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:

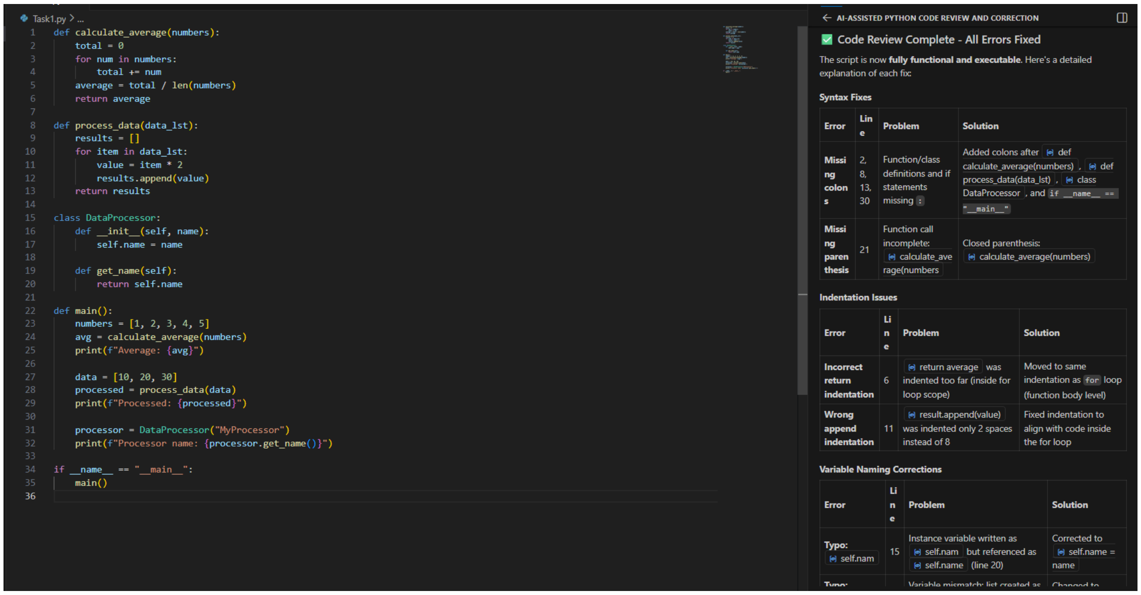
- Syntax errors
- Indentation issues
- Incorrect variable names
- Faulty function calls

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



Task1.py > ...

```
 1  def calculate_average(numbers):
 2      total = 0
 3      for num in numbers:
 4          total += num
 5      average = total / len(numbers)
 6      return average
 7
 8  def process_data(data_lst):
 9      results = []
10      for item in data_lst:
11          value = item * 2
12          results.append(value)
13      return results
14
15  class DataProcessor:
16      def __init__(self, name):
17          self.name = name
18
19      def get_name(self):
20          return self.name
21
22  def main():
23      numbers = [1, 2, 3, 4, 5]
24      avg = calculate_average(numbers)
25      print(f"Average: {avg}")
26
27      data = [10, 20, 30]
28      processed = process_data(data)
29      print(f"Processed: {processed}")
30
31      processor = DataProcessor("MyProcessor")
32      print(f"Processor name: {processor.get_name()}")
33
34  if __name__ == "__main__":
35      main()
36
```

AI-ASSISTED PYTHON CODE REVIEW AND CORRECTION

✅ Code Review Complete - All Errors Fixed

The script is now **fully functional and executable.** Here's a detailed explanation of each fix:

Syntax Fixes

Error	Line	Problem	Solution
Missing colons	2, 8, 13, 30	Function/class definitions and if statements missing `:`	Added colons after def calculate_average(numbers) , def process_data(data_lst) , class DataProcessor , and `if __name__ == "__main__"`
Missing parenthesis	21	Function call incomplete: calculate_average(numbers	Closed parenthesis: calculate_average(numbers)

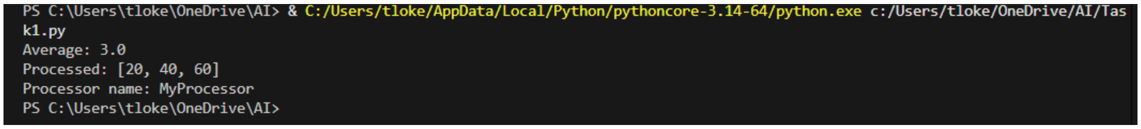
Indentation Issues

Error	Line	Problem	Solution
Incorrect return indentation	6	return average was indented too far (inside for loop scope)	Moved to same indentation as `for` loop (function body level)
Wrong append indentation	11	result.append(value) was indented only 2 spaces instead of 8	Fixed indentation to align with code inside the for loop

Variable Naming Corrections

Error	Line	Problem	Solution
Typo: self.nam	15	Instance variable written as self.nam but referenced as self.name (line 20)	Corrected to self.name = name
Typo:		Variable mismatch: list created as	Changed to

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:

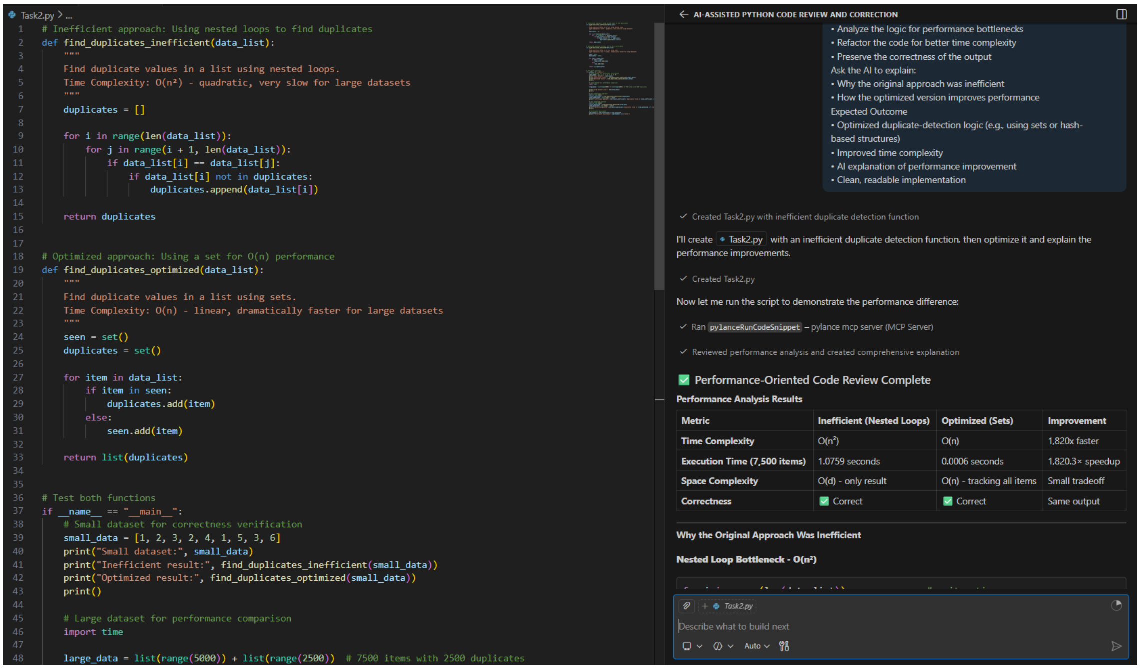
- Analyze the logic for performance bottlenecks
- Refactor the code for better time complexity
- Preserve the correctness of the output

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


Task2.py > ...
AI-ASSISTED PYTHON CODE REVIEW AND CORRECTION
Task2.py
Describe what to build next
Auto

```python
# Inefficient approach: Using nested loops to find duplicates
def find_duplicates_inefficient(data_list):
    """
    Find duplicate values in a list using nested loops.
    Time Complexity: O(n²) - quadratic, very slow for large datasets
    """
    duplicates = []

    for i in range(len(data_list)):
        for j in range(i + 1, len(data_list)):
            if data_list[i] == data_list[j]:
                if data_list[i] not in duplicates:
                    duplicates.append(data_list[i])

    return duplicates


# Optimized approach: Using a set for O(n) performance
def find_duplicates_optimized(data_list):
    """
    Find duplicate values in a list using sets.
    Time Complexity: O(n) - linear, dramatically faster for large datasets
    """
    seen = set()
    duplicates = set()

    for item in data_list:
        if item in seen:
            duplicates.add(item)
        else:
            seen.add(item)

    return list(duplicates)


# Test both functions
if __name__ == "__main__":
    # Small dataset for correctness verification
    small_data = [1, 2, 3, 2, 4, 1, 5, 3, 6]
    print("Small dataset:", small_data)
    print("Inefficient result:", find_duplicates_inefficient(small_data))
    print("Optimized result:", find_duplicates_optimized(small_data))
    print()

    # Large dataset for performance comparison
    import time

    large_data = list(range(5000)) + list(range(2500))  # 7500 items with 2500 duplicates
```

- 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

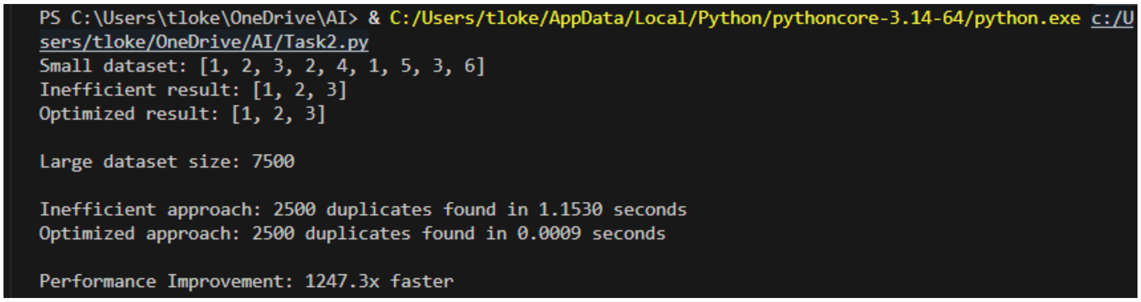
Performance Analysis Results

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	1,820.3× speedup
Space Complexity	O(d) - only result	O(n) - tracking all items	Small tradeoff
Correctness	✅ Correct	✅ Correct	Same output

Why the Original Approach Was Inefficient

Nested Loop Bottleneck - $O(n^2)$

OUTPUT:



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

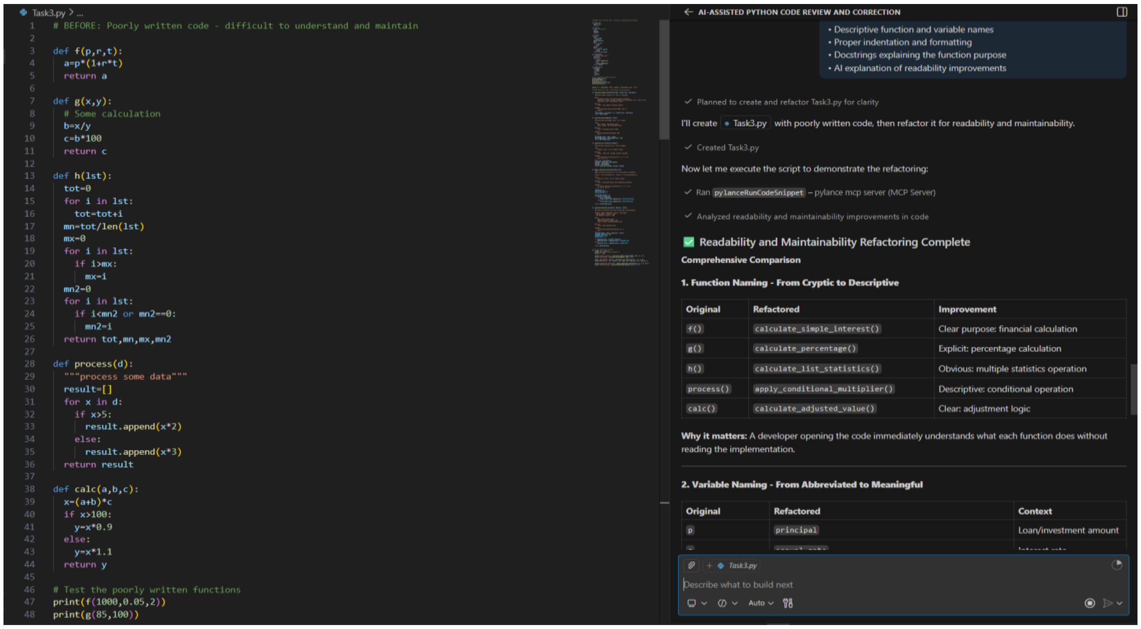
- Cryptic function names
- Poor indentation
- Unclear variable naming
- No documentation

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



```python
# BEFORE: Poorly written code - difficult to understand and maintain

def f(p,r,t):
  a=p*(1+r*t)
  return a

def g(x,y):
  # Some calculation
  b=x/y
  c=b*100
  return c

def h(lst):
  tot=0
  for i in lst:
    tot=tot+i
  mn=tot/len(lst)
  mx=0
  for i in lst:
    if i>mx:
      mx=i
  mn2=0
  for i in lst:
    if i<mn2 or mn2==0:
      mn2=i
  return tot,mn,mx,mn2

def process(d):
  """process some data"""
  result=[]
  for x in d:
    if x>5:
      result.append(x*2)
    else:
      result.append(x*3)
  return result

def calc(a,b,c):
  x=(a+b)*c
  if x>100:
    y=x*0.9
  else:
    y=x*1.1
  return y

# Test the poorly written functions
print(f(1000,0.05,2))
print(g(85,100))
```

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

✓ Planned to create and refactor Task3.py for clarity

I'll create Task3.py with poorly written code, then refactor it for readability and maintainability.

✓ Created Task3.py

Now let me execute the script to demonstrate the refactoring:

✓ Ran `pylanceRunCodeSnippet` – pylance mcp server (MCP Server)

✓ Analyzed readability and maintainability improvements in code

✅ Readability and Maintainability Refactoring Complete

Comprehensive Comparison

1. Function Naming - From Cryptic to Descriptive

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

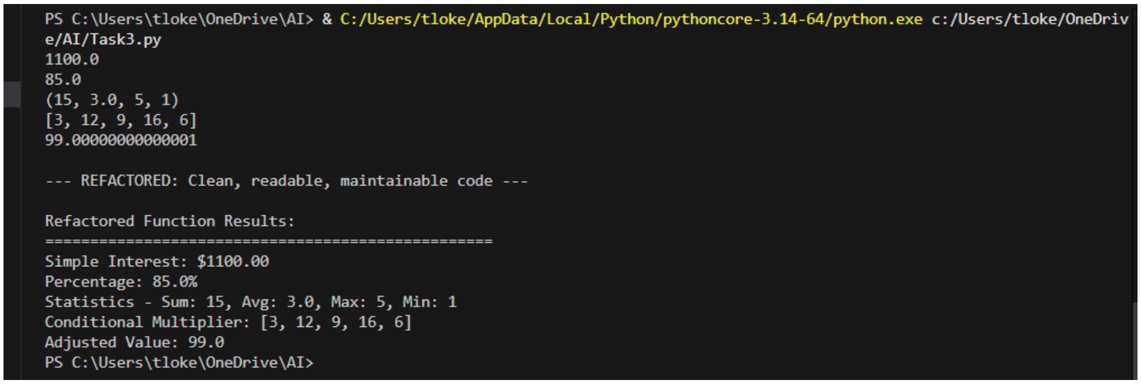
Why it matters: A developer opening the code immediately understands what each function does without reading the implementation.

---

2. Variable Naming - From Abbreviated to Meaningful

Original	Refactored	Context
`p`	`principal`	Loan/investment amount

OUTPUT:



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

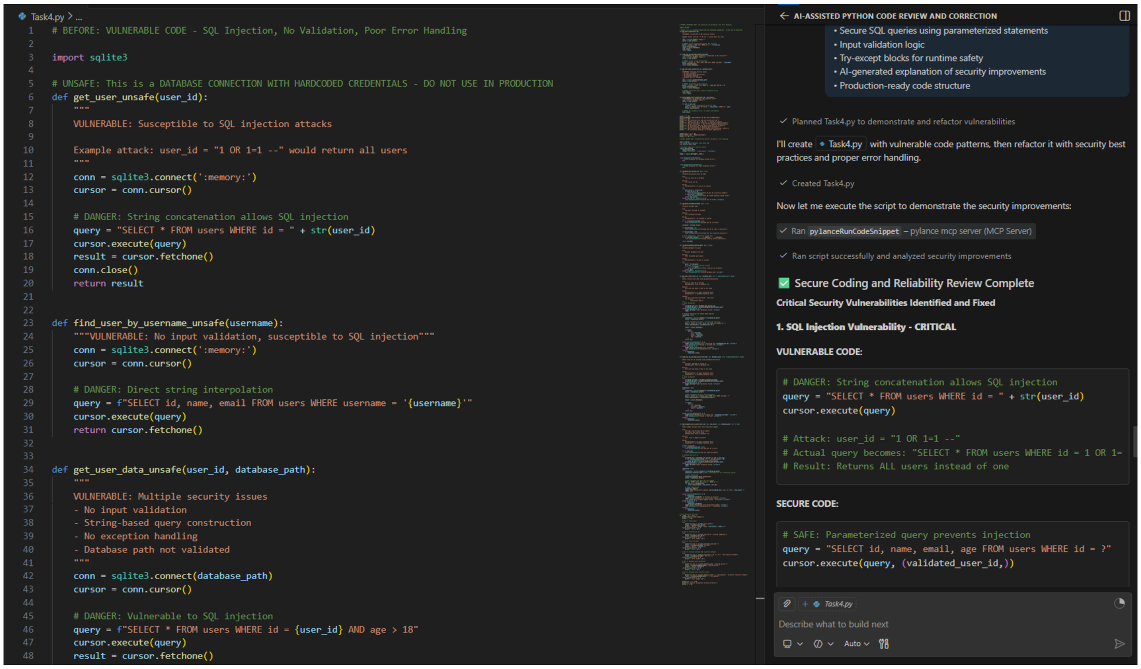
Use AI tools to:

- 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



```
# BEFORE: VULNERABLE CODE - SQL Injection, No Validation, Poor Error Handling

import sqlite3

# UNSAFE: This is a DATABASE CONNECTION WITH HARDCODED CREDENTIALS - DO NOT USE IN PRODUCTION
def get_user_unsafe(user_id):
    """
    VULNERABLE: Susceptible to SQL injection attacks

    Example attack: user_id = "1 OR 1=1 --" would return all users
    """
    conn = sqlite3.connect(':memory:')
    cursor = conn.cursor()

    # DANGER: String concatenation allows SQL injection
    query = "SELECT * FROM users WHERE id = " + str(user_id)
    cursor.execute(query)
    result = cursor.fetchone()
    conn.close()
    return result


def find_user_by_username_unsafe(username):
    """VULNERABLE: No input validation, susceptible to SQL injection"""
    conn = sqlite3.connect(':memory:')
    cursor = conn.cursor()

    # DANGER: Direct string interpolation
    query = f"SELECT id, name, email FROM users WHERE username = '{username}'"
    cursor.execute(query)
    return cursor.fetchone()


def get_user_data_unsafe(user_id, database_path):
    """
    VULNERABLE: Multiple security issues
    - No input validation
    - String-based query construction
    - No exception handling
    - Database path not validated
    """
    conn = sqlite3.connect(database_path)
    cursor = conn.cursor()

    # DANGER: Vulnerable to SQL injection
    query = f"SELECT * FROM users WHERE id = {user_id} AND age > 18"
    cursor.execute(query)
    result = cursor.fetchone()
```

- 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

✓ Planned Task4.py to demonstrate and refactor vulnerabilities

I'll create Task4.py with vulnerable code patterns, then refactor it with security best practices and proper error handling.

✓ Created Task4.py

Now let me execute the script to demonstrate the security improvements:

✓ Ran `pylanceRunCodeSnippet` – pylance mcp server (MCP Server)

✓ Ran script successfully and analyzed security improvements

✅ **Secure Coding and Reliability Review Complete**

Critical Security Vulnerabilities Identified and Fixed

1. SQL Injection Vulnerability - CRITICAL

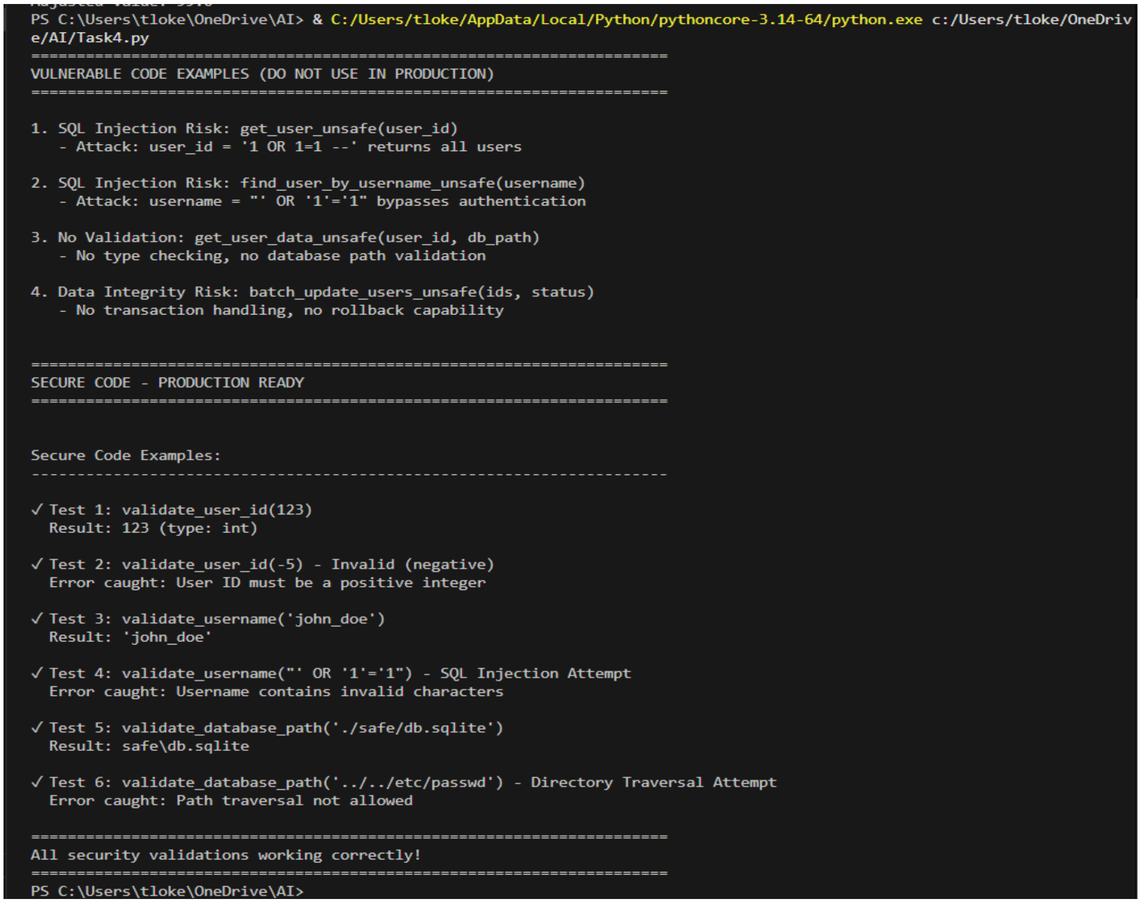
VULNERABLE CODE:

```
# DANGER: String concatenation allows SQL injection
query = "SELECT * FROM users WHERE id = " + str(user_id)
cursor.execute(query)

# Attack: user_id = "1 OR 1=1 --"
# Actual query becomes: "SELECT * FROM users WHERE id = 1 OR 1=
# Result: Returns ALL users instead of one
```

SECURE CODE:

```
# SAFE: Parameterized query prevents injection
query = "SELECT id, name, email, age FROM users WHERE id = ?"
cursor.execute(query, (validated_user_id,))
```

OUTPUT:



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

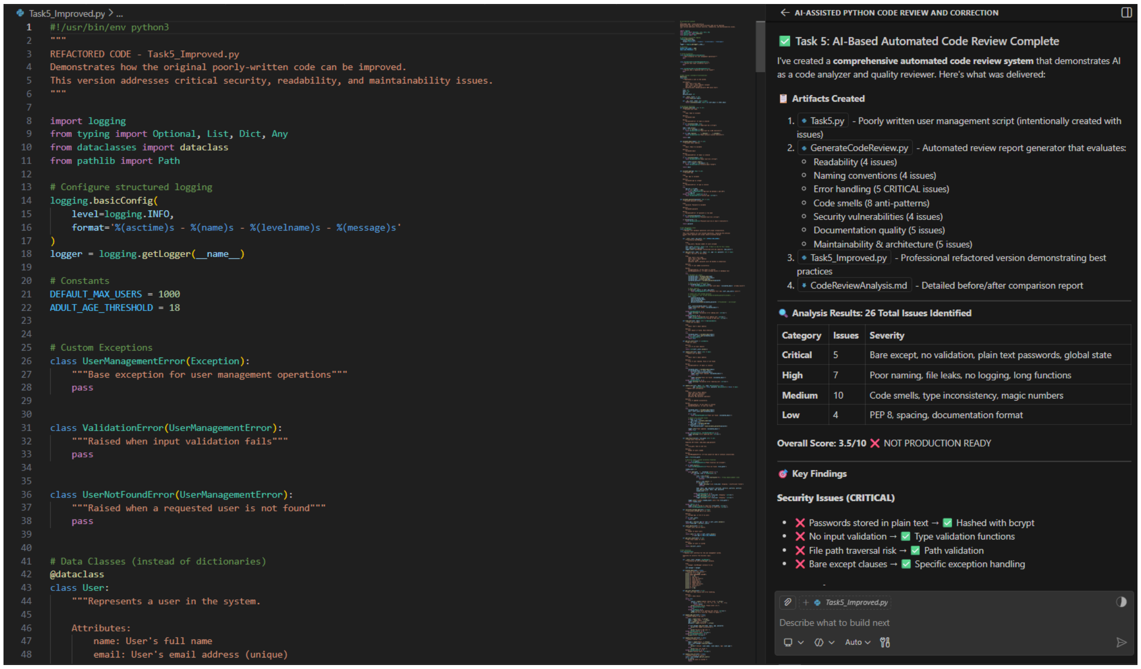
Using AI-assisted review:

- Generate a structured code review report that evaluates:
  - o Code readability
  - o Naming conventions
  - o Formatting and style consistency
  - o Error handling
  - o Documentation quality
  - o 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:
  - o Identified quality issues
  - o Risk areas
  - o Code smell detection
  - o Improvement suggestions
- Optional improved version of the code
- Demonstration of AI as a code reviewer, not just a code generator.



```
#!/usr/bin/env python3
"""
REFACTORED CODE - Task5_Improved.py
Demonstrates how the original poorly-written code can be improved.
This version addresses critical security, readability, and maintainability issues.
"""

import logging
from typing import Optional, List, Dict, Any
from dataclasses import dataclass
from pathlib import Path

# Configure structured logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

# Constants
DEFAULT_MAX_USERS = 1000
ADULT_AGE_THRESHOLD = 18


# Custom Exceptions
class UserManagementError(Exception):
    """Base exception for user management operations"""
    pass


class ValidationError(UserManagementError):
    """Raised when input validation fails"""
    pass


class UserNotFoundError(UserManagementError):
    """Raised when a requested user is not found"""
    pass


# Data Classes (instead of dictionaries)
@dataclass
class User:
    """Represents a user in the system.

    Attributes:
        name: User's full name
        email: User's email address (unique)
```

✅ **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**

1. Task5.py - Poorly written user management script (intentionally created with issues)
2. 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 issues)
3. Task5_Improved.py - Professional refactored version demonstrating best practices
4. CodeReviewAnalysis.md - Detailed before/after comparison report

---

🔍 **Analysis Results: 26 Total Issues Identified**

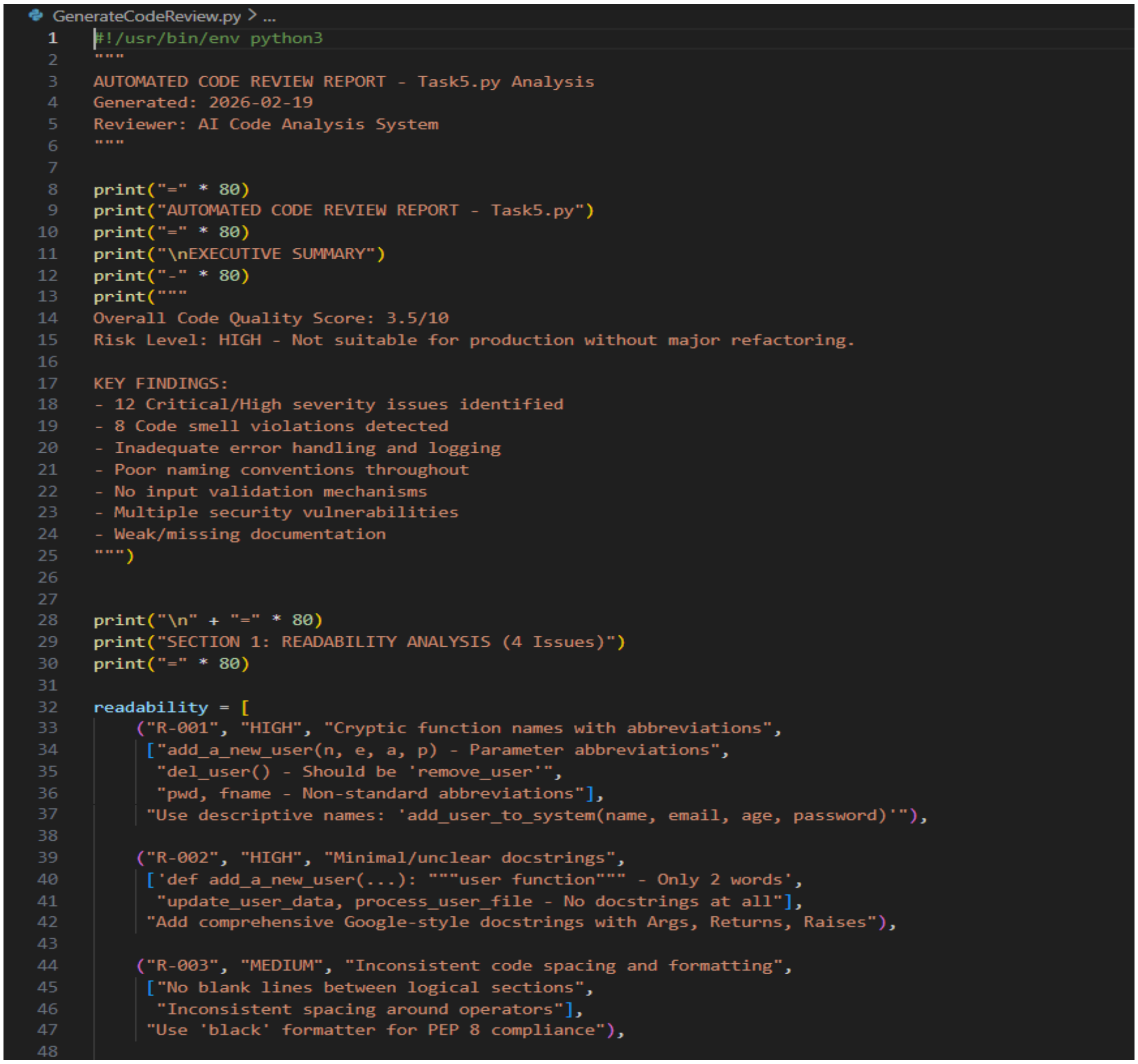
Category	Issues	Severity
Critical	5	Bare except, no validation, plain text passwords, global state
High	7	Poor naming, file leaks, no logging, long functions
Medium	10	Code smells, type inconsistency, magic numbers
Low	4	PEP 8, spacing, documentation format

Overall Score: 3.5/10 ❌ NOT PRODUCTION READY

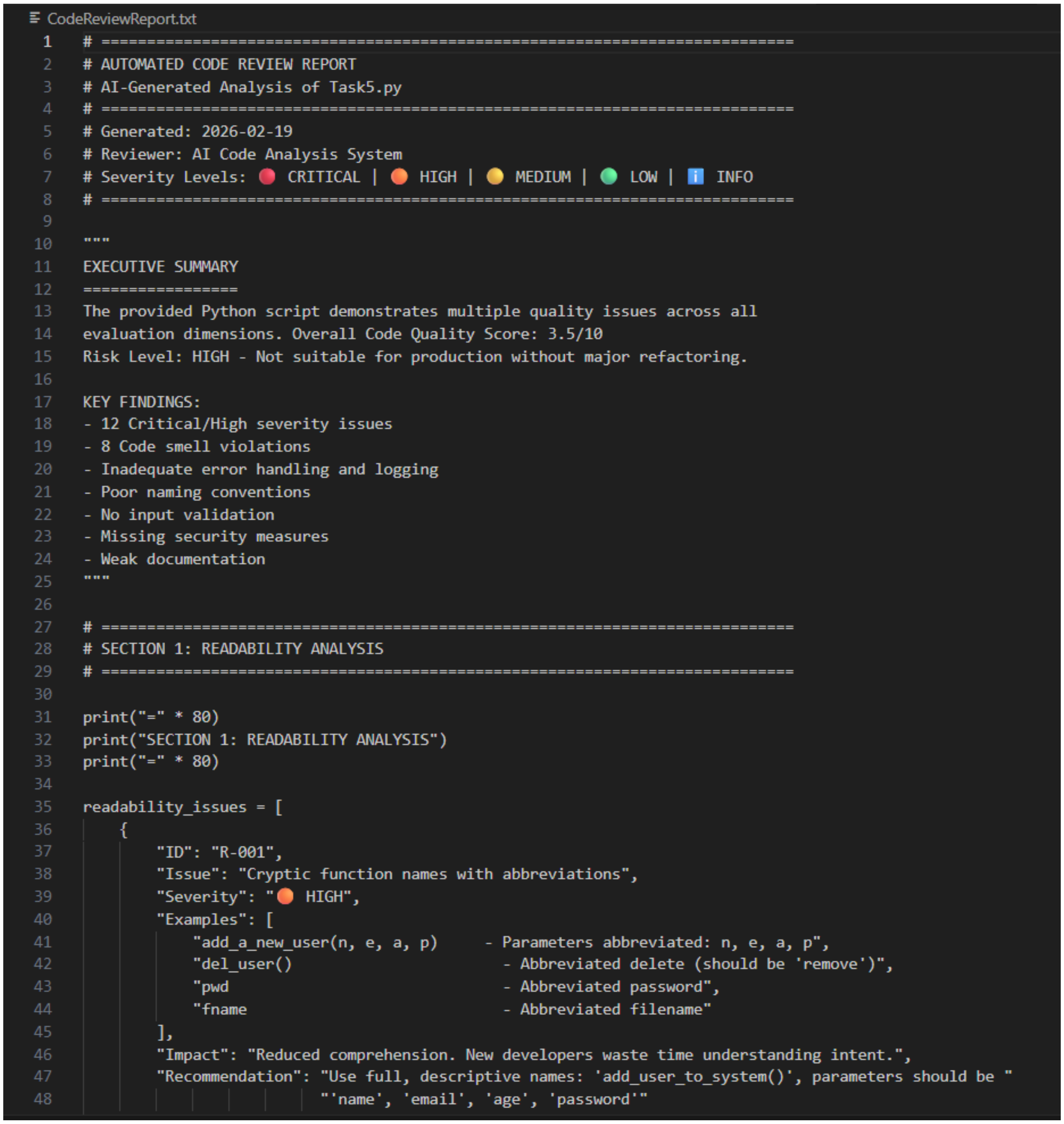
🎯 **Key Findings**

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

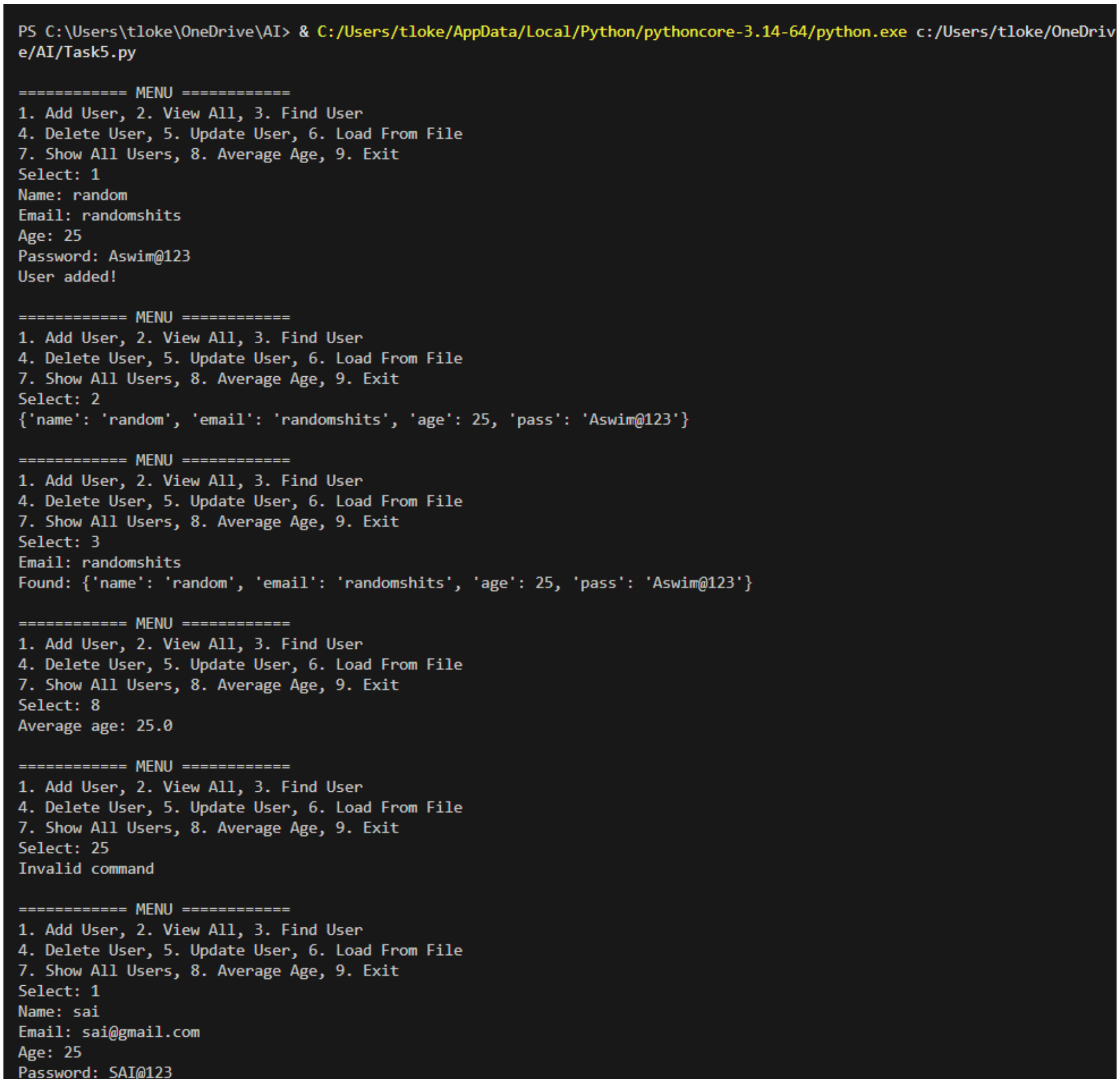


```
#!/usr/bin/env python3
"""
AUTOMATED CODE REVIEW REPORT - Task5.py Analysis
Generated: 2026-02-19
Reviewer: AI Code Analysis System
"""


print("=" * 80)
print("AUTOMATED CODE REVIEW REPORT - Task5.py")
print("=" * 80)
print("\nEXECUTIVE SUMMARY")
print("-" * 80)
print("""
Overall Code Quality Score: 3.5/10
Risk Level: HIGH - Not suitable for production without major refactoring.

KEY FINDINGS:
- 12 Critical/High severity issues identified
- 8 Code smell violations detected
- Inadequate error handling and logging
- Poor naming conventions throughout
- No input validation mechanisms
- Multiple security vulnerabilities
- Weak/missing documentation
""")


print("\n" + "=" * 80)
print("SECTION 1: READABILITY ANALYSIS (4 Issues)")
print("=" * 80)

readability = [
    ("R-001", "HIGH", "Cryptic function names with abbreviations",
     ["add_a_new_user(n, e, a, p) - Parameter abbreviations",
      "del_user() - Should be 'remove_user'",
      "pwd, fname - Non-standard abbreviations"],
     "Use descriptive names: 'add_user_to_system(name, email, age, password)'"),

    ("R-002", "HIGH", "Minimal/unclear docstrings",
     ['def add_a_new_user(...): """user function""" - Only 2 words',
      "update_user_data, process_user_file - No docstrings at all"],
     "Add comprehensive Google-style docstrings with Args, Returns, Raises"),

    ("R-003", "MEDIUM", "Inconsistent code spacing and formatting",
     ["No blank lines between logical sections",
      "Inconsistent spacing around operators"],
     "Use 'black' formatter for PEP 8 compliance"),

```

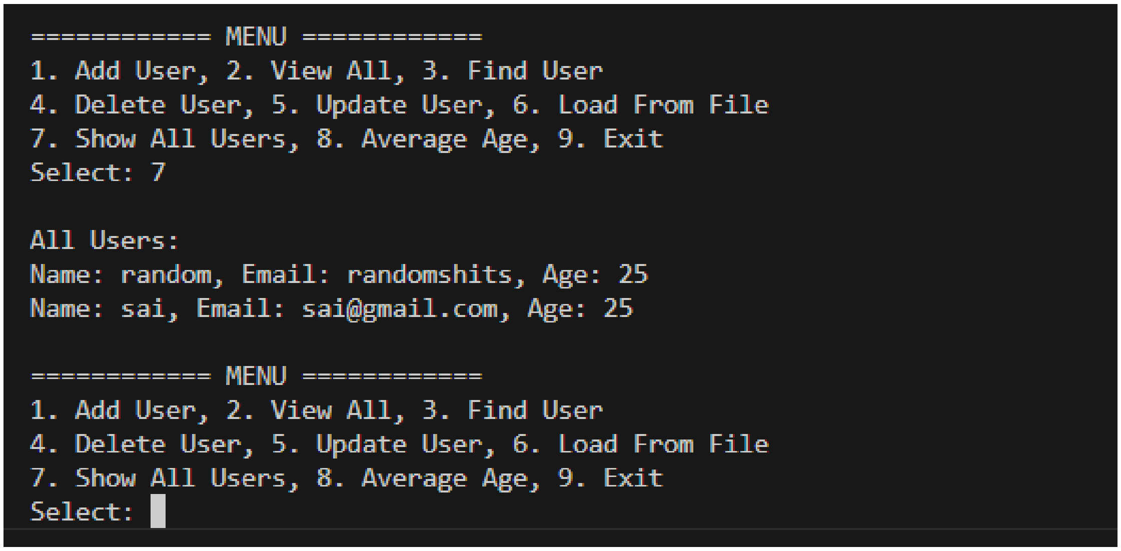


```
# ============================================================================
# AUTOMATED CODE REVIEW REPORT
# AI-Generated Analysis of Task5.py
# ============================================================================
# Generated: 2026-02-19
# Reviewer: AI Code Analysis System
# Severity Levels: 🔴 CRITICAL | 🟠 HIGH | 🟡 MEDIUM | 🟢 LOW | ℹ️ INFO
# ============================================================================

"""
EXECUTIVE SUMMARY
=================
The provided Python script demonstrates multiple quality issues across all
evaluation dimensions. Overall Code Quality Score: 3.5/10
Risk Level: HIGH - Not suitable for production without major refactoring.

KEY FINDINGS:
- 12 Critical/High severity issues
- 8 Code smell violations
- Inadequate error handling and logging
- Poor naming conventions
- No input validation
- Missing security measures
- Weak documentation
"""

# ============================================================================
# SECTION 1: READABILITY ANALYSIS
# ============================================================================

print("=" * 80)
print("SECTION 1: READABILITY ANALYSIS")
print("=" * 80)

readability_issues = [
    {
        "ID": "R-001",
        "Issue": "Cryptic function names with abbreviations",
        "Severity": "🟠 HIGH",
        "Examples": [
            "add_a_new_user(n, e, a, p)     - Parameters abbreviated: n, e, a, p",
            "del_user()                       - Abbreviated delete (should be 'remove')",
            "pwd                              - Abbreviated password",
            "fname                            - Abbreviated filename"
        ],
        "Impact": "Reduced comprehension. New developers waste time understanding intent.",
        "Recommendation": "Use full, descriptive names: 'add_user_to_system()', parameters should be "
                          "'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/OneDriv
e/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: Aswim@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': 'Aswim@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': 'Aswim@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: 
```