Chapter 19. Be a Pythonista

Always wanted to travel back in time to try fighting a younger version of yourself? Software development is the career for you!

—Elliot Loh

This chapter is devoted to the art and science of Python development, with "best practice" recommendations. Absorb them, and you too can be a card-carrying Pythonista.

About Programming

First, a few notes about programming, based on personal experience.

My original career path was science, and I taught myself programming to analyze and display experimental data. I expected computer programming to be like my impression of accounting—precise but dull. I was surprised to find that I enjoyed it. Part of the fun was its logical aspects—like solving puzzles—but part was creative. You had to write your program correctly to get the right results, but you had the freedom to write it any way you wanted. It was an unusual balance of right-brain and left-brain thinking.

After I wandered off into a career in programming, I also learned that the field had many niches, with very different tasks and types of people. You could delve into computer graphics, operating systems, business applications—even science.

If you're a programmer, you might have had a similar experience yourself. If you're not, you might try programming a bit to see if it fits your personality, or at least helps you to get something done. As I may have mentioned much earlier in this book, math skills are not so important. It seems that the ability to think logically is most important and that an aptitude for languages seems to help. Finally, patience also helps, especially when you’re tracking down an elusive bug in your code.

Find Python Code

When you need to develop some code, the fastest solution is to steal it… from a source that allows it.

The Python standard library is wide, deep, and mostly clear. Dive in and look for those pearls.

Like the halls of fame for various sports, it takes time for a module to get into the standard library. New packages are appearing outside constantly, and throughout this book, I’ve highlighted some that either do something new or do something old better. Python is advertised as *batteries included*, but you might need a new kind of battery.

So where, outside the standard library, should you look for good Python code?

The first place to look is the Python Package Index (PyPI). Formerly named the *Cheese Shop* after a Monty Python skit, this site is constantly updated with Python packages—more than 113,000 as I write this. When you use `pip` (see the next section), it searches PyPI. The main PyPI page shows the most recently added packages. You can also conduct a direct search by typing something into the search box in the middle of the PyPI home page. For example, `genealogy` yields 21 matches, and `movies` yields 528.

Another popular repository is GitHub. See what Python packages are currently trending.

Popular Python recipes has more than four thousand short Python programs, on every subject.

Install Packages

There are many ways to install Python packages:

- Use `pip` if you can. It's the most common method by far. You can install most of the Python packages you're likely to encounter with `pip`.
- Use `pipenv`, which combines `pip` and `virtualenv`
- Sometimes, you can use a package manager for your operating system.
- Use `conda` if you do a lot of scientific work and want to use the Anaconda distribution of Python. See "Install Anaconda" for details.
- Install from source.

If you're interested in several packages in the same area, you might find a Python distribution that already includes them. For instance, in Chapter 22, you can try out a number of numeric and scientific programs that would be tedious to install individually but are included with distributions such as Anaconda.

Use pip

Python packaging has had some limitations. An earlier installation tool called `easy_install` has been replaced by one called `pip`, but neither had been in the standard Python installation. If you're supposed to install things by using `pip`, from where did you get `pip`? Starting with Python 3.4, `pip` will finally be included with the rest of Python to avoid such existential crises. If you're using an earlier version of Python 3 and don't have `pip`, you can get it from *http://www.pip-installer.org*.

The simplest use of `pip` is to install the latest version of a single package by using the following command:

```
$ pip install flask
```

You will see details on what it's doing, just so you don't think it's goofing off: downloading, running *setup.py*, installing files on your disk, and other details.

You can also ask `pip` to install a specific version:

```
$ pip install flask==0.9.0
```

Or, a minimum version (this is useful when some feature that you can't live without turns up in a particular version):

```
$ pip install 'flask≥0.9.0'
```

In the preceding example, those single quotes prevent the `>` from being interpreted by the shell to redirect output to a file called `=0.9.0`.

If you want to install more than one Python package, you can use a requirements file. Although it has many options, the simplest use is a list of packages, one per line, optionally with a specific or relative version:

```
$ pip -r requirements.txt
```

Your sample *requirements.txt* file might contain this:

```
flask==0.9.0
django
psycopg2
```

A few more examples:

- Install the latest version: `pip install --upgrade` *`package`*
- Delete a package: `pip uninstall` *`package`*

Use virtualenv

The standard way of installing third-party Python packages is to use `pip` and `virtual env`. I show how to install `virtualenv` in "Install virtualenv".

A *virtual environment* is just a directory that contains the Python interpreter, some other programs like `pip`, and some packages. You *activate* it by running the shell script `activate` that's in the `bin` directory of that virtual environment. This sets the environment variable `$PATH` that your shell uses to find programs. By activating a virtual enviroment, you put its `bin` directory ahead of the usual directories in your `$PATH`. The result is that when you type a command like `pip` or `python`, your shell first finds the one in your virtual environment, instead of system directories like `/bin`, `/usr/bin`, or `/usr/local/bin`.

You don’t want to install software into those system directories anyhow, because:

- You don’t have permission to write to them.
- Even if you could, overwriting your system’s standard programs (like `python`) could cause problems.

Use pipenv

A recent package called pipenv combines our friends `pip` and `virtualenv`. It also addresses dependency issues that can arise when using `pip` in different environments (such as your local development machine, versus staging, versus production):

```
$ pip install pipenv
```

Its use is recommended by the Python Packaging Authority—a working group trying to improve Python’s packaging workflow. This is not the same as the group that defines core Python itself, so `pipenv` is not a part of the standard library.

Use a Package Manager

Apple’s macOS includes the third-party packagers homebrew (`brew`) and `ports`. They work a little like `pip`, but aren’t restricted to Python packages.

Linux has a different manager for each distribution. The most popular are `apt-get`, `yum`, `dpkg`, and `zypper`.

Windows has the Windows Installer and package files with a *.msi* suffix. If you installed Python for Windows, it was probably in the MSI format.

Install from Source

Occasionally, a Python package is new, or the author hasn't managed to make it available with `pip`. To build the package, you generally do the following:

- Download the code.
- Extract the files by using `zip`, `tar`, or another appropriate tool if they're archived or compressed.
- Run `python setup.py install` in the directory containing a *setup.py* file.

NOTE

As always, be careful what you download and install. It's a little harder to hide malware in Python programs, which are readable text, but it has happened.

Integrated Development Environments

I've used a plain-text interface for programs in this book, but that doesn't mean that you need to run everything in a console or text window. There are many free and commercial integrated development environments (IDEs), which are GUIs with support for such tools as text editors, debuggers, library searching, and so on.

IDLE

IDLE is the only Python IDE that's included with the standard distribution. It's based on tkinter, and its GUI is plain.

PyCharm

PyCharm is a recent graphic IDE with many features. The community edition is free, and you can get a free license for the professional edition to use in a classroom or an open source project. Figure 19-1 shows its initial display.

