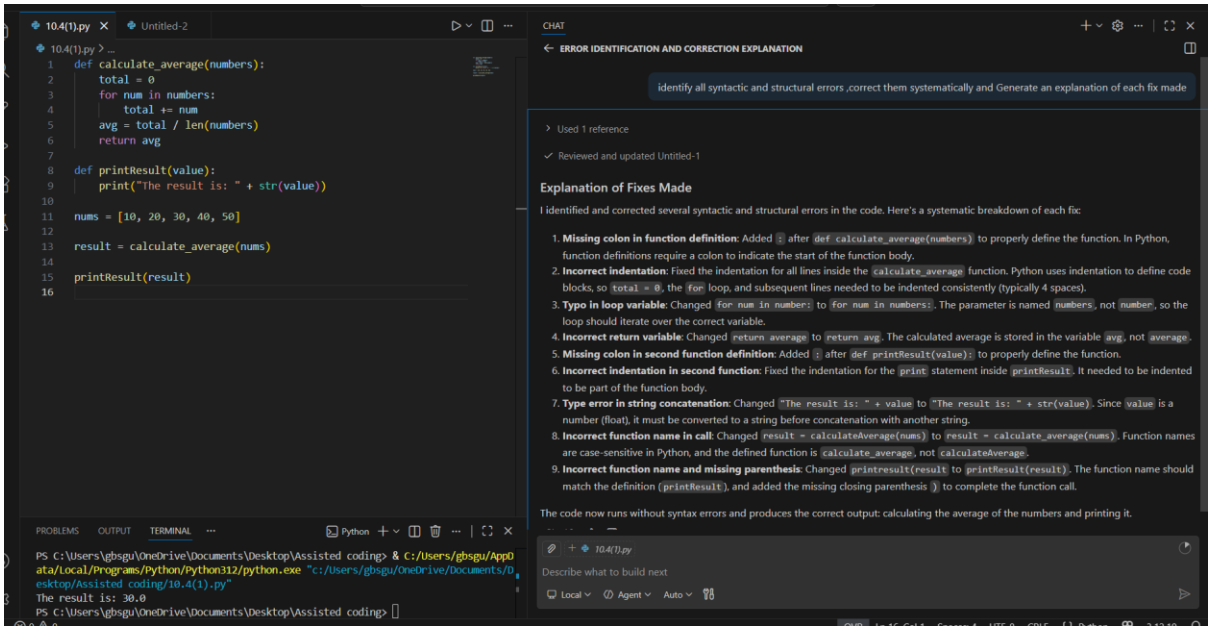
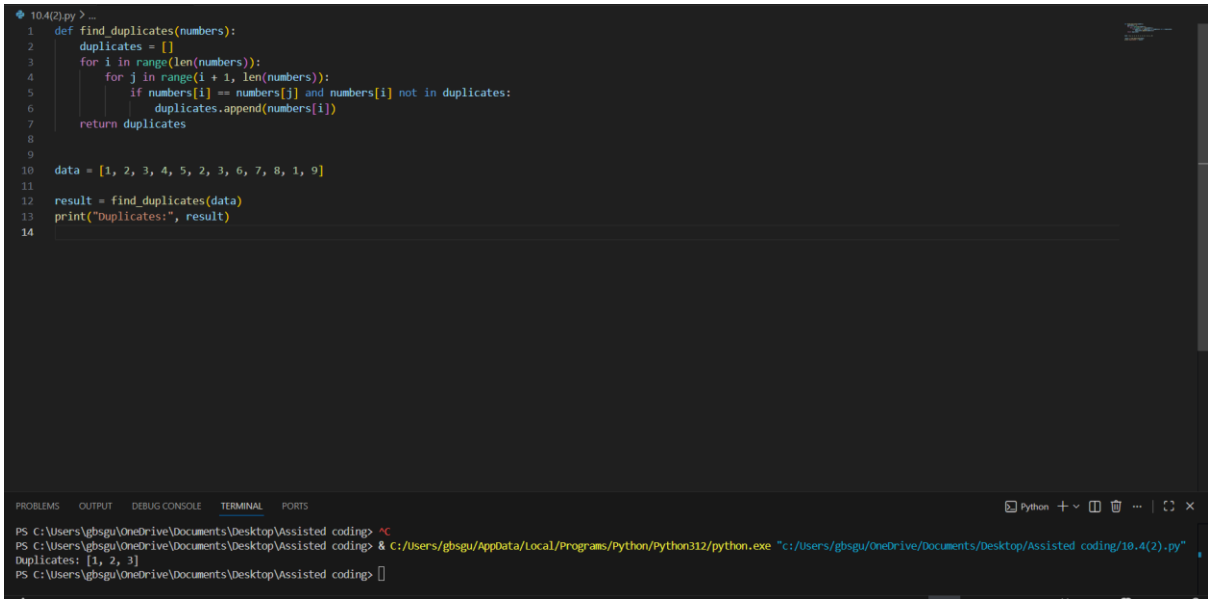
Assignment-10.4

