AI Assisted Coding
Lab Assignment - 9.4

Name: K.Bhavya Sri
Roll No: 2303A51863
Batch: 13

Lab 9 – Documentation Generation: Automatic Documentation and Code Comments

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.

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

Without Docstrings

One-Shot Code:

LAB-09 > 1.py > ...

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

    return a + b
def subtract(a, b):

    return a - b
def multiply(a, b):

    return a * b
def divide(a, b):

    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b
if __name__ == "__main__":
    print(add(5, 3))        # Output: 8
    print(subtract(5, 3))   # Output: 2
    print(multiply(5, 3))   # Output: 15
    print(divide(5, 3))     # Output: 1.6666666666666667
    # Uncomment the below line to see the error handling
    # print(divide(5, 0))   # Raises ValueError

```

With Docstrings:

LAB-09 > 1.py > divide

```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 a and b.
    Example:
        >>> add(5, 3)
        8
    """
    return a + b
def subtract(a, b):
    """
    Subtracts the second number from the first number.
    Args:
        a (int or float): The number from which subtraction is performed.
        b (int or float): The number to subtract.
    Returns:
        int or float: The result of a minus b.
    Example:
        >>> subtract(5, 3)
        2
    """
    return a - b
def multiply(a, b):
    """
    Multiplies two numbers and returns the product.
    Args:
        a (int or float): The first number.
        b (int or float): The second number.
    Returns:
        int or float: The product of a and b.
    Example:
        >>> multiply(5, 3)
        15
    """
    return a * b
def divide(a, b):
    """
    Divides the first number by the second number.
    Args:
        a (int or float): The numerator.
        b (int or float): The denominator.
    Returns:
        float: The result of division.
    Raises:
        ValueError: If b is zero.
    Example:
        >>> divide(5, 3)
        1.6666666666666667
    """
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b
if __name__ == "__main__":
    print(add(5, 3))        # Output: 8
    print(subtract(5, 3))   # Output: 2
    print(multiply(5, 3))   # Output: 15
    print(divide(5, 3))     # Output: 1.6666666666666667
    # Uncomment the below line to see the error handling
    # print(divide(5, 0))   # Raises ValueError
```

Output:

```
PS C:\AI Assisted Coding\LAB> & "C:\Program Files\Python311\python.exe" "c:/AI Assisted Coding/LAB/LAB-09/1.py"
8
2
15
2
15
1.6666666666666667
PS C:\AI Assisted Coding\LAB>
```

Zero-Shot Prompt:

Generate Google-style function docstrings, where each docstring should include: """ 1. A brief description of the functions 2. parameters with data types 3. return values 4. At least one example usage (if applicable). 5. Experiment with different prompting styles, zero-shot or context-based, to observe quality differences. And call the functions at last.

Without Docstrings :

11.py > ...

```python
#Generate a Python program that performs basic arithmetic operations (addition, subtraction, multiplication, division)
#on two numbers input by the user. The program should handle division by zero gracefully. zero shot.

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

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b

if __name__ == "__main__":
    a = float(input("Enter first number: "))
    b = float(input("Enter second number: "))

    print("Addition:", add(a, b))
    print("Subtraction:", subtract(a, b))
    print("Multiplication:", multiply(a, b))

    try:
        print("Division:", divide(a, b))
    except ValueError as e:
        print("Error:", e)

```

With Docstrings :

```python
def add(a, b):
    return 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."""
def subtract(a, b):
    return a - b
"""Subtracts the second number from the first number.
Args:
    a (int or float): The number from which subtraction is performed.
    b (int or float): The number to subtract.
    Returns:
    int or float: The result of a minus b."""
def multiply(a, b):
    return a * b
"""Multiplies two numbers and returns the product.
Args:
    a (int or float): The first number.
    b (int or float): The second number.
    Returns:
    int or float: The product of a and b."""
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b
"""Divides the first number by the second number.
Args:
    a (int or float): The numerator.
    b (int or float): The denominator.
    Returns:
    float: The result of division.
    Raises:
    ValueError: If b is zero."""
if __name__ == "__main__":
    num1 = float(input("Enter the first number: "))
    num2 = float(input("Enter the second number: "))
    print(f"Addition: {add(num1, num2)}")
    print(f"Subtraction: {subtract(num1, num2)}")
    print(f"Multiplication: {multiply(num1, num2)}")
    try:
        print(f"Division: {divide(num1, num2)}")
    except ValueError as e:
        print(e)
```

Output:

