SR-UNIVERSITY

ASSIGNMENT :9.4
BTNO:10

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.

Expected Outcome

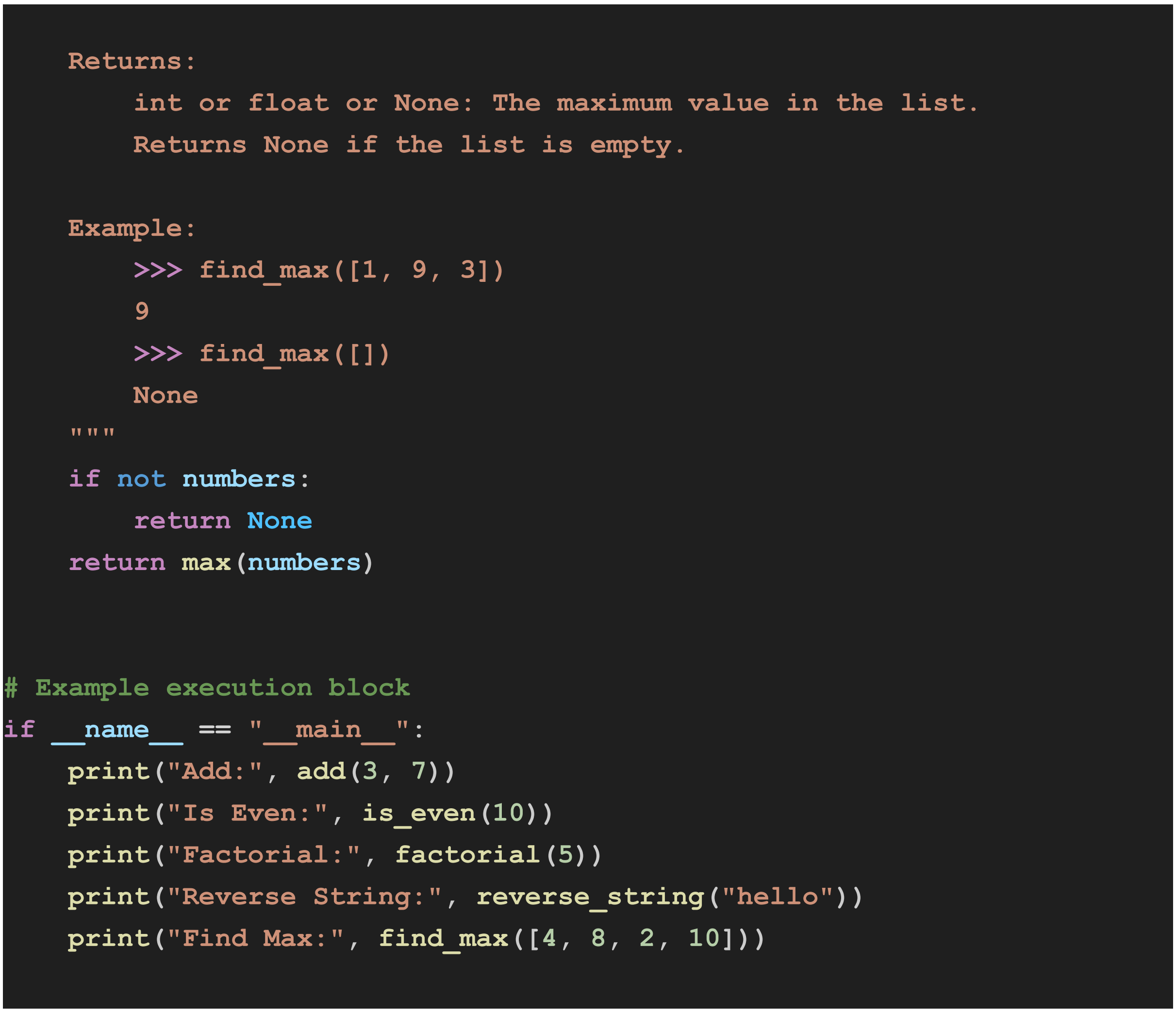
• A Python script with well-structured Google-style docstrings
• Docstrings that clearly explain function behavior and usage
. Improved readability and usability of the codebase

