

Assignment-10.4

Task1: AI-Assisted Syntax and Code Quality Review

Scenario

You join a development team and are asked to review a junior developer's Python script that fails to run correctly due to basic coding mistakes. Before deployment, the code must be corrected and standardized.

Task Description

You are given a Python script containing:

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

Use an AI tool (GitHub Copilot / Cursor AI) to:

- Identify all syntactic and structural errors
- Correct them systematically
- Generate an explanation of each fix made

Expected Outcome

- Fully corrected and executable Python code
- AI-generated explanation describing:
 - Syntax fixes
 - Naming corrections
 - Structural improvements
- Clean, readable version of the script

The screenshot shows the AI-assisted Python code review interface. On the left, the code file `Task1.py` is displayed with line numbers. The code defines a function `calculate_average`, a class `DataProcessor`, and a `main` function. On the right, three tables provide detailed feedback:

- Syntax Fixes:** Shows two errors: missing colons at line 2 and line 8, and a missing colon at line 30. Solutions include adding colons after `def` definitions and if statements, and adding a colon before the closing brace of the `__init__` method.
- Indentation Issues:** Shows two errors: incorrect return indentation at line 6 and wrong append indentation at line 11. Solutions include moving the `return` statement to the same indentation level and fixing the indentation of the `append` call.
- Variable Naming Corrections:** Shows one typo at line 15 where `self.name` is used instead of `name`. The solution suggests changing it to `name`.

OUTPUT:

```
PS C:\Users\tloke\OneDrive\AI> & C:/Users/tloke/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/tloke/OneDrive/AI/Tas k1.py
Average: 3.0
Processed: [20, 40, 60]
Processor name: MyProcessor
PS C:\Users\tloke\OneDrive\AI>
```

Task 2: Performance-Oriented Code Review

Scenario

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

Task Description

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

Ask the AI to explain:

- Why the original approach was inefficient
- How the optimized version improves performance

Expected Outcome

- Optimized duplicate-detection logic (e.g., using sets or hash-based structures)
- Improved time complexity
- AI explanation of performance improvement
- Clean, readable implementation

The screenshot shows a code editor with Task2.py and an AI-assisted code review interface.

Task2.py:

```
1  # Inefficient approach: Using nested loops to find duplicates
2  def find_duplicates_inefficient(data_list):
3      """
4          Find duplicate values in a list using nested loops.
5          Time Complexity: O(n^2) - quadratic, very slow for large datasets
6      """
7      duplicates = []
8
9      for i in range(len(data_list)):
10         for j in range(i + 1, len(data_list)):
11             if data_list[i] == data_list[j]:
12                 if data_list[i] not in duplicates:
13                     duplicates.append(data_list[i])
14
15     return duplicates
16
17
18 # Optimized approach: Using a set for O(n) performance
19 def find_duplicates_optimized(data_list):
20     """
21         Find duplicate values in a list using sets.
22         Time Complexity: O(n) - linear, dramatically faster for large datasets
23     """
24     seen = set()
25     duplicates = set()
26
27     for item in data_list:
28         if item in seen:
29             duplicates.add(item)
30         else:
31             seen.add(item)
32
33     return list(duplicates)
34
35
36 # Test both functions
37 if __name__ == "__main__":
38     # Small dataset for correctness verification
39     small_data = [1, 2, 3, 2, 4, 1, 5, 3, 6]
40     print("Small dataset:", small_data)
41     print("Inefficient result:", find_duplicates_inefficient(small_data))
42     print("Optimized result:", find_duplicates_optimized(small_data))
43     print()
44
45     # Large dataset for performance comparison
46     import time
47
48     large_data = list(range(5000)) + list(range(2500)) # 7500 items with 2500 duplicates
```

AI-Assisted Python Code Review and Correction:

- Analyze the logic for performance bottlenecks
- Refactor the code for better time complexity
- Preserve the correctness of the output

Ask the AI to explain:

- Why the original approach was inefficient
- How the optimized version improves performance

Expected Outcome

- Optimized duplicate-detection logic (e.g., using sets or hash-based structures)
- Improved time complexity
- AI explanation of performance improvement
- Clean, readable implementation

✓ Created Task2.py with inefficient duplicate detection function

I'll create Task2.py with an inefficient duplicate detection function, then optimize it and explain the performance improvements.

