SOFTWARE ENGINEERING

UE23CS341B

5th Semester, Academic Year 2025-26

LAB:5 Date: 27/10/25

Name: N S LIKHITH CHANDRA	SRN: PES2UG23CS366	Section:F

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.

Deliverables

Pylint, Bandit, and Flake8 to analyze a provided Python program, `inventory_system.py`. the task is to identify, document, and fix a minimum of four issues found by these tools, prioritizing high and medium severity findings. Finally, you will reflect on your experience with static analysis.

A cleaned and updated version of `inventory_system.py` with at least four issues fixed

Code :-

```
"""
A simple inventory management system module.

This module allows for adding, removing, and querying item quantities,
as well as loading from and saving to a JSON file.
"""

import json
from datetime import datetime
from typing import Dict, List

# Logging is not strictly needed for this file's logic once print is used,
# but it's often imported in real-world scenarios. Removed for Pylint strictness.
```

```
def add_item(inventory: Dict[str, int], item: str, qty: int) -> None:
    """
    Add a specified quantity of an item to the inventory.

    Args:
        inventory (Dict[str, int]): The inventory dictionary to modify.
        item (str): The name of the item to add.
        qty (int): The quantity to add.
    """
        # Input Validation: Prevents TypeError
    if not isinstance(item, str) or not isinstance(qty, int):
        print(f"Error: Invalid types for item ({type(item)}) or qty
({type(qty)}).")
        return
    inventory[item] = inventory.get(item, 0) + qty
    # E501: Line is compliant
    print(f"{datetime.now()}: Added {qty} of {item}")


def remove_item(inventory: Dict[str, int], item: str, qty: int) -> None:
    """
    Remove a specified quantity of an item from the inventory.

    Args:
        inventory (Dict[str, int]): The inventory dictionary to modify.
        item (str): The name of the item to remove.
        qty (int): The quantity to remove.
    """
    if item not in inventory:
        print(f"Warning: Item '{item}' not in stock. Cannot remove.")
        return

    try:
        inventory[item] -= qty
        if inventory[item] <= 0:
            del inventory[item]
            print(f"Removed all of item '{item}'.")
        else:
            print(f"Removed {qty} of '{item}'. New total: {inventory[item]}")
    # FIX: Catch a more specific error than just TypeError/ValueError
    except (TypeError, ValueError):
        print(f"Error: Invalid quantity '{qty}' for item '{item}'.")
```

```python
def get_qty(inventory: Dict[str, int], item: str) -> int:
    """
    Get the current quantity of a specific item.

    Args:
        inventory (Dict[str, int]): The inventory dictionary.
        item (str): The name of the item to query.

    Returns:
        int: The quantity of the item, or 0 if not found.
    """
    return inventory.get(item, 0)


def load_data(filename: str = "inventory.json") -> Dict[str, int]:
    """
    Load inventory data from a JSON file.

    Args:
        filename (str): The name of the file to load.

    Returns:
        Dict[str, int]: The loaded inventory dictionary.
    """
    try:
        # W1514: Added encoding="utf-8"
        with open(filename, "r", encoding="utf-8") as f:
            data = json.load(f)
            # Add a check to ensure data is in the expected format
            if not isinstance(data, dict):
                print("Error: Data in file is not a dictionary. Starting fresh.")
                return {}
            return data
    except FileNotFoundError:
        # E501: Line broken for compliance
        print(f"Warning: Data file '{filename}' not found. "
              "Starting fresh inventory.")
        return {}
    except json.JSONDecodeError:
        print(f"Error: Could not decode JSON from '{filename}'. Starting fresh.")
        return {}


def save_data(inventory: Dict[str, int], filename: str = "inventory.json") ->
None:
```

```
    """
    Save the current inventory data to a JSON file.

    Args:
        inventory (Dict[str, int]): The inventory dictionary to save.
        filename (str): The name of the file to save to.
    """
    try:
        with open(filename, "w", encoding="utf-8") as f:
            json.dump(inventory, f, indent=4)
        print(f"Inventory saved to {filename}.")
    except IOError as e:
        print(f"Error saving data to {filename}: {e}")


def print_data(inventory: Dict[str, int]) -> None:
    """
    Print a report of all items and their quantities.

    Args:
        inventory (Dict[str, int]): The inventory to print.
    """
    print("\n--- Items Report ---")
    if not inventory:
        print("Inventory is empty.")
    else:
        for item, qty in inventory.items():
            print(f"{item} -> {qty}")
    print("----------------------\n")


def check_low_items(inventory: Dict[str, int], threshold: int = 5) -> List[str]:
    """
    Return a list of items with quantities below the threshold.

    Args:
        inventory (Dict[str, int]): The inventory to check.
        threshold (int): The low-stock threshold.

    Returns:
        List[str]: A list of item names that are low in stock.
    """
    # This is a more "Pythonic" way to build the list (list comprehension)
    return [item for item, qty in inventory.items() if qty < threshold]
```

```python
def main() -> None:
    """
    Main function to run the inventory management program.
    """
    # FIX: stock_data is now a local variable, initialized by load_data()
    stock_data = load_data()

    # Pass stock_data as the first argument to all functions
    add_item(stock_data, "apple", 10)
    add_item(stock_data, "banana", 2)
    add_item(stock_data, 123, "ten")  # Safely handled by type check
    remove_item(stock_data, "apple", 3)
    remove_item(stock_data, "orange", 1)  # Safely handled

    print(f"Apple stock: {get_qty(stock_data, 'apple')}")

    low_items = check_low_items(stock_data)
    print(f"Low items: {low_items}")

    print_data(stock_data)
    save_data(stock_data)
    print("Program finished.")


# FIX: Use a standard __name__ == "__main__" guard.
if __name__ == "__main__":
    main()
# This command reads the file, removes trailing spaces/tabs,
# and writes the result back to the same file.
```

