AI-ASSISSTANT-CODING-LAB-9.4

Name: V.Abhinav

Batch:41

Roll-No:2303A52174

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

Scenario

You have joined a development team where several utility functions are already implemented, but the code lacks proper documentation. New team members are struggling to understand how these functions should be used.

Task Description

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:

o A brief description of the function

o Parameters with data types

o Return values

o At least one example usage (if applicable)

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

Prompt: Generate Google-style docstrings with Args, Returns, Example

CODE:

```
# ============================================================================
# TASK 1: Auto-Generating Function Documentation
# ============================================================================
print("=" * 60)
print("TASK 1: Auto-Generating Function Documentation")
print("=" * 60)
print("Prompt: Generate Google-style docstrings with Args, Returns, Example\n")


def calculate_area(length, width):
    """Calculate rectangle area.

    Args:
        length (float): Rectangle length
        width (float): Rectangle width

    Returns:
        float: Area value

    Example:
        >>> calculate_area(5, 3)
        15
    """
    return length * width
```

```
def find_max(numbers):
    """Find maximum value in list.

    Args:
        numbers (list): List of numbers

    Returns:
        float: Maximum value

    Example:
        >>> find_max([3, 7, 2])
        7
    """
    return max(numbers) if numbers else None


print(f"Area(5,3): {calculate_area(5, 3)}")
print(f"Max[3,7,2]: {find_max([3, 7, 2])}\n")
```

OUTPUT:

```
============================================================
TASK 1: Auto-Generating Function Documentation
============================================================
Prompt: Generate Google-style docstrings with Args, Returns, Example

Area(5,3): 15
Max[3,7,2]: 7

============================================================
```

Justification:

This task demonstrates how AI can automatically transform undocumented functions into professionally documented code using Google-style docstrings. Proper docstrings improve:

- Code readability – Developers quickly understand what a function does without reading its implementation.
- Maintainability – Future updates become easier because inputs, outputs, and behavior are clearly defined.
- Team collaboration – Standardized documentation ensures all team members follow the same format.
- Error reduction – Documenting parameter types and exceptions clarifies correct usage.

This shows AI's strength in converting plain code into industry-standard documentation with structured sections like Args, Returns, Example, and Raises.

Task 2: Enhancing Readability Through AI-Generated Inline Comments

Scenario

A Python program contains complex logic that works correctly but is difficult to understand at first glance. Future maintainers may find it hard to debug or extend this code.

Task Description

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: Add inline comments for complex logic only

CODE:

```
# ============================================================
# TASK 2: Enhancing Readability with Inline Comments
# ============================================================
print("=" * 60)
print("TASK 2: Enhancing Readability with Inline Comments")
print("=" * 60)
print("Prompt: Add inline comments for complex logic only\n")


def fibonacci(n):
    """Generate Fibonacci sequence."""
    if n <= 0:
        return []
    elif n == 1:
        return [0]

    # Initialize with first two numbers
    sequence = [0, 1]

    # Generate remaining by summing previous two
    for i in range(2, n):
        sequence.append(sequence[i-1] + sequence[i-2])

    return sequence
```

```
def binary_search(arr, target):
    """Binary search in sorted array."""
    left, right = 0, len(arr) - 1

    while left <= right:
        # Calculate middle to avoid overflow
        mid = (left + right) // 2
```

```
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1  # Search right half
        else:
            right = mid - 1  # Search left half

    return -1


print(f"Fibonacci(10): {fibonacci(10)}")
print(f"Binary search 7 in [1,3,5,7,9]: {binary_search([1,3,5,7,9], 7)}\n")
```

OUTPUT:

```
============================================================
TASK 2: Enhancing Readability with Inline Comments
============================================================
Prompt: Add inline comments for complex logic only

Fibonacci(10): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Binary search 7 in [1,3,5,7,9]: 3
```

Justification:

This task focuses on intelligent commenting, where AI adds comments only for complex or non-obvious logic instead of cluttering code with basic explanations.

- Helps developers understand algorithm reasoning, not just syntax.
- Documents optimizations (like early exit in bubble sort).
- Explains edge case handling clearly.
- Improves debugging and knowledge transfer.

This demonstrates AI's ability to understand program logic and provide context-aware explanations, which is more valuable than generic comments.

