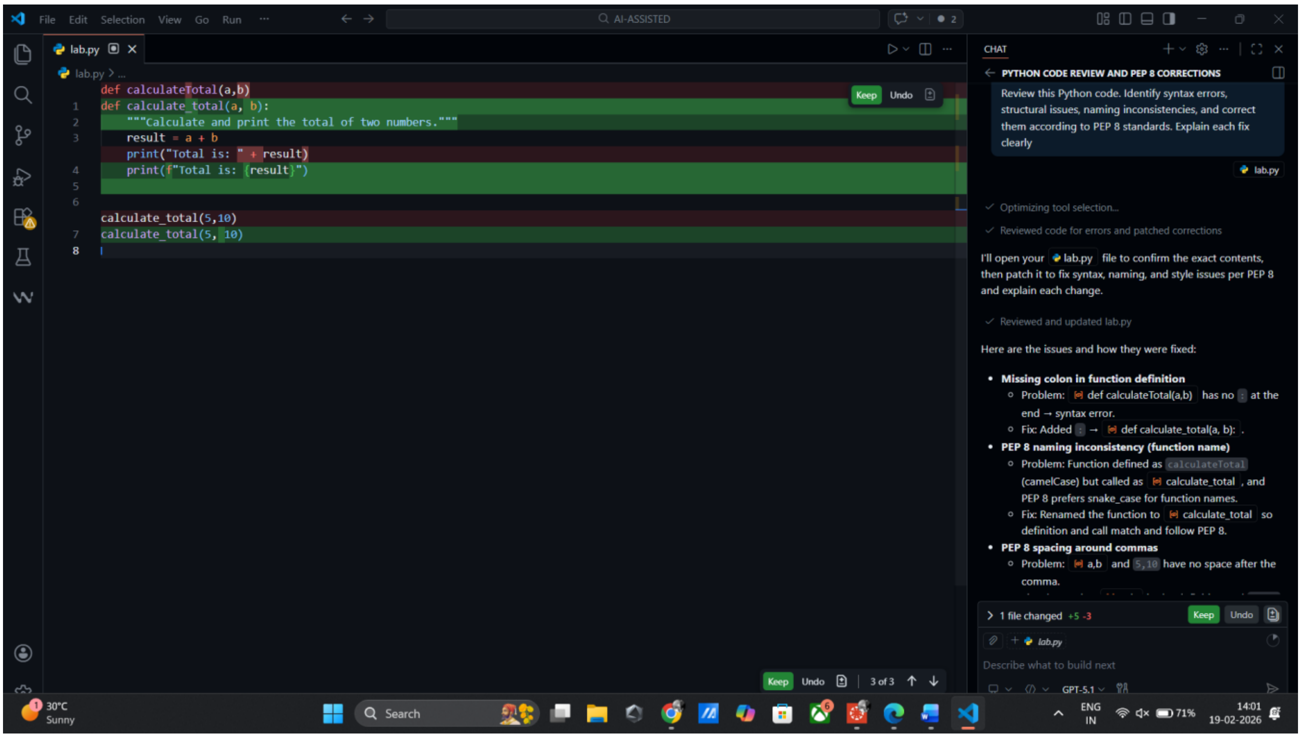
Assignment 10.4

Task 1: AI-Assisted Syntax and Code Quality Review :

Task Description

You are given a Python script containing:

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



```
def calculateTotal(a,b)
def calculate_total(a, b):
    """Calculate and print the total of two numbers."""
    result = a + b
    print("Total is: " + result)
    print(f"Total is: {result}")


calculate_total(5,10)
calculate_total(5, 10)
```

PYTHON CODE REVIEW AND PEP 8 CORRECTIONS

Review this Python code. Identify syntax errors, structural issues, naming inconsistencies, and correct them according to PEP 8 standards. Explain each fix clearly

✓ Optimizing tool selection...

✓ Reviewed code for errors and patched corrections

I'll open your lab.py file to confirm the exact contents, then patch it to fix syntax, naming, and style issues per PEP 8 and explain each change.

