O'REILLY®

20 Python Libraries You Aren't Using (But Should)

Caleb Hattingh

O'REILLY®

Additional Resources

4 Easy Ways to Learn More and Stay Current

Programming Newsletter
Get programming related news and content delivered weekly to your inbox.
oreilly.com/programming/newsletter

Free Webcast Series
Learn about popular programming topics from experts live, online.
webcasts.oreilly.com

O'Reilly Radar
Read more insight and analysis about emerging technologies.
radar.oreilly.com

Conferences
Immerse yourself in learning at an upcoming O'Reilly conference.
conferences.oreilly.com

20 Python Libraries You Aren't Using (But Should)

Caleb Hattingh

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

20 Python Libraries You Aren't Using (But Should)
by Caleb Hattingh

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Dawn Schanafelt
Production Editor: Colleen Lobner
Copyeditor: Christina Edwards
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

August 2016: First Edition

Revision History for the First Edition
2016-08-08: First Release

978-1-491-96792-8

[LSI]

Table of Contents

CHAPTER 1

Expanding Your Python Knowledge: Lesser-Known Libraries

The Python ecosystem is vast and far-reaching in both scope and depth. Starting out in this crazy, open-source forest is daunting, and even with years of experience, it still requires continual effort to keep up-to-date with the best libraries and techniques.

In this report we take a look at some of the lesser-known Python libraries and tools. Python itself already includes a huge number of high-quality libraries; collectively these are called the *standard library*. The standard library receives a lot of attention, but there are still some libraries within it that should be better known. We will start out by discussing several, extremely useful tools in the standard library that you may not know about.

We're also going to discuss several exciting, lesser-known libraries from the third-party ecosystem. Many high-quality third-party libraries are already well-known, including Numpy and Scipy, Django, Flask, and Requests; you can easily learn more about these libraries by searching for information online. Rather than focusing on those standouts, this report is instead going to focus on several interesting libraries that are growing in popularity.

Let's start by taking a look at the standard library.

The Standard Library

The libraries that tend to get all the attention are the ones heavily used for operating-system interaction, like `sys`, `os`, `shutil`, and to a slightly lesser extent, `glob`. This is understandable because most Python applications deal with input processing; however, the Python standard library is very rich and includes a bunch of additional functionality that many Python programmers take too long to discover. In this chapter we will mention a few libraries that every Python programmer should know very well.

collections

First up we have the `collections` module. If you've been working with Python for any length of time, it is very likely that you have made use of the this module; however, the batteries contained within are so important that we'll go over them anyway, *just in case*.

collections.OrderedDict

`collections.OrderedDict` gives you a `dict` that will preserve the order in which items are added to it; note that this is *not* the same as a sorted order.[1]

The need for an *ordered* `dict` comes up surprisingly often. A common example is processing lines in a file where the lines (or something within them) maps to other data. A mapping is the right solution, and you often need to produce results in the same order in which the input data appeared. Here is a simple example of how the ordering changes with a normal `dict`:

```
>>> dict(zip(ascii_lowercase, range(4)))
{'a': 0, 'b': 1, 'c': 2, 'd': 3}

>>> dict(zip(ascii_lowercase, range(5)))
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4}

>>> dict(zip(ascii_lowercase, range(6)))
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'f': 5, 'e': 4} ❶
```

1 Looking for *sorted* container types? The excellent sorted containers package has high-performance sorted versions of the `list`, `dict`, and `set` datatypes.

```
>>> dict(zip(ascii_lowercase, range(7)))
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'g': 6, 'f': 5, 'e': 4}
```

❶ See how the key "f" now appears before the "e" key in the sequence of keys? They no longer appear in the order of insertion, due to how the `dict` internals manage the assignment of hash entries.

The `OrderedDict`, however, retains the order in which items are inserted:

```
>>> from collections import OrderedDict

>>> OrderedDict(zip(ascii_lowercase, range(5)))
OrderedDict([('a', 0), ('b', 1), ('c', 2), ('d', 3),
('e', 4)])

>>> OrderedDict(zip(ascii_lowercase, range(6)))
OrderedDict([('a', 0), ('b', 1), ('c', 2), ('d', 3),
('e', 4), ('f', 5)])

>>> OrderedDict(zip(ascii_lowercase, range(7)))
OrderedDict([('a', 0), ('b', 1), ('c', 2), ('d', 3),
('e', 4), ('f', 5), ('g', 6)])
```

OrderedDict: Beware creation with keyword arguments

There is an unfortunate catch with `OrderedDict` you need to be aware of: it doesn't work when you *create* the `OrderedDict` with keyword arguments, a very common Python idiom:

```
>>> collections.OrderedDict(a=1,b=2,c=3)
OrderedDict([('b', 2), ('a', 1), ('c', 3)])
```

This seems like a bug, but as explained in the documentation, it happens because the keyword arguments are first processed as a *normal* `dict` before they are passed on to the `OrderedDict`.

collections.defaultdict

`collections.defaultdict` is another special-case dictionary: it allows you to specify a default value for all new keys.