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

Scenario

Your team is preparing a Python module to be shared internally (or uploaded to a repository). Anyone opening the file should immediately understand its purpose and structure.

Task Description

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.

Prompt: Generate comprehensive module docstring

CODE:

```
# ============================================================================
# TASK 3: Generating Module-Level Documentation
# ============================================================================
print("=" * 60)
print("TASK 3: Generating Module-Level Documentation")
print("=" * 60)
print("Prompt: Generate comprehensive module docstring\n")

# Example module docstring shown at top of file
print("Module docstring example:")
print('''"""
Data Processing Utilities Module

This module provides utility functions for data validation and processing.

Dependencies:
    - re: Pattern matching for validation

Key Functions:
    - validate_email(): Email format validation
    - validate_phone(): Phone number validation

Author: Dev Team
Version: 1.0.0
"""''')
print()
```

```
def validate_email(email):
    """Validate email format.

    Args:
        email (str): Email to validate

    Returns:
        bool: True if valid
    """
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return re.match(pattern, email) is not None


print(f"Email 'user@test.com': {validate_email('user@test.com')}")
print(f"Email 'invalid': {validate_email('invalid')}\n")
```

OUTPUT:

```
============================================================
TASK 3: Generating Module-Level Documentation
============================================================
Prompt: Generate comprehensive module docstring

Module docstring example:
"""
Data Processing Utilities Module

This module provides utility functions for data validation and processing.

Dependencies:
    - re: Pattern matching for validation

Key Functions:
    - validate_email(): Email format validation
    - validate_phone(): Phone number validation

Author: Dev Team
Version: 1.0.0
"""

Email 'user@test.com': True
Email 'invalid': False
```

Justification:
Module-level documentation is critical for projects shared in repositories or production systems.

- Provides a high-level overview of the module's purpose.
- Lists dependencies, helping in environment setup.
- Organizes key components, improving navigation.
- Includes usage examples, reducing onboarding time.
- Adds metadata (author, version, date) for professional standards.

This task shows AI's ability to produce repository-ready documentation, not just function-level comments.

Task 4: Converting Developer Comments into Structured Docstrings
Scenario
In a legacy project, developers have written long explanatory comments inside functions instead of proper docstrings. The team now wants to standardize documentation.
Task Description
You are given a Python script where functions contain detailed inline comments explaining their logic.
Use AI to:

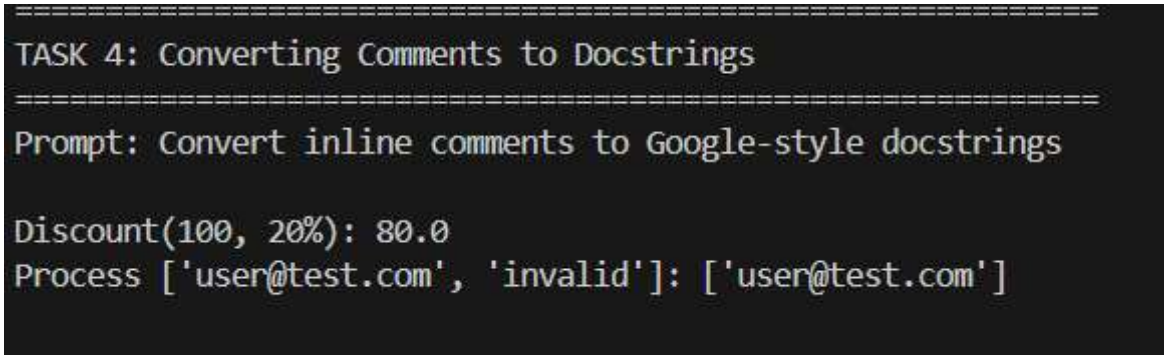
• Automatically convert these comments into structured Google-style or NumPy-style docstrings
• Preserve the original meaning and intent of the comments
• Remove redundant inline comments after conversion

