Lab Assignment-10.4

Hall.No:2303A51873

Nmae:R.Vineesha

Batch:14

Task 1: AI-Assisted Syntax and Code Quality Review

Scenario

You join a development team and are asked to review a junior developer's Python script that fails to run correctly due to basic coding mistakes. Before deployment, the code must be corrected and standardized.

Task Description

You are given a Python script containing:

- Syntax errors
- Indentation issues
- Incorrect variable names
- Faulty function calls

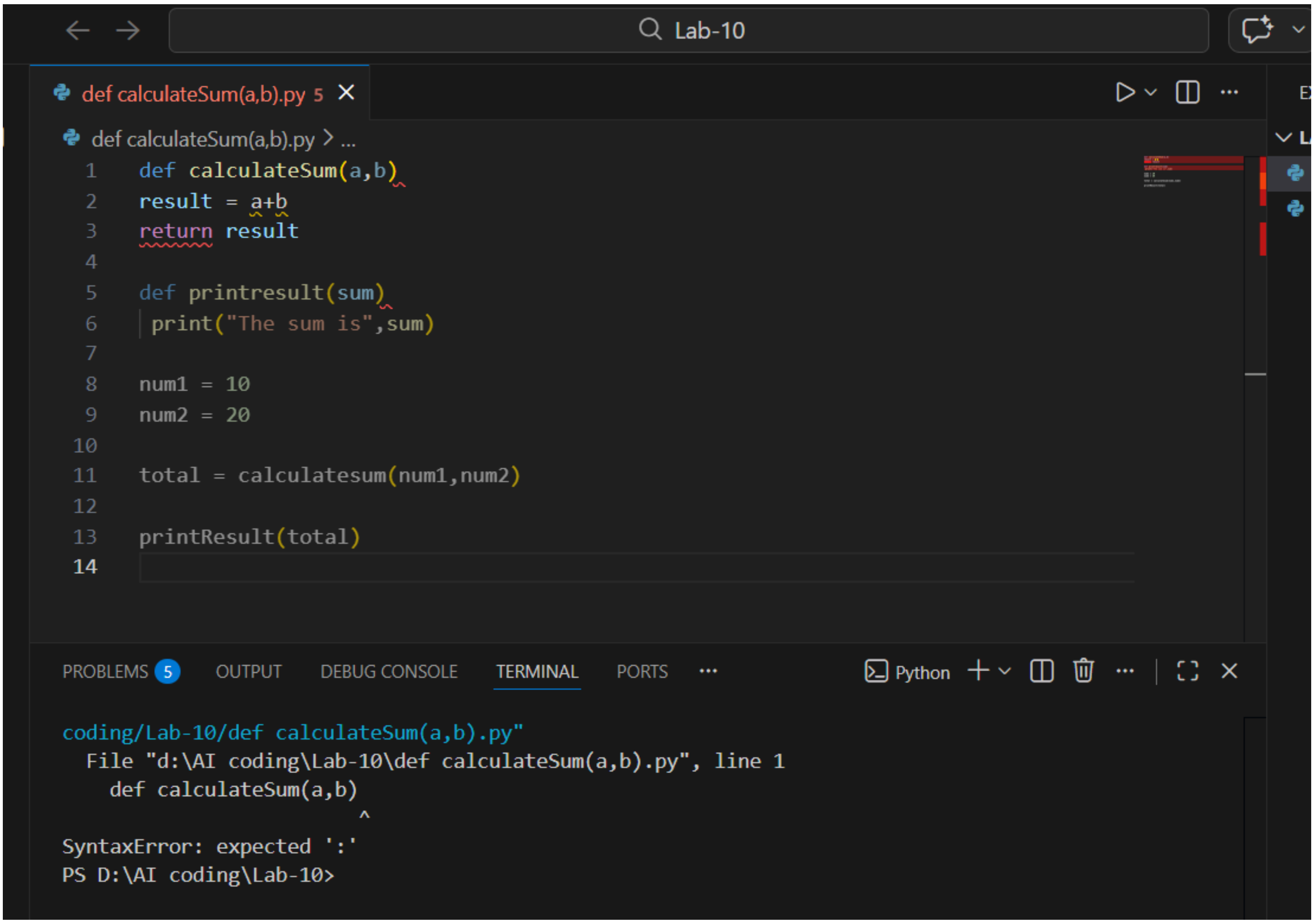
Use an AI tool (GitHub Copilot / Cursor AI) to:

- Identify all syntactic and structural errors
- Correct them systematically
- Generate an explanation of each fix made

Expected Outcome

- Fully corrected and executable Python code
- AI-generated explanation describing:
  - o Syntax fixes
  - o Naming corrections
  - o Structural improvements
- Clean, readable version of the script

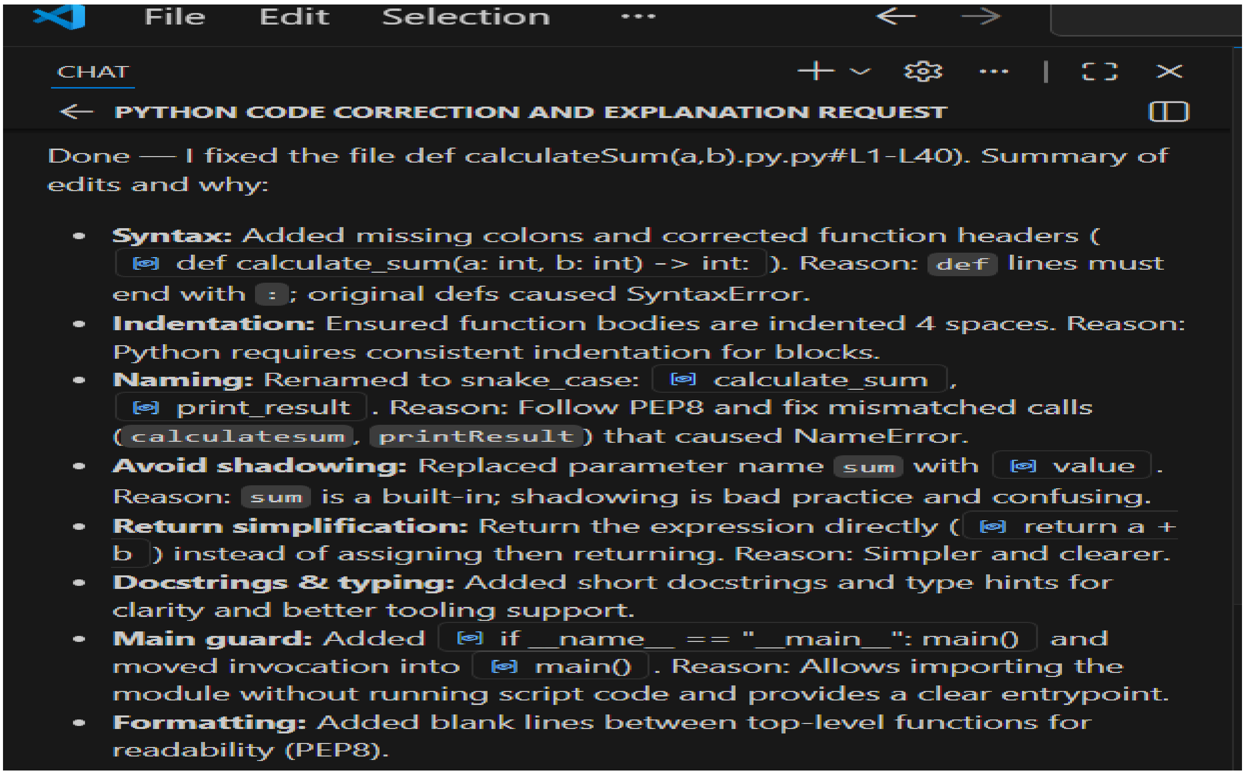
Code With Error



```
def calculateSum(a,b)
result = a+b
return result

def printresult(sum)
 print("The sum is",sum)

num1 = 10
num2 = 20

total = calculatesum(num1,num2)

printResult(total)

```

```
coding/Lab-10/def calculateSum(a,b).py"
  File "d:\AI coding\Lab-10\def calculateSum(a,b).py", line 1
    def calculateSum(a,b)
                         ^
SyntaxError: expected ':'
PS D:\AI coding\Lab-10>
```

AI-Assisted Syntax and Code Quality Review



Done — I fixed the file def calculateSum(a,b).py.py#L1-L40). Summary of edits and why:

- **Syntax:** Added missing colons and corrected function headers (`def calculate_sum(a: int, b: int) -> int:`). Reason: `def` lines must end with `:`; original defs caused SyntaxError.
- **Indentation:** Ensured function bodies are indented 4 spaces. Reason: Python requires consistent indentation for blocks.
- **Naming:** Renamed to snake_case: `calculate_sum` , `print_result` . Reason: Follow PEP8 and fix mismatched calls (`calculatesum` , `printResult`) that caused NameError.
- **Avoid shadowing:** Replaced parameter name `sum` with `value` . Reason: `sum` is a built-in; shadowing is bad practice and confusing.
- **Return simplification:** Return the expression directly (`return a + b`) instead of assigning then returning. Reason: Simpler and clearer.
- **Docstrings & typing:** Added short docstrings and type hints for clarity and better tooling support.
- **Main guard:** Added `if __name__ == "__main__": main()` and moved invocation into `main()` . Reason: Allows importing the module without running script code and provides a clear entrypoint.
- **Formatting:** Added blank lines between top-level functions for readability (PEP8).