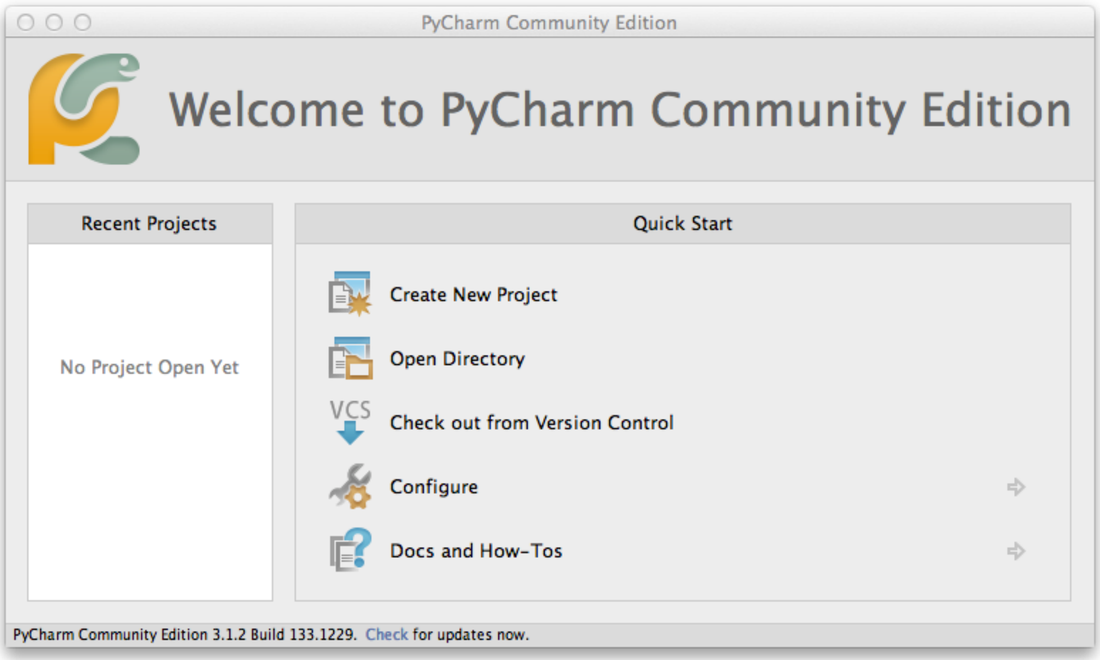
PyCharm Community Edition
Welcome to PyCharm Community Edition
Recent Projects
No Project Open Yet
Quick Start
Create New Project
Open Directory
VCS
Check out from Version Control
Configure
Docs and How-Tos
PyCharm Community Edition 3.1.2 Build 133.1229. Check for updates now.

Figure 19-1. Startup screen for PyCharm

IPython

iPython started as an enhanced terminal (text) Python IDE, but evolved a graphical interface with the metaphor of a *notebook*. It integrated many packages that are discussed in this book, including Matplotlib and NumPy, and became a popular tool in scientific computing.

You install the basic text version with (you guessed it) `pip install ipython` . When you start it, you'll see something like this:

```
$ ipython
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:39:00)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.3.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

As you know, the standard Python interpreter uses the input prompts

`>>>` and `...` to indicate where and when you should type code. IPython tracks everything you type in a list called `In`, and all your output in `Out`. Each input can be more than one line, so you submit it by holding the Shift key while pressing Enter.

Here's a one-line example:

```
In [1]: print("Hello? World?")
Hello? World?

In [2]:
```

`In` and `Out` are automatically numbered lists, letting you access any of the inputs you typed or outputs you received.

If you type `?` after a variable, IPython tells you its type, value, ways of making a variable of that type, and some explanation:

```
In [4]: answer = 42

In [5]: answer?


Type:       int
String Form:42
Docstring:
int(x=0) -> integer
int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments
are given.  If x is a number, return x.__int__().  For floating point
numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string,
bytes, or bytearray instance representing an integer literal in the
given base.  The literal can be preceded by '+' or '-' and be surrounded
by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
Base 0 means to interpret the base from the string as an integer literal.
>>> int('0b100', base=0)
4
```

Name lookup is a popular feature of IDEs such as IPython. If you press the Tab key right after some characters, IPython shows all variables, keywords, and functions that begin with those characters. Let's define some variables and then find everything that begins with the letter `f`:

```
In [6]: fee = 1

In [7]: fie = 2

In [8]: fo = 3

In [9]: fum = 4

In [10]: ftab
%%file    fie       finally   fo        format    frozenset
fee       filter    float     for       from      fum
```

If you type `fe` followed by the Tab key, it expands to the variable `fee`, which, in this program, is the only thing that starts with `fe`:

```
In [11]: fee
Out[11]: 1
```

There's much more to IPython. Take a look at its tutorial to get a feel for its features.

Jupyter Notebook

Jupyter is an evolution of IPython. The name combines the languages Julia, Python, and R—all of which are popular in data science and scientific computing. Jupyter Notebooks are a modern way to develop and publish your code with documentation for any of these languages.

If you'd like to play with it first before installing anything on your computer, you can first try it out in your web browser.

To install Jupyter Notebook locally, type `pip install jupyter`. Run it with `jupyter notebook`.

JupyterLab

JupyterLab is the next generation of Jupyter Notebook and will eventually replace it. As with the Notebook, you can first try out JupyterLab in your browser. You install it locally with `pip install jupyterlab` and then run it with `jupyter lab`.

Name and Document

You won’t remember what you wrote. Sometimes, I look at code that I wrote even recently and wonder where on earth it came from. That’s why it helps to document your code. Documentation can include comments and docstrings, but it can also incorporate informative naming of variables, functions, modules, and classes. Don’t be obsessive, as in this example:

```
>>> # I'm going to assign 10 to the variable "num" here:
... num = 10
>>> # I hope that worked
... print(num)
10
>>> # Whew.
```

Instead, say *why* you assigned the value `10`. Point out why you called the variable `num`. If you were writing the venerable Fahrenheit-to-Celsius converter, you might name variables to explain what they do rather than a lump of magic code. And a little test code wouldn’t hurt (Example 19-1).

Example 19-1. ftoc1.py

```
def ftoc(f_temp):
    "Convert Fahrenheit temperature <f_temp> to Celsius and return it."
    f_boil_temp = 212.0
    f_freeze_temp = 32.0
    c_boil_temp = 100.0
    c_freeze_temp = 0.0
    f_range = f_boil_temp - f_freeze_temp
    c_range = c_boil_temp - c_freeze_temp
    f_c_ratio = c_range / f_range
    c_temp = (f_temp - f_freeze_temp) * f_c_ratio + c_freeze_temp
    return c_temp

if __name__ == '__main__':
    for f_temp in [-40.0, 0.0, 32.0, 100.0, 212.0]:
        c_temp = ftoc(f_temp)
        print('%f F => %f C' % (f_temp, c_temp))
```

Let’s run the tests:

```
$ python ftoc1.py
```

```
-40.000000 F => -40.000000 C
0.000000 F => -17.777778 C
32.000000 F => 0.000000 C
100.000000 F => 37.777778 C
212.000000 F => 100.000000 C
```

We can make (at least) two improvements:

- Python doesn’t have constants, but the PEP8 stylesheet recommends using capital letters and underscores (e.g., `ALL_CAPS`) when naming variables that should be considered constants. Let’s rename those constant-y variables in our example.
- Because we precompute values based on constant values, let’s move them to the top level of the module. Then, they’ll be calculated only once rather than in every call to the `ftoc()` function.

Example 19-2 shows the result of our rework.

Example 19-2. ftoc2.py

```
F_BOIL_TEMP = 212.0
F_FREEZE_TEMP = 32.0
C_BOIL_TEMP = 100.0
C_FREEZE_TEMP = 0.0
```

```
F_RANGE = F_BOIL_TEMP - F_FREEZE_TEMP
C_RANGE = C_BOIL_TEMP - C_FREEZE_TEMP
F_C_RATIO = C_RANGE / F_RANGE

def ftoc(f_temp):
    "Convert Fahrenheit temperature <f_temp> to Celsius and return it."
    c_temp = (f_temp - F_FREEZE_TEMP) * F_C_RATIO + C_FREEZE_TEMP
    return c_temp

if __name__ == '__main__':
    for f_temp in [-40.0, 0.0, 32.0, 100.0, 212.0]:
        c_temp = ftoc(f_temp)
        print('%f F => %f C' % (f_temp, c_temp))
```

Add Type Hints

Static languages require you to define the types of your variables, and they can catch some errors at compile time. As you know, Python doesn't do this, and you can encounter bugs only when the code is run. Python variables are names and only refer to actual objects. Objects have strict types, but names can point to any object at any time.

Yet in real code (in Python and other languages), a name tends to refer to a particular object. It would help, at least in documentation, if we could annotate things (variables, function returns, and so on) with the object types we expect them to be referencing. Then, developers would not need to look through as much code to see how a particular variable is supposed to act.

Python 3.x added *type hints* (or *type annotations*) to address this. It's completely optional, and does not force types on variables. It helps developers who are used to static languages, where variable types *must* be declared.

Hints for a function that converts a number to a string would look like this:

```
def num_to_str(num: int) -> str:
    return str(num)
```

These are only hints, and they don't change how Python works. Their main use is for documentation, but people are finding more uses. For example, the FastAPI web framework uses hints to generate web documentation with live forms for testing.

Test

You probably already know this, but if not: even trivial code changes can break your program. Python lacks the type-checking of static languages, which makes some things easier but also lets undesirable results through the door. Testing is essential.

The very simplest way to test Python programs is to add `print()` statements. The Python interactive interpreter's Read-Evaluate-Print Loop (REPL) lets you edit and test changes quickly. However, you don't want `print()` statements in production code, so you need to remember to remove them all.

Check with pylint, pyflakes, flake8, or pep8

The next step, before creating actual test programs, is to run a Python code checker. The most popular are `pylint` and `pyflakes`. You can install either or both by using `pip`:

```
$ pip install pylint
$ pip install pyflakes
```

These check for actual code errors (such as referring to a variable before assigning it a value) and style faux pas (the code equivalent of wearing plaids and stripes). Example 19-3 is a fairly meaningless program with a bug and style issue.

Example 19-3. style1.py

```
a = 1
b = 2
print(a)
print(b)
print(c)
```

Here's the initial output of `pylint`:

```
$ pylint style1.py
No config file found, using default configuration
************* Module style1
C:  1,0: Missing docstring
C:  1,0: Invalid name "a" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C:  2,0: Invalid name "b" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
E:  5,6: Undefined variable 'c'
```

Much further down, under `Global evaluation`, is our score (10.0 is perfect):

```
Your code has been rated at -3.33/10
```

Ouch. Let's fix the bug first. That `pylint` output line starting with an `E` indicates an `Error`, which occurred because we didn't assign a value to `c` before we printed it. Take a look at Example 19-4 to see how we can fix that.

Example 19-4. style2.py

```
a = 1
b = 2
c = 3
print(a)
print(b)
print(c)
```

```
$ pylint style2.py


No config file found, using default configuration
************* Module style2
C:  1,0: Missing docstring
C:  1,0: Invalid name "a" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C:  2,0: Invalid name "b" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C:  3,0: Invalid name "c" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
```

Good, no more `E` lines. And our score jumped from -3.33 to 4.29:

```
Your code has been rated at 4.29/10
```

`pylint` wants a docstring (a short text at the top of a module or function, describing the code), and it thinks short variable names such as `a`, `b`, and `c` are tacky. Let's make `pylint` happier and improve *style2.py* to *style3.py* (Example 19-5).

Example 19-5. style3.py

```python
"Module docstring goes here"

def func():
    "Function docstring goes here. Hi, Mom!"
    first = 1
    second = 2
    third = 3
    print(first)
    print(second)
    print(third)

func()
```

```
$ pylint style3.py
No config file found, using default configuration
```

Hey, no complaints. And our score?

```
Your code has been rated at 10.00/10
```

And there was much rejoicing.

Another style checker is `pep8`, which you can install in your sleep by now:

```
$ pip install pep8
```

What does it say about our style makeover?

```
$ pep8 style3.py
style3.py:3:1: E302 expected 2 blank lines, found 1
```

To be really stylish, it's recommending that I add a blank line after the initial module docstring.

Test with unittest

We've verified that we're no longer insulting the style senses of the code gods, so let's move on to actual tests of the logic in your program.

It's a good practice to write independent test programs first, to ensure that they all pass before you commit your code to any source control system. Writing tests can seem tedious at first, but they really do help you find problems faster—especially *regressions* (breaking something that used to work). Painful experience teaches all developers that even the teeniest change, which they swear could not possibly affect anything else, actually does. If you look at well-written Python packages, they always include a test suite.

The standard library contains not one, but two test packages. Let's start with `unittest`. We'll write a module that capitalizes words. Our first version just uses the standard string function `capitalize()`, with some unexpected results as we'll see. Save this as *cap.py* (Example 19-6).

Example 19-6. cap.py

```
def just_do_it(text):
    return text.capitalize()
```

The basis of testing is to decide what outcome you want from a certain input (here, you want the capitalized version of whatever text you input), submit the input to the function you're testing, and then check whether it returned the expected results. The expected result is called an *assertion*, so in `unittest` you check your results by using methods with names that begin with `assert`, like the `assertEqual` method shown in Example 19-7.

Save this test script as *test_cap.py*.

Example 19-7. test_cap.py

```
import unittest
import cap

class TestCap(unittest.TestCase):

    def setUp(self):
        pass

    def tearDown(self):
        pass

    def test_one_word(self):
        text = 'duck'
        result = cap.just_do_it(text)
        self.assertEqual(result, 'Duck')

    def test_multiple_words(self):
        text = 'a veritable flock of ducks'
        result = cap.just_do_it(text)
        self.assertEqual(result, 'A Veritable Flock Of Ducks')


if __name__ == '__main__':
    unittest.main()
```

The `setUp()` method is called before each test method, and the `tearDown()` method is called after each. Their purpose is to allocate and free external resources needed by the tests, such as a database connection or some test data. In this case, our tests are self-contained, and we wouldn't even need to define `setUp()` and `tearDown()`, but it doesn't hurt to have empty versions there. At the heart of our test are the two functions named `test_one_word()` and `test_multiple_words()`. Each runs the `just_do_it()` function we defined with different input and checks whether we got back what we expect. Okay, let's run it. This will call our two test methods:

```
$ python test_cap.py
```

```
F.
======================================================================
FAIL: test_multiple_words (__main__.TestCap)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_cap.py", line 20, in test_multiple_words
 self.assertEqual(result, 'A Veritable Flock Of Ducks')
AssertionError: 'A veritable flock of ducks' != 'A Veritable Flock Of Ducks'
- A veritable flock of ducks
?   ^         ^     ^  ^
+ A Veritable Flock Of Ducks
?   ^         ^     ^  ^


----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

It liked the first test (`test_one_word`), but not the second (`test_multiple_words`). The up arrows (`^`) show where the strings actually differed.

What's special about multiple words? Reading the documentation for the `string` `capitalize` function yields an important clue: it capitalizes only the first letter of the first word. Maybe we should have read that first.

Consequently, we need another function. Gazing down that page a bit, we find `title()` . So, let's change *cap.py* to use `title()` instead of `capitalize()` (Example 19-8).

Example 19-8. cap.py, revised

```
def just_do_it(text):
    return text.title()
```

Rerun the tests, and let's see what happens:

```
$ python test_cap.py
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Everything is great. Well, actually, they're not. We need to add at least one more method to *test_cap.py* (Example 19-9).

Example 19-9. test_cap.py, revised

```
    def test_words_with_apostrophes(self):
        text = "I'm fresh out of ideas"
        result = cap.just_do_it(text)
        self.assertEqual(result, "I'm Fresh Out Of Ideas")
```

Go ahead and try it again:

```
$ python test_cap.py
```

```
..F
======================================================================
FAIL: test_words_with_apostrophes (__main__.TestCap)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_cap.py", line 25, in test_words_with_apostrophes
    self.assertEqual(result, "I'm Fresh Out Of Ideas")
AssertionError: "I'M Fresh Out Of Ideas" != "I'm Fresh Out Of Ideas"
- I'M Fresh Out Of Ideas
?   ^
+ I'm Fresh Out Of Ideas
?   ^


----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

Our function capitalized the `m` in `I'm`. A quick run back to the documentation for `title()` shows that it doesn't handle apostrophes well. We *really* should have read the entire text first.

At the bottom of the standard library's `string` documentation is another candidate: a helper function called `capwords()`. Let's use it in *cap.py* (Example 19-10).

Example 19-10. cap.py, re-revised

```
def just_do_it(text):
    from string import capwords
    return capwords(text)
```

```
$ python test_cap.py
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.004s

OK
```

At last, we're finally done! Uh, no. We have one more test to add to *test_cap.py* (Example 19-11).

