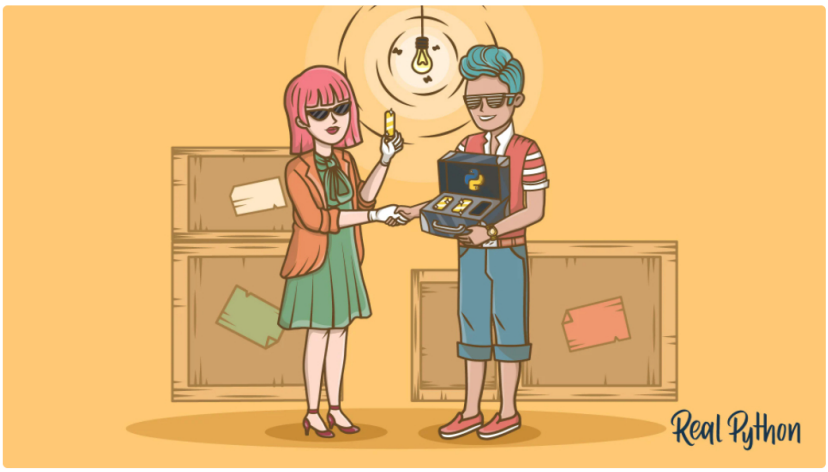

Python Code Quality: Tools & Best Practices

by Alexander VanTol Jul 30, 2018 8 Comments best-practices python tools

Mark as Completed

Tweet Share Email

Table of Contents

In this article, we'll identify high-quality Python code and show you how to improve the quality of your own code.

We'll analyze and compare tools you can use to take your code to the next level. Whether you've been using Python for a while, or just beginning, you can benefit from the practices and tools talked about here.

What is Code Quality?

Of course you want quality code, who wouldn't? But to improve code quality, we have to define what it is.

A quick Google search yields many results defining code quality. As it turns out, the term can mean many different things to people.

One way of trying to define code quality is to look at one end of the spectrum: high-quality

Table of Contents

- What is Code Quality?
- Why Does Code Quality Matter?
- How to Improve Python Code Quality
- When Can I Check My Code Quality?
- Conclusion

Mark as Completed

Tweet Share Email

code. Hopefully, you can agree on the following high-quality code identifiers:

- It does what it is supposed to do.
- It does not contain defects or problems.
- It is easy to read, maintain, and extend.

These three identifiers, while simplistic, seem to be generally agreed upon. In an effort to expand these ideas further, let's delve into why each one matters in the realm of software.

Why Does Code Quality Matter?

To determine why high-quality code is important, let's revisit those identifiers. We'll see what happens when code doesn't meet them.

It does **not** do what it is supposed to do

Meeting requirements is the basis of any product, software or otherwise. We make software to do something. If in the end, it doesn't do it… well it's definitely not high quality. If it doesn't meet basic requirements, it's hard to even call it low quality.

It **does** contain defects and problems

If something you're using has issues or causes you problems, you probably wouldn't call it high-quality. In fact, if it's bad enough, you may stop using it altogether.

For the sake of not using software as an example, let's say your vacuum works great on regular carpet. It cleans up all the dust and cat hair. One fateful night the cat knocks over a plant, spilling dirt everywhere. When you try to use the vacuum to clean the pile of dirt, it breaks, spewing the dirt everywhere.

While the vacuum worked under some circumstances, it didn't efficiently handle the occasional extra load. Thus, you wouldn't call it a high-quality vacuum cleaner.

That is a problem we want to avoid in our code. If things break on edge cases and defects cause unwanted behavior, we don't have a high-quality product.

It is **difficult** to read, maintain, or extend

Imagine this: a customer requests a new feature. The person who wrote the original code is gone. The person who has replaced them now has to make sense of the code that's already there. That person is you.

If the code is easy to comprehend, you'll be able to analyze the problem and come up with a solution much quicker. If the code is complex and convoluted, you'll probably take longer and possibly make some wrong assumptions.

It's also nice if it's easy to add the new feature without disrupting previous features. If the code is *not* easy to extend, your new feature could break other things.

No one *wants* to be in the position where they have to read, maintain, or extend low-quality code. It means more headaches and more work for everyone.

It's bad enough that you have to deal with low-quality code, but don't put someone else in the same situation. You can improve the quality of code that you write.

If you work with a team of developers, you can start putting into place methods to ensure better overall code quality. Assuming that you have their support, of course. You may have to win some people over (feel free to send them this article 😀).

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 of 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 like your favorite word processor does.

Linters analyze code to detect various categories of lint. Those categories can be broadly

defined as the following:

1. Logical Lint
    - Code errors
    - Code with potentially unintended results
    - Dangerous code patterns
2. 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 specification. 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:

Python Code With Lint Show/Hide

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.

Linter	Command	Time
Pylint	pylint code_with_lint.py	1.16s
PyFlakes	pyflakes code_with_lint.py	0.15s
pycodestyle	pycodestyle code_with_lint.py	0.14s
pydocstyle	pydocstyle code_with_lint.py	0.21s

For the outputs of each, see the sections below.

Pylint

Pylint is one of the oldest linters (circa 2006) and is still well-maintained. Some might call this software battle-hardened. It's been around long enough that contributors have fixed most major bugs and the core features are well-developed.

The common complaints against Pylint are that it is slow, too verbose by default, and takes a lot of configuration to get it working the way you want. Slowness aside, the other complaints are somewhat of a double-edged sword. Verbosity can be because of thoroughness. Lots of configuration can mean lots of adaptability to your preferences.

Without further ado, the output after running Pylint against the lint-filled code from above:

Text

```
No config file found, using default configuration
************ Module code_with_lint
W: 23, 0: Unnecessary semicolon (unnecessary-semicolon)
C: 27, 0: Unnecessary parens after 'return' keyword (superfluous-parens)
C: 27, 0: No space allowed after bracket
               return( 'an unlucky number!')
                     ^ (bad-whitespace)
C: 29, 0: Unnecessary parens after 'return' keyword (superfluous-parens)
C: 33, 0: Exactly one space required after comma
    def __init__(self, some_arg,  some_other_arg, verbose = False):
                               ^ (bad-whitespace)
C: 33, 0: No space allowed around keyword argument assignment
    def __init__(self, some_arg,  some_other_arg, verbose = False):
                                                           ^ (bad-whitespace)
C: 34, 0: Exactly one space required around assignment
        self.some_other_arg  =  some_other_arg
                             ^ (bad-whitespace)
C: 35, 0: Exactly one space required around assignment
```

```
        self.some_arg        =  some_arg
                             ^ (bad-whitespace)
C: 40, 0: Final newline missing (missing-final-newline)
W:  6, 0: Redefining built-in 'pow' (redefined-builtin)
W:  6, 0: Wildcard import math (wildcard-import)
C: 11, 0: Constant name "some_global_var" doesn't conform to UPPER_CASE naming style
C: 13, 0: Argument name "x" doesn't conform to snake_case naming style (invalid-name)
C: 13, 0: Argument name "y" doesn't conform to snake_case naming style (invalid-name)
C: 13, 0: Missing function docstring (missing-docstring)
W: 14, 4: Redefining name 'some_global_var' from outer scope (line 11) (redefined-outer
W: 17, 4: Unreachable code (unreachable)
W: 14, 4: Unused variable 'some_global_var' (unused-variable)
...
R: 24,12: Unnecessary "else" after "return" (no-else-return)
R: 20, 0: Either all return statements in a function should return an expression, or n
C: 31, 0: Missing class docstring (missing-docstring)
W: 37, 8: Redefining name 'time' from outer scope (line 9) (redefined-outer-name)
E: 37,15: Using variable 'time' before assignment (used-before-assignment)
W: 33,50: Unused argument 'verbose' (unused-argument)
W: 36, 8: Unused variable 'list_comprehension' (unused-variable)
W: 39, 8: Unused variable 'date_and_time' (unused-variable)
R: 31, 0: Too few public methods (0/2) (too-few-public-methods)
W:  5, 0: Unused import io (unused-import)
W:  6, 0: Unused import acos from wildcard import (unused-wildcard-import)
...
W:  9, 0: Unused time imported from time (unused-import)
```

Note that I've condensed this with ellipses for similar lines. It's quite a bit to take in, but there *is* a lot of lint in this code.

Note that Pylint prefixes each of the problem areas with a `R`, `C`, `W`, `E`, or `F`, meaning:

- [R]efactor for a "good practice" metric violation
- [C]onvention for coding standard violation
- [W]arning for stylistic problems, or minor programming issues
- [E]rror for important programming issues (i.e. most probably bug)
- [F]atal for errors which prevented further processing

The above list is directly from Pylint's user guide.

PyFlakes

Pyflakes "makes a simple promise: it will never complain about style, and it will try very, very hard to never emit false positives". This means that Pyflakes won't tell you about missing docstrings or argument names not conforming to a naming style. It focuses on logical code issues and potential errors.

The benefit here is speed. PyFlakes runs in a fraction of the time Pylint takes.

Output after running against lint-filled code from above:

Text

```
code_with_lint.py:5: 'io' imported but unused
code_with_lint.py:6: 'from math import *' used; unable to detect undefined names
code_with_lint.py:14: local variable 'some_global_var' is assigned to but never used
code_with_lint.py:36: 'pi' may be undefined, or defined from star imports: math
code_with_lint.py:36: local variable 'list_comprehension' is assigned to but never use
code_with_lint.py:37: local variable 'time' (defined in enclosing scope on line 9) re
code_with_lint.py:37: local variable 'time' is assigned to but never used
code_with_lint.py:39: local variable 'date_and_time' is assigned to but never used
```

The downside here is that parsing this output may be a bit more difficult. The various issues and errors are not labeled or organized by type. Depending on how you use this, that may not be a problem at all.

pycodestyle (formerly pep8)

Used to check *some* style conventions from PEP8. Naming conventions are not checked and neither are docstrings. The errors and warnings it does catch are categorized in this table.

Output after running against lint-filled code from above:

Text

```
code_with_lint.py:13:1: E302 expected 2 blank lines, found 1
code_with_lint.py:15:15: E225 missing whitespace around operator
code_with_lint.py:20:1: E302 expected 2 blank lines, found 1
code_with_lint.py:21:10: E711 comparison to None should be 'if cond is not None:'
code_with_lint.py:23:25: E703 statement ends with a semicolon
code_with_lint.py:27:24: E201 whitespace after '('
code_with_lint.py:31:1: E302 expected 2 blank lines, found 1
code_with_lint.py:33:58: E251 unexpected spaces around keyword / parameter equals
code_with_lint.py:33:60: E251 unexpected spaces around keyword / parameter equals
code_with_lint.py:34:28: E221 multiple spaces before operator
code_with_lint.py:34:31: E222 multiple spaces after operator
code_with_lint.py:35:22: E221 multiple spaces before operator
code_with_lint.py:35:31: E222 multiple spaces after operator
code_with_lint.py:36:80: E501 line too long (83 > 79 characters)
code_with_lint.py:40:15: W292 no newline at end of file
```

The nice thing about this output is that the lint is labeled by category. You can choose to ignore certain errors if you don't care to adhere to a specific convention as well.

pydocstyle (formerly pep257)

Very similar to pycodestyle, except instead of checking against PEP8 code style conventions, it checks docstrings against conventions from PEP257.

Output after running against lint-filled code from above:

Text

```
code_with_lint.py:1 at module level:
        D200: One-line docstring should fit on one line with quotes (found 3)
code_with_lint.py:1 at module level:
        D400: First line should end with a period (not '!')
code_with_lint.py:13 in public function `multiply`:
        D103: Missing docstring in public function
code_with_lint.py:20 in public function `is_sum_lucky`:
        D103: Missing docstring in public function
code_with_lint.py:31 in public class `SomeClass`:
        D101: Missing docstring in public class
code_with_lint.py:33 in public method `__init__`:
        D107: Missing docstring in __init__
```

Again, like pycodestyle, pydocstyle labels and categorizes the various errors it finds. And the list doesn't conflict with anything from pycodestyle since all the errors are prefixed with a `D` for docstring. A list of those errors can be found here.

Code Without Lint

You can adjust the previously lint-filled code based on the linter's output and you'll end up with something like the following:

Python Code Without Lint Show/Hide

That code is lint-free according to the linters above. While the logic itself is mostly nonsensical, you can see that at a minimum, consistency is enforced.

In the above case, we ran linters after writing all the code. However, that's not the only way to go about checking code quality.

When Can I Check My Code Quality?

You can check your code's quality:

- As you write it
- When it's checked in

- When you're running your tests

It's useful to have linters run against your code frequently. If automation and consistency aren't there, it's easy for a large team or project to lose sight of the goal and start creating lower quality code. It happens slowly, of course. Some poorly written logic or maybe some code with formatting that doesn't match the neighboring code. Over time, all that lint piles up. Eventually, you can get stuck with something that's buggy, hard to read, hard to fix, and a pain to maintain.

To avoid that, check code quality often!

As You Write

You can use linters as you write code, but configuring your environment to do so may take some extra work. It's generally a matter of finding the plugin for your IDE or editor of choice. In fact, most IDEs will already have linters built in.

Here's some general info on Python linting for various editors:

- Sublime Text
- VS Code
- Atom
- Vim
- Emacs

Before You Check In Code

If you're using Git, Git hooks can be set up to run your linters before committing. Other version control systems have similar methods to run scripts before or after some action in the system. You can use these methods to block any new code that doesn't meet quality standards.

While this may seem drastic, forcing every bit of code through a screening for lint is an important step towards ensuring continued quality. Automating that screening at the front gate to your code may be the best way to avoid lint-filled code.

When Running Tests

You can also place linters directly into whatever system you may use for continuous integration. The linters can be set up to fail the build if the code doesn't meet quality standards.

Again, this may seem like a drastic step, especially if there are already lots of linter errors in the existing code. To combat this, some continuous integration systems will allow you the option of only failing the build if the new code increases the number of linter errors that were already present. That way you can start improving quality without doing a whole rewrite of your existing code base.

Conclusion

High-quality code does what it's supposed to do without breaking. It is easy to read, maintain, and extend. It functions without problems or defects and is written so that it's easy for the next person to work with.

Hopefully it goes without saying that you should strive to have such high-quality code. Luckily, there are methods and tools to help improve code quality.

Style guides will bring consistency to your code. PEP8 is a great starting point for Python. Linters will help you identify problem areas and inconsistencies. You can use linters throughout the development process, even automating them to flag lint-filled code before it gets too far.

Having linters complain about style also avoids the need for style discussions during code reviews. Some people may find it easier to receive candid feedback from these tools instead of a team member. Additionally, some team members may not want to "nitpick" style during

code reviews. Linters avoid the politics, save time, and complain about any inconsistency.

In addition, all the linters mentioned in this article have various command line options and configurations that let you tailor the tool to your liking. You can be as strict or as loose as you want, which is an important thing to realize.

Improving code quality is a process. You can take steps towards improving it without completely disallowing all nonconformant code. Awareness is a great first step. It just takes a person, like you, to first realize how important high-quality code is.

About **Alexander VanTol**

Alexander is an avid Pythonista who spends his time on various creative projects involving programming, music, and creative writing.

» More about Alexander

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Adriana

 Dan

 Jim

 Joanna

Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Tweet Share Email

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: best-practices python tools

Keep Learning

Related Tutorial Categories: best-practices python tools