```
PS C:\AI Assisted Coding\LAB> & "C:\Program Files\Python311\python.exe" "c:/AI Assisted Coding/LAB/11.py"
Enter the first number: 9
Enter the second number: 7
Addition: 16.0
Subtraction: 2.0
Multiplication: 63.0
Division: 1.2857142857142858
PS C:\AI Assisted Coding\LAB>
```

Observation:

Adding proper Google-style docstrings significantly improves code readability and usability.They help new team members quickly understand the function purpose,inputs,outputs,and possible errors without analyzing the entire code logic.This makes the codebase more maintainable,professional and suitable for collaborative development.

Task 2: Enhancing Readability Through AI-Generated Inline Comments

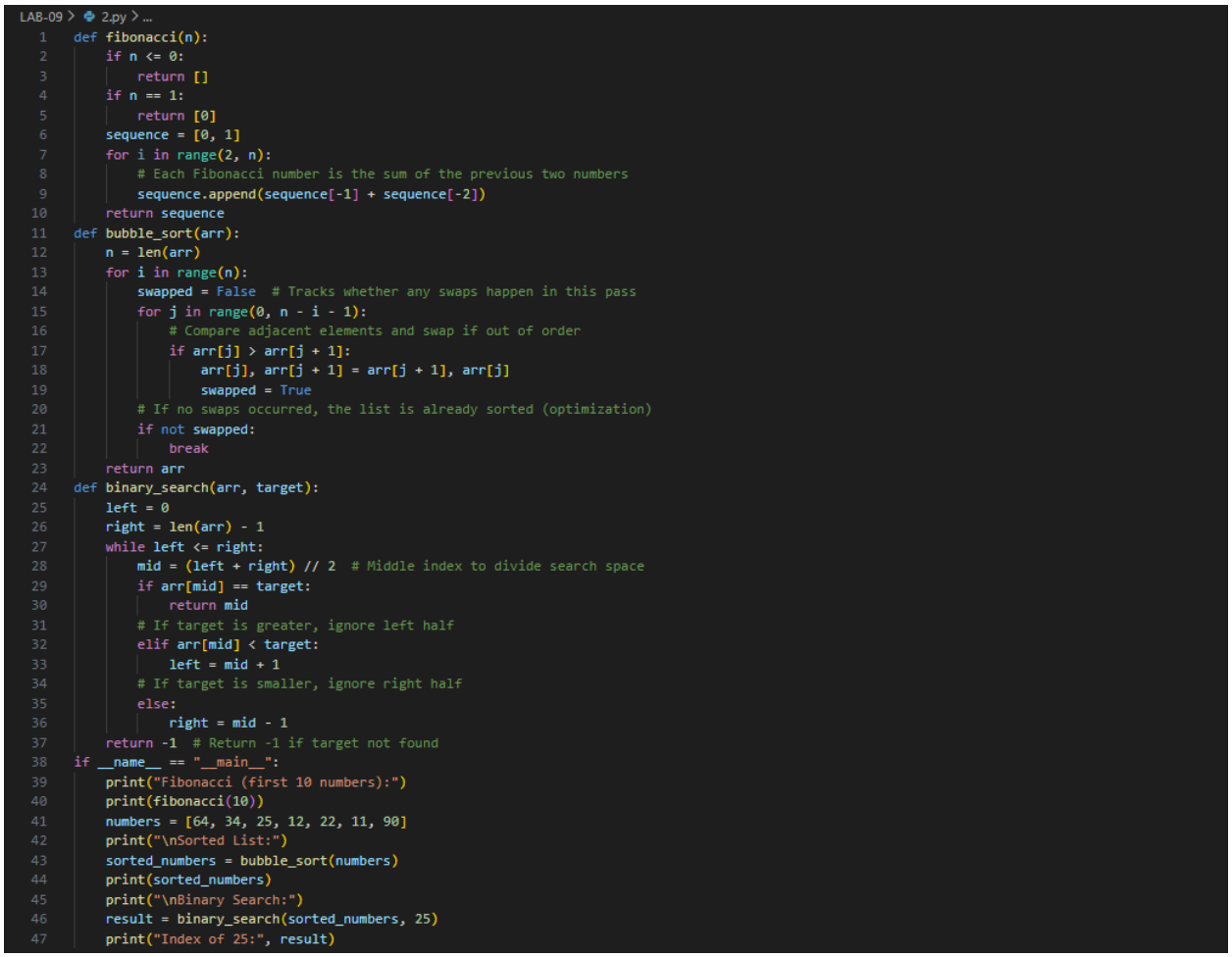
Scenario

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.

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

Code With Inline Comments :



```
LAB-09 > 2.py > ...
def fibonacci(n):
    if n <= 0:
        return []
    if n == 1:
        return [0]
    sequence = [0, 1]
    for i in range(2, n):
        # Each Fibonacci number is the sum of the previous two numbers
        sequence.append(sequence[-1] + sequence[-2])
    return sequence
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False  # Tracks whether any swaps happen in this pass
        for j in range(0, n - i - 1):
            # Compare adjacent elements and swap if out of order
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # If no swaps occurred, the list is already sorted (optimization)
        if not swapped:
            break
    return arr
def binary_search(arr, target):
    left = 0
    right = len(arr) - 1
    while left <= right:
        mid = (left + right) // 2  # Middle index to divide search space
        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
    return -1  # Return -1 if target not found
if __name__ == "__main__":
    print("Fibonacci (first 10 numbers):")
    print(fibonacci(10))
    numbers = [64, 34, 25, 12, 22, 11, 90]
    print("\nSorted List:")
    sorted_numbers = bubble_sort(numbers)
    print(sorted_numbers)
    print("\nBinary Search:")
    result = binary_search(sorted_numbers, 25)
    print("Index of 25:", result)
```

Output :

```
PS C:\AI Assisted Coding\LAB> & "C:\Program Files\Python311\python.exe" "c:/AI Assisted Coding/LAB/LAB-09/2.py"
Fibonacci (first 10 numbers):
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Sorted List:
[11, 12, 22, 25, 34, 64, 90]

Binary Search:
Index of 25: 3
PS C:\AI Assisted Coding\LAB>
```

Table :

Requirement	How It Is Achieved
Insert inline comments	Added only inside logic sections
Avoid trivial comments	No comments for obvious syntax
Explain why, not what	Comments describe purpose & reasoning
Improve readability	Makes algorithms easier to understand
Avoid clutter	Minimal, meaningful comments only

Observation:

In this task,AI was used to improve the readability of a python program containing loops,conditional logic,and algorithms such as Fibonacci,Bubble sort and Binary search.The main goal was to insert inline comments only where the logic was complex or non-obvious.

Instead of explaining basic python syntax,the comments focus on why the logic is written in a certain way,such as why a swap flag is used in Bubble sort,why the search space is used in Binary search,and how Fibonacci values are generated from previous elements.

As a result,the program becomes easier to understand,maintain,debug,and extend in a shared codebase.

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.

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.

Code with Docstring :

```python
#Generate a professional module-level docstring for the given Python file. The docstring should clearly describe the module's purpose, list any dependencies,
#summarize key functions or classes, and include a short example of how to use the module.
"""
Module: calculator_module
This module provides basic arithmetic operations including addition,
subtraction, multiplication, and division. It is designed for simple
mathematical computations in educational, internal, or lightweight
applications.
Dependencies:
    - Python 3.x
    - No external libraries required
Functions:
    add(a, b):
        Returns the sum of two numbers.
    subtract(a, b):
        Returns the difference between two numbers.
    multiply(a, b):
        Returns the product of two numbers.
    divide(a, b):
        Returns the quotient of two numbers.
        Raises a ValueError if division by zero is attempted.
Example Usage:
    from calculator_module import add, subtract, multiply, divide
    result = add(10, 5)
    print(result)  # Output: 15
"""
def add(a, b):
    """Return the sum of a and b."""
    return a + b
def subtract(a, b):
    """Return the difference of a and b."""
    return a - b
def multiply(a, b):
    """Return the product of a and b."""
    return a * b
def divide(a, b):
    """
    Return the quotient of a and b.
    Raises:
        ValueError: If b is zero.
    """
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b
```

```python
if __name__ == "__main__":
    # Example execution when run directly
    print("Addition:", add(5, 3))
    print("Subtraction:", subtract(5, 3))
    print("Multiplication:", multiply(5, 3))
    try:
        print("Division:", divide(5, 3))
        print("Division by Zero:", divide(5, 0))
    except ValueError as e:
        print("Error:", e)
```

Output :

```
PS C:\AI Assisted Coding\LAB> & "C:\Program Files\Python314\python.exe" "c:/AI Assisted Coding/LAB/LAB-09/3.py"
Addition: 8
Subtraction: 2
Multiplication: 15
Division: 1.6666666666666667
Error: Cannot divide by zero.
PS C:\AI Assisted Coding\LAB>
```

Step 1: The module-level docstring is placed at the top of the file so anyone opening it immediately understands what the module does.

Step 2: It clearly explains the purpose (basic arithmetic operations) and lists dependencies (Python 3.x, no external libraries).

Step 3: It summarises key functions and provides a short usage example, making the module easy to understand, use, and maintain in a shared codebase.

Observation:

The improved module-level docstring provides a structured, professional overview that makes the module easier to understand, maintain, and reuse in a shared codebase. It enhances readability and ensures that developers can quickly grasp the module's purpose and functionality without reviewing the entire implementation.

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.

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:

Convert the explanatory inline comments into a structured Google-style docstring placed at the top of each function. Preserve the original meaning and intent of the comments.

Include:

A brief description, Args section with parameter types, Returns section with return type, Any important notes from the original comments, Remove redundant explanatory inline comments after conversion. Keep only necessary inline comments that clarify complex logic (if needed) and maintain professional formatting and consistency.

```python
# This function calculates the nth Fibonacci number using iteration.
# Instead of recursion, we use a loop to avoid excessive memory usage.
# If n is less than or equal to 0, we return 0 since Fibonacci is undefined there.
# If n equals 1, we return 1.
# For all other cases, we build the sequence step by step.
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    prev, curr = 0, 1
    for _ in range(2, n + 1):
        temp = prev + curr
        prev = curr
        curr = temp
    return curr
# This function performs binary search on a sorted list.
# It repeatedly divides the search interval in half.
# If the value of the search key is less than the item in the middle,
# it narrows the interval to the lower half.
# Otherwise, it narrows it to the upper half.
# If the target is found, return its index.
# If not found, return -1.
def binary_search(arr, target):
    left = 0
    right = 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
# This function calculates total price including tax.
# It assumes each item in the list is a dictionary with a 'price' key.
# First, we calculate subtotal by summing all prices.
# Then we multiply subtotal by (1 + tax rate).
# Finally, we round to 2 decimal places.
```

```python
def calculate_total_price(items, tax_rate=0.1):
    subtotal = 0
    for item in items:
        subtotal += item['price']
    total = subtotal * (1 + tax_rate)
    return round(total, 2)
# This function validates email format.
# It checks whether '@' exists.
# Then it checks whether there is at least one '.' after '@'.
# This is a basic validation and does not follow full email RFC rules.
def validate_email(email):
    if "@" not in email:
        return False
    domain_part = email.split("@")[-1]
    if "." not in domain_part:
        return False
    return True
if __name__ == "__main__":
    print("Fibonacci(7):", fibonacci(7))
    numbers = [1, 3, 5, 7, 9, 11]
    print("Binary Search (7):", binary_search(numbers, 7))
    items = [{"price": 10.0}, {"price": 20.0}]
    print("Total Price:", calculate_total_price(items))
    print("Valid Email:", validate_email("user@example.com"))
    print("Invalid Email:", validate_email("invalid.email"))
```

Observation:

The refactoring process does not change functional behavior, so program output remains the same. The improvement is structural and documentation-focused rather than execution-focused.

Task 5: Building a Mini Automatic Documentation Generator

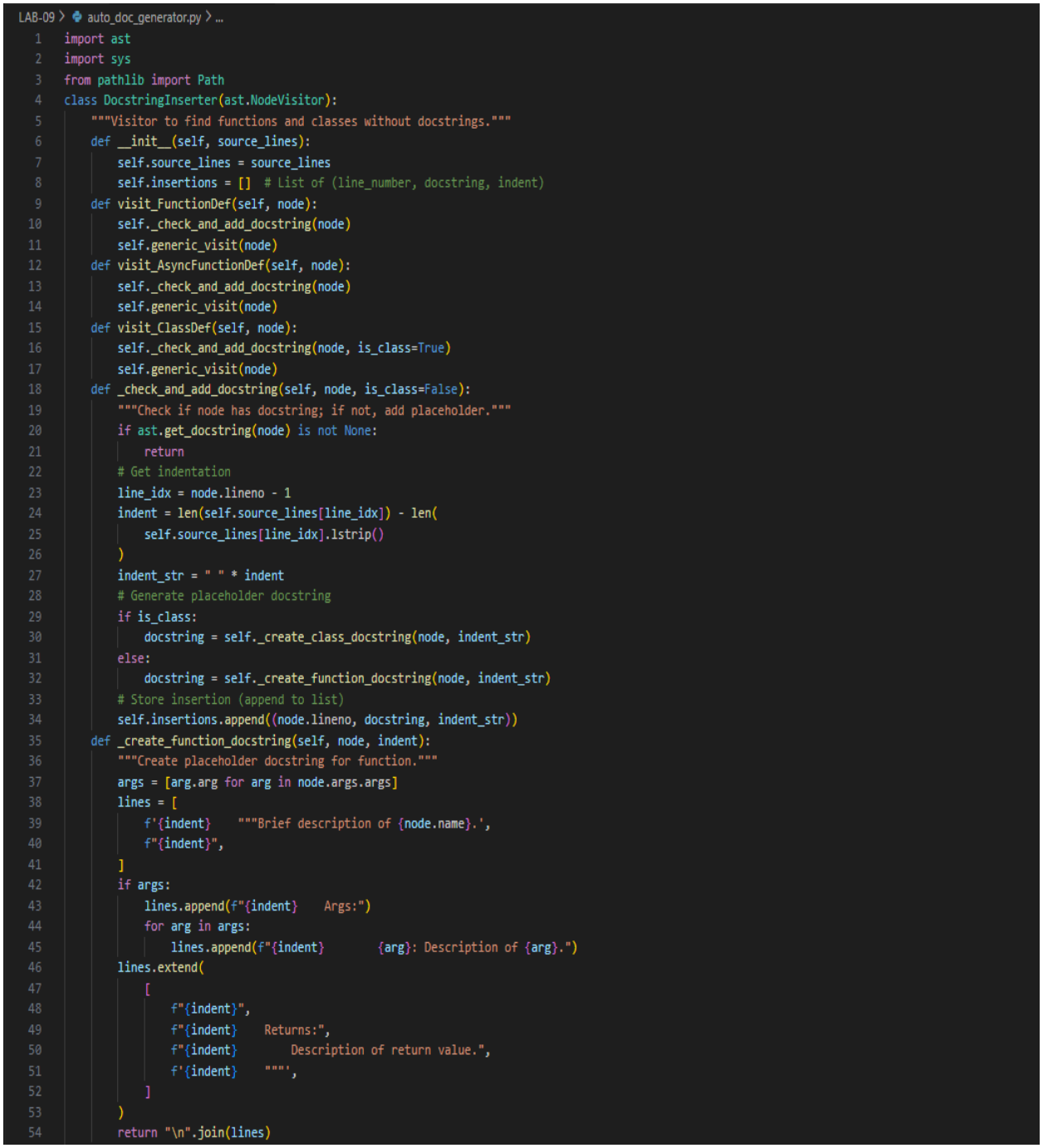
Scenario

Your team wants a simple internal tool that helps developers start documenting new Python files quickly, without writing documentation from scratch.

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:
Generate a Python script that reads a .py file, detects all functions and classes, and automatically inserts placeholder Google-style docstrings for them if they don't already have one. The script should overwrite the file with the updated content. Keep it simple and use built-in libraries like ast.



