SE Lab 5

Name: Paruchuri Anshul
SRN: PES2UG23CS405
Section: G

Known Issue Table

*Issues highlighted in Bold have been fixed
Repo link: https://github.com/amateurmonke/SE-Lab-5

Issue	Type	Line(s)	Description	Fix Approach
Mutable default arg	**Bug**	**8**	**W0102: Dangerous default value [] The list is shared across all function calls.**	**Change the default to None. Inside the function, add: if arg is None: arg = [].**
Insecure eval() use	**Security**	**59**	**B307: Use of possibly insecure function W0123: Use of eval eval() can execute arbitrary code.**	**Replace eval() with the safer ast.literal_eval() if you are only evaluating simple literals. Otherwise, refactor to avoid evaluating strings as code.**
Bare except	**Bug / Robustness**	**19**	**E722: do not use bare 'except' W0702: No exception type(s) specified B110: Try, Except, Pass detected This catches all errors, including system-exit signals, and silences them, hiding bugs.**	**Specify the exact exception(s) you intend to catch (e.g., except KeyError:). Avoid using pass unless you are explicitly and safely ignoring an error.**

Unsafe file handling	**Robustness**	**26, 32**	**R1732: Consider using 'with' W1514: Using open without explicitly specifying an encoding Files may not be closed properly, and lack of an encoding can cause errors on different OSes.**	**Use a with block for opening files (e.g., with open(..., 'r', encoding='utf-8') as f:) to ensure they are always closed.**
Unused import	**Cleanup**	**2**	**F401: 'logging' imported but unused W0611: Unused import logging Unnecessary import clutters the namespace.**	**Remove the import logging line.**
Naming convention	**Style**	**8, 14, 22, 25, 31, 36, 41**	**C0103: Function name ... doesn't conform to snake_case Uses camelCase (e.g., addItem) instead of the Python standard snake_case.**	**Rename functions to use snake_case (e.g., add_item, remove_item, get_qty).**
Code formatting	**Style**	**12, 8, 14, 22, 25, 31, 36, 41, 48, 61**	**E302/E305: expected 2 blank lines... C0209: consider-using-f-string Inconsistent vertical whitespace and outdated string formatting.**	**Run an auto-formatter like black or autopep8. Change string formatting (like "...".format(x)) to f-strings (e.g., f"..."{x}").**
Use of global	Refactor	27	W0603: Using the global statement Using global makes code state hard to track and test.	Refactor loadData to return the loaded data. Pass the data as an argument to functions that need it, or encapsulate the logic in a class.

Missing docstrings	Documentation	1, 8, 14, 22, 25, 31, 36, 41, 48	C0114: Missing module docstring C0116: Missing function... docstring Code lacks documentation, making it hard to maintain.	Add PEP 257 compliant docstrings to the module and all functions.

Reflections

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

- **Easiest:** The flake8 formatting issues and the unused import were the easiest. These are purely mechanical fixes where the tool tells you the exact line and what to do (e.g., "add two blank lines"). They require no logical thinking or understanding of the program's flow.
- **Hardest:** The dangerous-default-value (mutable default argument) was arguably the hardest. While the fix itself is small (changing [] to None), understanding *why* it's a critical bug requires deeper knowledge of how Python evaluates function defaults only once at definition time. The use-of-global issue is also conceptually hard because the "correct" fix isn't a simple change but a larger architectural refactor, like converting the script into a class.

2. Did the static analysis tools report any false positives? If so, describe one example.

No, in this specific lab, all the reports from pylint, bandit, and flake8 were accurate and pointed to legitimate issues.

- The eval-used (B307) was a real, high-priority security risk.
- The dangerous-default-value (W0102) was a subtle but critical bug.
- The bare-except (E722) was a genuine anti-pattern that was hiding bugs (like the KeyError).
- All the flake8 style violations were correct and fixing them made the code objectively cleaner.

3. How would you integrate static analysis tools into your actual software development workflow?

I would integrate them at two key points:

- **Local Development:** First, by integrating the linters directly into the code editor (like VS Code) to provide real-time feedback with squiggly lines. Second, I would use

pre-commit hooks. This would automatically run tools like `flake8` and `bandit` every time I try to make a commit, preventing bad code from even entering the repository.

- **Continuous Integration (CI):** I would add a "Lint & Test" stage to the CI pipeline (e.g., in GitHub Actions). This step would run all the static analysis tools. If any high-severity issues are found, the build would fail, which blocks the pull request from being merged. This enforces a consistent quality standard for the entire team.

4. What tangible improvements did you observe in the code quality, readability, or potential robustness after applying the fixes?

The improvements were significant and covered every category:

- Robustness: The code is far more resilient. It no longer crashes if you try to get a non-existent item (get_qty) or if the inventory.json file is missing (load_data). It also correctly catches *only* the specific KeyError in remove_item and properly closes files using the with statement, preventing resource leaks.
- Security: A critical vulnerability (eval()) was completely removed, making the script much safer.
- Readability: The code is much easier to read. All functions now follow the standard snake_case naming convention, and the entire file adheres to flake8 formatting rules for spacing, line length, and indentation.
- Correctness: We fixed a major, hidden bug by correcting the mutable default arg in add_item, which now correctly creates a new log list for each call.