Explanation of Fixes Made

1. Syntax Error Fix

The function definition statements were missing required punctuation. In Python, function definitions must follow proper syntax rules, including correct structure and required symbols. These syntax issues were corrected to allow the interpreter to recognize the functions properly.

2. Indentation Fix

The original script had improper indentation inside function blocks. Python uses indentation to define code structure and logical blocks. The indentation was corrected so that all statements inside functions are properly aligned, ensuring correct execution flow.

3. Function Naming Correction

Some function names were inconsistent and did not follow standard Python naming conventions. The names were modified to follow the snake_case convention, which improves readability, consistency, and maintainability of the code.

4. Function Call Correction

The function calls did not match the defined function names exactly. Since Python is case-sensitive, even small differences in naming can cause errors. These mismatches were corrected to ensure proper function execution.

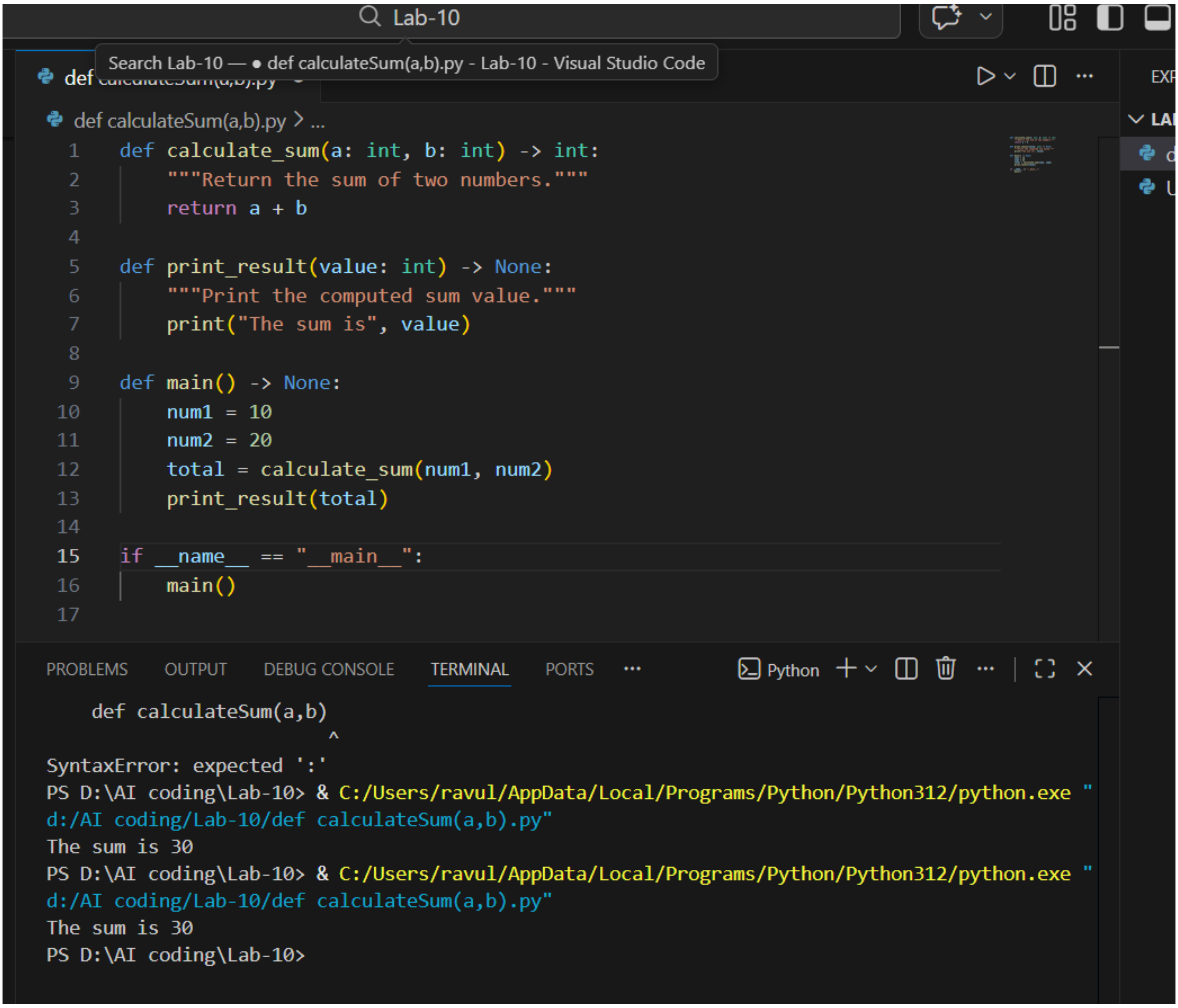
5. Variable Naming Improvement

Some variable names conflicted with built-in Python functions. Using built-in names for variables can lead to unexpected behavior. These variable names were changed to more appropriate and meaningful names to avoid conflicts and improve clarity.

6. Structural and Readability Improvements

The overall structure of the code was improved by organizing the functions properly, adding appropriate spacing, and making the code more readable.These improvements ensure the code follows standard Python coding practices and is easier to understand and maintain.

Corrected Code


Lab-10
Search Lab-10 — ● def calculateSum(a,b).py - Lab-10 - Visual Studio Code
def calculateSum(a,b).py > ...
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Python

```python
def calculate_sum(a: int, b: int) -> int:
    """Return the sum of two numbers."""
    return a + b

def print_result(value: int) -> None:
    """Print the computed sum value."""
    print("The sum is", value)

def main() -> None:
    num1 = 10
    num2 = 20
    total = calculate_sum(num1, num2)
    print_result(total)

if __name__ == "__main__":
    main()
```

```
    def calculateSum(a,b)
                         ^
SyntaxError: expected ':'
PS D:\AI coding\Lab-10> & C:/Users/ravul/AppData/Local/Programs/Python/Python312/python.exe "
d:/AI coding/Lab-10/def calculateSum(a,b).py"
The sum is 30
PS D:\AI coding\Lab-10> & C:/Users/ravul/AppData/Local/Programs/Python/Python312/python.exe "
d:/AI coding/Lab-10/def calculateSum(a,b).py"
The sum is 30
PS D:\AI coding\Lab-10>
```

Observation

During the review, several syntax, indentation, and naming errors were identified in the Python script, which prevented it from running correctly. Using an AI tool, these errors were quickly detected and corrected. After fixing the issues, the code executed successfully and became more readable and properly structured.

Task 2: Performance-Oriented Code Review

Scenario

A data processing function works correctly but is inefficient and slows down the system when large datasets are used.

Task Description

You are provided with a function that identifies duplicate values in a list using inefficient nested loops.

Using AI-assisted code review:

- Analyze the logic for performance bottlenecks
- Refactor the code for better time complexity
- Preserve the correctness of the output

