Name:Gaddam Sai

Hallticket:2303A52255

Batch:43

Assginment:9.4

Task:01

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)

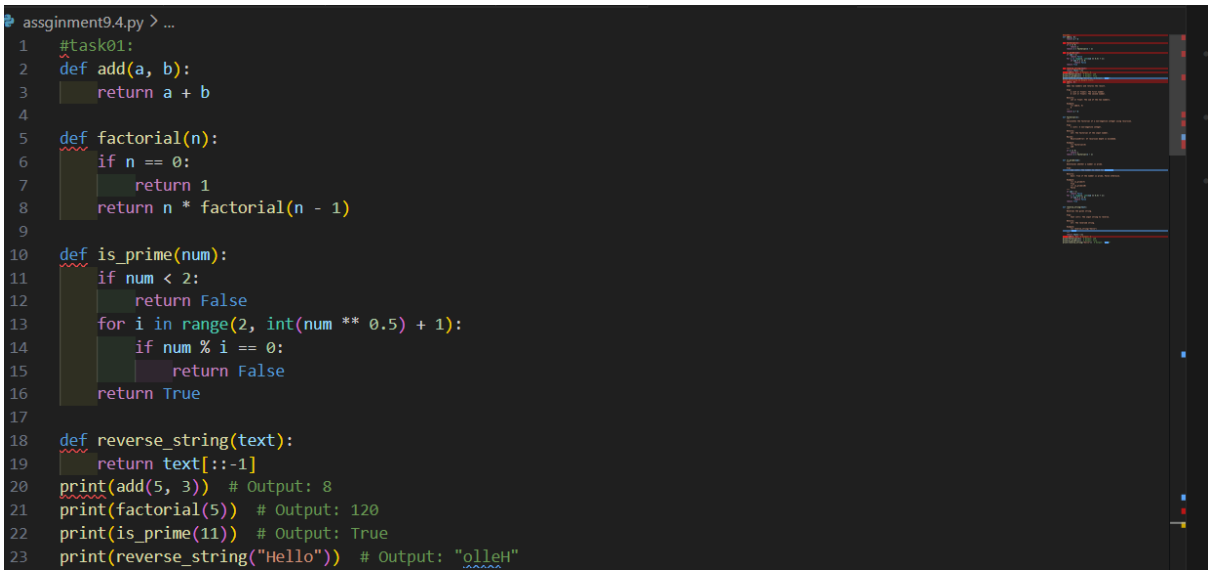
Experiment with different prompting styles (zero-shot or context-based) to observe quality differences.

