AI ASSISTED CODING

ASSIGNMENT 9.1

2403A51257

Shiva Sai.B

BATCH :11

Task 1: Add Google-Style Docstrings to Functions

: Use AI to add Google-style docstrings to all functions in a given Python script.
• Instructions:
o Prompt AI to generate docstrings without providing any input-output examples.
o Ensure each docstring includes:
▪ Function description
▪ Parameters with type hints
▪ Return values with type hints
▪ Example usage
o Review the generated docstrings for accuracy and formatting.

Objective: Use AI to generate standardized, detailed function docstrings.

Instructions:

- Use AI with **zero-shot** prompt (do not provide examples).
- Make sure each function's docstring includes:
    o **Function description**
    o **Parameters with type hints**
    o **Return values with type hints**
    o **Example usage**
- Manually review for clarity and format.

- Expected output:

```python
def sample_function(x: int, y: int) -> int:
    """Adds two integers and returns the result.

    Args:
        x (int): First integer.
        y (int): Second integer.

    Returns:
        int: The sum of x and y.

    Example:
        >>> sample_function(2, 3)
        5
    """
    return x + y
```

prompt:

#Add a Google-style docstring to this Python function. Include a description, parameter types, return type, and an example.

Task 2: Add Inline Comments for Complex Logic

Use AI to add meaningful inline comments to a Python program explaining only complex logic parts.
• Instructions:
o Provide a Python script without comments to the AI.
o Instruct AI to skip obvious syntax explanations and focus only on tricky or non-intuitive code sections.
o Verify that comments improve code readability and maintainability.
• Expected Output #2:
o Python code with concise, context-aware inline comments for complex logic blocks.

Objective: Focus AI-generated comments only on non-obvious logic.

Instructions:

- Input: Python code without comments.
- Skip simple lines like variable assignment or loops.
- Target:
  - Tricky conditions
  - Recursive logic
  - Algorithmic sections
- Ensure improved readability.

Expected output:

```
if a > b and c < d:
    # Check if a dominates b while c is still below d, indicating an edge
case
    handle_edge_case()
```

prompt:

#Add inline comments only to the non-obvious or complex parts of this code. Skip explaining simple syntax.

Task 3: Add Module-Level Docstring

Use AI to create a module-level docstring summarizing the purpose, dependencies, and main functions/classes of a Python file.

• Instructions:
o Supply the entire Python file to AI.
o Instruct AI to write a single multi-line docstring at the top of the file.
o Ensure the docstring clearly describes functionality and usage without rewriting the entire code.
• Expected Output #3:
o A complete, clear, and concise module-level docstring at the beginning of the file.

Objective: Provide a summary at the top of the Python file.

Instructions:

- Supply the **entire Python file**.
- AI should generate a multi-line docstring that includes:
  - **Purpose of the module**
  - **Dependencies (if any)**

    List of main functions and classes

  - **Brief description of usage**
  - **Expected output:**

```
"""
This module processes user data from a CSV file, validates entries,
and stores them in a SQLite database.

Dependencies:
- pandas
- sqlite3

Main Functions:
- load_csv_data
- validate_entries
- store_to_db

Usage:
    Run this script directly to process the default data.csv file.
"""
```

Prompt:

#Write a module-level docstring for this file describing the purpose, dependencies, and available functions.

Task 4: Convert Inline Comments to Google-Style Docstrings

Use AI to transform existing inline comments into structured function docstrings following Google style.
• Instructions:

o **Provide AI with Python code containing inline comments.**
o **Ask AI to move relevant details from comments into function docstrings.**
o **Verify that the new docstrings keep the meaning intact while improving structure.**
• **Expected Output #4:**
o **Python code with comments replaced by clear, standardized docstrings**

Objective: Refactor functions by moving inline comments into docstrings.

Instructions:

- Provide code that has inline comments.
- Instruct AI to extract relevant comments and move them into Google-style docstrings.
- Keep code logic untouched, remove in-code comments.
-
- Expected output:

```python
def calculate_area(radius: float) -> float:
    """Calculates the area of a circle.

    Args:
        radius (float): Radius of the circle.

    Returns:
        float: The calculated area.
    """
    return 3.1415 * radius * radius
```

prompt:

#Convert inline comments into a structured Google-style docstring.

Task 5: Review and Correct Existing Docstrings

Use AI to identify and correct inaccuracies in existing docstrings.
• **Instructions:**
o **Provide Python code with outdated or incorrect docstrings.**

○ **Instruct AI to rewrite each docstring to match the current code behavior.**
○ **Ensure corrections follow Google-style formatting.**
• **Expected Output #5:**
○ **Python file with updated, accurate, and standardized docstrings.**

Objective: Fix incorrect, outdated, or incomplete docstrings.

Instructions:

- Provide code with poor or outdated docstrings.
- Ask AI to:
  - Rewrite each docstring to reflect actual behavior.
  - Use proper Google-style formatting.
- Expected output:
- Before:

```python
def login(user):
    """Checks login."""
    ...
```

Expected output:

After:

```python
def login(user: str) -> bool:
    """Validates user credentials for login.

    Args:
        user (str): Username string.

    Returns:
        bool: True if login is successful, False otherwise.
    """
    ...
```

Prompt:

#Correct the docstring to accurately describe the function using Google style.

Task 6: Prompt Comparison Experiment

 Prompt Comparison Experiment Compare documentation output from a vague prompt and a detailed prompt for the same Python function.
• Instructions:
o Create two prompts: one simple ("Add comments to this function") and one detailed ("Add Google-style docstrings with parameters, return types, and examples").
o Use AI to process the same Python function with both prompts.
o Analyze and record differences in quality, accuracy, and completeness.
• Expected Output #6:
o A comparison table showing the results from both prompts with observations.

Objective: Compare AI output from vague vs detailed prompts.

Instructions:

- Use one simple prompt:
    - o `"Add comments to this function"`
- Use one detailed prompt:
    - o `"Add Google-style docstrings with parameters, return types, and examples"`
- Apply both to the same function.
- Create a comparison table with observations:
    - o **Clarity**
    - o **Completeness**
    - o **Correctness**
    - o **Structure**
- **Expected Output Table:**

Aspect	Vague Prompt Output	Detailed Prompt Output	Observation
Clarity	Basic one-line comment	Structured docstring with clear explanation	Detailed prompt much clearer
Completeness	Only what function does	Full param/return types, example usage	Detailed prompt is more complete
Correctness	Partially aligns with behavior	Matches function's logic closely	Detailed prompt produces accurate results

Aspect	Vague Prompt Output	Detailed Prompt Output	Observation
Structure	Informal style	Google-style standard	Detailed prompt adheres to best practices

#Prompt: Add a Google-style docstring to this function. Include a description, parameter types, return type, and an example.