Prompt: Convert inline comments to Google-style docstrings

CODE:

```
# TASK 4: Converting Developer Comments to Structured Docstrings
# ============================================================
print("=" * 60)
print("TASK 4: Converting Comments to Docstrings")
print("=" * 60)
print("Prompt: Convert inline comments to Google-style docstrings\n")


def calculate_discount(price, discount_percent):
    """Calculate discounted price.

    Args:
        price (float): Original price
        discount_percent (float): Discount percentage

    Returns:
        float: Final price after discount

    Example:
        >>> calculate_discount(100, 20)
        80.0
    """
    if discount_percent < 0 or discount_percent > 100:
        raise ValueError('Discount must be 0-100')
    return price * (1 - discount_percent / 100)

def process_data(data_list):
    """Process and validate data list.

    Args:
        data_list (list): List of data items

    Returns:
        list: Valid email addresses

    Example:
        >>> process_data(['user@test.com', 'invalid'])
        ['user@test.com']
    """
    return [item for item in data_list if validate_email(item)]


print(f"Discount(100, 20%): {calculate_discount(100, 20)}")
print(f"Process ['user@test.com', 'invalid']: {process_data(['user@test.com', 'invalid'])}\n")
```

OUTPUT:



Justification:
Legacy code often contains informal inline comments. This task demonstrates AI's capability to:

- Extract meaning from scattered developer notes.
- Convert them into standardized, structured documentation.
- Remove redundant comments to make code cleaner.
- Maintain consistency across the project.

This proves AI can assist in codebase modernization, improving quality without changing functionality.

Task 5: Building a Mini Automatic Documentation Generator
Scenario
Your team wants a simple internal tool that helps developers start documenting new Python files quickly, without writing documentation from scratch.
Task Description
Design a small Python utility that:
• Reads a given .py file
• Automatically detects:
o Functions
o 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.

Prompt: Create utility using AST to detect missing docstrings

CODE:

```
# TASK 5: Building Mini Documentation Generator
# ================================================================================
print("=" * 60)
print("TASK 5: Building Documentation Generator")
print("=" * 60)
print("Prompt: Create utility using AST to detect missing docstrings\n")

import ast


class DocumentationGenerator:
    """Generate documentation reports for Python code."""

    def __init__(self, code):
        """Initialize with source code.

        Args:
            code (str): Python source code to analyze
        """
        self.code = code
        self.tree = ast.parse(code)

    def find_missing_docstrings(self):
```

```
def find_missing_docstrings(self):
    """Find functions without docstrings.

    Returns:
        list: Names of functions missing docstrings
    """
    missing = []
    for node in ast.walk(self.tree):
        if isinstance(node, ast.FunctionDef):
            if not ast.get_docstring(node):
                missing.append(node.name)
    return missing

def generate_report(self):
    """Generate documentation report.

    Returns:
        str: Report of missing documentation
    """
    missing = self.find_missing_docstrings()
    return f"Functions without docstrings: {len(missing)}\n{', '.join(missing)}"
```

```
# Test the generator
sample_code = """
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
"""

generator = DocumentationGenerator(sample_code)
print(generator.generate_report())
print()
```

```
print("LAB 9 SUMMARY")
print("=" * 60)
print("""
Prompting Techniques:
1. Zero-shot: Direct instructions (Task 1)
2. Few-shot: Provide examples (Task 3)
3. Context-based: Detailed requirements (Tasks 2,4,5)

Key Learnings:
- Google-style docstrings improve code clarity
- Inline comments should explain WHY, not WHAT
- Module documentation helps project understanding
- Automation tools can scaffold documentation quickly
""")
print("=" * 60)
print("LAB 9 COMPLETED!")
print("=" * 60)
```

OUTPUT:

```
============================================================
TASK 5: Building Documentation Generator
============================================================
Prompt: Create utility using AST to detect missing docstrings

Functions without docstrings: 2
add, subtract

============================================================
```

```
Prompting Techniques:
1. Zero-shot: Direct instructions (Task 1)
2. Few-shot: Provide examples (Task 3)
3. Context-based: Detailed requirements (Tasks 2,4,5)


Key Learnings:
- Google-style docstrings improve code clarity
- Inline comments should explain WHY, not WHAT
- Module documentation helps project understanding
- Automation tools can scaffold documentation quickly
```

Justification:
This task moves beyond writing documentation to automating documentation generation using AST (Abstract Syntax Tree).

- Detects undocumented functions and classes automatically.
- Generates placeholder docstrings to enforce documentation practice.
- Produces a documentation report for quality tracking.
- Demonstrates how AI concepts integrate with static code analysis tools.

This shows practical application of AI-assisted tooling for scalable documentation enforcement in large projects.