Assignment 9.4

Name: Zobiya Fatima
Batch No: 14
Hall Ticket No: 2303A51879

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.

Task Description

You are given a Python script containing multiple functions without any docstrings.

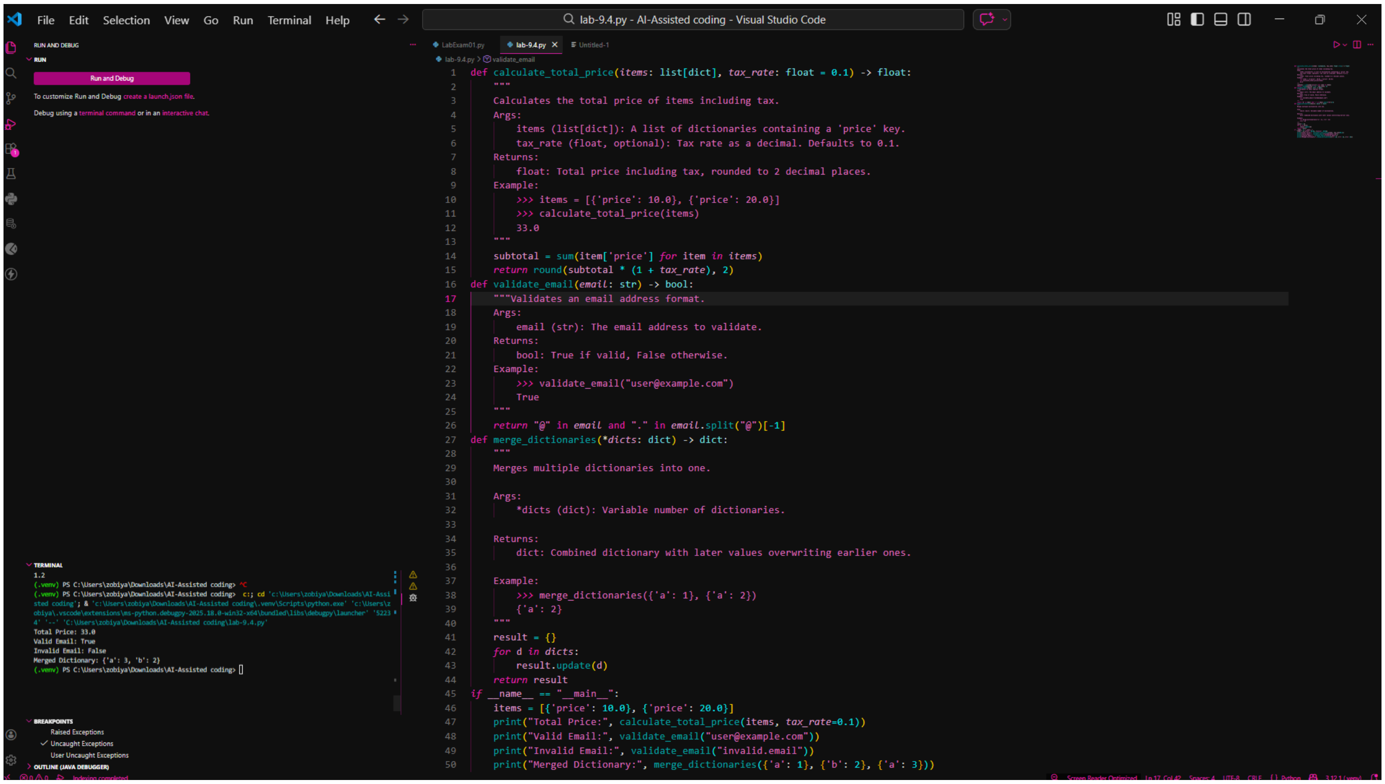
Using an AI-assisted coding tool:

- 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.

Zero-Shot Prompt:

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



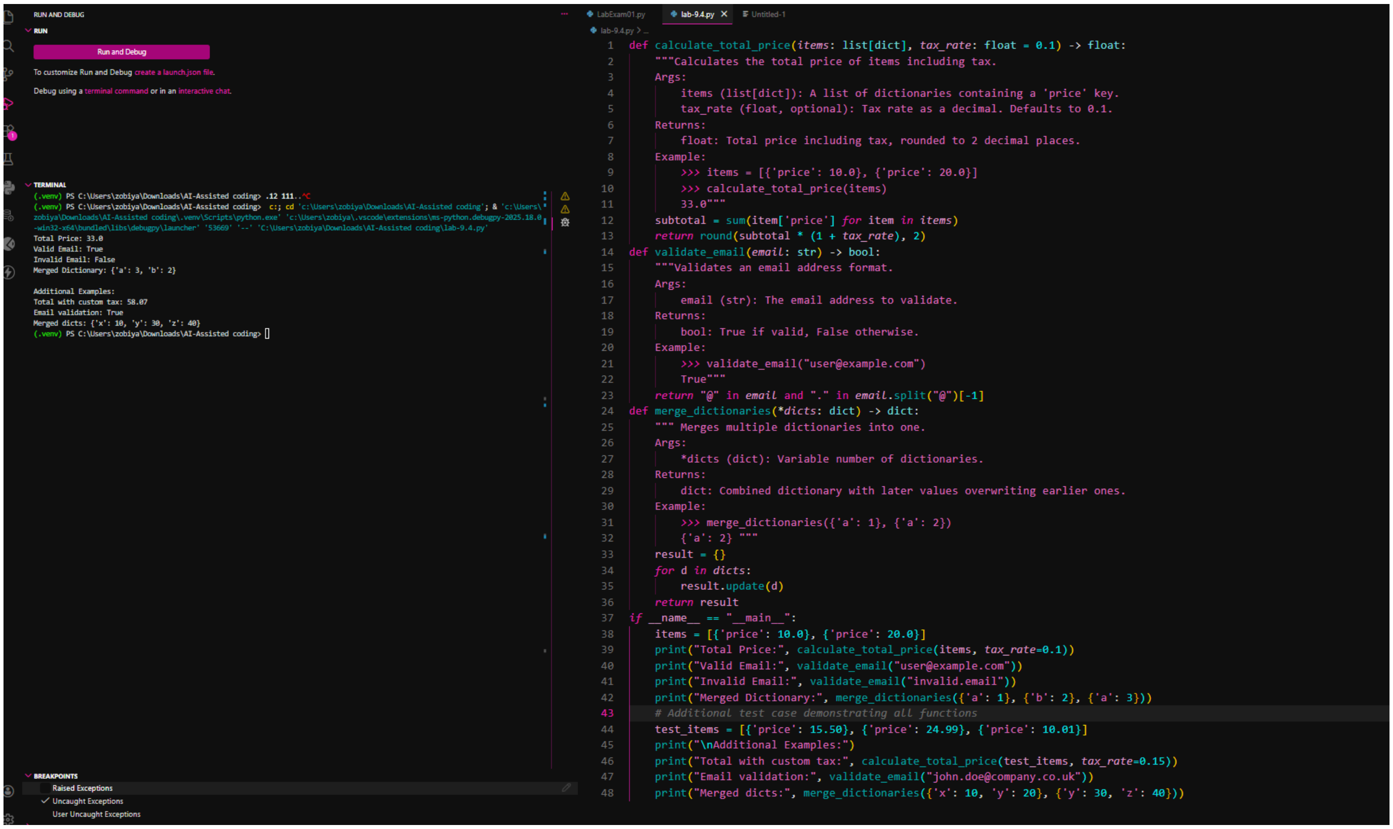
To customize Run and Debug create a launch.json file.