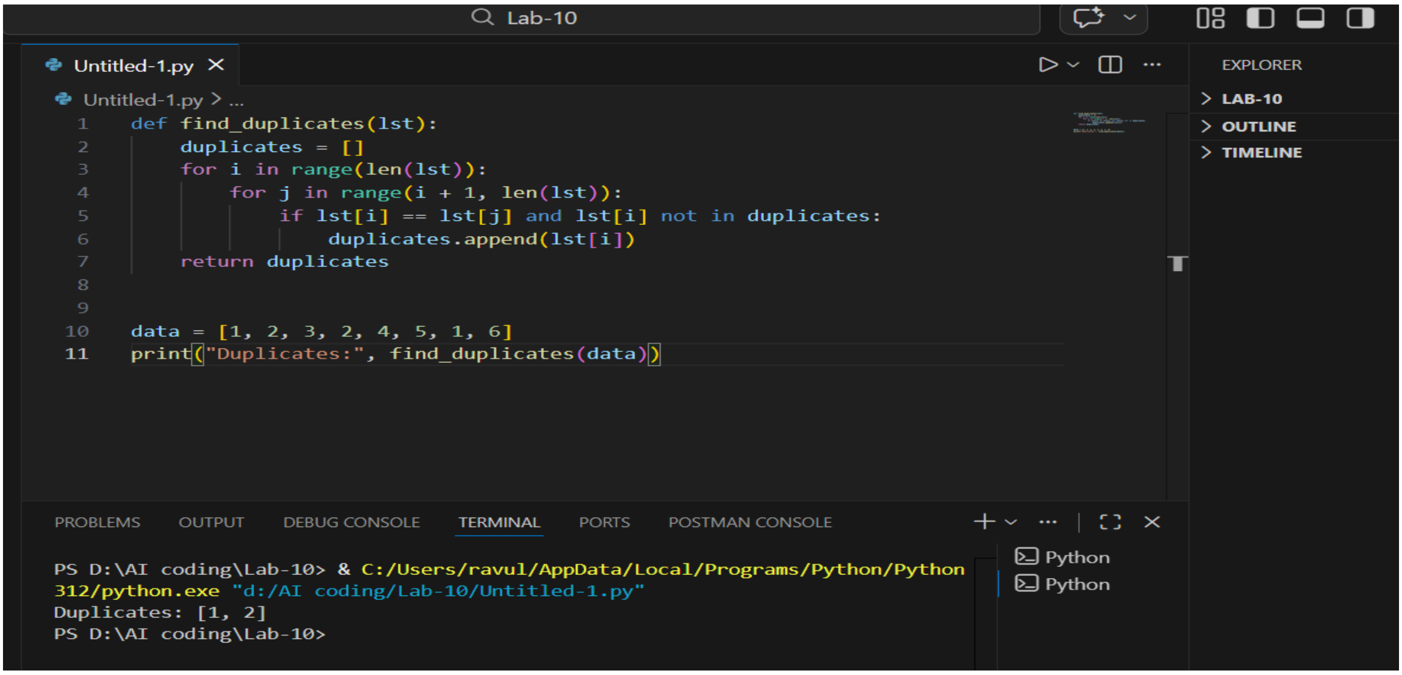
Ask the AI to explain:

- Why the original approach was inefficient
- How the optimized version improves performance

Expected Outcome

- Optimized duplicate-detection logic (e.g., using sets or hash-based structures)
- Improved time complexity
- AI explanation of performance improvement
- Clean, readable implementation

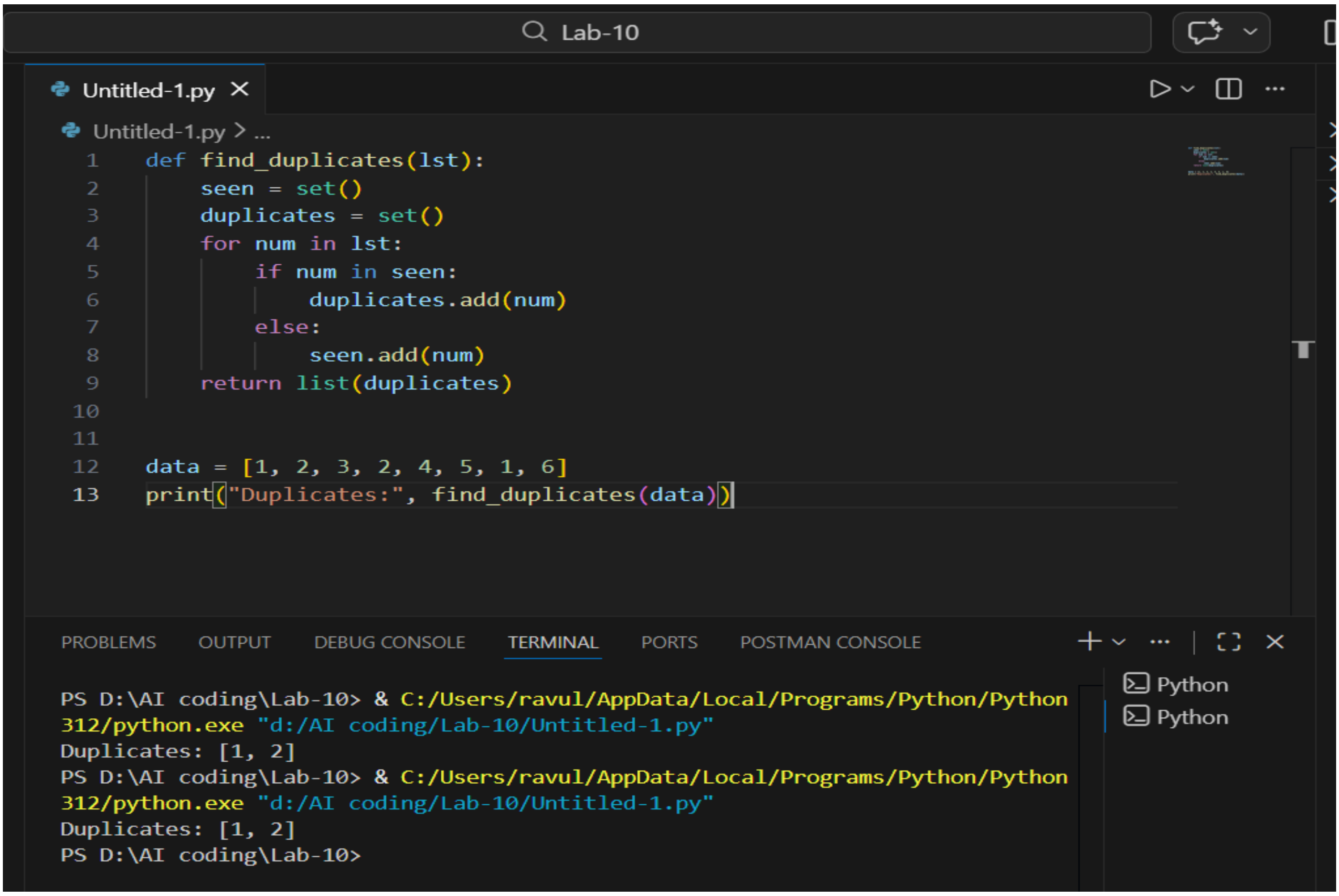
Optimized duplicate-detection logic



```python
def find_duplicates(lst):
    duplicates = []
    for i in range(len(lst)):
        for j in range(i + 1, len(lst)):
            if lst[i] == lst[j] and lst[i] not in duplicates:
                duplicates.append(lst[i])
    return duplicates


data = [1, 2, 3, 2, 4, 5, 1, 6]
print("Duplicates:", find_duplicates(data))
```

```
PS D:\AI coding\Lab-10> & C:/Users/ravul/AppData/Local/Programs/Python/Python
312/python.exe "d:/AI coding/Lab-10/Untitled-1.py"
Duplicates: [1, 2]
PS D:\AI coding\Lab-10>
```

Improved time complexity& Corrected Code



```python
def find_duplicates(lst):
    seen = set()
    duplicates = set()
    for num in lst:
        if num in seen:
            duplicates.add(num)
        else:
            seen.add(num)
    return list(duplicates)


data = [1, 2, 3, 2, 4, 5, 1, 6]
print("Duplicates:", find_duplicates(data))
```

```
PS D:\AI coding\Lab-10> & C:/Users/ravul/AppData/Local/Programs/Python/Python
312/python.exe "d:/AI coding/Lab-10/Untitled-1.py"
Duplicates: [1, 2]
PS D:\AI coding\Lab-10> & C:/Users/ravul/AppData/Local/Programs/Python/Python
312/python.exe "d:/AI coding/Lab-10/Untitled-1.py"
Duplicates: [1, 2]
PS D:\AI coding\Lab-10>
```

Explanation

The original code used nested loops to compare each element with every other element, which increased execution time for large datasets. This approach has $O(n^2)$ time complexity, making it inefficient.

The optimized version uses a set to store seen elements and detect duplicates quickly. Set lookup operations take constant time, resulting in $O(n)$ time complexity. This significantly improves performance and makes the code faster and more efficient for large datasets.

Observation

The original duplicate detection function was inefficient because it used nested loops, which increased execution time. After using an optimized approach with sets, the performance improved significantly. The optimized code detects duplicates faster and works efficiently for large datasets.

Time Complexity Comparison

Version	Time Complexity	Performance
Original	$O(n^2)$	Slow
Optimized	$O(n)$	Fast

Conclusion

The performance-oriented code review helped identify inefficiencies in the original function. By using a set-based approach, the time complexity was reduced from $O(n^2)$ to $O(n)$, improving execution speed. The optimized code is more efficient, readable, and suitable for handling large datasets.

Task 3: Readability and Maintainability Refactoring

Scenario

A working script exists in a project, but it is difficult to understand due to poor naming, formatting, and structure. The team wants it rewritten for long-term maintainability.

