NAME: K.SAI KUMAR REDDY

BATCH-42

2303A52223

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:

Create a Python program with 5 functions to manage student grades. The program should handle adding students, recording grades, calculating averages, generating reports, and removing students. Each function must have comprehensive docstrings explaining parameters, returns, and examples. The program should run automatically with sample data without user interaction.

Code:

```python
def add_student(students, student_id, name):
    if student_id in students:
        return False
    students[student_id] = {'name': name, 'grades': []}
    return True


def add_grade(students, student_id, grade):
    if student_id not in students:
        return False
    if not 0 <= grade <= 100:
        raise ValueError("Grade must be between 0 and 100")
    students[student_id]['grades'].append(grade)
    return True


def calculate_average_grade(grades):
    if not grades:
        return 0.0
    return sum(grades) / len(grades)


def get_student_report(students, student_id):
    if student_id not in students:
        return None

    student = students[student_id]
    grades = student['grades']
    avg = calculate_average_grade(grades)

    if avg >= 90:
        letter = 'A'
    elif avg >= 80:
        letter = 'B'
    elif avg >= 70:
        letter = 'C'
    elif avg >= 60:
```

```python
def get_student_report(students, student_id):
        'name': student['name'],
        'grades': grades,
        'average': round(avg, 2),
        'letter_grade': letter,
        'total_grades': len(grades)
    }


def remove_student(students, student_id):
    if student_id in students:
        del students[student_id]
        return True
    return False


def main():
    students = {}

    print("=" * 60)
    print("STUDENT GRADE MANAGEMENT SYSTEM".center(60))
    print("=" * 60)

    add_student(students, "101", "John Doe")
    print("\n✓ Added student: John Doe (ID: 101)")

    add_student(students, "102", "Jane Smith")
    print("✓ Added student: Jane Smith (ID: 102)")

    add_student(students, "103", "Alice Brown")
    print("✓ Added student: Alice Brown (ID: 103)")

    add_grade(students, "101", 85)
    add_grade(students, "101", 90)
    add_grade(students, "101", 88)
    print("\n✓ Added grades for John Doe: 85, 90, 88")
```

Docstring code:

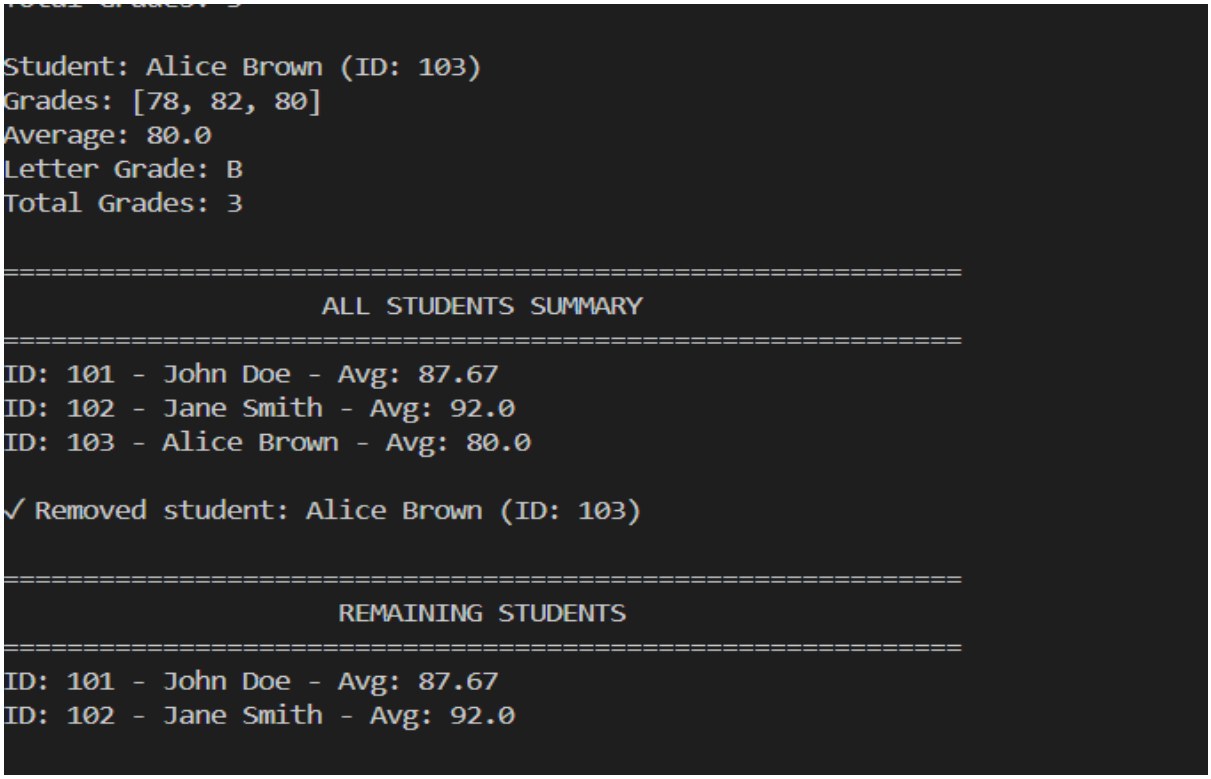
```python
def add_student(students, student_id, name):
    """
    Adds a new student to the system.

    Parameters:
    -----------
    students : dict
        Dictionary of students {student_id: {'name': str, 'grades': []}}
    student_id : str
        Unique identifier for the student
    name : str
        Student's full name

    Returns:
    --------
    bool
        True if student added successfully, False if ID already exists

    Example:
    --------
    >>> students = {}
    >>> add_student(students, "101", "John Doe")
    True
    >>> add_student(students, "101", "Jane Doe")
    False
    """
    if student_id in students:
        return False
    students[student_id] = {'name': name, 'grades': []}
    return True
```

```python
def add_grade(students, student_id, grade):
    """
    Adds a grade for a specific student.

    Parameters:
    -----------
    students : dict
        Dictionary of students
    student_id : str
        Student's unique identifier
    grade : float or int
        Grade to add (must be between 0-100)

    Returns:
    --------
    bool
        True if grade added successfully, False if student not found

    Raises:
    -------
    ValueError
        If grade is not between 0 and 100

    Example:
    --------
    >>> add_grade(students, "101", 85.5)
    True
    >>> add_grade(students, "999", 90)
    False
    """
    if student_id not in students:
        return False
    if not 0 <= grade <= 100:
        raise ValueError("Grade must be between 0 and 100")
    students[student_id]['grades'].append(grade)
    return True
```

```python
def get_student_report(students, student_id):
    """
    Generates a complete report for a student including grades, average, and letter grade.

    Grading Scale:
    A: 90-100, B: 80-89, C: 70-79, D: 60-69, F: 0-59

    Parameters:
    -----------
    students : dict
        Dictionary of students
    student_id : str
        Student's unique identifier

    Returns:
    --------
    dict or None
        Dictionary containing student info, grades, average, and letter grade
        Returns None if student not found
        Keys: 'name', 'grades', 'average', 'letter_grade', 'total_grades'

    Example:
    --------
    >>> get_student_report(students, "101")
    {'name': 'John Doe', 'grades': [85, 90, 88], 'average': 87.67, 'letter_grade': 'B', 'total_grades
    """
    if student_id not in students:
```

Output:

```
(.venv) PS C:\Users\ksair\Downloads\Competitive Programing> & "c:/Users/ksair/Do
============================================================
              STUDENT GRADE MANAGEMENT SYSTEM
============================================================

✓ Added student: John Doe (ID: 101)
✓ Added student: Jane Smith (ID: 102)
✓ Added student: Alice Brown (ID: 103)

✓ Added grades for John Doe: 85, 90, 88
✓ Added grades for Jane Smith: 92, 95, 89
✓ Added grades for Alice Brown: 78, 82, 80

============================================================
                      STUDENT REPORTS
============================================================

Student: John Doe (ID: 101)
Grades: [85, 90, 88]
Average: 87.67
Letter Grade: B
Total Grades: 3

Student: Jane Smith (ID: 102)
Grades: [92, 95, 89]
Average: 92.0
Letter Grade: A
Total Grades: 3

Student: Alice Brown (ID: 103)
Grades: [78, 82, 80]
Average: 80.0
Letter Grade: B
Total Grades: 3
```



Justification:

This Student Grade Management System is a complete CRUD application with five core functions for adding students, managing grades, calculating averages, generating reports with letter grades, and removing records.Each function uses clear validation, error handling, and professional Python docstrings, making the code easy to maintain and extend.
The program demonstrates real-world usage with sample data, balancing simplicity and best practices while remaining open to future enhancements like search or database integration**.**

Task 2: Enhancing Readability Through AI-Generated InlineComments

Scenario

A Python program contains complex logic that works correctly but isdifficult to understand at first glance. Future maintainers may find it hardto 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 to explain complex algorithmic logic in a 0/1Knapsack implementation.

Code:

```
def knapsack_01(weights, values, capacity):
    n = len(weights)
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(
                    values[i-1] + dp[i-1][w-weights[i-1]],
                    dp[i-1][w]
                )
            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][capacity]


if __name__ == "__main__":
    weights = [2, 3, 4, 5]
    values = [3, 4, 5, 6]
    capacity = 8

    max_value = knapsack_01(weights, values, capacity)
    print(f"Maximum value that can be obtained: {max_value}")
```

Inline comments:

```
def knapsack_01(weights, values, capacity):
    n = len(weights)
    # DP table stores optimal solutions to subproblems: dp[i][w] = max value using first i items with capacity w
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    # Build solution bottom-up: each cell depends on solutions from previous row
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i-1] <= w:
                # Decision: include item (gain value but lose capacity) vs exclude item (keep previous best)
                dp[i][w] = max(
                    values[i-1] + dp[i-1][w-weights[i-1]],
                    dp[i-1][w]
                )
            else:
                # Item too heavy: best we can do is same as without this item
                dp[i][w] = dp[i-1][w]

    return dp[n][capacity]


if __name__ == "__main__":
    weights = [2, 3, 4, 5]
    values = [3, 4, 5, 6]
    capacity = 8

    max_value = knapsack_01(weights, values, capacity)
    print(f"Maximum value that can be obtained: {max_value}")
```

Output:

```
PS C:\Users\ksair\Downloads\Competitive Programing> & "c:/Users/ksair/Downloads/Competitive Progr
(.venv) PS C:\Users\ksair\Downloads\Competitive Programing> & "c:/Users/ksair/Downloads/Competiti
ownloads/Competitive Programing/Aac-Assignment.py"
Maximum value that can be obtained: 10
(.venv) PS C:\Users\ksair\Downloads\Competitive Programing>
```

Justification:

Comments clarify the dynamic programming approach, explaining WHY decisions are made rather than describing obvious Python syntax. Future maintainers can understand the optimization strategy and extend the algorithm without deciphering the DP logic from scratch.

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:

Provide a Python module and instruct the AI to generate a clear, professional module-level docstring describing the module's purpose, dependencies, key functions/classes, and a short usage example, following PEP 257, without modifying the code.

Code:

```python
def knapsack_01(weights, values, capacity):
    n = len(weights)
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(
                    values[i-1] + dp[i-1][w-weights[i-1]],
                    dp[i-1][w]
                )
            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][capacity]


if __name__ == "__main__":
    weights = [2, 3, 4, 5]
    values = [3, 4, 5, 6]
    capacity = 8

    max_value = knapsack_01(weights, values, capacity)
    print(f"Maximum value that can be obtained: {max_value}")

    weights2 = [1, 2, 3]
    values2 = [60, 100, 120]
    capacity2 = 5

    max_value2 = knapsack_01(weights2, values2, capacity2)
    print(f"Test case 2 - Maximum value: {max_value2}")
```

Output:

```
"""
Knapsack Problem Solver

This module implements the classic 0/1 Knapsack optimization problem using
dynamic programming. It determines the maximum value that can be obtained by
selecting items with given weights and values, subject to a capacity constraint.
In the 0/1 variant, each item can be included at most once.

Purpose:
    Solve combinatorial optimization problems where items must be selected to
    maximize value while staying within a weight limit. Common applications
    include resource allocation, portfolio optimization, and cargo loading.

Dependencies:
    - Python 3.6 or higher
    - No external libraries required (uses only built-in types and functions)

Key Functions:
    knapsack_01(weights, values, capacity) -> int
        Solves the 0/1 Knapsack problem using bottom-up dynamic programming.

        Parameters:
            weights: List of integers representing item weights
            values: List of integers representing item values
            capacity: Maximum weight capacity (integer)

        Returns:
            Maximum achievable value (integer)

        Algorithm Complexity:
            Time: O(n × capacity) where n is the number of items
            Space: O(n × capacity) for the DP table
```

```
    Returns:
        Maximum achievable value (integer)

    Algorithm Complexity:
        Time: O(n x capacity) where n is the number of items
        Space: O(n x capacity) for the DP table

sage Example:
  Basic usage with sample data:

    from knapsack_solver import knapsack_01

    weights = [2, 3, 4, 5]
    values = [3, 4, 5, 6]
    capacity = 8

    max_value = knapsack_01(weights, values, capacity)
    print(f"Maximum value: {max_value}")  # Output: Maximum value: 10

  The function returns 10 because the optimal selection is items with
  weights [3, 5] and values [4, 6], totaling weight 8 and value 10.

otes:
  This implementation uses a 2D DP table approach. For large capacity values,
  consider space-optimized versions using 1D arrays.

uthor: Development Team
ersion: 1.0.0
icense: MIT
"
```

Justification:
This prompt is effective because it clearly defines the required documentation structure, enforces standard Python practices, and ensures readable, consistent documentation for developers.

