AI ASSISSTED CODING LAB -9.4

Name:M.Shashidhar Batch : 42 Rollno: 2303A52288

Task 1: Auto-Generating Function Documentation

Scenario

Existing utility functions lack documentation, making onboarding difficult.

Original code

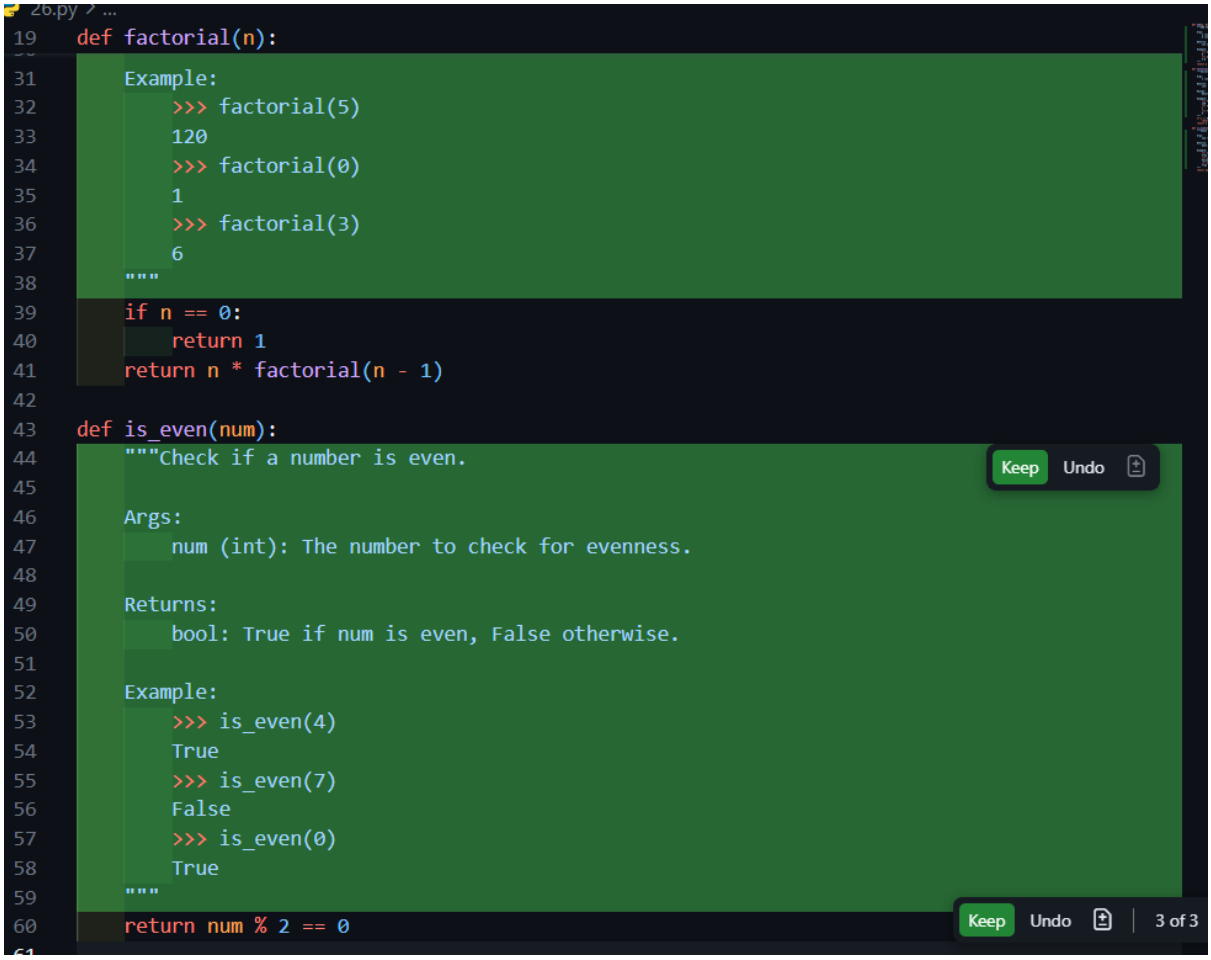
```python
def add(a, b):
    return a + b

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

def is_even(num):
    return num % 2 == 0
```

Prompt Used (Zero-Shot Prompt)

Generate Google-style docstrings for the following Python functions.
Include description, parameters, return values, and example usage.

