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

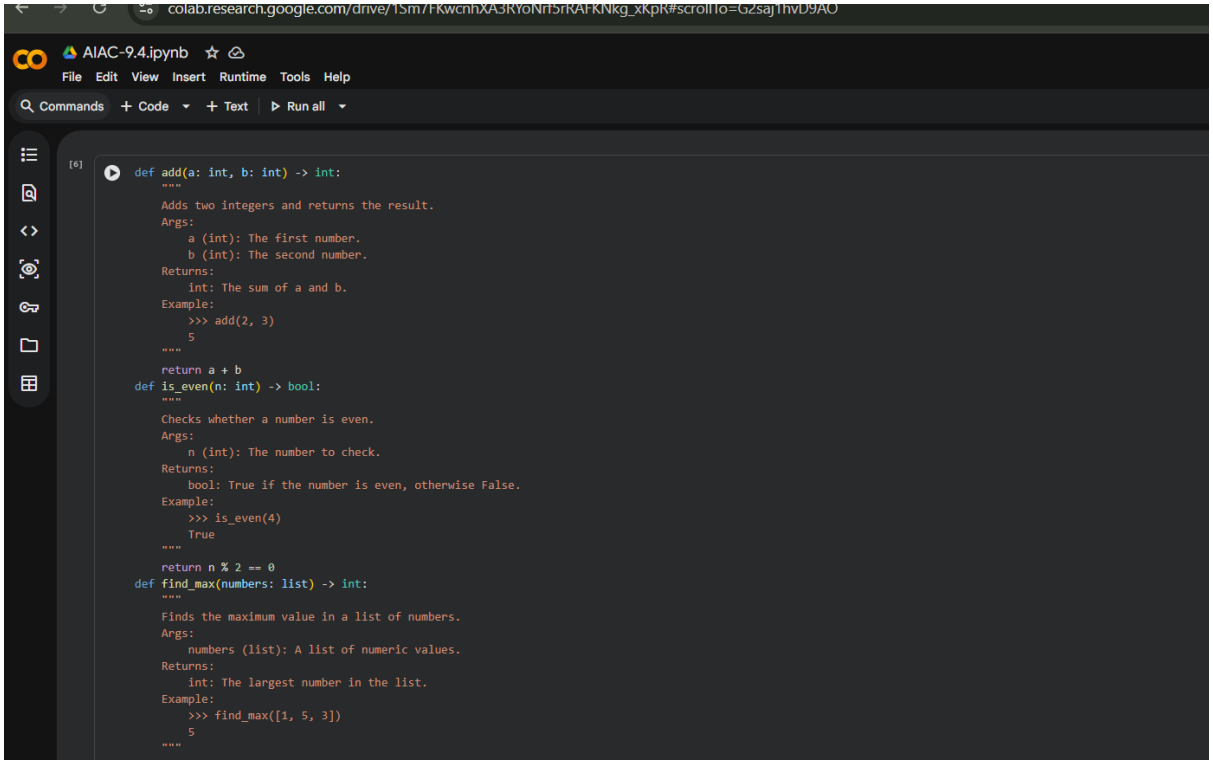
- Return values with data types

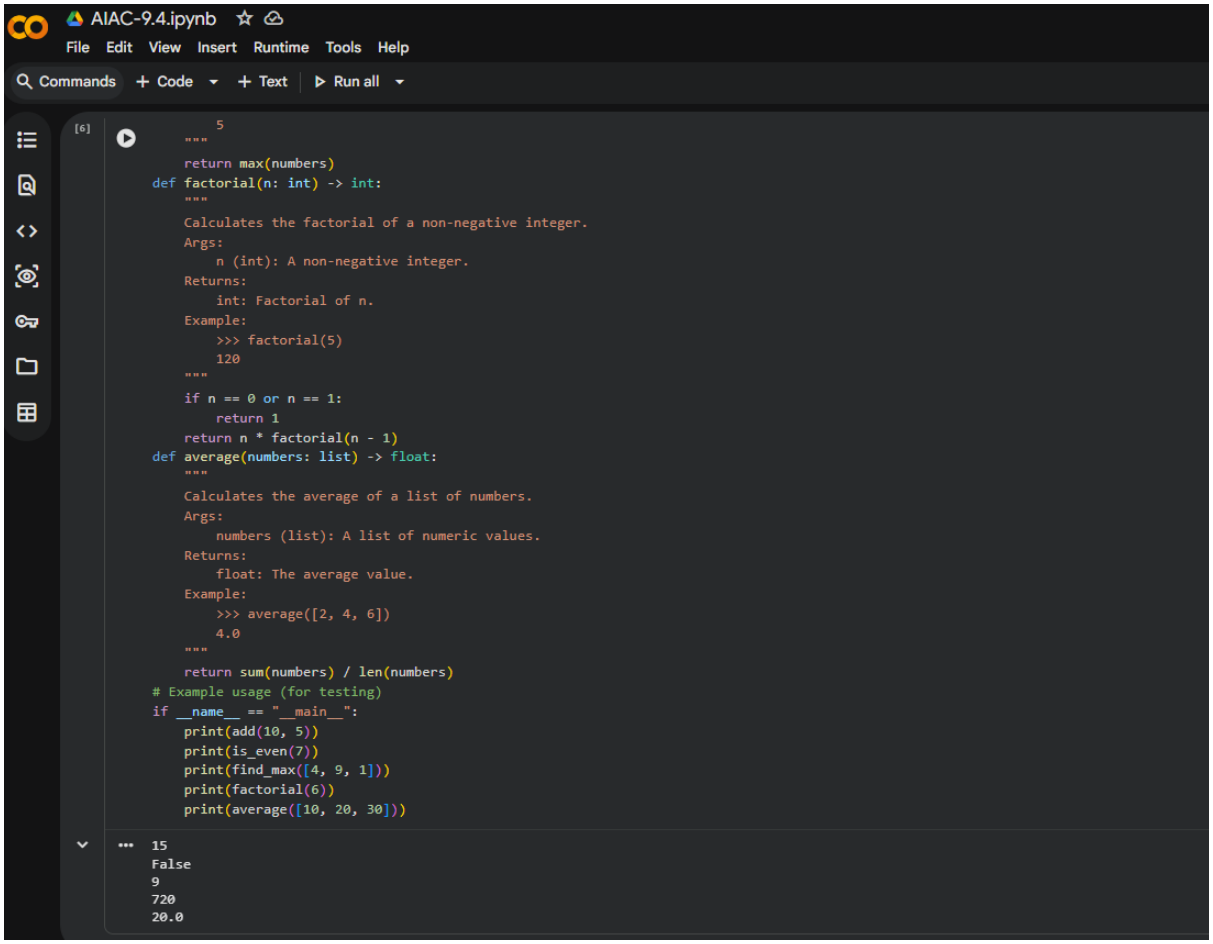
- At least one example usage in doctest format (if applicable)

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.





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.

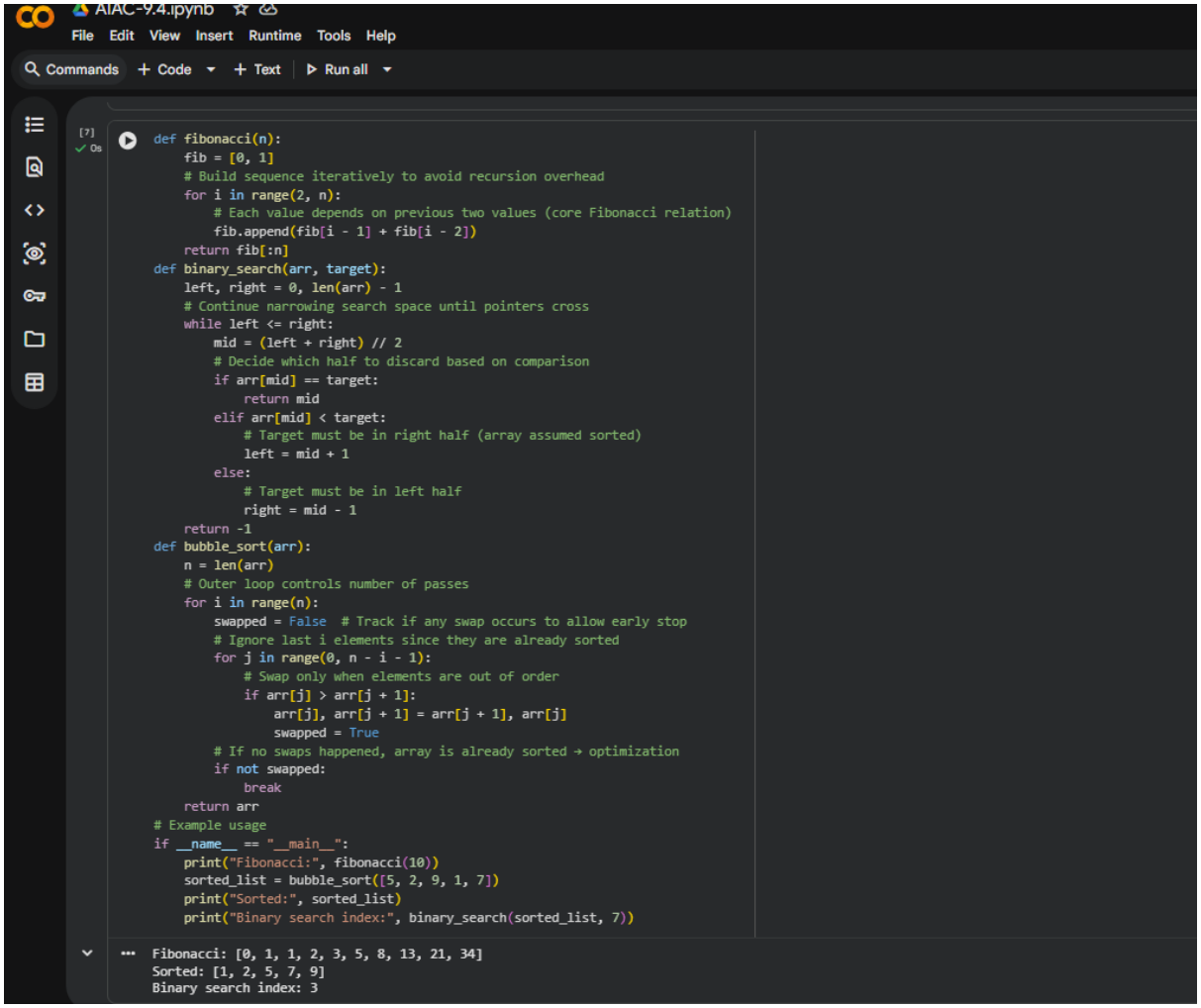
- Explain WHY the logic exists, not what basic Python syntax does.

- 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.



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

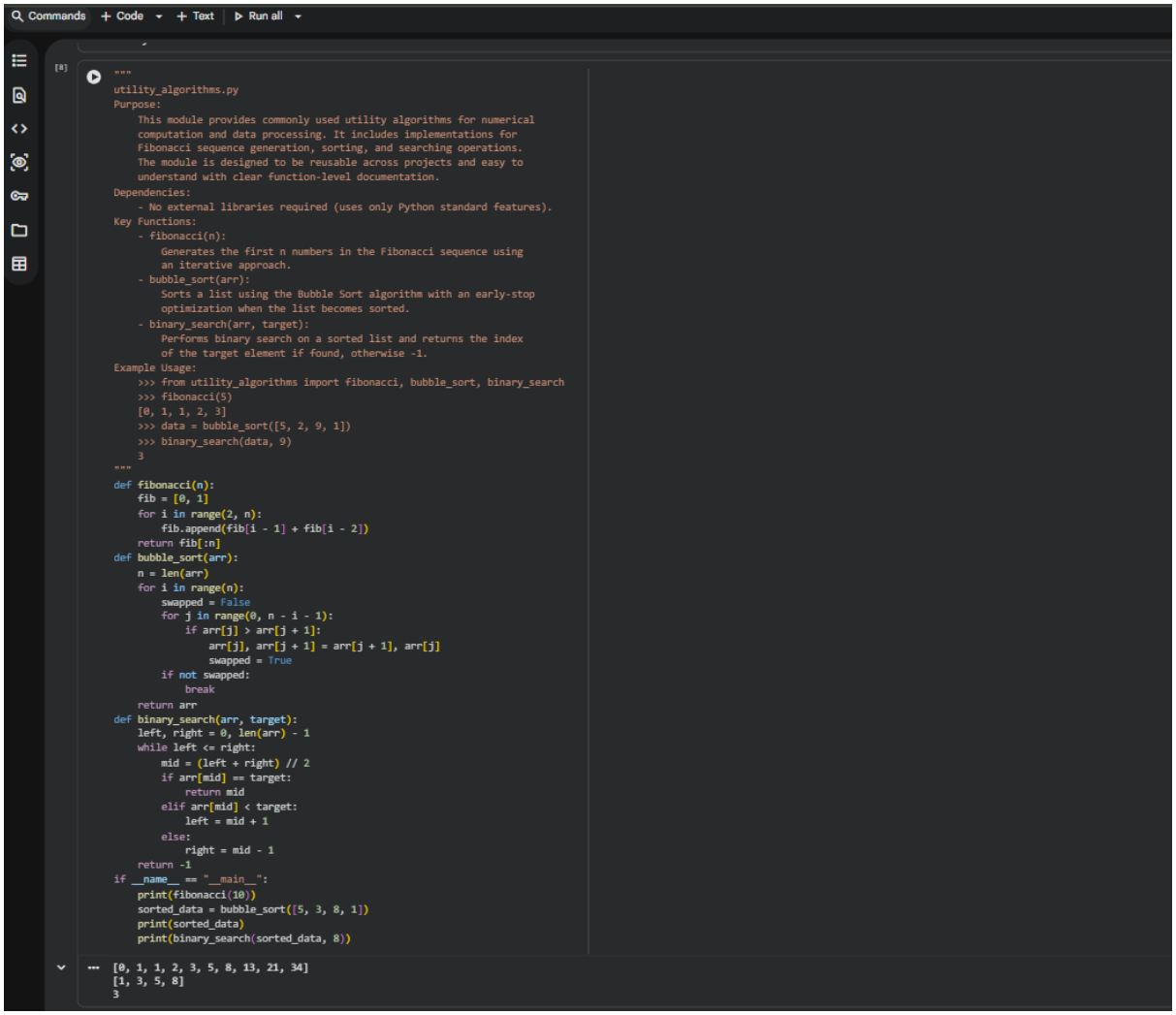
- A brief description of key functions and classes

- 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.



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

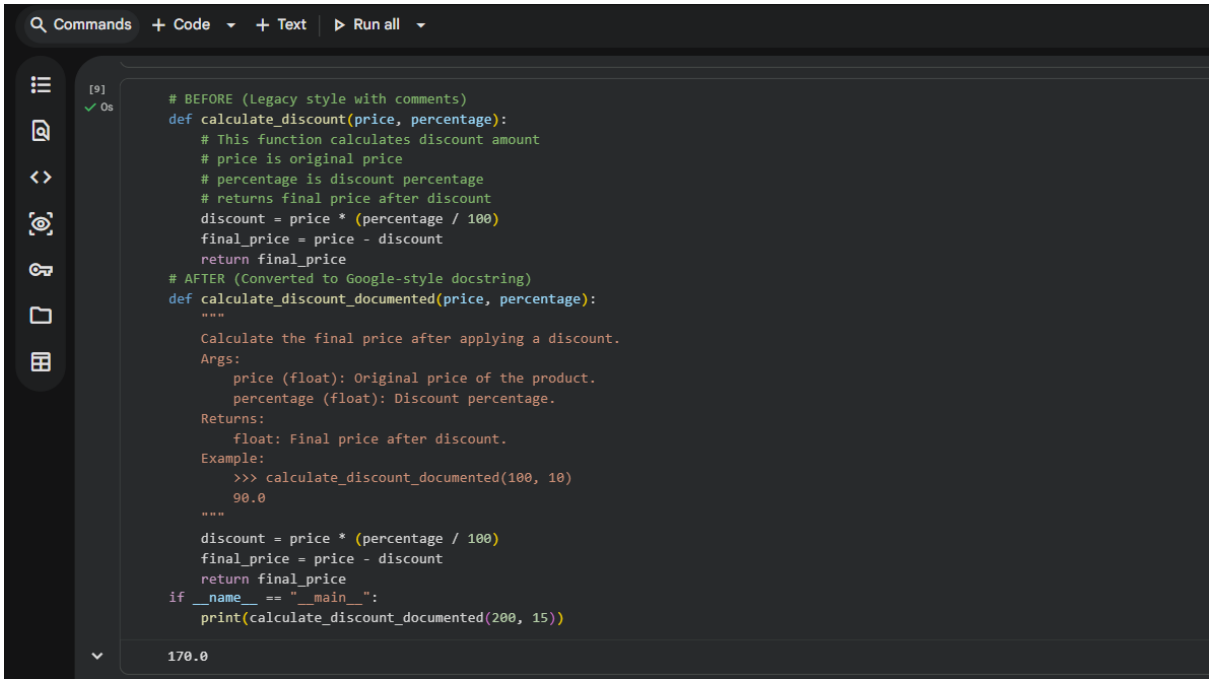
- Parameters with types

- Return values with types

- Remove redundant inline comments after conversion.

- Do not modify function logic.

Return the cleaned and standardized version of the script.



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.

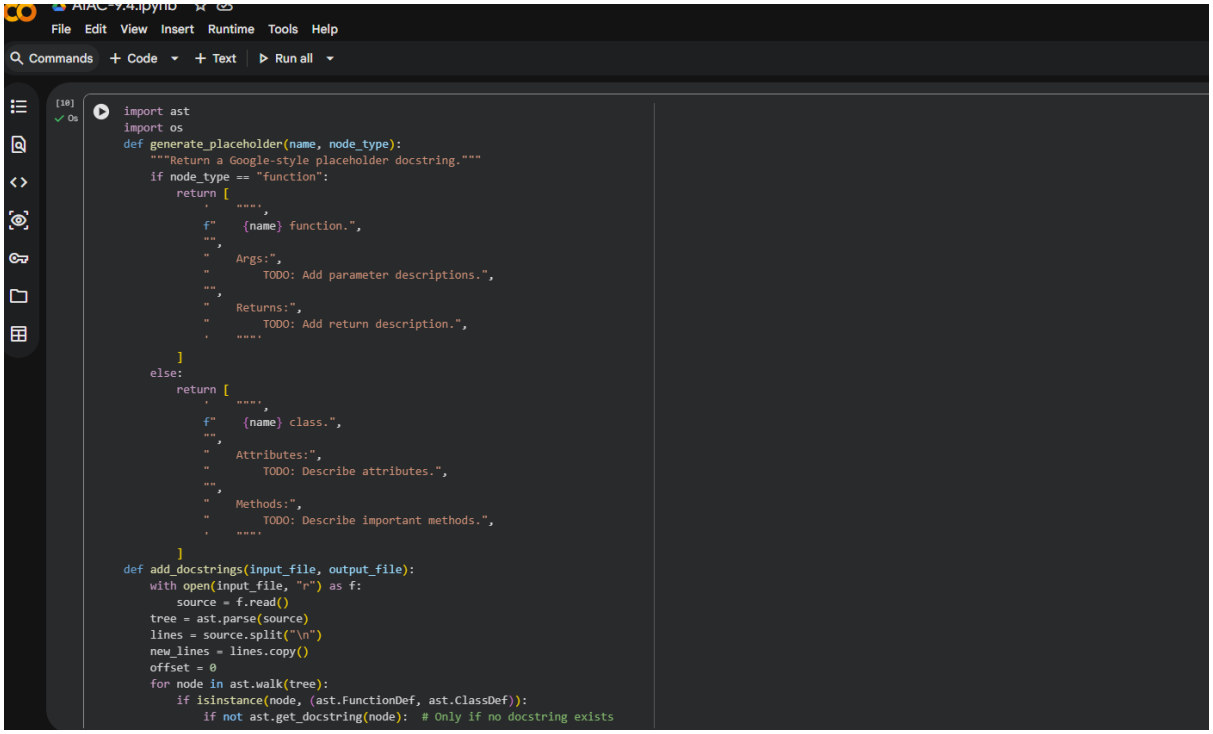
Provide:

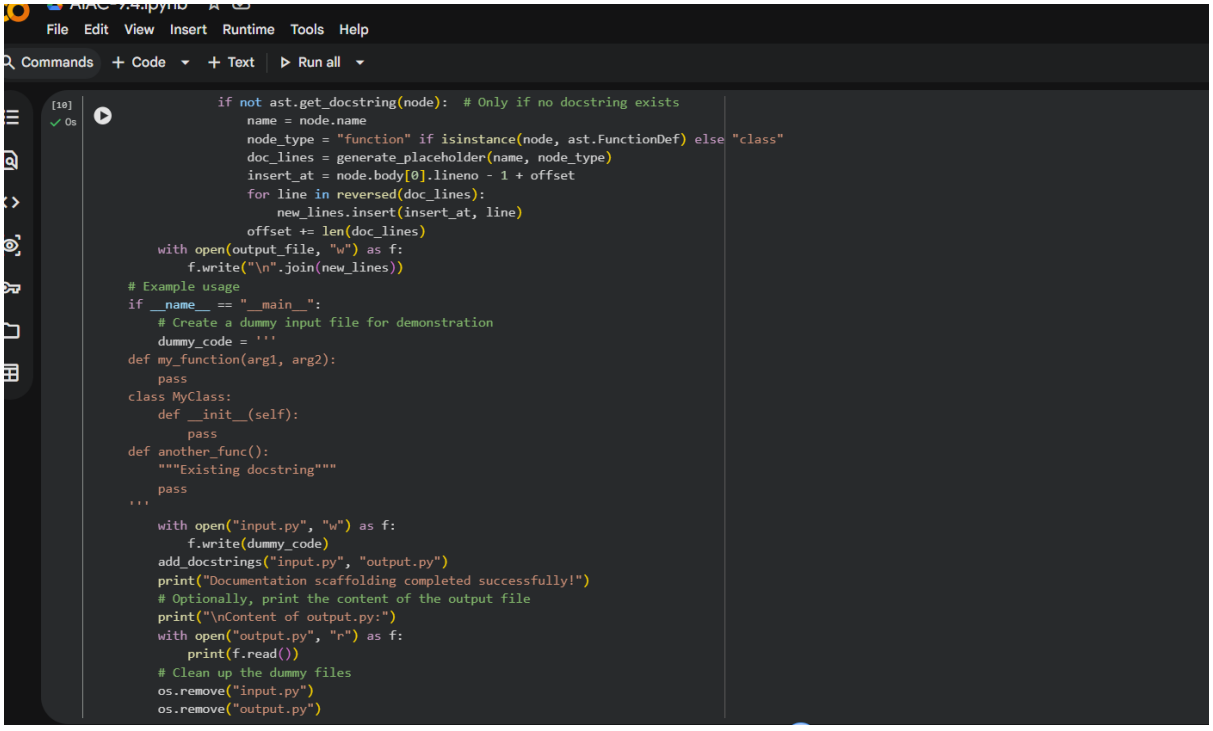
1. A complete working Python script

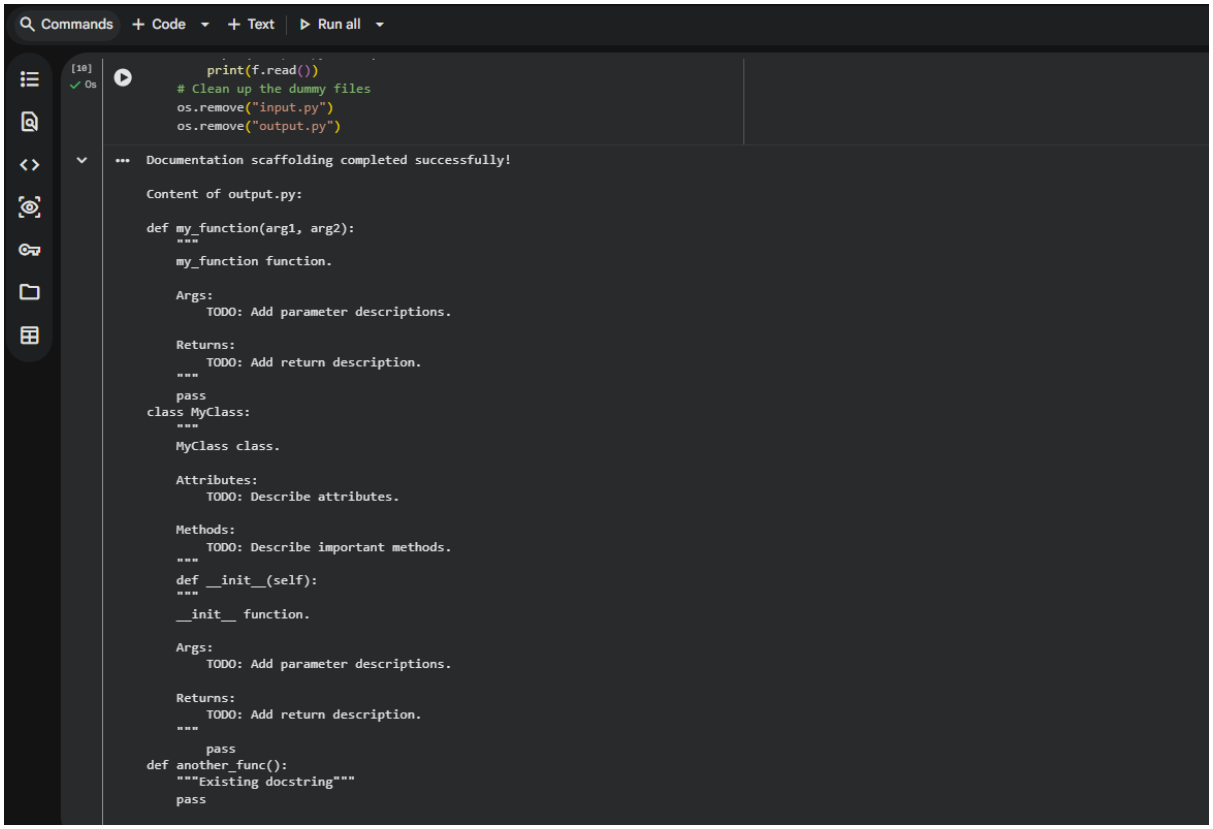
2. Clear explanation of how it works

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
[10]
0s
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
[10]
0s
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")


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