Prompt:

```
 # Generate a Google-style docstring for this function.
# Include description, Args with types, Returns with type,
# and one example usage. Do not change the logic.
```

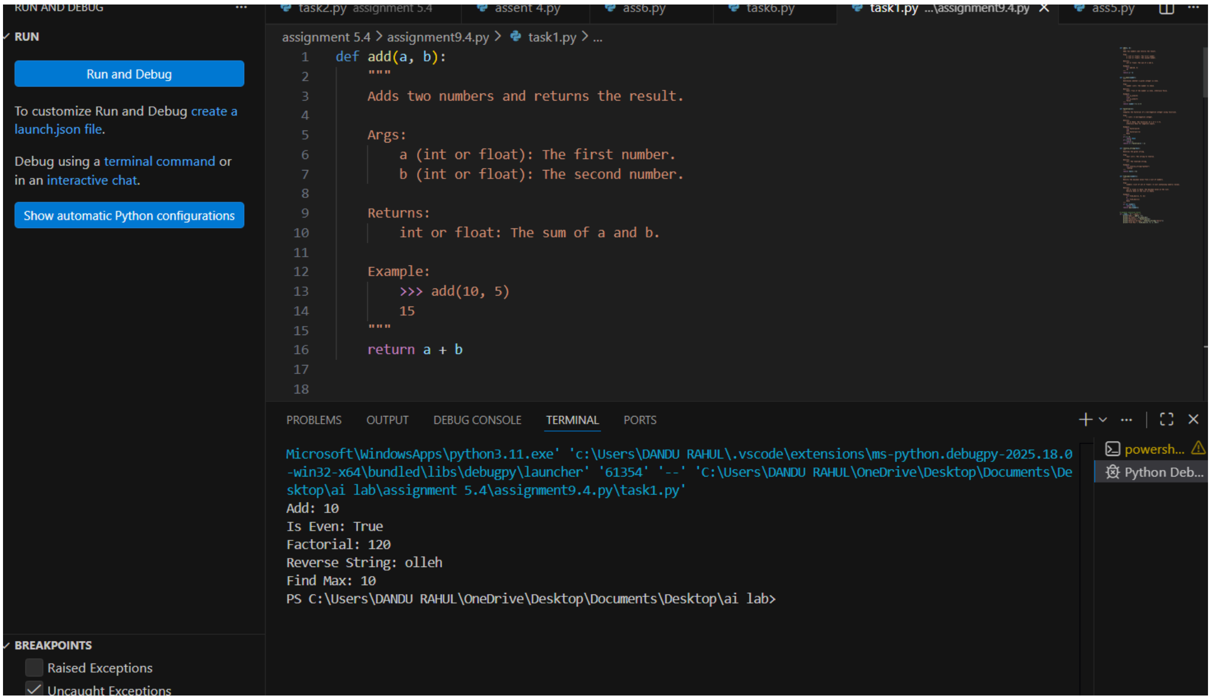
CODE:

```python
ef add(a, b):
    """
    Adds two numbers and returns the result.

    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:
        >>> add(10, 5)
        15
    """
    return a + b


def is_even(number):
    """
    Determines whether a given integer is even.

    Args:
        number (int): The number to check.

    Returns:
        bool: True if the number is even, otherwise False.

    Example:
        >>> is_even(4)
        True
        >>> is_even(7)
        False
    """
    return number % 2 == 0


def factorial(n):
    """
    Computes the factorial of a non-negative integer using recursion.

    Args:
        n (int): A non-negative integer.

    Returns:
        int or None: The factorial of n if n >= 0,
        otherwise None for negative inputs.

    Example:
        >>> factorial(5)
        120
        >>> factorial(-2)
        None
    """
    if n < 0:
        return None
    if n == 0:
        return 1
    return n * factorial(n - 1)


def reverse_string(text):
    """
    Reverses the given string.

    Args:
        text (str): The string to reverse.

    Returns:
        str: The reversed string.

    Example:
        >>> reverse_string("python")
        'nohtyp'
    """
    return text[::-1]


def find_max(numbers):
    """
    Returns the maximum value from a list of numbers.

    Args:
        numbers (list of int or float): A list containing numeric values.
```



```
    Returns:
        int or float or None: The maximum value in the list.
        Returns None if the list is empty.

    Example:
        >>> find_max([1, 9, 3])
        9
        >>> find_max([])
        None
    """
    if not numbers:
        return None
    return max(numbers)


# Example execution block
if __name__ == "__main__":
    print("Add:", add(3, 7))
    print("Is Even:", is_even(10))
    print("Factorial:", factorial(5))
    print("Reverse String:", reverse_string("hello"))
    print("Find Max:", find_max([4, 8, 2, 10]))
```

