AI ASSISTANT CODING

LAB - 9.5

NAME : G SAIANIRUDH

HTNO : 2303A51891

BATCH : 04

Lab Experiment:

Documentation Generation -Automatic documentation and code comments

Lab Objectives :

1. To understand automatic documentation generation.

2. To generate code comments and docstrings using AI tools.

3. To learn the importance of documentation in software development.

Lab Outcomes :

1. Students will be able to generate documentation automatically for code.

2. Students will be able to add clear comments and docstrings to programs.

3.Students will be able to improve code readability and maintainability using documentation.

Problem 1: String Utilities Function

Consider the following Python function:

def reverse_string(text):

return text[::-1]

Task:

1. Write documentation in:

(a) Docstring :

```
Lab9.5.py > ...
  1    '''Add a professional Python docstring to this function:
  2
  3    def reverse_string(text):
  4        return text[::-1]
  5    '''
  6    def reverse_string(text):
  7        """
  8        Reverses the given string.
  9
 10        Parameters:
 11        text (str): The string to be reversed.
 12
 13        Returns:
 14        str: The reversed string.
 15        """
 16        return text[::-1]
 17    # Example usage:
 18    input_string = "Hello, World!"
 19    reversed_string = reverse_string(input_string)
 20    print(reversed_string)  # Output: !dlroW ,olleH
 21
 22
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\aniru\OneDrive\AI_Assitant_Coding> & C:/Python314/python.
I_Assitant_Coding/Lab9.5.py
!dlroW ,olleH
```

(b) Inline comments :

```
 22
 23    '''Add clear inline comments to this function:
 24
 25    def reverse_string(text):
 26        return text[::-1] '''
 27    def reverse_string(text):
 28        # Reverse the input string using slicing
 29        return text[::-1]
 30    # Example usage:
 31    input_string = "Hello, World!"
 32    reversed_string = reverse_string(input_string)
 33    print(reversed_string)  # Output: !dlroW ,olleH
 34
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\aniru\OneDrive\AI_Assitant_Coding> & C:/Python314/pyt
I_Assitant_Coding/Lab9.5.py
!dlroW ,olleH
PS C:\Users\aniru\OneDrive\AI_Assitant_Coding>
```

(c) Google-style documentation

```
 34
 35    '''Rewrite this function with Google-style docstring:
 36
 37    def reverse_string(text):
 38        return text[::-1]'''
 39    def reverse_string(text):
 40        """Reverses the given string.
 41        Args:
 42            text (str): The string to be reversed.
 43        Returns:
 44            str: The reversed string.
 45        """
 46        return text[::-1]
 47    # Example usage:
 48    input_string = "Hello, World!"
 49    reversed_string = reverse_string(input_string)
 50    print(reversed_string)  # Output: !dlroW ,olleH
 51
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

```
PS C:\Users\aniru\OneDrive\AI_Assitant_Coding> & C:/Python314/python.exe c:
I_Assitant_Coding/Lab9.5.py
!dlroW ,olleH
PS C:\Users\aniru\OneDrive\AI_Assitant_Coding>
```

2. Compare the three documentation styles.

```
'''Compare docstring, inline comments, and Google-style documentation.
Explain differences in readability and usefulness.'''

Docstrings, inline comments, and Google-style documentation
serve different purposes in enhancing code readability and usefulness.
```

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

```
'''Which documentation style is best for a utility-based string library?
Justify briefly.'''

