AI ASSISTED CODING
ASSIGNMENT-9.2

AKULA VIVEK
2403A51258
BATCH-11

Task Description #1 (Documentation – Google-Style Docstrings for Python Functions)

• Task: Use AI to add Google-style docstrings to all functions in a given Python script.
• Instructions:
o Prompt AI to generate docstrings without providing any input-output examples.
o Ensure each docstring includes:
▪ Function description
▪ Parameters with type hints
▪ Return values with type hints
▪ Example usage
o Review the generated docstrings for accuracy and formatting.
• Expected Output #1:
o A Python script with all functions documented using correctly formatted Google-style docstrings.

PROMPT:

Task: Add Google-style docstrings to all functions in this file.

Requirements:

- Include function description

- Parameters with type hints

- Return values with type hints

- Example usage

- Do not include input/output examples other than Example section.

Code and Output:

```python
def add(a: int, b: int) -> int:
    """
    Adds two integers.

    Args:
        a (int): The first integer.
        b (int): The second integer.

    Returns:
        int: The sum of a and b.

    Example:
        >>> add(2, 3)
        5
    """
    return a + b

def multiply(x: float, y: float) -> float:
    """
    Multiplies two floating-point numbers.

    Args:
        x (float): The first number.
        y (float): The second number.

    Returns:
        float: The product of x and y.

    Example:
        >>> multiply(2.5, 4)
        10.0
    """
    return x * y
```

```python
t1.py > greet
def multiply(x: float, y: float) -> float:
    Returns:
        float: The product of x and y.

    Example:
        >>> multiply(2.5, 4)
        10.0
    """
    return x * y

def greet(name: str) -> str:
    """
    Returns a greeting message for the given name.

    Args:
        name (str): The name to greet.

    Returns:
        str: A greeting message.

    Example:
        >>> greet("Alice")
        'Hello, Alice!'
    """
    return f"Hello, {name}!"
```

Task Description #2 (Documentation – Inline Comments for Complex Logic)

• Task: Use AI to add meaningful inline comments to a Python program explaining only complex logic parts.

• Instructions:

o Provide a Python script without comments to the AI.

o Instruct AI to skip obvious syntax explanations and focus only on tricky or non-intuitive code sections.

o Verify that comments improve code readability and maintainability.

• Expected Output #2:
o Python code with concise, context-aware inline comments for complex logic blocks.

Prompt:

Task: Add meaningful inline comments to this Python program.

Instructions for AI (Copilot):

- Focus on complex or non-intuitive logic blocks only.

- Skip obvious syntax explanations (like variable assignments or basic loops).

- Comments should improve readability and maintainability of the code.

Code and Output:

t2.py > ...

```python
def fibonacci(n):
    # Iteratively compute the nth Fibonacci number using tuple unpacking.
    a, b = 0, 1
    for _ in range(n):
        # Update 'a' to the next Fibonacci number, 'b' to the one after that.
        a, b = b, a + b
    return a

```

Task Description #3 (Documentation – Module-Level Documentation)

• Task: Use AI to create a module-level docstring summarizing the purpose, dependencies, and main functions/classes of a Python file.

• Instructions:
o Supply the entire Python file to AI.
o Instruct AI to write a single multi-line docstring at the top of the file.
o Ensure the docstring clearly describes functionality and usage without rewriting the entire code.
• Expected Output #3:
o A complete, clear, and concise module-level docstring at the beginning of the file

Prompt:

Task: Create a module-level docstring summarizing the purpose, dependencies, and main functions/classes of this Python file.

Instructions for AI (Copilot):

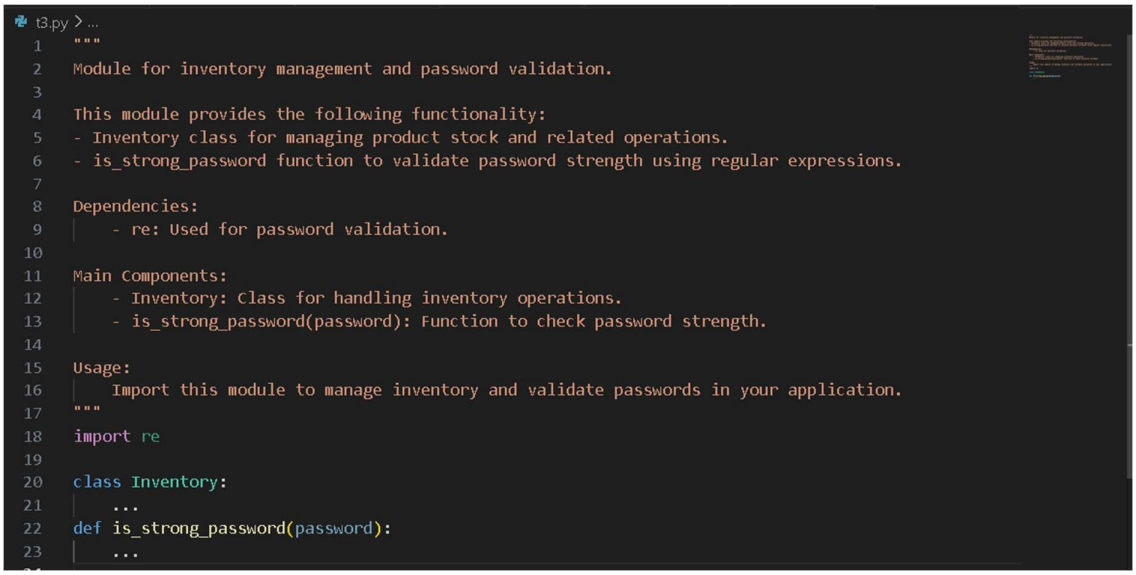
- Write a single multi-line docstring at the very top of the file.