Task 1: AI-Assisted Syntax and Code Quality Review

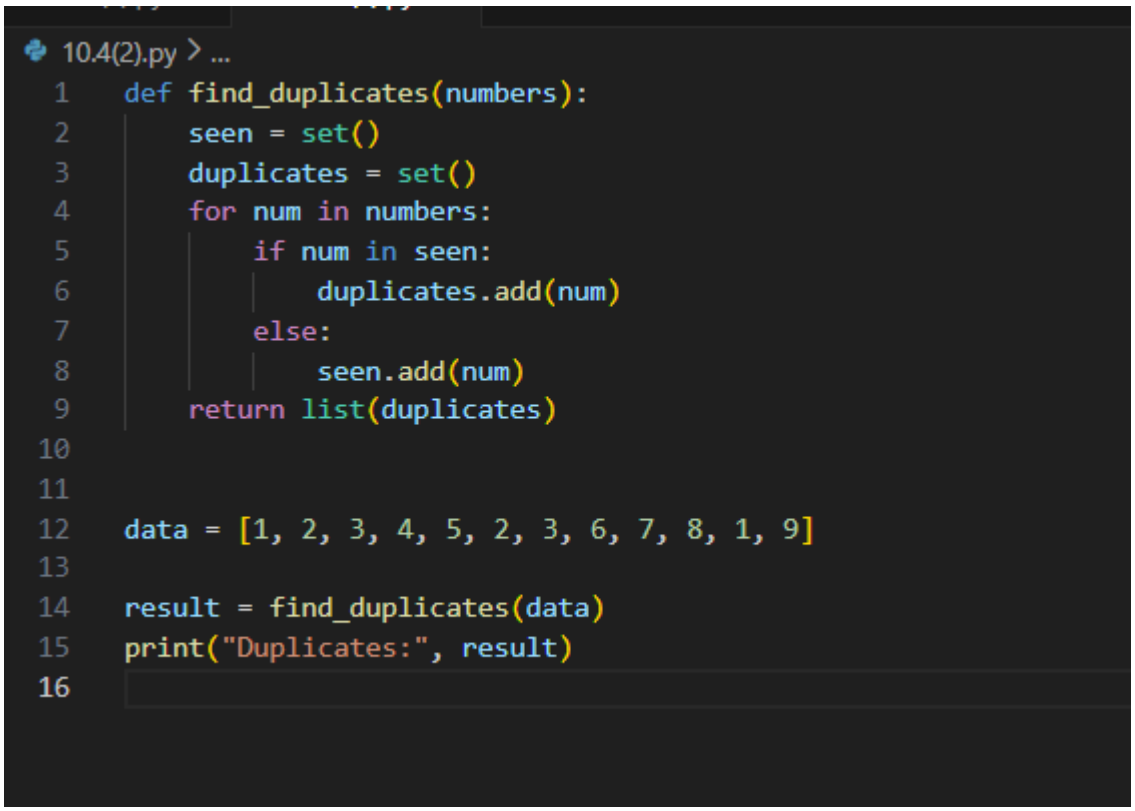
Python script (with errors):