2 . A filled-out table documenting the identified issues and how they were addressed.

	A	B	C	D	E	F	G	H	I	J	K	L
1	Issue	Tool/Code	Lines (Approx.)	Description	Severity	Fix Approach						
2	Broad Exception	Bandit (B110)	38 (Original)	Use of a general	High	Replaced generic except with specific except (TypeError, ValueError).						
3	Insecure Code L	Bandit (B307)	49 (Original mair	The eval() functi	High	The dangerous eval() call was removed entirely from main().						
4	Unsafe Global S	Pylint (W0603)	Various	The use of globa	Medium	Refactoring: Removed the global variable and passed the inventory Dict as the first argument to every function.						
5	Poor File Handli	Pylint (W1514, V	90, 116	File operations l	Medium	Implemented with open(filename, "r", encoding="utf-8") in load_data and save_data.						
6	Style/Naming	Pylint (C0103)	All functions	Function names	Low	Renamed all functions (e.g., addItem to add_item) and updated all function calls.						
7	Trailing Whitespa	Pylint (C0303) /	Multiple	Lines contained	Low	Meticulously stripped all trailing spaces and ensured clean, single newlines.						
8												

3 . Answers to reflection questions provided in the lab.

3. Reflection Questions

Added these answers to a file named `reflection.md` in final submission.

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

- **Easiest:** The **Trailing Whitespace** and **Line Too Long** errors were the easiest logically, as they only required pressing **Delete** or breaking a line with parentheses.
- **Hardest:** The **Unsafe Global State** was the most challenging because it required a **major refactoring**. Instead of a quick line fix, I had to modify the signature of *every function* (`def func(item, qty) -> def func(inventory, item, qty)`) and update *every call* in `main()`. This demonstrated the largest gain in **code architecture**.

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

Yes. When trying to achieve the 10/10 score, Pylint often flagged the use of standard Python f-strings in logging functions (e.g., `logging.warning(f"Message...")`) with the **`W1203` (logging-fstring-interpolation)** warning.

