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

**Course:** AI Assisted Coding

**Assignment Number:** 9.2

**Student Name:** Radha Krishna Jagarlapudi

**Roll Number:** 2303A52055

## Task 1: Documentation – Function Summary Generation

### Task Description

Use AI to generate concise functional summaries for each Python function in a given script. The goal is to describe the purpose of each function without including implementation details.

### Prompt

"Read the following Python code. For each function defined in the script, generate a concise summary describing its purpose. Insert these summaries as docstrings immediately after the function definition. Do not describe the code line-by-line, just the intent."

### Code (Input)

def calculate\_average(grades):  
 if not grades:  
 return 0  
 return sum(grades) / len(grades)  
  
def determine\_status(average):  
 if average >= 40:  
 return "Pass"  
 else:  
 return "Fail"

### Output (AI Generated)

def calculate\_average(grades):  
 """  
 Calculates the arithmetic mean of a list of numerical grades.  
 Returns 0 if the list is empty to prevent division by zero errors.  
 """  
 if not grades:  
 return 0  
 return sum(grades) / len(grades)  
  
def determine\_status(average):  
 """  
 Evaluates whether a given average score qualifies as a 'Pass' or 'Fail'   
 based on a threshold of 40.  
 """  
 if average >= 40:  
 return "Pass"  
 else:  
 return "Fail"

### Explanation

In this task, the AI analyzed the function signatures and bodies. For calculate\_average, it identified that the math operation was finding a mean and noted the edge case handling for empty lists. For determine\_status, it correctly identified the logic branching (pass/fail) and the threshold value. The AI inserted these as standard Python docstrings ("""..."""), which is the conventional way to document functions.

## Task 2: Documentation – Logical Explanation for Conditions and Loops

### Task Description

Use AI to document the logic behind conditional statements and loops in a Python program, skipping basic syntax explanations.

### Prompt

"Analyze the following Python code. Add comments explaining only the decision-making logic (if/else) and the loop behavior. Do not explain basic syntax like variable assignment."

### Code (Input)

students = [{"name": "Alice", "score": 85}, {"name": "Bob", "score": 30}, {"name": "Charlie", "score": 45}]  
  
for student in students:  
 if student["score"] >= 75:  
 print(f"{student['name']} gets Distinction")  
 elif student["score"] >= 40:  
 print(f"{student['name']} Passed")  
 else:  
 print(f"{student['name']} Failed")

### Output (AI Generated)

students = [{"name": "Alice", "score": 85}, {"name": "Bob", "score": 30}, {"name": "Charlie", "score": 45}]  
  
# Iterate through each dictionary in the students list to process grades individually  
for student in students:  
 # Check if the score is 75 or higher to award Distinction  
 if student["score"] >= 75:  
 print(f"{student['name']} gets Distinction")  
 # If not distinction, check if score is at least 40 to grant a Pass  
 elif student["score"] >= 40:  
 print(f"{student['name']} Passed")  
 # Fallback for any score below 40, resulting in a Fail  
 else:  
 print(f"{student['name']} Failed")

### Explanation

The AI successfully ignored the definition of the list students. It focused specifically on the for loop, explaining that it iterates through records. It then broke down the if/elif/else block, explaining the business logic (Distinction vs. Pass vs. Fail) rather than simply stating "if score is greater than 75". This helps a reader understand *why* the check exists, not just what it does.

## Task 3: Documentation – File-Level Overview

### Task Description

Use AI to generate a high-level overview describing the functionality of an entire Python file and place it at the top.

### Prompt

"Read the code below and generate a high-level file header docstring. It should summarize the file's overall purpose and functionality. Place it at the very top of the code."