Debug using a terminal command or in an interactive chat.

```python
def calculate_total_price(items: list[dict], tax_rate: float = 0.1) -> float:
    """

    Calculates the total price of items including tax.
    Args:
        items (list[dict]): A list of dictionaries containing a 'price' key.
        tax_rate (float, optional): Tax rate as a decimal. Defaults to 0.1.
    Returns:
        float: Total price including tax, rounded to 2 decimal places.
    Example:
        >>> items = [{'price': 10.0}, {'price': 20.0}]
        >>> calculate_total_price(items)
        33.0
    """
    subtotal = sum(item['price'] for item in items)
    return round(subtotal * (1 + tax_rate), 2)
def validate_email(email: str) -> bool:
    """Validates an email address format.
    Args:
        email (str): The email address to validate.
    Returns:
        bool: True if valid, False otherwise.
    Example:
        >>> validate_email("user@example.com")
        True
    """
    return "@" in email and "." in email.split("@")[-1]
def merge_dictionaries(*dicts: dict) -> dict:
    """

    Merges multiple dictionaries into one.

    Args:
        *dicts (dict): Variable number of dictionaries.

    Returns:
        dict: Combined dictionary with later values overwriting earlier ones.

    Example:
        >>> merge_dictionaries({'a': 1}, {'a': 2})
        {'a': 2}
    """
    result = {}
    for d in dicts:
        result.update(d)
    return result
if __name__ == "__main__":
    items = [{'price': 10.0}, {'price': 20.0}]
    print("Total Price:", calculate_total_price(items, tax_rate=0.1))
    print("Valid Email:", validate_email("user@example.com"))
    print("Invalid Email:", validate_email("invalid.email"))
    print("Merged Dictionary:", merge_dictionaries({'a': 1}, {'b': 2}, {'a': 3}))
```

```
1.2
(.venv) PS C:\Users\zobiya\Downloads\AI-Assisted coding> ^C
(.venv) PS C:\Users\zobiya\Downloads\AI-Assisted coding>  c:; cd 'c:\Users\zobiya\Downloads\AI-Assi
sted coding'; & 'c:\Users\zobiya\Downloads\AI-Assisted coding\.venv\Scripts\python.exe' 'c:\Users\z
obiya\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '5223
4' '--' 'C:\Users\zobiya\Downloads\AI-Assisted coding\lab-9.4.py'
Total Price: 33.0
Valid Email: True
Invalid Email: False
Merged Dictionary: {'a': 3, 'b': 2}
(.venv) PS C:\Users\zobiya\Downloads\AI-Assisted coding> 
```

One-shot Prompt: Give an one-shot example for the same code.



To customize Run and Debug create a launch.json file.

Debug using a terminal command or in an interactive chat.

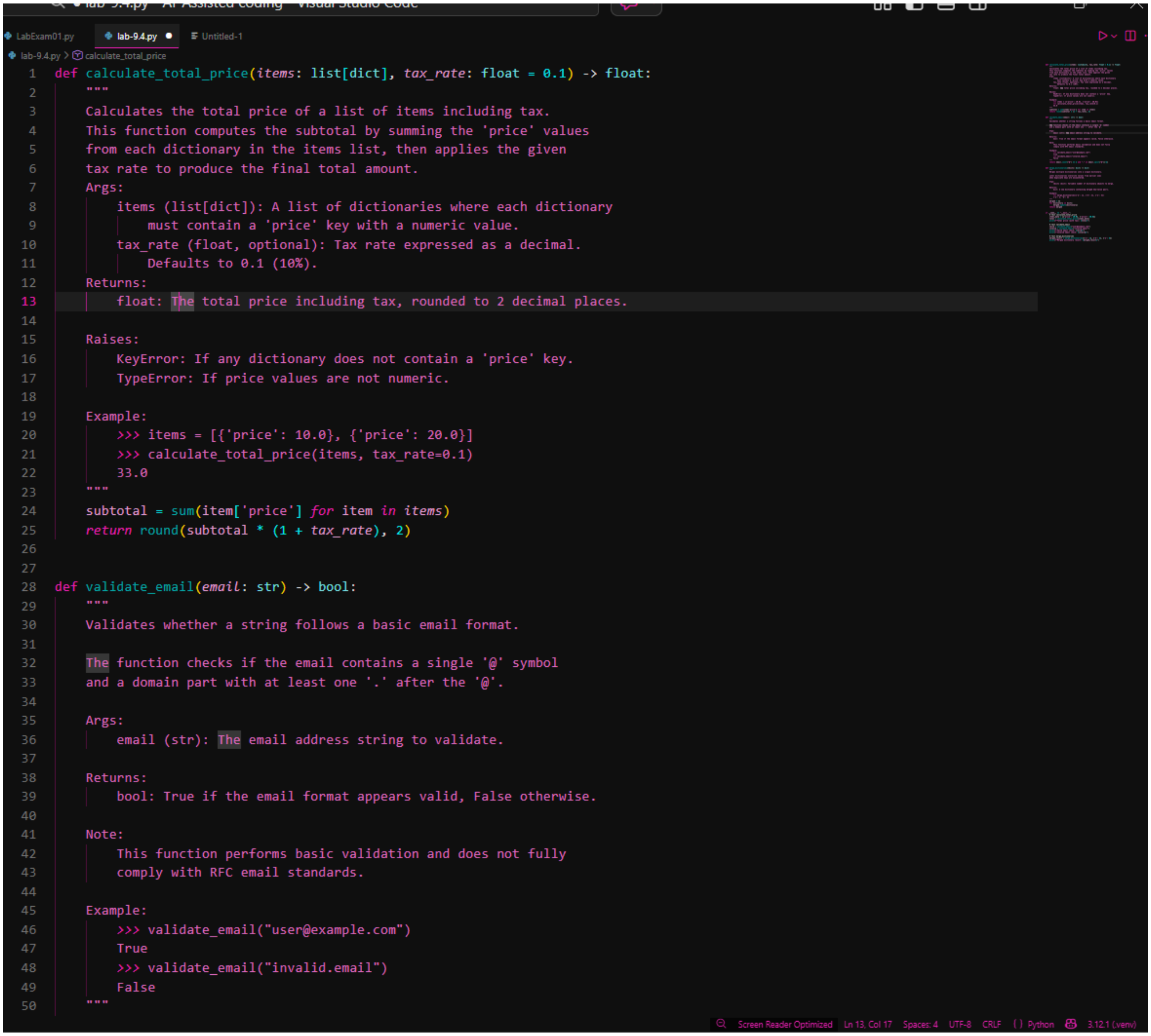
```python
def calculate_total_price(items: list[dict], tax_rate: float = 0.1) -> float:
    """Calculates the total price of items including tax.
    Args:
        items (list[dict]): A list of dictionaries containing a 'price' key.
        tax_rate (float, optional): Tax rate as a decimal. Defaults to 0.1.
    Returns:
        float: Total price including tax, rounded to 2 decimal places.
    Example:
        >>> items = [{'price': 10.0}, {'price': 20.0}]
        >>> calculate_total_price(items)
        33.0"""
    subtotal = sum(item['price'] for item in items)
    return round(subtotal * (1 + tax_rate), 2)
def validate_email(email: str) -> bool:
    """Validates an email address format.
    Args:
        email (str): The email address to validate.
    Returns:
        bool: True if valid, False otherwise.
    Example:
        >>> validate_email("user@example.com")
        True"""
    return "@" in email and "." in email.split("@")[-1]
def merge_dictionaries(*dicts: dict) -> dict:
    """ Merges multiple dictionaries into one.
    Args:
        *dicts (dict): Variable number of dictionaries.
    Returns:
        dict: Combined dictionary with later values overwriting earlier ones.
    Example:
        >>> merge_dictionaries({'a': 1}, {'a': 2})
        {'a': 2} """
    result = {}
    for d in dicts:
        result.update(d)
    return result
if __name__ == "__main__":
    items = [{'price': 10.0}, {'price': 20.0}]
    print("Total Price:", calculate_total_price(items, tax_rate=0.1))
    print("Valid Email:", validate_email("user@example.com"))
    print("Invalid Email:", validate_email("invalid.email"))
    print("Merged Dictionary:", merge_dictionaries({'a': 1}, {'b': 2}, {'a': 3}))
    # Additional test case demonstrating all functions
    test_items = [{'price': 15.50}, {'price': 24.99}, {'price': 10.01}]
    print("\nAdditional Examples:")
    print("Total with custom tax:", calculate_total_price(test_items, tax_rate=0.15))
    print("Email validation:", validate_email("john.doe@company.co.uk"))
    print("Merged dicts:", merge_dictionaries({'x': 10, 'y': 20}, {'y': 30, 'z': 40}))
```

```
(.venv) PS C:\Users\zobiya\Downloads\AI-Assisted coding> .12 111..^C
(.venv) PS C:\Users\zobiya\Downloads\AI-Assisted coding>  c:; cd 'c:\Users\zobiya\Downloads\AI-Assisted coding'; & 'c:\Users\
zobiya\Downloads\AI-Assisted coding\.venv\Scripts\python.exe' 'c:\Users\zobiya\.vscode\extensions\ms-python.debugpy-2025.18.0
-win32-x64\bundled\libs\debugpy\launcher' '53669' '--' 'C:\Users\zobiya\Downloads\AI-Assisted coding\lab-9.4.py'
Total Price: 33.0
Valid Email: True
Invalid Email: False
Merged Dictionary: {'a': 3, 'b': 2}

Additional Examples:
Total with custom tax: 58.07
Email validation: True
Merged dicts: {'x': 10, 'y': 30, 'z': 40}
(.venv) PS C:\Users\zobiya\Downloads\AI-Assisted coding> 
```