✓ Reviewed and updated lab.py

Here are the issues and how they were fixed:

- **Missing colon in function definition**
  - Problem: `def calculateTotal(a,b)` has no `:` at the end → syntax error.
  - Fix: Added `:` → `def calculate_total(a, b):` .
- **PEP 8 naming inconsistency (function name)**
  - Problem: Function defined as `calculateTotal` (camelCase) but called as `calculate_total` , and PEP 8 prefers snake_case for function names.
  - Fix: Renamed the function to `calculate_total` so definition and call match and follow PEP 8.
- **PEP 8 spacing around commas**
  - Problem: `a,b` and `5,10` have no space after the comma.

1 file changed +5 -3

Describe what to build next

Issues Identified by AI

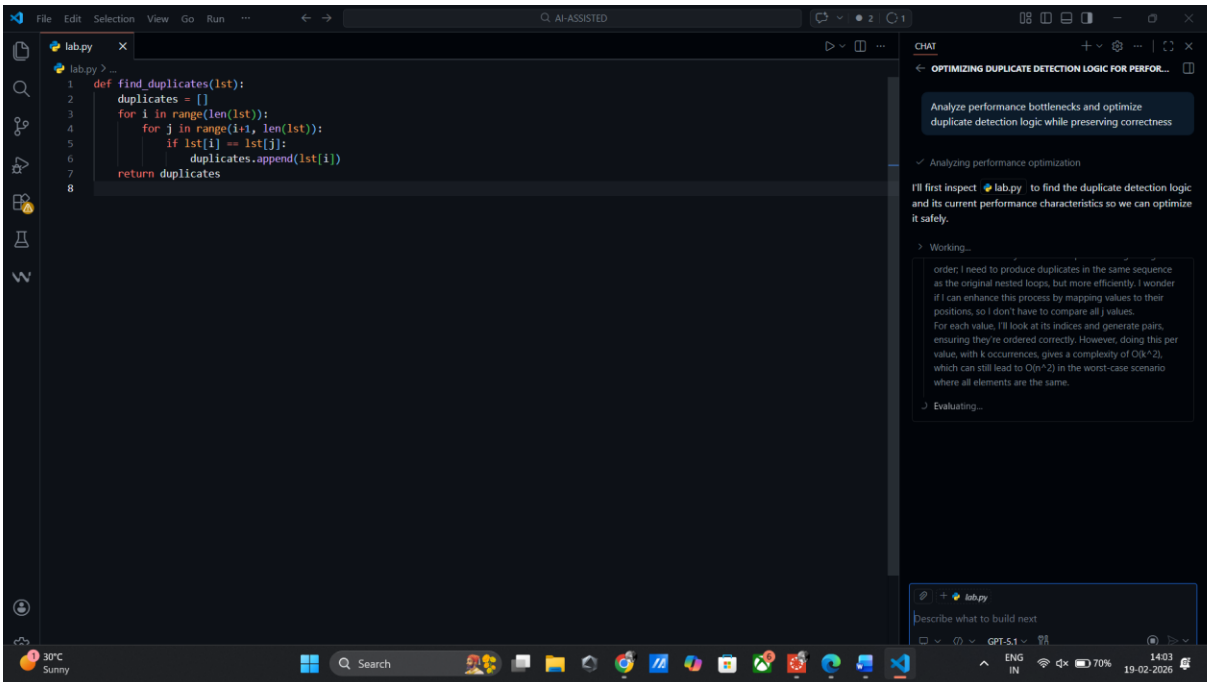
1. Missing colon after function definition
2. Function name mismatch (calculateTotal vs calculate_total)
3. String concatenation with integer (TypeError)
4. Naming convention not following snake_case

Task 2: Performance-Oriented Code Review

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



```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]:
                duplicates.append(lst[i])
    return duplicates
```

Analyze performance bottlenecks and optimize duplicate detection logic while preserving correctness