AI-Generated Code With Google-Style Docstrings

```python
def add(a, b):
    """Add two numbers together.

    Args:
        a (int or float): The first number to add.
        b (int or float): The second number to add.

    Returns:
        int or float: The sum of a and b.

    Example:
        >>> add(5, 3)
        8
        >>> add(2.5, 1.5)
        4.0
    """
    return a + b

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

    Args:
        n (int): A non-negative integer whose factorial is to be computed.

    Returns:
        int: The factorial of n (n!).

    Raises:
        RecursionError: If n is too large and exceeds maximum recursion de
```



Explanation

In this task, AI assistance is used to automatically generate Google-style function docstrings for an existing Python codebase that lacks documentation. The objective is to improve code understandability for new developers.

A zero-shot or context-based prompt is given to the AI without providing examples, asking it to analyze the function logic and generate structured docstrings. The AI inspects the function behavior, identifies parameters, return values, and usage patterns, and produces standardized documentation.

This task demonstrates how AI can quickly transform undocumented code into a readable and professional codebase, reducing onboarding time and improving API clarity without modifying the actual logic**.**

Task 2: Enhancing Readability With Inline Comments

Scenario

Working logic exists but is difficult to understand.

Orginal code

```python
def fibonacci(n):
    result = []
    a, b = 0, 1

    for _ in range(n):
        result.append(a)
        a, b = b, a + b

    return result
# Example usage:
```

Prompt Used (Context-Based Prompt)

Add inline comments only where logic is non-obvious.
Avoid commenting basic Python syntax.

Code with inline comments

```python
def fibonacci(n):
    result = []
    a, b = 0, 1  # Initialize first two Fibonacci numbers

    for _ in range(n):
        result.append(a)
        a, b = b, a + b  # Simultaneously update: a becomes b, b becomes a+b

    return result

```

Explanation

This task focuses on improving code readability by inserting meaningful inline comments only where the logic is complex or non-obvious. The goal is not to explain Python syntax, but to clarify *why* a particular piece of logic exists.

Using AI assistance, the code is analyzed to identify sections such as algorithmic updates, loop behavior, or value transformations that may confuse future maintainers. Comments are added selectively at these points.

This task highlights the importance of clean commenting practices and shows how AI helps prevent over-commenting while still improving maintainability and debugging efficiency.

Task 3: Module-Level Documentation

Scenario

Python module is being prepared for internal reuse or repository upload.

