

AI Assistant Coding

Assignment-9.3

Name: Shaik Nishaaj

Roll No: 2303a51613

Batch:06

Task 1: Basic Docstring Generation

Scenario: You are developing a utility function that processes numerical lists and must be properly documented for future maintenance.

Prompt:

generate a python code to return the sum of even and odd numbers using functions manually add a google docstring for the code

Code

```

❷ sum.py > main
1  def sum_even(numbers: list) -> int:
2      """Calculate the sum of even numbers in a list.
3
4      Args:
5          numbers: A list of integers.
6
7      Returns:
8          The sum of all even numbers in the list.
9
10     Raises:
11         TypeError: If numbers is not a list.
12
13     Example:
14         >>> sum_even([1, 2, 3, 4, 5, 6])
15         12
16
17     """
18
19     if not isinstance(numbers, list):
20         raise TypeError("Input must be a list")
21     return sum(num for num in numbers if num % 2 == 0)
22
23 def sum_odd(numbers: list) -> int:
24     """Calculate the sum of odd numbers in a list.
25
26     Args:
27         numbers: A list of integers.
28
29     Returns:
30         The sum of all odd numbers in the list.
31
32     Raises:
33         TypeError: If numbers is not a list.
34
35     Example:
36         >>> sum_odd([1, 2, 3, 4, 5, 6])
37         9
38
39     """
40
41     if not isinstance(numbers, list):
42         raise TypeError("Input must be a list")
43     return sum(num for num in numbers if num % 2 != 0)
44
45 def main():
46     """Main function to demonstrate even and odd sum calculations.
47
48     This function takes a list of numbers from the user and displays
49     the sum of even numbers and the sum of odd numbers.
50
51     """
52
53     try:
54         user_input = input("Enter numbers separated by spaces: ")
55         numbers = [int(num) for num in user_input.split()]
56
57         even_sum = sum_even(numbers)
58         odd_sum = sum_odd(numbers)
59
60         print(f"\nInput list: {numbers}")
61         print(f"Sum of even numbers: {even_sum}")

```

Output:

- Compare the AI-generated docstring with the manually written docstring

Key sections:

1. Manual vs AI-Generated Documentation - Overview of both approaches
2. Characteristics - What makes each approach unique
3. Google Docstring Structure - Detailed breakdown of each section (Args, Returns, Raises, Examples)
4. Benefits of AI-Generated Docs - Developer, code quality, and maintenance advantages
5. Side-by-side Examples - Practical comparison from your [sum.py](#) code
6. Best Practices - Hybrid approach combining both manual and AI methods

The document shows how your AI-generated Google docstrings are:

- Consistent and standardized
- Complete with type info, exceptions, and examples
- IDE-integrated for better development experience
- Maintainable and searchable.

Task 2: Automatic Inline Comments

Scenario

You are developing a student management module that must be easy to understand for new developers.

Prompt:

generate a python code for sru_student class having
Attributes: name, roll_no, hostel_status, Methods:
fee_update() and display_details() and Manually write
inline comments for each line or logical block

Code:

```

    sru_student.py > sru_student > display_details
1   # sru_student.py - defines the sru_student class with detailed inline comments
2   class sru_student:
3       """Class representing a student at SRU with basic details and fee handling.
4           This class stores name, roll number, hostel status, and outstanding fees,
5           and provides methods to update fees and display student details.
6       """
7       def __init__(self, name: str, roll_no: str, hostel_status: bool, fees_due: float = 0.0):
8           # Initialize a new sru_student instance with provided attributes
9           self.name = name # store the student's full name
10          self.roll_no = roll_no # store the student's roll number or ID
11          self.hostel_status = bool(hostel_status) # store whether the student is in hostel (True/False)
12          self.fees_due = float(fees_due) # store outstanding fees as a float (default 0.0)
13      def fee_update(self, amount: float) -> float:
14          """Apply a payment to the student's outstanding fees and return updated balance.
15          Args:
16              | amount: The payment amount to apply toward `fees_due`.
17          Returns:
18              | The updated `fees_due` after applying the payment.
19          Raises:
20              | TypeError: If `amount` is not a number.
21              | ValueError: If `amount` is negative.
22          """
23          # Validate that the provided amount is numeric (int or float)
24          if not isinstance(amount, (int, float)):
25              raise TypeError("amount must be a number") # raise if bad type
26          # Convert to float for consistent arithmetic
27          payment = float(amount) # local variable holding the payment as float
28          # Do not accept negative payments
29          if payment < 0:
30              raise ValueError("amount must be non-negative") # invalid negative payment
31          # Subtract payment from fees_due but never drop below zero
32          self.fees_due = max(0.0, self.fees_due - payment) # update outstanding fees safely
33          return self.fees_due # return the new fees_due value
34      def display_details(self) -> None:
35          """Print formatted student details to standard output.
36          This shows name, roll number, hostel status (Yes/No), and current fees due.
37          """
38          # Print the student's name
39          print(f"Name: {self.name}")
40          # Print the student's roll number
41          print(f"Roll No: {self.roll_no}")
42          # Print a human-friendly hostel status
43          print(f"Hostel Status: ('Yes' if self.hostel_status else 'No')")
44          # Print the outstanding fees with two decimal places
45          print(f"Fees Due: {self.fees_due:.2f}")
46      # Demonstration block: create an instance and show behavior when run directly
47      if __name__ == "__main__":
48          # Create a sample student with some fees due
49          student = sru_student("Alice Example", "SRU2026", True, fees_due=4500.0)
50
51          # Show details before payment
52          print("Before payment:")
53          student.display_details()
54
55          # Apply a payment and show updated fees
56          student.fee_update(1500) # apply a payment of 1500
57
58          # Show details after payment
59          print("\nAfter payment:")
60          student.display_details()