OUTPUT:



RUN AND DEBUG
RUN
Run and Debug
To customize Run and Debug create a launch.json file.
Debug using a terminal command or in an interactive chat.
Show automatic Python configurations
BREAKPOINTS
□ Raised Exceptions
☑ Uncaught Exceptions
task2.py assignment 5.4
assent 4.py
ass6.py
task6.py
task1.py ...\assignment9.4.py
ass5.py
assignment 5.4 > assignment9.4.py > task1.py > ...
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
powersh...
Python Deb...

```
def add(a, b):
    """
    Adds two numbers and returns the result.

    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:
        >>> add(10, 5)
        15
    """
    return a + b


```

```
Microsoft\WindowsApps\python3.11.exe' 'c:\Users\DANDU RAHUL\.vscode\extensions\ms-python.debugpy-2025.18.0
-win32-x64\bundled\libs\debugpy\launcher' '61354' '--' 'C:\Users\DANDU RAHUL\OneDrive\Desktop\Documents\De
sktop\ai lab\assignment 5.4\assignment9.4.py\task1.py'
Add: 10
Is Even: True
Factorial: 120
Reverse String: olleh
Find Max: 10
PS C:\Users\DANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai lab>
```

Analysis:

The given Python functions had no documentation.
This made it difficult for new developers to understand the code.
An AI tool was used to generate Google-style docstrings.
The docstrings included description, parameters, return values, and example usage.
After adding documentation, the code became easier to read and maintain.

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.

Expected Outcome

- **A Python script with concise, meaningful inline comments**
- **Comments that explain why the logic exists, not what Python syntax does**
- **Noticeable improvement in code readability**