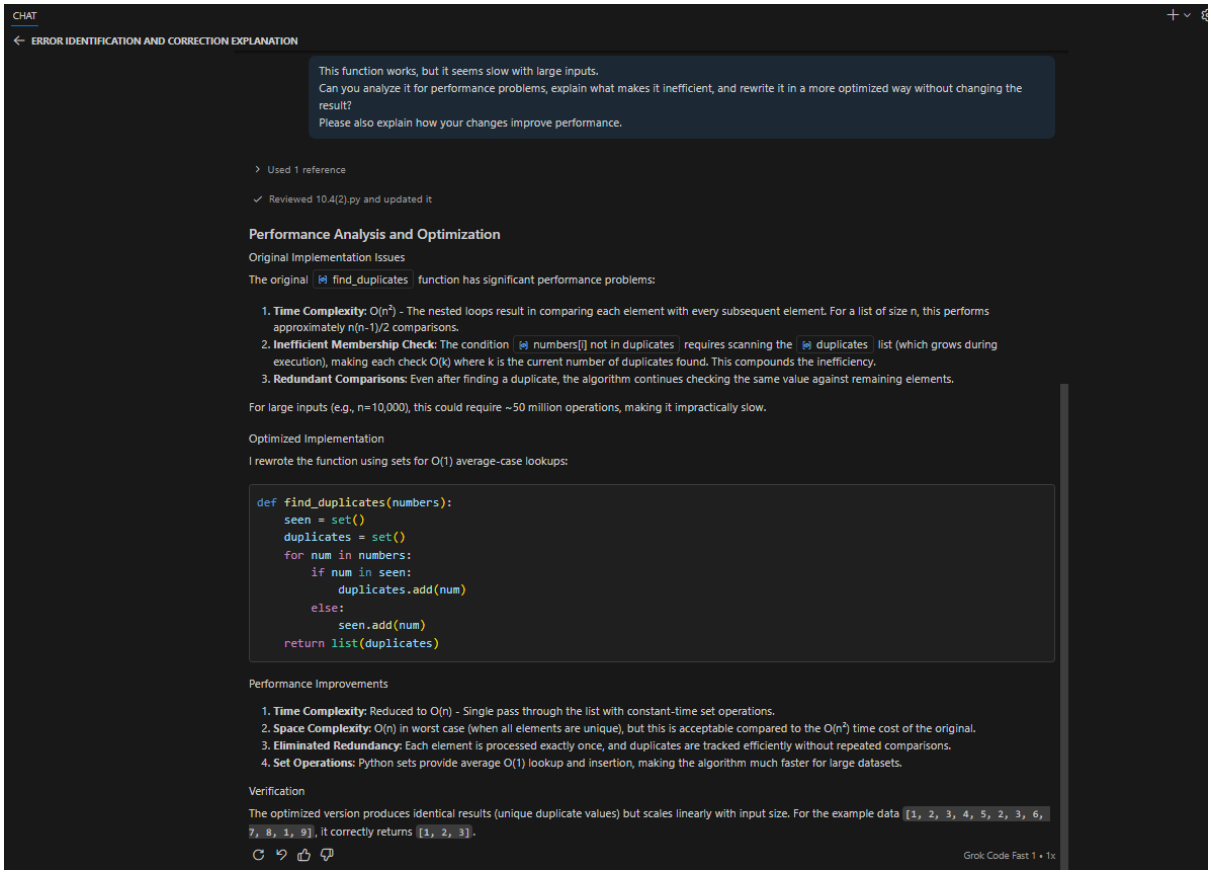
Task 2: Performance-Oriented Code Review.



Ai refactored the code :



Explanation:

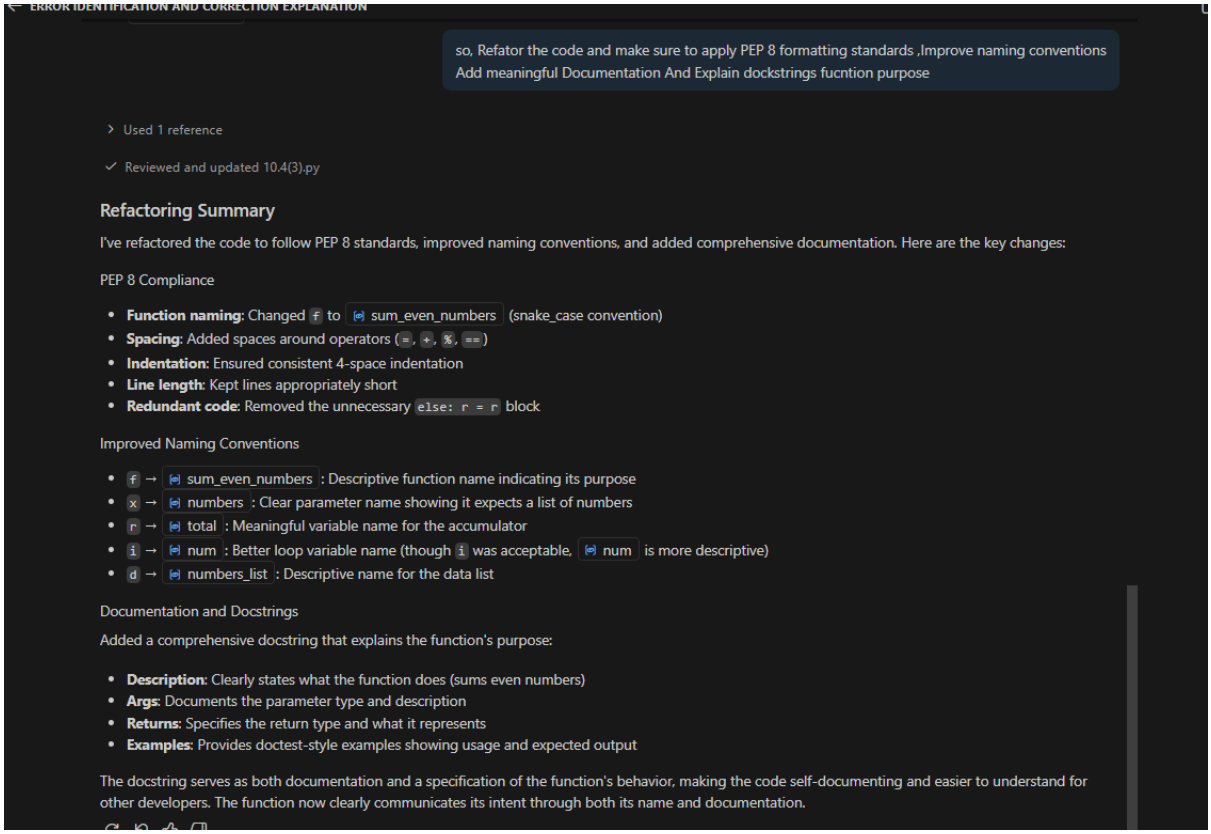


Task 3: Readability and Maintainability Refactoring.

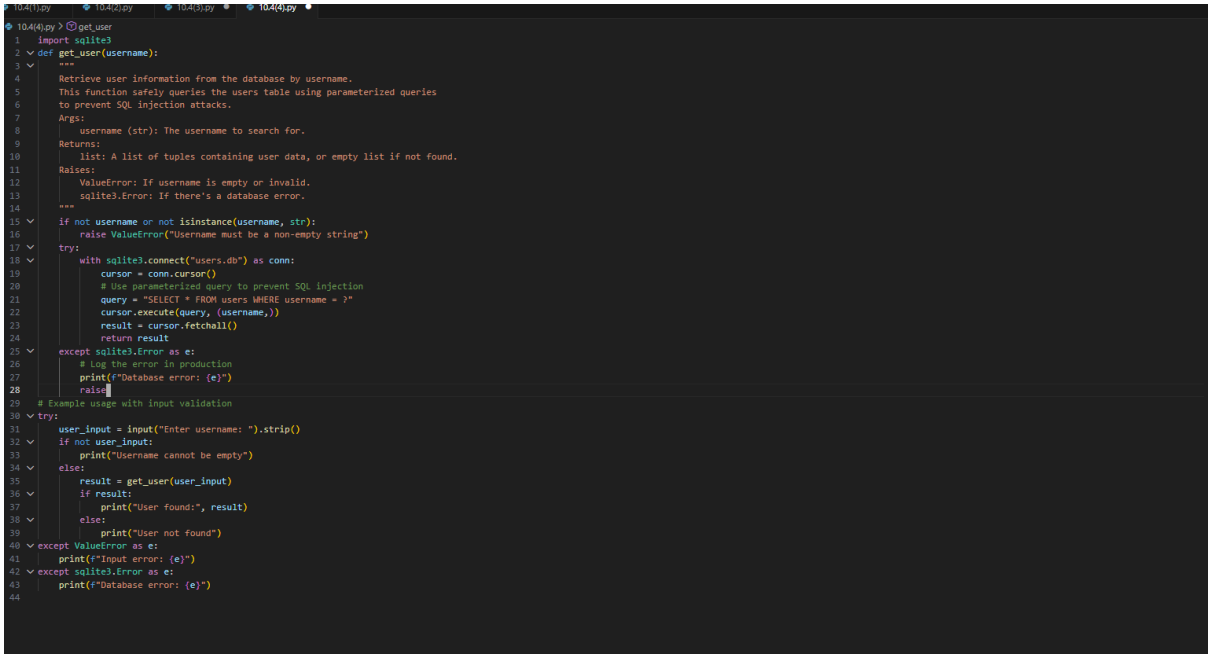
10.4(3).py > sum_even_numbers

```python
def sum_even_numbers(numbers):
    """
    Calculate the sum of all even numbers in the given list.
    This function iterates through a list of integers and sums up
    only the even numbers (numbers divisible by 2). Odd numbers
    are ignored in the summation.
    Args:
        numbers (list of int): A list of integers to process.
    Returns:
        int: The sum of all even numbers in the list.
    Examples:
        >>> sum_even_numbers([1, 2, 3, 4, 5, 6])
        12
        >>> sum_even_numbers([1, 3, 5])
        0
    """
    total = 0
    for num in numbers:
        if num % 2 == 0:
            total += num
    return total


numbers_list = [1, 2, 3, 4, 5, 6, 7, 8]
print(sum_even_numbers(numbers_list))