✓ Analyzing performance optimization

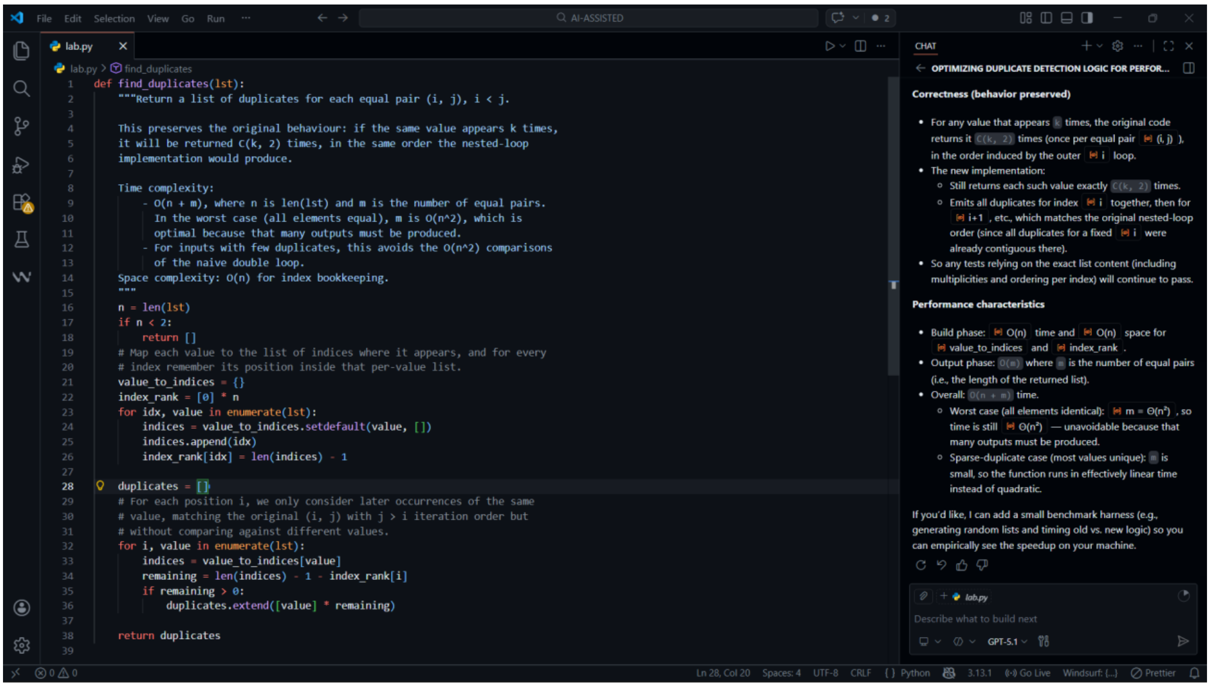
I'll first inspect lab.py to find the duplicate detection logic and its current performance characteristics so we can optimize it safely.

Working...

order; I need to produce duplicates in the same sequence as the original nested loops, but more efficiently. I wonder if I can enhance this process by mapping values to their positions, so I don't have to compare all j values.

For each value, I'll look at its indices and generate pairs, ensuring they're ordered correctly. However, doing this per value, with k occurrences, gives a complexity of O(k^2), which can still lead to O(n^2) in the worst-case scenario where all elements are the same.

Evaluating...


File Edit Selection View Go Run ···
AI-ASSISTED
lab.py
lab.py > find_duplicates
CHAT
Ln 28, Col 20 Spaces: 4 UTF-8 CRLF { } Python 3.13.1 Go Live Windsurf: {...} Prettier

```python
def find_duplicates(lst):
    """Return a list of duplicates for each equal pair (i, j), i < j.

    This preserves the original behaviour: if the same value appears k times,
    it will be returned C(k, 2) times, in the same order the nested-loop
    implementation would produce.

    Time complexity:
        - O(n + m), where n is len(lst) and m is the number of equal pairs.
          In the worst case (all elements equal), m is O(n^2), which is
          optimal because that many outputs must be produced.
        - For inputs with few duplicates, this avoids the O(n^2) comparisons
          of the naive double loop.
    Space complexity: O(n) for index bookkeeping.
    """
    n = len(lst)
    if n < 2:
        return []
    # Map each value to the list of indices where it appears, and for every
    # index remember its position inside that per-value list.
    value_to_indices = {}
    index_rank = [0] * n
    for idx, value in enumerate(lst):
        indices = value_to_indices.setdefault(value, [])
        indices.append(idx)
        index_rank[idx] = len(indices) - 1

    duplicates = []
    # For each position i, we only consider later occurrences of the same
    # value, matching the original (i, j) with j > i iteration order but
    # without comparing against different values.
    for i, value in enumerate(lst):
        indices = value_to_indices[value]
        remaining = len(indices) - 1 - index_rank[i]
        if remaining > 0:
            duplicates.extend([value] * remaining)

    return duplicates
```