Here's a common example:

```
>>> d = collections.defaultdict(list)
>>> d['a']  ❶
[]
```

❶ You didn't create this item yet? No problem! Key lookups automatically create *values* using the function provided when creating the `defaultdict` instance.

By setting up the default value as the `list` constructor in the preceding example, you can avoid wordy code that looks like this:

```
d = {}
for k in keydata:
    if not k in d:
            d[k] = []
    d[k].append(...)
```

The `setdefault()` method of a `dict` can be used in a somewhat similar way to initialize items with defaults, but `defaultdict` generally results in clearer code.[2]

In the preceding examples, we're saying that every new element, by default, will be an empty list. If, instead, you wanted every new element to contain a dictionary, you might say `defaultdict(dict)`.

collections.namedtuple

The next tool, `collections.namedtuple`, is magic in a bottle! Instead of working with this:

```
tup = (1, True, "red")
```

You get to work with this:

```
>>> from collections import namedtuple
>>> A = namedtuple('A', 'count enabled color')
>>> tup = A(count=1, enabled=True, color="red")
>>> tup.count
1
>>> tup.enabled
True
```

2 For instance, this example with `setdefault()` looks like `d.setdefault(k, []).append(...)`. The default value is always evaluated, whereas with `defaultdict` the default value generator is only evaluated when necessary. But there are still cases where you'll need `setdefault()`, such as when using different default values depending on the key.

```
>>> tup.color
"red"
>>> tup
A(count=1, enabled=True, color='red')
```

The best thing about `namedtuple` is that you can add it to existing code and use it to progressively replace tuples: it can appear anywhere a tuple is currently being used, without breaking existing code, and without using any extra resources beyond what plain tuples require. Using `namedtuple` incurs no extra runtime cost, and can make code much easier to read. The most common situation where a `namedtuple` is recommended is when a function returns multiple results, which are then unpacked into a tuple. Let's look at an example of code that uses plain tuples, to see why such code can be problematic:

```
>>> def f():
...     return 2, False, "blue"  ❶
>>> count, enabled, color = f()  ❷

>>> tup = f()  ❸
>>> enabled = tup[1]
```

❶ Simple function returning a tuple.

❷ When the function is evaluated, the results are unpacked into separate names.

❸ Worse, the caller might access values inside the returned tuple *by index*.

The problem with this approach is that this code is fragile to future changes. If the function changes (perhaps by changing the order of the returned items, or adding more items), the unpacking of the returned value will be incorrect. Instead, you can modify *existing* code to return a `namedtuple` instance:

```
>>> def f():
...     # Return a namedtuple!
...     return A(2, False, "blue")

>>> count, enabled, color = f()  ❶
```

❶ Even though our function now returns a `namedtuple`, the same calling code stills works.

You now also have the option of working with the returned `namedtuple` in the calling code:

```
>>> tup = f()
>>> print(tup.count)  ❶
2
```

❶ Being able to use *attributes* to access data inside the tuple is much safer rather than relying on indexing alone; if future changes in the code added new fields to the `namedtuple`, the `tup.count` would continue to work.

The `collections` module has a few other tricks up its sleeve, and your time is well spent brushing up on the documentation. In addition to the classes shown here, there is also a `Counter` class for easily counting occurrences, a *list-like* container for efficiently appending and removing items from either end (`deque`), and several helper classes to make subclassing lists, dicts, and strings easier.

contextlib

A context manager is what you use with the `with` statement. A very common idiom in Python for working with file data demonstrates the context manager:

```
with open('data.txt', 'r') as f:
    data = f.read()
```

This is good syntax because it simplifies the *cleanup* step where the file handle is closed. Using the context manager means that you don't have to remember to do `f.close()` yourself: this will happen automatically when the `with` block exits.

You can use the `contextmanager` decorator from the `contextlib` library to benefit from this language feature in your own nefarious schemes. Here's a creative demonstration where we create a new context manager to print out performance (timing) data.

This might be useful for quickly testing the time cost of code snippets, as shown in the following example. The numbered notes are intentionally not in numerical order in the code. Follow the notes in numerical order as shown following the code snippet.

```
from time import perf_counter
from array import array
from contextlib import contextmanager
```

```
@contextmanager ❷
def timing(label: str):
    t0 = perf_counter() ❸
    yield lambda: (label, t1 - t0) ❹
    t1 = perf_counter() ❺

with timing('Array tests') as total: ❼
    with timing('Array creation innermul') as inner:
        x = array('d', [0] * 1000000) ❶

    with timing('Array creation outermul') as outer:
        x = array('d', [0]) * 1000000 ❻


print('Total [%s]: %.6f s' % total())
print('    Timing [%s]: %.6f s' % inner())
print('    Timing [%s]: %.6f s' % outer())
```

❶ The `array` module in the standard library has an unusual approach to initialization: you pass it an existing sequence, such as a large list, and it converts the data into the datatype of your array if possible; however, you *can* also create an array from a short sequence, after which you expand it to its full size. Have you ever wondered which is faster? In a moment, we'll create a *timing* context manager to measure this and know for sure!

❷ The key step you need to do to make your own context manager is to use the `@contextmanager` decorator.

❸ The section before the `yield` is where you can write code that must execute before the *body* of your context manager will run. Here we record the timestamp *before* the body will run.

❹ The `yield` is where execution is transferred to the body of your context manager; in our case, this is where our arrays get created. You can also return data: here I return a closure that will calculate the elapsed time when called. It's a little *clever* but hopefully not excessively so: the *final time* `t1` is captured within the closure even though it will only be determined on the next line.

❺ After the `yield`, we write the code that will be executed when the context manager finishes. For scenarios like file handling, this would be where you close them. In this example, this is where we record the final time `t1`.

❻ Here we try the alternative array-creation strategy: first, create the array and then increase size.

❼ For fun, we'll use our awesome, new context manager to also measure the total time.

On my computer, this code produces this output:

```
Total [Array tests]: 0.064896 s
    Timing [Array creation innermul]: 0.064195 s
    Timing [Array creation outermul]: 0.000659 s
```

Quite surprisingly, the *second* method of producing a large array is around 100 times faster than the first. This means that it is *much* more efficient to create a small array, and then expand it, rather than to create an array entirely from a large list.

The point of this example is not to show the best way to create an array: rather, it is that the `contextmanager` decorator makes it exceptionally easy to create your own context manager, and context managers are a great way of providing a clean and safe means of managing before-and-after coding tasks.

concurrent.futures

The `concurrent.futures` module that was introduced in Python 3 provides a convenient way to manage pools of workers. If you have previously used the threading module in the Python standard library, you will have seen code like this before:

```
import threading

def work():
    return sum(x for x in range(1000000))

thread = threading.Thread(target=work)
thread.start()
thread.join()
```

This code is very clean with only one thread, but with many threads it can become quite tricky to deal with sharing work between them. Also, in this example the result of the `sum` is not obtained from the work function, simply to avoid all the extra code that would be required to do so. There are various techniques for obtaining the result of a work function, such as passing a queue to the function, or subclassing `threading.Thread`, but we're not going discuss them

any further, because the multiprocessing package provides a better method for using pools, and the `concurrent.futures` module goes even further to simplify the interface. And, similar to multiprocessing, both thread-based pools and process-based pools have the same interface making it easy to switch between either thread-based or process-based approaches.

Here we have a trivial example using the `ThreadPoolExecutor`. We download the landing page of a plethora of popular social media sites, and, to keep the example simple, we print out the size of each. Note that in the results, we show only the first four to keep the output short.

```
from concurrent.futures import ThreadPoolExecutor as Executor

urls = """google twitter facebook youtube pinterest tumblr
instagram reddit flickr meetup classmates microsoft apple
linkedin xing renren disqus snapchat twoo whatsapp""".split()

def fetch(url):  ❶
    from urllib import request, error  ❷
    try:
        data = request.urlopen(url).read()
        return '{}: length {}'.format(url, len(data))
    except error.HTTPError as e:
        return '{}: {}'.format(url, e)

with Executor(max_workers=4) as exe:  ❸
    template = 'http://www.{}.com'
    jobs = [exe.submit(
        fetch, template.format(u)) for u in urls]  ❹
    results = [job.result() for job in jobs]  ❺

print('\n'.join(results))
```

❶ Our work function, `fetch()`, simply downloads the given URL.

❷ Yes, it *is* rather odd nowadays to see `urllib` because the fantastic third-party library requests is a great choice for all your web-access needs. However, `urllib` still exists and depending on your needs, may allow you to avoid an external dependency.

❸ We create a `ThreadPoolExecutor` instance, and here you can specify how many workers are required.

❹ *Jobs* are created, one for every URL in our considerable list. The *executor* manages the delivery of jobs to the four threads.

❺ This is a simple way of waiting for all the threads to return.

This produces the following output (I've shortened the number of results for brevity):

```
http://www.google.com: length 10560
http://www.twitter.com: length 268924
http://www.facebook.com: length 56667
http://www.youtube.com: length 437754
[snip]
```

Even though one job is created for every URL, we limit the number of active threads to only four using `max_workers` and the results are all captured in the results list as they become available. If you wanted to use processes instead of threads, all that needs to change is the first line, from this:

```
from concurrent.futures import ThreadPoolExecutor as Executor
```

To this:

```
from concurrent.futures import ProcessPoolExecutor as Executor
```

Of course, for this kind of application, which is limited by network latency, a thread-based pool is fine. It is with CPU-bound tasks that Python threads are problematic because of how thread safety has been in implemented inside the CPython[3] runtime interpreter, and in these situations it is best to use a process-based pool instead.

The primary problem with using processes for parallelism is that each process is confined to its own memory space, which makes it difficult for multiple workers to chew on the same large chunk of data. There are ways to get around this, but in such situations threads provide a much simpler programming model. However, as we shall see in "Cython" on page 59, there is a third-party package called Cython that makes it very easy to circumvent this problem with threads.

3 *CPython* means the specific implementation of the Python language that is written in the C language. There are other implementations of Python, created with various other languages and technologies such as .NET, Java and even subsets of Python itself.

logging

The `logging` module is very well known in the web development community, but is far less used in other domains, such as the scientific one; this is unfortunate, because even for general use, the logging module is far superior to the `print()` function. It doesn't seem that way at first, because the `print()` function is so simple; however, once you initialize `logging`, it can look very similar. For instance, compare these two:

```
print('This is output to the console')

logger.debug('This is output to the console')
```

The huge advantage of the latter is that, with a single change to a setting on the logger instance, you can either show or hide all your debugging messages. This means you no longer have to go through the process of commenting and uncommenting your `print()` statements in order to show or hide them. `logging` also gives you a few different *levels* so that you can adjust the verbosity of output in your programs. Here's an example of different levels:

```
logger.debug('This is for debugging. Very talkative!')
logger.info('This is for normal chatter')
logger.warning('Warnings should almost always be seen.')
logger.error('You definitely want to see all errors!')
logger.critical('Last message before a program crash!')
```

Another really neat trick is that when you use logging, writing messages during exception handling is a whole lot easier. You don't have to deal with `sys.exc_info()` and the `traceback` module merely for printing out the exception message with a traceback. You can do this instead:

```
try:
    1/0
except:
    logger.exception("Something failed:")
```

Just those four lines produces a full traceback in the output:

```
ERROR:root:Something failed:
Traceback (most recent call last):
  File "logtb.py", line 5, in <module>
    1/0
ZeroDivisionError: division by zero
```

Earlier I said that logging requires some setup. The documentation for the `logging` module is extensive and might seem overwhelming; here is a quick recipe to get you started:

```
# Top of the file
import logging
logger = logging.getLogger()  ❶

# All your normal code goes here
def blah():
    return 'blah'

# Bottom of the file
if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)  ❷
```

❶ Without arguments, the `getLogger()` function returns the *root* logger, but it's more common to create a *named* logger. If you're just starting out, exploring with the root logger is fine.

❷ You need to call a `config` method; otherwise, calls to the logger will not log anything. The config step is necessary.

In the preceding `basicConfig()` line, by changing only `logging.DEBUG` to, say, `logging.WARNING`, you can affect which messages get processed and which don't.

Finally, there's a neat trick you can use for easily changing the logging level on the command line. Python scripts that are directly executable usually have the startup code in a conditional block beginning with `if __name__ == '__main__'`. This is where *command-line parameters* are handled, e.g., using the `argparse` library in the Python standard library. We can create command-line arguments specifically for the logging level:

```
# Bottom of the file
if __name__ == '__main__':
    from argparse import ArgumentParser
    parser = ArgumentParser(description='My app which is mine')
    parser.add_argument('-ll', '--loglevel',
        type=str,
        choices=['DEBUG','INFO','WARNING','ERROR','CRITICAL'],
        help='Set the logging level')
    args = parser.parseargs()
    logging.basicConfig(level=args.loglevel)
```

With this setup, if you call your program with

```
$ python main.py -ll DEBUG
```

it will run with the logging level set to `DEBUG` (so all logger messages will be shown), whereas if you run it with

```
$ python main.py -ll WARNING
```

it will run at the `WARNING` level for logging, so all `INFO` and `DEBUG` logger messages will be hidden.

There are many more features packed into the `logging` module, but I hope I've convinced you to consider using it instead of using `print()` for your next program. There is much more information about the `logging` module online, both in the official Python documentation and elsewhere in blogs and tutorials. The goal here is only to convince you that the `logging` module is worth investigating, and getting started with it is easy to do.

sched

There is increasing interest in the creation of bots[4] and other monitoring and automation applications. For these applications, a common requirement is to perform actions at specified times or specified intervals. This functionality is provided by the `sched` module in the standard library. There are already similar tools provided by operating systems, such as `cron` on Linux and *Windows Task Scheduler*, but with Python's own `sched` module you can ignore these platform differences, as well as incorporate scheduled tasks into a program that might have many other functions.

The documentation for sched is rather terse, but hopefully these examples will get you started. The easy way to begin is to schedule a function to be executed after a specified delay (this is a complete example to make sure you can run it successfully):

```
import sched
import time
from datetime import datetime, timedelta

scheduler = sched.scheduler(timefunc=time.time)  ❶

def saytime():  ❷
```

4 Programs that automate some task, often communicating data across different network services like Twitter, IRC, and Slack.

```
    print(time.ctime())
    scheduler.enter(10, priority=0, action=saytime)  ❸

saytime()
try:
    scheduler.run(blocking=True)  ❹
except KeyboardInterrupt:
    print('Stopped.')
```

❶ A scheduler instance is created.

❷ A work function is declared, in our case `saytime()`, which simply prints out the current time.

❸ Note that we reschedule the function inside itself, with a ten-second delay.

❹ The scheduler is started with `run(blocking=True)`, and the execution point remains here until the program is terminated or Ctrl-C is pressed.

There are a few annoying details about using `sched`: you have to pass `timefunc=time.time` as this isn't set by default, and you have to supply a priority even when not required. However, overall, the `sched` module still provides a clean way to get `cron`-like behavior.

Working with delays can be frustrating if what you *really* want is for a task to execute at specific times. In addition to `enter()`, a `sched` instance also provides the `enterabs()` method with which you can trigger an event at a specific time. We can use that method to trigger a function, say, every *whole* minute:

```
import sched
import time
from datetime import datetime, timedelta

scheduler = sched.scheduler(timefunc=time.time)  ❶

def reschedule():
    new_target = datetime.now().replace(
        second=0, microsecond=0)  ❷
    new_target += timedelta(minutes=1)  ❸
    scheduler.enterabs(
        new_target.timestamp(), priority=0, action=saytime)  ❹

def saytime():
    print(time.ctime(), flush=True)
```

```
        reschedule()

    reschedule()
    try:
        scheduler.run(blocking=True)
    except KeyboardInterrupt:
        print('Stopped.')
```

❶ Create a `scheduler` instance, as before.

❷ Get *current time*, but strip off seconds and microseconds to obtain a whole minute.

❸ The target time is exactly one minute ahead of the current whole minute.

❹ The `enterabs()` method schedules the task.

This code produces the following output:

```
Sat Jun 18 18:14:00 2016
Sat Jun 18 18:15:00 2016
Sat Jun 18 18:16:00 2016
Sat Jun 18 18:17:00 2016
Stopped.
```

With the growing interest in "Internet of Things" applications, the built-in `sched` library provides a convenient way to manage repetitive tasks. The documentation provides further information about how to cancel future tasks.

In the Wild

From here on we'll look at some third-party Python libraries that you might not yet have discovered. There are thousands of excellent packages described at the Python guide.

There are quite a few guides similar to this report that you can find online. There could be as many "favorite Python libraries" lists as there are Python developers, and so the selection presented here is necessarily subjective. I spent a lot of time finding and testing various third-party libraries in order to find these hidden gems.

My selection criteria were that a library should be:

- easy to use
- easy to install
- cross-platform
- applicable to more than one domain
- not yet super-popular, but likely to become so
- the *X* factor

The last two items in that list bear further explanation.

Popularity is difficult to define exactly, since different libraries tend to get used to varying degrees within different Python communities. For instance, Numpy and Scipy are much more heavily used within the *scientific* Python community, while Django and Flask enjoy more attention in the web development community. Furthermore, the popularity of these libraries is such that everybody already knows about them. A candidate for this list *might* have been something like Dask, which seems poised to become an eventual successor to Numpy, but in the medium term it is likely to be mostly applicable to the scientific community, thus failing my applicability test.

The *X* factor means that really cool things are likely to be built with that Python library. Such a criterion is of course strongly subjective, but I hope, in making these selections, to inspire you to experiment and create something new!

Each of the following chapters describes a library that met all the criteria on my list, and which I think you'll find useful in your everyday Python activities, no matter your specialization.

Easier Python Packaging with flit

flit is a tool that dramatically simplifies the process of submitting a Python package to the Python Package Index (PyPI). The traditional process begins by creating a `setup.py` file; simply figuring out how to do that requires a considerable amount of work even to understand what to do. In contrast, flit will create its config file interactively, and for typical simple packages you'll be ready to upload to PyPI almost immediately. Let's have a look: consider this simple package structure:

```
$ tree
.
└── mypkg
    ├── __init__.py  ❶
    └── main.py
```

❶ The package's *init* file is a great place to add package information like documentation, version numbers, and author information.

After installing flit into your environment with `pip install flit`, you can run the interactive initializer, which will create your package configuration. It asks only five questions, most of which will have applicable defaults once you have made your first package with `flit`:

```
$ flit init
Module name [mypkg]:
Author [Caleb Hattingh]:
Author email [caleb.hattingh@gmail.com]:
Home page [https://github.com/cjrh/mypkg]:
Choose a license
 1. MIT
 2. Apache
 3. GPL
 4. Skip - choose a license later
Enter 1-4 [2]: 2
Written flit.ini; edit that file to add optional extra info.
```

The final line tells you that a *flit.ini* file was created. Let's have a look at that:

```
$ cat flit.ini
// [metadata]
module = mypkg
author = Caleb Hattingh
author-email = caleb.hattingh@gmail.com
home-page = https://github.com/cjrh/mypkg
classifiers = License :: OSI Approved :: Apache Software License
```

It's pretty much what we specified in the interactive `flit init` sequence. Before you can submit our package to the online PyPI, there are two more steps that you must complete. The first is to give your package a docstring. You add this to the *mypkg/__init__.py* file at the top using triple quotes (`"""`). The second is that you must add a line for the version to the same file. Your finished *__init__.py* file might look like this:

```
# file: __init__.py
""" This is the documentation for the package. """ ❶

__version__ = '1.0.0'  ❷
```

❶ This documentation will be used as your package description on PyPI.

❷ Likewise, the version tag within your package will also be reused for PyPI when the package is uploaded. These automatic integrations help to simplify the packaging process. It might not seem like much is gained, but experience with Python packaging will show that too many steps (even when they're simple), when combined, can lead to a complex packaging experience.[5]

After filling in the basic description and the version, you are ready to build a wheel and upload it to PyPI:

```
$ flit wheel --upload
Copying package file(s) from mypkg
Writing metadata files
Writing the record of files
Wheel built: dist/mypkg-1.0.0-py2.py3-none-any.whl
Using repository at https://pypi.python.org/pypi
Uploading dist/mypkg-1.0.0-py2.py3-none-any.whl...
Starting new HTTPS connection (1): pypi.python.org
Uploading forbidden; trying to register and upload again ❶
Starting new HTTPS connection (1): pypi.python.org
Registered mypkg with PyPI
Uploading dist/mypkg-1.0.0-py2.py3-none-any.whl...
Starting new HTTPS connection (1): pypi.python.org
Package is at https://pypi.python.org/pypi/mypkg
```

❶ Note that flit automatically registers your package if the initial upload fails.

And that's it! Figure 1-1 shows our package on PyPI.

5 The `X.Y.Z` versioning scheme shown here is known as *semantic versioning* ("semver"), but an alternative scheme worth investigating further is *calendar versioning*, which you can learn more about at *calver.org*.

