returns:
22         str: A greeting message.
23     Example:
24         >>> greet("Alice")
25         'Hello Alice'
26     """
27     return f"Hello {name}"
28 class Student:
29     """A class representing a student with attributes name and age, and a method to display this information."""
30     def __init__(self, name, age):
31         """Initialize the Student object with name and age.
32
33         Args:
34             name (str): The name of the student.
35             age (int): The age of the student.
```

```

35     age (int): The age of the student.
36     """
37     self.name = name
38     self.age = age
39     def get_display(self):
40         """Return a formatted string displaying the student's name and age.
41
42     Returns:
43         str: A string in the format "Name: {name}, Age: {age} years old."
44     Example:
45         >>> student = Student("Alice", 30)
46         >>> student.get_display()
47         'Name: Alice, Age: 30 years old.'
48
49     return f"Name: {self.name}, Age: {self.age} years old."
50 def main():
51     """Main function to execute the program logic."""
52     name = input("Enter your name: ")
53     while True:
54         try:
55             age = int(input("Enter your age: "))
56             break
57         except ValueError:
58             print("Please enter a valid integer for age.")
59     print(greet(name))
60     student1 = Student(name, age)
61     print(student1.get_display())
62 if __name__ == "__main__":
63     main()

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

```

PS C:\Users\akhil\OneDrive\Documents\AI(vs)> & C:/Users/akhil/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/akhil/OneDrive/Documents/AI(vs)/greet.py"
Enter your name: Akhil
Enter your age: 20
Hello Akhil
Name: Akhil, Age: 20 years old.
PS C:\Users\akhil\OneDrive\Documents\AI(vs)> █

```

#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.