

Assignment-10.4

Task 1: AI-Assisted Syntax and Code Quality Review

Python script (with errors):

```
10.4(1).py > ...
1  def calculate_average(numbers)
2  total = 0
3      for num in number:
4          total += num
5      avg = total / len(numbers)
6      return average
7
8  def printResult(value):
9      print("The result is: " + value)
10
11  nums = [10, 20, 30, 40, 50]
12
13  result = calculateAverage(nums)
14
15  printresult(result
16  )
```

The screenshot shows a code editor with the original Python script and a sidebar with an AI-generated review. The review identifies nine errors and provides explanations for each fix.

Explanation of Fixes Made

- Missing colon in function definition:** Added `:` after `def calculate_average(numbers)` to properly define the function. In Python, function definitions require a colon to indicate the start of the function body.
- Incorrect indentation:** Fixed the indentation for all lines inside the `calculate_average` function. Python uses indentation to define code blocks, so `total = 0`, the `for` loop, and subsequent lines needed to be indented consistently (typically 4 spaces).
- Type in loop variable:** Changed `for num in numbers:` to `for num in numbers:`. The parameter is named `numbers`, not `number`, so the loop should iterate over the correct variable.
- Incorrect return variable:** Changed `return average` to `return avg`. The calculated average is stored in the variable `avg`, not `average`.
- Missing colon in second function definition:** Added `:` after `def printResult(value)` to properly define the function.
- Incorrect indentation in second function:** Fixed the indentation for the `print` statement inside `printResult`. It needed to be indented to be part of the function body.
- Type error in string concatenation:** Changed `"The result is: " + value` to `"The result is: " + str(value)`. Since `value` is a number (float), it must be converted to a string before concatenation with another string.
- Incorrect function name in call:** Changed `result = calculateAverage(nums)` to `result = calculate_average(nums)`. Function names are case-sensitive in Python, and the defined function is `calculate_average`, not `calculateAverage`.
- Incorrect function name and missing parenthesis:** Changed `printresult(result` to `printResult(result)`. The function name should match the definition (`printResult`), and added the missing closing parenthesis `)` to complete the function call.

The code now runs without syntax errors and produces the correct output: calculating the average of the numbers and printing it.

Task 2: Performance-Oriented Code Review.

```
10.4(2).py > ...
1 def find_duplicates(numbers):
2     duplicates = []
3     for i in range(len(numbers)):
4         for j in range(i + 1, len(numbers)):
5             if numbers[i] == numbers[j] and numbers[i] not in duplicates:
6                 duplicates.append(numbers[i])
7     return duplicates
8
9
10 data = [1, 2, 3, 4, 5, 2, 3, 6, 7, 8, 1, 9]
11
12 result = find_duplicates(data)
13 print("Duplicates:", result)
14
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python + -

PS C:\Users\ghsgu\OneDrive\Documents\Desktop\Assisted coding> ^C
PS C:\Users\ghsgu\OneDrive\Documents\Desktop\Assisted coding> & C:\Users\ghsgu\AppData\Local\Programs\Python\Python312\python.exe "c:/Users/ghsgu/OneDrive/Documents/Desktop/Assisted coding/10.4(2).py"
Duplicates: [1, 2, 3]
PS C:\Users\ghsgu\OneDrive\Documents\Desktop\Assisted coding> []

Ai refactored the code :