```

Output:

Comparative Analysis

Purpose & Focus:

- *Manual*: explains developer intent and business reasoning.
- *AI*: describes code behavior, syntax, and structure.

Consistency:

- *Manual*: varies by author.
- *AI*: uniform style across the codebase.

Completeness:

- *Manual*: often partial or missing.
- *AI*: systematically includes Args>Returns/Examples.

Speed:

- *Manual*: slow and effort-intensive.
- *AI*: fast and scalable.

Accuracy:

- *Manual*: high for intent.
- *AI*: good for obvious logic, may miss corner cases.

Strengths of AI Comments

- Rapid generation at scale
- Standardized documentation format
- Good use of type hints
- Helps junior developers
- Easy CI/CD automation

Limitations of AI Comments

- Misses “why” and business context
- Lacks domain knowledge
- Risk of incorrect interpretations
- Can become outdated
- Over-verbose/redundant
- May expose sensitive details
- Lacks human tone and ownership.

Task 3: Module-Level and Function-Level Documentation

Scenario You are building a small calculator module that will be shared across multiple projects and

requires structured documentation.

Prompt:

generate a Python code containing 3–4 functions (e.g., add, subtract, multiply, divide) Manually write NumPy Style docstrings for each function

Code:

```

15  from typing import Union
16  Number = Union[int, float]
17  def add(a: Number, b: Number) -> Number:
18      """Add two numbers.
19      Parameters
20      -----
21      a : int or float
22          First operand.
23      b : int or float
24          Second operand.
25      Returns
26      -----
27      int or float
28          The sum of ``a`` and ``b``. If both inputs are integers the result
29          will be an integer; otherwise a float is returned.
30      Raises
31      -----
32      TypeError
33          If either ``a`` or ``b`` is not a number.
34      Examples
35      -----
36      >>> add(1, 2)
37      3
38      >>> add(1.5, 2.0)
39      3.5
40      """
41      # Validate types
42      if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
43          raise TypeError("Both a and b must be int or float")
44      return a + b
45  def subtract(a: Number, b: Number) -> Number:
46      """Subtract one number from another.

```

```

69  def multiply(a: Number, b: Number) -> Number:
70      """
71      if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
72          raise TypeError("Both a and b must be int or float")
73      return a * b
74  def divide(a: Number, b: Number) -> float:
75      """Divide one number by another.
76      Parameters
77      -----
78      a : int or float
79          Numerator.
80      b : int or float
81          Denominator.
82      Returns
83      -----
84      float
85          The division result as a float.
86      Raises
87      -----
88      TypeError
89          If either ``a`` or ``b`` is not a number.
90      ZeroDivisionError
91          If ``b`` is zero.
92      Examples
93      -----
94      >>> divide(10, 2)
95      5.0
96      >>> divide(5, 2)
97      2.5
98      """
99      if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
100         raise TypeError("Both a and b must be int or float")
101     if b == 0:
102         raise ZeroDivisionError("Denominator b must not be zero")
103     return float(a) / float(b)
104 if __name__ == "__main__":
105     # Simple demonstration when run as a script
106     print("Demo: basic arithmetic functions from math_ops")
107     print("add(2, 3) ->", add(2, 3))
108     print("subtract(5, 2) ->", subtract(5, 2))
109     print("multiply(3, 4) ->", multiply(3, 4))
110     print("divide(7, 2) ->", divide(7, 2))

```

Output:

Comparison between AI-generated docstrings with manually written ones

- Purpose: AI—produce structured docs quickly;
Manual—explain intent, rationale, domain nuance.
- Speed:
AI—instant and scalable; Manual—time-consuming.
- Consistency:
AI—uniform format; Manual—varies by author.
- Completeness:
AI—reliable sections (Args/Returns/Examples); Manual—often incomplete.
- Accuracy:
Manual—better for intent/edge cases; AI—good for surface logic, may misread intent.
- Context:
Manual—aware of history and business rules; AI—limited without prompts.
- Maintainability:
AI—easy to regenerate but can go stale; Manual—accurate if maintained, often neglected.
- Tone & Readability:
AI—neutral and predictable; Manual—team voice with caveats.
- Granularity:
AI—mechanics and types; Manual—“why,” pitfalls, alternatives.
- Risk:
AI—trivial or incorrect details, oversharing; Manual—inconsistency or missing docs.

