**Assignment 9.1**

**Lab 9 – Documentation Generation**

**Course**: AI Assisted Coding

**Name**: Mohd Aadil Ashraf

**Roll No**: 2403A51315

**Instructo**r: Dr. Mohammand Ali Shaik

**Date**: 25/08/2025

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

## Objectives

- Use AI to generate Python documentation and comments.

- Try zero-shot, few-shot, and context prompts for docs.

- Practice docstrings, inline comments, and module docs.

- Compare quality of prompts for documentation.

## Task 1 – Google-Style Docstrings

Prompt: Add Google-style docstrings to all functions in the script.

Code Example:

def add(a: int, b: int) -> int:  
 """  
 Adds two integers.  
  
 Args:  
 a (int): first number  
 b (int): second number  
  
 Returns:  
 int: sum of a and b  
  
 Example:  
>>> add(2, 3)  
 5  
 """  
 return a + b

Output: Script functions now have proper docstrings with parameters, return, and examples.

## Task 2 – Inline Comments for Complex Logic

Prompt: Add inline comments only for tricky code.

Code Example:

def factorial(n):  
 result = 1  
 for i in range(1, n+1):  
 # multiply step keeps growing factorial value  
 result \*= i  
 return result

Output: Comments explain the logic but skip obvious syntax.

## Task 3 – Module-Level Documentation

Prompt: Add a module-level docstring describing the file.

Code Example:

"""  
This module provides math utility functions.  
  
Functions:  
 add(a, b): adds two numbers  
 factorial(n): computes factorial  
Dependencies:  
 None  
"""

Output: Clear summary at top of file.

## Task 4 – Convert Comments to Docstrings

Prompt: Turn inline comments into docstrings.

Before:

def square(x):  
 # returns square of number  
 return x \* x

After:

def square(x: int) -> int:  
 """  
 Returns square of a number.  
  
 Args:  
 x (int): number input  
  
 Returns:  
 int: square of x  
 """  
 return x \* x

Output: Inline comments replaced by structured docstring.

## Task 5 – Review and Correct Docstrings

Prompt: Fix incorrect docstrings.

Before:

def divide(a, b):  
 """Multiplies a and b"""   
 return a / b

After:

def divide(a: float, b: float) -> float:  
 """  
 Divides a by b.  
  
 Args:  
 a (float): numerator  
 b (float): denominator  
  
 Returns:  
 float: result of division  
 """  
 return a / b

Output: Docstrings corrected to match function behavior.

## Task 6 – Prompt Comparison

Function Used:

def greet(name):  
 return "Hello " + name

Vague Prompt Output:

# adds greeting to name  
def greet(name):  
 return "Hello " + name

Detailed Prompt Output:

def greet(name: str) -> str:  
 """  
 Greets the user by name.  
  
 Args:  
 name (str): name of the user  
  
 Returns:  
 str: greeting message  
  
 Example:  
>>> greet("Alice")  
 'Hello Alice'  
 """  
 return "Hello " + name

Comparison Table:

|  |  |  |
| --- | --- | --- |
| Prompt Type | Result | Quality |
| Vague | Only small comment | Low detail |
| Detailed | Full Google-style docstring | High detail + accurate |

## Conclusion

AI can generate proper docstrings and comments.  
Zero-shot sometimes gives vague output.  
Few-shot and detailed prompts improve quality.  
Structured docstrings help readability and maintainability.