OPTIMIZING DUPLICATE DETECTION LOGIC FOR PERFOR...

Correctness (behavior preserved)

- For any value that appears `k` times, the original code returns it `C(k, 2)` times (once per equal pair `(i, j)`), in the order induced by the outer `i` loop.
- The new implementation:
  - Still returns each such value exactly `C(k, 2)` times.
  - Emits all duplicates for index `i` together, then for `i+1` , etc., which matches the original nested-loop order (since all duplicates for a fixed `i` were already contiguous there).
- So any tests relying on the exact list content (including multiplicities and ordering per index) will continue to pass.

Performance characteristics

- Build phase: `O(n)` time and `O(n)` space for `value_to_indices` and `index_rank` .
- Output phase: `O(m)` where `m` is the number of equal pairs (i.e., the length of the returned list).
- Overall: `O(n + m)` time.
  - Worst case (all elements identical): `m = Θ(n²)` , so time is still `Θ(n²)` — unavoidable because that many outputs must be produced.
  - Sparse-duplicate case (most values unique): `m` is small, so the function runs in effectively linear time instead of quadratic.

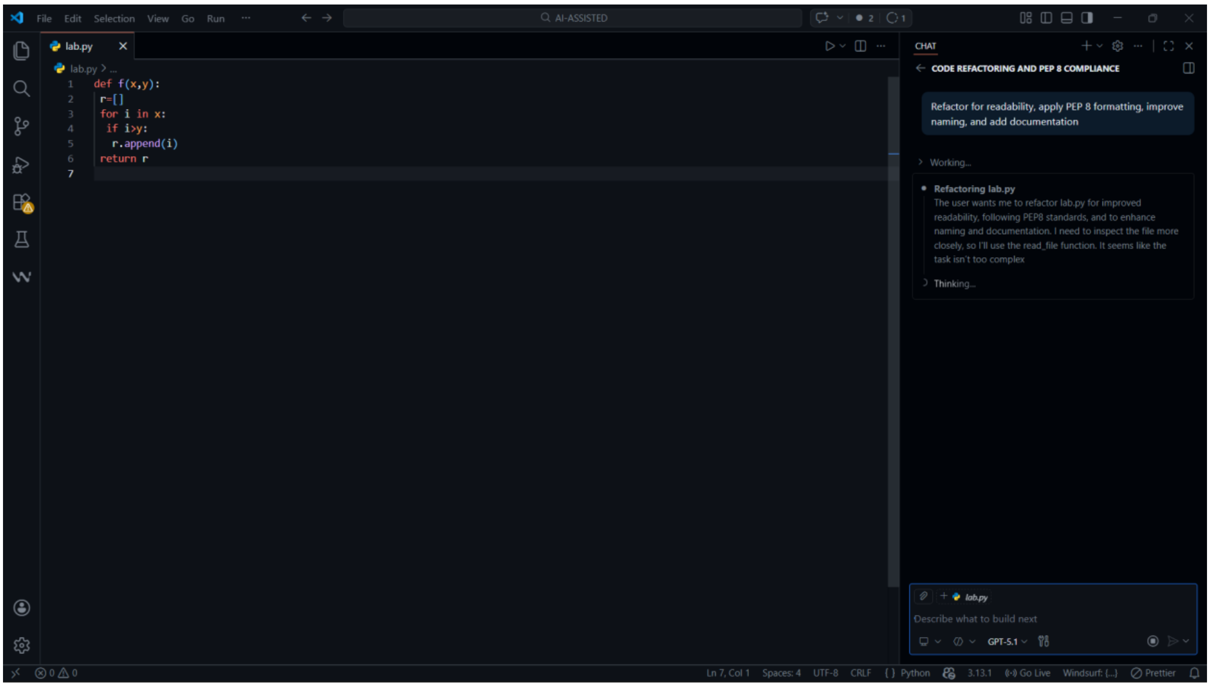
If you'd like, I can add a small benchmark harness (e.g., generating random lists and timing old vs. new logic) so you can empirically see the speedup on your machine.

