AI Assistant Coding

Assignment-9.3

Name: Shah Alam

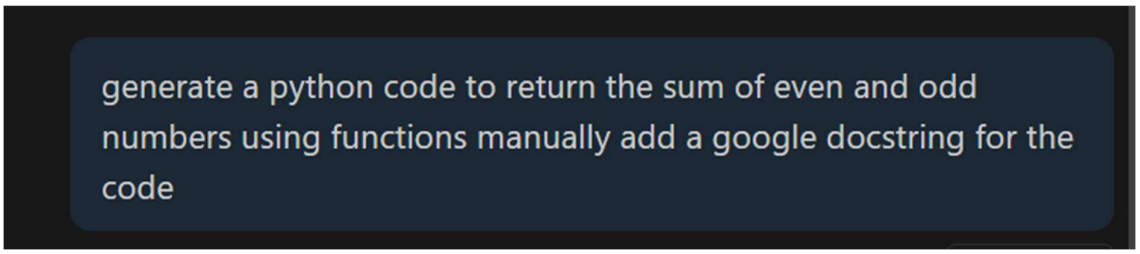
Roll No: 2303A510C1

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:



Code

sum.py > main

```python
def sum_even(numbers: list) -> int:
    """Calculate the sum of even numbers in a list.
    Args:
        numbers: A list of integers.
    Returns:
        The sum of all even numbers in the list.
    Raises:
        TypeError: If numbers is not a list.
    Example:
        >>> sum_even([1, 2, 3, 4, 5, 6])
        12
    """
    if not isinstance(numbers, list):
        raise TypeError("Input must be a list")
    return sum(num for num in numbers if num % 2 == 0)
def sum_odd(numbers: list) -> int:
    """Calculate the sum of odd numbers in a list.
    Args:
        numbers: A list of integers.
    Returns:
        The sum of all odd numbers in the list.
    Raises:
        TypeError: If numbers is not a list.
    Example:
        >>> sum_odd([1, 2, 3, 4, 5, 6])
        9
    """
    if not isinstance(numbers, list):
        raise TypeError("Input must be a list")
    return sum(num for num in numbers if num % 2 != 0)
def main():
    """Main function to demonstrate even and odd sum calculations.

    This function takes a list of numbers from the user and displays
    the sum of even numbers and the sum of odd numbers.
    """
    try:
        user_input = input("Enter numbers separated by spaces: ")
        numbers = [int(num) for num in user_input.split()]

        even_sum = sum_even(numbers)
        odd_sum = sum_odd(numbers)

        print(f"\nInput list: {numbers}")
        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:

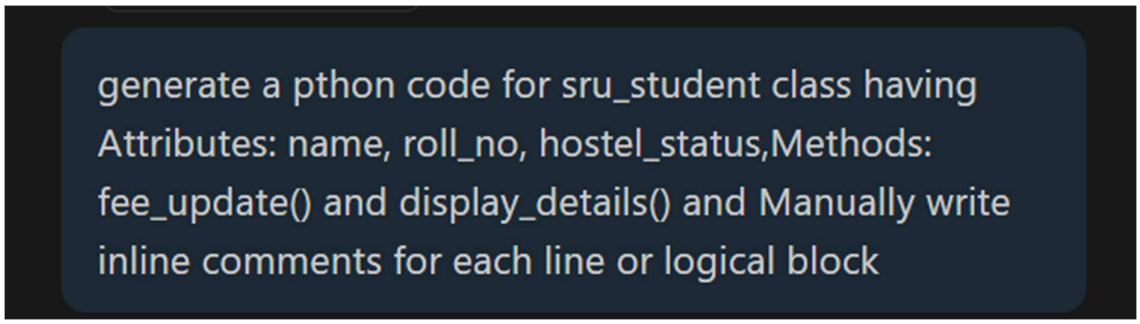
- Consistent and standardized
- Complete with type info, exceptions, and examples
- IDE-integrated for better development experience
- Maintainable and searchable.

Task 2: Automatic Inline Comments

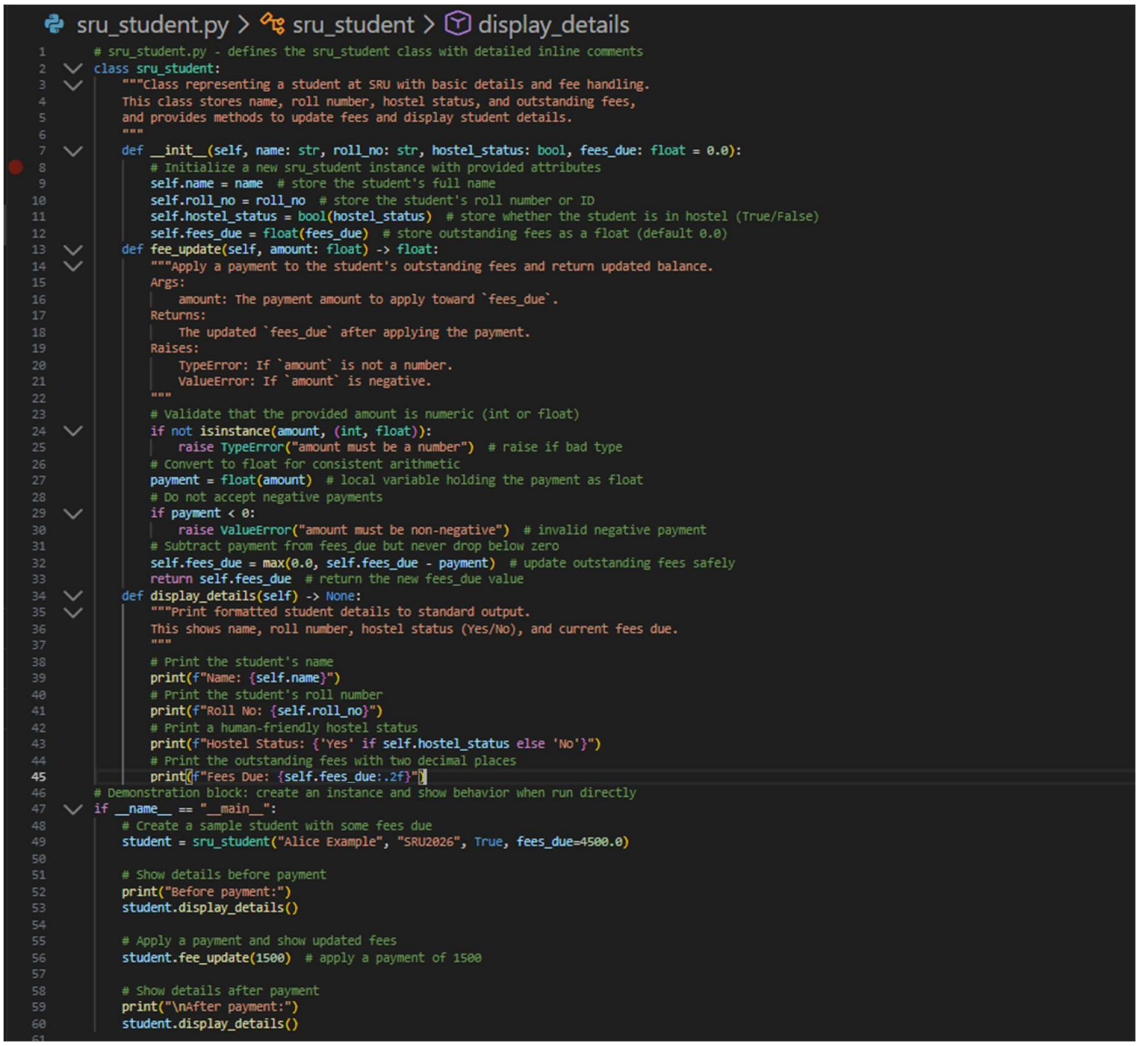
Scenario

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

Prompt:



Code:



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

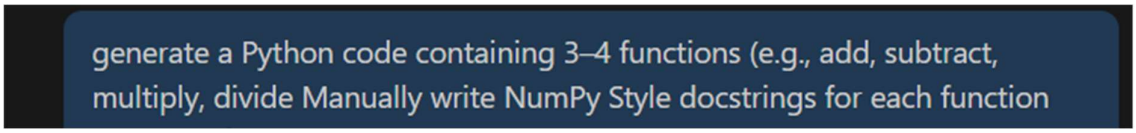
- Misses “why” and business context
- Lacks domain knowledge
- Risk of incorrect interpretations
- Can become outdated
- Over-verbose/redundant
- May expose sensitive details
- Lacks human tone and ownership.

Task 3: Module-Level and Function-Level Documentation

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

requires structured documentation.

Prompt:



Code:

```python
    from typing import Union
    Number = Union[int, float]
    def add(a: Number, b: Number) -> Number:
        """Add two numbers.
        Parameters
        ----------
        a : int or float
            First operand.
        b : int or float
            Second operand.
        Returns
        -------
        int or float
            The sum of ``a`` and ``b``. If both inputs are integers the result
            will be an integer; otherwise a float is returned.
        Raises
        ------
        TypeError
            If either ``a`` or ``b`` is not a number.
        Examples
        --------
        >>> add(1, 2)
        3
        >>> add(1.5, 2.0)
        3.5
        """
        # Validate types
        if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
            raise TypeError("Both a and b must be int or float")
        return a + b
    def subtract(a: Number, b: Number) -> Number:
        """Subtract one number from another.
```

```python
    def multiply(a: Number, b: Number) -> Number:
        """
        if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
            raise TypeError("Both a and b must be int or float")
        return a * b
    def divide(a: Number, b: Number) -> float:
        """Divide one number by another.
        Parameters
        ----------
        a : int or float
            Numerator.
        b : int or float
            Denominator.
        Returns
        -------
        float
            The division result as a float.
        Raises
        ------
        TypeError
            If either ``a`` or ``b`` is not a number.
        ZeroDivisionError
            If ``b`` is zero.
        Examples
        --------
        >>> divide(10, 2)
        5.0
        >>> divide(5, 2)
        2.5
        """
        if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
            raise TypeError("Both a and b must be int or float")
        if b == 0:
            raise ZeroDivisionError("Denominator b must not be zero")
        return float(a) / float(b)
    if __name__ == "__main__":
        # Simple demonstration when run as a script
        print("Demo: basic arithmetic functions from math_ops")
        print("add(2, 3) ->", add(2, 3))
        print("subtract(5, 2) ->", subtract(5, 2))
        print("multiply(3, 4) ->", multiply(3, 4))
        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.