### Code (Input)

*(Using the combined code from previous tasks)*

import math  
  
def calculate\_area(radius):  
 return math.pi \* radius \* radius  
  
def calculate\_circumference(radius):  
 return 2 \* math.pi \* radius  
  
def main():  
 r = 5  
 print(f"Area: {calculate\_area(r)}")  
 print(f"Circumference: {calculate\_circumference(r)}")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

### Output (AI Generated)

"""  
Circle Geometry Calculator Module  
  
This script provides utility functions to calculate geometric properties of a circle,  
specifically its area and circumference, based on a given radius.   
It includes a main execution block to demonstrate usage with a sample radius.  
"""  
import math  
  
def calculate\_area(radius):  
 return math.pi \* radius \* radius  
# ... rest of code ...

### Explanation

The AI scanned the entire file and synthesized a summary. It identified that the file isn't just random math, but specifically "Circle Geometry". It correctly noted the presence of utility functions and a main execution block. This provides a developer opening the file with immediate context before they read a single line of actual code.

## Task 4: Documentation – Refine Existing Documentation

### Task Description

Use AI to improve clarity and consistency of existing documentation in Python code, rewriting vague or basic comments.

### Prompt

"The following code has poor quality comments. Rewrite the comments to be professional, clear, and consistent. Do not change the code itself."

### Code (Input)

def process\_data(data):  
 # check if data is there  
 if not data:  
 return None  
   
 # make list empty  
 result = []  
   
 # loop loop  
 for item in data:  
 # math stuff  
 result.append(item \* 2)  
   
 return result

### Output (AI Generated)

def process\_data(data):  
 # Validate input: return None immediately if the data list is empty or None  
 if not data:  
 return None  
   
 # Initialize container for processed values  
 result = []  
   
 # Iterate through each item in the input data  
 for item in data:  
 # Apply transformation: double the value of the item  
 result.append(item \* 2)  
   
 return result

### Explanation

The original comments were extremely unhelpful ("math stuff", "loop loop"). The AI replaced them with professional technical writing. "check if data is there" became "Validate input...". "math stuff" became "Apply transformation: double the value...". This transforms the code from amateurish to professional quality documentation without altering the logic.

## Task 5: Documentation – Prompt Detail Impact Study

### Task Description

Study the impact of prompt detail on AI-generated documentation quality by comparing a brief prompt vs. a detailed prompt.

### Code (Input)

def fetch\_user(user\_id, db\_conn):  
 query = f"SELECT \* FROM users WHERE id = {user\_id}"  
 return db\_conn.execute(query)

### Prompt A (Brief)

"Add a docstring to this function."

### Prompt B (Detailed)

"Generate a Google-style docstring for this function. Include sections for Args (listing parameters and types), Returns (describing output), and a Warning about any security risks visible in the code."

### Output Comparison

|  |  |  |
| --- | --- | --- |
| **Feature** | **Output A (Brief Prompt)** | **Output B (Detailed Prompt)** |
| **Docstring** | """Fetches a user from the database using their ID.""" | (See full block below) |
| **Completeness** | Low. Only describes the basic action. | High. Includes params, returns, and security warnings. |
| **Clarity** | Simple, but misses context. | Very clear structure. |
| **Accuracy** | Accurate summary. | Accurate summary + identifies SQL injection risk. |

**Output B (Actual Result):**

def fetch\_user(user\_id, db\_conn):  
 """  
 Fetches user details from the database.  
  
 Args:  
 user\_id (int): The unique identifier of the user.  
 db\_conn (object): The active database connection object.  
  
 Returns:  
 cursor: The result of the database query execution.  
  
 Warning:  
 This function constructs a raw SQL query using f-strings and is   
 highly vulnerable to SQL Injection attacks.  
 """  
 query = f"SELECT \* FROM users WHERE id = {user\_id}"  
 return db\_conn.execute(query)

### Explanation

This comparison demonstrates that "Zero-shot" or brief prompting often yields generic results suitable for quick comments. However, "Context-based" or detailed prompting (Prompt B) forces the AI to structure the output specifically (Google-style) and can even prompt it to analyze code quality (security warnings). The detailed prompt resulted in documentation that serves not just to explain the code, but to warn future developers of critical flaws.