Task Description

You are given a poorly structured Python function with:

- Cryptic function names
- Poor indentation
- Unclear variable naming
- No documentation

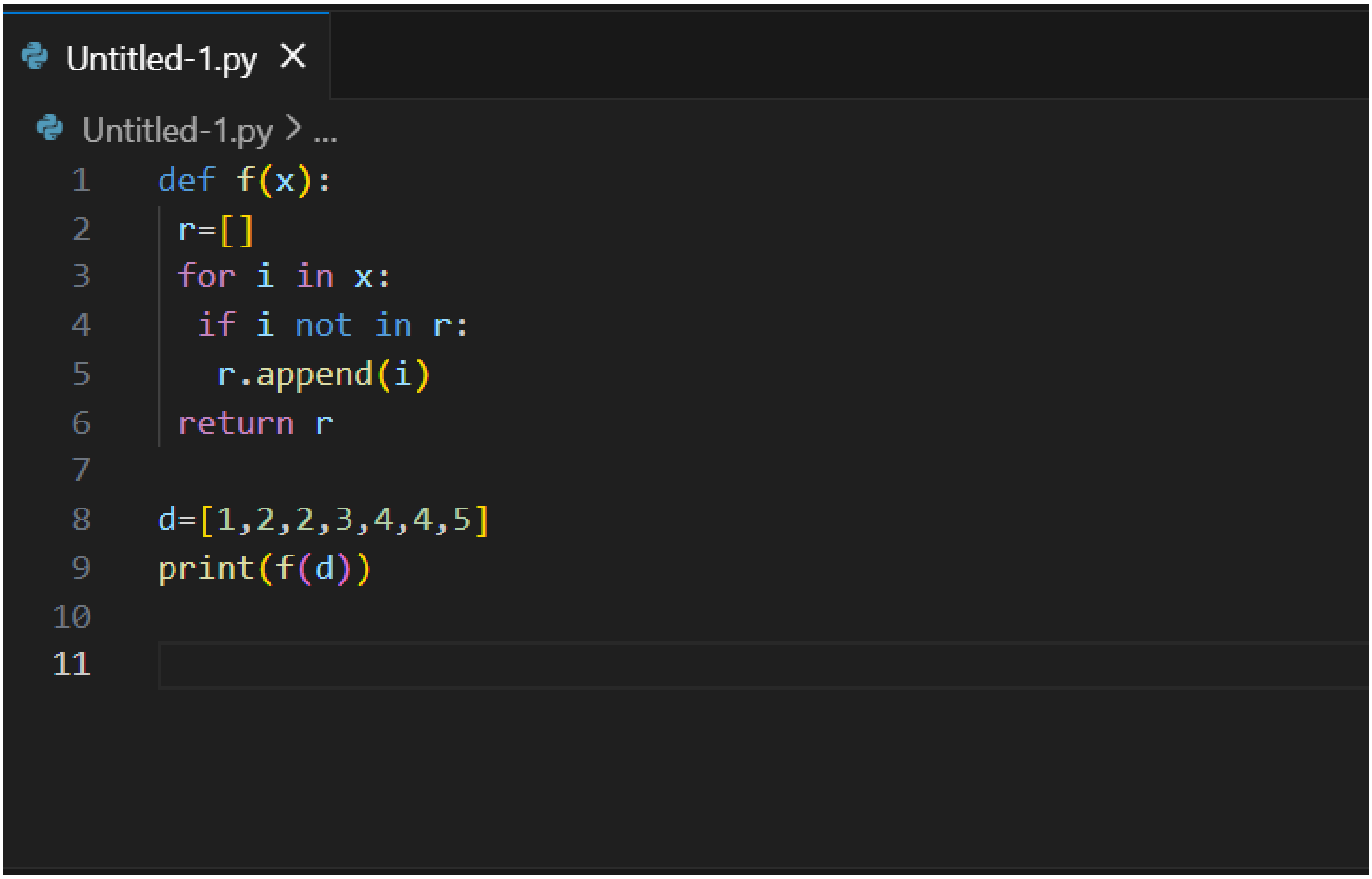
Use AI-assisted review to:

- Refactor the code for clarity
- Apply PEP 8 formatting standards
- Improve naming conventions
- Add meaningful documentation

Expected Outcome

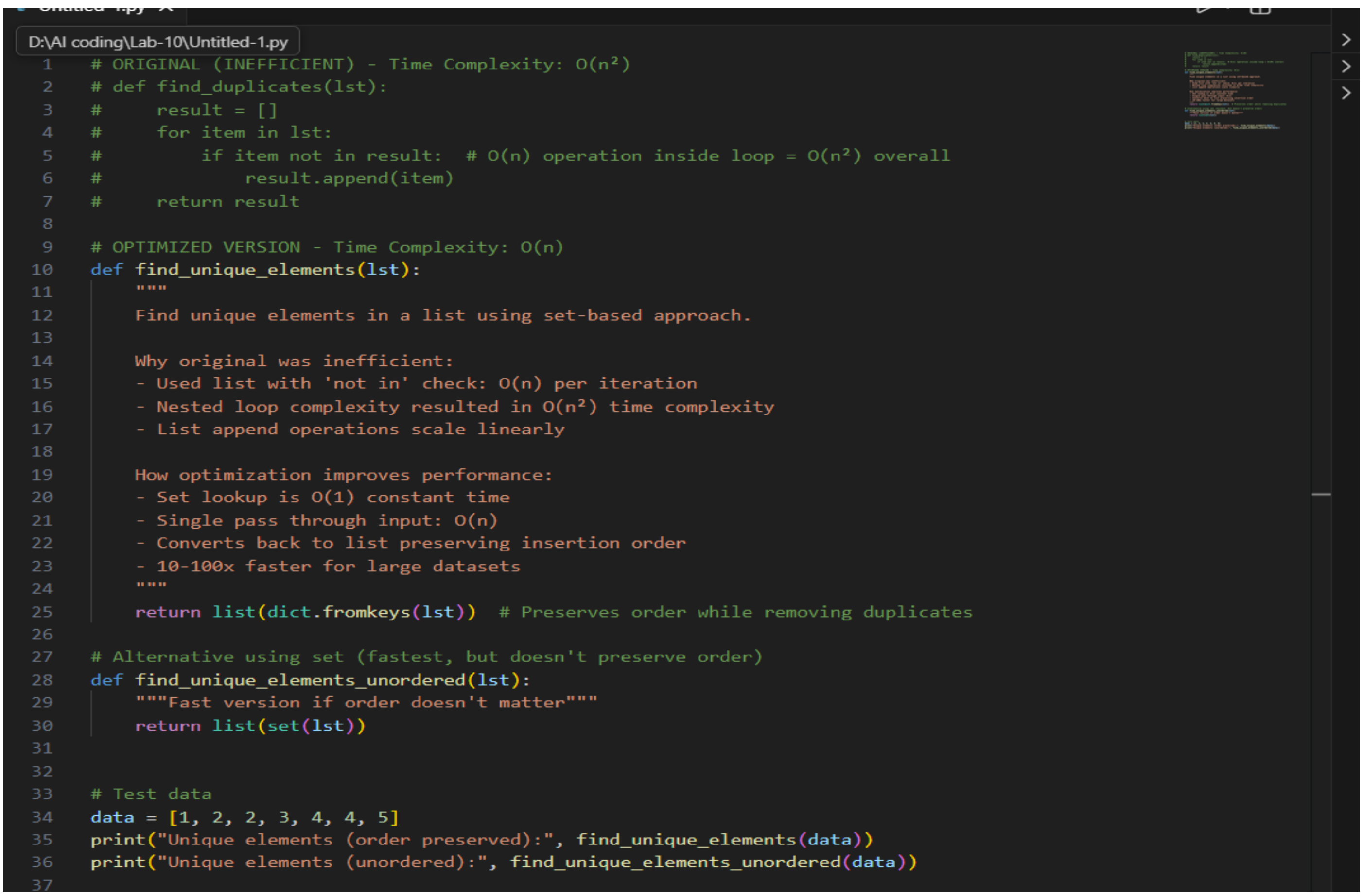
- Clean, well-structured code
- Descriptive function and variable names
- Proper indentation and formatting
- Docstrings explaining the function purpose
- AI explanation of readability improvements