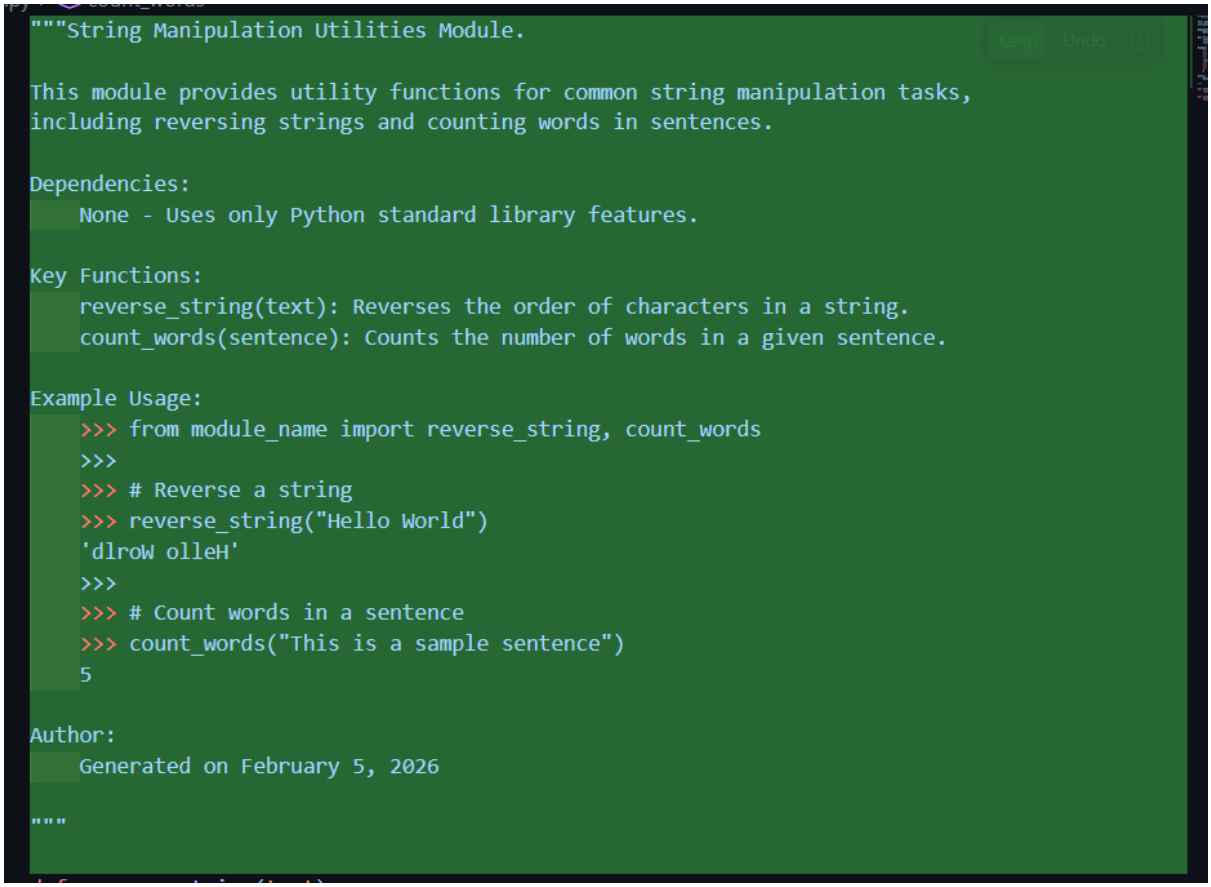
Python module code

```python
28.py > count_words
def reverse_string(text):
    return text[::-1]

def count_words(sentence):
    return len(sentence.split())
```

Prompt Used

Generate a professional module-level docstring explaining purpose, dependencies, key functions, and example usage.

AI-Generated Module-Level Docstring



```
"""String Manipulation Utilities Module.

This module provides utility functions for common string manipulation tasks,
including reversing strings and counting words in sentences.

Dependencies:
    None - Uses only Python standard library features.

Key Functions:
    reverse_string(text): Reverses the order of characters in a string.
    count_words(sentence): Counts the number of words in a given sentence.

Example Usage:
    >>> from module_name import reverse_string, count_words
    >>>
    >>> # Reverse a string
    >>> reverse_string("Hello World")
    'dlroW olleH'
    >>>
    >>> # Count words in a sentence
    >>> count_words("This is a sample sentence")
    5

Author:
    Generated on February 5, 2026

"""
```

Explanation

In this task, AI is used to generate a **module-level docstring** that appears at the top of a Python file. This documentation provides a high-level overview of the module's purpose and structure.

The AI analyzes all functions and imports in the module and generates a professional, multi-line docstring describing:

- The purpose of the module
- Dependencies used
- Key functions or classes
- Example usage

This task ensures that anyone opening the file can immediately understand what the module does, making it suitable for real-world projects and shared repositories.

Task 4: Converting Developer Comments into Docstrings

Scenario

Legacy code contains long comments instead of docstrings.

Original code

```
def divide(a, b):
    # This function divides two numbers
    # a is numerator
    # b is denominator
    # returns division result
    if b == 0:
        return None
    return a / b
```

Prompt Used

Convert inline developer comments into a structured Google-style docstring
and remove redundant comments.

Converted code

```
def divide(a, b):
    """Divide two numbers.

    Args:
        a (int or float): The numerator (dividend).
        b (int or float): The denominator (divisor).

    Returns:
        float or None: The division result (a/b), or None if b is 0.

    Example:
        >>> divide(10, 2)
        5.0
        >>> divide(7, 2)
        3.5
        >>> divide(5, 0)
        None
    """
    if b == 0:
        return None
    return a / b
```

Explanation

This task addresses legacy code where developers have written long explanatory comments inside functions instead of proper docstrings. Such comments increase clutter and break documentation standards.

AI assistance is used to extract the intent of these inline comments and convert them into structured Google-style or NumPy-style docstrings. After conversion, redundant inline comments are removed, keeping the function body clean.

This task demonstrates how AI helps in documentation standardization, improves consistency across the codebase, and aligns old projects with modern Python best practices.

Task 5: Building a Mini Automatic Documentation Generator

Scenario

Your team wants a simple internal tool that helps developers start

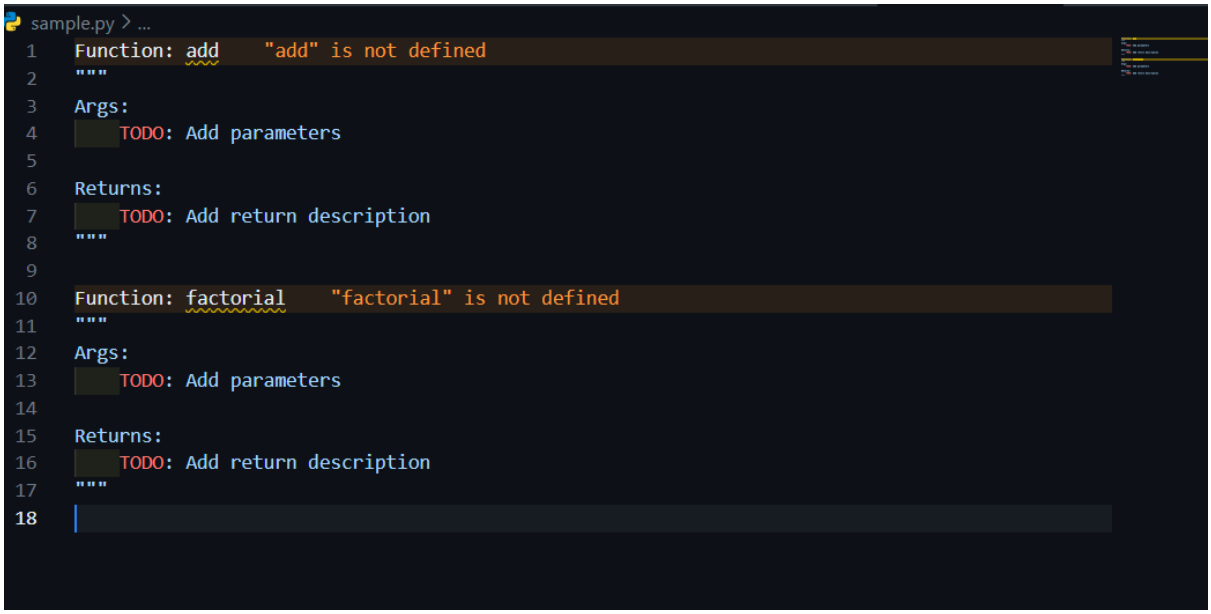
documenting new Python files quickly, without writing documentation

from scratch.

Python utility script

```
import ast

def generate_docstrings(file_path):
    with open(file_path, "r") as file:
        tree = ast.parse(file.read())

    for node in tree.body:
        if isinstance(node, ast.FunctionDef):
            print(f'Function: {node.name}')
            print('''"""
Args:
    TODO: Add parameters

Returns:
    TODO: Add return description
"""
''')
        elif isinstance(node, ast.ClassDef):
            print(f'Class: {node.name}')
            print('''"""
Description:
    TODO: Class description
"""
''')

generate_docstrings("sample.py")
```

Output Example



Explanation

In this task, a small Python utility is designed to automatically **detect functions and classes** in a given .py file and insert placeholder docstrings.

The script uses Python's ast (Abstract Syntax Tree) module to parse the source file and identify code structures without executing it. For each detected function or class, the tool generates a basic **Google-style documentation template**.

This task shows how AI and automation can assist developers by providing **documentation scaffolding**, allowing them to focus on logic while maintaining documentation discipline from the start.