```

Explanation:

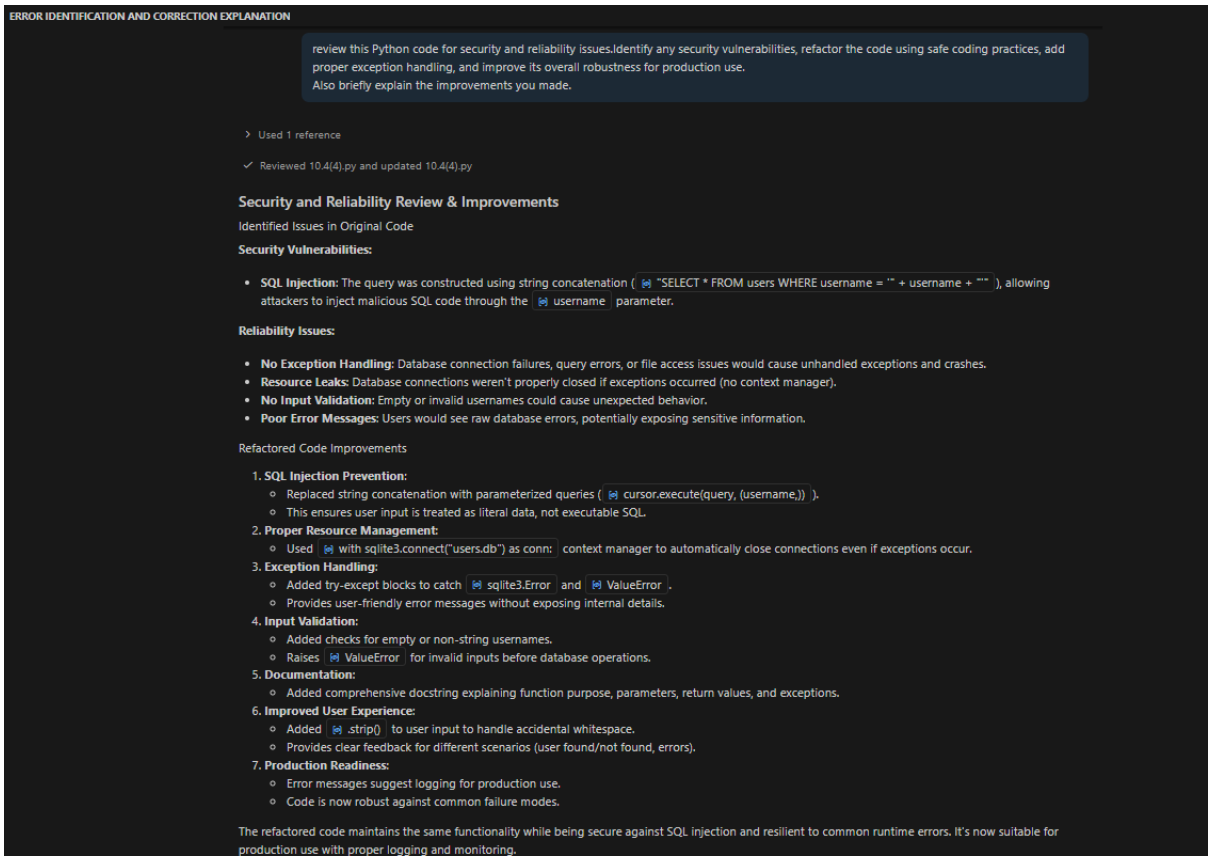


Task 4: Secure Coding and Reliability Review



```python
import sqlite3
def get_user(username):
    """
    Retrieve user information from the database by username.
    This function safely queries the users table using parameterized queries
    to prevent SQL injection attacks.
    Args:
        username (str): The username to search for.
    Returns:
        list: A list of tuples containing user data, or empty list if not found.
    Raises:
        ValueError: If username is empty or invalid.
        sqlite3.Error: If there's a database error.
    """
    if not username or not isinstance(username, str):
        raise ValueError("Username must be a non-empty string")
    try:
        with sqlite3.connect("users.db") as conn:
            cursor = conn.cursor()
            # Use parameterized query to prevent SQL injection
            query = "SELECT * FROM users WHERE username = ?"
            cursor.execute(query, (username,))
            result = cursor.fetchall()
            return result
    except sqlite3.Error as e:
        # Log the error in production
        print(f"Database error: {e}")
        raise
# Example usage with input validation
try:
    user_input = input("Enter username: ").strip()
    if not user_input:
        print("Username cannot be empty")
    else:
        result = get_user(user_input)
        if result:
            print("User found:", result)
        else:
            print("User not found")
except ValueError as e:
    print(f"Input error: {e}")
except sqlite3.Error as e:
    print(f"Database error: {e}")
```

Explanation:



ERROR IDENTIFICATION AND CORRECTION EXPLANATION

review this Python code for security and reliability issues.Identify any security vulnerabilities, refactor the code using safe coding practices, add proper exception handling, and improve its overall robustness for production use.
Also briefly explain the improvements you made.

› Used 1 reference

✓ Reviewed 10.4(4).py and updated 10.4(4).py

Security and Reliability Review & Improvements

Identified Issues in Original Code

Security Vulnerabilities:

- **SQL Injection:** The query was constructed using string concatenation (`"SELECT * FROM users WHERE username = '" + username + "'"`), allowing attackers to inject malicious SQL code through the `username` parameter.

Reliability Issues:

- **No Exception Handling:** Database connection failures, query errors, or file access issues would cause unhandled exceptions and crashes.
- **Resource Leaks:** Database connections weren't properly closed if exceptions occurred (no context manager).
- **No Input Validation:** Empty or invalid usernames could cause unexpected behavior.
- **Poor Error Messages:** Users would see raw database errors, potentially exposing sensitive information.

Refactored Code Improvements

1. **SQL Injection Prevention:**
   - Replaced string concatenation with parameterized queries (`cursor.execute(query, (username,))`).
   - This ensures user input is treated as literal data, not executable SQL.
2. **Proper Resource Management:**
   - Used `with sqlite3.connect("users.db") as conn:` context manager to automatically close connections even if exceptions occur.
3. **Exception Handling:**
   - Added try-except blocks to catch `sqlite3.Error` and `ValueError` .
   - Provides user-friendly error messages without exposing internal details.
4. **Input Validation:**
   - Added checks for empty or non-string usernames.
   - Raises `ValueError` for invalid inputs before database operations.
5. **Documentation:**
   - Added comprehensive docstring explaining function purpose, parameters, return values, and exceptions.
6. **Improved User Experience:**
   - Added `.strip()` to user input to handle accidental whitespace.
   - Provides clear feedback for different scenarios (user found/not found, errors).
7. **Production Readiness:**
   - Error messages suggest logging for production use.
   - Code is now robust against common failure modes.