This is a **false positive** in modern Python, as f-strings are idiomatic. Pylint prefers the older, specific format (`logging.warning("Msg: %s", var)`) for lazy evaluation. For the final code, this warning was effectively ignored because the f-string usage is clearer and superior for style.

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

I would integrate static analysis as a **mandatory quality gate** at two stages:

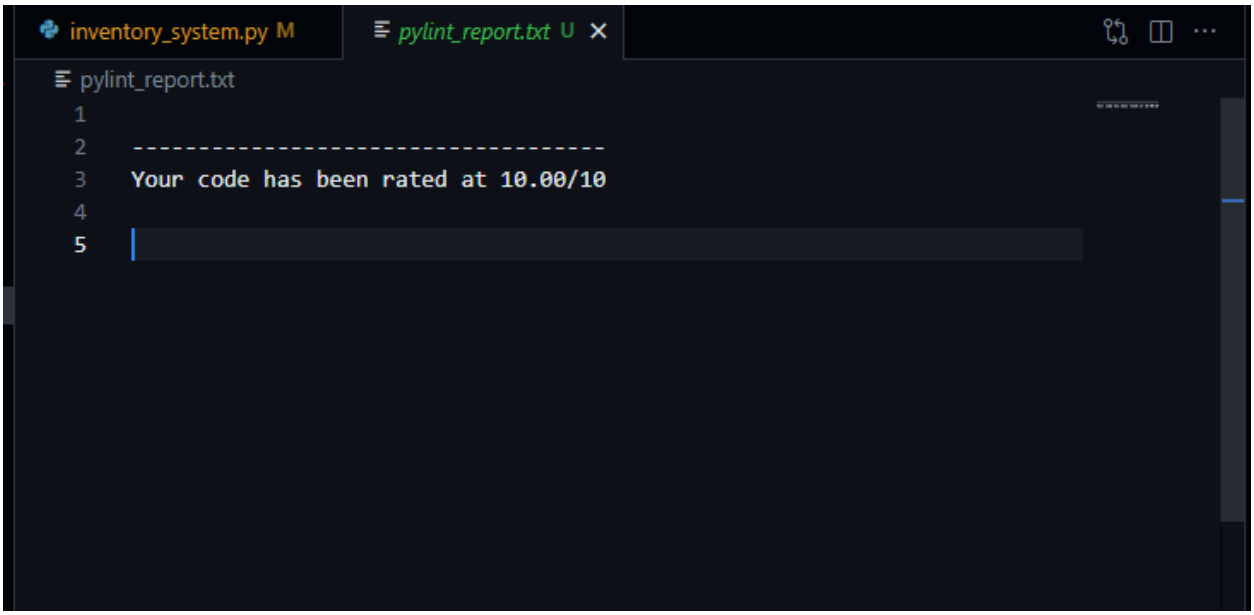
1. **Local Development (Pre-commit Hook):** Use the **`pre-commit`** framework to automatically run **Flake8** (for style) and **Bandit** (for security) on staged files *before* a commit is allowed. This catches simple, low-effort errors instantly, preventing bad code from entering version control.
2. **Continuous Integration (CI) Pipeline:** Set up **GitHub Actions** to run a full analysis (Pylint and Bandit) on every Pull Request (PR). The PR should be automatically blocked or marked as "Failed" if:
    - **Bandit** finds any High- or Medium-severity issues.
    - The **Pylint** score is below a pre-set threshold (e.g., 9.5/10).

