SR UNIVERSITY

AI Assistant Coding
Assignment-9.4

Name of Student	**: V. Abhinav**	**Batch : 41**
Enrollment No.	**: 2303A52486**	

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

You are given a Python script containing multiple functions without any docstrings.
Using an AI-assisted coding tool:

- **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 (if applicable)**

Experiment with different prompting styles (zero-shot or context-based) to observe quality differences.

Original Code:

Assign 8.py > filter_even

```python
def add(a, b):
    return a + b


def factorial(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result


def filter_even(numbers):
    return [x for x in numbers if x % 2 == 0]
```

Prompt:
For the above code generate Google-style function docstrings for each function, include description , parameters , return and example usage.

AI-Generated Code with Google Style Docstrings:

```
Assign 8.py > ...
15  def add(a, b):
23          int | float: The sum of the two input numbers.
24
25      Examples:
26          >>> add(2, 3)
27          5
28          >>> add(1.5, 2.5)
29          4.0
30      """
31      return a + b
32  | Ctrl+L to chat, Ctrl+K to generate
33
34  def factorial(n):
35      """Compute the factorial of a non-negative integer.
36
37      Args:
38          n (int): A non-negative integer whose factorial is to be computed.
39
40      Returns:
41          int: The factorial of the input integer.
42
43      Raises:
44          ValueError: If `n` is negative.
45
46      Examples:
47          >>> factorial(0)
48          1
49          >>> factorial(5)
50          120
51      """
52      if n < 0:
53          raise ValueError("n must be non-negative")
54      result = 1
55      for i in range(2, n + 1):
56          result *= i
57      return result
```

Explanation:

This task focuses on using AI to automatically generate Google-style docstrings for existing Python functions that originally lacked documentation. The AI analyzes function logic and produces structured docstrings including descriptions, parameters, return values, and example usage. Minor syntax issues in the original code, such as missing colons, were identified and corrected to allow proper parsing. The output demonstrates well-formatted and standardized documentation, improving code readability and maintainability.

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.

Original Code:

```
100    def bubble_sort(arr):
101        n = len(arr)
102        for i in range(n):
103            swapped = False
104            for j in range(0, n - i - 1):
105                if arr[j] > arr[j + 1]:
106                    arr[j], arr[j + 1] = arr[j + 1], arr[j]
107                    swapped = True
108            if not swapped:
109                break
110        return arr
111
112
113    def process_numbers(nums):
114        result = []
115        for i, x in enumerate[Any](nums):
116            if x % 2 == 0 and i % 3 == 0:
117                result.append(x * 2)
118            elif x % 5 == 0:
119                result.append(x - 1)
120            else:
121                if x > 10:
122                    result.append(x // 2)
123                else:
124                    result.append(x)
125        return result
```

Prompt:

Add concise inline comments to the above Python codes only where the logic is complex or non-obvious, focusing on explaining why the code is written that

way or what the key steps in the algorithm are; do not explain basic Python syntax (such as what a loop, if-statement, or return does), avoid commenting on trivial or self-explanatory lines, and keep each comment short and placed either on the line above or at the end of the relevant line.

Code with inline Comments:

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            # Compare adjacent elements and swap to "bubble up" larger values
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # If no swaps occurred in this pass, the list is already sorted
        if not swapped:
            break
    return arr


def process_numbers(nums):
    result = []
    for i, x in enumerate[Any](nums):
        # Double even numbers that appear at positions divisible by 3
        if x % 2 == 0 and i % 3 == 0:
            result.append(x * 2)
        # Slightly reduce numbers that are multiples of 5
        elif x % 5 == 0:
            result.append(x - 1)
        else:
            # For larger numbers, scale them down; keep smaller ones unchanged
            if x > 10:
                result.append(x // 2)
            else:
                result.append(x)
    return result
```

Explanation:

In this task, AI is used to insert concise inline comments only where the logic is complex or non-obvious. The code includes algorithms like bubble sort and conditional number processing, where comments explain the reasoning behind optimizations and decision branches. Trivial Python syntax is intentionally left uncommented to avoid clutter. The output shows clearer algorithm flow, making the code easier to understand without over-documentation.

SR UNIVERSITY

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**

Focus on clarity and professional tone.

Original Code:

Assign 8.py > ...

