ASSIGNMENT-10.4

Name:G.Mythili

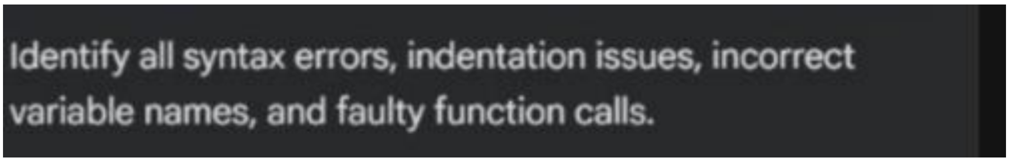
Ht.no:2303A51283

Batch:05

Task 1:AI-Assisted Syntax and Code QualityReview

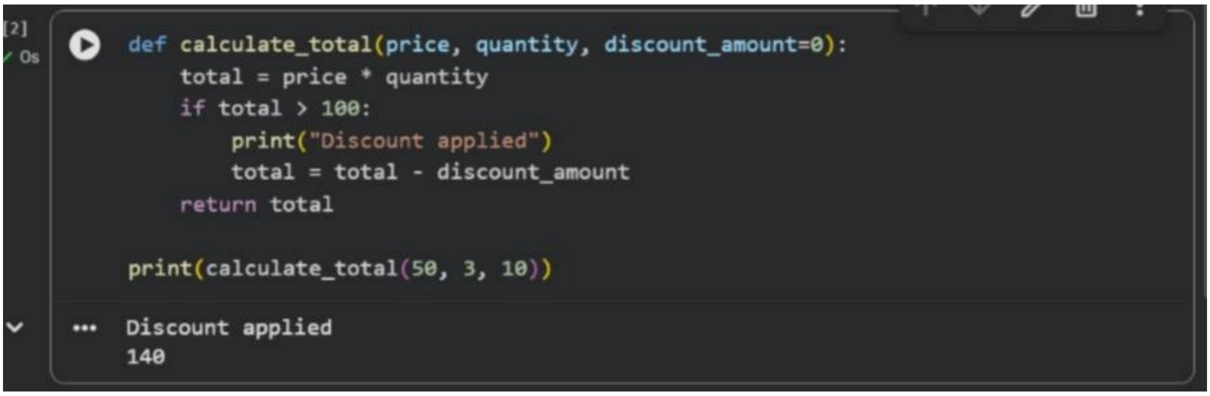
Scenario
You join a development team and are asked to review a juniordeveloper’s Python script that fails to run correctly due to basic codingmistakes. Before deployment, the code must be corrected andstandardized.

Prompt:



Identify all syntax errors, indentation issues, incorrect variable names, and faulty function calls.

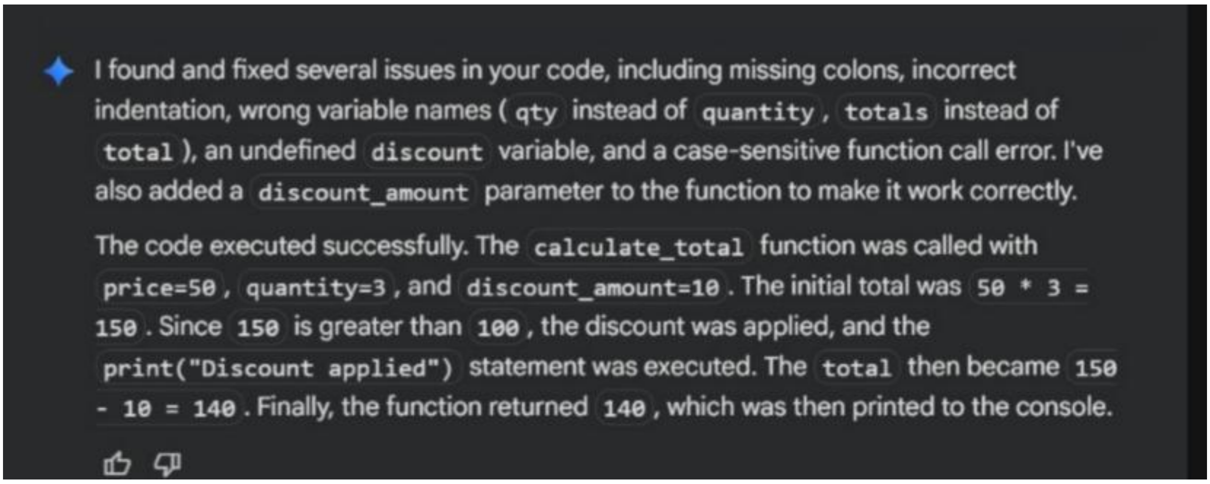
Code:



```python
def calculate_total(price, quantity, discount_amount=0):
    total = price * quantity
    if total > 100:
        print("Discount applied")
        total = total - discount_amount
    return total

print(calculate_total(50, 3, 10))
```

```
Discount applied
140
```

Error Explanation:



I found and fixed several issues in your code, including missing colons, incorrect indentation, wrong variable names (`qty` instead of `quantity`, `totals` instead of `total`), an undefined `discount` variable, and a case-sensitive function call error. I've also added a `discount_amount` parameter to the function to make it work correctly.