✓ Created Task2.py

Now let me run the script to demonstrate the performance difference:

✓ Ran pyLanceRunCodeSnippet - pylance mcp server (MCP Server)

✓ Reviewed performance analysis and created comprehensive explanation

✓ Performance-Oriented Code Review Complete

Performance Analysis Results

Metric	Inefficient (Nested Loops)	Optimized (Sets)	Improvement
Time Complexity	O(n ²)	O(n)	1.820x faster
Execution Time (7,500 Items)	1.0759 seconds	0.0006 seconds	1.820.3x speedup
Space Complexity	O(d) - only result	O(n) - tracking all items	Small tradeoff
Correctness	<input checked="" type="checkbox"/> Correct	<input checked="" type="checkbox"/> Correct	Same output

Why the Original Approach Was Inefficient

Nested Loop Bottleneck - O(n²)

Task2.py

Describe what to build next

OUTPUT:

```
PS C:\Users\tloke\OneDrive\AI> & C:/Users/tloke/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/tloke/OneDrive/AI/Task2.py
Small dataset: [1, 2, 3, 2, 4, 1, 5, 3, 6]
Inefficient result: [1, 2, 3]
Optimized result: [1, 2, 3]

Large dataset size: 7500

Inefficient approach: 2500 duplicates found in 1.1530 seconds
Optimized approach: 2500 duplicates found in 0.0009 seconds

Performance Improvement: 1247.3x faster
```

Task 3: Readability and Maintainability Refactoring

Scenario

A working script exists in a project, but it is difficult to understand due to poor naming, formatting, and structure. The team wants it rewritten for

long-term maintainability.

Task Description

You are given a poorly structured Python function with:

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

Use AI-assisted review to:

- Refactor the code for clarity
- Apply PEP 8 formatting standards
- Improve naming conventions
- Add meaningful documentation

Expected Outcome

- Clean, well-structured code
- Descriptive function and variable names
- Proper indentation and formatting
- Docstrings explaining the function purpose
- AI explanation of readability improvements

The screenshot shows the PyLance AI-Assisted Python Code Review and Correction interface. On the left, the original code (Task3.py) is displayed with numerous errors underlined in red. The code contains several cryptic functions like f, g, h, and process, along with poor indentation and variable naming. On the right, the AI provides a detailed report and refactored code. The report highlights the following improvements:

- Descriptive function and variable names
- Proper indentation and formatting
- Docstrings explaining the function purpose
- AI explanation of readability improvements

The refactored code is shown in a table comparing the original and refactored versions, along with their respective improvements:

Original	Refactored	Improvement
f()	calculate_simple_interest()	Clear purpose: financial calculation
g()	calculate_percentage()	Explicit: percentage calculation
h()	calculate_list_statistics()	Obvious: multiple statistics operation
process()	apply_conditional_multiplier()	Descriptive: conditional operation
calc()	calculate_adjusted_value()	Clear: adjustment logic

A note at the bottom states: "Why it matters: A developer opening the code immediately understands what each function does without reading the implementation."

Below this, another table compares variable naming:

Original	Refactored	Context
p	principal	Loan/investment amount

The interface also includes a status bar at the bottom with various icons and a message: "Describe what to build next".

```
◆ Task3.py > ...
1  # BEFORE: Poorly written code - difficult to understand and maintain
2
3  def f(p,r,t):
4      a=p*(1+r*t)
5      return a
6
7  def g(x,y):
8      # Some calculation
9      b=x*y
10     c=b*100
11     return c
12
13  def h(lst):
14      tot=0
15      for i in lst:
16          tot+=i
17      mn=tot/len(lst)
18      mx=0
19      for i in lst:
20          if i>mx:
21              mx=i
22      mn2=0
23      for i in lst:
24          if i<mn or mn2==0:
25              mn2=i
26      return tot,mn,mx,mn2
27
28  def process(d):
29      """process some data"""
30      result=[]
31      for x in d:
32          if x>5:
33              result.append(x*2)
34          else:
35              result.append(x*3)
36      return result
37
38  def calc(a,b,c):
39      x=(a+b)*c
40      if x>100:
41          y=x*0.9
42      else:
43          y=x*1.1
44      return y
45
46  # Test the poorly written functions
47  print(f(1000,0.05,2))
48  print(g(85,100))
```

OUTPUT:

```
PS C:\Users\tloke\OneDrive\AI> & C:/Users/tloke/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/tloke/OneDrive\AI\Task3.py
1100.0
85.0
(15, 3.0, 5, 1)
[3, 12, 9, 16, 6]
99.00000000000001

--- REFACTORED: Clean, readable, maintainable code ---

Refactored Function Results:
=====
Simple Interest: $1100.00
Percentage: 85.0%
Statistics - Sum: 15, Avg: 3.0, Max: 5, Min: 1
Conditional Multiplier: [3, 12, 9, 16, 6]
Adjusted Value: 99.0
PS C:\Users\tloke\OneDrive\AI>
```

Task 4: Secure Coding and Reliability Review

Scenario

A backend function retrieves user data from a database but has security vulnerabilities and poor error handling, making it unsafe for production deployment.