Expected Outcome

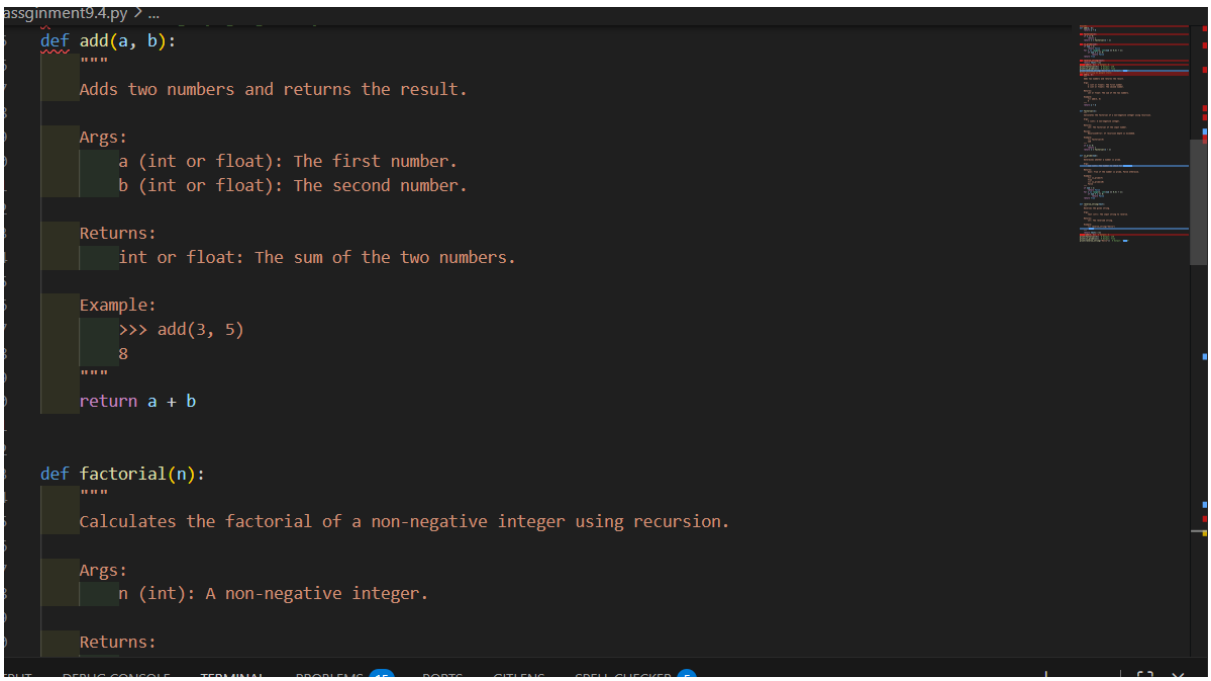
- A Python script with well-structured Google-style docstrings
- Docstrings that clearly explain function behavior and usage
- Improved readability and usability of the codebase

Prompt:give me code for creating multiple functions and also give me the google style documentation in docstring

Code:



```python
#task01:
def add(a, b):
    return a + b

def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            return False
    return True

def reverse_string(text):
    return text[::-1]
print(add(5, 3))  # Output: 8
print(factorial(5))  # Output: 120
print(is_prime(11))  # Output: True
print(reverse_string("Hello"))  # Output: "olleH"
```



```python
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 the two numbers.

    Example:
        >>> add(3, 5)
        8
    """
    return a + b


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

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

    Returns:
```

Output:

```
/assginment9.4.py"
8
120
True
olleH
8
120
True
olleH
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding>
```

Observations:

The addition of Google-style docstrings makes each function's purpose immediately clear.

New developers can understand functionality without reading the implementation.

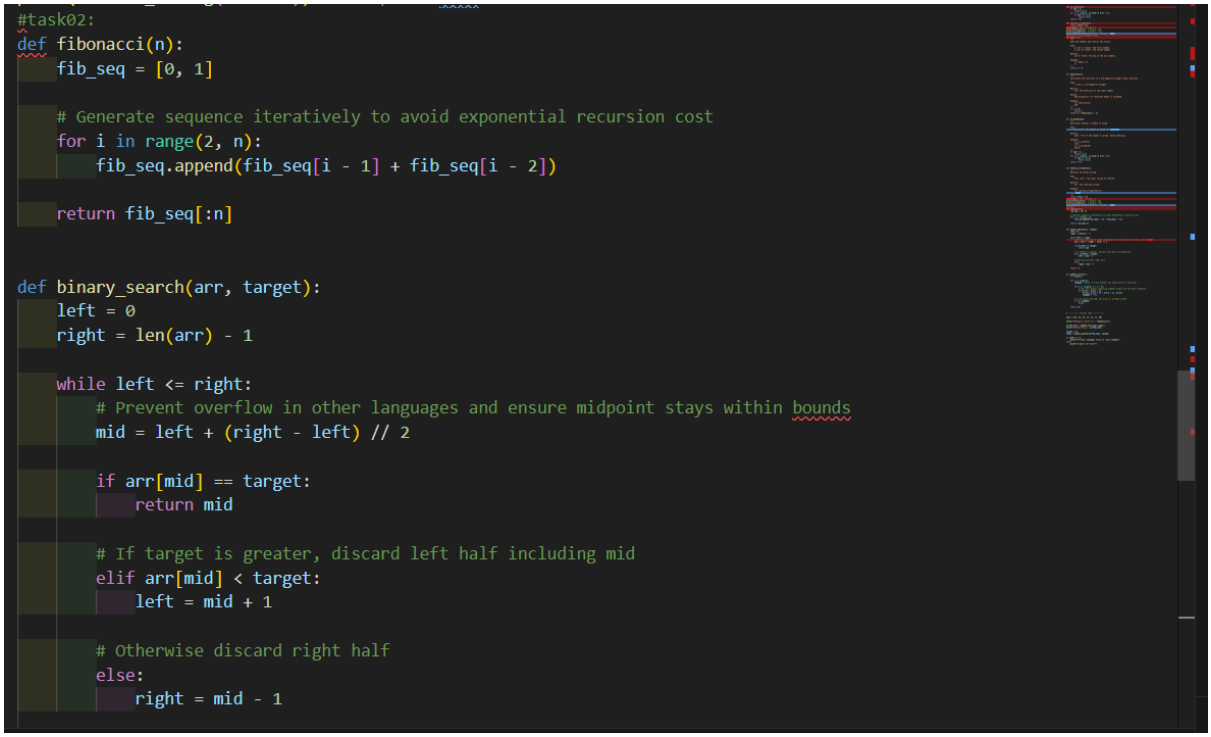
The code becomes self-documenting and easier to maintain