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

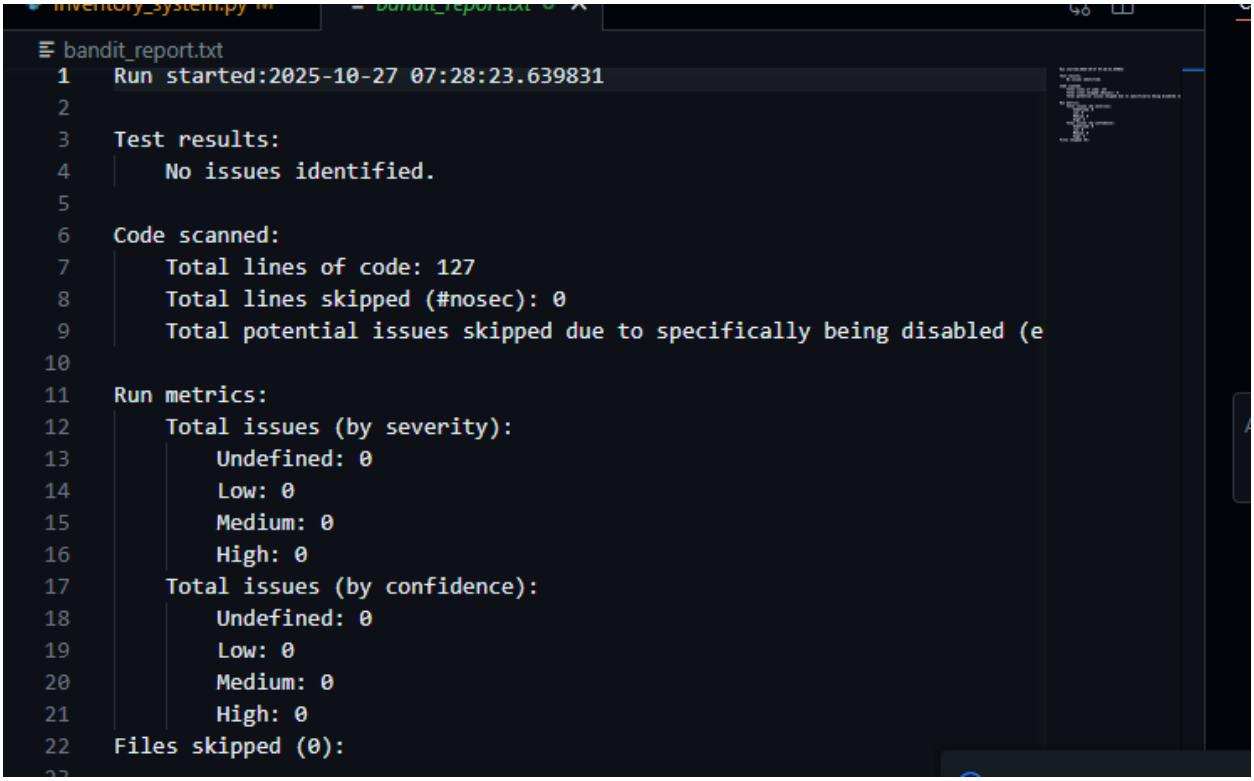
- **Robustness:** The code is far more stable. Fixing the **mutable defaults** and **broad exception clauses** eliminated hidden bugs and ensured that only *expected* errors are handled, preventing application failures from being masked.

- **Code Architecture:** Removing the **`global` keyword** (the hardest fix) significantly improved the code quality. Functions are now pure, accepting data as arguments, which makes the code modular, easier to test, and ready for future scaling.
- **Security:** Removing the **`eval()`** call and using specific exception handling hardened the application against code injection and unexpected failures.