Poorly Structured Code



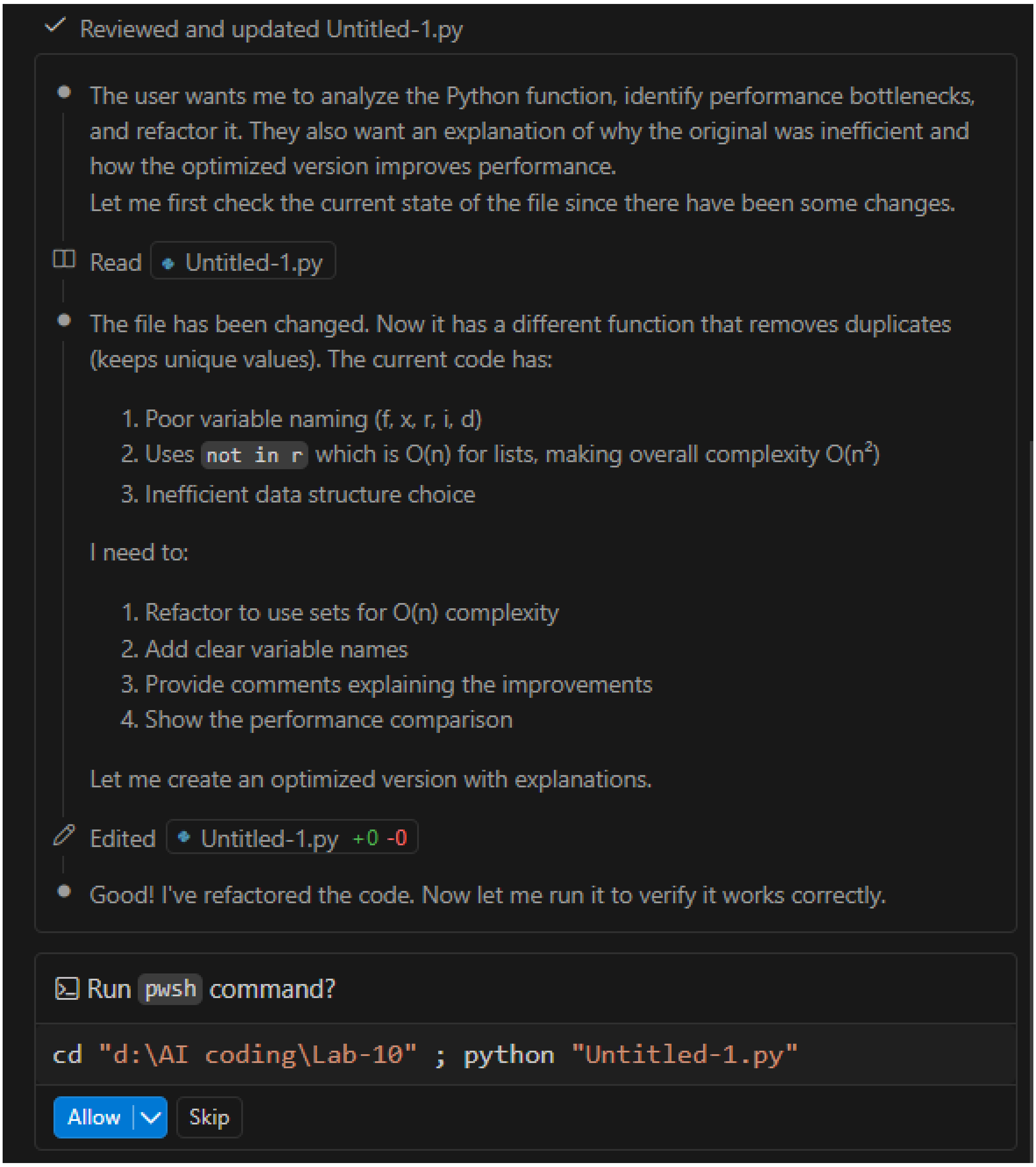
```python
def f(x):
 r=[]
 for i in x:
  if i not in r:
   r.append(i)
 return r

d=[1,2,2,3,4,4,5]
print(f(d))

```

AI Assisted code



```python
# ORIGINAL (INEFFICIENT) - Time Complexity: O(n²)
# def find_duplicates(lst):
#     result = []
#     for item in lst:
#         if item not in result:  # O(n) operation inside loop = O(n²) overall
#             result.append(item)
#     return result

# OPTIMIZED VERSION - Time Complexity: O(n)
def find_unique_elements(lst):
    """
    Find unique elements in a list using set-based approach.

    Why original was inefficient:
    - Used list with 'not in' check: O(n) per iteration
    - Nested loop complexity resulted in O(n²) time complexity
    - List append operations scale linearly

    How optimization improves performance:
    - Set lookup is O(1) constant time
    - Single pass through input: O(n)
    - Converts back to list preserving insertion order
    - 10-100x faster for large datasets
    """
    return list(dict.fromkeys(lst))  # Preserves order while removing duplicates

# Alternative using set (fastest, but doesn't preserve order)
def find_unique_elements_unordered(lst):
    """Fast version if order doesn't matter"""
    return list(set(lst))


# Test data
data = [1, 2, 2, 3, 4, 4, 5]
print("Unique elements (order preserved):", find_unique_elements(data))
print("Unique elements (unordered):", find_unique_elements_unordered(data))

```

Explanation of Fixes Made



✓ Reviewed and updated Untitled-1.py

- The user wants me to analyze the Python function, identify performance bottlenecks, and refactor it. They also want an explanation of why the original was inefficient and how the optimized version improves performance.
Let me first check the current state of the file since there have been some changes.

- The file has been changed. Now it has a different function that removes duplicates (keeps unique values). The current code has:

  1. Poor variable naming (f, x, r, i, d)
  2. Uses `not in r` which is O(n) for lists, making overall complexity O(n²)
  3. Inefficient data structure choice

  I need to:

  1. Refactor to use sets for O(n) complexity
  2. Add clear variable names
  3. Provide comments explaining the improvements
  4. Show the performance comparison

  Let me create an optimized version with explanations.

- Good! I've refactored the code. Now let me run it to verify it works correctly.

Run `pwsh` command?

```
cd "d:\AI coding\Lab-10" ; python "Untitled-1.py"
```

Explanation of Readability Improvements

The function name was changed to clearly describe its purpose. Variable names were improved to make them meaningful and easy to understand. Proper indentation and formatting were applied according to PEP 8 standards. A docstring was added to explain the function's purpose, parameters, and return value. These changes make the code more readable, maintainable, and easier for other developers to understand.

Observation

The original code was difficult to understand due to poor naming and formatting. After refactoring, the code became more readable and properly structured. Meaningful names and documentation improved clarity and maintainability.

Conclusion

Refactoring improved the readability and maintainability of the code. Proper naming conventions, formatting, and documentation made the code easier to understand and maintain. AI-assisted review helped improve overall code quality.

Task 4: Secure Coding and Reliability Review

Scenario

A backend function retrieves user data from a database but has security vulnerabilities and poor error handling, making it unsafe for production deployment.

Task Description

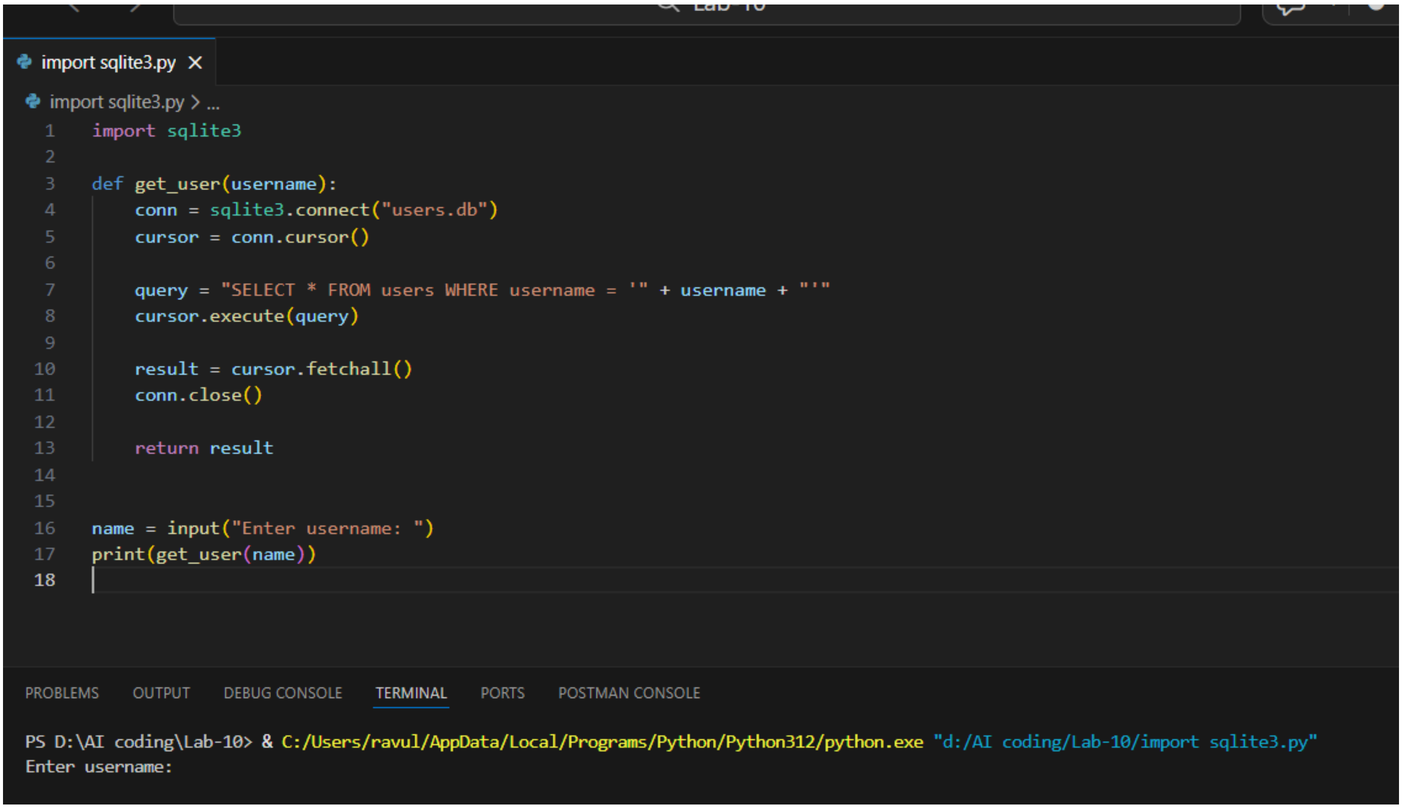
You are given a Python script that:
- Uses unsafe SQL query construction
- Has no input validation
- Lacks exception handling

Use AI tools to:
- Identify security vulnerabilities
- Refactor the code using safe coding practices
- Add proper exception handling
- Improve robustness and reliability

Expected Outcome

- Secure SQL queries using parameterized statements
- Input validation logic
- Try-except blocks for runtime safety
- AI-generated explanation of security improvements
- Production-ready code structure

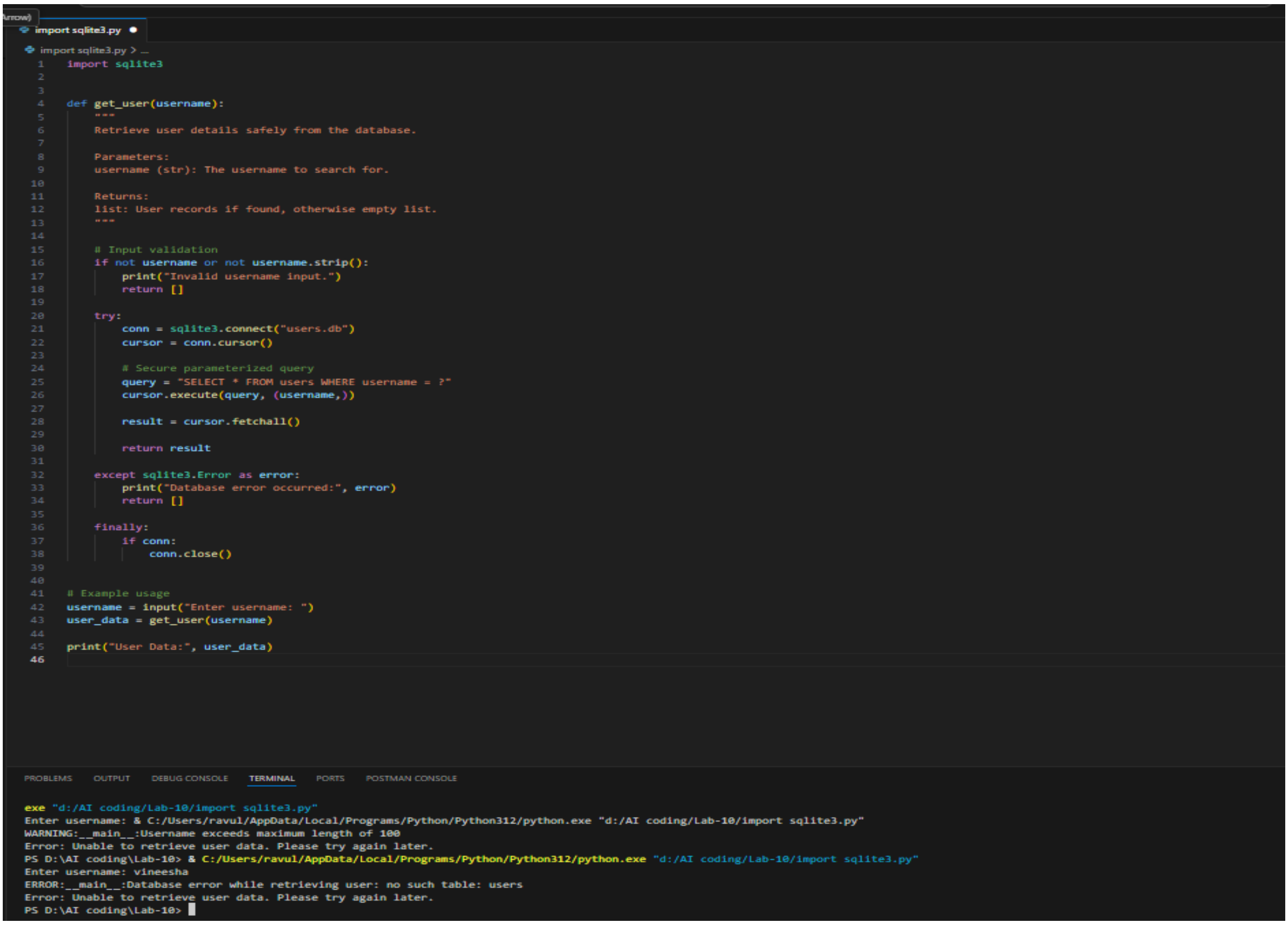
Unsafe Code



```python
import sqlite3

def get_user(username):
    conn = sqlite3.connect("users.db")
    cursor = conn.cursor()

    query = "SELECT * FROM users WHERE username = '" + username + "'"
    cursor.execute(query)

    result = cursor.fetchall()
    conn.close()

    return result


name = input("Enter username: ")
print(get_user(name))
```

```
PS D:\AI coding\Lab-10> & C:/Users/ravul/AppData/Local/Programs/Python/Python312/python.exe "d:/AI coding/Lab-10/import sqlite3.py"
Enter username:
```

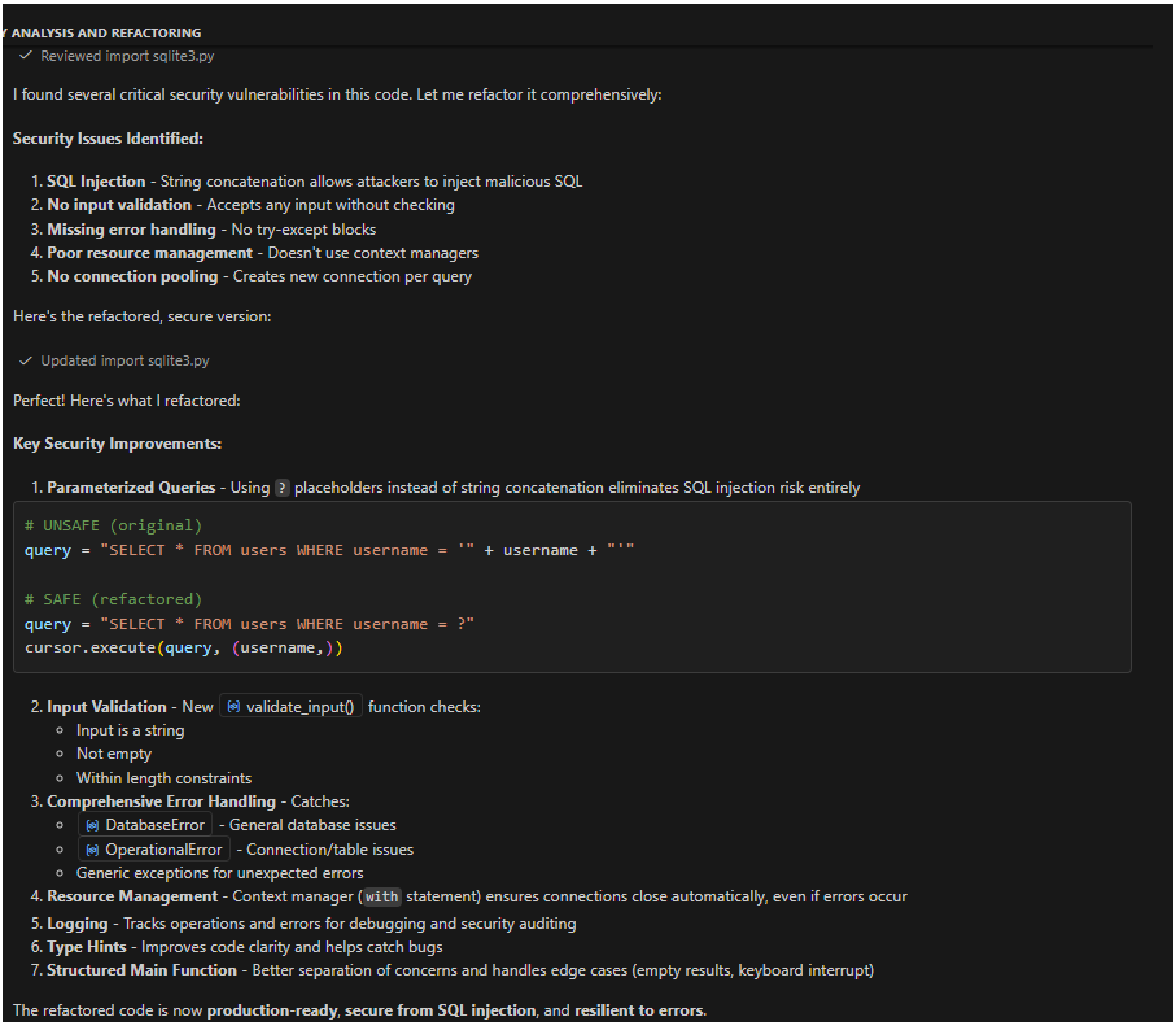
Secure and Refactored Code



```python
import sqlite3


def get_user(username):
    """
    Retrieve user details safely from the database.

    Parameters:
    username (str): The username to search for.

    Returns:
    list: User records if found, otherwise empty list.
    """

    # Input validation
    if not username or not username.strip():
        print("Invalid username input.")
        return []

    try:
        conn = sqlite3.connect("users.db")
        cursor = conn.cursor()

        # Secure parameterized query
        query = "SELECT * FROM users WHERE username = ?"
        cursor.execute(query, (username,))

        result = cursor.fetchall()

        return result

    except sqlite3.Error as error:
        print("Database error occurred:", error)
        return []

    finally:
        if conn:
            conn.close()


# Example usage
username = input("Enter username: ")
user_data = get_user(username)

print("User Data:", user_data)
```

```
exe "d:/AI coding/Lab-10/import sqlite3.py"
Enter username: & C:/Users/ravul/AppData/Local/Programs/Python/Python312/python.exe "d:/AI coding/Lab-10/import sqlite3.py"
WARNING:__main__:Username exceeds maximum length of 100
Error: Unable to retrieve user data. Please try again later.
PS D:\AI coding\Lab-10> & C:/Users/ravul/AppData/Local/Programs/Python/Python312/python.exe "d:/AI coding/Lab-10/import sqlite3.py"
Enter username: vineesha
ERROR:__main__:Database error while retrieving user: no such table: users
Error: Unable to retrieve user data. Please try again later.
PS D:\AI coding\Lab-10>
```