Task:02

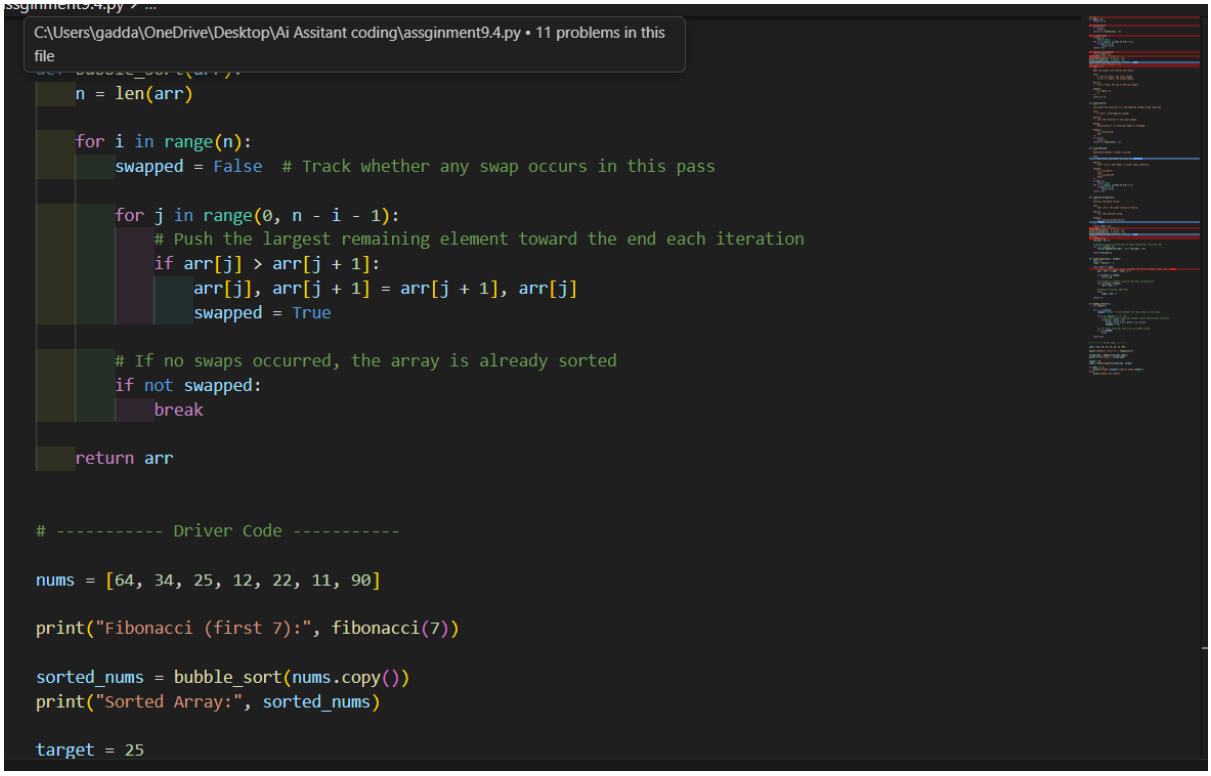
Prompt:

inline comments to the Python code explaining only complex or non-obvious logic (algorithms, loops, conditionals)

