Lab 5: Static Code Analysis

Objective

To enhance Python code quality, security, and style by utilizing **static analysis tools** (Pylint, Bandit, and Flake8) to detect and rectify common programming issues.

Known Issue Table:

Issue	Type	Line(s)	Description	Fix Approach
eval usage	Security	59	Use of eval in code	Removed eval and directly wrote the code
Bare except	Exception Handling	19-20	Exception handler without specific exception type	Added specific exception types and error messages
Unused import	Import	2	Logging module imported but not used	Removed import
Missing main guard	Code Structure	62	Script execution without proper main guard	Added if __name__ == "__main__" guard
Missing module docstring	Documentation	1	Module lacks proper documentation	Added comprehensive module docstring

Dangerous default value	Mutable Default	8	Mutable list [] as default argument	Changed to None and initialize inside function
Missing encoding	File Operations	26,32	File operations without explicit encoding	Added encoding="utf-8" parameter
No value validation	Input Validation	Multiple	Functions don't validate input values	Added checks for empty strings, negative values

1. **Which issues were the easiest to fix, and which were the hardest? Why?**

   Easy

   a. **Trailing whitespace - Simple find/replace operation, no logic changes needed**
   b. **Missing encoding in file operations - Just add encoding="utf-8" parameter**
   c. **Unused imports - Delete the unused import line**
   d. **Missing docstrings - Add documentation, no code logic affected**
   e. **String formatting - Replace % with f-strings, straightforward syntax change**

These were easy because they required minimal changes to existing logic and had clear, mechanical fixes.

Hard

   f. **Too many return statements - Required complete function restructuring, changing control flow from multiple exit points to single return with validation flags**
   g. **Complex validation logic - Needed to create helper functions and reorganize scattered validation into cohesive patterns**
   h. **Dangerous default mutable arguments - Required understanding of Python's default argument evaluation and changing function signatures**
   i. **Global statement handling - Needed to balance pylint warnings with legitimate global variable usage**

These were difficult because they required architectural changes and deep understanding of Python best practices, not just syntax fixes.

2. **Did the static analysis tools report any false positives?** If so, describe one example.
    a. Yes, there was one notable false positive:
    Global statement warning in load_data() function - Pylint flagged the global stock_data statement as problematic, but it was necessary for the function's purpose of updating the global inventory. This is a legitimate use case where we need to modify global state, so I added a pylint disable comment rather than "fixing" it.
3. **How would you integrate static analysis tools into your actual software development workflow?** Consider continuous integration (CI) or local development practices.
    a. Local Development:
        i. Pre-commit hooks running pylint on changed files
        ii. IDE integration (VS Code pylint extension) for real-time feedback
        iii. Local scripts to run pylint before pushing code
        iv. Git hooks to prevent commits below a certain quality threshold
    b. Continuous Integration:
        i. Add pylint checks to CI pipeline (GitHub Actions, Jenkins, etc.)
        ii. Fail builds if pylint score drops below 9.0/10
        iii. Generate pylint reports as CI artifacts
        iv. Use pylint-fail-under parameter to enforce minimum standards
        v. Integrate with code review tools to show quality metrics
    c. Team Workflow:
        i. Establish team coding standards based on pylint rules
        ii. Regular code quality reviews focusing on static analysis results
        iii. Mentoring junior developers using pylint feedback as teaching moments
4. **What tangible improvements did you observe in the code quality, readability, or potential robustness after applying the fixes?**

    a. Code Quality:

        i. Score improved from unknown baseline to perfect 10.00/10

        ii. Eliminated 25+ distinct categories of issues

        iii. Standardized coding conventions across the entire codebase

    b. Readability:

        i. Consistent snake_case naming makes functions more discoverable

        ii. Comprehensive docstrings make the code self-documenting

        iii. Clear error messages improve debugging experience

        iv. Single return points make control flow easier to follow

c. **Robustness:**

    i. **Type validation prevents runtime TypeErrors**

    ii. **Input validation catches edge cases (empty strings, negative numbers)**

    iii. **Proper exception handling prevents silent failures**

    iv. **Resource management with context managers prevents file handle leaks**

    v. **JSON validation prevents data corruption from malformed files**

d. **Maintainability:**

    i. **Helper functions reduce code duplication**

    ii. **Consistent error handling patterns across all functions**

    iii. **Clear separation between validation and business logic**

    iv. **Standardized return value patterns (boolean success indicators)**

The most significant improvement was transforming the code from a basic script to production-ready software with comprehensive error handling and validation.