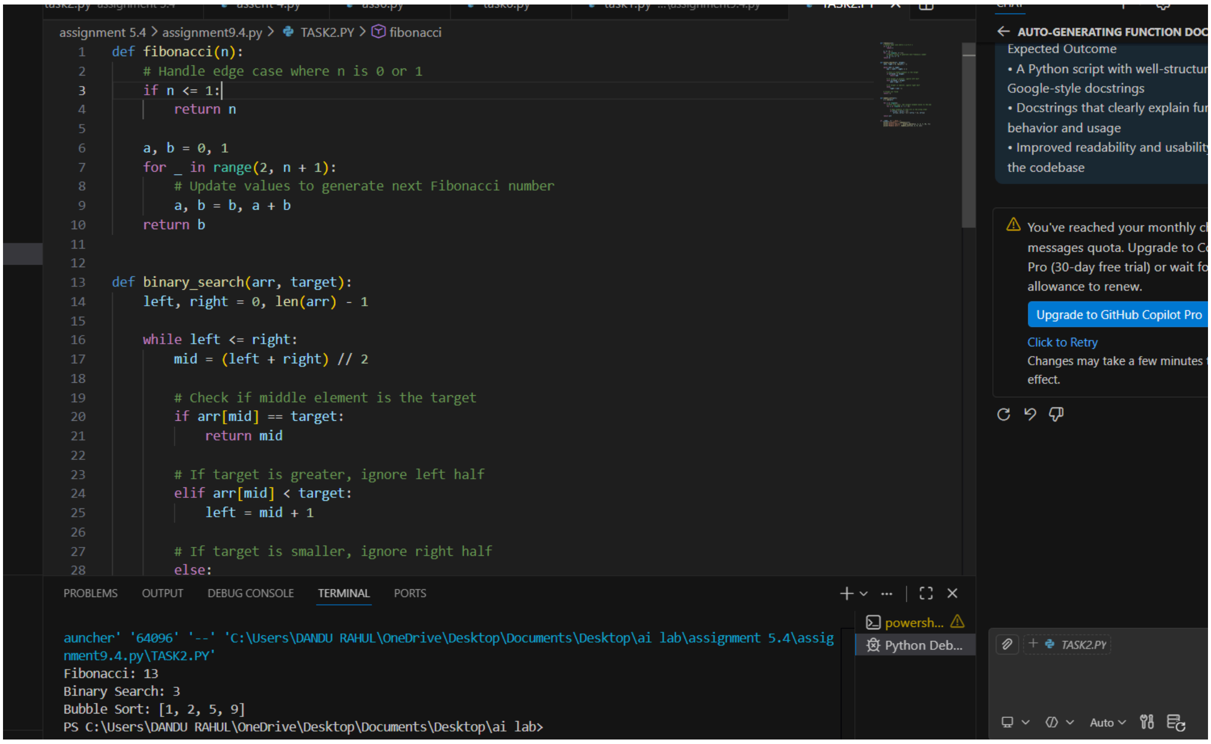
Prompt:

# Add concise inline comments explaining complex or non-obvious logic.
# Do not comment simple syntax.
# Explain why the logic is used, not what Python syntax does.
# Keep comments clear and minima**l.**

Code:

```
def fibonacci(n):
    # Handle edge case where n is 0 or 1
    if n <= 1:
        return n

    a, b = 0, 1
    for _ in range(2, n + 1):
        # Update values to generate next Fibonacci number
        a, b = b, a + b
    return b


def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        # Check if middle element is the target
        if arr[mid] == target:
            return mid

        # If target is greater, ignore left half
        elif arr[mid] < target:
            left = mid + 1

        # If target is smaller, ignore right half
        else:
            right = mid - 1

    # Target not found
    return -1


def bubble_sort(arr):
    n = len(arr)

    for i in range(n):
        # After each pass, the largest element moves to the end
        for j in range(0, n - i - 1):

            # Swap elements if they are in the wrong order
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

    return arr


if __name__ == "__main__":
    print("Fibonacci:", fibonacci(7))
    print("Binary Search:", binary_search([1, 3, 5, 7, 9], 7))
    print("Bubble Sort:", bubble_sort([5, 2, 9, 1]))
```

Output:



```python
def fibonacci(n):
    # Handle edge case where n is 0 or 1
    if n <= 1:
        return n

    a, b = 0, 1
    for _ in range(2, n + 1):
        # Update values to generate next Fibonacci number
        a, b = b, a + b
    return b


def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        # Check if middle element is the target
        if arr[mid] == target:
            return mid

        # If target is greater, ignore left half
        elif arr[mid] < target:
            left = mid + 1

        # If target is smaller, ignore right half
        else:
```

```
auncher' '64096' '--' 'C:\Users\DANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai lab\assignment 5.4\assig
nment9.4.py\TASK2.PY'
Fibonacci: 13
Binary Search: 3
Bubble Sort: [1, 2, 5, 9]
PS C:\Users\DANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai lab>
```

AUTO-GENERATING FUNCTION DOC

Expected Outcome
• A Python script with well-structur
Google-style docstrings
• Docstrings that clearly explain fu
behavior and usage
• Improved readability and usabilit
the codebase

⚠ You've reached your monthly c
messages quota. Upgrade to C
Pro (30-day free trial) or wait fo
allowance to renew.