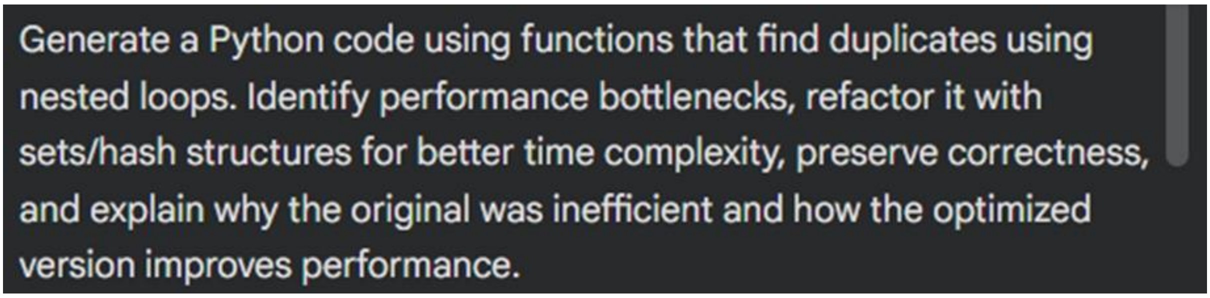
The code executed successfully. The `calculate_total` function was called with `price=50`, `quantity=3`, and `discount_amount=10`. The initial total was `50 * 3 = 150`. Since `150` is greater than `100`, the discount was applied, and the `print("Discount applied")` statement was executed. The `total` then became `150 - 10 = 140`. Finally, the function returned `140`, which was then printed to the console.

<u>Task 2:</u> Performance-Oriented Code Review

Scenario

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

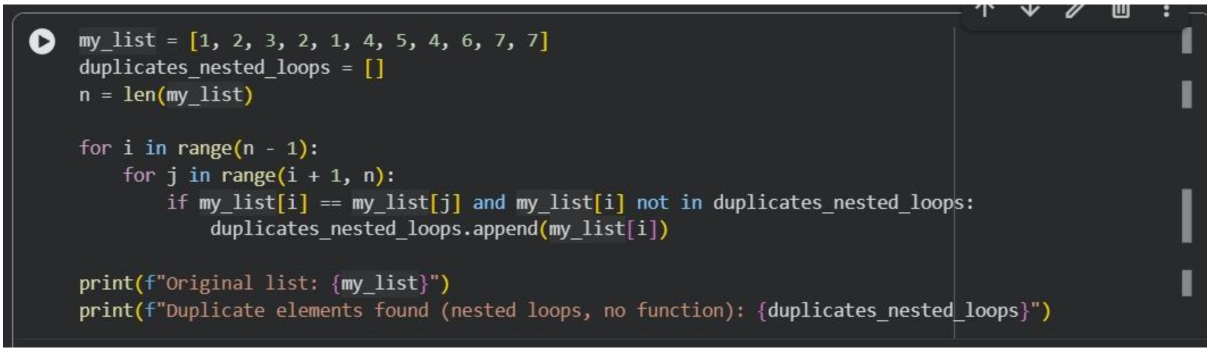
Prompt:



Generate a Python code using functions that find duplicates using nested loops. Identify performance bottlenecks, refactor it with sets/hash structures for better time complexity, preserve correctness, and explain why the original was inefficient and how the optimized version improves performance.

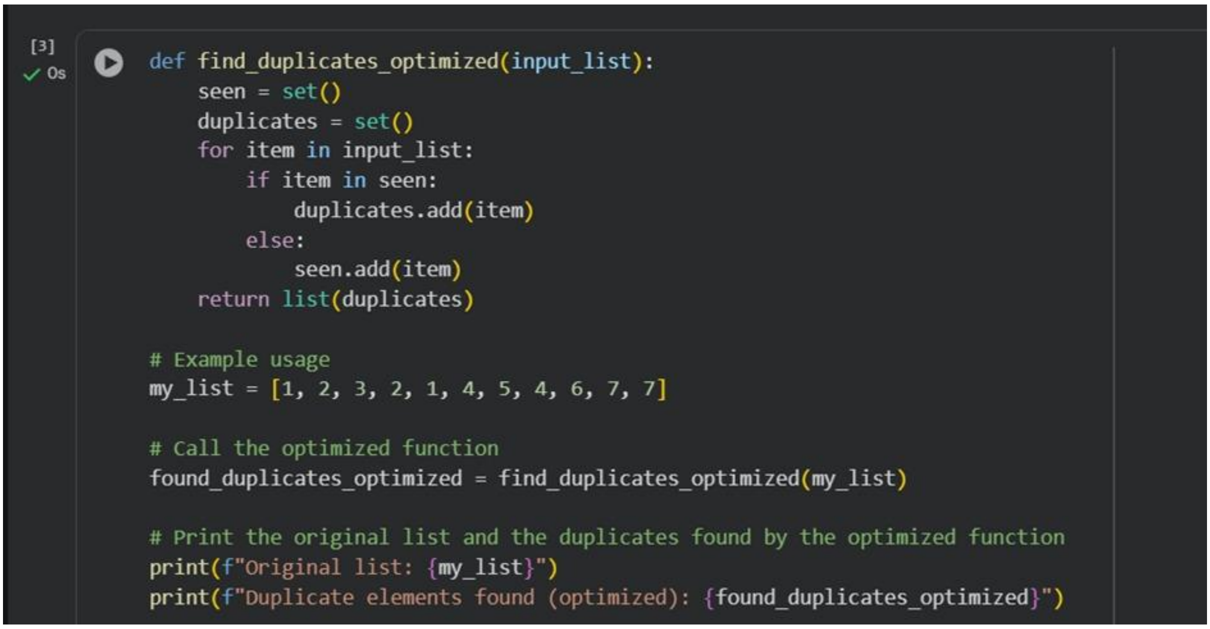
Code:

Without using functions:



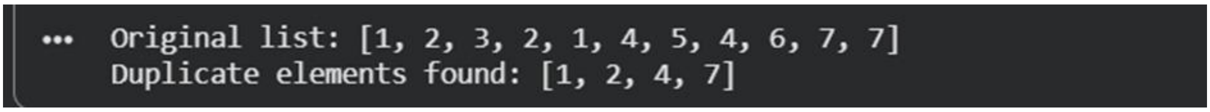
```python
my_list = [1, 2, 3, 2, 1, 4, 5, 4, 6, 7, 7]
duplicates_nested_loops = []
n = len(my_list)

for i in range(n - 1):
    for j in range(i + 1, n):
        if my_list[i] == my_list[j] and my_list[i] not in duplicates_nested_loops:
            duplicates_nested_loops.append(my_list[i])

print(f"Original list: {my_list}")
print(f"Duplicate elements found (nested loops, no function): {duplicates_nested_loops}")
```

With using functions:



```python
def find_duplicates_optimized(input_list):
    seen = set()
    duplicates = set()
    for item in input_list:
        if item in seen:
            duplicates.add(item)
        else:
            seen.add(item)
    return list(duplicates)

# Example usage
my_list = [1, 2, 3, 2, 1, 4, 5, 4, 6, 7, 7]

# Call the optimized function
found_duplicates_optimized = find_duplicates_optimized(my_list)

# Print the original list and the duplicates found by the optimized function
print(f"Original list: {my_list}")
print(f"Duplicate elements found (optimized): {found_duplicates_optimized}")
```

Output:



```
Original list: [1, 2, 3, 2, 1, 4, 5, 4, 6, 7, 7]
Duplicate elements found: [1, 2, 4, 7]
```

Why the original approach was inefficient:

The nested loop compares each element with every other element.

- This results in O(n²) time complexity, which grows very slow for large lists.

- Example: For a list of 10,000 items, it could require up to 100 million comparisons.

- It also repeatedly checks membership in the duplicates list (in operator), which itself is O(n), adding further inefficiency.

How the optimized version improves performance?

By using a set (hash-based structure), we can track seen elements in O(1) average time per lookup.

- We only need a single pass through the list, making the algorithm O(n).
- This drastically reduces runtime for large datasets while preserving correctness.
- Sets also automatically prevent duplicate entries, simplifying the logic.

Improved Time Complexity

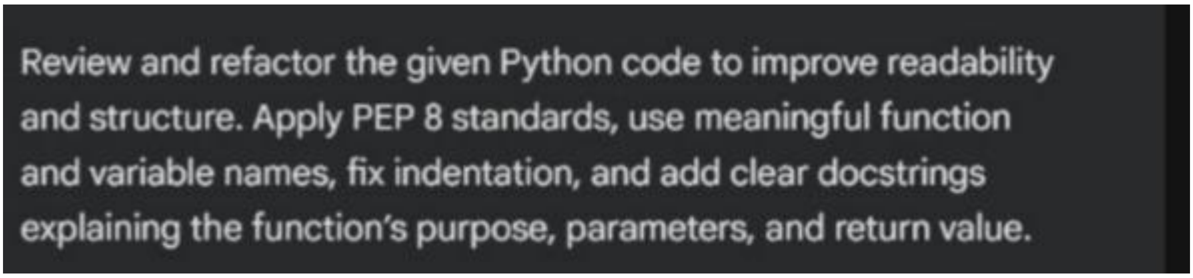
- Original: $O(n^2)$ due to nested loops and repeated membership checks.
- Optimized: O(n) because each element is processed once, with O(1) average lookup in a set.

