

# AI ASSISTED CODING

## ASSIGNMENT-9.2

M.KEERTHANA

2403A51259

CSE-GEN

BATCH-11

### Task 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:

```
1 def add(a: int, b: int) -> int:
2     """
3     Adds two integers.
4
5     Args:
6         a (int): The first integer.
7         b (int): The second integer.
8
9     Returns:
10        int: The sum of a and b.
11
12     Example:
13         >>> add(2, 3)
14         5
15     """
16     return a + b
17
18 def multiply(x: float, y: float) -> float:
19     """
20     Multiplies two floating-point numbers.
21
22     Args:
23         x (float): The first number.
24         y (float): The second number.
25
26     Returns:
27         float: The product of x and y.
28
29     Example:
30         >>> multiply(2.5, 4)
31         10.0
32     """
33     return x * y
```

```
tipy > greet
18 def multiply(x: float, y: float) -> float:
26     Returns:
27         float: The product of x and y.
28
29     Example:
30         >>> multiply(2.5, 4)
31         10.0
32     """
33     return x * y
34
35 def greet(name: str) -> str:
36     """
37     Returns a greeting message for the given name.
38
39     Args:
40         name (str): The name to greet.
41
42     Returns:
43         str: A greeting message.
44
45     Example:
46         >>> greet("Alice")
47         'Hello, Alice!'
48     """
49     return f"Hello, {name}!"
```

## Task 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.

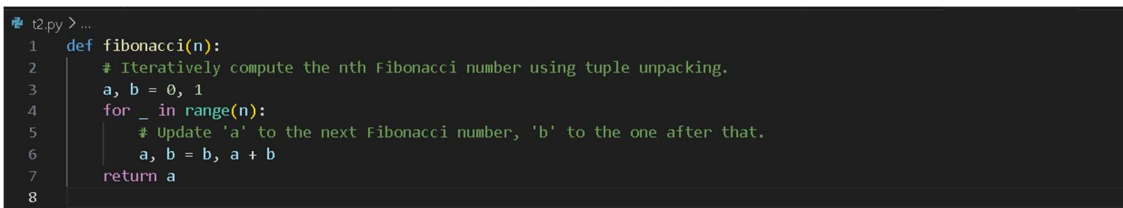
## PROMPT:

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:



```

12.py > ...
1  def fibonacci(n):
2      # Iteratively compute the nth Fibonacci number using tuple unpacking.
3      a, b = 0, 1
4      for _ in range(n):
5          # Update 'a' to the next Fibonacci number, 'b' to the one after that.
6          a, b = b, a + b
7      return a
8

```

## Task 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:

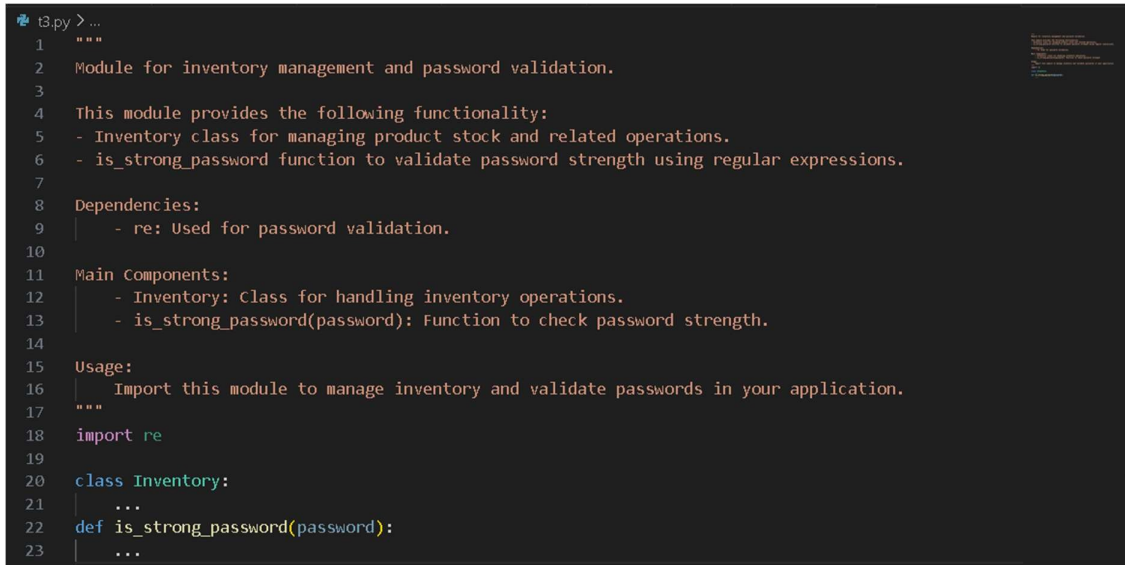
Task: Create a module-level docstring summarizing the purpose, dependencies, and main functions/classes of this Python file.