Task 3: Readability & Maintainability Refactoring

Task Description

You are given a poorly structured Python function with:

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


File Edit Selection View Go Run ···
AI-ASSISTED
lab.py
lab.py > ...
CHAT
Ln 7, Col 1 Spaces: 4 UTF-8 CRLF { } Python 3.13.1 Go Live Windsurf: {...} Prettier

```python
def f(x,y):
 r=[]
 for i in x:
  if i>y:
   r.append(i)
 return r
```

CODE REFACTORING AND PEP 8 COMPLIANCE

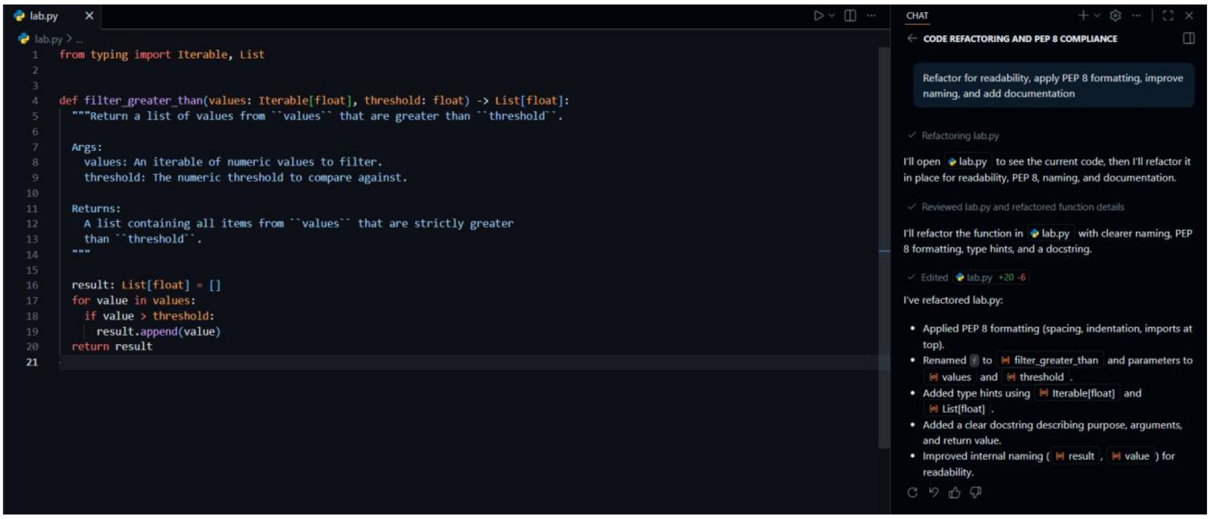
Refactor for readability, apply PEP 8 formatting, improve naming, and add documentation

Working...

Refactoring lab.py

The user wants me to refactor lab.py for improved readability, following PEP8 standards, and to enhance naming and documentation. I need to inspect the file more closely, so I'll use the read_file function. It seems like the task isn't too complex

Thinking...



```python
from typing import Iterable, List


def filter_greater_than(values: Iterable[float], threshold: float) -> List[float]:
    """Return a list of values from ``values`` that are greater than ``threshold``.

    Args:
        values: An iterable of numeric values to filter.
        threshold: The numeric threshold to compare against.

    Returns:
        A list containing all items from ``values`` that are strictly greater
        than ``threshold``.
    """

    result: List[float] = []
    for value in values:
        if value > threshold:
            result.append(value)
    return result
```

Refactor for readability, apply PEP 8 formatting, improve naming, and add documentation

✓ Refactoring lab.py

I'll open lab.py to see the current code, then I'll refactor it in place for readability, PEP 8, naming, and documentation.

