How to Improve Python Code Quality

There are a few things to consider on our journey for high-quality code. First, this journey is not one of pure objectivity. There are some strong feelings about what high-quality code looks like.

While everyone can hopefully agree on the identifiers mentioned above, the way they get achieved is a subjective road. The most opinionated topics usually come up when you talk about achieving readability, maintenance, and extensibility.

So keep in mind that while this article will try to stay objective throughout, there is a very-opinionated world out there when it comes to code.

So, let's start with the most opinionated topic: code style.

Style Guides

Ah, yes. The age-old question: spaces or tabs?

Regardless of your personal view on how to represent whitespace, it's safe to assume that you at least want consistency in code.

A style guide serves the purpose of defining a consistent way to write your code. Typically this is all cosmetic, meaning it doesn't change the logical outcome of the code. Although, some stylistic choices do avoid common logical mistakes.

Style guides serve to help facilitate the goal of making code easy to read, maintain, and extend.

As far as Python goes, there is a well-accepted standard. It was written, in part, by the author of the Python programming language itself.

PEP 8 provides coding conventions for Python code. It is fairly common for Python code to follow this style guide. It's a great place to start since it's already well-defined.

A sister Python Enhancement Proposal, PEP 257 describes conventions for Python's docstrings, which are strings intended to document modules, classes, functions, and methods. As an added bonus, if docstrings are consistent, there are tools capable of generating documentation directly from the code.

All these guides do is *define* a way to style code. But how do you enforce it? And what about defects and problems in the code, how can you detect those? That's where linters come in.

Linters

WHAT IS A LINTER?

First, let's talk about lint. Those tiny, annoying little defects that somehow get all over your clothes. Clothes look and feel much better without all that lint. Your code is no different. Little mistakes, stylistic inconsistencies, and dangerous logic don't make your code feel great.

But we all make mistakes. You can't expect yourself to always catch them in time. Mistyped variable names, forgetting a closing bracket, incorrect tabbing in Python, calling a function with the wrong number of arguments, the list goes on and on. Linters help to identify those problem areas.

Additionally, most editors and IDEs have the ability to run linters in the background as you type. This results in an environment capable of highlighting, underlining, or otherwise identifying problem areas in the code before you run it. It is like an advanced spell-check for code. It underlines issues in squiggly red lines much as your favorite word processor does.

Linters analyze code to detect various categories of lint. Those categories can be broadly defined as the following:

- Logical Lint
    - Code errors
    - Code with potentially unintended results
    - Dangerous code patterns
- Stylistic Lint
    - Code not conforming to defined conventions

There are also code analysis tools that provide other insights into your code. While maybe not linters by definition, these tools are usually used side-by-side with linters. They too hope to improve the quality of the code.

Finally, there are tools that automatically format code to some specifications. These automated tools ensure that our inferior human minds don't mess up conventions.

WHAT ARE MY LINTER OPTIONS FOR PYTHON?

Before delving into your options, it's important to recognize that some "linters" are just multiple linters packaged nicely together. Some popular examples of those combo-linters are the following:

Flake8: Capable of detecting both logical and stylistic lint. It adds the style and complexity checks of pycodestyle to the logical lint detection of PyFlakes. It combines the following linters:

- PyFlakes
- pycodestyle (formerly pep8)
- Mccabe

Pylama: A code audit tool composed of a large number of linters and other tools for analyzing code. It combines the following:

- pycodestyle (formerly pep8)
- pydocstyle (formerly pep257)
- PyFlakes
- Mccabe
- Pylint
- Radon
- gjslint

Here are some stand-alone linters categorized with brief descriptions:

Linter	Category	Description
Pylint	Logical & Stylistic	Checks for errors, tries to enforce a coding standard, looks for code smells
PyFlakes	Logical	Analyzes programs and detects various errors
pycodestyle	Stylistic	Checks against some of the style conventions in PEP 8
pydocstyle	Stylistic	Checks compliance with Python docstring conventions
Bandit	Logical	Analyzes code to find common security issues

MyPy	Logical	Checks for optionally-enforced static types

And here are some code analysis and formatting tools:

Tool	Category	Description
Mccabe	Analytical	Checks McCabe complexity
Radon	Analytical	Analyzes code for various metrics (lines of code, complexity, and so on)
Black	Formatter	Formats Python code without compromise
Isort	Formatter	Formats imports by sorting alphabetically and separating into sections

COMPARING PYTHON LINTERS

Let's get a better idea of what different linters are capable of catching and what the output looks like. To do this, I ran the same code through a handful of different linters with the default settings.

The code I ran through the linters is below. It contains various logical and stylistic issues:

The comparison below shows the linters I used and their runtime for analyzing the above file. I should point out that these aren't all entirely comparable as they serve different purposes. PyFlakes, for example, does not identify stylistic errors like Pylint does.