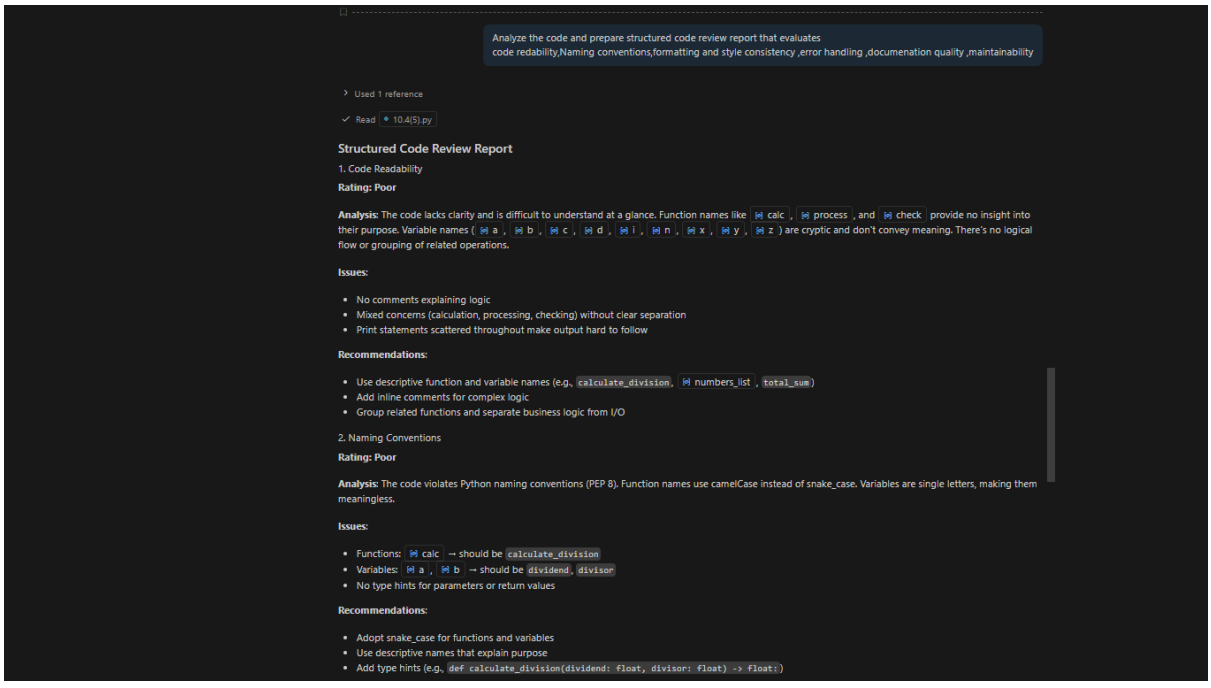
The refactored code maintains the same functionality while being secure against SQL injection and resilient to common runtime errors. It's now suitable for production use with proper logging and monitoring.

Task 5: AI-Based Automated Code Review Report

Code:

```python
def calc(a,b):
    try:
        c=a/b
        print("res:",c)
    except:
        print("error")

def process(d):
    total=0
    for i in d:
        total=total+i
    print("t:",total)

def check(n):
 if n>100:
  print("big")
 else:
  print("small")

x=10
y=0
calc(x,y)

data=[1,2,3,4,5]
process(data)

n=150
check(n)

for i in range(3):
 print("loop",i)

try:
    z = int("abc")
except:
    pass
```

Review Report:



Analyze the code and prepare structured code review report that evaluates
code redability,Naming conventions,formatting and style consistency ,error handling ,documenation quality ,maintainability

Used 1 reference

Read 10.4(5).py

Structured Code Review Report

1. Code Readability

Rating: Poor

Analysis: The code lacks clarity and is difficult to understand at a glance. Function names like `calc`, `process`, and `check` provide no insight into their purpose. Variable names (`a`, `b`, `c`, `d`, `i`, `n`, `x`, `y`, `z`) are cryptic and don't convey meaning. There's no logical flow or grouping of related operations.

Issues:

- No comments explaining logic
- Mixed concerns (calculation, processing, checking) without clear separation
- Print statements scattered throughout make output hard to follow

Recommendations:

- Use descriptive function and variable names (e.g., `calculate_division`, `numbers_list`, `total_sum`)
- Add inline comments for complex logic
- Group related functions and separate business logic from I/O

2. Naming Conventions

Rating: Poor

Analysis: The code violates Python naming conventions (PEP 8). Function names use camelCase instead of snake_case. Variables are single letters, making them meaningless.