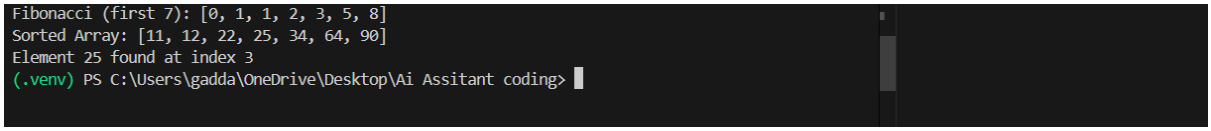
Code:



```python
#task02:
def fibonacci(n):
    fib_seq = [0, 1]

    # Generate sequence iteratively to avoid exponential recursion cost
    for i in range(2, n):
        fib_seq.append(fib_seq[i - 1] + fib_seq[i - 2])

    return fib_seq[:n]


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

    while left <= right:
        # Prevent overflow in other languages and ensure midpoint stays within bounds
        mid = left + (right - left) // 2

        if arr[mid] == target:
            return mid

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

        # Otherwise discard right half
        else:
            right = mid - 1
```



Output:



Observation:

AI added comments mainly around algorithms, nested loops, and tricky conditions.

Difficult parts of the code became easier to follow without tracing execution line-by-line.

Maintainers can quickly grasp the intent behind the logic**.**

Task:03

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:

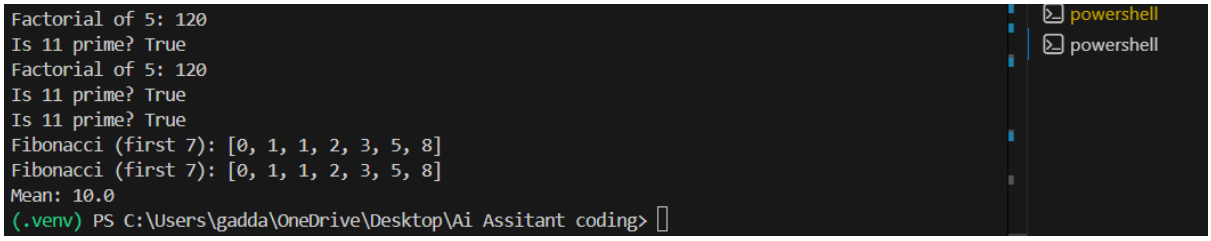
Generate a professional module-level docstring for the following Python file that describes the module's purpose, dependencies, key functions/classes, and provides a brief example of usage, without modifying the code

Code:

```
#task03:
"""
math_utils.py

A utility module that provides commonly used mathematical operations
such as factorial computation, prime number checking, Fibonacci
sequence generation, and basic statistical calculations.

Dependencies:
    - math (standard library)

Key Functions:
    factorial(n): Computes the factorial of a non-negative integer.
    is_prime(num): Determines whether a number is prime.
    fibonacci(n): Generates the first n Fibonacci numbers.
    mean(numbers): Calculates the arithmetic mean of a list of numbers.

Example Usage:
    >>> import math_utils as mu
    >>> mu.factorial(5)
    120
    >>> mu.is_prime(7)
    True
    >>> mu.fibonacci(6)
    [0, 1, 1, 2, 3, 5]
    >>> mu.mean([10, 20, 30])
    20.0

This module is intended for educational purposes and small-scale
applications requiring basic mathematical utilities.
"""
```

```
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(math.sqrt(num)) + 1):
        if num % i == 0:
            return False
    return True


def fibonacci(n):
    if n <= 0:
        return []
    seq = [0, 1]
    for i in range(2, n):
        seq.append(seq[i - 1] + seq[i - 2])
    return seq[:n]


def mean(numbers):
    if not numbers:
        raise ValueError("List cannot be empty")
    return sum(numbers) / len(numbers)


# ----------- Example Execution -----------

if __name__ == "__main__":
    print("Factorial of 5:", factorial(5))
    print("Is 11 prime?", is_prime(11))
    print("Fibonacci (first 7):", fibonacci(7))
    print("Mean:", mean([5, 10, 15]))
```

