Basic Details:

Name: Aahan Sharma
SRN: PES2UG23CS007
Class: 5A
Subject: Software Engineering

Known Issue Table:

Issue	Type	Line(s)	Description
Mutable default arg	Bug	8	Dangerous default value [] as argument. The default list is shared across all function calls.
Insecure eval use	Security (Medium)	59	Use of possibly insecure function - consider using safer `ast.literal_eval`. The eval() function can execute arbitrary code.
Bare except / try...pass	Bug / Security (Low)	19	`Try, Except, Pass` detected. The code uses a bare except , which catches all exceptions (like `SystemExit`) and silently ignores them.

Issue	Type	Line(s)	Description
File handling without with	Bug / Resource Leak	26,32	Consider using `with` for resource-allocating operations. The file may not be closed properly if an error occurs.
File open() without encoding	Bug / Portability	26, 32	Using open without explicitly specifying an encoding. This can cause errors as the default encoding is system-dependent.
Unused import	Code Smell / Style	2	`logging` imported but unused. The logging module is imported but never used.
Use of global statement	Code Smell	27	Using the global statement. Modifying global variables makes code harder to debug.
Missing docstrings	Maintainability	1,8,14,22,25,31,36,41,48	Missing module docstring and Missing function or method docstring for all functions.

Issue	Type	Line(s)	Description
Naming convention	Style	8,14,22,25,31,36,41	Function names (addItem, removeItem, etc.) do not conform to snake_case naming style.
Incorrect blank lines	Style	8,14,22,25,31,36,41,48,61	expected 2 blank lines, found 1. PEP 8 requires two blank lines between top-level function definitions.
Old string formatting	Style	12	Formatting a regular string which could be an f-string.

Fixed Code:

```
"""
This module implements a basic inventory system.
FIX: Added missing module docstring.
"""

import json
# FIX: Removed unused 'logging' import
from datetime import datetime

# FIX: Removed the 'stock_data = {}' global variable.
# State will now be passed as an argument to functions.


# FIX: Expected 2 blank lines before function definition
# FIX: Renamed function to snake_case (was 'addItem')
# FIX: Added 'stock' as the first argument to operate on.
```

```python
def add_item(stock, item="default", qty=0, logs=None):
    # FIX: Added missing function docstring
    """Adds an item to the stock data."""

    # FIX: Dangerous default value [] as argument.
    # Initialize a new list inside the function if one isn't provided.
    if logs is None:
        logs = []

    # FIX: Added basic type validation based on the original file's
    # comment about 'addItem(123, "ten")'
    if not isinstance(item, str) or not isinstance(qty, int):
        print(
            f"Error: Invalid type for item ({type(item)}) or "
            f"qty ({type(qty)}). Skipping."
        )
        return

    if not item:
        return

    # FIX: Operate on the passed 'stock' argument
    stock[item] = stock.get(item, 0) + qty

    # FIX: Converted to an f-string for better readability
    logs.append(f"{datetime.now()}: Added {qty} of {item}")


# FIX: Renamed function to snake_case (was 'removeItem')
# FIX: Added 'stock' as the first argument
def remove_item(stock, item, qty):
    """Removes a specified quantity of an item from stock."""
    try:
        # FIX: (W0603) Operate on the passed 'stock' argument
        stock[item] -= qty
        if stock[item] <= 0:
            del stock[item]
    # FIX: Do not use bare 'except'.
    # This now specifically catches KeyError if the item doesn't exist.
    except KeyError:
        pass  # Silently ignore if the item is not in stock


# FIX: Added 'stock' as the first argument
def get_qty(stock, item):
    """Gets the quantity of a specific item."""
```

```
    # Robustness fix: Use .get() to return 0 instead of
    # raising a KeyError if the item doesn't exist.
    # FIX: Operate on the passed 'stock' argument
    return stock.get(item, 0)


def load_data(file="inventory.json"):
    """Loads the stock data from a JSON file."""
    # FIX: Use 'with' for resource-allocating operations
    # FIX: Specify 'encoding' when opening files
    try:
        with open(file, "r", encoding="utf-8") as f:
            # FIX: Avoid 'global' statement.
            # Return the loaded data instead.
            return json.load(f)
    except FileNotFoundError:
        print(f"Warning: {file} not found. "
              "Starting with empty inventory.")
        return {}  # Return an empty dict if file doesn't exist
    except json.JSONDecodeError:
        print(f"Error: Could not decode {file}. "
              "Starting with empty inventory.")
        return {}


# FIX: Added 'stock' as the first argument
def save_data(stock, file="inventory.json"):
    """Saves the current stock data to a JSON file."""
    # FIX: Use 'with' for resource-allocating operations
    # FIX: Specify 'encoding' when opening files
    with open(file, "w", encoding="utf-8") as f:
        # Added indent=4 for human-readable JSON
        # FIX: Dump the passed 'stock' argument
        json.dump(stock, f, indent=4)


# FIX: Added 'stock' as the first argument
def print_data(stock):
    """Prints a report of all items and their quantities."""
    print("Items Report")
    # FIX: Iterate over the passed 'stock' argument
    for i in stock:
        # FIX: Converted to f-string
        print(f"{i} -> {stock[i]}")
```

```python
# FIX: Added 'stock' as the first argument
def check_low_items(stock, threshold=5):
    """Returns a list of items below a given stock threshold."""
    result = []
    # FIX: Iterate over the passed 'stock' argument
    for i in stock:
        if stock[i] < threshold:
            result.append(i)
    return result


def main():
    """Main function to run the inventory system demo."""

    # FIX: Removed 'global stock_data'.
    # 'stock' is now a local variable, initialized here.
    stock = load_data()

    # FIX: Pass the local 'stock' variable to all functions
    add_item(stock, "apple", 10)
    add_item(stock, "banana", -2)

    # This invalid call is now gracefully handled by the type check in add_item
    add_item(stock, 123, "ten")

    remove_item(stock, "apple", 3)
    remove_item(stock, "orange", 1)  # Will now silently fail (KeyError caught)

    print(f"Apple stock: {get_qty(stock, 'apple')}")
    print(f"Low items: {check_low_items(stock)}")

    save_data(stock)

    # Re-load data into the local 'stock' variable
    stock = load_data()
    print_data(stock)

    # FIX: Removed dangerous and insecure 'eval' call
    # eval("print('eval used')")
    print("--- End of program ---")


# Added a standard __name__ == "__main__" guard
if __name__ == "__main__":
```

```
    main()
```

Reflection

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

The easiest issues to fix were definitely the simple style warnings, like adding blank lines or removing unused imports. They're quick, mechanical changes. The hardest was refactoring the global variable, since that required changing function definitions and modifying how data was passed through the entire program. It was much more involved than any other fix.

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

I didn't really run into any obvious false positives in this lab. All the major issues flagged by the tools, especially the security and bug warnings, seemed like legitimate problems that were good to fix.

How would you integrate static analysis tools into your actual software development workflow? Consider continuous integration (CI) or local development practices.

For a real workflow, I'd integrate these tools in two key places. First, I'd set them up as a local pre-commit hook. This would automatically check my code for issues before I even commit it. Second, I'd add a step to the CI pipeline (like a GitHub Action) to run the static analysis. This would act as a safety net to block any pull requests that introduce new issues.

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

The code quality improved a lot. It's much more readable now with the consistent naming and the new docstrings. It also feels much more robust and safer; getting rid of the bare `except` and the dangerous `eval` function makes it less likely to crash or be exploited. Finally,

removing the global state makes the code feel much cleaner and easier to maintain or add to in the future.