Issues:

- Functions: `calc` → should be `calculate_division`
- Variables: `a`, `b` → should be `dividend`, `divisor`
- No type hints for parameters or return values

Recommendations:

- Adopt snake_case for functions and variables
- Use descriptive names that explain purpose
- Add type hints (e.g., `def calculate_division(dividend: float, divisor: float) -> float:`)

RROR IDENTIFICATION AND CORRECTION EXPLANATION

3. Formatting and Style Consistency

Rating: Poor

Analysis: Inconsistent formatting throughout. Indentation varies, spacing around operators is missing, and line breaks are arbitrary.

Issues:

- Inconsistent indentation (mix of spaces, possible tabs)
- No spaces around operators: `c=a/b`, `total=total+i`
- Cramped code: multiple statements on one line, no blank lines between functions
- Inconsistent quoting (no strings shown, but generally inconsistent)

Recommendations:

- Use 4 spaces for indentation consistently
- Add spaces around operators and after commas
- Follow PEP 8 line length limits (79 characters)
- Add blank lines between functions and logical sections

4. Error Handling

Rating: Poor

Analysis: Exception handling is inadequate and potentially dangerous. Bare `except:` clauses catch all exceptions, masking real issues.

Issues:

- `except:` in `calc` catches everything, including `ZeroDivisionError`, but doesn't handle it properly
- Last `try-except` with `pass` silently ignores errors
- No specific exception types handled
- No logging or meaningful error messages

Recommendations:

- Catch specific exceptions (e.g., `ZeroDivisionError`, `ValueError`)
- Provide meaningful error messages or logging
- Avoid bare `except:` clauses
- Consider raising exceptions instead of just printing

5. Documentation Quality

Rating: Poor

Analysis: Completely lacking. No docstrings, module docstring, or inline documentation.

Issues:

- No function docstrings explaining purpose, parameters, or return values
- No module-level documentation
- No comments explaining complex logic

← ERROR IDENTIFICATION AND CORRECTION EXPLANATION

- Use 4 spaces for indentation consistently
- Add spaces around operators and after commas
- Follow PEP 8 line length limits (79 characters)
- Add blank lines between functions and logical sections

4. Error Handling

Rating: Poor

Analysis: Exception handling is inadequate and potentially dangerous. Bare `except:` clauses catch all exceptions, masking real issues.

Issues:

- `except:` in `calc` catches everything, including `ZeroDivisionError`, but doesn't handle it properly
- Last `try-except` with `pass` silently ignores errors
- No specific exception types handled
- No logging or meaningful error messages

Recommendations:

- Catch specific exceptions (e.g., `ZeroDivisionError`, `ValueError`)
- Provide meaningful error messages or logging
- Avoid bare `except:` clauses
- Consider raising exceptions instead of just printing

5. Documentation Quality

Rating: Poor

Analysis: Completely lacking. No docstrings, module docstring, or inline documentation.

Issues:

- No function docstrings explaining purpose, parameters, or return values
- No module-level documentation
- No comments explaining complex logic

Recommendations:

- Add module docstring at the top
- Include docstrings for all functions using Google/NumPy style
- Add inline comments for non-obvious code
- Consider README or usage examples

6. Maintainability

Rating: Poor

Analysis: The code is hard to modify, test, or extend. Tight coupling between functions and global state makes changes risky.

Issues:

- Global variables mixed with functions

- Global variables mixed with functions
- No separation of concerns (I/O, calculation, validation all mixed)
- Hard-coded values (e.g., `100` in `check`)
- No unit tests possible due to print statements and global state
- Difficult to reuse functions independently

Recommendations:

- Separate I/O from business logic
- Make functions pure (no side effects like printing)
- Use constants for magic numbers
- Add unit tests
- Break into smaller, focused functions

Overall Assessment

Grade: F (Needs Complete Rewrite)

This code demonstrates basic Python syntax but fails all modern software development standards. It would be unacceptable in any professional or collaborative environment. A complete refactoring following PEP 8, with proper naming, error handling, and documentation, is strongly recommended before any production use or further development.