```
142     class Circle:
143         def __init__(self, radius: float):
144             self.radius = radius
145
146         def area(self) -> float:
147             return math.pi * self.radius ** 2
148
149         def circumference(self) -> float:
150             return 2 * math.pi * self.radius
151
152
153     def calculate_distances(points: List[tuple]) -> List[f
154         distances = []
155         for x, y in points:
156             distances.append(math.sqrt(x**2 + y**2))
157         return distances
158
159
160     def normalize(values: List[float]) -> List[float]:
161         if not values:
162             return []
163         max_val = max(values)
164         if max_val == 0:
165             return [0.0 for _ in values]
166         return [v / max_val for v in values]
167
168
169     def main():
170         c = Circle(5)
171         print("Area:", c.area())
172         print("Circumference:", c.circumference())
173
174         pts = [(3, 4), (0, 5), (6, 8)]
175         dists = calculate_distances(pts)
176         print("Distances:", dists)
177         print("Normalized distances:", normalize(dists))
178
179
180     if __name__ == "__main__":
```

Prompt:

Generate a professional, multi-line module-level docstring to insert at the very top of this Python file. The docstring should describe the module's overall purpose, its main dependencies, briefly summarize the key functions and classes, and include a short example of typical usage. Do not change any existing code; only add the docstring.

AI-Generated Module-Level Docstring:

```
# Task-3:
"""
Utility module providing simple numeric helpers for geometry and vector-like
operations, including circle measurements, distance calculations, and value
normalization. Designed as an example of a small, self-contained Python module
with clear functions and a lightweight class.

Dependencies:
    - math: for square roots and π-based calculations.
    - typing.List: for type-annotated collections of numeric values.

Key Components:
    - class Circle: Represents a circle with a given radius and provides
      methods to compute its area and circumference.
    - calculate_distances(points): Computes Euclidean distances of (x, y)
      points from the origin.
    - normalize(values): Scales a list of numeric values into the range
      \[0.0, 1.0] based on the maximum value.
    - main(): Demonstrates basic usage of the Circle class and helper
      functions when the module is run as a script.

Example:
    >>> from assign_8 import Circle, calculate_distances, normalize
    >>> c = Circle(5)
    >>> c.area()
    78.53981633974483
    >>> points = [(3, 4), (0, 5), (6, 8)]
    >>> dists = calculate_distances(points)
    >>> dists
    [5.0, 5.0, 10.0]
    >>> normalize(dists)
    [0.5, 0.5, 1.0]
"""
```

Explanation: This task involves generating a professional module-level docstring for an entire Python file using AI assistance. The generated documentation clearly explains the module's purpose, dependencies, key classes and functions, and provides an example of usage. The original code remains unchanged, ensuring functional integrity. The output reflects industry-standard documentation practices that improve usability and professionalism of the module.

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**

Original Code:

```
def compute_average(scores):
    # This function takes a list of numeric scores and computes the average.
    # It ignores any None values in the list because those represent missing
    # or invalid entries that should not affect the result.
    # If, after filtering, there are no valid scores left, it returns None
    # to indicate that an average cannot be computed.
    valid_scores = [s for s in scores if s is not None]
    if not valid_scores:
        return None
    total = sum(valid_scores)
    count = len(valid_scores)
    return total / count


def find_top_student(students):
    # The students parameter is a list of dictionaries like:
    # {"name": "Alice", "score": 92}
    # The function loops through the list to find the student with the
    # highest score. If the list is empty, it returns None to indicate that
    # there is no top student.
    if not students:
        return None
    top = students[0]
    for s in students[1:]:
        # If this student's score is higher than the current top, update top.
        if s["score"] > top["score"]:
            top = s
    return top
```

SR UNIVERSITY

Prompt:
Generate structured Google-style docstrings for each function in the following code by converting the existing detailed inline comments into docstrings, preserving their meaning. Remove any now-redundant inline comments while keeping the code logic unchanged. Do not modify variable names or functionality.

Converted code:

```
def compute_average(scores):
    """Compute the average of a list of numeric scores.