Upgrade to GitHub Copilot Pro

Click to Retry
Changes may take a few minutes
effect.

Analysis:

The Python program had complex logic that was hard to understand.
AI was used to add inline comments for difficult parts of the code.
Only complex logic was commented, not simple syntax.
The comments explain why the logic is used.
This improved readability and maintainability of the code.

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:

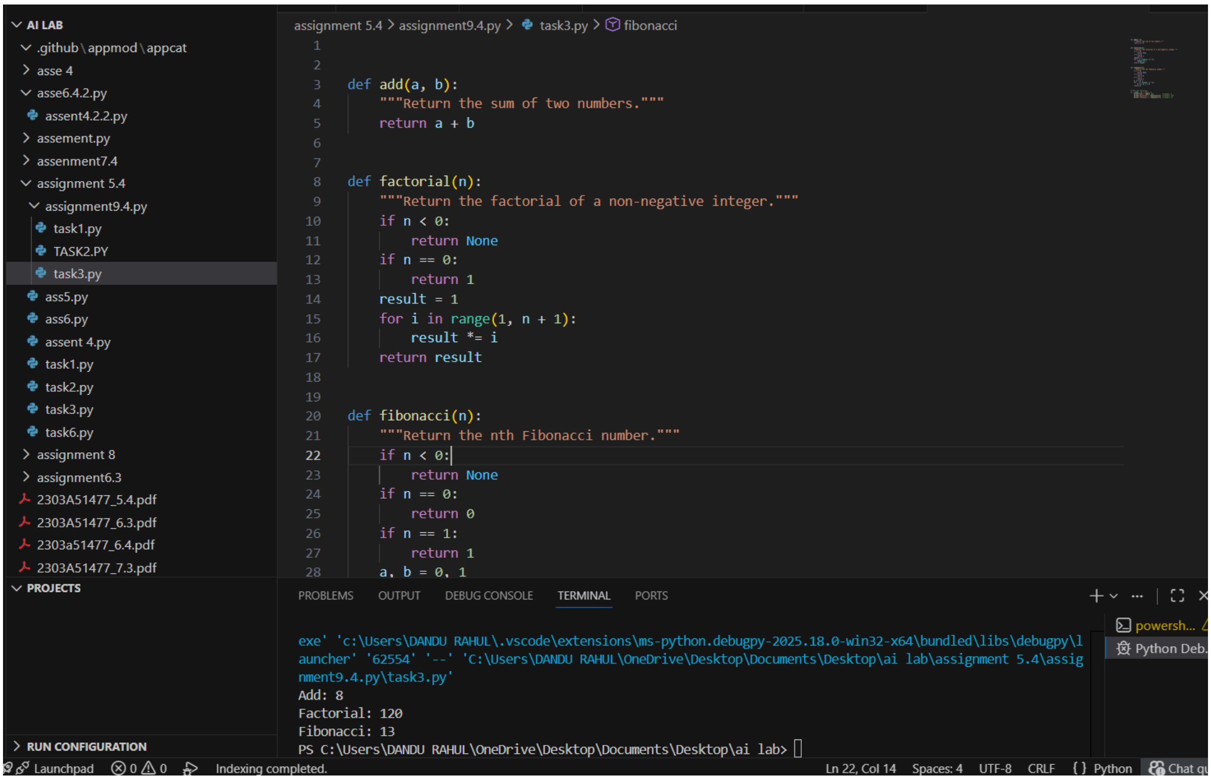
```
# Generate a professional module-level docstring for this Python file.
# Include:
# - Purpose of the module
# - Required libraries or dependencies
```

# - Brief description of key functions or classes
# - Short example of how to use the module
# Keep it clear and professional.