```
10.4(2).py > ...
1 def find_duplicates(numbers):
2     seen = set()
3     duplicates = set()
4     for num in numbers:
5         if num in seen:
6             duplicates.add(num)
7         else:
8             seen.add(num)
9     return list(duplicates)
10
11
12 data = [1, 2, 3, 4, 5, 2, 3, 6, 7, 8, 1, 9]
13
14 result = find_duplicates(data)
15 print("Duplicates:", result)
16
```

Explanation:

CHAT

← ERROR IDENTIFICATION AND CORRECTION EXPLANATION

This function works, but it seems slow with large inputs. Can you analyze it for performance problems, explain what makes it inefficient, and rewrite it in a more optimized way without changing the result? Please also explain how your changes improve performance.

> Used 1 reference

✓ Reviewed 10.4(2).py and updated it

Performance Analysis and Optimization

Original Implementation Issues

The original `find_duplicates` function has significant performance problems:

1. **Time Complexity:** $O(n^2)$ - The nested loops result in comparing each element with every subsequent element. For a list of size n , this performs approximately $n(n-1)/2$ comparisons.

2. **Inefficient Membership Check:** The condition `numbers[i] not in duplicates` requires scanning the `duplicates` list (which grows during execution), making each check $O(k)$ where k is the current number of duplicates found. This compounds the inefficiency.

3. **Redundant Comparisons:** Even after finding a duplicate, the algorithm continues checking the same value against remaining elements.

For large inputs (e.g., $n=10,000$), this could require ~50 million operations, making it impractically slow.

Optimized Implementation

I rewrote the function using sets for $O(1)$ average-case lookups:

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

Performance Improvements

1. **Time Complexity:** Reduced to $O(n)$ - Single pass through the list with constant-time set operations.

2. **Space Complexity:** $O(n)$ in worst case (when all elements are unique), but this is acceptable compared to the $O(n^2)$ time cost of the original.

3. **Eliminated Redundancy:** Each element is processed exactly once, and duplicates are tracked efficiently without repeated comparisons.

4. **Set Operations:** Python sets provide average $O(1)$ lookup and insertion, making the algorithm much faster for large datasets.

Verification

The optimized version produces identical results (unique duplicate values) but scales linearly with input size. For the example data `[1, 2, 3, 4, 5, 2, 3, 6, 7, 8, 1, 9]`, it correctly returns `[1, 2, 3]`.

🔄 ↻ 📄 🔍

Grok Code Fast 1 • 1x

Task 3: Readability and Maintainability Refactoring.

10.4(3).py > sum_even_numbers

```
1 def sum_even_numbers(numbers):
2     """
3     Calculate the sum of all even numbers in the given list.
4     This function iterates through a list of integers and sums up
5     only the even numbers (numbers divisible by 2). Odd numbers
6     are ignored in the summation.
7     Args:
8     |   numbers (list of int): A list of integers to process.
9     Returns:
10    |   int: The sum of all even numbers in the list.
11    Examples:
12    |   >>> sum_even_numbers([1, 2, 3, 4, 5, 6])
13    |       12
14    |   >>> sum_even_numbers([1, 3, 5])
15    |       0
16    """
17    total = 0
18    for num in numbers:
19        if num % 2 == 0:
20            total += num
21    return total
22
23
24 numbers_list = [1, 2, 3, 4, 5, 6, 7, 8]
25 print(sum_even_numbers(numbers_list))
26
```

Explanation:

← ERROR IDENTIFICATION AND CORRECTION EXPLANATION

so, Refactor the code and make sure to apply PEP 8 formatting standards ,Improve naming conventions
Add meaningful Documentation And Explain docstrings function purpose

> Used 1 reference

✓ Reviewed and updated 10.4(3).py

Refactoring Summary

I've refactored the code to follow PEP 8 standards, improved naming conventions, and added comprehensive documentation. Here are the key changes:

PEP 8 Compliance

- **Function naming:** Changed `f` to `sum_even_numbers` (snake_case convention)
- **Spacing:** Added spaces around operators (`=`, `+`, `%`, `==`)
- **Indentation:** Ensured consistent 4-space indentation
- **Line length:** Kept lines appropriately short
- **Redundant code:** Removed the unnecessary `else: r = r` block

Improved Naming Conventions

- `f` → `sum_even_numbers` : Descriptive function name indicating its purpose
- `x` → `numbers` : Clear parameter name showing it expects a list of numbers
- `r` → `total` : Meaningful variable name for the accumulator
- `i` → `num` : Better loop variable name (though `i` was acceptable, `num` is more descriptive)
- `d` → `numbers_list` : Descriptive name for the data list

Documentation and Docstrings

Added a comprehensive docstring that explains the function's purpose:

- **Description:** Clearly states what the function does (sums even numbers)
- **Args:** Documents the parameter type and description
- **Returns:** Specifies the return type and what it represents
- **Examples:** Provides doctest-style examples showing usage and expected output

The docstring serves as both documentation and a specification of the function's behavior, making the code self-documenting and easier to understand for other developers. The function now clearly communicates its intent through both its name and documentation.

⌕ ↻ ⏮ ⏭ ⏪ ⏩ ⏴ ⏵ ⏶ ⏷

Task 4: Secure Coding and Reliability Review