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:



```

1  """
2  Module for inventory management and password validation.
3
4  This module provides the following functionality:
5  - Inventory class for managing product stock and related operations.
6  - is_strong_password function to validate password strength using regular expressions.
7
8  Dependencies:
9  | - re: Used for password validation.
10
11 Main Components:
12 | - Inventory: Class for handling inventory operations.
13 | - is_strong_password(password): Function to check password strength.
14
15 Usage:
16 | Import this module to manage inventory and validate passwords in your application.
17 """
18 import re
19
20 class Inventory:
21     ...
22 def is_strong_password(password):
23     ...

```

## Task 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.

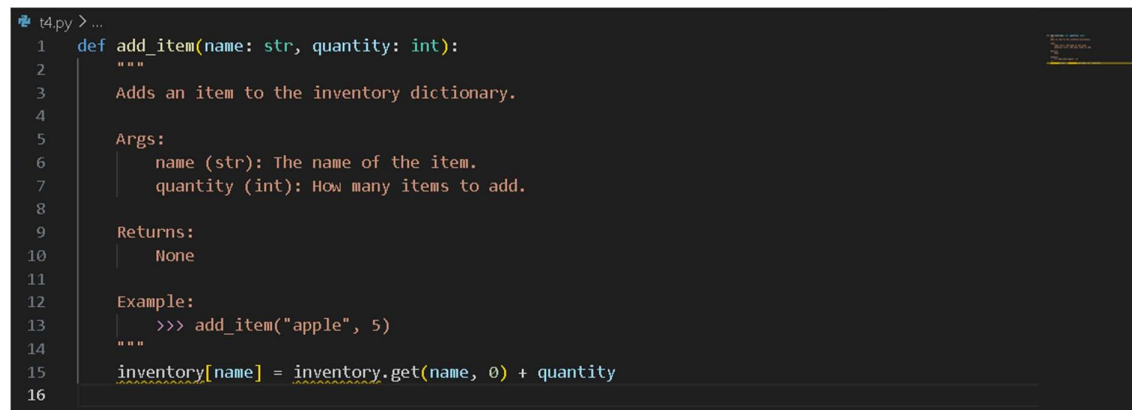
## Prompt:

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:



```

1  def add_item(name: str, quantity: int):
2      """
3      Adds an item to the inventory dictionary.
4
5      Args:
6          name (str): The name of the item.
7          quantity (int): How many items to add.
8
9      Returns:
10         None
11
12     Example:
13         >>> add_item("apple", 5)
14         """
15     inventory[name] = inventory.get(name, 0) + quantity
16

```

## Task 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 > ...
1 class Calculator:
2     def add(self, a, b):
3         """Adds two numbers together and returns a string."""
4         return a + b
5
6     def subtract(self, a, b):
7         """Multiply two numbers."""
8         return a - b
9
10    def greet(name: str) -> str:
11        """Greets the user by printing hello."""
12        return f"Hello, {name}!"
```

## Updated code:

```
t5.py > ...
1 class Calculator:
2     """
3     A simple calculator class for basic arithmetic operations.
4     """
5
6     def add(self, a, b):
7         """
8         Adds two numbers.
9
10        Args:
11            a (int or float): The first number.
12            b (int or float): The second number.
13
14        Returns:
15            int or float: The sum of a and b.
16
17        Example:
18            >>> Calculator().add(2, 3)
19            5
20        """
21        return a + b
22
23    def subtract(self, a, b):
24        """
25        Subtracts the second number from the first.
26
27        Args:
28            a (int or float): The number to subtract from.
29            b (int or float): The number to subtract.
30
31        Returns:
32            int or float: The result of a minus b.
```

```
33
34        Example:
35            >>> Calculator().subtract(5, 2)
36            3
37        """
38        return a - b
39
40    def greet(name: str) -> str:
41        """
42        Returns a greeting message for the given name.
43
44        Args:
45            name (str): The name to greet.
46
47        Returns:
48            str: A greeting message.
49
50        Example:
51            >>> greet("Alice")
52            'Hello, Alice!'
53        """
54        return f"Hello, {name}!"
55
```

## Task 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:

```
t6.py > ...
1 def factorial(n: int) -> int:
2     # Base case: factorial of 0 or 1 is always 1.
3     if n == 0 or n == 1:
4         return 1
5     # Recursive case: multiply n by the factorial of (n - 1).
6     return n * factorial(n - 1)
7
```

### Detailed Prompt:

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

### Code for detailed prompt:

```
t6.py > ...
1 def factorial(n: int) -> int:
2     """
3     Calculates the factorial of a non-negative integer using recursion.
4
5     Args:
6         n (int): The non-negative integer to compute the factorial of.
7
8     Returns:
9         int: The factorial of n.
10
11     Example:
12         >>> factorial(5)
13         120
14     """
15     # Base case: factorial of 0 or 1 is always 1.
16     if n == 0 or n == 1:
17         return 1
18     # Recursive case: multiply n by the factorial of (n - 1).
19     return n * factorial(n - 1)
20
```