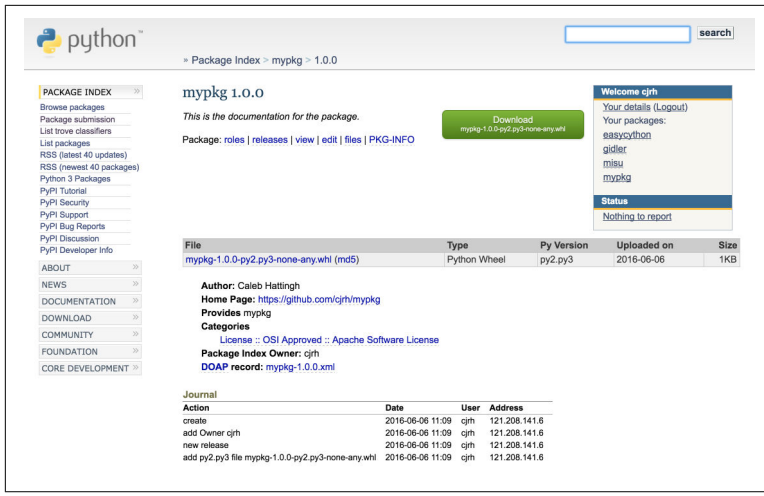

Figure 1-1. It's alive! Our demo package on the PyPI.

`flit` allows you to specify more options, but for simple packages what you see here can get you pretty far.

Command-Line Applications

If you've spent any time whatsoever developing Python code, you will surely have used several command-line applications. Some command-line programs seem much friendlier than others, and in this chapter we show two fantastic libraries that will make it easy for you to offer the best experience for users of your own command-line applications. `colorama` allows you to use colors in your output, while `begins` makes it easy to provide a rich interface for specifying and processing command-line options.

colorama

Many of your desktop Python programs will only ever be used on the command line, so it makes sense to do whatever you can to improve the experience of your users as much as possible. The use of *color* can dramatically improve your user interface, and `colorama` makes it very easy to add splashes of color into your command-line applications.

Let's begin with a simple example.

```
from colorama import init, Fore, Back, Style
init(autoreset=True)

messages = [
    'blah blah blah',
    (Fore.LIGHTYELLOW_EX + Style.BRIGHT
        + BACK.MAGENTA + 'Alert!!!'),
    'blah blah blah'
]

for m in messages:
    print(m)
```

Notice in Figure 1-2 how the important message jumps right out at you? There are other packages like `curses`, `blessings`, and `prompt-toolkit` that let you do a whole lot more with the terminal screen itself, but they also have a slightly steeper learning curve; with `colorama` the API is simple enough to be easy to remember.

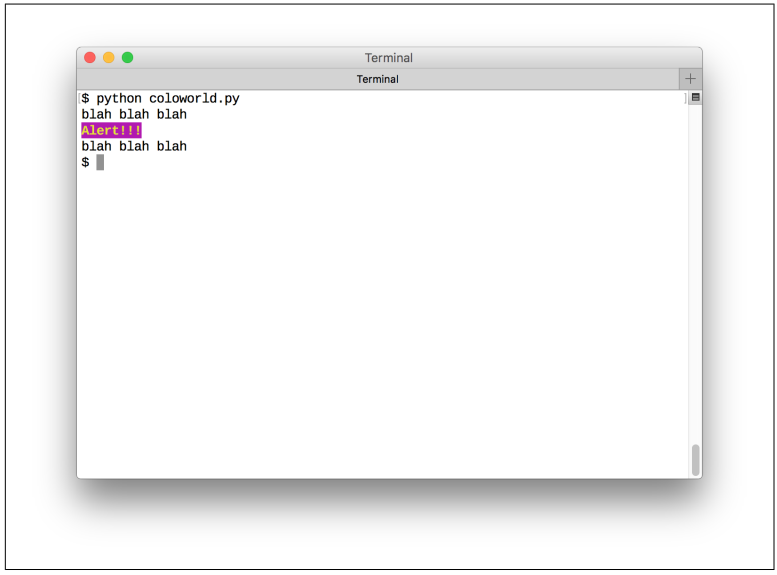

Figure 1-2. Add color to your output messages with simple string concatenation.