Output:



Observation:

the generated documentation provides an immediate overview of the module, making it easier for developers to understand structure and functionality.

It improves usability, maintainability, and code sharing, though accuracy depends on the clarity of the prompt and should be reviewed.

Task:04

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

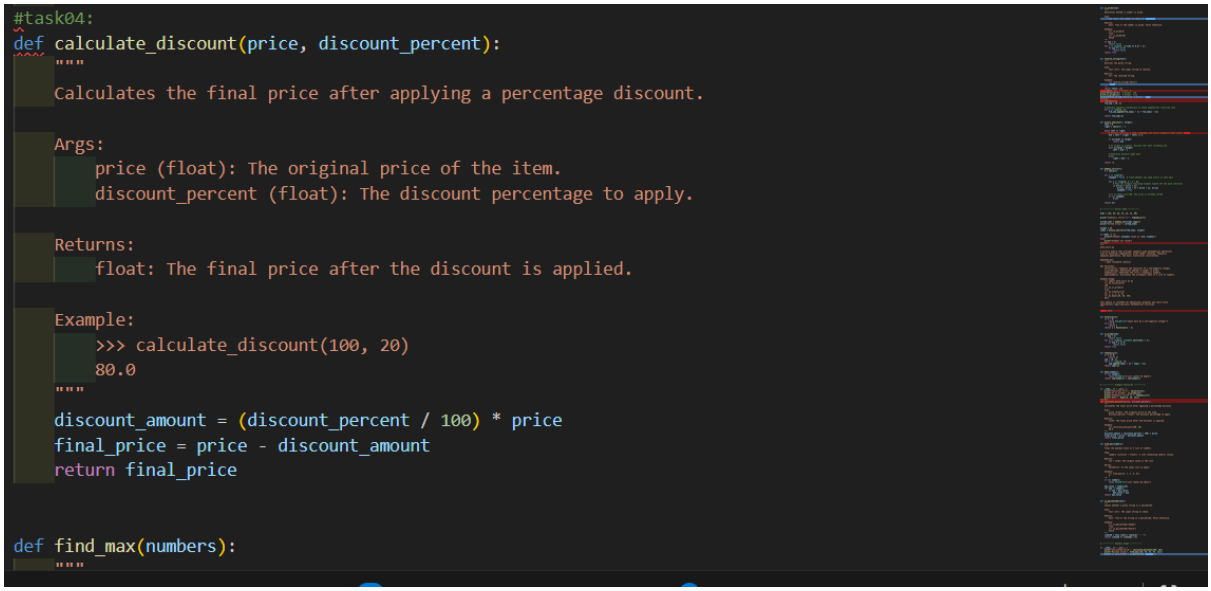
• Reduced clutter inside function bodies