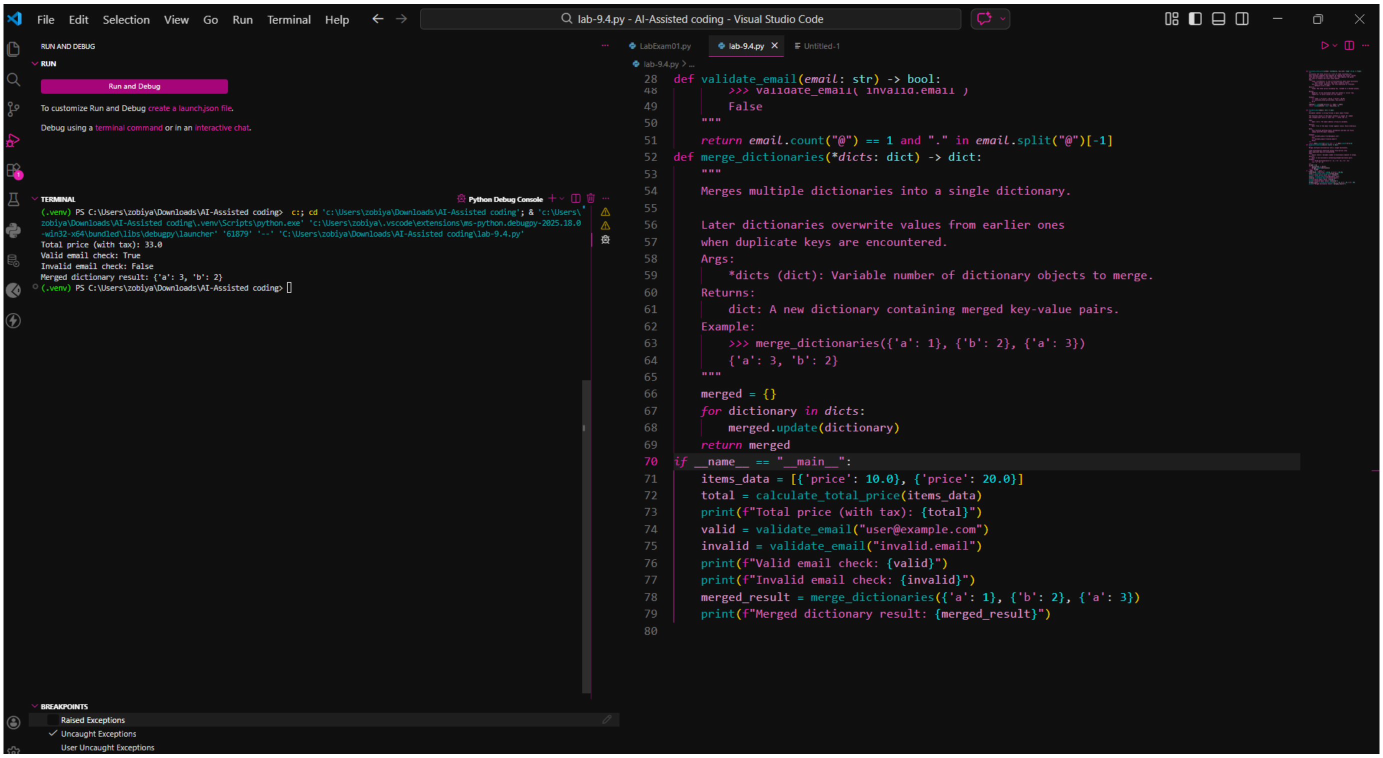
Context-based prompt: give the same code but in context based follow the below instructions: Improve code readability and maintainability by:

1. Writing detailed Google-style docstrings for each function.
2. Clearly explaining function purpose and behavior.
3. Specifying parameter types and expected data structures.
4. Mentioning edge cases or assumptions where relevant.
5. Providing realistic example usage.
6. Adding a well-structured main testing section.
7. Printing labeled outputs for clarity.

Ensure the documentation is professional and consistent across all functions.



```python
def calculate_total_price(items: list[dict], tax_rate: float = 0.1) -> float:
    """
    Calculates the total price of a list of items including tax.
    This function computes the subtotal by summing the 'price' values
    from each dictionary in the items list, then applies the given
    tax rate to produce the final total amount.
    Args:
        items (list[dict]): A list of dictionaries where each dictionary
            must contain a 'price' key with a numeric value.
        tax_rate (float, optional): Tax rate expressed as a decimal.
            Defaults to 0.1 (10%).
    Returns:
        float: The total price including tax, rounded to 2 decimal places.

    Raises:
        KeyError: If any dictionary does not contain a 'price' key.
        TypeError: If price values are not numeric.

    Example:
        >>> items = [{'price': 10.0}, {'price': 20.0}]
        >>> calculate_total_price(items, tax_rate=0.1)
        33.0
    """
    subtotal = sum(item['price'] for item in items)
    return round(subtotal * (1 + tax_rate), 2)


def validate_email(email: str) -> bool:
    """
    Validates whether a string follows a basic email format.

    The function checks if the email contains a single '@' symbol
    and a domain part with at least one '.' after the '@'.

    Args:
        email (str): The email address string to validate.

    Returns:
        bool: True if the email format appears valid, False otherwise.

    Note:
        This function performs basic validation and does not fully
        comply with RFC email standards.

    Example:
        >>> validate_email("user@example.com")
        True
        >>> validate_email("invalid.email")
        False
    """
```



To customize Run and Debug create a launch.json file.

Debug using a terminal command or in an interactive chat.

```
(.venv) PS C:\Users\zobiya\Downloads\AI-Assisted coding>  c:; cd 'c:\Users\zobiya\Downloads\AI-Assisted coding'; & 'c:\Users\
zobiya\Downloads\AI-Assisted coding\.venv\Scripts\python.exe' 'c:\Users\zobiya\.vscode\extensions\ms-python.debugpy-2025.18.0
-win32-x64\bundled\libs\debugpy\launcher' '61879' '--' 'C:\Users\zobiya\Downloads\AI-Assisted coding\lab-9.4.py'
Total price (with tax): 33.0
Valid email check: True
Invalid email check: False
Merged dictionary result: {'a': 3, 'b': 2}
(.venv) PS C:\Users\zobiya\Downloads\AI-Assisted coding> 
```

```python
def validate_email(email: str) -> bool:
        >>> validate_email("invalid.email")
        False
    """
    return email.count("@") == 1 and "." in email.split("@")[-1]
def merge_dictionaries(*dicts: dict) -> dict:
    """
    Merges multiple dictionaries into a single dictionary.

    Later dictionaries overwrite values from earlier ones
    when duplicate keys are encountered.
    Args:
        *dicts (dict): Variable number of dictionary objects to merge.
    Returns:
        dict: A new dictionary containing merged key-value pairs.
    Example:
        >>> merge_dictionaries({'a': 1}, {'b': 2}, {'a': 3})
        {'a': 3, 'b': 2}
    """
    merged = {}
    for dictionary in dicts:
        merged.update(dictionary)
    return merged
if __name__ == "__main__":
    items_data = [{'price': 10.0}, {'price': 20.0}]
    total = calculate_total_price(items_data)
    print(f"Total price (with tax): {total}")
    valid = validate_email("user@example.com")
    invalid = validate_email("invalid.email")
    print(f"Valid email check: {valid}")
    print(f"Invalid email check: {invalid}")
    merged_result = merge_dictionaries({'a': 1}, {'b': 2}, {'a': 3})
    print(f"Merged dictionary result: {merged_result}")

```

Differences:

Prompt Type	Description Detail	Example Quality	Testing Block	Professionalism
Zero-shot	Basic	Simple	Minimal	Medium
One-shot	Structured	Clean	Better	High
Context-based	Detailed	Realistic	Clear & labeled	Very High

Summary:

Using context-based prompting with GitHub Copilot significantly improves documentation quality in a shared codebase. Compared to zero-shot and one-shot prompting, context-based instructions generate more detailed descriptions, clearly structured Google-style docstrings, edge-case considerations, and better formatted testing sections. The resulting documentation is more professional, readable, and maintainable. Additionally, the inclusion of a main execution block ensures the script produces visible outputs, improving usability for new developers. While AI greatly accelerates documentation generation, manual review is still necessary to ensure correctness and completeness.

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:

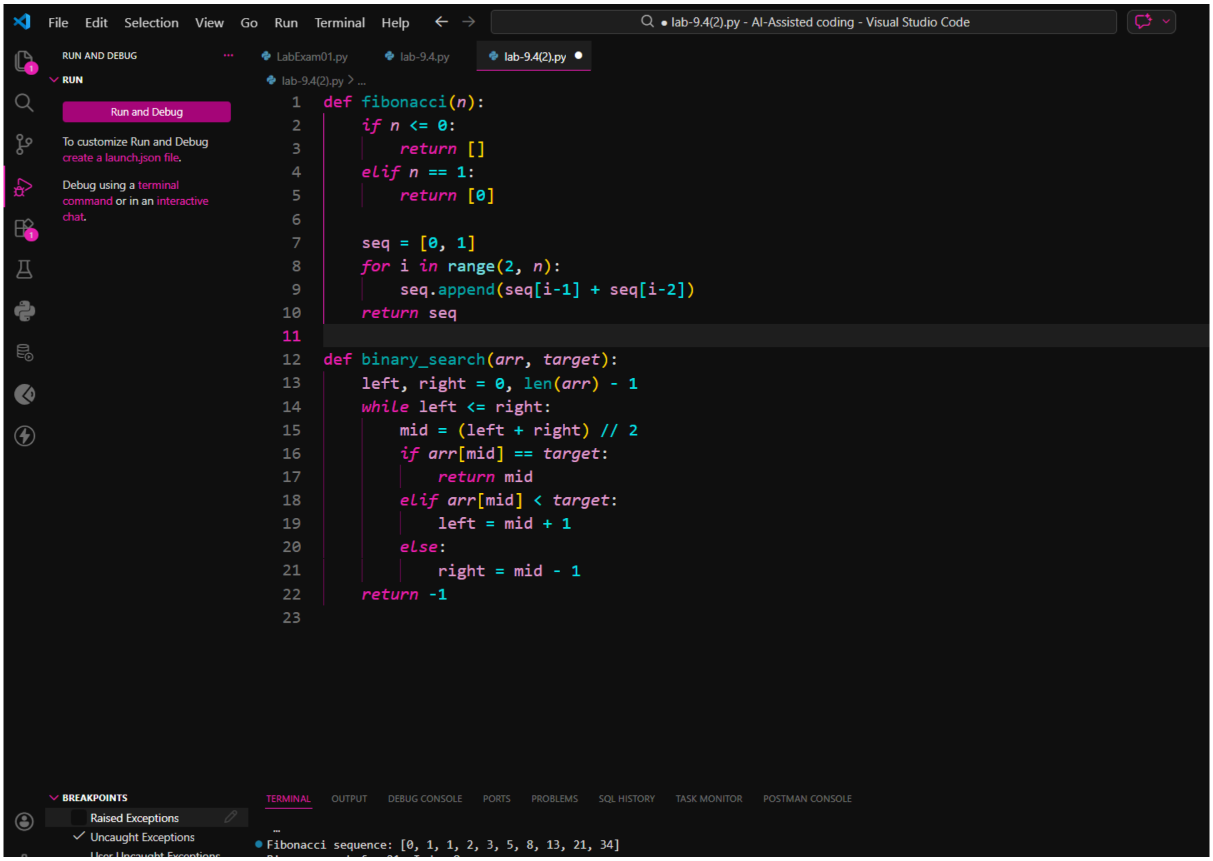
- Loops
- Conditional logic
- Algorithms (such as Fibonacci sequence, sorting, or searching)

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

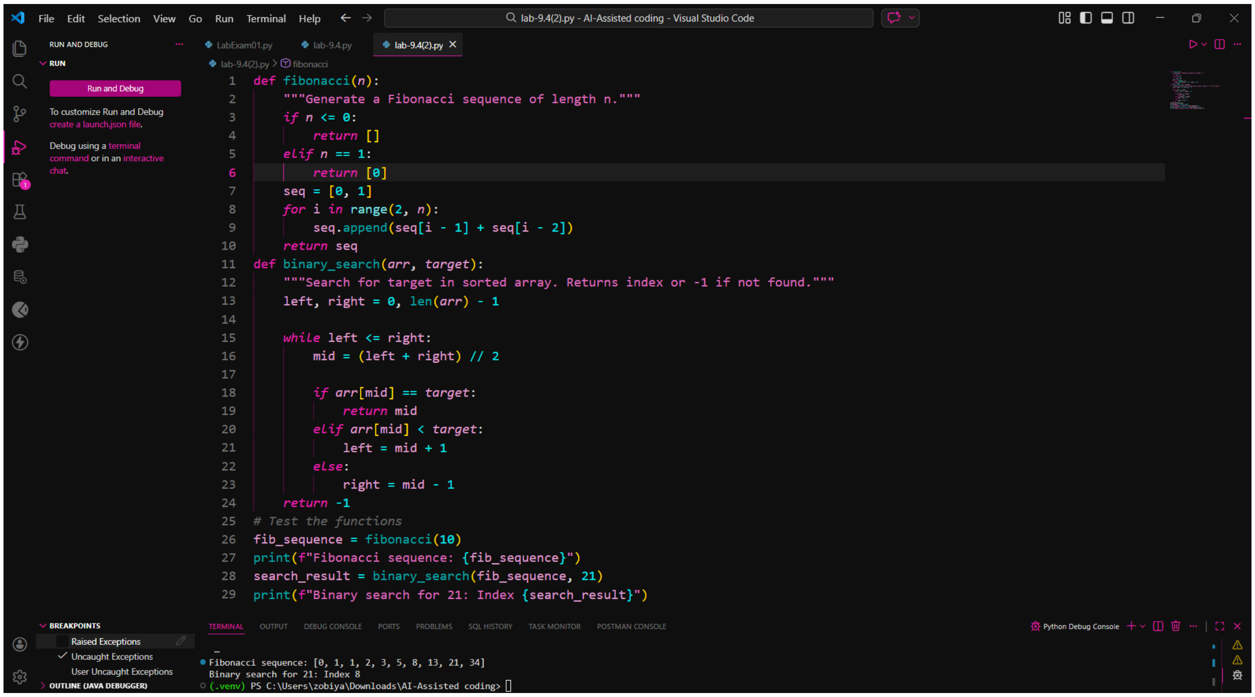
Normal code:


File Edit Selection View Go Run Terminal Help
● lab-9.4(2).py - AI-Assisted coding - Visual Studio Code
RUN AND DEBUG
RUN
Run and Debug
To customize Run and Debug create a launch.json file.
Debug using a terminal command or in an interactive chat.
LabExam01.py
lab-9.4.py
lab-9.4(2).py
lab-9.4(2).py > ...
BREAKPOINTS
Raised Exceptions
✓ Uncaught Exceptions
TERMINAL
OUTPUT
DEBUG CONSOLE
PORTS
PROBLEMS
SQL HISTORY
TASK MONITOR
POSTMAN CONSOLE

```python
def fibonacci(n):
    if n <= 0:
        return []
    elif n == 1:
        return [0]

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

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

```

```
...
Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Prompt: Improve the readability of the code:



```python
def fibonacci(n):
    """Generate a Fibonacci sequence of length n."""
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    seq = [0, 1]
    for i in range(2, n):
        seq.append(seq[i - 1] + seq[i - 2])
    return seq
def binary_search(arr, target):
    """Search for target in sorted array. Returns index or -1 if not found."""
    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
# Test the functions
fib_sequence = fibonacci(10)
print(f"Fibonacci sequence: {fib_sequence}")
search_result = binary_search(fib_sequence, 21)
print(f"Binary search for 21: Index {search_result}")
```

```
...
Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Binary search for 21: Index 8
(.venv) PS C:\Users\zobiya\Downloads\AI-Assisted coding>
```

Before vs After codes:

Aspect	Before	After AI
Readability	Moderate	High
Algorithm clarity	Needs thinking	Clear reasoning
Edge-case explanation	Missing	Explained
Over-commenting	None	Avoided
Maintainability	Medium	Improved

Summary:

AI assistance was used to insert meaningful inline comments into complex logic sections of the program. The AI was instructed to avoid trivial syntax explanations and focus only on non-obvious algorithm reasoning and decision-making steps. The resulting script improved readability without cluttering the code. The comments explain why certain logic exists (e.g., midpoint calculation in binary search), making future debugging and extension easier.

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

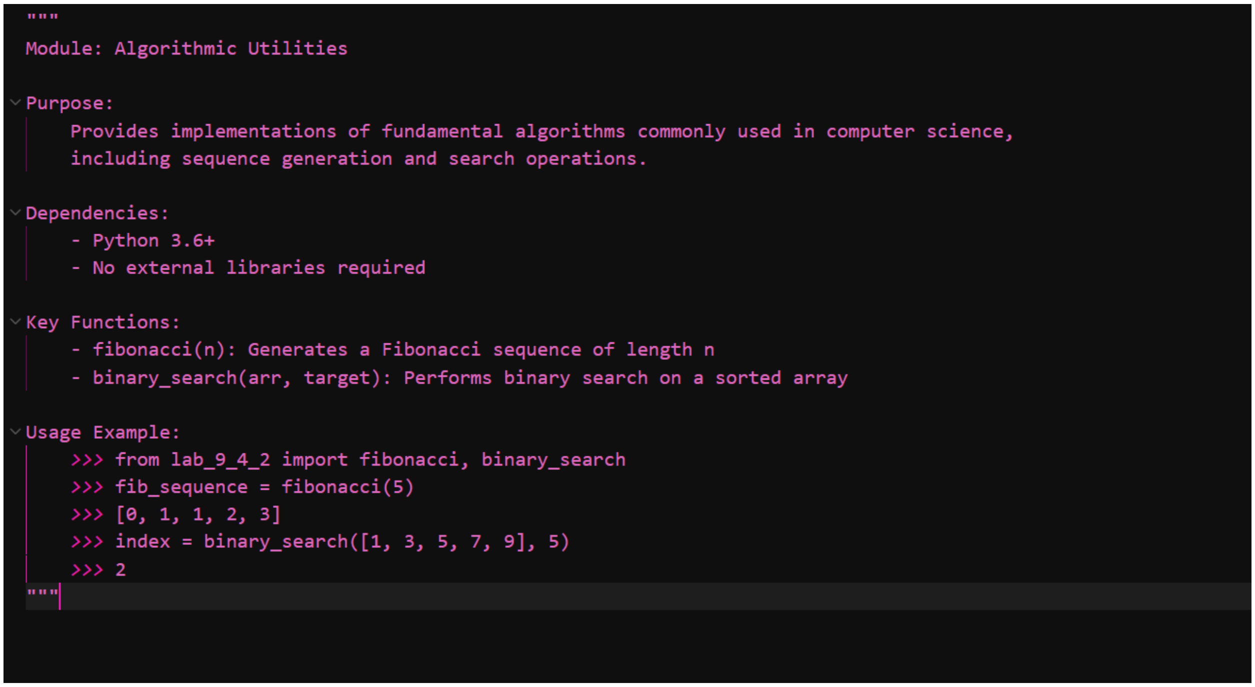
Provide a complete Python module to an AI tool and instruct it to automatically generate a module-level docstring at the top of the file that includes:

- 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.

<u>Prompt:</u> Generate a professional module-level docstring to be placed at the top of the file. The docstring must include:

A clear purpose of the module, Required libraries or dependencies (if any), A brief overview of key functions and classes, A short usage example showing how to import and use the module, Maintain a professional tone suitable for production-level code.



```
"""
Module: Algorithmic Utilities

Purpose:
    Provides implementations of fundamental algorithms commonly used in computer science,
    including sequence generation and search operations.

Dependencies:
    - Python 3.6+
    - No external libraries required

Key Functions:
    - fibonacci(n): Generates a Fibonacci sequence of length n
    - binary_search(arr, target): Performs binary search on a sorted array

Usage Example:
    >>> from lab_9_4_2 import fibonacci, binary_search
    >>> fib_sequence = fibonacci(5)
    >>> [0, 1, 1, 2, 3]
    >>> index = binary_search([1, 3, 5, 7, 9], 5)
    >>> 2
"""
```

Summary:

AI assistance was used to generate a structured module-level docstring summarizing the purpose, dependencies, and key components of the module. Context-based prompting resulted in a professional, well-organized documentation block suitable for repository distribution. The generated overview improves maintainability, onboarding efficiency, and overall code clarity.

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

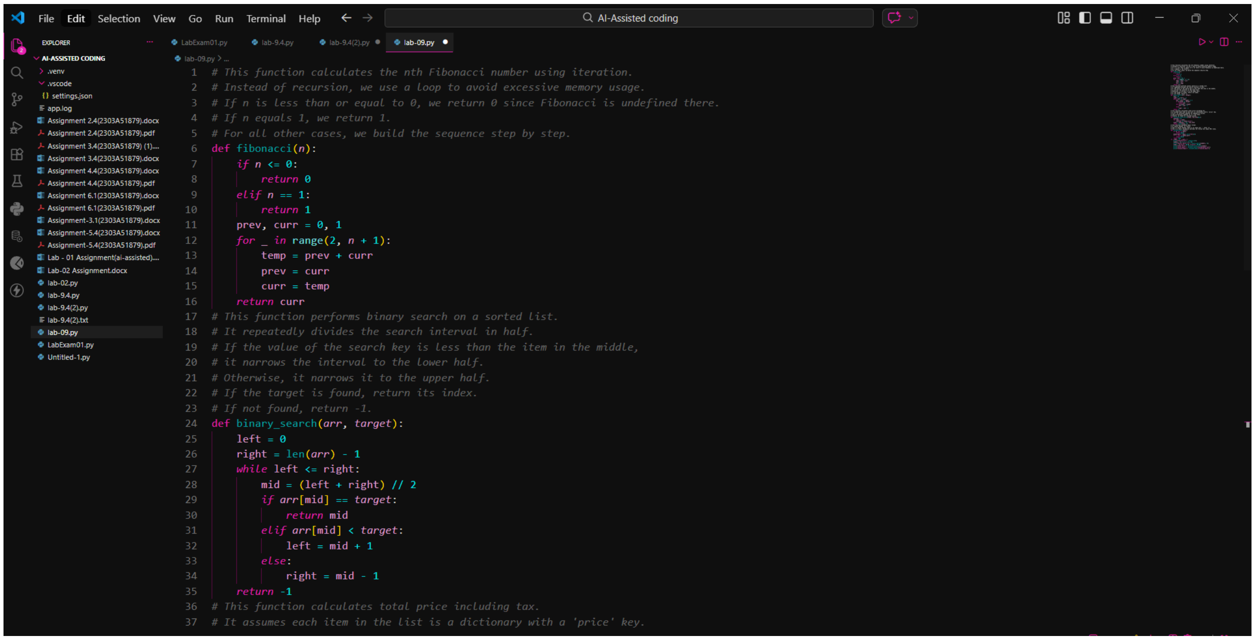
- Functions with clean, standardized docstrings
- Reduced clutter inside function bodies
- Improved consistency across the codebase

Prompt:

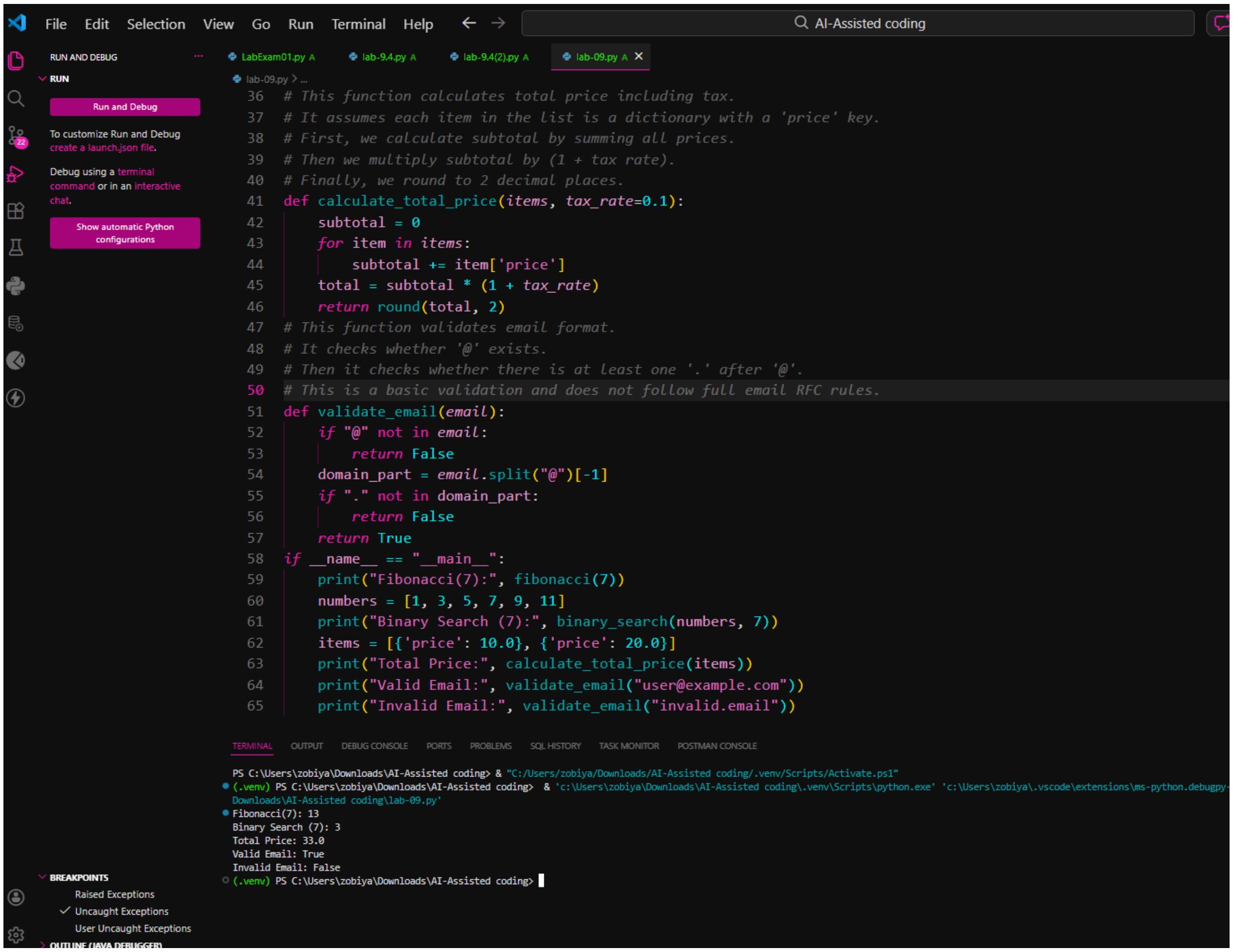
Convert the explanatory inline comments into a structured Google-style docstring placed at the top of each function. Preserve the original meaning and intent of the comments.

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.
```



```python
# 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.
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"))
```

```
PS C:\Users\zobiya\Downloads\AI-Assisted coding> & "C:/Users/zobiya/Downloads/AI-Assisted coding/.venv/Scripts/Activate.ps1"
(.venv) PS C:\Users\zobiya\Downloads\AI-Assisted coding>  & 'c:\Users\zobiya\Downloads\AI-Assisted coding\.venv\Scripts\python.exe' 'c:\Users\zobiya\.vscode\extensions\ms-python.debugpy-
Downloads\AI-Assisted coding\lab-09.py'
Fibonacci(7): 13
Binary Search (7): 3
Total Price: 33.0
Valid Email: True
Invalid Email: False
(.venv) PS C:\Users\zobiya\Downloads\AI-Assisted coding>
```

Summary:

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.

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.

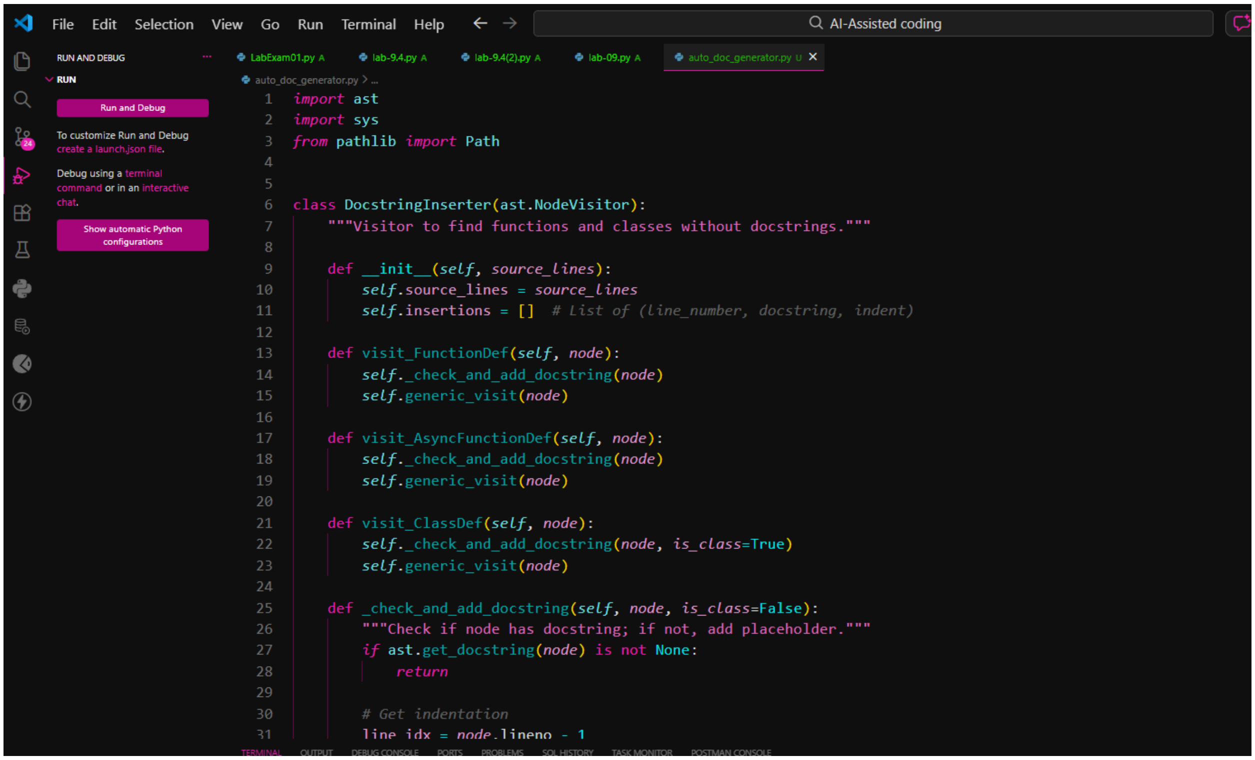
Note: The goal is documentation scaffolding, not perfect documentation.

Expected Outcome

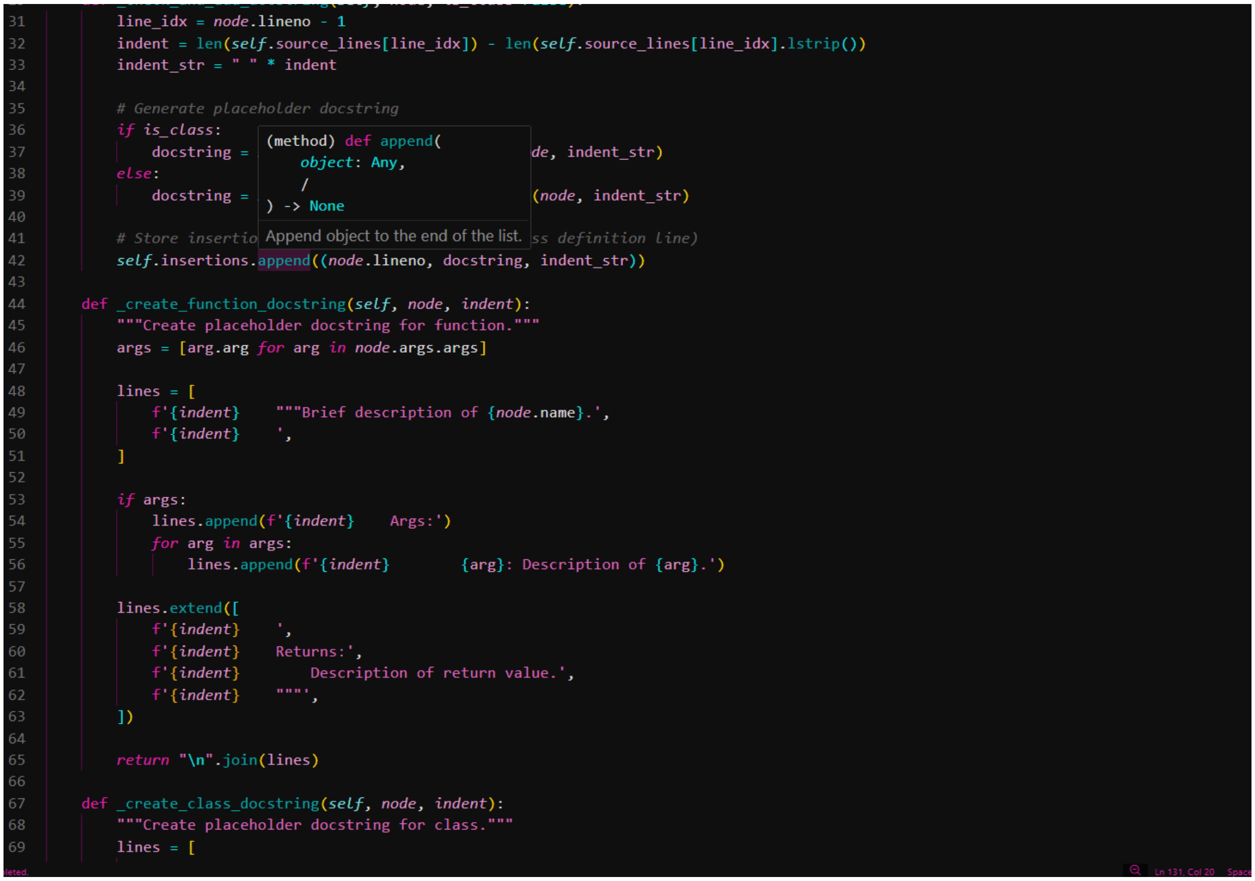
- A working Python script that processes another .py file
- Automatically inserted placeholder docstrings
- Clear demonstration of how AI can assist in documentation automation

Prompt:

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.



