Software Engineering

UE23CS341A

5th Semester, Academic Year 2023

Date: 27/10/2025

Name: Roshini Ramesh	SRN: PES1UG23CS488	Section: H

LAB 4: STATIC CODE ANALYSIS – SELF REFLECTION QUESTIONS

1. **Which issues were the easiest to fix, and which were the hardest? Why?**
    - Easiest fixes:
        - Formatting and style issues like line length, missing docstrings, inconsistent naming, and unused imports were the easiest to fix. Flake8 and Pylint made it easy and straightforward since they clearly pointed to the offending lines.
    - Hardest fixes:
        - Broad exception handling was the trickiest because it required understanding what exceptions were actually possible and replacing except Exception: with specific exceptions (KeyError, TypeError, ValueError).

2. **Did the static analysis tools report any false positives? If so, describe one example.**
    - Yes, there was a false positive from Pylint regarding “variable logs might be uninitialized.”
    - In the original version, logs was an optional argument that defaulted to None, but later reassigned to an empty list if None. Pylint incorrectly flagged it as a potential uninitialized variable.
    - The logic was safe, so it was a false positive — but the fix (explicit type annotation and Optional[List[str]]) made the intent clearer.

3. **How would you integrate static analysis tools into your actual software development workflow? Consider continuous integration (CI) or local development practices.**
    - Local Development:
        - Configure pre-commit hooks to automatically run flake8, pylint, and bandit before every commit. This ensures that code pushed to the repo already meets the quality baseline.

    - Use VS Code or PyCharm integrations to run linting continuously while editing.
- Continuous Integration (CI):
    - Integrate static analysis in a GitHub Actions or GitLab CI pipeline with stages like:

      ```
      name: Run Flake8

      run: flake8 .

      name: Run Pylint

      run: pylint inventory_system.py

      name: Run Bandit

      run: bandit inventory_system.py
      ```

    - Fail the pipeline if the linting score drops below a threshold (e.g., 9/10) or if type errors occur.
    - Combine results into a code quality dashboard or PR comments for visibility.

4. **What tangible improvements did you observe in the code quality, readability, or potential robustness after applying the fixes?**
    - Improved Readability:
        - Consistent naming, type hints, and docstrings make the code easier to understand and maintain.
        - Line wrapping and logical grouping of functions improved visual clarity.
    - Increased Robustness:
        - Narrowing exception handling and adding explicit input validation prevented silent runtime errors.
        - Type hints (Dict[str, int], Optional[List[str]]) reduced potential type misuse and improved static safety.
    - Better Maintainability:
        - The structured error messages and consistent logging approach simplify debugging.