Example 19-11. test_cap.py, re-revised

```
    def test_words_with_quotes(self):
        text = "\"You're despicable,\" said Daffy Duck"
        result = cap.just_do_it(text)
        self.assertEqual(result, "\"You're Despicable,\" Said Daffy Duck")
```

Did it work?

```
$ python test_cap.py
```

```
...F
======================================================================
FAIL: test_words_with_quotes (__main__.TestCap)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_cap.py", line 30, in test_words_with_quotes
    self.assertEqual(result, "\"You're
    Despicable,\" Said Daffy Duck") AssertionError: '"you\'re Despicable,"
        Said Daffy Duck'
 != '"You\'re Despicable," Said Daffy Duck' - "you're Despicable,"
        Said Daffy Duck
?  ^ + "You're Despicable," Said Daffy Duck
?  ^
----------------------------------------------------------------------
Ran 4 tests in 0.004s

FAILED (failures=1)
```

It looks like that first double quote confused even `capwords`, our favorite capitalizer thus far. It tried to capitalize the `"`, and lowercased the rest (`You're`). We should have also tested that our capitalizer left the rest of the string untouched.

People who do testing for a living have a knack for spotting these edge cases, but developers often have blind spots when it comes to their own code.

`unittest` provides a small but powerful set of assertions, letting you check values, confirm whether you have the class you want, determine whether an error was raised, and so on.

Test with doctest

The second test package in the standard library is `doctest`. With this package, you can write tests within the docstring itself, also serving as documentation. It looks like the interactive interpreter: the characters `>>>`, followed by the call, and then the results on the following line. You can run some tests in the interactive interpreter and just paste the results into your test file. Let's modify our old *cap.py* as *cap2.py* (without that troublesome last test with quotes), as shown in Example 19-12.

Example 19-12. cap2.py

```
def just_do_it(text):
    """
    >>> just_do_it('duck')
    'Duck'
    >>> just_do_it('a veritable flock of ducks')
    'A Veritable Flock Of Ducks'
    >>> just_do_it("I'm fresh out of ideas")
    "I'm Fresh Out Of Ideas"
    """
    from string import capwords
    return capwords(text)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

When you run it, it doesn’t print anything if all tests passed:

```
$ python cap2.py
```

Give it the verbose (`-v`) option to see what actually happened:

```
$ python cap2.py -v
```

```
Trying:
    just_do_it('duck')
    Expecting:
    'Duck'
ok
Trying:
     just_do_it('a veritable flock of ducks')
Expecting:
     'A Veritable Flock Of Ducks'
ok
Trying:
     just_do_it("I'm fresh out of ideas")
Expecting:
     "I'm Fresh Out Of Ideas"
ok
1 items had no tests:
     __main__
1 items passed all tests:
    3 tests in __main__.just_do_it
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
```

Test with nose

The third-party package called `nose` is another alternative to `unittest` . Here’s the command to install it:

```
$ pip install nose
```

You don’t need to create a class that includes test methods, as we did with `unittest`. Any function with a name matching `test` somewhere in its name will be run. Let’s modify our last version of our `unittest` tester and save it as *test_cap_nose.py* (Example 19-13).

Example 19-13. test_cap_nose.py

```
import cap2
from nose.tools import eq_

def test_one_word():
    text = 'duck'
    result = cap.just_do_it(text)
    eq_(result, 'Duck')

def test_multiple_words():
    text = 'a veritable flock of ducks'
    result = cap.just_do_it(text)
    eq_(result, 'A Veritable Flock Of Ducks')

def test_words_with_apostrophes():
    text = "I'm fresh out of ideas"
    result = cap.just_do_it(text)
    eq_(result, "I'm Fresh Out Of Ideas")

def test_words_with_quotes():
    text = "\"You're despicable,\" said Daffy Duck"
    result = cap.just_do_it(text)
    eq_(result, "\"You're Despicable,\" Said Daffy Duck")
```

Run the tests:

```
$ nosetests test_cap_nose.py
```

```
...F
======================================================================
FAIL: test_cap_nose.test_words_with_quotes
```

```
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/.../site-packages/nose/case.py", line 198, in runTest
    self.test(*self.arg)
  File "/Users/.../book/test_cap_nose.py", line 23, in test_words_with_quotes
    eq_(result, "\"You're Despicable,\" Said Daffy Duck")
AssertionError: '"you\'re Despicable," Said Daffy Duck'      !=
'"You\'re Despicable," Said Daffy Duck'
----------------------------------------------------------------------
Ran 4 tests in 0.005s

FAILED (failures=1)
```

This is the same bug we found when we used `unittest` for testing; fortunately, there's an exercise to fix it at the end of this chapter.

Other Test Frameworks

For some reason, people like to write Python test frameworks. If you're curious, you can check out some other popular ones:

- `tox`
- `py.test`
- `green`

Continuous Integration

When your group is cranking out a lot of code daily, it helps to automate tests as soon as changes arrive. You can automate source control systems to run tests on all code as it's checked in. This way, everyone knows whether someone *broke the build* and just disappeared for an early lunch—or a new job.

These are big systems, and I'm not going into installation and usage details here. In case you need them someday, you'll know where to find them:

`buildbot`

Written in Python, this source control system automates building, testing, and releasing.

`jenkins`
: This is written in Java, and seems to be the preferred CI tool of the moment.

`travis-ci`
: This automates projects hosted at GitHub, and is free for open source projects.

`circleci`
: This one is commercial but free for open source and private projects.

Debug Python Code

Debugging is like being the detective in a crime movie where you are also the murderer.

—Filipe Fortes

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

—Brian Kernighan

Test first. The better your tests are, the less you'll have to fix later. Yet, bugs happen and need to be fixed when they're found later.

When code breaks, it's usually because of something you just did. So you typically debug "from the bottom up," starting with your most recent changes.[1]

But sometimes the cause is elsewhere, in something that you trusted and thought worked. You would think that if there were problems in something that many people use, someone would have noticed by now. That is not always what happens. The trickiest bugs that I've encountered, that each took more than a week to fix, had external causes. So after blaming the person in the mirror, question your assumptions. This is a "top-down" approach, and it takes longer.

The following are some debugging techniques, from quick and dirty to slower, but often just as dirty.

Use print()

The simplest way to debug in Python is to print out strings. Some useful things to print include `vars()`, which extracts the values of your local variables, including function arguments:

```
>>> def func(*args, **kwargs):
...     print(vars())
...
>>> func(1, 2, 3)
{'args': (1, 2, 3), 'kwargs': {}}
>>> func(['a', 'b', 'argh'])
{'args': (['a', 'b', 'argh'],), 'kwargs': {}}
```

Other things that are often worth printing are `locals()` and `globals()`.

If your code also includes normal prints to standard output, you can write your debugging messages to standard error output with `print(`*`stuff`*`, file=sys.stderr)`.

Use Decorators

As you read in "Decorators", a decorator can call code before or after a function without modifying the code within the function itself. This means that you can use a decorator to do something before or after any Python function, not just ones that you wrote. Let's define the decorator `dump` to print the input arguments and output values of any function as it's called (designers know that a dump often needs decorating), as shown in Example 19-14.

Example 19-14. dump.py

```
def dump(func):
    "Print input arguments and output value(s)"
    def wrapped(*args, **kwargs):
        print("Function name:", func.__name__)
        print("Input arguments:", ' '.join(map(str, args)))
        print("Input keyword arguments:", kwargs.items())
        output = func(*args, **kwargs)
        print("Output:", output)
        return output
    return wrapped
```

Now the decoratee. This is a function called `double()` that expects numeric arguments, either named or unnamed, and returns them in a list with their values doubled (Example 19-15).

Example 19-15. test_dump.py

```
from dump import dump

@dump
def double(*args, **kwargs):
    "Double every argument"
    output_list = [ 2 * arg for arg in args ]
    output_dict =  { k:2*v for k,v in kwargs.items() }
    return output_list, output_dict

if __name__ == '__main__':
    output = double(3, 5, first=100, next=98.6, last=-40)
```

Take a moment to run it:

```
$ python test_dump.py
```

```
Function name: double
Input arguments: 3 5
Input keyword arguments: dict_items([('first', 100), ('next', 98.6),
    ('last', -40)])
```

```
Output: ([6, 10], {'first': 200, 'next': 197.2, 'last': -80})
```

Use pdb

These techniques help, but sometimes there's no substitute for a real debugger. Most IDEs include a debugger, with varying features and user interfaces. Here, I describe use of the standard Python debugger, `pdb`.

NOTE

If you run your program with the `-i` flag, Python will drop you into its interactive interpreter if the program fails.

Here's a program with a bug that depends on data—the kind of bug that can be particularly hard to find. This is a real bug from the early days of computing, and it baffled programmers for quite a while.

We're going to read a file of countries and their capital cities, separated by a comma, and write them out as *capital*, *country*. They might be capitalized incorrectly, so we should fix that also when we print. Oh, and there might be extra spaces here and there, and you'll want to get rid of those, too. Finally, although it would make sense for the program to just read to the end of the file, for some reason our manager told us to stop when we encounter the word `quit` (in any mixture of uppercase and lowercase characters). Example 19-16 shows a sample data file.

Example 19-16. cities.csv

```
France, Paris
venuzuela,caracas
  LithuaniA,vilnius
     quit
```

Let's design our *algorithm* (method for solving the problem). This is *pseudocode*—it looks like a program, but is just a way to explain the logic in normal language before converting it to an actual program. One reason programmers like Python is because it *looks a lot like pseudocode*, so there's less work involved when it's time to convert it to a working program:

```
for each line in the text file:
    read the line
    strip leading and trailing spaces
    if `quit` occurs in the lower-case copy of the line:
        stop
    else:
        split the country and capital by the comma character
        trim any leading and trailing spaces
        convert the country and capital to titlecase
        print the capital, a comma, and the country
```

We need to strip initial and trailing spaces from the names because that was a requirement. Likewise for the lowercase comparison with `quit` and converting the city and country names to title case. That being the case, let's whip out *capitals.py*, which is sure to work perfectly (Example 19-17).

Example 19-17. capitals.py

```python
def process_cities(filename):
    with open(filename, 'rt') as file:
        for line in file:
            line = line.strip()
            if 'quit' in line.lower():
                return
            country, city = line.split(',')
            city = city.strip()
            country = country.strip()
            print(city.title(), country.title(), sep=',')

if __name__ == '__main__':
    import sys
    process_cities(sys.argv[1])
```

Let's try it with that sample data file we made earlier. Ready, fire, aim:

```
$ python capitals.py cities.csv
Paris,France
Caracas,Venuzuela
Vilnius,Lithuania
```

Looks great! It passed one test, so let's put it in production, processing capitals and countries from around the world—until it fails, but only for this data file (Example 19-18).

Example 19-18. cities2.csv

```
argentina,buenos aires
bolivia,la paz
brazil,brasilia
chile,santiago
colombia,Bogotá
ecuador,quito
falkland islands,stanley
french guiana,cayenne
guyana,georgetown
paraguay,Asunción
peru,lima
suriname,paramaribo
uruguay,montevideo
venezuela,caracas
quit
```

The program ends after printing only 5 lines of the 15 in the data file, as demonstrated here:

```
$ python capitals.py cities2.csv
Buenos Aires,Argentina
La Paz,Bolivia
Brazilia,Brazil
Santiago,Chile
Bogotá,Colombia
```

What happened? We can keep editing *capitals.py*, putting `print()` statements in likely places, but let's see if the debugger can help us.

To use the debugger, import the `pdb` module from the command line by typing **`-m pdb`**, like so:

```
$ python -m pdb capitals.py cities2.csv
```

```
> /Users/williamlubanovic/book/capitals.py(1)<module>()
-> def process_cities(filename):
(Pdb)
```

This starts the program and places you at the first line. If you type **`c`** (*continue*), the program will run until it ends, either normally or with an error:

```
(Pdb) c
```

```
Buenos Aires,Argentina
La Paz,Bolivia
Brazilia,Brazil
Santiago,Chile
Bogotá,Colombia
The program finished and will be restarted
> /Users/williamlubanovic/book/capitals.py(1)<module>()
-> def process_cities(filename):
```

It completed normally, just as it did when we ran it earlier outside of the debugger. Let's try again, using some commands to narrow down where the problem lies. It seems to be a *logic error* rather than a syntax problem or exception (which would have printed error messages).

Type **`s`** (*step*) to single-step through Python lines. This steps through *all* Python code lines: yours, the standard library's, and any other modules you might be using. When you use `s`, you also go into functions and single-step within them. Type **`n`** (*next*) to single-step but *not* to go inside functions; when you get to a function, a single `n` causes the entire function to execute and take you to the next line of your program. Thus, use `s` when you're not sure where the problem is; use `n` when you're sure that a particular function isn't the cause, especially if it's a long function. Often you'll single-step through your own code and step over library code, which is presumably well tested. Let's use `s` to step from the beginning of the program into the function `process_cities()`:

```
(Pdb) s


> /Users/williamlubanovic/book/capitals.py(12)<module>()
 -> if __name__ == '__main__':</pre>


(Pdb) s


 > /Users/williamlubanovic/book/capitals.py(13)<module>()
 -> import sys


(Pdb) s


 > /Users/williamlubanovic/book/capitals.py(14)<module>()
 -> process_cities(sys.argv[1])


(Pdb) s


 --Call--
 > /Users/williamlubanovic/book/capitals.py(1)process_cities()
 -> def process_cities(filename):


(Pdb) s


 > /Users/williamlubanovic/book/capitals.py(2)process_cities()
 -> with open(filename, 'rt') as file:
```

Type **`l`** (*list*) to see the next few lines of your program:

```
(Pdb) l
```

```
  1     def process_cities(filename):
  2  ->     with open(filename, 'rt') as file:
  3             for line in file:
  4                 line = line.strip()
  5                 if 'quit' in line.lower():
  6                     return
  7                 country, city = line.split(',')
  8                 city = city.strip()
  9                 country = country.strip()
 10                 print(city.title(), country.title(), sep=',')
 11
 (Pdb)
```

The arrow (`->`) denotes the current line.

We could continue using `s` or `n`, hoping to spot something, but let's use one of the main features of a debugger: *breakpoints*. A breakpoint stops execution at the line you indicate. In our case, we want to know why `process_cities()` bails out before it's read all of the input lines. Line 3 (`for line in file:`) will read every line in the input file, so that seems innocent. The only other place where we could return from the function before reading all of the data is at line 6 (`return`). Let's set a breakpoint on line 6:

```
(Pdb) b 6
```

```
Breakpoint 1 at /Users/williamlubanovic/book/capitals.py:6
```

Next, let's continue the program until it either hits the breakpoint or reads all of the input lines and finishes normally:

```
(Pdb) c
```

```
 Buenos Aires,Argentina
```

```
La Paz,Bolivia
Brasilia,Brazil
Santiago,Chile
Bogotá,Colombia
> /Users/williamlubanovic/book/capitals.py(6)process_cities()
-> return
```

Aha! It stopped at our line 6 breakpoint. This indicates that the program wants to return early after reading the country after Colombia. Let's print the value of `line` to see what we just read:

```
(Pdb) p line
```

```
'ecuador,quito'
```

What's so special about—oh, never mind.

Really? **`quit`** o? Our manager never expected the string `quit` to turn up inside normal data, so using it as a *sentinel* (end indicator) value like this was a boneheaded idea. You march right in there and tell him that—while I wait here.

If at this point you still have a job, you can see all your breakpoints by using a plain `b` command:

```
(Pdb) b
```

```
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/williamlubanovic/book/capitals.py:6
    breakpoint already hit 1 time
```

An `l` will show your code lines, the current line (`->`), and any breakpoints (`B`). A plain `l` will start listing from the end of your previous call to `l`, so include the optional starting line (here, let's start from line `1`):

```
(Pdb) l 1

  1     def process_cities(filename):
  2         with open(filename, 'rt') as file:
  3             for line in file:
  4                 line = line.strip()
  5                 if 'quit' in line.lower():
  6 B->                 return
  7                 country, city = line.split(',')
  8                 city = city.strip()
  9                 country = country.strip()
 10                 print(city.title(), country.title(), sep=',')
 11
```

OK, as shown in Example 19-19, let’s fix that `quit` test to match only the full line, not within other characters.

Example 19-19. capitals2.py

```python
def process_cities(filename):
    with open(filename, 'rt') as file:
        for line in file:
            line = line.strip()
            if 'quit' == line.lower():
                return
            country, city = line.split(',')
            city = city.strip()
            country = country.strip()
            print(city.title(), country.title(), sep=',')

if __name__ == '__main__':
    import sys
    process_cities(sys.argv[1])
```

Once more, with feeling:

```
$ python capitals2.py cities2.csv
```

```
Buenos Aires,Argentina
La Paz,Bolivia
Brasilia,Brazil
Santiago,Chile
Bogotá,Colombia
Quito,Ecuador
Stanley,Falkland Islands
Cayenne,French Guiana
Georgetown,Guyana
Asunción,Paraguay
Lima,Peru
Paramaribo,Suriname
Montevideo,Uruguay
Caracas,Venezuela
```

That was a skimpy overview of the debugger—just enough to show you what it can do and what commands you'd use most of the time.

Remember: more tests, less debugging.

Use breakpoint()

In Python 3.7, there's a new built-in function called `breakpoint()`. If you add it to your code, a debugger will automatically start up and pause at each location. Without this, you would need to fire up a debugger like `pdb` and set breakpoints manually, as you saw earlier.

The default debugger is the one you've just seen (`pdb`), but this can be changed by setting the environment variable `PYTHONBREAKPOINT`. For example, you could specify use of the web-based remote debugger web-pdb:

```
$ export PYTHONBREAKPOINT='web_pdb.set_trace'
```

The official documentation is a bit dry, but there are good overviews here and there.

Log Error Messages

At some point you might need to graduate from using `print()` statements to logging messages. A log is usually a system file that accumulates messages, often inserting useful information such as a timestamp or the name of the user who's running the program. Often logs are *rotated* (renamed) daily and compressed; by doing so, they don't fill up your disk and cause problems themselves. When something goes wrong with your program, you can look at the appropriate log file to see what happened. The contents of exceptions are especially useful in logs because they show you the actual line at which your program croaked, and why.

The standard Python library module is `logging`. I've found most descriptions of it somewhat confusing. After a while it makes more sense, but it does seem overly complicated at first. The `logging` module includes these concepts:

- The *message* that you want to save to the log
- Ranked priority *levels* and matching functions: `debug()`, `info()`, `warn()`, `error()`, and `critical()`
- One or more *logger* objects as the main connection with the module
- *Handlers* that direct the message to your terminal, a file, a database, or somewhere else
- *Formatters* that create the output
- *Filters* that make decisions based on the input

For the simplest logging example, just import the module and use some of its functions:

```
>>> import logging
>>> logging.debug("Looks like rain")
>>> logging.info("And hail")
>>> logging.warn("Did I hear thunder?")
WARNING:root:Did I hear thunder?
>>> logging.error("Was that lightning?")
ERROR:root:Was that lightning?
>>> logging.critical("Stop fencing and get inside!")
CRITICAL:root:Stop fencing and get inside!
```

Did you notice that `debug()` and `info()` didn't do anything, and the other two printed *LEVEL*: `root` : before each message? So far, it’s like a `print()` statement with multiple personalities, some of them hostile.

But it is useful. You can scan for a particular value of *LEVEL* in a log file to find particular messages, compare timestamps to see what happened before your server crashed, and so on.

A lot of digging through the documentation answers the first mystery (we get to the second one in a page or two): the default priority *level* is `WARNING` , and that got locked in as soon as we called the first function (`logging.debug()`). We can set the default level by using `basicConfig()` . `DEBUG` is the lowest level, so this enables it and all the higher levels to flow through:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> logging.debug("It's raining again")
DEBUG:root:It's raining again
>>> logging.info("With hail the size of hailstones")
INFO:root:With hail the size of hailstones
```

We did all that with the default `logging` functions without actually creating a *logger* object. Each logger has a name. Let’s make one called `bunyan` :

```
>>> import logging
>>> logging.basicConfig(level='DEBUG')
>>> logger = logging.getLogger('bunyan')
>>> logger.debug('Timber!')
DEBUG:bunyan:Timber!
```

If the logger name contains any dot characters, they separate levels of a hierarchy of loggers, each with potentially different properties. This means that a logger named `quark` is higher than one named `quark.charmed` . The special *root logger* is at the top and is called `''` .

So far, we’ve just printed messages, which is not a great improvement over `print()` . We use *handlers* to direct the messages to different places.

The most common is a *log file*, and here's how you do it:

```
>>> import logging
>>> logging.basicConfig(level='DEBUG', filename='blue_ox.log')
>>> logger = logging.getLogger('bunyan')
>>> logger.debug("Where's my axe?")
>>> logger.warn("I need my axe")
>>>
```

Aha! The lines aren't on the screen anymore; instead, they're in the file named *blue_ox.log*:

```
DEBUG:bunyan:Where's my axe?
WARNING:bunyan:I need my axe
```

Calling `basicConfig()` with a `filename` argument created a `FileHandler` for you and made it available to your logger. The `logging` module includes at least 15 handlers to send messages to places such as email and web servers as well as the screen and files.

Finally, you can control the *format* of your logged messages. In our first example, our default gave us something similar to this:

```
WARNING:root:Message...
```

If you provide a `format` string to `basicConfig()`, you can change to the format of your preference:

```
>>> import logging
>>> fmt = '%(asctime)s %(levelname)s %(lineno)s %(message)s'
>>> logging.basicConfig(level='DEBUG', format=fmt)
>>> logger = logging.getLogger('bunyan')
>>> logger.error("Where's my other plaid shirt?")
2014-04-08 23:13:59,899 ERROR 1 Where's my other plaid shirt?
```

We let the logger send output to the screen again, but changed the format.

The `logging` module recognizes a number of variable names in the `fmt` format string. We used `asctime` (date and time as an ISO 8601 string), `levelname`, `lineno` (line number), and the `message` itself. There are other built-ins, and you can provide your own variables, as well.

There's much more to `logging` than this little overview can provide. You can log to more than one place at the same time, with different priorities and formats. The package has a lot of flexibility but sometimes at the cost of simplicity.

Optimize

Python is usually fast enough—until it isn't. In many cases, you can gain speed by using a better algorithm or data structure. The trick is knowing where to do this. Even experienced programmers guess wrong surprisingly often. You need to be like the careful quiltmaker and measure before you cut. And this leads us to *timers*.

Measure Timing

You've seen that the `time()` function in the `time` module returns the current epoch time as a floating-point number of seconds. A quick way of timing something is to get the current time, do something, get the new time, and then subtract the original time from the new time. Let's write this up, as presented in Example 19-20, and call it (what else?) *time1.py*.

Example 19-20. time1.py

```
from time import time

t1 = time()
num = 5
num *= 2
print(time() - t1)
```

In this example, we're measuring the time it takes to assign the value `5` to the name `num` and multiply it by `2`. This is *not* a realistic benchmark, just an example of how to measure some arbitrary Python code. Try running it a few times, just to see how much it can vary:

```
$ python time1.py
2.1457672119140625e-06
$ python time1.py
2.1457672119140625e-06
$ python time1.py
2.1457672119140625e-06
$ python time1.py
1.9073486328125e-06
$ python time1.py
3.0994415283203125e-06
```

That was about two or three millionths of a second. Let's try something slower, such as `sleep()`.[2] If we sleep for a second, our timer should take a tiny bit more than a second. Example 19-21 shows the code; save this as *time2.py*.

Example 19-21. time2.py

```
from time import time, sleep

t1 = time()
sleep(1.0)
print(time() - t1)
```

Let's be certain of our results, so run it a few times:

```
$ python time2.py
1.000797986984253
$ python time2.py
1.0010130405426025
$ python time2.py
1.0010390281677246
```

As expected, it takes about a second to run. If it didn't, either our timer or `sleep()` should be embarrassed.

There's a handier way to measure code snippets like this: using the standard module `timeit`. It has a function called (you guessed it) `timeit()`, which will do *count* runs of your test *code* and print some results. The syntax is: `timeit.timeit(` *`code`* `, number=` *`count`* `)`.