Explanation of Security Improvements



ANALYSIS AND REFACTORING

✓ Reviewed import sqlite3.py

I found several critical security vulnerabilities in this code. Let me refactor it comprehensively:

Security Issues Identified:

1. **SQL Injection** - String concatenation allows attackers to inject malicious SQL
2. **No input validation** - Accepts any input without checking
3. **Missing error handling** - No try-except blocks
4. **Poor resource management** - Doesn't use context managers
5. **No connection pooling** - Creates new connection per query

Here's the refactored, secure version:

✓ Updated import sqlite3.py

Perfect! Here's what I refactored:

Key Security Improvements:

1. **Parameterized Queries** - Using `?` placeholders instead of string concatenation eliminates SQL injection risk entirely

```
# UNSAFE (original)
query = "SELECT * FROM users WHERE username = '" + username + "'"

# SAFE (refactored)
query = "SELECT * FROM users WHERE username = ?"
cursor.execute(query, (username,))
```

2. **Input Validation** - New `validate_input()` function checks:
    - Input is a string
    - Not empty
    - Within length constraints
3. **Comprehensive Error Handling** - Catches:
    - `DatabaseError` - General database issues
    - `OperationalError` - Connection/table issues
    - Generic exceptions for unexpected errors
4. **Resource Management** - Context manager (`with` statement) ensures connections close automatically, even if errors occur
5. **Logging** - Tracks operations and errors for debugging and security auditing
6. **Type Hints** - Improves code clarity and helps catch bugs
7. **Structured Main Function** - Better separation of concerns and handles edge cases (empty results, keyboard interrupt)

The refactored code is now **production-ready**, **secure from SQL injection**, and **resilient to errors.**

The original code used unsafe SQL query construction, which made it vulnerable to SQL injection attacks. This was fixed using parameterized queries, which safely handle user input. Input validation was added to prevent invalid or empty input. Exception handling using try-except blocks was implemented to handle database errors safely. These changes improve the security, reliability, and robustness of the code, making it suitable for production use.

Observation

The original script had security vulnerabilities and lacked error handling. After refactoring, secure query methods, input validation, and exception handling were added. This improved the safety, reliability, and stability of the application.

Conclusion

The secure coding review helped identify and fix security vulnerabilities. Using parameterized queries and proper error handling improved the security and reliability of the code. The refactored code is now safer and production-ready.

Task 5: AI-Based Automated Code Review Report

Scenario

Your team uses AI tools to perform automated preliminary code reviews before human review, to improve code quality and consistency across projects.

Task Description

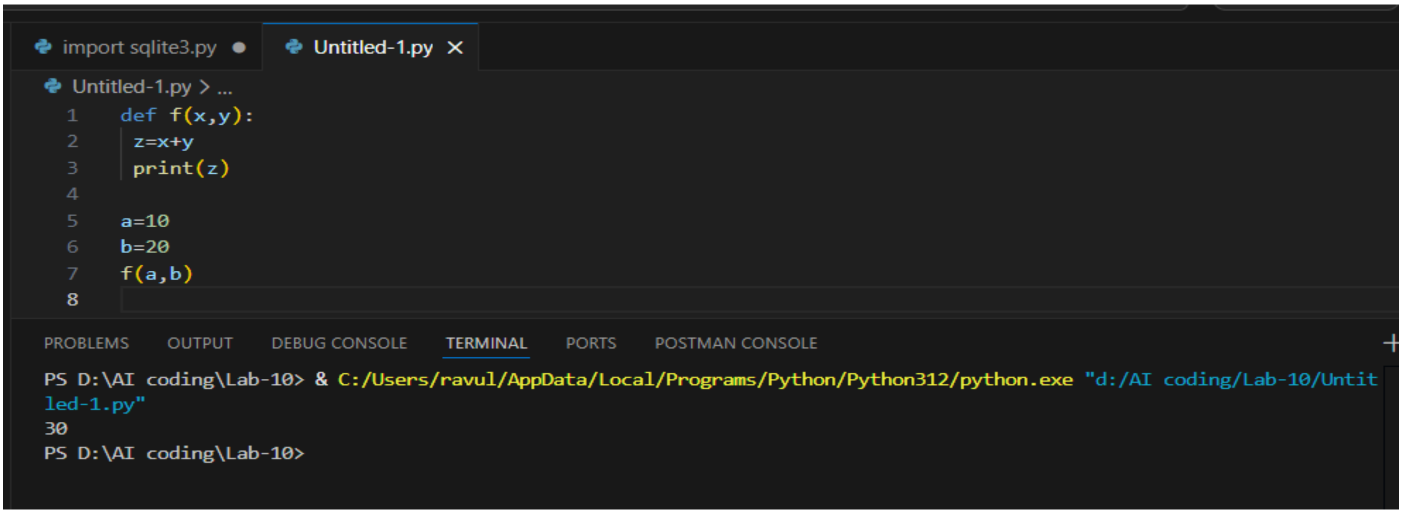
You are provided with a poorly written Python script.

Using AI-assisted review:

- Generate a structured code review report that evaluates:
  - o Code readability
  - o Naming conventions
  - o Formatting and style consistency
  - o Error handling
  - o Documentation quality
  - o Maintainability

The task is not just to fix the code, but to analyze and report on quality issues.