For a utility-based string library, Google-style documentation is often the best choice.
This is because it provides a clear and structured format for documenting functions, including
parameters, return values, and any exceptions that may be raised. This level of detail is
particularly useful for a library that may be used by other developers, as it helps them understand
how to use the functions correctly and what to expect from them. Additionally, Google-style documentation
can be easily parsed by tools that generate API documentation, making it more accessible for users of the library.
```

Justification:

Google-style documentation is most suitable for a utility-based string library because it provides a **clear, structured, and standardized format**. It improves readability, makes functions easier to understand, and works well with automatic documentation tools like pydoc. This style is widely used in professional projects, enhancing maintainability and consistency.

Problem 2: Password Strength Checker
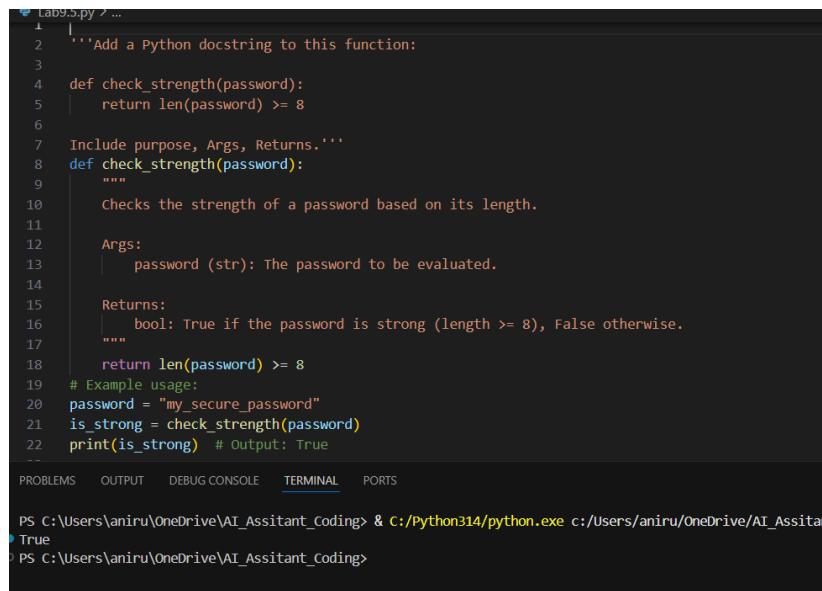
Consider the function:

def check_strength(password):

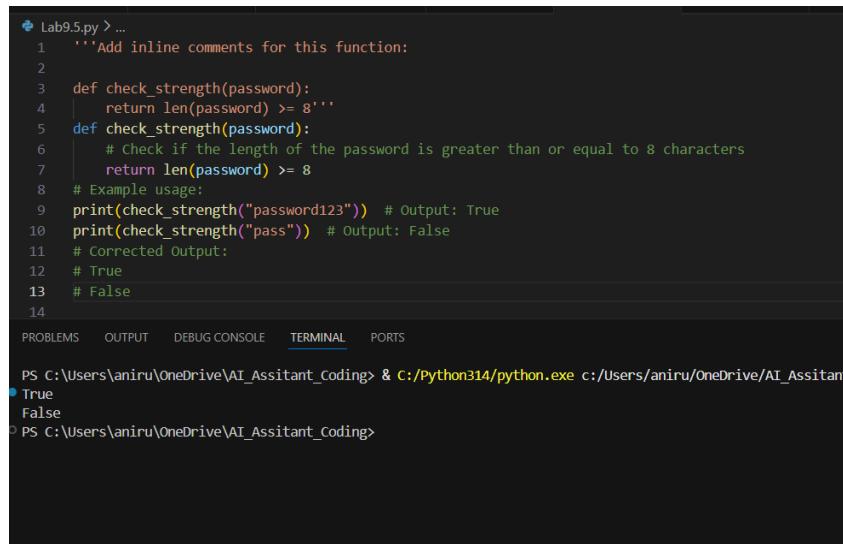**return len(password) >= 8**

Task:

1. Document the function using docstring, inline comments, and Google style.

(a) Docstring:

```
Lab9.5.py > ...
1
2    '''Add a Python docstring to this function:
3
4    def check_strength(password):
5        return len(password) >= 8
6
7    Include purpose, Args, Returns.'''
8    def check_strength(password):
9        """
10       Checks the strength of a password based on its length.
11
12       Args:
13           password (str): The password to be evaluated.
14
15       Returns:
16           bool: True if the password is strong (length >= 8), False otherwise.
17       """
18       return len(password) >= 8
19   # Example usage:
20   password = "my_secure_password"
21   is_strong = check_strength(password)
22   print(is_strong)  # Output: True
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   **TERMINAL**   PORTS

```
PS C:\Users\aniru\OneDrive\AI_Assitant_Coding> & C:/Python314/python.exe c:/Users/aniru/OneDrive/AI_Assita
True
PS C:\Users\aniru\OneDrive\AI_Assitant_Coding>
```

(b) Inline

```
Lab9.5.py > ...
1    '''Add inline comments for this function:
2
3    def check_strength(password):
4        return len(password) >= 8'''
5    def check_strength(password):
6        # Check if the length of the password is greater than or equal to 8 characters
7        return len(password) >= 8
8    # Example usage:
9    print(check_strength("password123"))  # Output: True
10   print(check_strength("pass"))  # Output: False
11   # Corrected Output:
12   # True
13   # False
14
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   **TERMINAL**   PORTS

```
PS C:\Users\aniru\OneDrive\AI_Assitant_Coding> & C:/Python314/python.exe c:/Users/aniru/OneDrive/AI_Assitant
True
False
PS C:\Users\aniru\OneDrive\AI_Assitant_Coding>
```

c Google style