Task 3:Readability and Maintainability Refactoring

Scenario

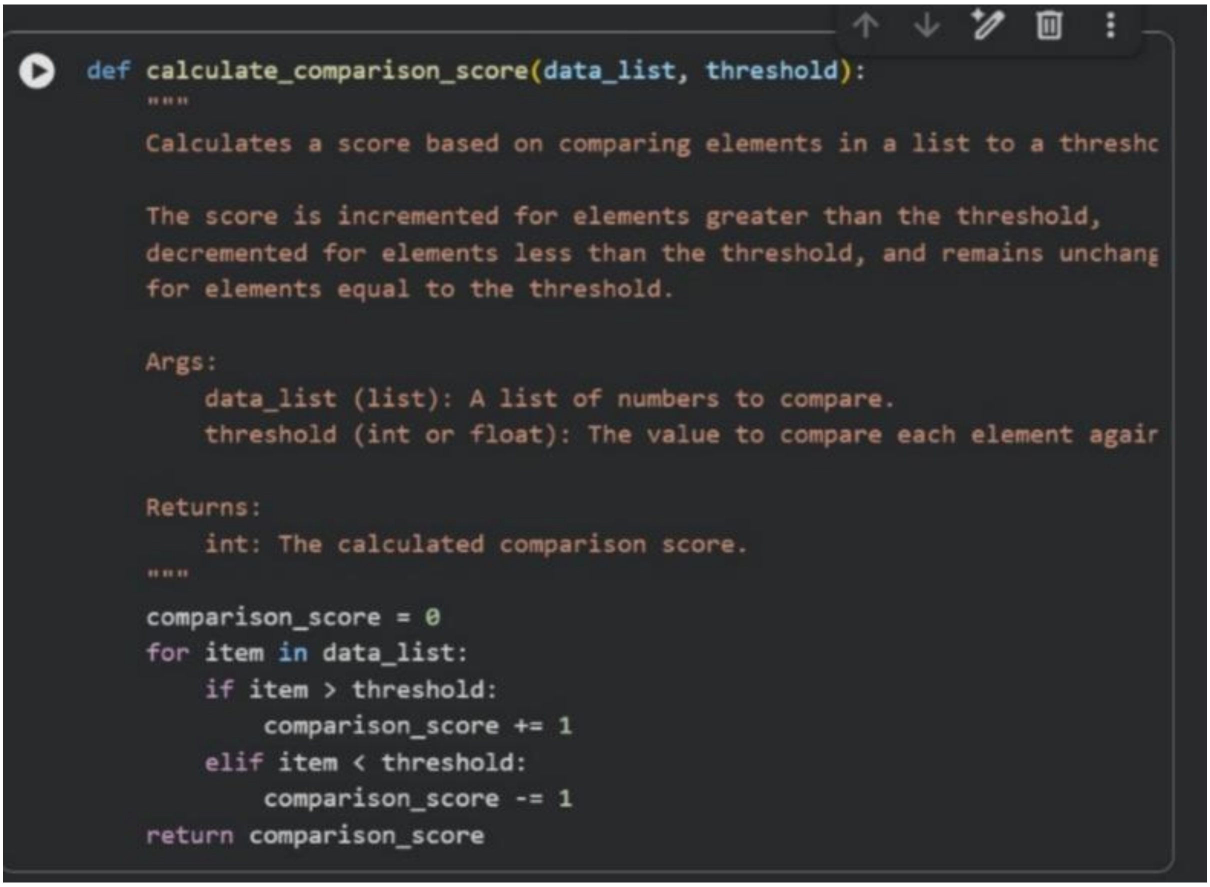
A working script exists in a project, but it is difficult to understand due topoor naming, formatting, and structure. The team wants it rewritten forlong-term maintainability.

Prompt:



Review and refactor the given Python code to improve readability and structure. Apply PEP 8 standards, use meaningful function and variable names, fix indentation, and add clear docstrings explaining the function’s purpose, parameters, and return value.

Code:



```
def calculate_comparison_score(data_list, threshold):
    """
    Calculates a score based on comparing elements in a list to a threshc

    The score is incremented for elements greater than the threshold,
    decremented for elements less than the threshold, and remains unchang
    for elements equal to the threshold.

    Args:
        data_list (list): A list of numbers to compare.
        threshold (int or float): The value to compare each element agair

    Returns:
        int: The calculated comparison score.
    """
    comparison_score = 0
    for item in data_list:
        if item > threshold:
            comparison_score += 1
        elif item < threshold:
            comparison_score -= 1
    return comparison_score
```

Explanation:

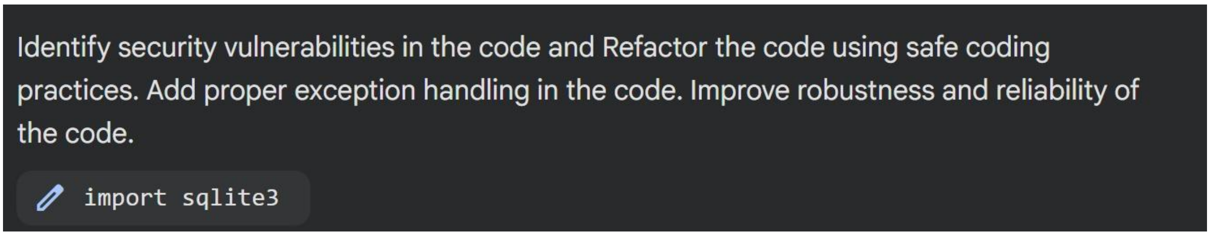
The original code was difficult to understand due to poor naming and formatting. After refactoring, clear names, proper indentation, and PEP 8 standards were applied. Docstrings were added to explain what the function does, its inputs, and outputs. This helps make the code clean, readable, and easy to reuse.

Task 4: Secure Coding and Reliability Review

Scenario

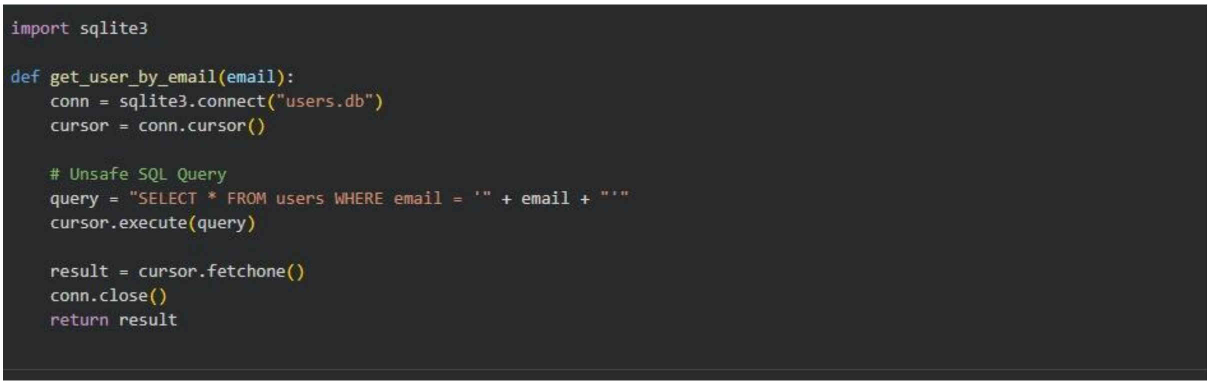
A backend function retrieves user data from a database but has securityvulnerabilities and poor error handling, making it unsafe for productiondeployment.

Prompt:



Identify security vulnerabilities in the code and Refactor the code using safe coding practices. Add proper exception handling in the code. Improve robustness and reliability of the code.

Unsafe:



```
import sqlite3

def get_user_by_email(email):
    conn = sqlite3.connect("users.db")
    cursor = conn.cursor()

    # Unsafe SQL Query
    query = "SELECT * FROM users WHERE email = '" + email + "'"
    cursor.execute(query)

    result = cursor.fetchone()
    conn.close()
    return result
```

Analyze SQL Injection Vulnerability:

The get_user_by_email function in the provided code is vulnerable to SQL injection because it constructs the SQL query by directly concatenating user-supplied input (email) into the SQL string. This method does not sanitize or escape special characters, allowing an attacker to manipulate the query's logic.

Vulnerability Explanation:

The line query = "SELECT * FROM users WHERE email = '" + email + "'" is the source of the vulnerability. If the email parameter contains SQL keywords or special characters, they are interpreted as part of the SQL query rather than as literal string data.

Example of a Malicious Input:

Consider the following malicious email input: ' OR 1=1 --If this input is passed to the get_user_by_email function, the query variable will become: SELECT * FROM users WHERE email = '' OR 1=1 --'

Impact of the Malicious Input:

Let's break down what this malicious query does:

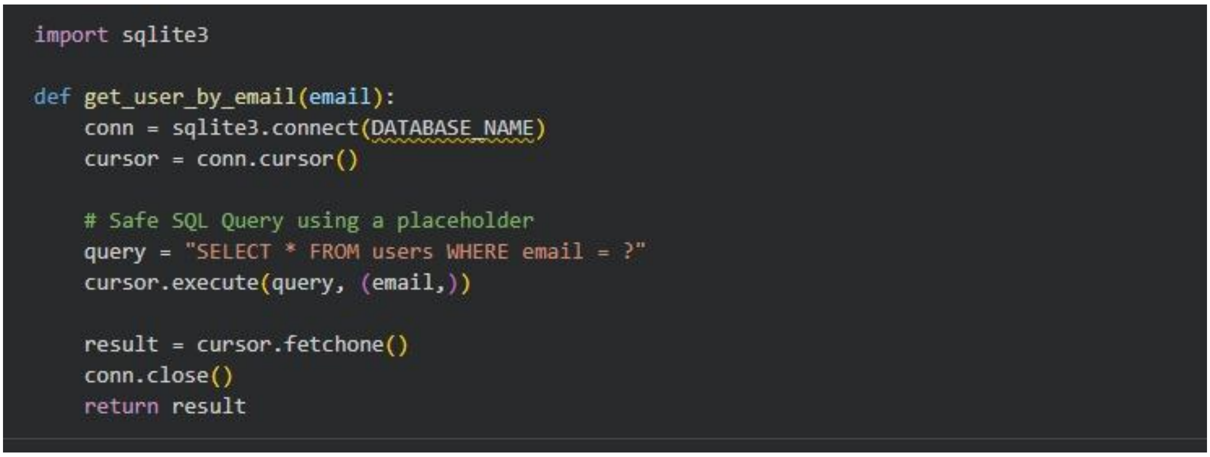
email = '': This part evaluates to false for any legitimate email.OR 1=1: This is always true. Because of the OR operator, the entire WHERE clause becomes true, regardless of the email field's actual content.

--: This is a SQL comment marker. Anything after -- is ignored by the database. In this case, it comments out the closing single quote (') that would normally terminate the email string, preventing a syntax error.

As a result, the query SELECT * FROM users WHERE email = '' OR 1=1 --' will return all rows from the users table, effectively bypassing any authentication or data access control that relies on the email address. An attacker could retrieve sensitive user information without knowing a valid email or password.

Refactor with Parameterized Queries:

Reasoning: The subtask requires refactoring the get_user_by_email function to use parameterized queries to prevent SQL injection. This involves changing the SQL query to use a ? placeholder and passing the email as a tuple to cursor.execute(), as well as ensuring the DATABASE_NAME variable is used for the database connection.

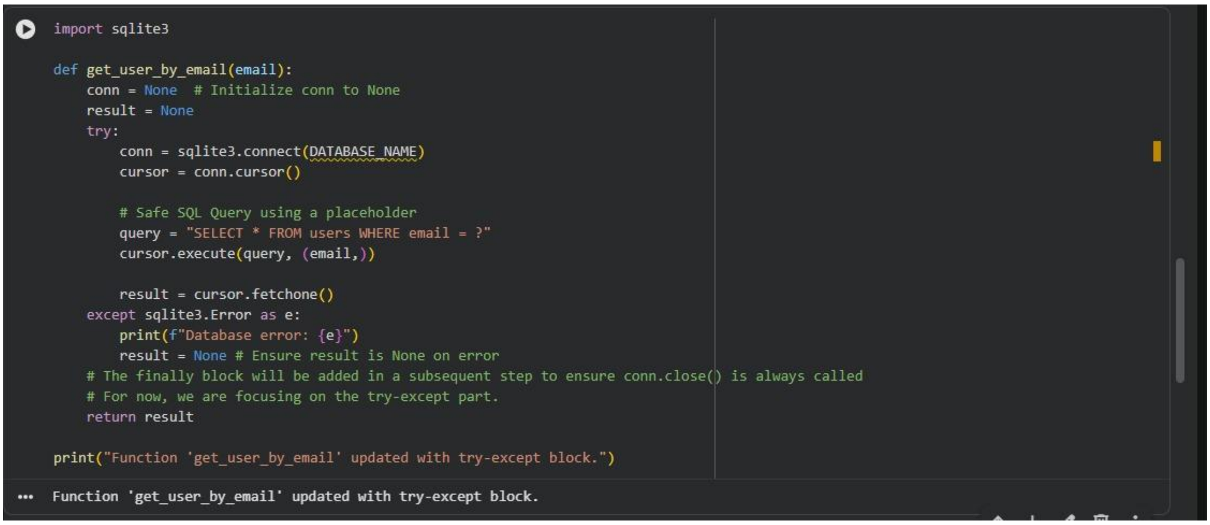


```python
import sqlite3

def get_user_by_email(email):
    conn = sqlite3.connect(DATABASE_NAME)
    cursor = conn.cursor()

    # Safe SQL Query using a placeholder
    query = "SELECT * FROM users WHERE email = ?"
    cursor.execute(query, (email,))

    result = cursor.fetchone()
    conn.close()
    return result
```

Implement Exception Handling

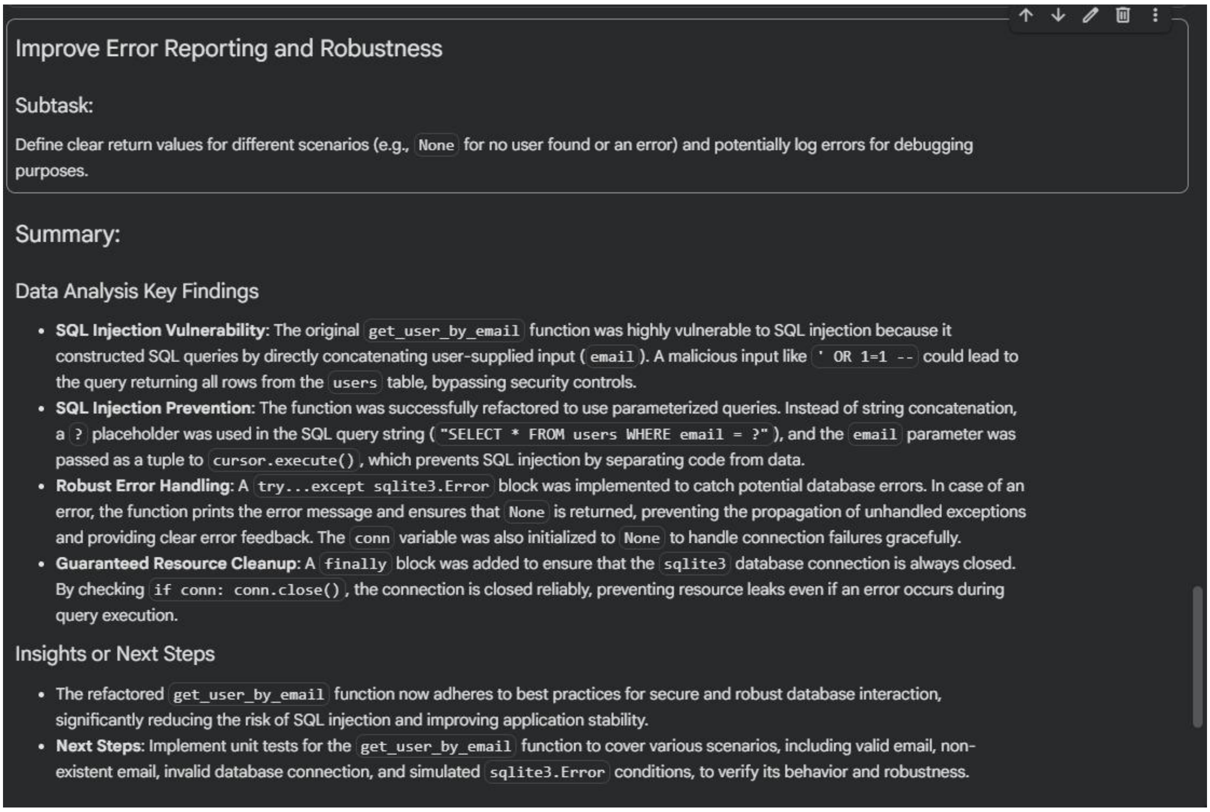
Subtask:

Add a try...except sqlite3.Error block to catch potential database errors during connection, query execution, or data fetching. Handle specific exceptions as needed.



```python
import sqlite3

def get_user_by_email(email):
    conn = None  # Initialize conn to None
    result = None
    try:
        conn = sqlite3.connect(DATABASE_NAME)
        cursor = conn.cursor()

        # Safe SQL Query using a placeholder
        query = "SELECT * FROM users WHERE email = ?"
        cursor.execute(query, (email,))

        result = cursor.fetchone()
    except sqlite3.Error as e:
        print(f"Database error: {e}")
        result = None # Ensure result is None on error
    # The finally block will be added in a subsequent step to ensure conn.close() is always called
    # For now, we are focusing on the try-except part.
    return result

print("Function 'get_user_by_email' updated with try-except block.")
```

```
Function 'get_user_by_email' updated with try-except block.
```



Improve Error Reporting and Robustness

Subtask:

Define clear return values for different scenarios (e.g., `None` for no user found or an error) and potentially log errors for debugging purposes.

Summary:

Data Analysis Key Findings

- **SQL Injection Vulnerability**: The original `get_user_by_email` function was highly vulnerable to SQL injection because it constructed SQL queries by directly concatenating user-supplied input (`email`). A malicious input like `' OR 1=1 --` could lead to the query returning all rows from the `users` table, bypassing security controls.
- **SQL Injection Prevention**: The function was successfully refactored to use parameterized queries. Instead of string concatenation, a `?` placeholder was used in the SQL query string (`"SELECT * FROM users WHERE email = ?"`), and the `email` parameter was passed as a tuple to `cursor.execute()`, which prevents SQL injection by separating code from data.
- **Robust Error Handling**: A `try...except sqlite3.Error` block was implemented to catch potential database errors. In case of an error, the function prints the error message and ensures that `None` is returned, preventing the propagation of unhandled exceptions and providing clear error feedback. The `conn` variable was also initialized to `None` to handle connection failures gracefully.
- **Guaranteed Resource Cleanup**: A `finally` block was added to ensure that the `sqlite3` database connection is always closed. By checking `if conn: conn.close()`, the connection is closed reliably, preventing resource leaks even if an error occurs during query execution.