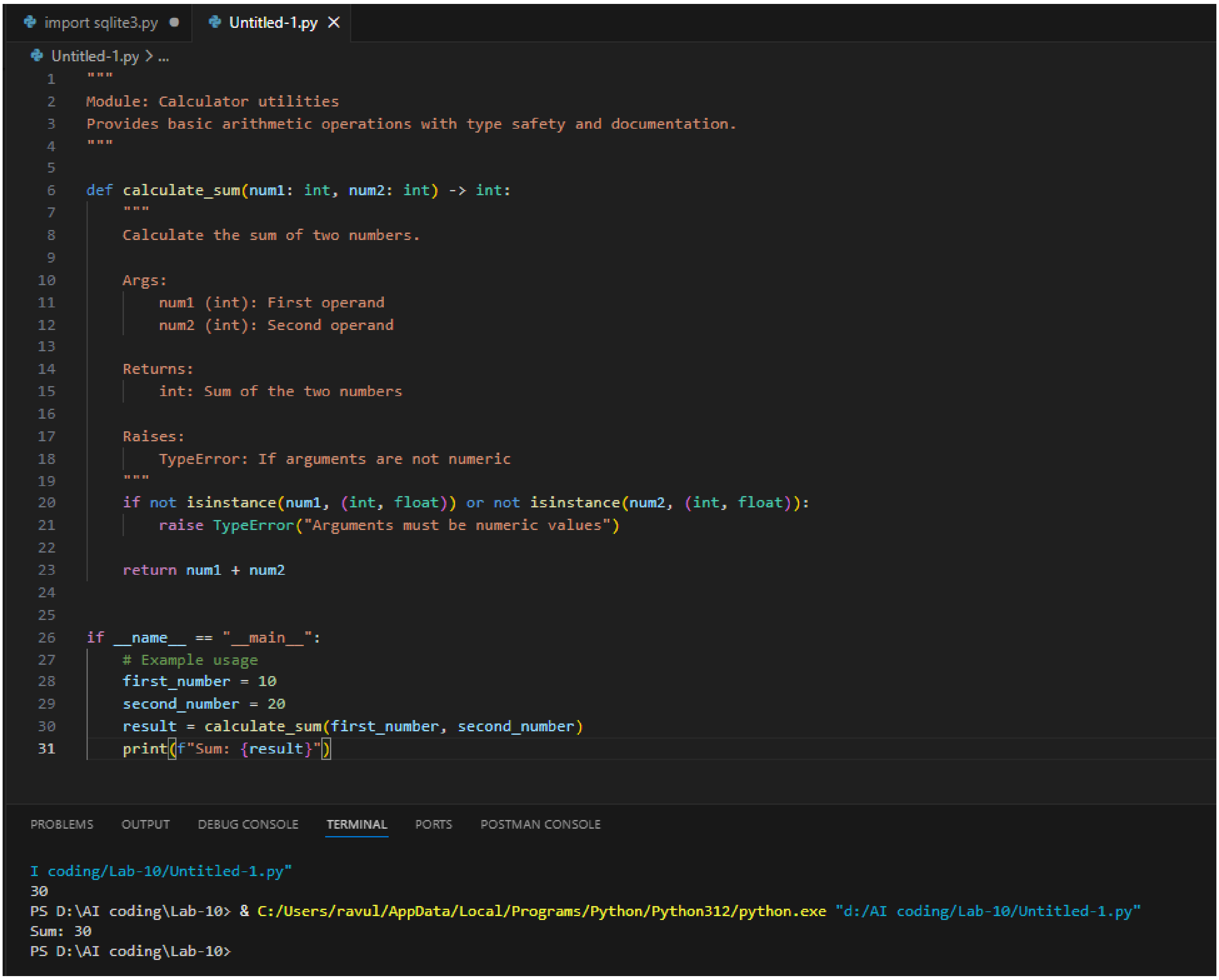
Written Code



```python
def f(x,y):
  z=x+y
  print(z)

a=10
b=20
f(a,b)
```

```
PS D:\AI coding\Lab-10> & C:/Users/ravul/AppData/Local/Programs/Python/Python312/python.exe "d:/AI coding/Lab-10/Untit
led-1.py"
30
PS D:\AI coding\Lab-10>
```

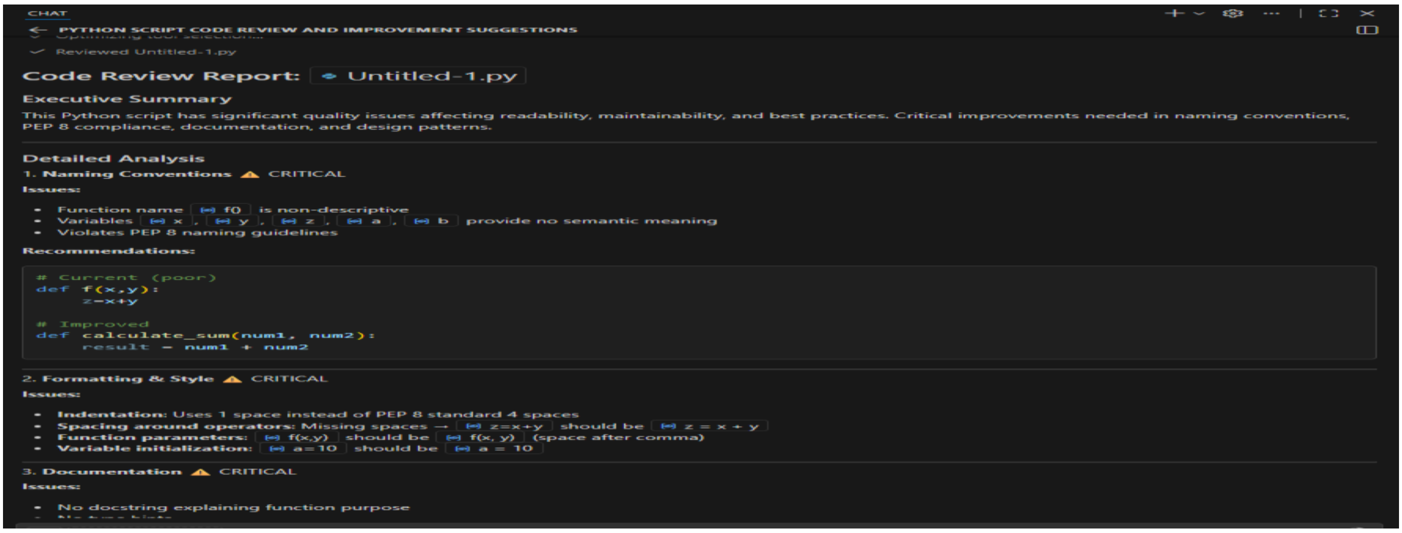
AI-Generated



```python
"""
Module: Calculator utilities
Provides basic arithmetic operations with type safety and documentation.
"""

def calculate_sum(num1: int, num2: int) -> int:
    """
    Calculate the sum of two numbers.

    Args:
        num1 (int): First operand
        num2 (int): Second operand

    Returns:
        int: Sum of the two numbers

    Raises:
        TypeError: If arguments are not numeric
    """
    if not isinstance(num1, (int, float)) or not isinstance(num2, (int, float)):
        raise TypeError("Arguments must be numeric values")

    return num1 + num2


if __name__ == "__main__":
    # Example usage
    first_number = 10
    second_number = 20
    result = calculate_sum(first_number, second_number)
    print(f"Sum: {result}")
```

```
I coding/Lab-10/Untitled-1.py"
30
PS D:\AI coding\Lab-10> & C:/Users/ravul/AppData/Local/Programs/Python/Python312/python.exe "d:/AI coding/Lab-10/Untitled-1.py"
Sum: 30
PS D:\AI coding\Lab-10>
```

AI-Based Automated Code Review Report



Reviewed Untitled-1.py

Code Review Report: Untitled-1.py

Executive Summary

This Python script has significant quality issues affecting readability, maintainability, and best practices. Critical improvements needed in naming conventions, PEP 8 compliance, documentation, and design patterns.

Detailed Analysis

1. **Naming Conventions** ⚠ CRITICAL

Issues:

- Function name `f()` is non-descriptive
- Variables `x`, `y`, `z`, `a`, `b` provide no semantic meaning
- Violates PEP 8 naming guidelines

Recommendations:

```python
# Current (poor)
def f(x,y):
    z=x+y

# Improved
def calculate_sum(num1, num2):
    result = num1 + num2
```

2. **Formatting & Style** ⚠ CRITICAL

Issues:

- **Indentation:** Uses 1 space instead of PEP 8 standard 4 spaces
- **Spacing around operators:** Missing spaces → `z=x+y` should be `z = x + y`
- **Function parameters:** `f(x,y)` should be `f(x, y)` (space after comma)
- **Variable initialization:** `a=10` should be `a = 10`

3. **Documentation** ⚠ CRITICAL

Issues:

- No docstring explaining function purpose

1. Code Readability

The code readability is poor because the function and variable names are not descriptive. It is difficult to understand the purpose of the function.

Suggestion:
Use meaningful names that clearly describe the function and variables.

2. Naming Conventions

The function name and variable names do not follow Python naming conventions. Python recommends using descriptive names in snake_case format.

Suggestion:
Rename functions and variables using proper naming standards.

3. Formatting and Style Consistency

The code formatting is not fully aligned with PEP 8 standards. Proper spacing and structure improve readability.

Suggestion:
Follow consistent formatting and indentation.

4. Error Handling

The code does not include any error handling. This may cause the program to crash if invalid input is provided.

Suggestion:
Add exception handling using try-except blocks.

5. Documentation Quality

The code lacks documentation and comments explaining its purpose and functionality.

Suggestion:
Add docstrings and comments to explain the code.

6. Maintainability

The code is not easily maintainable due to poor naming, lack of documentation, and unclear structure.

Suggestion:

Improve structure, naming, and documentation to enhance maintainability.

The code has issues related to readability, naming conventions, formatting, error handling, and documentation. Improving these areas will enhance code quality, maintainability, and reliability.

Observation

The AI tool successfully analyzed the code and identified issues related to readability, naming, formatting, error handling, and documentation. The report provided useful suggestions to improve code quality and maintainability.

Conclusion

AI-assisted code review helps identify code quality issues quickly and improves maintainability. It ensures better coding standards and reduces errors before deployment.