**1) For code Generation and Debugging**

You are an expert Python developer. Generate a clean, production-ready multi-file Python project with the following exact structure and rules:

Project Structure:

diagnostic.py → Define a class Diagnostic with fields: id (int), type (string), value (float).

Types are strictly limited to: "RPM", "EngineLoad", "CoolantTemp".

Add validation so any other type raises ValueError.

car.py → Define a Car class that:

Holds multiple Diagnostic objects.

Provides a method compute\_performance\_score() that:

Uses RPM, EngineLoad, and CoolantTemp if available.

Formula (simple example, you decide reasonable weights):

score = (100 - abs(RPM/100 - 30)) + (50 - EngineLoad/2) + (60 - abs(CoolantTemp - 90)/2).

Returns score as float (max 100, min 0).

garage\_monitor.py → Define a GarageMonitor class that:

Manages multiple Car objects.

Provides check\_alerts() that:

If car’s performance score < 40 → print “Severe Engine Stress”.

If any required diagnostic is missing → print “Sensor Failure Detected”.

main.py → Main script that:

Reads diagnostics from a CSV file with columns: CarID, DiagnosticType, Value.

Creates Car and Diagnostic objects.

Uses GarageMonitor to process all cars.

Prints each car’s performance score and alerts.

Rules for Code Generation:

Follow Python best practices

Each file must import only what it needs

Use main guard in main.py.

Do NOT use external libraries except Python standard library.

Output clean, readable, beginner-friendly code.

Avoid hardcoding sample data. Read CSV dynamically.

Provide minimal but working example CSV inside comments in main.py.

Now generate the full project with four separate files.

1. **For code refactoring**

You are an expert Python developer. Refactor the following messy code into a clean, production-ready multi-file OOP design:

Original Messy Code (for context):

------------------------------------------------

rpm = [6500, 3000]

load = [95, 40]

temp = [120, 85]

for i in range(2):

score = 100 - (rpm[i] / 100 + load[i] \* 0.5 + (temp[i] - 90) \* 2)

print(f"Car {i} score={score}")

if score < 40:

print("ALERT")

------------------------------------------------

Refactor Guidelines:

1. Implement proper OOP design with classes:

- class Diagnostic → holds id, type, value, validates type.

- class Car → holds diagnostics, computes performance score, uses constants instead of magic numbers.

- class GarageMonitor → manages multiple cars, checks alerts.

2. Remove hardcoding:

- Load diagnostics from a CSV file with columns: CarID, DiagnosticType, Value.

- Use Python's `csv` module to parse input.

3. Best practices:

- Encapsulation: use constructors, private attributes if needed.

- Constants: define thresholds (e.g., MAX\_SCORE, ENGINE\_TEMP\_OPTIMAL, STRESS\_THRESHOLD) instead of magic numbers.

- Error handling: handle empty/malformed CSV rows gracefully with exceptions and warnings.

4. Maintain functionality:

- Compute performance score based on diagnostics.

- Print alerts:

- If score < threshold → “Severe Engine Stress”.

- If missing diagnostic → “Sensor Failure Detected”.

5. Code style:

- Follow PEP8 and Python best practices.

- Add docstrings and type hints.

- Ensure modular design with diagnostic.py, car.py, garage\_monitor.py, main.py.

Now refactor the given code into a proper 4-file OOP project that satisfies these requirements.

**3) For unit testing**

Generate a well-structured pytest test file for a Python project with classes: Diagnostic, Car, GarageMonitor. Follow these requirements:

1. Use pytest fixtures to create Car and Garage objects as needed.

2. Test cases:

a) Car with rpm=6500, load=95, temp=120 → score < 40 → assert "Severe Engine Stress" alert.

b) Car missing CoolantTemp → assert "Sensor Failure Detected" alert.

c) Garage with cars having scores 70 and 30 → assert average score = 50.

d) Boundary case: score = 40 → assert no alert is raised.

e) Empty CSV → assert Exception is raised.

3. Include descriptive test function names for each scenario.

4. Use assert statements clearly, and ensure tests are modular and readable.

5. Include necessary imports (pytest, the project modules) and setup/teardown if needed.

6. No external libraries; only use pytest and Python standard library.

**4) Documentation**

Generate a professional README.md file for a Python project called "Car Diagnostics Monitoring System".

The README must include the following sections in well-structured markdown:

1. Introduction

- Explain the purpose of the project.

- Mention that it monitors car diagnostics like RPM, Engine Load, and Coolant Temperature.

- State how it computes performance scores and alerts for engine stress or sensor failures.

2. Explanation of Classes and Functions

- Diagnostic: holds id, type, value with validation.

- Car: stores diagnostics, computes performance score.

- GarageMonitor: manages multiple cars, checks alerts.

- main.py: loads diagnostics from CSV, processes cars, prints results.

3. Sample Input/Output

- Show a sample CSV input format (CarID, DiagnosticType, Value).

- Show example console output with performance score and alerts.

Formatting rules:

- Use markdown headers (##, ###).

- Use bullet points or code blocks where appropriate.

- Keep explanations clear and beginner-friendly.

- Do not include installation instructions or dependencies (standard library only).

**5)Concurency**

Enhance the Car Diagnostics Monitoring System project with concurrency.

Requirements:

1. Use Python's threading module to simulate real-time diagnostic updates for RPM, EngineLoad, and CoolantTemp.

2. Use threading.Lock in GarageMonitor to prevent race conditions when aggregating car scores.

3. Implement both single-threaded and multi-threaded execution modes and compare their performance.

4. Ensure the program’s output format and functionality remain exactly the same as before (same score calculation and alerts).

5. All existing test cases (pytest) must still pass without modification.

6. Follow Python best practices: proper thread management, safe locking, no deadlocks.

Generate the updated code without altering existing project structure (diagnostic.py, car.py, garage\_monitor.py, main.py).