```
Lab9.5.py > ...
  1    '''Document this function using Google-style docstring:
  2
  3    def check_strength(password):
  4        return len(password) >= 8'''
  5    def check_strength(password):
  6        """Check the strength of a password.
  7
  8        A password is considered strong if it has at least 8 characters.
  9
 10        Args:
 11            password (str): The password to check.
 12        Returns:
 13            bool: True if the password is strong, False otherwise.
 14        """
 15        return len(password) >= 8
 16    print(check_strength("password123"))  # Output: True
 17    print(check_strength("pass"))  # Output: False
 18    # Corrected Output:
 19    # True
 20    # False
 21
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\aniru\OneDrive\AI_Assitant_Coding> & C:/Python314/python.exe c:/Users/aniru/OneDrive/AI_Assita
● True
 False
○ PS C:\Users\aniru\OneDrive\AI_Assitant_Coding>
```

2. Compare documentation styles for security-related code.

```
'''Compare docstring, inline comments, and Google-style documentation.
Explain differences in readability and usefulness.'''
# Docstring style provides a clear and structured way to document the function's purpose, arguments, and return value.
# It is useful for generating documentation and is easily accessible through help() in Python.
# Inline comments can provide quick explanations of specific lines of code, but they can become cluttered and less readable if overused.
#   They are best used for complex logic that may not be immediately clear.
# Google-style docstrings are similar to regular docstrings but follow a specific format that includes sections for Args and Returns.
# This can enhance readability and consistency, especially in larger codebases where multiple developers are contributing.
# It provides a clear and standardized way to document functions, making it easier for others to understand the code quickly.
```

3. Recommend the most appropriate style.

```
'''Which documentation style is best for a utility-based string library?
Justify briefly.'''
# For a utility-based string library, the Google-style docstring is often the best choice.
# This is because it provides a clear and standardized format for documenting the purpose of each function, its parameters, and its return values.
# This can enhance readability and maintainability, especially in a library that may have many functions and be used by multiple developers.
# The structured format of Google-style docstrings can help users quickly understand how to use the functions and what to expect from them, which is crucial
# for a utility library that may be widely used.
```

Justification:

Security-related code requires **precise and unambiguous documentation**. Google-style docstrings clearly define validation rules, parameters, return values, and potential exceptions. This reduces misuse, helps during code reviews and audits, and ensures developers correctly understand the function's behavior.

Problem 3: Math Utilities Module

Task:

1. Create a module math_utils.py with functions:

Generate docstrings automatically using AI tools.

o square(n)

```
Lab9.5.py > ...
  1    '''Create a Python module math_utils.py with functions:
  2
  3    square(n)
  4    cube(n)
  5    factorial(n)
  6
  7    Include Google-style docstrings.
  8    Factorial should raise ValueError for negative input.'''
  9    def square(n):
 10        """Returns the square of a number.
 11
 12        Args:
 13            n (int or float): The number to be squared.
 14
 15        Returns:
 16            int or float: The square of the input number.
 17        """
 18        return n ** 2
 19    def cube(n):
 20        """Returns the cube of a number.
 21
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   **TERMINAL**   PORTS

PS C:\Users\aniru\OneDrive\AI_Assitant_Coding> & C:/Python314/python.exe c:/
16

o cube(n)

```
 18        return n ** 2
 19    def cube(n):
 20        """Returns the cube of a number.
 21
 22        Args:
 23            n (int or float): The number to be cubed.
 24
 25        Returns:
 26            int or float: The cube of the input number.
 27        """
 28        return n ** 3
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   **TERMINAL**   PORTS

PS C:\Users\aniru\OneDrive\AI_Assitant_Coding> & C:/Python314/python.exe c:/Users/ar
● 64

o factorial(n)

```python
Lab9.5.py > ...
29    def factorial(n):
30        """Returns the factorial of a non-negative integer.
31
32        Args:
33            n (int): The number to compute the factorial of. Must be non-negative.
34
35        Returns:
36            int: The factorial of the input number.
37
38        Raises:
39            ValueError: If n is negative.
40        """
41        if n < 0:
42            raise ValueError("Input must be a non-negative integer.")
43        elif n == 0 or n == 1:
44            return 1
45        else:
46            result = 1
47            for i in range(2, n + 1):
48                result *= i
49            return result
50    print(factorial(5))  # Output: 120
51
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

● PS C:\Users\aniru\OneDrive\AI_Assitant_Coding> & C:/Python314/python.exe c:/Users/aniru/OneDrive/AI_
  120
○ PS C:\Users\aniru\OneDrive\AI_Assitant_Coding>
```

3. Export documentation as an HTML file.