Task Description

You are given a Python script that:

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

Use AI tools to:

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

Expected Outcome

- Secure SQL queries using parameterized statements
- Input validation logic
- Try-except blocks for runtime safety

- AI-generated explanation of security improvements

- Production-ready code structure

The screenshot shows a comparison between the original vulnerable code and the secure, refactored version. The left pane displays the original code with numerous red underlines and error markers. The right pane shows the improved code with green checkmarks indicating successful refactoring. A sidebar on the right provides a summary of the changes made.

```

Task4.py > ...
1 # BEFORE: VULNERABLE CODE - SQL Injection, No Validation, Poor Error Handling
2
3 import sqlite3
4
5 # UNSAFE: This is a DATABASE CONNECTION WITH HARDCODED CREDENTIALS - DO NOT USE IN PRODUCTION
6 def get_user_unsafe(user_id):
7     """
8         VULNERABLE: Susceptible to SQL injection attacks
9
10        Example attack: user_id = "1 OR 1=1 --" would return all users
11        """
12        conn = sqlite3.connect(':memory:')
13        cursor = conn.cursor()
14
15        # DANGER: String concatenation allows SQL injection
16        query = f"SELECT * FROM users WHERE id = {str(user_id)}"
17        cursor.execute(query)
18        result = cursor.fetchone()
19        conn.close()
20        return result
21
22
23 def find_user_by_username_unsafe(username):
24     """
25         VULNERABLE: No input validation, susceptible to SQL injection"""
26     conn = sqlite3.connect(':memory:')
27     cursor = conn.cursor()
28
29     # DANGER: Direct string interpolation
30     query = f"SELECT id, name, email FROM users WHERE username = '{username}'"
31     cursor.execute(query)
32     return cursor.fetchone()
33
34 def get_user_data_unsafe(user_id, database_path):
35     """
36         VULNERABLE: Multiple security issues
37         - No input validation
38         - String-based query construction
39         - No exception handling
40         - Database path not validated
41     """
42     conn = sqlite3.connect(database_path)
43     cursor = conn.cursor()
44
45     # DANGER: Vulnerable to SQL injection
46     query = f"SELECT * FROM users WHERE id = {user_id} AND age > 18"
47     cursor.execute(query)
48     result = cursor.fetchone()

```

OUTPUT:

```

PS C:\Users\tloke\OneDrive\AI> & C:/Users/tloke/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/tloke/OneDrive/AI/Task4.py
=====
VULNERABLE CODE EXAMPLES (DO NOT USE IN PRODUCTION)
=====

1. SQL Injection Risk: get_user_unsafe(user_id)
   - Attack: user_id = "1 OR 1=1 --" returns all users

2. SQL Injection Risk: find_user_by_username_unsafe(username)
   - Attack: username = "" OR '1'='1" bypasses authentication

3. No Validation: get_user_data_unsafe(user_id, db_path)
   - No type checking, no database path validation

4. Data Integrity Risk: batch_update_users_unsafe(ids, status)
   - No transaction handling, no rollback capability

=====
SECURE CODE - PRODUCTION READY
=====

Secure Code Examples:

✓ Test 1: validate_user_id(123)
  Result: 123 (type: int)

✓ Test 2: validate_user_id(-5) - Invalid (negative)
  Error caught: User ID must be a positive integer

✓ Test 3: validate_username('john_doe')
  Result: 'john_doe'

✓ Test 4: validate_username(" OR '1'='1") - SQL Injection Attempt
  Error caught: Username contains invalid characters

✓ Test 5: validate_database_path('./safe/db.sqlite')
  Result: safe\db.sqlite

✓ Test 6: validate_database_path('../etc/passwd') - Directory Traversal Attempt
  Error caught: Path traversal not allowed

=====
All security validations working correctly!
=====

PS C:\Users\tloke\OneDrive\AI>

```

Task 5: AI-Based Automated Code Review Report

Scenario Your team uses AI tools to perform automated preliminary code reviews before human review, to improve code quality and consistency across projects.

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.