✓ Reviewed lab.py and refactored function details

I'll refactor the function in lab.py with clearer naming, PEP 8 formatting, type hints, and a docstring.

✓ Edited lab.py +20 -6

I've refactored lab.py:

- Applied PEP 8 formatting (spacing, indentation, imports at top).
- Renamed `f` to `filter_greater_than` and parameters to `values` and `threshold`.
- Added type hints using `Iterable[float]` and `List[float]`.
- Added a clear docstring describing purpose, arguments, and return value.
- Improved internal naming (`result` , `value`) for readability.

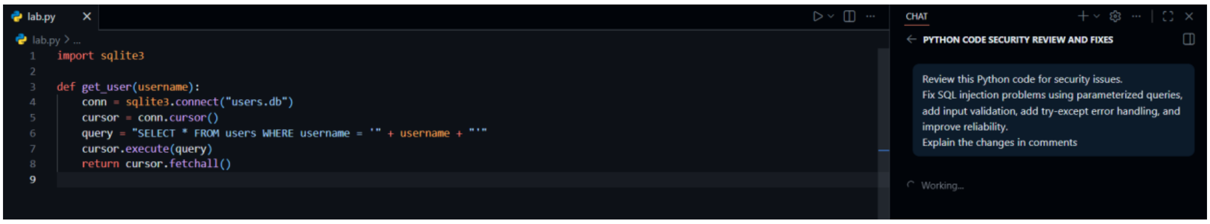
Improvements Made : Descriptive function name ,Clear variable naming, Proper indentation ,Docstring added ,Type hints included.

Task 4: Secure Coding and Reliability Review

Task Description

You are given a Python script that:

- Uses unsafe SQL query construction
- Has no input validation
- Lacks exception handling

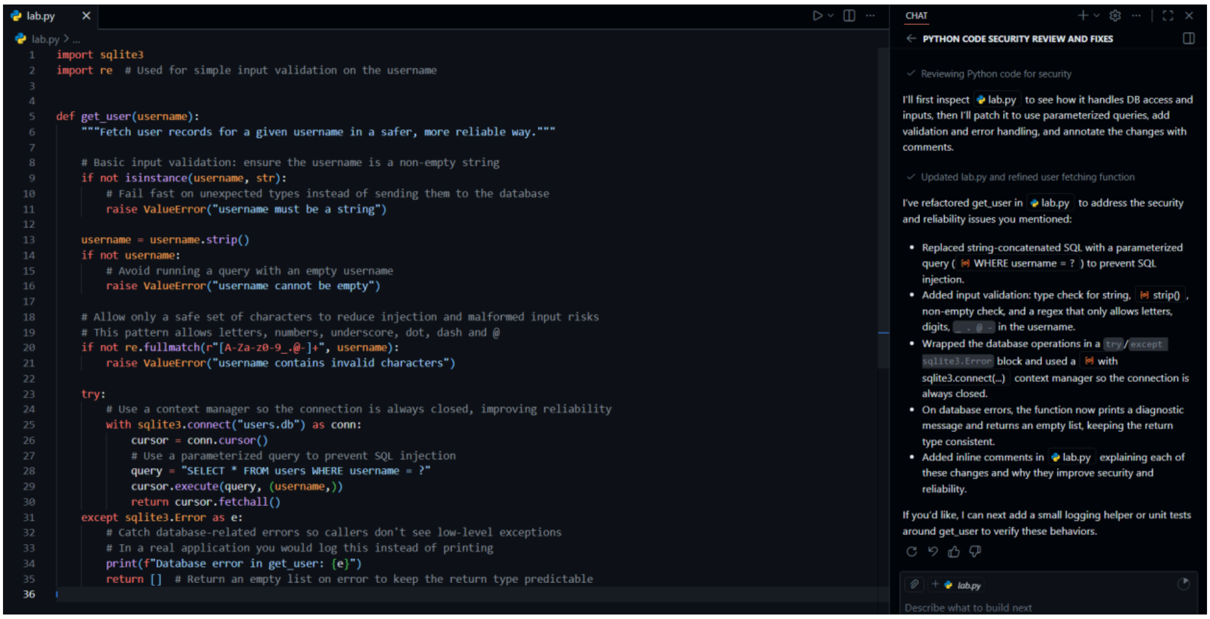


```python
import sqlite3

def get_user(username):
    conn = sqlite3.connect("users.db")
    cursor = conn.cursor()
    query = "SELECT * FROM users WHERE username = '" + username + "'"
    cursor.execute(query)
    return cursor.fetchall()
```