In the examples in this section, the `code` needs to be within quotes so that it is not executed after you press the Return key but is executed inside `timeit()`. (In the next section, you'll see how to time a function by passing its name to `timeit()`.) Let's run our previous example just once and time it. Call this file *timeit1.py* (Example 19-22).

Example 19-22. timeit1.py

```
from timeit import timeit
print(timeit('num = 5; num *= 2', number=1))
```

Run it a few times:

```
$ python timeit1.py
2.5600020308047533e-06
$ python timeit1.py
1.9020008039660752e-06
$ python timeit1.py
1.7380007193423808e-06
```

Again, these two code lines ran in about two millionths of a second. We can use the `repeat` argument of the `timeit` module's `repeat()` function to run more sets. Save this as *timeit2.py* (Example 19-23).

Example 19-23. timeit2.py

```
from timeit import repeat
print(repeat('num = 5; num *= 2', number=1, repeat=3))
```

Try running it to see what transpires:

```
$ python timeit2.py
```

```
[1.691998477326706e-06, 4.070025170221925e-07, 2.4700057110749185e-07]
```

The first run took two millionths of a second, and the second and third runs were faster. Why? There could be many reasons. For one thing, we're testing a very small piece of code, and its speed could depend on what else the computer was doing in those instants, how the Python system optimizes calculations, and many other things.

Using `timeit()` meant wrapping the code you're trying to measure as a string. What if you have multiple lines of code? You could pass it a triple-quoted multiline string, but that might be hard to read.

Let's define a lazy `snooze()` function that nods off for a second, as we all do occasionally.

First, we can wrap the `snooze()` function itself. We need to include the arguments `globals=globals()` (this helps Python to find `snooze`) and `number=1` (run it only once; the default is 1000000, and we don't have that much time):

```
>>> import time
>>> from timeit import timeit
>>>
>>> def snooze():
...     time.sleep(1)
...
>>> seconds = timeit('snooze()', globals=globals(), number=1)
>>> print("%.4f" % seconds)
1.0035
```

Or, we can use a decorator:

```
>>> import time
>>>
>>> def snooze():
...     time.sleep(1)
...
>>> def time_decorator(func):
...     def inner(*args, **kwargs):
```

```
...         t1 = time.time()
...         result = func(*args, **kwargs)
...         t2 = time.time()
...         print(f"{(t2-t1):.4f}")
...         return result
...     return inner
...
>>> @time_decorator
... def naptime():
...     snooze()
...
>>> naptime()
1.0015
```

Another way is to use a context manager:

```
>>> import time
>>>
>>> def snooze():
...     time.sleep(1)
...
>>> class TimeContextManager:
...     def __enter__(self):
...         self.t1 = time.time()
...         return self
...     def __exit__(self, type, value, traceback):
...         t2 = time.time()
...         print(f"{(t2-self.t1):.4f}")
...
>>>
>>> with TimeContextManager():
...     snooze()
...
1.0019
```

The `__exit()__` method takes three extra arguments that we don't use here; we could have used `*args` in their place.

Okay, we've seen many ways to do timing. Now, let's time some code to compare the efficiency of different algorithms (program logic) and data structures (storage mechanisms).

Algorithms and Data Structures

The Zen of Python declares that *There should be one—and preferably only one—obvious way to do it*. Unfortunately, sometimes it isn't obvious, and you need to compare alternatives. For example, is it better to use a `for` loop or a list comprehension to build a list? And what do we mean by *better*? Is it faster, easier to understand, using less memory, or more "Pythonic"?

In this next exercise, we build a list in different ways, comparing speed, readability, and Python style. Here's *time_lists.py* (Example 19-24).

Example 19-24. time_lists.py

```
from timeit import timeit

def make_list_1():
    result = []
    for value in range(1000):
        result.append(value)
    return result

def make_list_2():
    result = [value for value in range(1000)]
    return result

print('make_list_1 takes', timeit(make_list_1, number=1000), 'seconds')
print('make_list_2 takes', timeit(make_list_2, number=1000), 'seconds')
```

In each function, we add 1,000 items to a list, and we call each function 1,000 times. Notice that in this test we called `timeit()` with the function name as the first argument rather than code as a string. Let's run it:

```
$ python time_lists.py
```

```
make_list_1 takes 0.14117428699682932 seconds
make_list_2 takes 0.06174145900149597 seconds
```

The list comprehension is at least twice as fast as adding items to the list by using `append()`. In general, comprehensions are faster than manual construction.

Use these ideas to make your own code faster.

Cython, NumPy, and C Extensions

If you're pushing Python as hard as you can and still can't get the performance you want, you have yet more options.

Cython is a hybrid of Python and C, designed to translate Python with some performance annotations to compiled C code. These annotations are fairly small, like declaring the types of some variables, function arguments, or function returns. For scientific-style loops of numeric calculations, adding these hints will make them much faster—as much as a thousand times faster. See the Cython wiki for documentation and examples.

You can read much more about NumPy in Chapter 22. It's a Python math library, written in C for speed.

Many parts of Python and its standard library are written in C for speed and wrapped in Python for convenience. These hooks are available to you for your applications. If you know C and Python and really want to make your code fly, writing a C extension is harder, but the improvements can be worth the trouble.

PyPy

When Java first appeared about 20 years ago, it was as slow as an arthritic Schnauzer. When it started to mean real money to Sun and other companies, though, they put millions into optimizing the Java interpreter and the underlying Java virtual machine (JVM), borrowing techniques from earlier languages like Smalltalk and LISP. Microsoft likewise put great effort into optimizing its rival C# language and .NET VM.

No one owns Python, so no one has pushed that hard to make it faster.

You're probably using the standard Python implementation. It's written in C, and often called CPython (not the same as Cython).

Like PHP, Perl, and even Java, Python is not compiled to machine language, but translated to an intermediate language (with names such as *bytecode* or p-code) which is then interpreted in a *virtual machine*.

PyPy is a new Python interpreter that applies some of the tricks that sped up Java. Its benchmarks show that PyPy is faster than CPython in every test—more than six times faster on average, and up to 20 times faster in some cases. It works with Python 2 and 3. You can download it and use it instead of CPython. PyPy is constantly being improved, and it might even replace CPython some day. Read the latest release notes on the site to see whether it could work for your purposes.

Numba

You can use Numba to compile your Python code on the fly to machine code and speed it up.

Install with the usual:

```
$ pip install numba
```

Let's first time a normal Python function that calculates a hypotenuse:

```
>>> import math
>>> from timeit import timeit
>>> from numba import jit
>>>
>>> def hypot(a, b):
...     return math.sqrt(a**2 + b**2)
...
>>> timeit('hypot(5, 6)', globals=globals())
0.6349189280000189
>>> timeit('hypot(5, 6)', globals=globals())
0.6348589239999853
```

Use the `@jit` decorator to speed up calls after the first:

```
>>> @jit
... def hypot_jit(a, b):
...     return math.sqrt(a**2 + b**2)
...
>>> timeit('hypot_jit(5, 6)', globals=globals())
0.5396156099999985
>>> timeit('hypot_jit(5, 6)', globals=globals())
0.1534771130000081
```

Use `@jit(nopython=True)` to avoid the overhead of the normal Python interpreter:

```
>>> @jit(nopython=True)
... def hypot_jit_nopy(a, b):
...     return math.sqrt(a**2 + b**2)
...
>>> timeit('hypot_jit_nopy(5, 6)', globals=globals())
0.18343535700000757
>>> timeit('hypot_jit_nopy(5, 6)', globals=globals())
0.15387067300002855
```

Numba is especially useful with NumPy and other mathematically demanding packages.

Source Control

When you're working on a small group of programs, you can usually keep track of your changes—until you make a boneheaded mistake and clobber a few days of work. Source control systems help protect your code from dangerous forces, like you. If you work with a group of developers, source control becomes a necessity. There are many commercial and open source packages in this area. The most popular in the open source world where Python lives are Mercurial and Git. Both are examples of *distributed* version control systems, which produce multiple copies of code repositories. Earlier systems such as Subversion run on a single server.

Mercurial

Mercurial is written in Python. It's fairly easy to learn, with a handful of subcommands to download code from a Mercurial repository, add files, check in changes, and merge changes from different sources. bitbucket and other sites offer free or commercial hosting.