```
10.4(1).py 10.4(2).py 10.4(3).py 10.4(4).py
10.4(4).py > get_user
1 import sqlite3
2 def get_user(username):
3     """
4     Retrieve user information from the database by username.
5     This function safely queries the users table using parameterized queries
6     to prevent SQL injection attacks.
7     Args:
8         username (str): The username to search for.
9     Returns:
10        list: A list of tuples containing user data, or empty list if not found.
11    Raises:
12        ValueError: If username is empty or invalid.
13        sqlite3.Error: If there's a database error.
14    """
15    if not username or not isinstance(username, str):
16        raise ValueError("Username must be a non-empty string")
17    try:
18        with sqlite3.connect("users.db") as conn:
19            cursor = conn.cursor()
20            # Use parameterized query to prevent SQL injection
21            query = "SELECT * FROM users WHERE username = ?"
22            cursor.execute(query, (username,))
23            result = cursor.fetchall()
24            return result
25    except sqlite3.Error as e:
26        # Log the error in production
27        print(f"Database error: {e}")
28        raise
29
30 # Example usage with input validation
31 try:
32     user_input = input("Enter username: ").strip()
33     if not user_input:
34         print("Username cannot be empty")
35     else:
36         result = get_user(user_input)
37         if result:
38             print("User found:", result)
39         else:
40             print("User not found")
41 except ValueError as e:
42     print(f"Input error: {e}")
43 except sqlite3.Error as e:
44     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

- 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.
- Proper Resource Management:**
 - Used `with sqlite3.connect("users.db") as conn:` context manager to automatically close connections even if exceptions occur.
- Exception Handling:**
 - Added try-except blocks to catch `sqlite3.Error` and `ValueError`.
 - Provides user-friendly error messages without exposing internal details.
- Input Validation:**
 - Added checks for empty or non-string usernames.
 - Raises `ValueError` for invalid inputs before database operations.
- Documentation:**
 - Added comprehensive docstring explaining function purpose, parameters, return values, and exceptions.
- Improved User Experience:**
 - Added `.strip()` to user input to handle accidental whitespace.
 - Provides clear feedback for different scenarios (user found/not found, errors).
- 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:

```
104(5).py > ...
1 def calc(a,b):
2     try:
3         c=a/b
4         print("res:",c)
5     except:
6         print("error")
7
8 def process(d):
9     total=0
10    for i in d:
11        total=total+i
12    print("t:",total)
13
14 def check(n):
15     if n>100:
16         print("big")
17     else:
18         print("small")
19
20 x=10
21 y=0
22 calc(x,y)
23
24 data=[1,2,3,4,5]
25 process(data)
26
27 n=150
28 check(n)
29
30 for i in range(3):
31     print("loop",i)
32
33 try:
34     z = int("abc")
35 except:
36     pass
37
```

Review Report:

Analyze the code and prepare structured code review report that evaluates code readability,naming conventions,formatting and style consistency,error handling,documentation quality,maintainability

Used 1 reference

Read 104(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:)

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

- 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.