```python
    def _create_class_docstring(self, node, indent):
        """Create placeholder docstring for class."""
        lines = [
            f'{indent}    """Brief description of {node.name}.',
            f"{indent}",
            f"{indent}    Attributes:",
            f"{indent}        attribute_name: Description of attribute.",
            f"{indent}",
            f"{indent}    Methods:",
            f"{indent}        method_name: Description of method.",
            f'{indent}    """',
        ]
        return "\n".join(lines)
def process_file(filepath):
    """Read, parse, and insert docstrings into Python file."""
    path = Path(filepath)
    if not path.exists():
        print(f"Error: File '{filepath}' not found.")
        sys.exit(1)
    if not filepath.endswith(".py"):
        print("Error: File must be a .py file.")
        sys.exit(1)
    # Read file
    with open(path, "r", encoding="utf-8") as f:
        content = f.read()
        lines = content.splitlines(keepends=True)
    # Parse AST
    try:
        tree = ast.parse(content)
    except SyntaxError as e:
        print(f"Error: Unable to parse file. {e}")
        sys.exit(1)
    # Find missing docstrings
    visitor = DocstringInserter(content.splitlines())
    visitor.visit(tree)
    if not visitor.insertions:
        print("No docstrings needed to be added.")
        return
    # Insert docstrings (reverse order to maintain line numbers)
    for line_num, docstring, _ in sorted(visitor.insertions, reverse=True):
        insertion_point = line_num
        lines.insert(insertion_point, docstring + "\n")
    # Write back to file
    with open(path, "w", encoding="utf-8") as f:
        f.writelines(lines)
    count = len(visitor.insertions)
    print(f"Success! Added {count} docstring(s) to '{filepath}'.")
if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python auto_doc_generator.py <filepath.py>")
        sys.exit(1)
    process_file(sys.argv[1])
```

```
PS C:\AI Assisted Coding\LAB\LAB-09> python auto_doc_generator.py 1.py
>>
No docstrings needed to be added.
PS C:\AI Assisted Coding\LAB\LAB-09>
```

Observation:

These tasks demonstrated how AI can assist in improving code readability and documentation by generating inline comments, structured docstrings, and documentation scaffolding automatically. It reduces manual effort and increases consistency across projects. However, human review is still essential to ensure accuracy and maintain quality standards.