Git

Git was originally written for Linux kernel development, but now dominates open source in general. It's similar to Mercurial, although some find it slightly trickier to master. GitHub is the largest git host, with more than a million repositories, but there are many other hosts.

The standalone program examples in this book are available in a public Git repository at GitHub. If you have the `git` program on your computer, you can download these programs by using this command:

```
$ git clone https://github.com/madscheme/introducing-python
```

You can also download the code from the GitHub page:

- Click "Clone in Desktop" to open your computer's version of `git`, if installed.
- Click "Download ZIP" to get a zipped archive of the programs.

If you don't have `git` but would like to try it, read the installation guide. I talk about the command-line version here, but you might be interested in sites such as GitHub that have extra services and might be easier to use in some cases; `git` has many features, but is not always intuitive.

Let's take `git` for a test drive. We won't go far, but the ride will show a few commands and their output.

Make a new directory and change to it:

```
$ mkdir newdir
$ cd newdir
```

Create a local Git repository in your current directory *newdir*:

```
$ git init
```

```
Initialized empty Git repository in /Users/williamlubanovic/newdir/.git/
```

Create a Python file called *test.py*, shown in Example 19-25, with these contents in *newdir*.

Example 19-25. test.py

```
print('Oops')
```

Add the file to the Git repository:

```
$ git add test.py
```

What do you think of that, Mr. Git?

```
$ git status
```

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   test.py
```

This means that *test.py* is part of the local repository but its changes have not yet been committed. Let's *commit* it:

```
$ git commit -m "simple print program"
```

```
 [master (root-commit) 52d60d7] my first commit
  1 file changed, 1 insertion(+)
  create mode 100644 test.py
```

That `-m "my first commit"` was your *commit message*. If you omitted that, `git` would pop you into an editor and coax you to enter the message that way. This becomes a part of the `git` change history for that file.

Let's see what our current status is:

```
$ git status
```

```
 On branch master
 nothing to commit, working directory clean
```

Okay, all current changes have been committed. This means that we can change things and not worry about losing the original version. Make an adjustment now to *test.py*—change `Oops` to `Ops!` and save the file (Example 19-26).

Example 19-26. test.py, revised

```
print('Ops!')
```

Let's check to see what `git` thinks now:

```
$ git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
	modified:   test.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Use `git diff` to see what lines have changed since the last commit:

```
$ git diff
```

```
diff --git a/test.py b/test.py
index 76b8c39..62782b2 100644
--- a/test.py
+++ b/test.py
@@ -1 +1 @@
-print('Oops')
+print('Ops!')
```

If you try to commit this change now, `git` complains:

```
$ git commit -m "change the print string"
```

```
On branch master
Changes not staged for commit:
	modified:   test.py

no changes added to commit
```

That `staged for commit` phrase means you need to `add` the file, which roughly translated means *hey git, look over here*:

```
$ git add test.py
```

You could have also typed `git add .` to add *all* changed files in the current directory; that's handy when you actually have edited multiple files and want to ensure that you check in all their changes. Now we can commit the change:

```
$ git commit -m "my first change"
```

```
[master e1e11ec] my first change
 1 file changed, 1 insertion(&plus;), 1 deletion(-)
```

If you'd like to see all the terrible things that you've done to *test.py*, most recent first, use `git log`:

```
$ git log test.py
```

```
commit e1e11ecf802ae1a78debe6193c552dcd15ca160a
Author: William Lubanovic <bill@madscheme.com>
Date:   Tue May 13 23:34:59 2014 -0500

    change the print string

commit 52d60d76594a62299f6fd561b2446c8b1227cfe1
Author: William Lubanovic <bill@madscheme.com>
Date:   Tue May 13 23:26:14 2014 -0500

    simple print program
```

Distribute Your Programs

You know that your Python files can be installed in files and directories, and you know that you can run a Python program file with the `python` interpreter.

It's less well known that the Python interpreter can also execute Python code packaged in ZIP files. It's even less well known that special ZIP files known as pex files can also be used.

Clone This Book

You can get a copy of all the programs in this book. Visit the Git repository and follow the directions to copy it to your local machine. If you have `git`, run the command `git clone https://github.com/madscheme/introducing-python` to make a Git repository on your computer. You can also download the files in ZIP format.

How You Can Learn More

This is an introduction. It almost certainly says too much about some things that you don't care about and not enough about some things that you do. Let me recommend some Python resources that I've found helpful.

Books

I've found the books in the following list to be especially useful. These range from introductory to advanced, with mixtures of Python 2 and 3:

- Barry, Paul. *Head First Python (2nd Edition)* O'Reilly, 2016.
- Beazley, David M. *Python Essential Reference (5th Edition)*. Addison-Wesley, 2019.
- Beazley, David M. and Brian K. Jones. *Python Cookbook (3rd Edition)*. O'Reilly, 2013.
- Gorelick, Micha and Ian Ozsvald. *High Performance Python*. O'Reilly, 2014.
- Maxwell, Aaron. *Powerful Python*. Powerful Python Press, 2017.
- McKinney, Wes. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly, 2012.
- Ramalho, Luciano. *Fluent Python*. O'Reilly, 2015.
- Reitz, Kenneth and Tanya Schlusser. *The Hitchhiker's Guide to Python*. O'Reilly, 2016.
- Slatkin, Brett. *Effective Python*. Addison-Wesley, 2015.
- Summerfield, Mark. *Python in Practice: Create Better Programs Using Concurrency, Libraries, and Patterns*. Addison-Wesley, 2013.

Of course, there are many more.

Websites

Here are some websites where you can find helpful tutorials:

- Python for You and Me is an introduction, with good Windows coverage
- Real Python by various authors
- Learn Python the Hard Way by Zed Shaw
- Dive into Python 3 by Mark Pilgrim
- Mouse Vs. Python by Michael Driscoll

If you're interested in keeping up with what's going on in the Pythonic world, check out these news websites:

- comp.lang.python
- comp.lang.python.announce
- r/python subreddit
- Planet Python

Finally, here are some good websites for finding and downloading packages:

- The Python Package Index
- Awesome Python
- Stack Overflow Python questions
- ActiveState Python recipes
- Python packages trending on GitHub

Groups

Computing communities have varied personalities: enthusiastic, argumentative, dull, hipster, button-down, and many others across a broad range. The Python community is friendly and civil. You can find Python groups based on location—meetups and local user groups around the world. Other groups are distributed and based on common interests. For instance, PyLadies is a support network for women who are interested in Python and open source.

Conferences

Of the many conferences and workshops around the world, the largest are held annually in North America and Europe.

Getting a Python Job

Useful search sites include:

- Indeed
- Stack Overflow
- ZipRecruiter
- Simply Hired
- CareerBuilder
- Google
- LinkedIn

For most of these sites, type `python` in the first box and your location in the other. Good local sites include the Craigslist ones, like this link for Seattle. Simply change the `seattle` part to `sfbay`, `boston`, `nyc`, or other craigslist site prefixes to search those areas. For remote (telecommuting, or "work from home") Python jobs, special sites include:

- Indeed
- Google
- LinkedIn
- Stack Overflow
- Remote Python
- We Work Remotely
- ZipRecruiter
- Glassdoor
- Remotely Awesome Jobs
- Working Nomads
- GitHub

Coming Up

But wait, there’s more! The next three chapters offer tours of Python in the arts, business, and science. You’ll find at least one package that you’ll want to explore. Bright and shiny objects abound on the net. Only you can tell which are costume jewelry and which are silver bullets. And even if you’re not currently pestered by werewolves, you might want some of those silver bullets in your pocket. Just in case.

Finally, we have answers to those annoying end-of-chapter exercises, details on installation of Python and friends, and a few cheat sheets for things that I always need to look up. Your brain is almost certainly better tuned, but they’re there if needed.

Things to Do

(Pythonistas don’t have homework today.)

[1] You, as the detective: “I know I’m in there! And if I don’t come out with my hands up, I’m coming in after me!”

[2] Many computer books use Fibonacci number calculations in timing examples, but I’d rather sleep than calculate Fibonacci numbers.