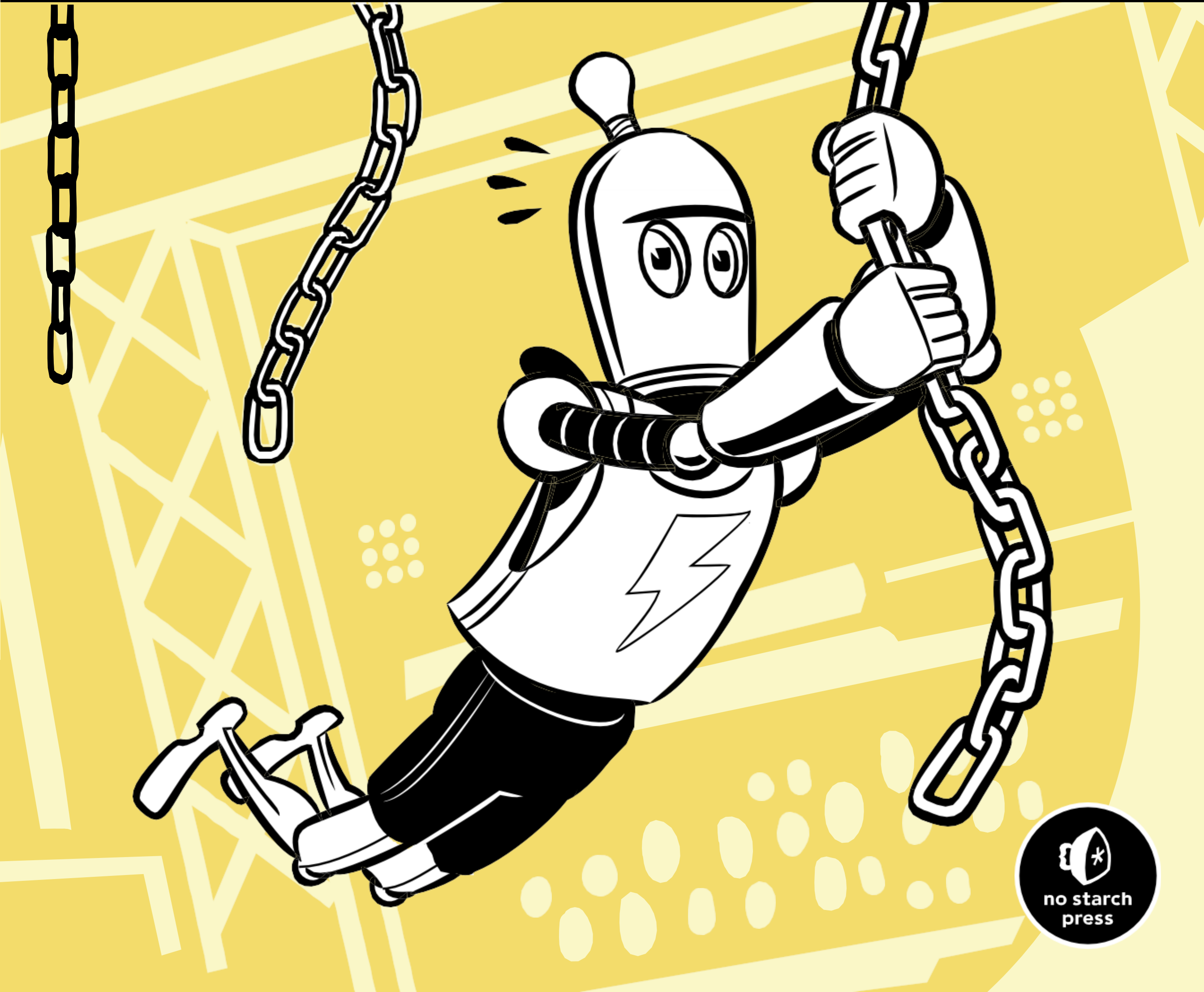
BEYOND THE BASIC STUFF WITH PYTHON

BEST PRACTICES FOR WRITING CLEAN CODE

AL SWEIGART



no starch press

ISBN-13: 978-1-59327-966-0 (print)
ISBN-13: 978-1-59327-967-7 (ebook)

Publisher: William Pollock
Executive Editor: Barbara Yien
Production Editor: Maureen Forys, Happenstance Type-O-Rama
Developmental Editor: Frances Saux
Cover Design: Octopod Studios
Interior Design: Octopod Studios
Cover Illustration: Josh Ellingson
Technical Reviewer: Kenneth Love
Copyeditor: Anne Marie Walker
Compositor: Happenstance Type-O-Rama
Proofreader: Rachel Monaghan
Indexer: Valerie Perry

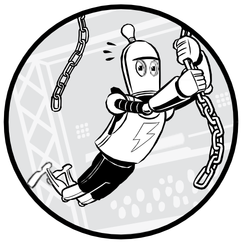
For information on book distributors or translations, please contact No Starch Press, Inc. directly:
No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1-415-863-9900; info@nostarch.com
www.nostarch.com

Library of Congress Cataloging-in-Publication Data

Library of Congress Cataloging-in-Publication Data Names:

Sweigart, Al, author.
Title: Beyond the basic stuff with python : best practices for writing clean code / Al Sweigart.
Description: San Francisco, CA : No Starch Press, Inc., [2021] | Includes index.
Identifiers: LCCN 2020034287 (print) | LCCN 2020034288 (ebook) | ISBN
9781593279660 (paperback) | ISBN 9781593279677 (ebook)
Subjects: LCSH: Python (Computer program language) | Computer programming.
Classification: LCC QA76.73.P98 S943 2021 (print) | LCC QA76.73.P98 (ebook) | DDC
005.13/3—dc23
LC record available at https://lccn.loc.gov/2020034287
LC ebook record available at https://lccn.loc.gov/2020034288

3
CODE FORMATTING WITH BLACK



Code formatting is applying a set of rules to source code to give it a certain appearance. Although unimportant to the computer parsing your program, code formatting is vital for readability, which is necessary for maintaining your code. If your code is difficult for humans (whether it's you or a co-worker) to understand, it will be hard to fix bugs or add new features. Formatting code isn’t a mere cosmetic issue. Python’s readability is a critical reason for the language’s popularity.

This chapter introduces you to Black, a code formatting tool that can automatically format your source code into a consistent, readable style without changing your program’s behavior. Black is useful, because it’s tedious to manually format your code in a text editor or IDE. You’ll first learn about the rationalization for the code style choices Black makes. Then you’ll learn how to install, use, and customize the tool.

How to Lose Friends and Alienate Co-Workers

We can write code in many ways that result in identical behavior. For example, we can write a list with a single space after each comma and use one kind of quote character consistently:

```
spam = ['dog', 'cat', 'moose']
```

But even if we wrote the list with a varying number of spaces and quote styles, we'd still have syntactically valid Python code:

```
spam= [ 'dog' ,'cat',"moose"]
```

Programmers who prefer the former approach might like the visual separation that the spaces add and the uniformity of the quote characters. But programmers sometimes choose the latter, because they don't want to worry about details that have no impact on whether the program works correctly.

Beginners often ignore code formatting because they're focused on programming concepts and language syntax. But it's valuable for beginners to establish good code formatting habits. Programming is difficult enough, and writing understandable code for others (or for yourself in the future) can minimize this problem.

Although you might start out coding on your own, programming is often a collaborative activity. If several programmers working on the same source code files write in their own style, the code can become an inconsistent mess, even if it runs without error. Or worse, the programmers will constantly be reformatting each other's code to their own style, wasting time and causing arguments. Deciding whether to, say, put one or zero spaces after a comma is a matter of personal preference. These style choices can be much like deciding which side of the road to drive on; it doesn't matter whether people drive on the right side of the road or the left side, as long as everyone consistently drives on the same side.

Style Guides and PEP 8

An easy way to write readable code is to follow a *style guide*, a document that outlines a set of formatting rules a software project should follow. The *Python Enhancement Proposal 8* (*PEP 8*) is one such style guide written by the Python core development team. But some software companies have established their own style guides as well.

You can find PEP 8 online at *https://www.python.org/dev/peps/pep-0008/*. Many Python programmers view PEP 8 as an authoritative set of rules, although the PEP 8 creators argue otherwise. The "A Foolish Consistency Is the Hobgoblin of Little Minds" section of the guide reminds the reader that maintaining consistency and readability within a project, rather than adhering to any individual formatting rule, is the prime reason for enforcing style guides.

PEP 8 even includes the following advice: "Know when to be inconsistent—sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment." Whether you follow all of it, some of it, or none of it, it's worthwhile to read the PEP 8 document.

Because we're using the Black code formatter, our code will follow Black's style guide, which is adapted from PEP 8's style guide. You should learn these code formatting guidelines, because you might not always have Black conveniently at hand. The Python code guidelines you learn in this chapter also generally apply to other languages, which might not have automatic formatters available.

I don't like everything about how Black formats code, but I take that as the sign of a good compromise. Black uses formatting rules that programmers can live with, letting us spend less time arguing and more time programming.

Horizontal Spacing

Empty space is just as important for readability as the code you write. These spaces help separate distinct parts of code from each other, making them easier to identify. This section explains *horizontal spacing*—that is, the placement of blank space within a single line of code, including the indentation at the front of the line.

Use Space Characters for Indentation

Indentation is the whitespace at the beginning of a code line. You can use one of two whitespace characters, a space or a tab, to indent your code. Although either character works, the best practice is to use spaces instead of tabs for indentation.

The reason is that these two characters behave differently. A space character is always rendered on the screen as a string value with a single space, like this ' '. But a tab character, which is rendered as a string value containing an escape character, or '\t', is more ambiguous. Tabs often, but not always, render as a variable amount of spacing so the following text begins at the next tab stop. The tab stop are positioned every eight spaces across the width of a text file. You can see this variation in the following interactive shell example, which first separates words with space characters and then with tab characters:

```
>>> print('Hello there, friend!\nHow are you?')
Hello there, friend!
How are you?
>>> print('Hello\tthere,\tfriend!\nHow\tare\tyou?')
Hello   there,  friend!
How     are     you?
```

Because tabs represent a varying width of whitespace, you should avoid using them in your source code. Most code editors and IDEs will automatically insert four or eight space characters when you press the TAB key instead of one tab character.

You also can't use tabs *and* spaces for indentation in the same block of code. Using both for indentation was such a source of tedious bugs in earlier Python programs that Python 3 won't even run code indented like this; it raises a `TabError: inconsistent use of tabs and spaces in indentation` exception instead. Black automatically converts any tab characters you use for indentation into four space characters.

As for the length of each indentation level, the common practice in Python code is four spaces per level of indentation. The space characters in the following example have been marked with periods to make them visible:

```
def getCatAmount():
....numCats = input('How many cats do you have?')
....if int(numCats) < 6:
........print('You should get more cats.')
```

The four-space standard has practical benefits compared to the alternatives; using eight spaces per level of indentation results in code that quickly runs up against line length limits, whereas using two space characters per level of indentation can make the differences in indentation hard to see. Programmers often don't consider other amounts, such as three or six spaces, because they, and binary computing in general, have a bias for numbers that are powers of two: 2, 4, 8, 16, and so on.

Spacing Within a Line

Horizontal spacing has more to it than just the indentation. Spaces are important for making different parts of a code line appear visually distinct. If you never use space characters, your line can end up dense and hard to parse. The following subsections provide some spacing rules to follow.

Put a Single Space Between Operators and Identifiers

If you don't leave spaces between operators and identifiers, your code will appear to run together. For example, this line has spaces separating operators and variables:

```
YES: blanks = blanks[:i] + secretWord[i] + blanks[i + 1 :]
```

This line removes all spacing:

```
NO: blanks=blanks[:i]+secretWord[i]+blanks[i+1:]
```

In both cases, the code uses the `+` operator to add three values, but without spacing, the `+` in `blanks[i+1:]` can appear to be adding a fourth value. The spaces make it more obvious that this `+` is part of a slice for the value in `blanks`.

Put No Spaces Before Separators and a Single Space After Separators

We separate the items lists and dictionaries, as well as the parameters in function def statements, using comma characters. You should place no spaces before these commas and a single space after them, as in this example:

```
YES: def spam(eggs, bacon, ham):
YES:         weights = [42.0, 3.1415, 2.718]
```

Otherwise, you'll end up with "bunched up" code that is harder to read:

```
NO: def spam(eggs,bacon,ham):
NO:          weights = [42.0,3.1415,2.718]
```

Don't add spaces before the separator, because that unnecessarily draws the eye to the separator character:

```
NO: def spam(eggs , bacon , ham):
NO:          weights = [42.0 , 3.1415 , 2.718]
```

Black automatically inserts a space after commas and removes spaces before them.

Don't Put Spaces Before or After Periods

Python allows you to insert spaces before and after the periods marking the beginning of a Python attribute, but you should avoid doing so. By not placing spaces there, you emphasize the connection between the object and its attribute, as in this example:

```
YES: 'Hello, world'.upper()
```

If you put spaces before or after the period, the object and attribute look like they're unrelated to each other:

```
NO: 'Hello, world' . upper()
```

Black automatically removes spaces surrounding periods.

Don't Put Spaces After a Function, Method, or Container Name

We can readily identify function and method names because they're followed by a set of parentheses, so don't put a space between the name and the opening parenthesis. We would normally write a function call like this:

```
YES: print('Hello, world!')
```

But adding a space makes this singular function call look like it's two separate things:

```
NO: print ('Hello, world!')
```

Black removes any spaces between a function or method name and its opening parenthesis.

Similarly, don't put spaces before the opening square bracket for an index, slice, or key. We normally access items inside a container type (such as a list, dictionary, or tuple) without adding spaces between the variable name and opening square bracket, like this:

```
YES: spam[2]
YES: spam[0:3]
YES: pet['name']
```

Adding a space once again makes the code look like two separate things:

```
NO: spam [2]
NO:  spam        [0:3]
NO: pet ['name']
```

Black removes any spaces between the variable name and opening square bracket.

Don't Put Spaces After Opening Brackets or Before Closing Brackets

There should be no spaces separating parentheses, square brackets, or braces and their contents. For example, the parameters in a `def` statement or values in a list should start and end immediately after and before their parentheses and square brackets:

```
YES: def spam(eggs, bacon, ham):
YES:          weights = [42.0, 3.1415, 2.718]
```

You should not put a space after an opening or before a closing parentheses or square brackets:

```
NO: def spam( eggs, bacon, ham ):
NO:          weights = [ 42.0, 3.1415, 2.718 ]
```

Adding these spaces doesn't improve the code's readability, so it's unnecessary. Black removes these spaces if they exist in your code.

Put Two Spaces Before End-of-Line Comments

If you add comments to the end of a code line, put two spaces after the end of the code and before the `#` character that begins the comment:

```
YES: print('Hello, world!') # Display a greeting.
```

The two spaces make it easier to distinguish the code from the comment. A single space, or worse, no space, makes it more difficult to notice this separation:

```
NO: print('Hello, world!') # Display a greeting.
NO: print('Hello, world!')# Display a greeting.
```

Black puts two spaces between the end of the code and the start of the comment.

In general, I advise against putting comments at the end of a code line, because they can make the line too lengthy to read onscreen.

Vertical Spacing

Vertical spacing is the placement of blank lines between lines of code. Just as a new paragraph in a book keeps sentences from forming a wall of text, vertical spacing can group certain lines of code together and separate those groups from one another.

PEP 8 has several guidelines for inserting blank lines in code: it states that you should separate functions with two blank lines, classes with two blank lines, and methods within a class with one blank line. Black automatically follows these rules by inserting or removing blank lines in your code, turning this code:

```
NO: class ExampleClass:
        def exampleMethod1():
            pass
        def exampleMethod2():
            pass
    def exampleFunction():
        pass
```

. . . into this code:

```
YES: class ExampleClass:
        def exampleMethod1():
            pass

        def exampleMethod2():
            pass


    def exampleFunction():
        pass
```

A Vertical Spacing Example

What Black *can't* do is decide where blank lines within your functions, methods, or global scope should go. Which of those lines to group together is a subjective decision that is up to the programmer.

For example, let's look at the `EmailValidator` class in *validators.py* in the Django web app framework. It's not necessary for you to understand how this code works. But pay attention to how blank lines separate the_call_() method's code into four groups:

```
--snip--
    def_call_(self, value):
    1 if not value or '@' not in value:
            raise ValidationError(self.message, code=self.code)

    2 user_part, domain_part = value.rsplit('@', 1)

    3 if not self.user_regex.match(user_part):
            raise ValidationError(self.message, code=self.code)

    4 if (domain_part not in self.domain_whitelist and
                not self.validate_domain_part(domain_part)): # Try
            for possible IDN domain-part
            try:
                domain_part = punycode(domain_part)
            except UnicodeError:
                pass
            else:
                if self.validate_domain_part(domain_part): return
            raise ValidationError(self.message, code=self.code)
--snip--
```

Even though there are no comments to describe this part of the code, the blank lines indicate that the groups are conceptually distinct from each other. The first group 1 checks for an @symbol in the valueparameter. This task is different from that of the second group 2, which splits the email address string in valueinto two new variables, user_partand domain_part. The third 3 and fourth 4 groups use these variables to validate the user and domain parts of the email address, respectively.

Although the fourth group has 11 lines, far more than the other groups, they're all related to the same task of validating the domain of the email address. If you felt that this task was really composed of multiple subtasks, you could insert blank lines to separate them.

The programmer for this part of Django decided that the domain validation lines should all belong to one group, but other programmers might disagree. Because it's subjective, Black won't modify the vertical spacing within functions or methods.

Vertical Spacing Best Practices

One of Python's lesser-known features is that you can use a semicolon to separate multiple statements on a single line. This means that the following two lines:

```
print('What is your name?')
name = input()
```

. . . can be written on the same line if separated by a semicolon:

```
print('What is your name?'); name = input()
```

As you do when using commas, you should put no space before the semicolon and one space after it.

For statements that end with a colon, such as `if`, `while`, `for`, `def`, or `class` statements, a single-line block, like the call to `print()` in this example:

```
if name == 'Alice': print('Hello,
    Alice!')
```

. . . can be written on the same line as its `if` statement:

```
if name == 'Alice': print('Hello, Alice!')
```

But just because Python allows you to include multiple statements on the same line doesn't make it a good practice. It results in overly wide lines of code and too much content to read on a single line. Black splits these statements into separate lines.

Similarly, you can import multiple modules with a single `import`statement:

```
import math, os, sys
```

Even so, PEP 8 recommends that you split this statement into one `import` statement per module:

```
import math
import os
import sys
```

If you write separate lines for imports, you'll have an easier time spotting any additions or removals of imported modules when you're comparing changes in a version control system's diff tool. (Version control systems, such as Git, are covered in Chapter 12.)

PEP 8 also recommends grouping `import`statements into the following three groups in this order:

1. Modules in the Python standard library, like `math`, `os`, and `sys`
2. Third-party modules, like Selenium, Requests, or Django
3. Local modules that are a part of the program

These guidelines are optional, and Black won't change the formatting of your code's `import`statements.

Black: The Uncompromising Code Formatter

Black automatically formats the code inside your *.py* files. Although you should understand the formatting rules covered in this chapter, Black can do all the actual styling for you. If you're working on a coding project with others, you can instantly settle many arguments on how to format code by just letting Black decide.

You can't change many of the rules that Black follows, which is why it's described as "the uncompromising code formatter." Indeed, the tool's name comes from Henry Ford's quote about the automobile colors choices he offered his customers: "You can have any color you want, as long as it's black."

I've just described the exact styles that Black uses; you can find Black's full style guide at *https://black.readthedocs.io/en/stable/the_black_code_style.html*.

Installing Black

Install Black using the pip tool that comes with Python. In Windows, do this by opening a Command Prompt window and entering the following:

```
C:\Users\Al\>python -m pip install --user black
```

On macOS and Linux, open a Terminal window and enter python3 rather than python (you should do this for all the instructions in this book that use python):

```
Als-MacBook-Pro:~ al$ python3 -m pip install --user black
```

The -m option tells Python to run the pip module as an application, which some Python modules are set up to do. Test that the installation was successful by running python -m black. You should see the message No paths given. Nothing to do. rather than No module named black.

Running Black from the Command Line

You can run Black for any Python file from the Command Prompt or Terminal window. In addition, your IDE or code editor can run Black in the background. You'll find instructions for getting Black to work with Jupyter Notebook, Visual Studio Code, PyCharm, and other editors on Black's home page at *https://github.com/psf/black/*.

Let's say that you want to format a file called *yourScript.py* automatically. From the command line in Windows, run the following (on macOS and Linux, use the python3 command instead of python):

```
C:\Users\Al>python -m black yourScript.py
```

After you run this command, the content of *yourScript.py* will be formatted according to Black's style guide.

Your PATH environment variable might already be set up to run Black directly, in which case you can format *yourScript.py* by simply entering the following:

```
C:\Users\Al>black yourScript.py
```

If you want to run Black over every *.py* file in a folder, specify a single folder instead of an individual file. The following Windows example formats every file in the *C:\yourPythonFiles* folder, including its subfolders:

```
C:\Users\Al>python -m black C:\yourPythonFiles
```

Specifying the folder is useful if your project contains several Python files and you don't want to enter a command for each one.

Although Black is fairly strict about how it formats code, the next three subsections describe a few options that you *can* change. To see the full range of options that Black offers, run `python -m black --help`.

Adjusting Black's Line Length Setting

The standard line of Python code is 80 characters long. The history of the 80 character line dates back to the era of punch card computing in the 1920s when IBM introduced punch cards that had 80 columns and 12 rows. The 80 column standard remained for the printers, monitors, and command line windows developed over the next several decades.

But in the 21st century, high-resolution screens can display text that is more than 80 characters wide. A longer line length can keep you from having to scroll vertically to view a file. A shorter line length can keep too much code from crowding on a single line and allow you to compare two source code files side by side without having to scroll horizontally.

Black uses a default of 88 characters per line for the rather arbitrary reason that it is 10 percent more than the standard 80 character line. My preference is to use 120 characters. To tell Black to format your code with, for example, a 120-character line length limit, use the `-l 120` (that's the lowercase letter *L,* not the number 1) command line option. On Windows, the command looks like this:

```
C:\Users\Al>python -m black -l 120 yourScript.py
```

No matter what line length limit you choose for your project, all *.py* files in a project should use the same limit.

Disabling Black's Double-Quoted Strings Setting

Black automatically changes any string literals in your code from using single quotes to double quotes unless the string contains double quote characters, in which case it uses single quotes. For example, let's say *yourScript.py* contains the following:

```
a = 'Hello' b =
"Hello"
c = 'Al\'s cat, Zophie.' d =
'Zophie said, "Meow"'
e = "Zophie said, \"Meow\"" f =
'''Hello'''
```

Running Black on *yourScript.py* would format it like this:

```
1 a = "Hello" b
  = "Hello"
  c = "Al's cat, Zophie."
2 d = 'Zophie said, "Meow"' e =
  'Zophie said, "Meow"'
3 f = """Hello"""
```

Black's preference for double quotes makes your Python code look similar to code written in other programming languages, which often use double quotes for string literals. Notice that the strings for variables `a`, `b`, and `c` use double quotes. The string for variable `d`retains its original single quotes to avoid escaping any double quotes within the string 2. Note that Black also uses double quotes for Python's triple-quoted, multiline strings 3.

But if you want Black to leave your string literals as you wrote them and not change the type of quotes used, pass it the `-S` command line option. (Note that the *S* is uppercase.) For example, running Black on the original *yourScript.py* file in Windows would produce the following output:

```
C:\Users\Al>python –m black -S yourScript.py
All done!
1 file left unchanged.
```

You can also use the `-l`line length limit and `-S`options in the same command:

```
C:\Users\Al>python –m black –l 120 -S yourScript.py
```

Previewing the Changes Black Will Make

Although Black won't rename your variable or change your program's behavior, you might not like the style changes Black makes. If you want to stick to your original formatting, you could either use version control for your source code or maintain your own backups. Alternatively, you can preview the changes Black would make without letting it actually alter your files by running Black with the `--diff` command line option. In Windows, it looks like this:

```
C:\Users\Al>python -m black --diff yourScript.py
```

This command outputs the differences in the diff format commonly used by version control software, but it's generally readable by humans. For example, if *yourScript.py* contains the line `weights=[42.0,3.1415,2.718]`, running the `--diff` option would display this result:

```
C:\Users\Al\>python -m black --diff yourScript.py
--- yourScript.py                2020-12-07 02:04:23.141417 +0000
```

```
+++ yourScript.py               2020-12-07 02:08:13.893578 +0000
@@ -1 +1,2 @@
-weights=[42.0,3.1415,2.718]
+weights = [42.0, 3.1415, 2.718]
```

The minus sign indicates that Black would remove the line weights=[42.0,3.1415,2.718] and replace it with the line prefixed with a plus sign: weights = [42.0, 3.1415, 2.718]. Keep in mind that once you've run Black to change your source code files, there's no way to undo this change. You need to either make backup copies of your source code or use version control software, such as Git, before running Black.

Disabling Black for Parts of Your Code

As great as Black is, you might not want it to format some sections of your code. For example, I like to do my own special spacing whenever I'm lining up multiple related assignment statements, as in the following example:

```
# Set up constants for different time amounts:
SECONDS_PER_MINUTE = 60
SECONDS_PER_HOUR   = 60 * SECONDS_PER_MINUTE
SECONDS_PER_DAY    = 24 * SECONDS_PER_HOUR
SECONDS_PER_WEEK   = 7 * SECONDS_PER_DAY
```

Black would remove the additional spaces before the = assignment operator, making them, in my opinion, less readable:

```
# Set up constants for different time amounts:
SECONDS_PER_MINUTE = 60
SECONDS_PER_HOUR = 60 *
SECONDS_PER_MINUTE SECONDS_PER_DAY = 24 *
SECONDS_PER_HOUR SECONDS_PER_WEEK = 7 *
SECONDS_PER_DAY
```

By adding # fmt: off and # fmt: on comments, we can tell Black to turn off its code formatting for these lines and then resume code formatting afterward:

```
# Set up constants for different time amounts:
# fmt: off
SECONDS_PER_MINUTE =
60
SECONDS_PER_HOUR   = 60 * SECONDS_PER_MINUTE
SECONDS_PER_DAY    = 24 * SECONDS_PER_HOUR
SECONDS_PER_WEEK   = 7 * SECONDS_PER_DAY
# fmt: on
```

Running Black on this file now won't affect the unique spacing, or any other formatting, in the code between these two comments.

Summary

Although good formatting can be invisible, poor formatting can make reading code frustrating. Style is subjective, but the software development field generally agrees on what constitutes good and poor formatting while still leaving room for personal preferences.

Python's syntax makes it rather flexible when it comes to style. If you're writing code that nobody else will ever see, you can write it however you like. But much of software development is collaborative. Whether you're working with others on a project or simply want to ask more experienced developers to review your work, formatting your code to fit accepted style guides is important.

Formatting your code in an editor is a boring task that you can automate with a tool like Black. This chapter covered several of the guidelines that Black follows to make your code more readable, including spacing code vertically and horizontally, which keeps it from being too dense to read easily, and setting a limit on how long each line should be. Black enforces these rules for you, preventing potential style arguments with collaborators.

But there's more to code style than spacing and deciding between single and double quotes. For instance, choosing descriptive variable names is also a critical factor for code readability. Although automated tools like Black can make *syntactic* decisions, such as the amount of spacing code should have, they can't make *semantic* decisions, such as what a good variable name is. That responsibility is yours, and we'll discuss this topic in the next chapter.