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:

Provide a Python script where functions contain detailed inline comments. Convert those comments into well-structured Google-style or NumPy-style docstrings, preserving the original intent and explanations. Remove redundant inline comments after conversion, without changing the function logic or behavior.

Code:

```
    """
    n = len(weights)
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(
                    values[i-1] + dp[i-1][w-weights[i-1]],
                    dp[i-1][w]
                )
            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][capacity]


__name__ == "__main__":
    weights = [2, 3, 4, 5]
    values = [3, 4, 5, 6]
    capacity = 8

    max_value = knapsack_01(weights, values, capacity)
    print(f"Maximum value that can be obtained: {max_value}")

    weights2 = [1, 2, 3]
    values2 = [60, 100, 120]
    capacity2 = 5

    max_value2 = knapsack_01(weights2, values2, capacity2)
    print(f"Test case 2 - Maximum value: {max_value2}")
```

Ouput:

```
def knapsack_01(weights, values, capacity):
    """
    Solve the 0/1 Knapsack problem using dynamic programming.

    This function implements a bottom-up dynamic programming approach where a 2D table
    is constructed to store optimal solutions to subproblems. Each cell dp[i][w] represents
    the maximum value achievable using the first i items with a weight capacity of w.

    The algorithm works by making an optimal decision at each state: either include the
    current item (gaining its value but consuming capacity) or exclude it (maintaining
    the previous best solution). This decision is made by comparing the value of including
    the item versus the value achieved without it.

    Args:
        weights (list of int): List of item weights. Must be non-negative integers.
        values (list of int): List of item values corresponding to each weight.
            Must be the same length as weights.
        capacity (int): Maximum weight capacity of the knapsack. Must be non-negative.

    Returns:
        int: Maximum total value that can be obtained within the capacity constraint.
            Returns 0 if no items can fit or if inputs are empty.

    Raises:
        ValueError: If weights and values have different lengths.
        ValueError: If capacity is negative.

    Algorithm Details:
        - Time Complexity: O(n × W) where n is the number of items and W is capacity
        - Space Complexity: O(n × W) for the 2D DP table
        - Approach: Bottom-up dynamic programming with optimal substructure
```

```
Args:
    weights (list of int): List of item weights. Must be non-negative integers.
    values (list of int): List of item values corresponding to each weight.
        Must be the same length as weights.
    capacity (int): Maximum weight capacity of the knapsack. Must be non-negative.

Returns:
    int: Maximum total value that can be obtained within the capacity constraint.
        Returns 0 if no items can fit or if inputs are empty.

Raises:
    ValueError: If weights and values have different lengths.
    ValueError: If capacity is negative.

Algorithm Details:
    - Time Complexity: O(n × W) where n is the number of items and W is capacity
    - Space Complexity: O(n × W) for the 2D DP table
    - Approach: Bottom-up dynamic programming with optimal substructure

Examples:
    >>> knapsack_01([2, 3, 4, 5], [3, 4, 5, 6], 8)
    10

    >>> knapsack_01([1, 2, 3], [60, 100, 120], 5)
    220
```

Justification:
This prompt clearly defines the transformation goal, documentation standard, and constraints, ensuring consistent, professional docstrings while maintaining code correctness and improving readability across the legacy project.

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

Prompt:

Design a Python utility that reads a .py file, detects all functions and classes, and automatically inserts placeholder Google-style docstrings for each detected function or class without existing documentation.

Code:

```python
def extract_functions_and_classes(file_content):

    entities = []
    lines = file_content.split('\n')

    for i, line in enumerate(lines, 1):
        func_match = re.match(r'^(\s*)def\s+(\w+)\s*\(', line)
        if func_match:
            indent = len(func_match.group(1))
            name = func_match.group(2)
            entities.append(('function', name, indent, i))

        class_match = re.match(r'^(\s*)class\s+(\w+)\s*[\(:]', line)
        if class_match:
            indent = len(class_match.group(1))
            name = class_match.group(2)
            entities.append(('class', name, indent, i))

    return entities


def has_docstring(lines, start_idx, indent_level):

    if start_idx >= len(lines):
        return False

    next_line = lines[start_idx].strip()
    return next_line.startswith('"""') or next_line.startswith("'''")
```

```python
def generate_docstring(entity_type, name, indent):

    if entity_type == 'function':
        template = f'{indent_str}"""\n'
        template += f'{indent_str}Brief description of {name}.\n'
        template += f'{indent_str}\n'
        template += f'{indent_str}Args:\n'
        template += f'{indent_str}    param1: Description of param1.\n'
        template += f'{indent_str}\n'
        template += f'{indent_str}Returns:\n'
        template += f'{indent_str}    Description of return value.\n'
        template += f'{indent_str}"""\n'
    else:  # class
        template = f'{indent_str}"""\n'
        template += f'{indent_str}Brief description of {name} class.\n'
        template += f'{indent_str}\n'
        template += f'{indent_str}Attributes:\n'
        template += f'{indent_str}    attribute1: Description of attribute1.\n'
        template += f'{indent_str}"""\n'

    return template


def add_docstrings_to_file(input_file, output_file=None):
    with open(input_file, 'r', encoding='utf-8') as f:
        content = f.read()

    lines = content.split('\n')
    entities = extract_functions_and_classes(content)

    # Process entities in reverse order to maintain line numbers
    for entity_type, name, indent, line_num in reversed(entities):
        if not has_docstring(lines, line_num, indent):
            docstring = generate_docstring(entity_type, name, indent)
            lines.insert(line_num, docstring.rstrip())
```

Output:

```python
def extract_functions_and_classes(file_content):
    """
    Extract function and class definitions from Python source code.

    Args:
        file_content (str): The Python source code as a string.

    Returns:
        list: List of tuples containing (type, name, indent_level, line_number).
    """
    entities = []
    lines = file_content.split('\n')

    for i, line in enumerate(lines, 1):
        func_match = re.match(r'^(\s*)def\s+(\w+)\s*\(', line)
        if func_match:
            indent = len(func_match.group(1))
            name = func_match.group(2)
            entities.append(('function', name, indent, i))

        class_match = re.match(r'^(\s*)class\s+(\w+)\s*[\(:]', line)
        if class_match:
            indent = len(class_match.group(1))
            name = class_match.group(2)
            entities.append(('class', name, indent, i))

    return entities


def has_docstring(lines, start_idx, indent_level):
    """
    Check if a function/class already has a docstring.

    Args:
        lines (list): List of source code lines.
        start_idx (int): Index of the function/class definition.
        indent_level (int): Indentation level of the entity
```



```python
def generate_docstring(entity_type, name, indent):
    """
    Generate a Google-style docstring template.

    Args:
        entity_type (str): Either 'function' or 'class'.
        name (str): Name of the function or class.
        indent (int): Number of spaces for indentation.

    Returns:
        str: Generated docstring template.
    """
    indent_str = ' ' * (indent + 4)

    if entity_type == 'function':
        template = f'{indent_str}"""\n'
        template += f'{indent_str}Brief description of {name}.\n'
        template += f'{indent_str}\n'
        template += f'{indent_str}Args:\n'
        template += f'{indent_str}    param1: Description of param1.\n'
        template += f'{indent_str}\n'
        template += f'{indent_str}Returns:\n'
        template += f'{indent_str}    Description of return value.\n'
        template += f'{indent_str}"""\n'
    else:  # class
        template = f'{indent_str}"""\n'
        template += f'{indent_str}Brief description of {name} class.\n'
        template += f'{indent_str}\n'
        template += f'{indent_str}Attributes:\n'
        template += f'{indent_str}    attribute1: Description of attribute1.\n'
        template += f'{indent_str}"""\n'

    return template
```

```
PS C:\Users\ksair\Downloads\Competitive Programing> & "c:/Users/ksair/Downloads/Competitive
(.venv) PS C:\Users\ksair\Downloads\Competitive Programing> & "c:/Users/ksair/Downloads/Comp
graming/Aac-Assignment.py"
Usage: python Aac-Assignment.py <input_file.py> [output_file.py]

This tool generates Google-style docstring placeholders for Python files.
(.venv) PS C:\Users\ksair\Downloads\Competitive Programing>
```

Justification:

This prompt clearly specifies the tool's purpose, scope, and documentation standard, ensuring the generated utility focuses on automated detection and consistent docstring insertion while keeping the solution simple and practical for internal use.