    Ignores any None values, which represent missing or invalid entries.
    If there are no valid scores, returns None to indicate that an
    average cannot be computed.

    Args:
        scores (list[float | None]): List of numeric scores that may
            include None for missing values.

    Returns:
        float | None: The average of all non-None scores, or None if no
        valid scores are present.
    """
    valid_scores = [s for s in scores if s is not None]
    if not valid_scores:
        return None
    total = sum(valid_scores)
    count = len(valid_scores)
    return total / count


def find_top_student(students):
    """Find the student with the highest score.

    Expects each student to be represented as a dictionary with at least
    a "name" and a "score" key. If the list is empty, returns None.

    Args:
        students (list[dict]): List of student dictionaries, e.g.
            {"name": str, "score": int | float}.

    Returns:
        dict | None: The dictionary representing the student with the
        highest score, or None if the input list is empty.
    """
    if not students:
        return None
```

Explanation:

This task converts detailed inline developer comments into structured Google-style docstrings. The AI preserves the original intent and logic while removing redundant inline comments after conversion. Functions such as average computation and top-student selection now include clear descriptions, arguments, and return values. The output results in cleaner code with centralized, readable documentation that enhances long-term maintainability.

SR UNIVERSITY

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**

AI tools may be used to assist in generating or refining this utility.

Note: The goal is documentation scaffolding, not perfect documentation.

Python Utility Script:

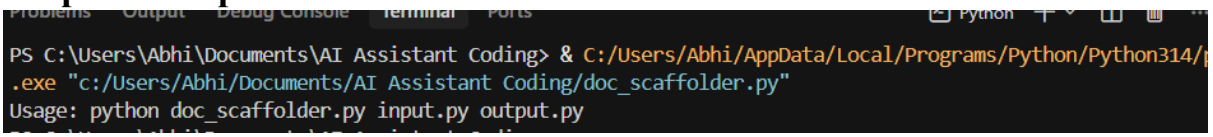
```
import ast
import pathlib
import sys
from typing import Union


def make_placeholder_docstring(node: Union[ast.FunctionDef, ast.AsyncFunctionDef, ast.ClassDef]):
    """Create a simple Google-style placeholder docstring based on the node type."""
    if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)):
        # Collect parameter names (skip self/cls for methods)
        params = [
            arg.arg for arg in node.args.args
            if arg.arg not in {"self", "cls"}
        ]
        params_section = ""
        if params:
            params_section = "    Args:\n" + "".join(
                f"        {name} (type): Description.\n" for name in params
            )

        return (
            f'"""TODO: Add description for function `{node.name}`.\n\n'
            f"{params_section}"
            "    Returns:\n"
            "        type: Description.\n"
            '"""'
        )
    else:  # ClassDef
        return (
            f'"""TODO: Add description for class `{node.name}`.\n\n'
            "    Attributes:\n"
            "        attr (type): Description.\n"
            '"""'
        )


class DocstringInserter(ast.NodeTransformer):
    """AST transformer that inserts placeholder docstrings for functions and classes."""
```

Output Example:



Explanation:

This task designs a Python utility that automatically scans a `.py` file to detect functions and classes using the `ast` module. The script inserts placeholder Google-style docstrings for each detected element, enabling documentation scaffolding. While the generated docstrings are generic, they provide a structured starting point for developers. The output demonstrates how AI-assisted tooling can automate repetitive documentation tasks efficiently.