Insights or Next Steps

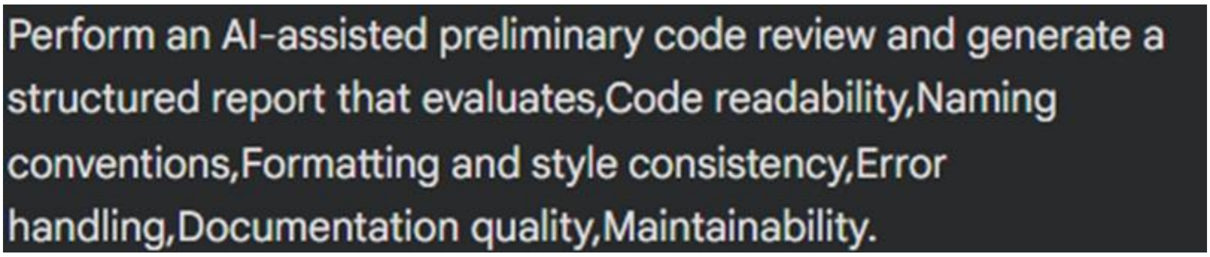
- The refactored `get_user_by_email` function now adheres to best practices for secure and robust database interaction, significantly reducing the risk of SQL injection and improving application stability.
- **Next Steps**: Implement unit tests for the `get_user_by_email` function to cover various scenarios, including valid email, non-existent email, invalid database connection, and simulated `sqlite3.Error` conditions, to verify its behavior and robustness.

Task 5: AI-Based Automated Code Review Report

Scenario

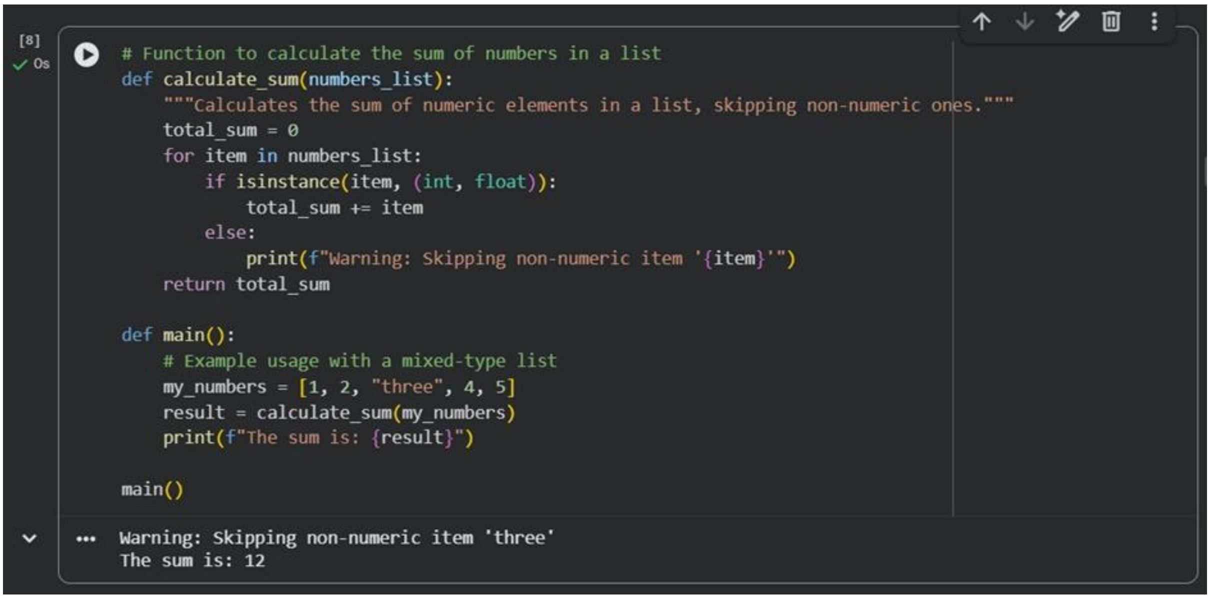
Your team uses AI tools to perform automated preliminary code reviewsbefore human review, to improve code quality and consistency acrossprojects.

Prompt:



Perform an AI-assisted preliminary code review and generate a structured report that evaluates,Code readability,Naming conventions,Formatting and style consistency,Error handling,Documentation quality,Maintainability.

Code:



```
# Function to calculate the sum of numbers in a list
def calculate_sum(numbers_list):
    """Calculates the sum of numeric elements in a list, skipping non-numeric ones."""
    total_sum = 0
    for item in numbers_list:
        if isinstance(item, (int, float)):
            total_sum += item
        else:
            print(f"Warning: Skipping non-numeric item '{item}'")
    return total_sum

def main():
    # Example usage with a mixed-type list
    my_numbers = [1, 2, "three", 4, 5]
    result = calculate_sum(my_numbers)
    print(f"The sum is: {result}")

main()
```

```
Warning: Skipping non-numeric item 'three'
The sum is: 12
```

Error Explanation:

The primary error we encountered was a TypeError: unsupported operand type(s) for +: 'int' and 'str' within the SUM function.

This error occurred because the function tried to directly add an integer (the running total) with a string (like 'three') from the input list. Python's + operator cannot perform addition between these incompatible types, leading to the crash. The solution was to modify the function to explicitly check the type of each item and only add numeric values to the sum, skipping non-numeric ones.