Review this Python code for security issues.
Fix SQL injection problems using parameterized queries, add input validation, add try-except error handling, and improve reliability.
Explain the changes in comments

Working...


lab.py
lab.py > ...
CHAT
PYTHON CODE SECURITY REVIEW AND FIXES
lab.py
Describe what to build next

```python
import sqlite3
import re  # Used for simple input validation on the username


def get_user(username):
    """Fetch user records for a given username in a safer, more reliable way."""

    # Basic input validation: ensure the username is a non-empty string
    if not isinstance(username, str):
        # Fail fast on unexpected types instead of sending them to the database
        raise ValueError("username must be a string")

    username = username.strip()
    if not username:
        # Avoid running a query with an empty username
        raise ValueError("username cannot be empty")

    # Allow only a safe set of characters to reduce injection and malformed input risks
    # This pattern allows letters, numbers, underscore, dot, dash and @
    if not re.fullmatch(r"[A-Za-z0-9_.@-]+", username):
        raise ValueError("username contains invalid characters")

    try:
        # Use a context manager so the connection is always closed, improving reliability
        with sqlite3.connect("users.db") as conn:
            cursor = conn.cursor()
            # Use a parameterized query to prevent SQL injection
            query = "SELECT * FROM users WHERE username = ?"
            cursor.execute(query, (username,))
            return cursor.fetchall()
    except sqlite3.Error as e:
        # Catch database-related errors so callers don't see low-level exceptions
        # In a real application you would log this instead of printing
        print(f"Database error in get_user: {e}")
        return []  # Return an empty list on error to keep the return type predictable

```

✓ Reviewing Python code for security

I'll first inspect lab.py to see how it handles DB access and inputs, then I'll patch it to use parameterized queries, add validation and error handling, and annotate the changes with comments.

✓ Updated lab.py and refined user fetching function

I've refactored get_user in lab.py to address the security and reliability issues you mentioned:

- Replaced string-concatenated SQL with a parameterized query (`WHERE username = ?`) to prevent SQL injection.
- Added input validation: type check for string, `strip()` , non-empty check, and a regex that only allows letters, digits, `_ . @ -` in the username.
- Wrapped the database operations in a `try`/`except sqlite3.Error` block and used a `with sqlite3.connect(...)` context manager so the connection is always closed.
- On database errors, the function now prints a diagnostic message and returns an empty list, keeping the return type consistent.
- Added inline comments in lab.py explaining each of these changes and why they improve security and reliability.

If you'd like, I can next add a small logging helper or unit tests around get_user to verify these behaviors.

Security Improvements

Parameterized query (prevents SQL injection)
Input validation
Try-except for safety
Proper resource cleanup

Task 5: AI-Based Automated Code Review Report

Task Description

You are provided with a poorly written Python script.