- Clearly describe the module’s functionality and usage.

- Include dependencies (imports), main functions, and classes.

- Do not rewrite or duplicate the entire code.

Code and output:



```
t3.py > ...
"""
Module for inventory management and password validation.

This module provides the following functionality:
- Inventory class for managing product stock and related operations.
- is_strong_password function to validate password strength using regular expressions.

Dependencies:
    - re: Used for password validation.

Main Components:
    - Inventory: Class for handling inventory operations.
    - is_strong_password(password): Function to check password strength.

Usage:
    Import this module to manage inventory and validate passwords in your application.
"""
import re

class Inventory:
    ...
def is_strong_password(password):
    ...
```

Task Description #4 (Documentation – Convert Comments to Structured Docstrings)

• Task: Use AI to transform existing inline comments into structured function docstrings following Google style.
• Instructions:
o Provide AI with Python code containing inline comments.
o Ask AI to move relevant details from comments into function docstrings.
o Verify that the new docstrings keep the meaning intact while improving structure.
• Expected Output #4:
o Python code with comments replaced by clear, standardized docstrings.

Prompt:

Task: Transform existing inline comments into structured Google-style docstrings.

Instructions for AI (Copilot):

- For each function, move meaningful inline comments into a docstring directly under the function definition.

- Use Google-style docstring format.

- Include function description, parameters (with type hints), returns (with type hints), and an Example section.

- Remove or minimize redundant inline comments once docstring is added.

- Keep the function logic unchanged.

Code and output:

t4.py > ...

```python
def add_item(name: str, quantity: int):
    """
    Adds an item to the inventory dictionary.

    Args:
        name (str): The name of the item.
        quantity (int): How many items to add.

    Returns:
        None

    Example:
        >>> add_item("apple", 5)
    """
    inventory[name] = inventory.get(name, 0) + quantity

```

Task Description #5 (Documentation – Review and Correct Docstrings)

• Task: Use AI to identify and correct inaccuracies in existing docstrings.
• Instructions:
o Provide Python code with outdated or incorrect docstrings.
o Instruct AI to rewrite each docstring to match the current code behavior.
o Ensure corrections follow Google-style formatting.
• Expected Output #5:
o Python file with updated, accurate, and standardized docstrings.

Prompt:

I have Python code with outdated or incorrect docstrings. Please review the code, correct the docstrings to accurately describe the functions/classes, and rewrite them in Google-style formatting.

Code with outdated docstrings :

t5.py > ...

```python
class Calculator:
    def add(self, a, b):
        """Adds two numbers together and returns a string."""
        return a + b

    def subtract(self, a, b):
        """Multiply two numbers."""
        return a - b

def greet(name: str) -> str:
    """Greets the user by printing hello."""
    return f"Hello, {name}!"
```

Updated code:

t5.py > ...

```python
class Calculator:
    """
    A simple calculator class for basic arithmetic operations.
    """

    def add(self, a, b):
        """
        Adds two numbers.

        Args:
            a (int or float): The first number.
            b (int or float): The second number.

        Returns:
            int or float: The sum of a and b.

        Example:
            >>> Calculator().add(2, 3)
            5
        """
        return a + b

    def subtract(self, a, b):
        """
        Subtracts the second number from the first.

        Args:
            a (int or float): The number to subtract from.
            b (int or float): The number to subtract.

        Returns:
            int or float: The result of a minus b.

        Example:
            >>> Calculator().subtract(5, 2)
            3
        """
        return a - b

def greet(name: str) -> str:
    """
    Returns a greeting message for the given name.

    Args:
        name (str): The name to greet.

    Returns:
        str: A greeting message.

    Example:
        >>> greet("Alice")
        'Hello, Alice!'
    """
    return f"Hello, {name}!"
```

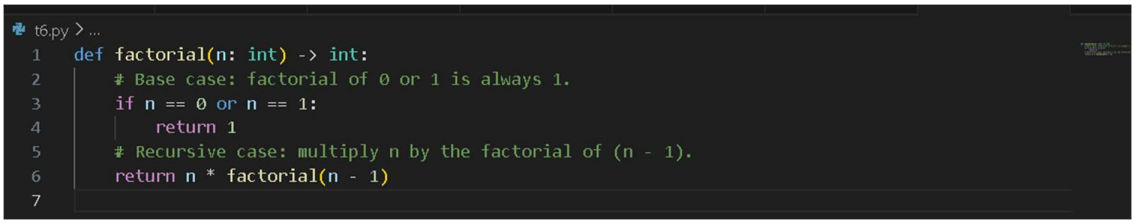
Task Description #6 (Documentation – Prompt Comparison Experiment)

• Task: Compare documentation output from a vague prompt and a detailed prompt for the same Python function.
• Instructions:
o Create two prompts: one simple ("Add comments to this function") and one detailed ("Add Google-style docstrings with parameters, return types, and examples").
o Use AI to process the same Python function with both prompts.
o Analyze and record differences in quality, accuracy, and completeness.
• Expected Output #6:
o A comparison table showing the results from both prompts with observations.

Vague Prompt:

Add comments to this function.

Code for vague prompt:



Detailed Prompt:

Add Google-style docstrings with parameters, return types, and examples.

Code for detailed prompt: