Lab Exercise 1 - GitHub Copilot for Code Generation, Explanation, Refactoring, and Automated Documentation

1. Objective

This lab demonstrates how to use **GitHub Copilot** to:

- Generate code from natural language prompts
- Explain existing code
- Refactor and improve code quality
- Automatically generate documentation

The lab uses **Python (Flask-style service)** as an example, but the steps apply to other languages like Java or JavaScript.

---

2. Prerequisites

- GitHub account with **GitHub Copilot enabled**
- IDE: **VS Code / PyCharm** (with Copilot extension installed)
- Basic knowledge of Python

---

3. Lab Setup

Step 1: Enable GitHub Copilot

1. Open your IDE
2. Install **GitHub Copilot** extension
3. Sign in with your GitHub account
4. Verify Copilot is active (Copilot icon visible)

---

4. Part A: Code Generation Using Copilot

Task

Generate a Python service class that calculates the **total order amount** for a customer.

Step 1: Create a New Python File

Create a file named:

order_service.py

Step 2: Write a Natural Language Prompt

Type the following comment and wait for Copilot suggestions:

```
# Create a service class to calculate total order amount for a customer
# Input: list of order amounts
# Output: total amount
```

Step 3: Accept Copilot Suggestion

Copilot will generate code similar to:

```
class OrderService:
    def calculate_total(self, orders: list[float]) -> float:
        total = 0
        for order in orders:
            total += order
        return total
```

Outcome: Code is generated automatically from a comment.

---

5. Part B: Code Explanation Using Copilot

Task

Use Copilot to explain existing code.

Step 1: Select the Code

Highlight the calculate_total method.

Step 2: Ask Copilot to Explain

Use Copilot Chat or comment:

```
# Explain what this method does
```

Example Explanation (Copilot Output)

- Iterates through a list of order values
- Adds each value to a running total
- Returns the final sum as the total order amount

Outcome: Copilot provides a clear, human-readable explanation.

---

6. Part C: Code Refactoring Using Copilot

Task

Refactor the code to use **Pythonic built-in functions**.

Step 1: Add a Refactoring Prompt

Above the method, add:

```
# Refactor this method using Python's built-in sum function
```

Step 2: Accept Refactored Code

Copilot suggests:

```
def calculate_total(self, orders: list[float]) -> float:
    return sum(orders)
```

Improvements:

- Cleaner code
- Better readability
- Pythonic style using built-in functions

---

7. Part D: Automated Documentation Generation

Task

Generate docstrings automatically using Copilot.

Step 1: Place Cursor Above Method

Type:

```
"""

"""
```

Step 2: Let Copilot Generate Docstring

Copilot generates:

```
"""
Calculates the total amount of all orders for a customer.

Args:
    orders: List of order amounts

Returns:
    Total order amount as a float
"""
```

Outcome: Documentation created with minimal effort.

---

8. Part E: Copilot for Test Case Generation

Task

Generate a unit test for the service.

Step 1: Create Test File

test_order_service.py

Step 2: Prompt Copilot

```
# Write a pytest test for calculate_total method
```

Example Output

```
import pytest
from order_service import OrderService

def test_calculate_total():
    service = OrderService()
    orders = [100.0, 200.0, 300.0]
    assert service.calculate_total(orders) == 600.0
```

Outcome: Copilot generates a ready-to-use unit test.

---

9. Part F: Copilot for Code Improvement Suggestions

Task

Ask Copilot to improve robustness.

Prompt:

```
# Improve this code to handle None or empty lists
```

Copilot suggests:

```
def calculate_total(self, orders: list[float] | None) -> float:
    if orders is None or len(orders) == 0:
        return 0.0
    return sum(orders)
```

Outcome: Safer, production-ready code.

---

10. Observations and Learning Outcomes

- GitHub Copilot accelerates **code writing**
- Helps understand **unfamiliar code**
- Encourages **best practices and refactoring**

- Reduces effort in **documentation and testing**
- Acts as an AI pair programmer, not a replacement

---

11. Conclusion

This lab demonstrated how GitHub Copilot can be effectively used for **code generation, explanation, refactoring, and automated documentation**. By leveraging AI-assisted development, developers can improve productivity, code quality, and maintainability while focusing on design and problem-solving.