

Assignment-9.4

Name: Sushma Purella

H. No:2303A52188

Batch:44

Task 1: Auto-Generating Function Documentation in a Shared Codebase

Ask the AI to automatically generate Google-style function docstrings for each function

- Each docstring should include:
- A brief description of the function
- Parameters with data types
- Return values
- At least one example usage

Prompt:

"Generate Google-style docstrings for the following Python function."

Code:

```
Assignment-9.4.py > classify_number
1 #automatically generate Google-style function docstrings for each function
2 ## Each docstring should include:
3 #o A brief description of the function
4 #o Parameters with data types
5 #o Return values
6 #o At least one example usage (if applicable)
7 #Experiment with different prompting styles (zero-shot or context-based)
8 #to observe quality differences.
9 def classify_number(n):
10     """Classifies a number as Positive, Negative, or Zero.
11
12     Args:
13         n (int or float): The number to be classified. Can also be None or a string for invalid input
14
15     Returns:
16         str: A string indicating whether the number is "Positive", "Negative", "Zero", or "Invalid input"
17
18     Example:
19         >>> classify_number(10)
20         'Positive'
21         >>> classify_number(-5)
22         'Negative'
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
python313/python.exe c:/Users/sushm/OneDrive/Attachments/Desktop/Ai-Assist/Assignment-9.4.py
○ Positive
Negative
Zero
Invalid input
Invalid input
```

```
18     Example:
19         >>> classify_number(10)
20             'Positive'
21         >>> classify_number(-5)
22             'Negative'
23         >>> classify_number(0)
24             'Zero'
25         >>> classify_number("invalid")
26             'Invalid input'
27         >>> classify_number(None)
28             'Invalid input'
29
30     """
31     if n is None or isinstance(n, str):
32         return "Invalid input"
33     elif n > 0:
34         return "Positive"
35     elif n < 0:
36         return "Negative"
37     else:
38         return "Zero"

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

python313/python.exe c:/Users/sushm/OneDrive/Attachments/Desktop/Ai-Assist/Assignment-9.4.py

- Positive
- Negative
- Zero
- Invalid input
- Invalid input

```

36     else:
37         return "Zero"
38 # Assert test cases
39 assert classify_number(10) == "Positive" # Test with positive number
40 assert classify_number(-5) == "Negative" # Test with negative number
41 assert classify_number(0) == "Zero" # Test with zero
42 assert classify_number("invalid") == "Invalid input" # Test with string input
43 assert classify_number(None) == "Invalid input" # Test with None input
44 print(classify_number(10))
45 print(classify_number(-5))
46 print(classify_number(0))
47 print(classify_number("invalid"))
48 print(classify_number(None))
49

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

`python313/python.exe c:/Users/sushm/OneDrive/Attachments/Desktop/Ai-Assist/Assignment-9.4.py`

- Positive
- Negative
- Zero
- Invalid input
- Invalid input

Observation:

- The AI generated a basic docstring.
- Description and parameters were included.
- Examples were sometimes missing or minimal.
- Formatting was correct but less detailed.

Task 2: Enhancing Readability Through AI-Generated Inline Comments

You are provided with a Python script containing:

- Loops
- Conditional logic
- Algorithms (such as Fibonacci sequence, sorting, or searching)

Use AI assistance to:

- Automatically insert inline comments only for complex or non-obvious logic
 - Avoid commenting on trivial or self-explanatory syntax

The goal is to improve clarity without cluttering the code.

Prompt:

Insert meaningful inline comments in the following Python code.
Comment only complex or non-obvious logic and avoid explaining
basic Python syntax

Code:

```
74     return sequence
75 # Assert test cases
76 assert fibonacci(0) == [] # Test with n = 0
77 assert fibonacci(1) == [0] # Test with n = 1
78 assert fibonacci(5) == [0, 1, 1, 2, 3] # Test with n = 5
79 assert fibonacci(10) == [0, 1, 1, 2, 3, 5, 8, 13, 21, 34] # Test with n = 10
80 print(fibonacci(0)) # Should return []
81 print(fibonacci(1)) # Should return [0]
82 print(fibonacci(5)) # Should return [0, 1, 1, 2, 3]
83 print(fibonacci(10)) # Should return [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
84
85
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS +

```
Invalid input
Invalid input
[]
[0]
[0, 1, 1, 2, 3]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Observation

- The AI added comments only where the logic was not immediately obvious.
- The tuple update in the Fibonacci algorithm was clearly explained.
- Trivial lines like variable initialization and loop syntax were not commented.
- Code readability improved without adding unnecessary clutter.

