

Assignment 10.4

Name : N.Akhil

HT.NO :2303A51065

BATCH-16

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

The screenshot shows a code editor interface with a Python file named `lab.py` open. The code contains several errors and inconsistencies:`def calculateTotal(a,b):
 """calculate and print the total of two numbers."""
 result = a + b
 print("total is: " + result)
 print("total is: (result)")

calculate_total(5,10)
calculate_total(5, 10)`The AI tool has identified the following issues:- Missing colon in function definition
- PEP 8 naming inconsistency (function name)
- PEP 8 spacing around commas

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

The screenshot shows a code editor window with a Python file named 'lab.py'. The code contains a function 'find_duplicates' that uses two nested loops to iterate through a list and find duplicates. The sidebar on the right is titled 'CHAI' and displays a task for 'OPTIMIZING DUPLICATE DETECTION LOGIC FOR PERFORMANCE'. It includes a sub-task 'Analyze performance bottlenecks and optimize duplicate detection logic while preserving correctness'. The interface shows the AI's thought process, starting with inspecting the code to find the logic, then working on potential optimizations, and finally evaluating the changes. The desktop environment at the bottom shows a weather icon for 30°C and sunny conditions, along with a taskbar with various application icons.

```
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
```

The screenshot shows an IDE interface with a code editor and a right-hand panel for refactoring.

Code Editor:

```

laby > find_duplicates
1 def find_duplicates(lst):
2     """Return a list of duplicates for each equal pair (i, j), i < j.
3
4     This preserves the original behaviour: if the same value appears k times,
5     it will be returned  $O(k, 2)$  times, in the same order the nested-loop
6     implementation would produce.
7
8     Time complexity:
9         -  $O(n + m)$ , where  $n$  is  $\text{len}(lst)$  and  $m$  is the number of equal pairs.
10        In the worst case (all elements equal),  $m$  is  $O(n^2)$ , which is
11        optimal because that many outputs must be produced.
12        - For inputs with few duplicates, this avoids the  $O(n^2)$  comparisons
13        of the naive double loop.
14
15     Space complexity:  $O(n)$  for index bookkeeping.
16
17     n = len(lst)
18     if n < 2:
19         return []
20
21     # Map each value to the list of indices where it appears, and for every
22     # index remember its position inside that per-value list.
23     value_to_indices = {}
24     for idx, value in enumerate(lst):
25         indices = value_to_indices.setdefault(value, [])
26         indices.append(idx)
27         index_rank[idx] = len(indices) - 1
28
29     duplicates = []
30
31     # For each position i, we only consider later occurrences of the same
32     # value, matching the original (i, j) with  $j > i$  iteration order but
33     # without comparing against different values.
34     for i, value in enumerate(lst):
35         remaining = len(indices) - i - index_rank[i]
36         if remaining > 0:
37             duplicates.extend([value] * remaining)
38
39     return duplicates

```

Refactoring Panel:

- Correctness (behavior preserved)**
 - For any value that appears m times, the original code returns it $O(m, 2)$ times (once per equal pair (i, j)). In the order induced by the outer `for` loop.
 - The new implementation:
 - Still returns each such value exactly $O(m, 2)$ 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: $O(n)$ where n is the number of equal pairs (i.e., the length of the returned list).
 - Overall: $O(n + m)$ time.
 - Worst case (all elements identical): $m = O(n^2)$, so time is still $O(n^2)$ — 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

The screenshot shows an IDE interface with a code editor and a right-hand panel for refactoring.

Code Editor:

```

laby > ...
1 def f(x,y):
2     r=[]
3     for i in x:
4         if i>y:
5             r.append(i)
6     return r

```

Refactoring Panel:

- CODE REFACTORING AND PEP 8 COMPLIANCE**
 - Refactor for readability, apply PEP 8 formatting, improve naming, and add documentation
- Working...**
 - Refactoring `laby.py`
 - The user wants me to refactor `laby` 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...

```
1 from typing import Iterable, List
2
3
4 def filter_greater_than(values: Iterable[float], threshold: float) -> List[float]:
5     """Return a list of values from "values" that are greater than "threshold".
6
7     Args:
8         values: An iterable of numeric values to filter.
9         threshold: The numeric threshold to compare against.
10
11    Returns:
12        A list containing all items from "values" that are strictly greater
13        than "threshold".
14
15    """
16
17    result: List[float] = []
18    for value in values:
19        if value > threshold:
20            result.append(value)
21    return result
```

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

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` +20.4

I've refactored lab.py:

- Applied PEP 8 formatting (spacing, indentation, imports at top).
- Renamed `filter` 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` → `values`) 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

```
1 import sqlite3
2
3 def get_user(username):
4     conn = sqlite3.connect("users.db")
5     cursor = conn.cursor()
6     query = "SELECT * FROM users WHERE username = " + username + ""
7     cursor.execute(query)
8     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..

The screenshot shows a code editor window for `lab.py` and a separate window titled "PYTHON CODE SECURITY REVIEW AND FIXES". The code editor contains Python code for a database query. The review window lists several security improvements made to the code:

- Replaced string-concatenated SQL with a parameterized query (`cursor.execute("SELECT * FROM users WHERE username = ?")`) to prevent SQL injection.
- Added input validation type check for string (`username`), non-empty check, and a regex that only allows letters, digits, and underscores in the username.
- Wrapped the database operations in a `try/except` block and used a `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

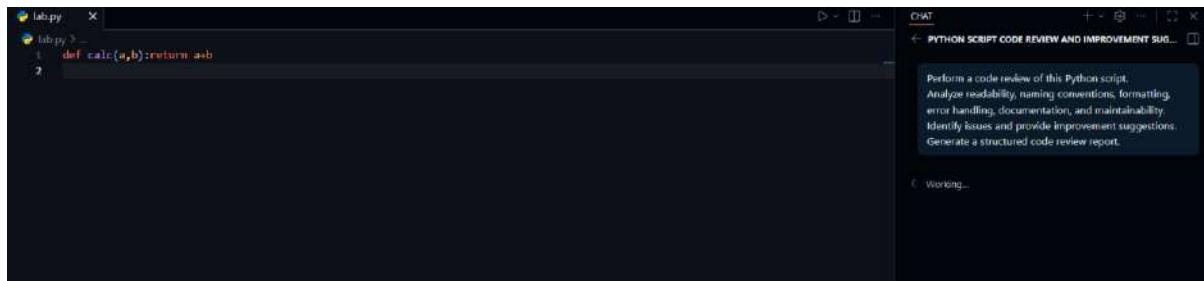
Formatting and style consistency

Error handling

Documentation quality

Maintainability

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



◆ AI-Generated Code Review Report

1 Readability

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

2 Naming Issues

- calc is unclear
- Variables a and b not descriptive

3 Documentation

- No docstring
- No explanation of purpose

4 Maintainability 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 file

The screenshot shows a code editor window with a Python file named 'lab.py'. The file contains the following code:

```
1 lab.py >-
2
3 def add_numbers(x: float, y: float) -> float:
4     """
5         Add two numbers.
6
7     Args:
8         x: The first number.
9         y: The second number.
10
11    Returns:
12        The sum of x and y.
13
14    return x + y
15
```

To the right of the code editor is a 'Suggested Improvements' panel. It displays a more maintainable and readable version of the function:

```
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
```

Below this, there is a note: "If you need basic input validation (e.g., ensuring numeric types), you could extend it slightly."