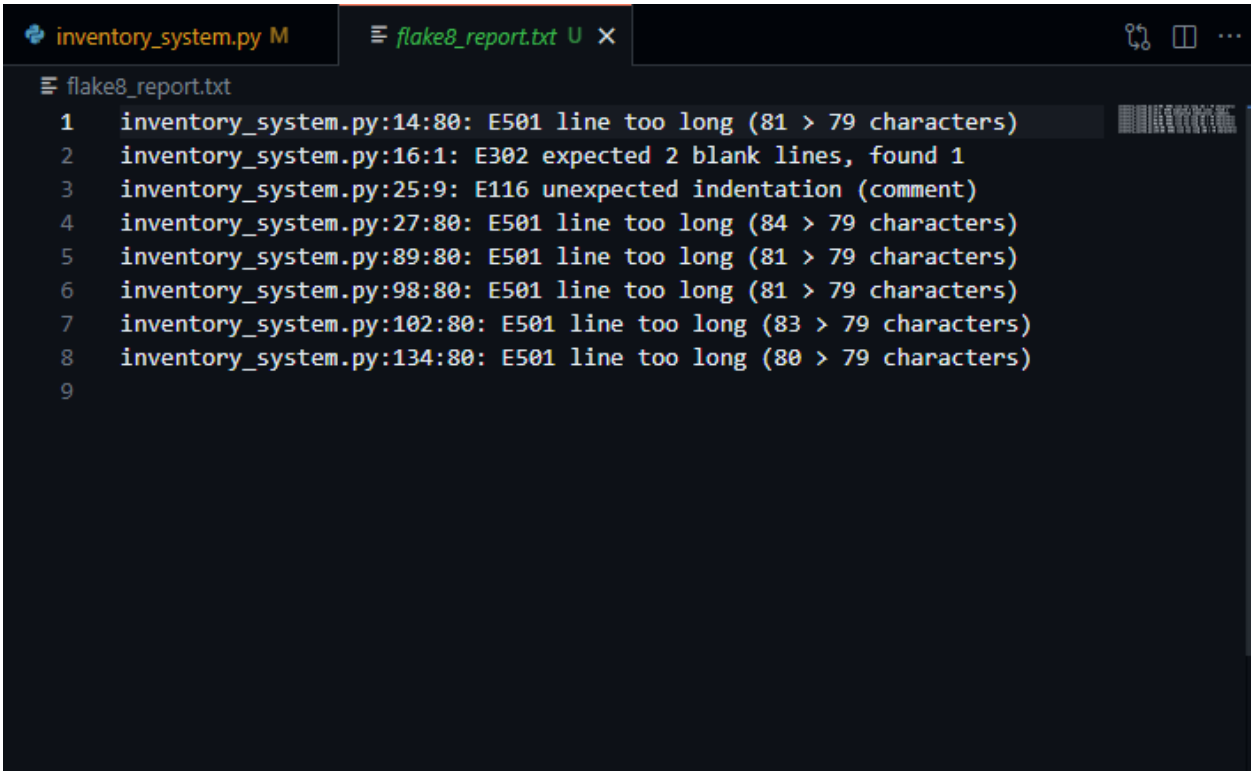
Pylint report :



Bandit report :



Flake8 report :-



Execution terminal :-

```
@PES2UG23CS366-NSLC ➜/workspaces/SE-LAB5-Static-Code_Analysis (main) $  python inventory_system.
py
Warning: Data file 'inventory.json' not found. Starting fresh inventory.
2025-10-27 07:27:52.633676: Added 10 of apple
2025-10-27 07:27:52.633703: Added 2 of banana
Error: Invalid types for item (<class 'int'>) or qty (<class 'str'>).
Removed 3 of 'apple'. New total: 7
Warning: Item 'orange' not in stock. Cannot remove.
Apple stock: 7
Low items: ['banana']

--- Items Report ---
apple -> 7
banana -> 2
---------------------

Inventory saved to inventory.json.
Program finished.
@PES2UG23CS366-NSLC ➜/workspaces/SE-LAB5-Static-Code_Analysis (main) $ pylint inventory_system.p
y > pylint_report.txt
@PES2UG23CS366-NSLC ➜/workspaces/SE-LAB5-Static-Code_Analysis (main) $ bandit -r inventory_syste
m.py > bandit_report.txt
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.12.1
```

```
@PES2UG23CS366-NSLC ➜/workspaces/SE-LAB5-Static-Code_Analysis (main) $ bandit -r inventory_syste
m.py > bandit_report.txt
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.12.1
@PES2UG23CS366-NSLC ➜/workspaces/SE-LAB5-Static-Code_Analysis (main) $ flake8 inventory_system.p
y > flake8_report.txt
@PES2UG23CS366-NSLC ➜/workspaces/SE-LAB5-Static-Code_Analysis (main) $
```

Inventory.json :