Task 3: Generating Module-Level Documentation for a Python Package

Provide a complete Python module to an AI tool and instruct it to automatically generate a module-level docstring at the top of the file that includes:

- The purpose of the module

- Required libraries or dependencies
- A brief description of key functions and classes
- A short example of how the module can be used

Prompt:

Generate a professional module-level docstring for this Python module. Include the purpose, dependencies, key functions, and example usage

Code:

```

98     Dependencies:
99     - No external libraries are required.
100    Key Functions:
101    - classify_number(n): Classifies a number as Positive, Negative, or Zero, and handles invalid inputs.
102    - fibonacci(n): Generates the Fibonacci sequence up to the nth number.
103    Example Usage:""""
104    print(classify_number(10)) # Output: 'Positive'
105    print(classify_number(-5)) # Output: 'Negative'
106    print(classify_number(0)) # Output: 'Zero'
107    print(classify_number("invalid")) # Output: 'Invalid input'
108    print(classify_number(None)) # Output: 'Invalid input'
109    print(fibonacci(10)) # Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
110
111
112

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\sushm\OneDrive\Attachments\Desktop\Ai-Assist> & C:/Users/sushm/AppData/Local/Programs/Python/Python313/python.exe c:/Users/sushm/OneDrive/Attachments/Desktop/Ai-Assist/Assignment-9.4.py

Zero
Invalid input
Invalid input
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Observation

- The AI generated a structured multi-line docstring at the top of the file.
- The documentation clearly explained the module's purpose and functionality.
- Dependencies and key functions were summarized concisely.
- Example usage improved clarity for new users of the module.

- The tone was professional and suitable for real-world repositories.

Task 4: Converting Developer Comments into Structured Docstrings

You are given a Python script where functions contain detailed inline comments explaining their logic.

Use AI to:

- Automatically convert these comments into structured Google-style or NumPy-style docstrings
- Preserve the original meaning and intent of the comments
- Remove redundant inline comments after conversion

Prompt:

Convert the following inline explanatory comments into a structured Google-style docstring. Preserve the original meaning and remove redundant inline comments

Code:

```
120
121
122 def calculate_average(numbers):
123     # This function takes a list of numbers as input.
124     # It first checks if the list is empty.
125     # If the list is empty, it returns 0 to avoid division by zero.
126     # Otherwise, it calculates the total sum of the numbers.
127     # Then it divides the total by the number of elements
128     # to compute the average value.
129
130     if not numbers:
131         return 0
132
133     total = sum(numbers)
134     average = total / len(numbers)
135     return average
136
137
138 def calculate_average(numbers):
139     """
140         Calculates the average of a list of numbers.
141
142     Args:
143         numbers (list of int or float): A list containing numeric values.
144
145     Returns:
146         float: The average of the numbers in the list.
147         | Returns 0 if the list is empty.
148
149     Examples:
150         >>> calculate_average([10, 20, 30])
151         20.0
152         >>> calculate_average([])
153         0
154     """
155     if not numbers:
156         return 0
157
158     total = sum(numbers)
159     return total / len(numbers)
160
161 # Assert test cases
162 assert calculate_average([10, 20, 30]) == 20.0 # Test with a list of numbers
163 assert calculate_average([]) == 0 # Test with an empty list
164 print(calculate_average([10, 20, 30])) # Should return 20.0
165 print(calculate_average([])) # Should return 0
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
20.0
0
```

```
143     numbers (list of int or float): A list containing numeric values.
144
145     Returns:
146         float: The average of the numbers in the list.
147         | Returns 0 if the list is empty.
148
149     Examples:
150         >>> calculate_average([10, 20, 30])
151         20.0
152         >>> calculate_average([])
153         0
154     """
155     if not numbers:
156         return 0
157
158     total = sum(numbers)
159     return total / len(numbers)
160
161 # Assert test cases
162 assert calculate_average([10, 20, 30]) == 20.0 # Test with a list of numbers
163 assert calculate_average([]) == 0 # Test with an empty list
164 print(calculate_average([10, 20, 30])) # Should return 20.0
165 print(calculate_average([])) # Should return 0
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
20.0
0
```

Observation:

- The AI successfully converted descriptive inline comments into a structured docstring.
- The original meaning of the logic was preserved.

- Redundant inline comments were removed.
- The function body became cleaner and more readable.
- Documentation format became consistent with professional standards.

Task 5: Building a Mini Automatic Documentation Generator

Design a small Python utility that:

- Reads a given .py file
- Automatically detects:
 - Functions
 - Classes
- Inserts placeholder Google-style docstrings for each detected function or class

Prompt:

Help me build a Python utility that reads a .py file, detects functions and classes using AST, and inserts placeholder Google-style docstrings where missing

Code:

```
168  #Design a small Python utility that:
169  ## Reads a given .py file
170  ## Automatically detects:
171  ## Functions
172  ## Classes
173  ## Inserts placeholder Google-style docstrings for each detected function or class
174  import ast
175
176  def insert_docstrings(file_path):
177      with open(file_path, 'r') as file:
178          source = file.read()
179      tree = ast.parse(source)
180      for node in ast.walk(tree):
181          if isinstance(node, (ast.FunctionDef, ast.ClassDef)):
182              if not ast.get_docstring(node):
183                  docstring = f"TODO: Add Google-style docstring for {node.name}."
184                  node.body.insert(
185                      0,
186                      ast.Expr(value=ast.Constant(value=docstring)))
187
188      new_source = ast.unparse(tree)
189      with open(file_path, 'w') as file:
190          file.write(new_source)
191
192
```

Observation:

- AI suggested using the `ast` module to parse and modify the syntax tree.
- Functions and classes were automatically detected.
- Placeholder docstrings were inserted only where missing.
- The tool successfully scaffolded documentation without affecting existing docstrings.
- Demonstrates automation of documentation setup.