CODE:

```
def add(a, b):
    """Return the sum of two numbers."""
    return a + b


def factorial(n):
    """Return the factorial of a non-negative integer."""
    if n < 0:
        return None
    if n == 0:
        return 1
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result


def fibonacci(n):
    """Return the nth Fibonacci number."""
    if n < 0:
        return None
    if n == 0:
        return 0
    if n == 1:
        return 1
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b


# Test the functions
if __name__ == "__main__":
    print("Add:", add(5, 3))             # Output: 8
    print("Factorial:", factorial(5))    # Output: 120
    print("Fibonacci:", fibonacci(7))    # Output: 13
```

OUTPUT:



```python


def add(a, b):
    """Return the sum of two numbers."""
    return a + b


def factorial(n):
    """Return the factorial of a non-negative integer."""
    if n < 0:
        return None
    if n == 0:
        return 1
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result


def fibonacci(n):
    """Return the nth Fibonacci number."""
    if n < 0:
        return None
    if n == 0:
        return 0
    if n == 1:
        return 1
    a, b = 0, 1
```

```
exe' 'c:\Users\DANDU RAHUL\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\l
auncher' '62554' '--' 'C:\Users\DANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai lab\assignment 5.4\assig
nment9.4.py\task3.py'
Add: 8
Factorial: 120
Fibonacci: 13
PS C:\Users\DANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai lab> 
```

Analysis:

The module has functions like add, factorial, and fibonacci.

Without a docstring, it’s hard to understand the module.

AI was used to add a professional module-level docstring.

The docstring explains purpose, dependencies, functions, and example usage.

This makes the code easier to read and maintain.

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

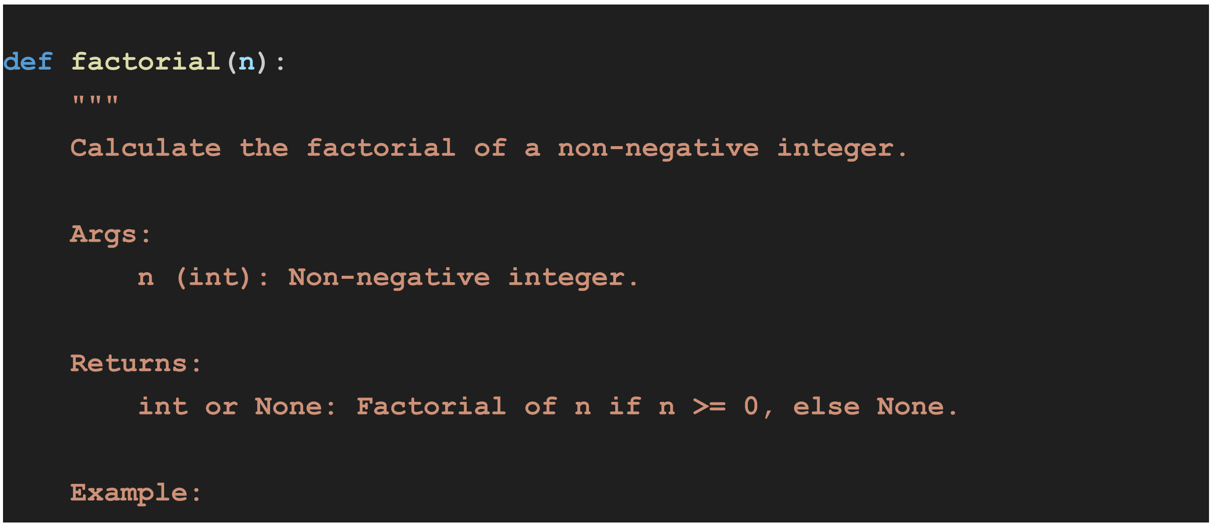
Expected Outcome

- Functions with clean, standardized docstrings
- Reduced clutter inside function bodies
- Improved consistency across the codebase

Prompt:

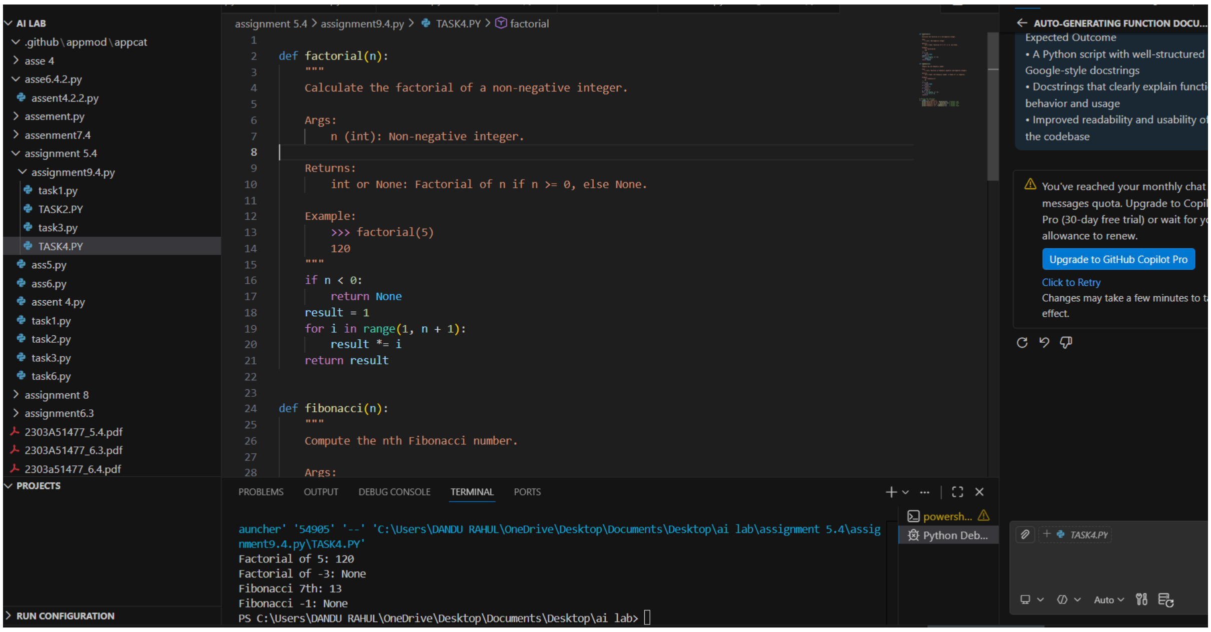
# Add a module-level docstring for this Python file.
# Include:
# - Purpose of the module
# - Required libraries or dependencies
# - Key functions or classes
# - Short example usage
# Keep it clear and professional.

CODE:



```python
def factorial(n):
    """
    Calculate the factorial of a non-negative integer.

    Args:
        n (int): Non-negative integer.

    Returns:
        int or None: Factorial of n if n >= 0, else None.

    Example:
        >>> factorial(5)
        120
    """
    if n < 0:
        return None
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result


def fibonacci(n):
    """
    Compute the nth Fibonacci number.

    Args:
        n (int): Position in Fibonacci sequence (non-negative integer).

    Returns:
        int or None: nth Fibonacci number, or None if n is negative.

    Example:
        >>> fibonacci(7)
        13
    """
    if n < 0:
        return None
    if n == 0:
        return 0
    if n == 1:
        return 1
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b


# Testing the functions
if __name__ == "__main__":
    print("Factorial of 5:", factorial(5))    # Output: 120
    print("Factorial of -3:", factorial(-3)) # Output: None
    print("Fibonacci 7th:", fibonacci(7))     # Output: 13
    print("Fibonacci -1:", fibonacci(-1))     # Output: None
```

OUTPUT:



```python

def factorial(n):
    """
    Calculate the factorial of a non-negative integer.

    Args:
        n (int): Non-negative integer.

    Returns:
        int or None: Factorial of n if n >= 0, else None.

    Example:
        >>> factorial(5)
        120
    """
    if n < 0:
        return None
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result


def fibonacci(n):
    """

    Compute the nth Fibonacci number.

    Args:
```

```
auncher' '54905' '--' 'C:\Users\DANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai lab\assignment 5.4\assig
nment9.4.py\TASK4.PY'
Factorial of 5: 120
Factorial of -3: None
Fibonacci 7th: 13
Fibonacci -1: None
PS C:\Users\DANDU RAHUL\OneDrive\Desktop\Documents\Desktop\ai lab>
```

Expected Outcome
• A Python script with well-structured Google-style docstrings
• Docstrings that clearly explain functi behavior and usage
• Improved readability and usability of the codebase

You've reached your monthly chat messages quota. Upgrade to Copil Pro (30-day free trial) or wait for yo allowance to renew.