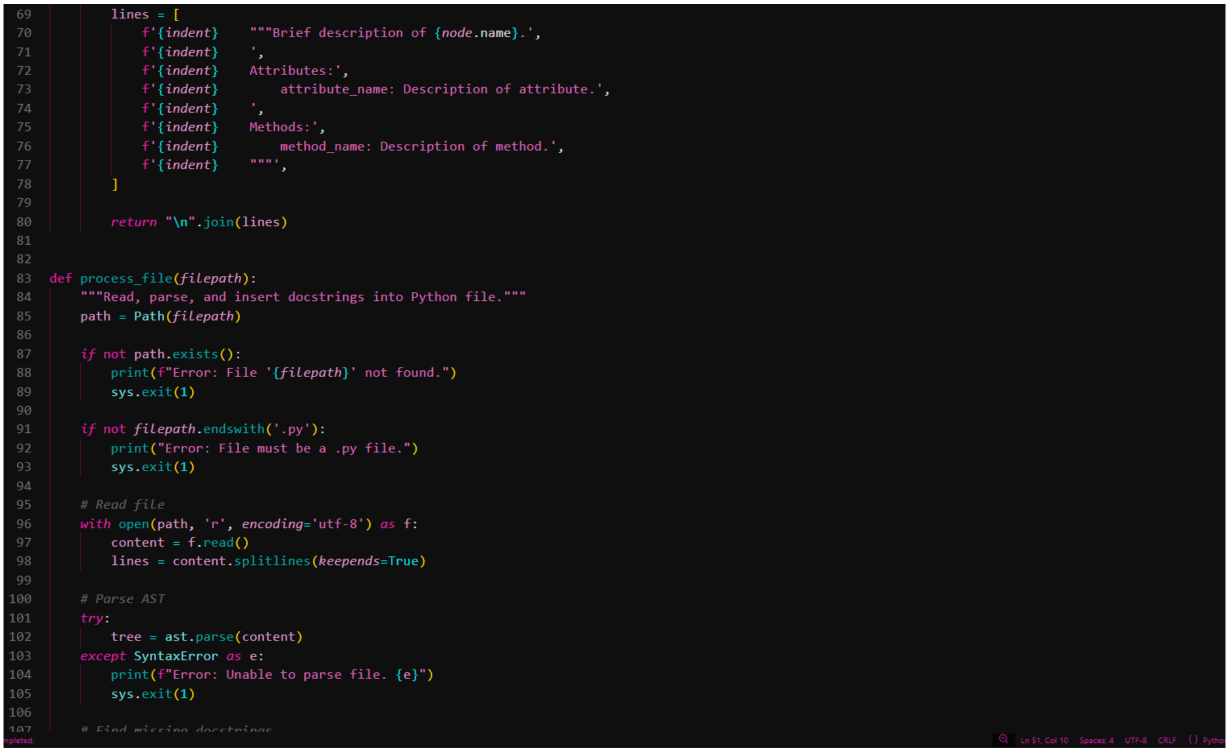
```python
import ast
import sys
from pathlib import Path


class DocstringInserter(ast.NodeVisitor):
    """Visitor to find functions and classes without docstrings."""

    def __init__(self, source_lines):
        self.source_lines = source_lines
        self.insertions = []  # List of (line_number, docstring, indent)

    def visit_FunctionDef(self, node):
        self._check_and_add_docstring(node)
        self.generic_visit(node)

    def visit_AsyncFunctionDef(self, node):
        self._check_and_add_docstring(node)
        self.generic_visit(node)

    def visit_ClassDef(self, node):
        self._check_and_add_docstring(node, is_class=True)
        self.generic_visit(node)

    def _check_and_add_docstring(self, node, is_class=False):
        """Check if node has docstring; if not, add placeholder."""
        if ast.get_docstring(node) is not None:
            return

        # Get indentation
        line_idx = node.lineno - 1
```



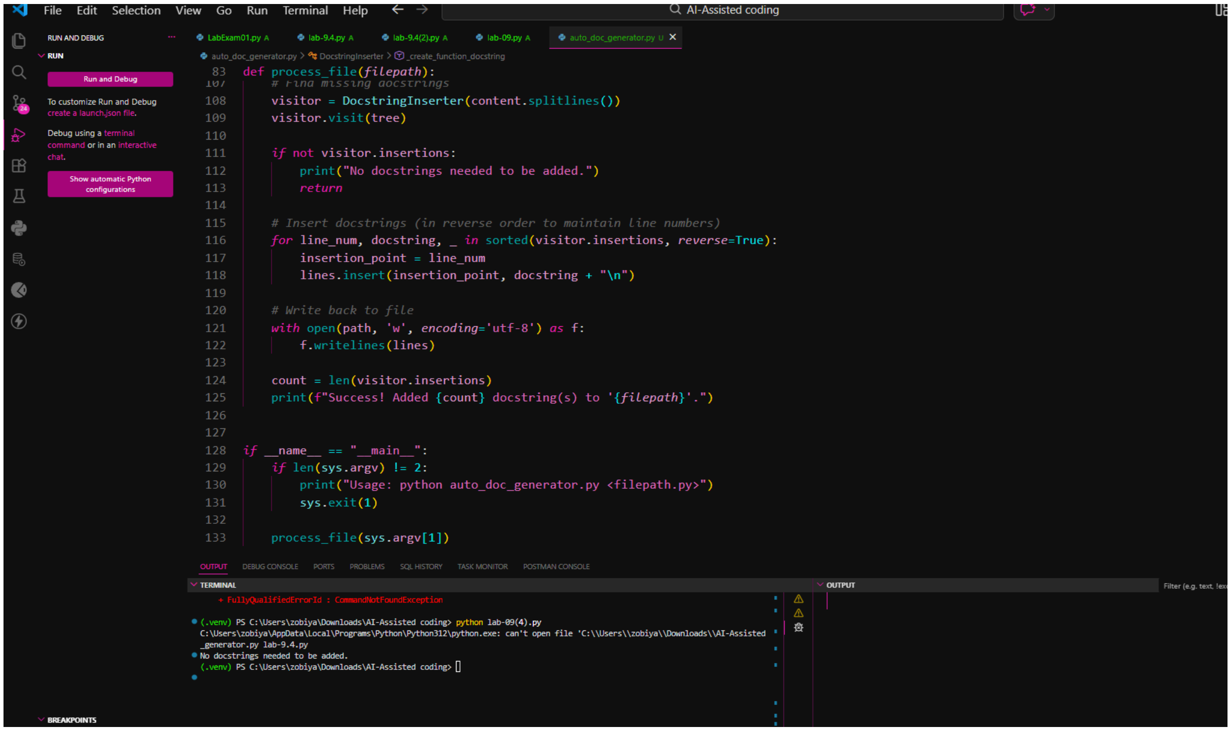
```python
        line_idx = node.lineno - 1
        indent = len(self.source_lines[line_idx]) - len(self.source_lines[line_idx].lstrip())
        indent_str = " " * indent

        # Generate placeholder docstring
        if is_class:
            docstring =                               de, indent_str)
        else:
            docstring =                               (node, indent_str)

        # Store insertio                              ss definition line)
        self.insertions.append((node.lineno, docstring, indent_str))

    def _create_function_docstring(self, node, indent):
        """Create placeholder docstring for function."""
        args = [arg.arg for arg in node.args.args]

        lines = [
            f'{indent}    """Brief description of {node.name}.',
            f'{indent}    ',
        ]

        if args:
            lines.append(f'{indent}    Args:')
            for arg in args:
                lines.append(f'{indent}        {arg}: Description of {arg}.')

        lines.extend([
            f'{indent}    ',
            f'{indent}    Returns:',
            f'{indent}        Description of return value.',
            f'{indent}    """',
        ])

        return "\n".join(lines)

    def _create_class_docstring(self, node, indent):
        """Create placeholder docstring for class."""
        lines = [
```

```
(method) def append(
    object: Any,
    /
) -> None
```

Append object to the end of the list.



```python
        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)
```



To customize Run and Debug create a launch.json file.

Debug using a terminal command or in an interactive chat.

```python
def process_file(filepath):
    # Find missing docstrings
    visitor = DocstringInserter(content.splitlines())
    visitor.visit(tree)

    if not visitor.insertions:
        print("No docstrings needed to be added.")
        return

    # Insert docstrings (in 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])
```

```
    + FullyQualifiedErrorId : CommandNotFoundException

(.venv) PS C:\Users\zobiya\Downloads\AI-Assisted coding> python lab-09(4).py
C:\Users\zobiya\AppData\Local\Programs\Python\Python312\python.exe: can't open file 'C:\\Users\\zobiya\\Downloads\\AI-Assisted
_generator.py lab-9.4.py
No docstrings needed to be added.
(.venv) PS C:\Users\zobiya\Downloads\AI-Assisted coding> 
```

Summary:

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.