Coding & Development / Python Prompts

Take messy or inefficient Python code and refactor it for readability, performance, and PEP 8 compliance.
Difficulty: Intermediate → Advanced
Model: GPT-4 / Claude / Gemini
Use Case: Code Refactoring, Maintenance, Code Quality
Updated: May 2026
Why This Prompt Exists
Most legacy code is messy, hard to read, and inefficient — but refactoring is time-consuming.

You get:

- unclear variable names (a, b, x, tmp)
- duplicated code blocks (copy-paste programming)
- no functions (spaghetti code)
- PEP 8 violations (inconsistent spacing, line length)
- inefficient algorithms ($O(n^2)$ when $O(n)$ would work)

But refactoring is not rewriting.

It is improving structure without changing behavior.

- Rename variables: meaningful names
- Extract functions: break long blocks into named functions
- Remove duplication: DRY principle
- Fix PEP 8: spacing, line length, naming conventions
- Optimize algorithms: better time/space complexity

Without refactoring, technical debt compounds.

This framework forces AI to refactor code systematically.

The Prompt

```
Assume the role of a senior Python developer who refactors messy code
into clean, maintainable code.

Your task is to refactor Python code.

Generate:

1. DIAGNOSIS
   - List of issues found (naming, duplication, structure,
performance)

2. REFACTORED CODE
   - Clean, PEP 8 compliant
   - Extracted functions where appropriate
   - Meaningful variable names

3. CHANGE SUMMARY
   - What was changed and why

4. BEFORE/AFTER COMPARISON
   - Key improvements highlighted

5. PERFORMANCE NOTES
   - Any performance improvements made
```

```
INPUTS:

Original Code (paste):
[PASTE YOUR CODE HERE]

Refactoring Goals (select all that apply):
[READABILITY / PERFORMANCE / MAINTAINABILITY / TESTABILITY]

Python Version:
[3.8 / 3.9 / 3.10 / 3.11 / 3.12]

Known Issues (if any):
[LIST]

RULES:
- Preserve original behavior (don't change what it does)
- Use meaningful variable names (not single letters)
- Extract repeated logic into functions
- Follow PEP 8 (4 spaces, 79/99 character lines)
- Add docstrings to extracted functions
- Don't over-refactor (balance clarity and simplicity)
```

How To Use It

- Run tests before refactoring to ensure behavior is preserved.
- Refactor in small chunks — test after each change.
- Use version control (git) to track changes and roll back if needed.
- The diagnosis helps you understand what was wrong.
- Run a linter (pylint, flake8) on the refactored code.

Example Input

Original Code:

```
def calc(a,b):
x=0
for i in range(len(a)):
if a[i]>b:
x=x+a[i]
return x
```

Refactoring Goals: READABILITY, MAINTAINABILITY

Python Version: 3.11

Known Issues: Unclear variable names, no docstring, inefficient loop pattern

Why It Works

Most legacy code is hard to maintain.

This framework improves outcomes by forcing:

- issue diagnosis (understanding)
- PEP 8 compliance (standards)
- extracted functions (modularity)
- meaningful names (readability)
- performance improvements (efficiency)

Great refactoring doesn’t change what code does — it changes how easily you can change it later.

Build Better AI Systems

Subscribe for advanced prompt engineering, AI coding tools, Python frameworks, and practical strategies for developers and engineers.

Carefully engineered prompts for people doing real work.

Share this:

- Share on Facebook (Opens in new window) Facebook
- Share on X (Opens in new window) X

See also The Python Function Generator