Changes may take a few minutes to t effect.

analysis :

- The code had long inline comments inside functions.
- AI converted them into proper docstrings.
- Docstrings include purpose, arguments, return values, and examples.
- Inline comments were removed to make code clean.
- Now the code is easier to read and maintain.

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.

Expected Outcome

- **A working Python script that processes another .py file**
- **Automatically inserted placeholder docstrings**
- **Clear demonstration of how AI can assist in documentation Automatio**

Prompt:

# Write a Python utility that reads a given .py file,
# detects all functions and classes, and inserts placeholder
# Google-style docstrings for each function or class.
# The docstrings should be in the format:
# """TODO: Add description."""
# Save the updated content to a new file.
# Keep the code clean and runnable.

CODE:

```
 import ast
import sys


def generate_function_docstring(func_node):
    """Generate a Google-style placeholder docstring for a function."""

    params = [arg.arg for arg in func_node.args.args]
    indent = " " * (func_node.col_offset + 4)

    docstring = f'{indent}"""\n'
    docstring += f'{indent}TODO: Describe the purpose of
`{func_node.name}`.\n\n'

    if params:
        docstring += f"{indent}Args:\n"
        for param in params:
            docstring += f"{indent}    {param} (TYPE): Description.\n"
        docstring += "\n"

    docstring += f"{indent}Returns:\n"
    docstring += f"{indent}    TYPE: Description.\n"
    docstring += f'{indent}"""\n'

    return docstring
```

```python
def generate_class_docstring(class_node):
    """Generate a Google-style placeholder docstring for a class."""

    indent = " " * (class_node.col_offset + 4)

    docstring = f'{indent}"""\n'
    docstring += f'{indent}TODO: Describe the purpose of class `{class_node.name}`.\n\n'
    docstring += f"{indent}Attributes:\n"
    docstring += f"{indent}    TODO: Add class attributes.\n"
    docstring += f'{indent}"""\n'

    return docstring


def insert_docstrings(source_code):
    """Insert docstrings into functions and classes without docstrings."""

    tree = ast.parse(source_code)
    lines = source_code.split("\n")
    offset = 0

    for node in ast.walk(tree):
        if isinstance(node, (ast.FunctionDef, ast.ClassDef)):

            if ast.get_docstring(node) is not None:
                continue

            insert_line = node.body[0].lineno - 1 + offset

            if isinstance(node, ast.FunctionDef):
                docstring = generate_function_docstring(node)
            else:
                docstring = generate_class_docstring(node)

            lines.insert(insert_line, docstring.rstrip("\n"))
            offset += docstring.count("\n")

    return "\n".join(lines)
```

```
def process_file(filename):
    """Read file, insert docstrings, and save updated version."""

    with open(filename, "r", encoding="utf-8") as file:
        source_code = file.read()

    updated_code = insert_docstrings(source_code)

    output_filename = "updated_" + filename

    with open(output_filename, "w", encoding="utf-8") as file:
        file.write(updated_code)

    print("Documentation added successfully!")
    print("Output saved to:", output_filename)


if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python auto_doc_generator.py <file.py>")
    else:
        process_file(sys.argv[1])
```

Analysis:

- **This tool reads a Python `.py` file.**
- **It finds all functions and classes.**
- **If they don't have docstrings, it adds Google-style placeholder docstrings.**
- **The updated file is saved as `updated_<filename>.py`.**
- **Helps developers save time and keep code readable.**
- **Does not write real descriptions, only placeholders.**
- **Can be improved later using AI to generate real docstrings.**