

ASSIGNMENT – 9.4

G. Srinidhi

2303A52285

Batch – 43

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 google-style docstrings for the following Python functions.

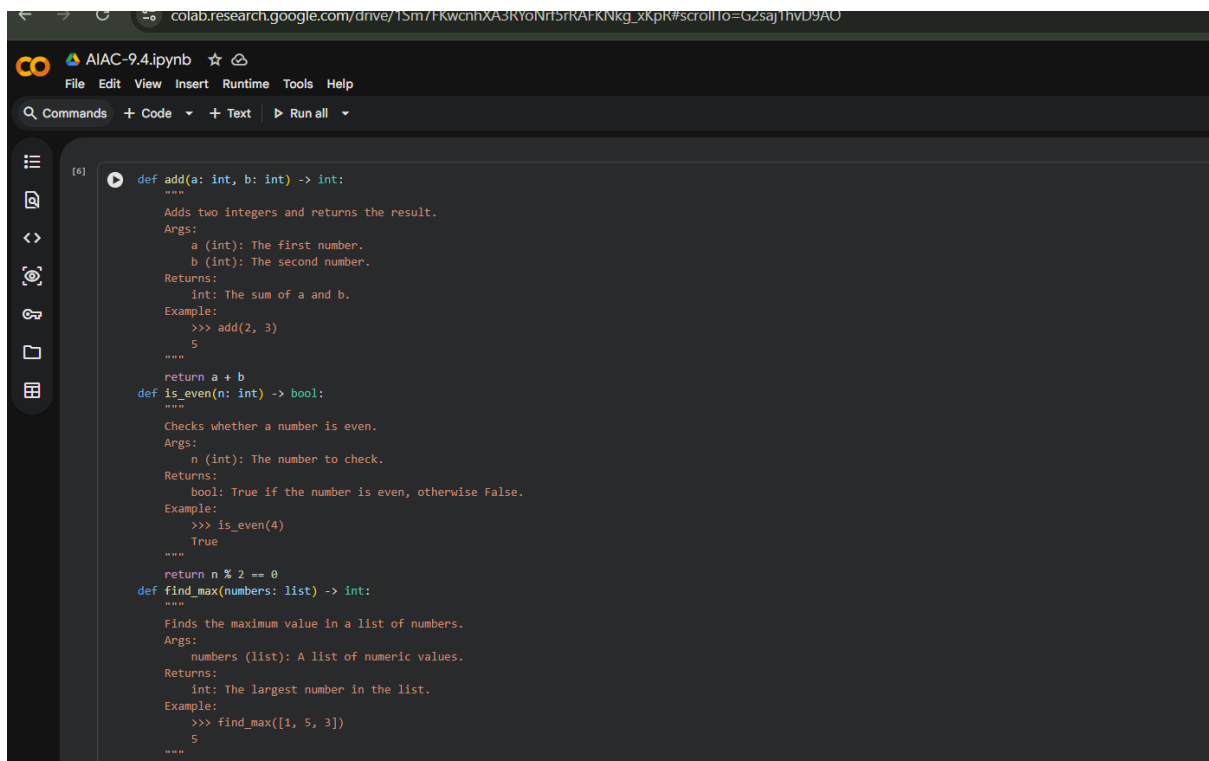
Each docstring should include:

- A brief description of the function
- Parameters with data types and short explanations
- Return values with data types
- At least one example usage in doctest format (if applicable)

Do not modify the original function logic.

Ensure the formatting follows the Google Python Style Guide.

Keep the tone professional and clear for use in a shared codebase.

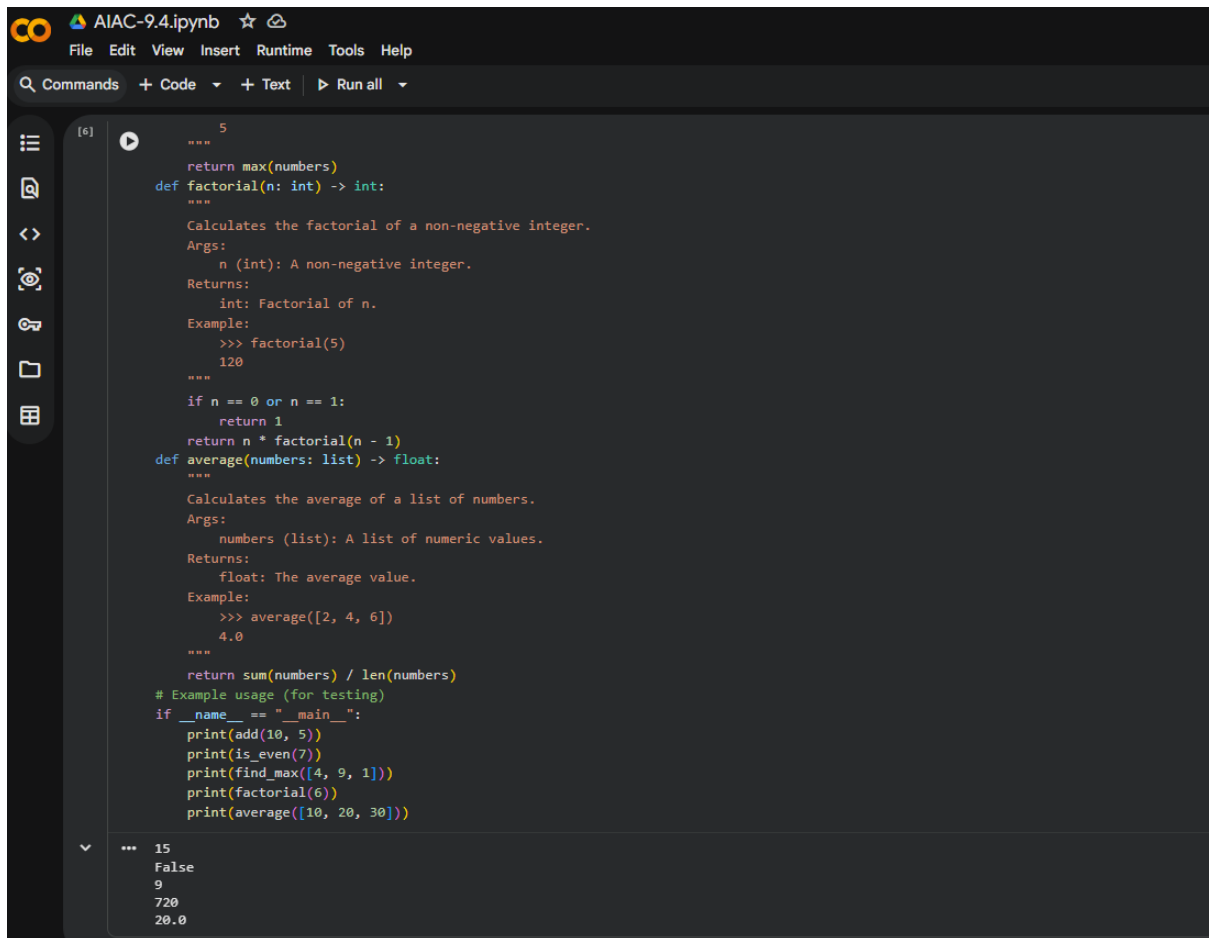


The screenshot shows a Google Colab notebook interface with a dark theme. The notebook is titled "AIAC-9.4.ipynb". The code cell contains three Python functions, each with a Google-style docstring. The first function, `add`, takes two integers and returns their sum. The second function, `is_even`, checks if a number is even. The third function, `find_max`, finds the maximum value in a list of numbers. Each docstring includes a brief description, parameter information, return type, and an example usage in doctest format.

```
[6] def add(a: int, b: int) -> int:
    """
    Adds two integers and returns the result.
    Args:
        a (int): The first number.
        b (int): The second number.
    Returns:
        int: The sum of a and b.
    Example:
        >>> add(2, 3)
        5
    """
    return a + b

def is_even(n: int) -> bool:
    """
    Checks whether a number is even.
    Args:
        n (int): The number to check.
    Returns:
        bool: True if the number is even, otherwise False.
    Example:
        >>> is_even(4)
        True
    """
    return n % 2 == 0

def find_max(numbers: list) -> int:
    """
    Finds the maximum value in a list of numbers.
    Args:
        numbers (list): A list of numeric values.
    Returns:
        int: The largest number in the list.
    Example:
        >>> find_max([1, 5, 3])
        5
    """
```



The screenshot shows a Jupyter Notebook window titled "AIAC-9.4.ipynb". The interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help) and a toolbar (Commands, + Code, + Text, Run all). The code is written in a dark-themed editor. It defines two functions: `factorial` and `average`. The `factorial` function takes an integer `n` and returns its factorial. The `average` function takes a list of numbers and returns their average. Both functions include docstrings with descriptions, arguments, and return values. Example calls are shown for both functions. At the bottom, a code cell shows the output of the example usage: 15, False, 9, 720, and 20.0.

```
[6]
5
"""
return max(numbers)
def factorial(n: int) -> int:
    """
    Calculates the factorial of a non-negative integer.
    Args:
        n (int): A non-negative integer.
    Returns:
        int: Factorial of n.
    Example:
        >>> factorial(5)
        120
    """
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
def average(numbers: list) -> float:
    """
    Calculates the average of a list of numbers.
    Args:
        numbers (list): A list of numeric values.
    Returns:
        float: The average value.
    Example:
        >>> average([2, 4, 6])
        4.0
    """
    return sum(numbers) / len(numbers)
# Example usage (for testing)
if __name__ == "__main__":
    print(add(10, 5))
    print(is_even(7))
    print(find_max([4, 9, 1]))
    print(factorial(6))
    print(average([10, 20, 30]))

*** 15
False
9
720
20.0
```

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

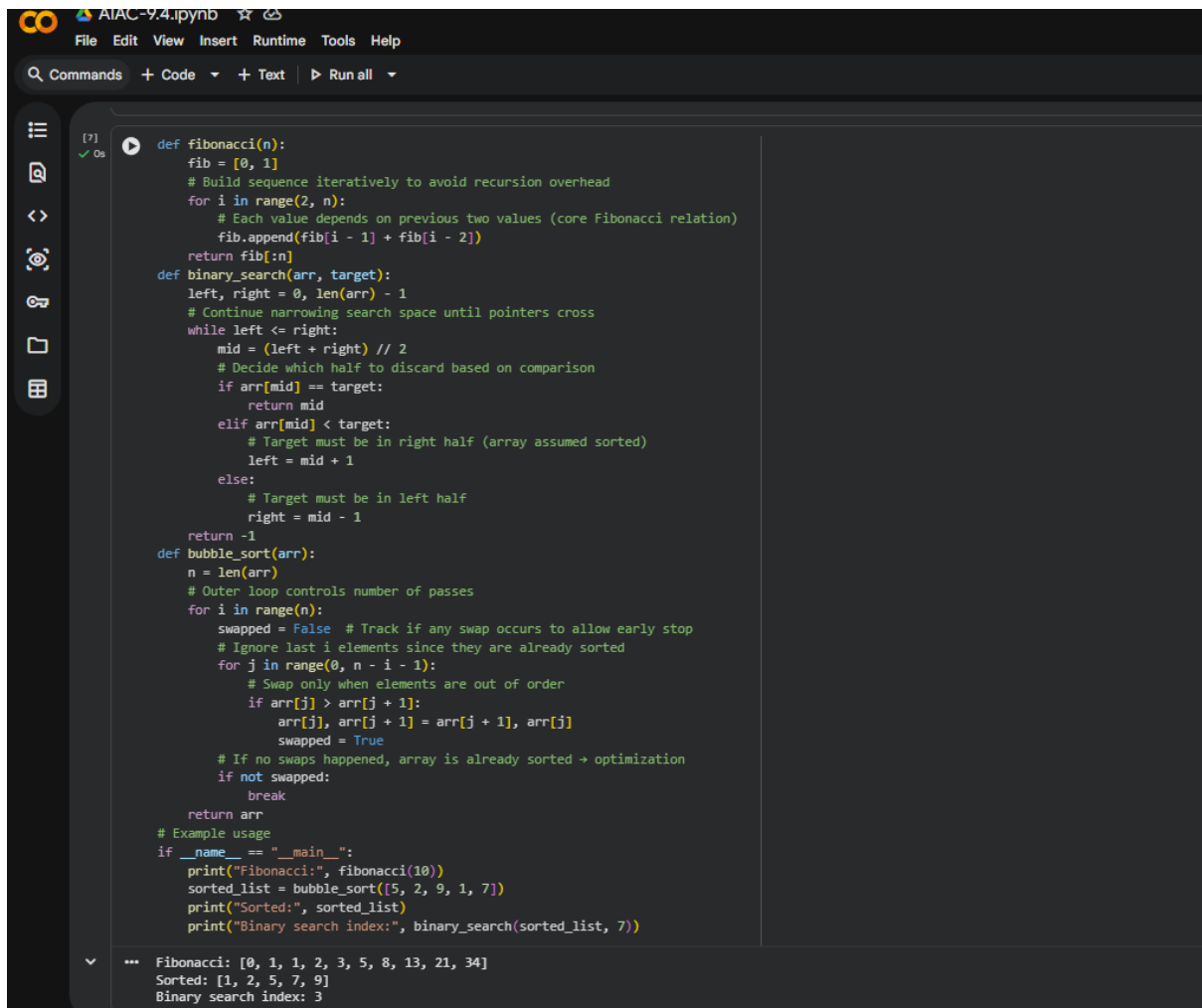
You are a senior Python developer reviewing a codebase for readability improvements.

The following Python script contains complex logic, including loops, conditional statements, and algorithms. The code works correctly but lacks clarity.

Your task:

- Insert concise inline comments ONLY where the logic is complex or non-obvious.
- Explain WHY the logic exists, not what basic Python syntax does.
- Avoid commenting on trivial or self-explanatory lines.
- Do not modify the original logic or structure.
- Keep comments professional and minimal to avoid clutter.

Return the improved version of the script with meaningful inline comments added.



```
def fibonacci(n):
    fib = [0, 1]
    # Build sequence iteratively to avoid recursion overhead
    for i in range(2, n):
        # Each value depends on previous two values (core Fibonacci relation)
        fib.append(fib[i - 1] + fib[i - 2])
    return fib[:n]

def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    # Continue narrowing search space until pointers cross
    while left <= right:
        mid = (left + right) // 2
        # Decide which half to discard based on comparison
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            # Target must be in right half (array assumed sorted)
            left = mid + 1
        else:
            # Target must be in left half
            right = mid - 1
    return -1

def bubble_sort(arr):
    n = len(arr)
    # Outer loop controls number of passes
    for i in range(n):
        swapped = False # Track if any swap occurs to allow early stop
        # Ignore last i elements since they are already sorted
        for j in range(0, n - i - 1):
            # Swap only when elements are out of order
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # If no swaps happened, array is already sorted -> optimization
        if not swapped:
            break
    return arr

# Example usage
if __name__ == "__main__":
    print("Fibonacci:", fibonacci(10))
    sorted_list = bubble_sort([5, 2, 9, 1, 7])
    print("Sorted:", sorted_list)
    print("Binary search index:", binary_search(sorted_list, 7))
```

... Fibonacci: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Sorted: [1, 2, 5, 7, 9]
Binary search index: 3

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.

Expected Outcome

- A well-written multi-line module-level docstring
- Clear overview of what the module does and how to use it
- Documentation suitable for real-world projects or repositories

Prompt

You are a Python documentation expert preparing a module for internal or public use. Analyze the following complete Python module and generate a professional multi-line module-level docstring to be placed at the top of the file. The docstring must include:

- A clear explanation of the module's purpose
- Required libraries or dependencies
- A brief description of key functions and classes
- A short example demonstrating how to use the module
- Professional tone suitable for production code

Do not modify the existing code.

Only generate the module-level docstring.

```
Q Commands + Code + Text ▶ Run all

[5] utility_algorithms.py
Purpose:
    This module provides commonly used utility algorithms for numerical
    computation and data processing. It includes implementations for
    Fibonacci sequence generation, sorting, and searching operations.
    The module is designed to be reusable across projects and easy to
    understand with clear function-level documentation.
Dependencies:
    - No external libraries required (uses only Python standard features).
Key Functions:
    - fibonacci(n):
        Generates the first n numbers in the Fibonacci sequence using
        an iterative approach.
    - bubble_sort(arr):
        Sorts a list using the Bubble Sort algorithm with an early-stop
        optimization when the list becomes sorted.
    - binary_search(arr, target):
        Performs binary search on a sorted list and returns the index
        of the target element if found, otherwise -1.
Example Usage:
    >>> from utility_algorithms import fibonacci, bubble_sort, binary_search
    >>> fibonacci(5)
    [0, 1, 1, 2, 3]
    >>> data = bubble_sort([5, 2, 9, 1])
    >>> binary_search(data, 9)
    3
'''
def fibonacci(n):
    fib = [0, 1]
    for i in range(2, n):
        fib.append(fib[i - 1] + fib[i - 2])
    return fib[:n]
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
if __name__ == "__main__":
    print(fibonacci(10))
    sorted_data = bubble_sort([5, 3, 8, 1])
    print(sorted_data)
    print(binary_search(sorted_data, 8))

'''
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
[1, 3, 5, 8]
3
```

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

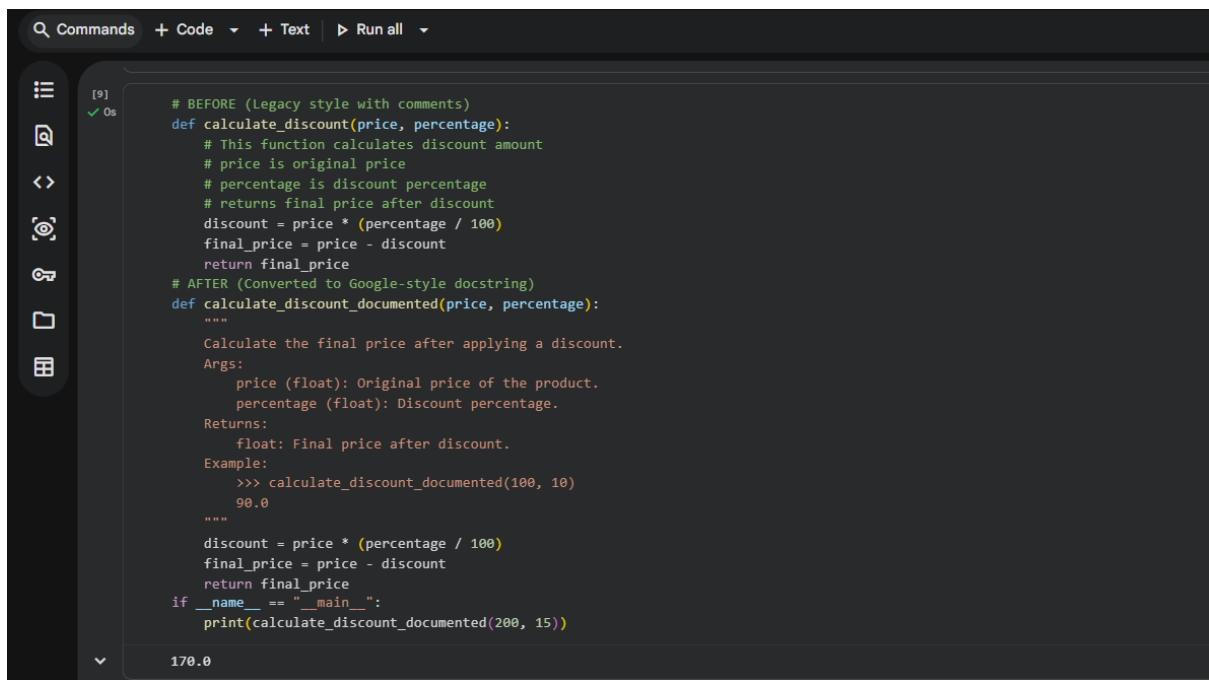
Prompt

You are a Python refactoring specialist helping standardize documentation in a legacy project. The following script contains detailed inline comments inside functions explaining their behavior. These should be converted into structured docstrings.

Your task:

- Convert relevant explanatory comments into proper Google-style (or NumPy-style) docstrings.
- Preserve the original meaning and intent.
- Include:
 - Description
 - Parameters with types
 - Return values with types
- Remove redundant inline comments after conversion.
- Do not modify function logic.

Return the cleaned and standardized version of the script.

A screenshot of a code editor interface. The top bar shows 'Commands', '+ Code', '+ Text', and 'Run all'. The left sidebar has icons for file explorer, search, and other tools. The main editor area displays Python code. It starts with a comment '# BEFORE (legacy style with comments)' followed by a function definition 'def calculate_discount(price, percentage):'. The function body includes comments and calculations for discount and final price. Below this is a comment '# AFTER (Converted to Google-style docstring)' followed by a function definition 'def calculate_discount_documented(price, percentage):'. This function has a docstring with a description, arguments, returns, and an example. The code ends with a main block that prints the result of 'calculate_discount_documented(200, 15)'. The output '170.0' is shown at the bottom.

```
[9]
✓ Os
# BEFORE (legacy style with comments)
def calculate_discount(price, percentage):
    # This function calculates discount amount
    # price is original price
    # percentage is discount percentage
    # returns final price after discount
    discount = price * (percentage / 100)
    final_price = price - discount
    return final_price

# AFTER (Converted to Google-style docstring)
def calculate_discount_documented(price, percentage):
    """
    Calculate the final price after applying a discount.
    Args:
        price (float): Original price of the product.
        percentage (float): Discount percentage.
    Returns:
        float: Final price after discount.
    Example:
        >>> calculate_discount_documented(100, 10)
        90.0
    """
    discount = price * (percentage / 100)
    final_price = price - discount
    return final_price

if __name__ == "__main__":
    print(calculate_discount_documented(200, 15))

170.0
```

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 automation

Prompt

You are a Python developer building an internal documentation scaffolding tool. Design a small Python utility that:

- Reads a given .py file
- Automatically detects all functions and classes
- Inserts placeholder Google-style docstrings if none exist
- Preserves indentation and formatting
- Does not modify existing docstrings

The generated placeholder docstrings should include:

- Short description placeholder - Args section
- Returns section (if applicable)

The goal is documentation scaffolding, not perfect documentation.

Provide:

1. A complete working Python script
2. Clear explanation of how it works
3. Example of how to run it

Ensure the solution is clean, readable, and suitable for internal tooling.

```
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all

[18]
✓ Os
import ast
import os
def generate_placeholder(name, node_type):
    """Return a Google-style placeholder docstring."""
    if node_type == "function":
        return [
            "",
            f" {name} function.",
            "",
            " Args:",
            "     TODO: Add parameter descriptions.",
            "",
            " Returns:",
            "     TODO: Add return description.",
            "",
            ""
        ]
    else:
        return [
            "",
            f" {name} class.",
            "",
            " Attributes:",
            "     TODO: Describe attributes.",
            "",
            " Methods:",
            "     TODO: Describe important methods.",
            "",
            ""
        ]
def add_docstrings(input_file, output_file):
    with open(input_file, "r") as f:
        source = f.read()
    tree = ast.parse(source)
    lines = source.split("\n")
    new_lines = lines.copy()
    offset = 0
    for node in ast.walk(tree):
        if isinstance(node, (ast.FunctionDef, ast.ClassDef)):
            if not ast.get_docstring(node): # Only if no docstring exists
```

```
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all

[18]
✓ Os
        if not ast.get_docstring(node): # Only if no docstring exists
            name = node.name
            node_type = "function" if isinstance(node, ast.FunctionDef) else "class"
            doc_lines = generate_placeholder(name, node_type)
            insert_at = node.body[0].lineno - 1 + offset
            for line in reversed(doc_lines):
                new_lines.insert(insert_at, line)
            offset += len(doc_lines)
        with open(output_file, "w") as f:
            f.write("\n".join(new_lines))
# Example usage
if __name__ == "__main__":
    # Create a dummy input file for demonstration
    dummy_code = '''
def my_function(arg1, arg2):
    pass
class MyClass:
    def __init__(self):
        pass
def another_func():
    """Existing docstring"""
    pass
...
    with open("input.py", "w") as f:
        f.write(dummy_code)
    add_docstrings("input.py", "output.py")
    print("Documentation scaffolding completed successfully!")
    # Optionally, print the content of the output file
    print("\nContent of output.py:")
    with open("output.py", "r") as f:
        print(f.read())
    # Clean up the dummy files
    os.remove("input.py")
    os.remove("output.py")
```

Q Commands + Code + Text ▶ Run all

18

0s

▶

```
print(f.read())
# Clean up the dummy files
os.remove("input.py")
os.remove("output.py")
```

⌵

*** Documentation scaffolding completed successfully!

Content of output.py:

```
def my_function(arg1, arg2):
    """
    my_function function.

    Args:
        TODO: Add parameter descriptions.

    Returns:
        TODO: Add return description.
    """
    pass
class MyClass:
    """
    MyClass class.

    Attributes:
        TODO: Describe attributes.

    Methods:
        TODO: Describe important methods.
    """
    def __init__(self):
        """
        __init__ function.

    Args:
        TODO: Add parameter descriptions.

    Returns:
        TODO: Add return description.
    """
    pass
def another_func():
    """Existing docstring"""
    pass
```