Expected Outcome

- AI-generated review report including:
 - Identified quality issues
 - Risk areas
 - Code smell detection
 - Improvement suggestions
- Optional improved version of the code
- Demonstration of AI as a code reviewer, not just a code generator.

task5_Improved.py > ...

```

1  #!/usr/bin/env python3
2  """
3  REFACTORING CODE - Task5_Improved.py
4  Demonstrates how the original poorly-written code can be improved.
5  This version addresses critical security, readability, and maintainability issues.
6  """
7
8  import logging
9  from typing import Optional, List, Dict, Any
10 from dataclasses import dataclass
11 from pathlib import Path
12
13 # Configure structured logging
14 logging.basicConfig(
15     level=logging.INFO,
16     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
17 )
18 logger = logging.getLogger(__name__)
19
20 # Constants
21 DEFAULT_MAX_USERS = 1000
22 ADULT_AGE_THRESHOLD = 18
23
24
25 # Custom Exceptions
26 class UserManagementError(Exception):
27     """Base exception for user management operations"""
28     pass
29
30
31 class ValidationError(UserManagementError):
32     """Raised when input validation fails"""
33     pass
34
35
36 class UserNotFoundError(UserManagementError):
37     """Raised when a requested user is not found"""
38     pass
39
40
41 # Data Classes (instead of dictionaries)
42 @dataclass
43 class User:
44     """Represents a user in the system.
45
46     Attributes:
47         name: User's full name
48         email: User's email address (unique)

```

AI-ASSISTED PYTHON CODE REVIEW AND CORRECTION

Task 5: AI-Based Automated Code Review Complete

I've created a **comprehensive automated code review system** that demonstrates AI as a code analyzer and quality reviewer. Here's what was delivered:

Artifacts Created

- Task5.py - Poorly written user management script (intentionally created with issues)
- GenerateCodeReview.py - Automated review report generator that evaluates:
 - Readability (4 issues)
 - Naming conventions (4 issues)
 - Error handling (5 CRITICAL issues)
 - Code smells (8 anti-patterns)
 - Security vulnerabilities (4 issues)
 - Documentation quality (5 issues)
 - Maintainability & architecture (5 issue)
- Task5_Improved.py - Professional refactored version demonstrating best practices
- CodeReviewAnalysis.md - Detailed before/after comparison report

Analysis Results: 26 Total Issues Identified

Category	Issues	Severity
Critical	5	Bare except, no validation, plain text passwords, global state
High	7	Poor naming, file leaks, no logging, long functions
Medium	10	Code smells, type inconsistency, magic numbers
Low	4	PEP 8, spacing, documentation format

Overall Score: 3.5/10 X NOT PRODUCTION READY

Key Findings

Security Issues (CRITICAL)

- Passwords stored in plain text → ✓ Hashed with bcrypt
- No input validation → ✓ Type validation functions
- File path traversal risk → ✓ Path validation
- Bare except clauses → ✓ Specific exception handling

Task5_Improved.py

GenerateCodeReview.py > ...

```

1  #!/usr/bin/env python3
2  """
3
4  AUTOMATED CODE REVIEW REPORT - Task5.py Analysis
5  Generated: 2026-02-19
6  Reviewer: AI Code Analysis System
7  """
8
9  print("=" * 80)
10 print("AUTOMATED CODE REVIEW REPORT - Task5.py")
11 print("=" * 80)
12 print("\nEXECUTIVE SUMMARY")
13 print("-" * 80)
14 print(""""
15 Overall Code Quality Score: 3.5/10
16 Risk Level: HIGH - Not suitable for production without major refactoring.
17
18 KEY FINDINGS:
19 - 12 Critical/High severity issues identified
20 - 8 Code smell violations detected
21 - Inadequate error handling and logging
22 - Poor naming conventions throughout
23 - No input validation mechanisms
24 - Multiple security vulnerabilities
25 - Weak/missing documentation
26
27
28 print("\n" + "=" * 80)
29 print("SECTION 1: READABILITY ANALYSIS (4 Issues)")
30 print("=" * 80)
31
32 readability = [
33     ("R-001", "HIGH", "Cryptic function names with abbreviations",
34      ["add_a_new_user(n, e, a, p) - Parameter abbreviations",
35       "del_user() - Should be 'remove_user'",
36       "pwd, fname - Non-standard abbreviations"],
37      "Use descriptive names: 'add_user_to_system(name, email, age, password)'"),
38
39     ("R-002", "HIGH", "Minimal/unclear docstrings",
40      ['def add_a_new_user(...): """user function"" - Only 2 words',
41       "update_user_data, process_user_file - No docstrings at all"],
42      "Add comprehensive Google-style docstrings with Args, Returns, Raises"),
43
44     ("R-003", "MEDIUM", "Inconsistent code spacing and formatting",
45      ["No blank lines between logical sections",
46       "Inconsistent spacing around operators"],
47      "Use 'black' formatter for PEP 8 compliance"),

```

```

1 # =====
2 # AUTOMATED CODE REVIEW REPORT
3 # AI-Generated Analysis of Task5.py
4 # =====
5 # Generated: 2026-02-19
6 # Reviewer: AI Code Analysis System
7 # Severity Levels: 🚨 CRITICAL | 🛡 HIGH | 🟠 MEDIUM | 🟢 LOW | 🗃 INFO
8 # =====
9 """
10 """
11 EXECUTIVE SUMMARY
12 =====
13 The provided Python script demonstrates multiple quality issues across all
14 evaluation dimensions. Overall Code Quality Score: 3.5/10
15 Risk Level: HIGH - Not suitable for production without major refactoring.
16
17 KEY FINDINGS:
18 - 12 Critical/High severity issues
19 - 8 Code smell violations
20 - Inadequate error handling and logging
21 - Poor naming conventions
22 - No input validation
23 - Missing security measures
24 - Weak documentation
25 """
26
27 # =====
28 # SECTION 1: READABILITY ANALYSIS
29 # =====
30
31 print("=" * 80)
32 print("SECTION 1: READABILITY ANALYSIS")
33 print("=" * 80)
34
35 readability_issues = [
36     {
37         "ID": "R-001",
38         "Issue": "Cryptic function names with abbreviations",
39         "Severity": "🔴 HIGH",
40         "Examples": [
41             "add_a_new_user(n, e, a, p)",      - Parameters abbreviated: n, e, a, p",
42             "del_user()",                   - Abbreviated delete (should be 'remove')",
43             "pwd",                         - Abbreviated password",
44             "fname",                        - Abbreviated filename"
45         ],
46         "Impact": "Reduced comprehension. New developers waste time understanding intent.",
47         "Recommendation": "Use full, descriptive names: 'add_user_to_system()', parameters should be "
48         | | | | "'name', 'email', 'age', 'password'"

```

OUTPUT:

```
PS C:\Users\tloke\OneDrive\AI> & C:/Users/tloke/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/tloke/OneDrive\AI\Task5.py

===== MENU =====
1. Add User, 2. View All, 3. Find User
4. Delete User, 5. Update User, 6. Load From File
7. Show All Users, 8. Average Age, 9. Exit
Select: 1
Name: random
Email: randomshits
Age: 25
Password: Aswin@123
User added!

===== MENU =====
1. Add User, 2. View All, 3. Find User
4. Delete User, 5. Update User, 6. Load From File
7. Show All Users, 8. Average Age, 9. Exit
Select: 2
{'name': 'random', 'email': 'randomshits', 'age': 25, 'pass': 'Aswin@123'}

===== MENU =====
1. Add User, 2. View All, 3. Find User
4. Delete User, 5. Update User, 6. Load From File
7. Show All Users, 8. Average Age, 9. Exit
Select: 3
Email: randomshits
Found: {'name': 'random', 'email': 'randomshits', 'age': 25, 'pass': 'Aswin@123'}

===== MENU =====
1. Add User, 2. View All, 3. Find User
4. Delete User, 5. Update User, 6. Load From File
7. Show All Users, 8. Average Age, 9. Exit
Select: 8
Average age: 25.0

===== MENU =====
1. Add User, 2. View All, 3. Find User
4. Delete User, 5. Update User, 6. Load From File
7. Show All Users, 8. Average Age, 9. Exit
Select: 25
Invalid command

===== MENU =====
1. Add User, 2. View All, 3. Find User
4. Delete User, 5. Update User, 6. Load From File
7. Show All Users, 8. Average Age, 9. Exit
Select: 1
Name: sai
Email: sai@gmail.com
Age: 25
Password: SAI@123
```

```
===== MENU =====
1. Add User, 2. View All, 3. Find User
4. Delete User, 5. Update User, 6. Load From File
7. Show All Users, 8. Average Age, 9. Exit
Select: 7

All Users:
Name: random, Email: randomshits, Age: 25
Name: sai, Email: sai@gmail.com, Age: 25

===== MENU =====
1. Add User, 2. View All, 3. Find User
4. Delete User, 5. Update User, 6. Load From File
7. Show All Users, 8. Average Age, 9. Exit
Select: █
```