Using AI-assisted review:

- Generate a structured code review report that evaluates:

Code readability

Naming conventions

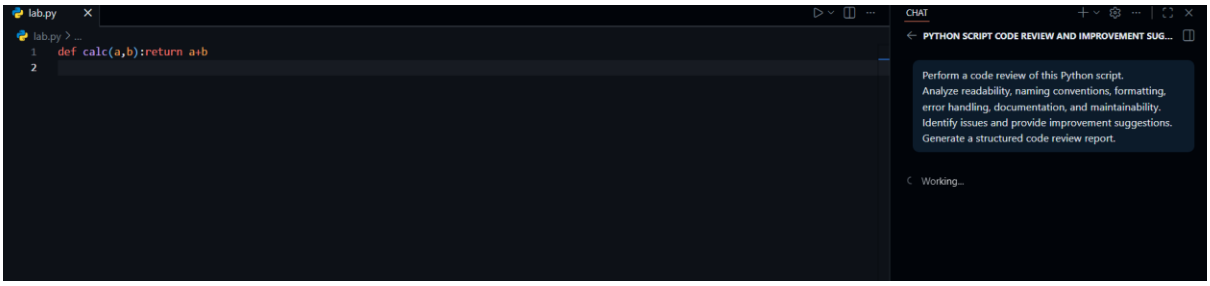
Formatting and style consistency

Error handling

Documentation quality

Maintainability

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



```
def calc(a,b):return a+b
```

Perform a code review of this Python script.
Analyze readability, naming conventions, formatting, error handling, documentation, and maintainability.
Identify issues and provide improvement suggestions.
Generate a structured code review report.

◆ AI-Generated Code Review Report

1️⃣Readability

- Poor formatting
- No indentation
- One-line implementation reduces clarity

2️⃣Naming Issues

- calc is unclear
- Variables a and b not descriptive

3️⃣Documentation

- No docstring
- No explanation of purpose

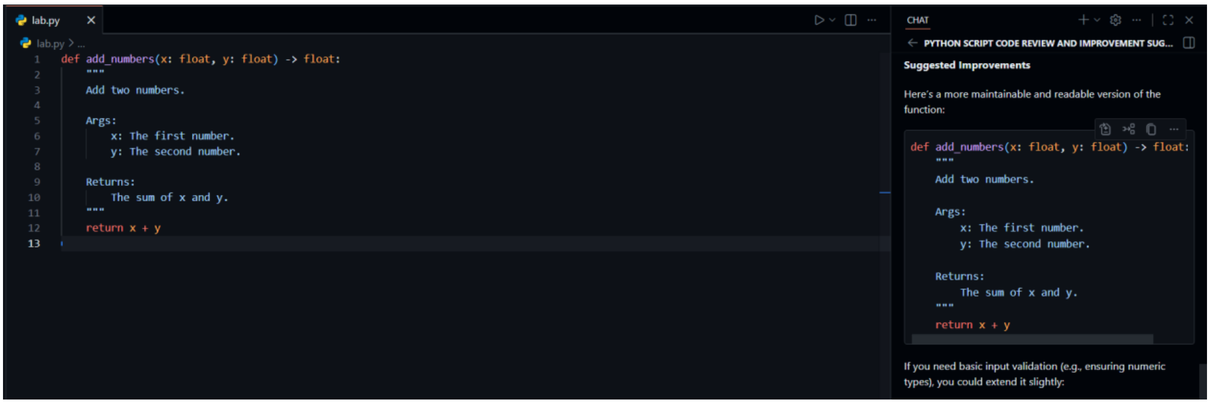
4️⃣Maintainability Risk

- Hard to scale
- Difficult for team collaboration

Actionable Recommendations

- Rename calc to something more descriptive (e.g., add_numbers).
- Reformat the function to a multi-line definition with proper spaces around operators and after commas.
- Add a clear docstring describing purpose, parameters, and return value.
- Add type hints for parameters and return type.

- (Optional, depending on context) Add basic type validation to guard against incorrect usage.
- (Optional) If this module is to grow, add a module-level docstring and consider adding simple unit tests in a separate test fil



```python
def add_numbers(x: float, y: float) -> float:
    """
    Add two numbers.

    Args:
        x: The first number.
        y: The second number.

    Returns:
        The sum of x and y.
    """
    return x + y
```

Suggested Improvements

Here's a more maintainable and readable version of the function:

```python
def add_numbers(x: float, y: float) -> float:
    """
    Add two numbers.

    Args:
        x: The first number.
        y: The second number.

    Returns:
        The sum of x and y.
    """
    return x + y
```

If you need basic input validation (e.g., ensuring numeric types), you could extend it slightly: