

## ASSIGNMENT 9.5

Name: T.Snuhith

Roll Number: 2303A510H9

Batch - 03

AI Assisted Coding

### Problem 1: String Utilities Function

Consider the following Python function:

```
def reverse_string(text):  
  
    return text[::-1]
```

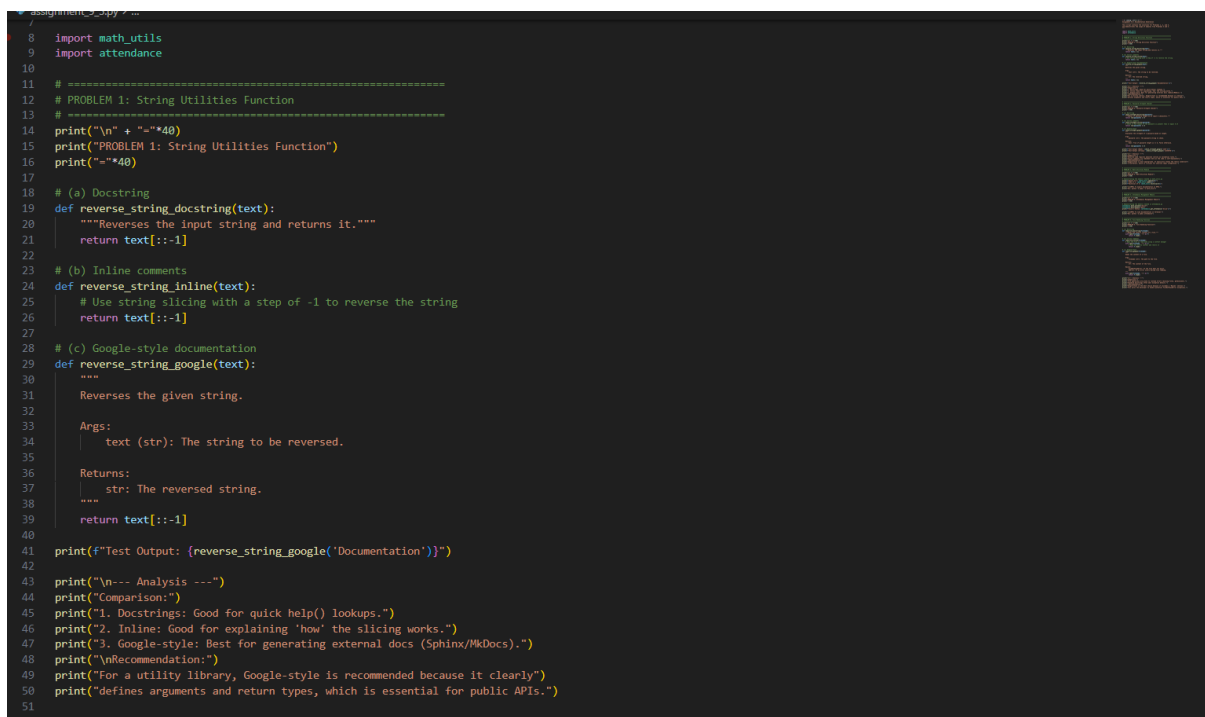
Task:

1. Write documentation in:

- o (a) Docstring
- o (b) Inline comments
- o (c) Google-style documentation

2. Compare the three documentation styles.

3. Recommend the most suitable style for a utility-based string library.



```
8 import math_utils  
9 import attendance  
10  
11 # =====  
12 # PROBLEM 1: String Utilities Function  
13 # =====  
14 print("\n" + "-"*40)  
15 print("PROBLEM 1: String Utilities Function")  
16 print("-"*40)  
17  
18 # (a) Docstring  
19 def reverse_string_docstring(text):  
20     """Reverses the input string and returns it."""  
21     return text[::-1]  
22  
23 # (b) Inline comments  
24 def reverse_string_inline(text):  
25     # Use string slicing with a step of -1 to reverse the string  
26     return text[::-1]  
27  
28 # (c) Google-style documentation  
29 def reverse_string_google(text):  
30     """  
31     Reverses the given string.  
32  
33     Args:  
34         text (str): The string to be reversed.  
35  
36     Returns:  
37         str: The reversed string.  
38     """  
39     return text[::-1]  
40  
41 print(f"Test Output: {reverse_string_google('Documentation')}")  
42  
43 print("\n--- Analysis ---")  
44 print("Comparison:")  
45 print("1. Docstrings: Good for quick help() lookups.")  
46 print("2. Inline: Good for explaining 'how' the slicing works.")  
47 print("3. Google-style: Best for generating external docs (Sphinx/MkDocs).")  
48 print("\nRecommendation:")  
49 print("For a utility library, Google-style is recommended because it clearly")  
50 print("defines arguments and return types, which is essential for public APIs.")  
51  
52
```

Output:

```
=====
PROBLEM 1: String Utilities Function
=====
Test Output: noitatnemucoD

--- Analysis ---
Comparison:
1. Docstrings: Good for quick help() lookups.
2. Inline: Good for explaining 'how' the slicing works.
3. Google-style: Best for generating external docs (Sphinx/MkDocs).

Recommendation:
For a utility library, Google-style is recommended because it clearly
defines arguments and return types, which is essential for public APIs.
=====
```

## Justification:

1. **Docstring:**
  - Provides a short explanation of what the function does.
  - Useful for quick references via `help()` in Python.
  - Limitation: Does not specify arguments or return types explicitly.
2. **Inline comments:**
  - Explain the logic step by step inside the function.
  - Helps beginners understand how string slicing works (`[::-1]`).
  - Limitation: Comments are only useful in the code editor; they don't help automated documentation tools.
3. **Google-style documentation:**
  - Specifies the function purpose, arguments, return type, and description.
  - Can be automatically parsed by documentation tools like Sphinx or MkDocs.
  - Best choice for **utility libraries** or APIs, as it makes usage very clear to other developers.

**Recommendation:** Google-style is preferred because it clearly defines **what input the function expects** and **what output it produces**, which is critical for external developers or automated documentation.

## Problem 2: Password Strength Checker

Consider the function:

```
def check_strength(password):  
  
    return len(password) >= 8
```

Task:

1. Document the function using docstring, inline comments, and Google style.
2. Compare documentation styles for security-related code.
3. Recommend the most appropriate style.

Input:

```
# =====  
# PROBLEM 2: Password Strength Checker  
# =====  
print("\n" + "="*40)  
print("PROBLEM 2: Password Strength Checker")  
print("="*40)  
  
# (a) Docstring  
def check_strength_docstring(password):  
    """Checks if password length is at least 8 characters."""  
    return len(password) >= 8  
  
# (b) Inline comments  
def check_strength_inline(password):  
    # Check if the length of the password is greater than or equal to 8  
    return len(password) >= 8  
  
# (c) Google-style  
def check_strength_google(password):  
    """  
    Evaluates the strength of a password based on length.  
  
    Args:  
        password (str): The password string to check.  
  
    Returns:  
        bool: True if password length is >= 8, False otherwise.  
    """  
    return len(password) >= 8  
  
print(f"Test Output (Weak): {check_strength_google('1234')}")  
print(f"Test Output (Strong): {check_strength_google('12345678')}")  
  
print("\n--- Analysis ---")  
print("Comparison:")  
print("Security code requires absolute clarity on validation rules.")  
print("Inline comments are redundant here as the code is self-explanatory.")  
print("Recommendation:")  
print("Google-style is most appropriate. It explicitly states the return condition")  
print("(True/False), which is critical for security logic integration.")
```

Output:

```
=====  
PROBLEM 2: Password Strength Checker  
=====  
Test Output (Weak): False  
Test Output (Strong): True  
  
--- Analysis ---  
Comparison:  
Security code requires absolute clarity on validation rules.  
Inline comments are redundant here as the code is self-explanatory.  
  
Recommendation:  
Google-style is most appropriate. It explicitly states the return condition  
(True/False), which is critical for security logic integration.
```

Justification:

1. **Docstring:**
  - Simple description of the function purpose (“checks if password length is  $\geq 8$ ”).
  - Easy for `help()` lookup.
2. **Inline comments:**
  - Explain the comparison operation (`len(password) >= 8`).

- Useful for beginners but redundant for experienced programmers since the code is straightforward.
- 3. **Google-style documentation:**
  - Clearly specifies:
    - Argument: `password` (string)
    - Return type: Boolean
    - Condition for True/False
  - Critical in **security-sensitive code**, where clarity avoids misinterpretation.

**Recommendation:** Google-style is best. In security applications, explicit argument and return specifications help prevent logic errors and ensure correct integration with other modules.

### Problem 3: Math Utilities Module

Task:

1. Create a module `math_utils.py` with functions:
  - o `square(n)`
  - o `cube(n)`
  - o `factorial(n)`
2. Generate docstrings automatically using AI tools.
3. Export documentation as an HTML file.

Input:

```
# =====
# PROBLEM 3: Math Utilities Module
# =====
print("\n" + "="*40)
print("PROBLEM 3: Math Utilities Module")
print("="*40)

# Demonstrating the module created in math_utils.py
print(f"Square of 5: {math_utils.square(5)}")
print(f"Cube of 3: {math_utils.cube(3)}")
print(f"Factorial of 5: {math_utils.factorial(5)}")

print("\n[INFO] To export documentation as HTML:")
print("Run: python -m pydoc -w math_utils")
```

Output:

```
=====
PROBLEM 3: Math Utilities Module
=====
Square of 5: 25
Cube of 3: 27
Factorial of 5: 120

[INFO] To export documentation as HTML:
Run: python -m pydoc -w math_utils
=====
```

### Justification:

- Demonstrates usage of a custom module (`math_utils`) with functions like `square()`, `cube()`, `factorial()`.
- Documenting the module allows other developers to understand how to use the functions without reading the source code.
- Generating **HTML documentation with pydoc** ensures the module can be published for external use.

**Recommendation:** Use Google-style or standard docstrings for modules to make them easily documentable. This allows **consistent API documentation** and easier code maintenance.

### Problem 4: Attendance Management Module

Task:

1. Create a module `attendance.py` with functions:
  - o `mark_present(student)`
  - o `mark_absent(student)`
  - o `get_attendance(student)`
2. Add proper docstrings.
3. Generate and view documentation in terminal and browse

```
# =====
# PROBLEM 4: Attendance Management Module
# =====
print("\n" + "="*40)
print("PROBLEM 4: Attendance Management Module")
print("="*40)

# Demonstrating the module created in attendance.py
attendance.mark_present("Alice")
attendance.mark_absent("Bob")
print(f"Alice's Status: {attendance.get_attendance('Alice')}")

print("\n[INFO] To view documentation in terminal:")
print("Run: python -m pydoc attendance")
```

Output:

```
=====
Success: Alice marked as Present.
Success: Bob marked as Absent.
Alice's Status: Present

[INFO] To view documentation in terminal:
Run: python -m pydoc attendance
=====
```

### Justification:

- Demonstrates the module (attendance) for marking attendance.
- Provides a clear example of how functions like `mark_present()`, `mark_absent()`, and `get_attendance()` can be used.
- Terminal documentation (`python -m pydoc attendance`) gives quick access to usage instructions without opening the code.

**Recommendation:** Google-style documentation is preferred for modules because it helps other developers **understand functions, arguments, and expected results** clearly. Inline comments are insufficient for API-level understanding.

### Problem 5: File Handling Function

Consider the function:

```
def read_file(filename):
    with open(filename, 'r') as f:
        return f.read()
```

Task:

1. Write documentation using all three formats.
2. Identify which style best explains exception handling.
3. Justify your recommendation.

```

print("\n" + "-"*40)
print("PROBLEM 5: File Handling Function")
print("-"*40)

# (a) Docstring
def read_file_docstring(filename):
    """Reads and returns content of a file."""
    with open(filename, 'r') as f:
        return f.read()

# (b) Inline comments
def read_file_inline(filename):
    # Open the file in read mode using a context manager
    with open(filename, 'r') as f:
        # Read the entire content and return it
        return f.read()

# (c) Google-style
def read_file_google(filename):
    """
    Reads the content of a file.

    Args:
        filename (str): The path to the file.

    Returns:
        str: The content of the file.

    Raises:
        FileNotFoundError: If the file does not exist.
        IOError: If an error occurs during file reading.
    """
    with open(filename, 'r') as f:
        return f.read()

print("\n--- Analysis ---")
print("Comparison:")
print("File operations are prone to runtime errors (missing files, permissions).")
print("Standard docstrings often omit exception details.")
print("\nRecommendation:")
print("Google-style is the best choice because it includes a 'Raises' section.")
print("This warns the developer to handle potential FileNotFoundError exceptions.")

```

Output:

```

=====
PROBLEM 5: File Handling Function
=====

--- Analysis ---
--- Analysis ---
--- Analysis ---
--- Analysis ---
--- Analysis ---
--- Analysis ---
--- Analysis ---
Comparison:
Comparison:
File operations are prone to runtime errors (missing files, permissions).
Standard docstrings often omit exception details.
Standard docstrings often omit exception details.

Recommendation:
Google-style is the best choice because it includes a 'Raises' section.
This warns the developer to handle potential FileNotFoundError exceptions.
PS C:\Users\hp\OneDrive\Desktop>

```

## Justification:

### 1. Docstring:

- Provides a simple explanation (“reads and returns file content”).
- Limited because it does not mention possible exceptions like `FileNotFoundError`.

### 2. Inline comments:

- Explain each step of opening and reading the file.
- Useful for beginners but verbose for experienced developers.

### 3. Google-style documentation:

- Includes:
  - Arguments (`filename`)
  - Return type (`str`)
  - Potential exceptions (`FileNotFoundError`, `IOError`)
- Essential for **file operations**, where errors are common.
- Helps other developers **anticipate and handle exceptions properly**.

**Recommendation:** Google-style is the best choice for **robust, production-ready file handling functions**, because it communicates all necessary details to users, including exceptions.