• Improved consistency across the codebase

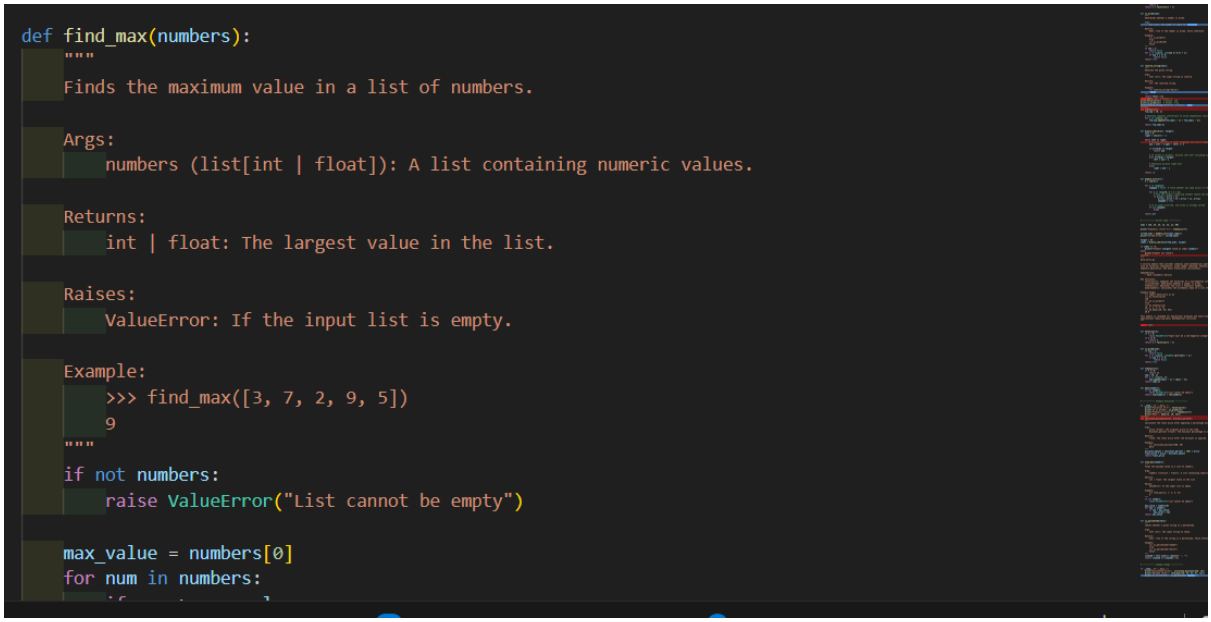
Prompt:

Convert the detailed inline comments inside each function into structured Google-style docstrings, preserving their original meaning, removing redundant comments from the function body, and keeping the code functionality unchanged.

Code:



```
#task04:
def calculate_discount(price, discount_percent):
    """
    Calculates the final price after applying a percentage discount.

    Args:
        price (float): The original price of the item.
        discount_percent (float): The discount percentage to apply.

    Returns:
        float: The final price after the discount is applied.

    Example:
        >>> calculate_discount(100, 20)
        80.0
    """
    discount_amount = (discount_percent / 100) * price
    final_price = price - discount_amount
    return final_price


def find_max(numbers):
    """
```



```
def find_max(numbers):
    """
    Finds the maximum value in a list of numbers.

    Args:
        numbers (list[int | float]): A list containing numeric values.

    Returns:
        int | float: The largest value in the list.

    Raises:
        ValueError: If the input list is empty.

    Example:
        >>> find_max([3, 7, 2, 9, 5])
        9
    """
    if not numbers:
        raise ValueError("List cannot be empty")

    max_value = numbers[0]
    for num in numbers:
```

```python
def is_palindrome(text):
    Args:
        text (str): The input string to check.

    Returns:
        bool: True if the string is a palindrome, False otherwise.

    Example:
        >>> is_palindrome("madam")
        True
        >>> is_palindrome("hello")
        False
    """
    cleaned = text.lower().replace(" ", "")
    return cleaned == cleaned[::-1]


# ----------- Example Usage -----------

if __name__ == "__main__":
    print("Discounted price:", calculate_discount(250, 10))
    print("Maximum value:", find_max([10, 45, 23, 67, 12]))
    print("Is palindrome?", is_palindrome("Racecar"))
```

PUT DEBUG CONSOLE TERMINAL PROBLEMS 22 PORTS GITLENS SPELL CHECKER 6

Output:

```
Discounted price: 225.0
Maximum value: 67
Is palindrome? True
```

Observation:

AI successfully transformed explanatory comments into clean, standardized docstrings, making functions easier to read without scanning internal logic. The removal of redundant inline comments reduced visual clutter while preserving important documentation. Overall, the codebase became more consistent and maintainable, though results depended on how clearly the original comments described intent.

Task:05

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

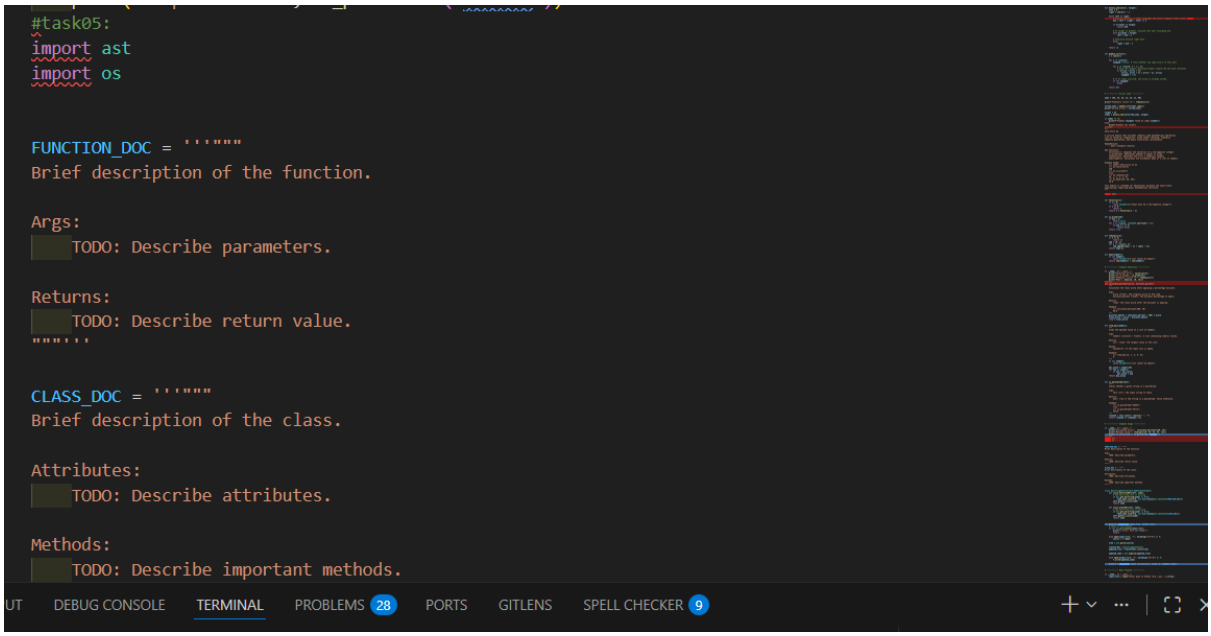
• Inserts placeholder Google-style docstrings for each detected function or class

AI tools may be used to assist in generating or refining this utility.

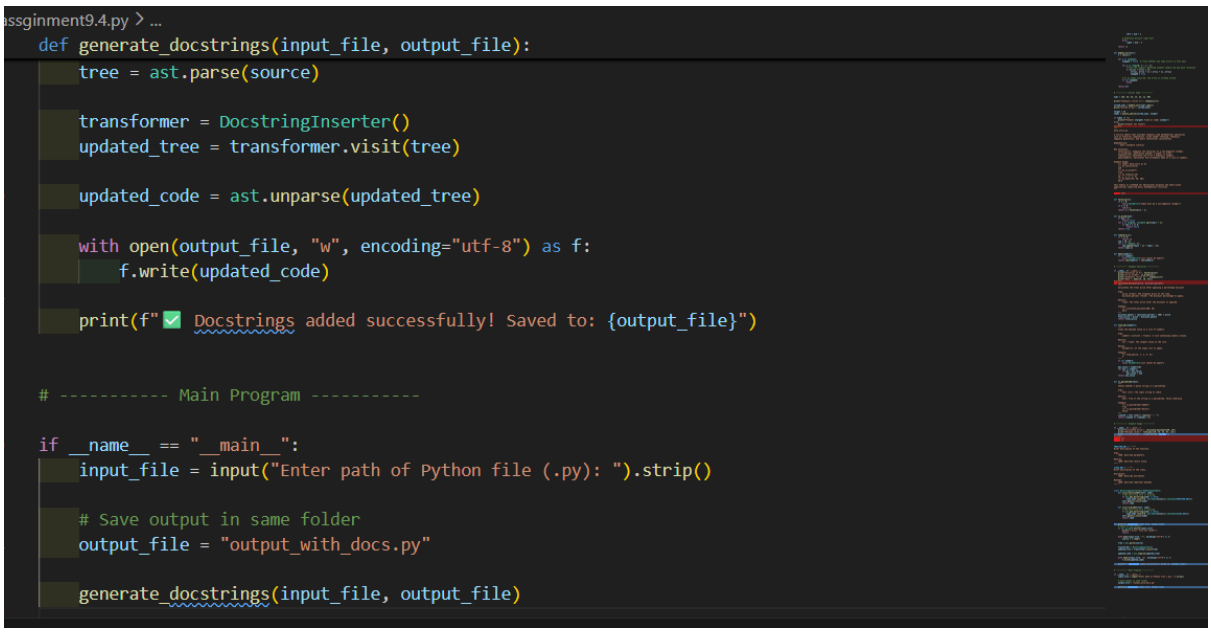
Prompt:

Create a Python utility that reads a .py file, detects all functions and classes, and automatically inserts placeholder Google-style docstrings where missing, without altering existing documentation or code behavior.

Code:

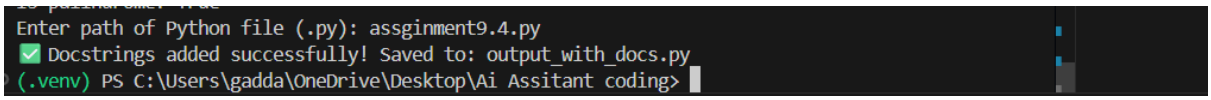


```python
class DocstringInserter(ast.NodeTransformer):
    def visit_FunctionDef(self, node):
        # Add docstring only if missing
        if ast.get_docstring(node) is None:
            node.body.insert(0, ast.Expr(value=ast.Constant(FUNCTION_DOC)))
        self.generic_visit(node)
        return node

    def visit_ClassDef(self, node):
        # Add docstring only if missing
        if ast.get_docstring(node) is None:
            node.body.insert(0, ast.Expr(value=ast.Constant(CLASS_DOC)))
        self.generic_visit(node)
        return node


def generate_docstrings(input_file, output_file):
    # Check if file exists
    if not os.path.exists(input_file):
        print("❌ Error: File not found!")
        return

    with open(input_file, "r", encoding="utf-8") as f:
```



```python
def generate_docstrings(input_file, output_file):
    tree = ast.parse(source)

    transformer = DocstringInserter()
    updated_tree = transformer.visit(tree)

    updated_code = ast.unparse(updated_tree)

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

    print(f"✅ Docstrings added successfully! Saved to: {output_file}")


# ---------- Main Program ----------

if __name__ == "__main__":
    input_file = input("Enter path of Python file (.py): ").strip()

    # Save output in same folder
    output_file = "output_with_docs.py"

    generate_docstrings(input_file, output_file)
```

Output:



```
Enter path of Python file (.py): assginment9.4.py
✅ Docstrings added successfully! Saved to: output_with_docs.py
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding>
```

Observation:

The tool successfully identifies functions and classes using Python's AST, ensuring accurate detection without relying on fragile text parsing.

Placeholder Google-style docstrings provide a structured starting point, enabling developers to fill in documentation quickly and consistently.

The utility improves documentation adoption and code maintainability, though manual editing is still required to replace placeholders with meaningful content.