```
'''Explain how to generate HTML documentation for math_utils.py
using pydoc in terminal.'''
'''To generate HTML documentation for the `math_utils.py` module using `pydoc`, you can follow these steps:
1. Open your terminal or command prompt.
2. Navigate to the directory where your `math_utils.py` file is located using the `cd` command. For example:
   ```
   cd path/to/your/directory
   ```
3. Use the following command to generate HTML documentation for the `math_utils` module:
   ```
   pydoc -w math_utils
   ```
4. This command will create an HTML file named `math_utils.html` in the same directory. You can open this file in a web browser to view the documentation
for the functions defined in `math_utils.py`.
'''
```

Justification:

Docstrings help describe the purpose, inputs, outputs, and exceptions of each function. They enable automatic documentation generation and make the module easier for other developers to understand and reuse.

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

```
'''How to view documentation of attendance.py
in terminal and browser using pydoc?'''
To view the documentation of the `attendance.py` module using `pydoc`, you can follow these steps:
1. Open your terminal or command prompt.
2. Navigate to the directory where your `attendance.py` file is located using the `cd`
command. For example:
   ```bashcd path/to/your/directory

3. To view the documentation in the terminal, run the following command:
   ```bashpydoc attendance
   ```This will display the documentation for the `attendance` module directly in the terminal.
4. To view the documentation in a web browser, run the following command:
   ```bashpydoc -b attendance
   ```This will open the documentation for the `attendance` module in your default web browser, allowing you to browse through it more easily.
```

Justification :

Documentation ensures developers understand how attendance is stored, updated, and retrieved. It improves collaboration, reduces confusion, and supports maintainability.

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.

```python
Lab9.5.py > ...
1    '''Add a detailed Python docstring with Raises section:
2
3    def read_file(filename):
4        with open(filename, 'r') as f:
5            return f.read()'''
6    def read_file(filename):
7        """
8        Reads the contents of a file and returns it as a string.
9
10       Args:
11           filename (str): The name of the file to be read.
12       Returns:
13           str: The contents of the file.
14       Raises:
15           FileNotFoundError: If the file does not exist.
16           IOError: If there is an error reading the file.
17       """
18       try:
19           with open(filename, 'r') as f:
20               return f.read()
21       except FileNotFoundError:
22           raise FileNotFoundError(f"File '{filename}' not found.")
23       except IOError:
24           raise IOError(f"Error reading file '{filename}'.")
25
26
```

```
'''Add inline comments explaining file handling:

def read_file(filename):
    with open(filename, 'r') as f:
        return f.read()'''
def read_file(filename):
    """
    Reads the contents of a file and returns it as a string.

    Args:
        filename (str): The name of the file to be read.
    Returns:
        str: The contents of the file.
    Raises:          FileNotFoundError: If the file does not exist.
        IOError: If there is an error reading the file.
    """
    try:
        # Open the file in read mode ('r')
        with open(filename, 'r') as f:
            # Read the contents of the file and return it
            return f.read()
    except FileNotFoundError:
        # Raise an error if the file is not found
        raise FileNotFoundError(f"File '{filename}' not found.")
    except IOError:
        # Raise an error if there is an issue reading the file
        raise IOError(f"Error reading file '{filename}'.")
```

```
'''Rewrite this function with Google-style docstring
including exception details:

def read_file(filename):
    with open(filename, 'r') as f:
        return f.read()
'''
def read_file(filename):
    """
    Reads the contents of a file and returns it as a string.

    Args:
        filename (str): The name of the file to be read.
    Returns:
        str: The contents of the file.
    Raises:
        FileNotFoundError: If the file does not exist.
        IOError: If there is an error reading the file.
    """
    try:
        with open(filename, 'r') as f:
            return f.read()
    except FileNotFoundError:
        raise FileNotFoundError(f"File '{filename}' not found.")
    except IOError:
        raise IOError(f"Error reading file '{filename}'.")
```

2. Identify which style best explains exception handling.

```
'''Which documentation style best explains exception handling?
Compare docstring vs inline vs Google-style.'''

'''The best documentation style for explaining exception handling depends on the context and the audience.
- Docstring: This style is useful for providing a high-level overview of the function, including its purpose, parameters,
  return value, and exceptions. It is typically used in libraries and APIs where users need to understand the function's
behavior without diving into the implementation details.'''
```

3. Justify your recommendation.

```
'''Justify why Google-style documentation is preferred for explaining exceptions.

Google-style documentation is often preferred for explaining exceptions because it provides a clear and structured format
that is easy to read and understand. The "Raises" section in Google-style docstrings explicitly lists the exceptions that
can be raised by the function, along with a brief description of each exception. This makes it easier for developers to quickly
identify potential error conditions and understand how to handle them effectively. Additionally, Google-style documentation is
widely adopted in the Python community, making it a familiar and consistent choice for documenting exceptions.'''
```