The great thing about `colorama` is that it also works on Windows, in addition to Linux and Mac OS X. In the preceding example, we used the `init()` function to enable automatic reset to default colors after each `print()`, but even when not required, `init()` should always be

called (when your code runs on Windows, the `init()` call enables the mapping from ANSI color codes to the Windows color system).

```
from colorama import init
init()
```

The preceding example is clear enough to follow, but I would be a sorry author if I—after having emphasized the benefits of the *logging* module—told you that the only way to get colors into your console was to use `print()`. As usual with Python, it turns out that the hard work has already been done for us. After installing the `colorlog` package,[6] you can use colors in your log messages immediately:

```
import colorlog

logger = colorlog.getLogger()  ❶
logger.setLevel(colorlog.colorlog.logging.DEBUG)  ❷

handler = colorlog.StreamHandler()
handler.setFormatter(colorlog.ColoredFormatter())  ❸
logger.addHandler(handler)

logger.debug("Debug message")
logger.info("Information message")
logger.warning("Warning message")
logger.error("Error message")
logger.critical("Critical message")
```

❶ Obtain a `logger` instance, exactly as you would normally do.

❷ Set the logging level. You can also use the constants like `DEBUG` and `INFO` from the logging module directly.

❸ Set the message formatter to be the `ColoredFormatter` provided by the `colorlog` library.

This produces the output shown in Figure 1-3.

There are several other similar Python packages worth watching, such as the `fabulous` command-line beautifier, which would have made this list if it had been updated for Python 3 just a few weeks sooner!

6 `pip install colorlog`

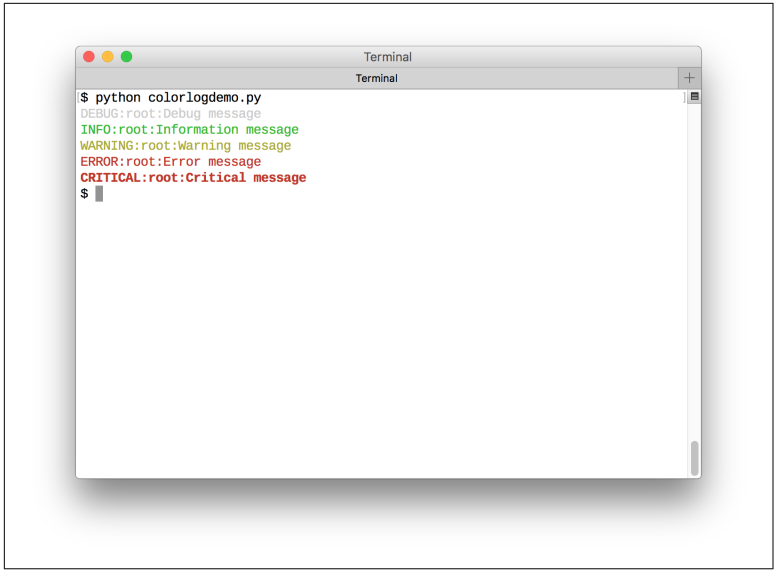

Figure 1-3. Beautiful and automatic colors for your logging messages.

begins

As far as user-interfaces go, most Python programs start out as command-line applications, and many remain so. It makes sense to offer your users the very best experience you can. For such programs, options are specified with command-line arguments, and the Python standard library offers the argparse library to help with that.

`argparse` is a robust, solid implementation for command-line processing, but it is somewhat verbose to use. For example, here we have an extremely simple script that will add two numbers passed on the command line:

```
import argparse

def main(a, b):
    """ Short script to add two numbers """
    return a + b

if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description="Add two numbers")
    parser.add_argument('-a',
                        help='First value',
                        type=float,
                        default=0)
```

```
    parser.add_argument('-b',
                        help='Second value',
                        type=float,
                        default=0)
    args = parser.parse_args()
    print(main(args.a, args.b))
```

As one would expect, help can be obtained by passing `-h` to this program:

```
$ python argparsedemo.py -h
usage: argparsedemo.py [-h] [-a A] [-b B]

Add two numbers

optional arguments:
  -h, --help  show this help message and exit
  -a A        First value
  -b B        Second value
```

In contrast, the `begins` library takes a machete to the API of `argparse` and maximally exploits features of the Python language to simplify setting up the same command-line interface:

```
import begin

@begin.start(auto_convert=True)
def main(a: 'First value' = 0.0, b: 'Second value' = 0.0):
    """ Add two numbers """
    print(a + b)
```

There is so much happening in so few lines, yet everything is still explicit:

- Each parameter in the main function becomes a command-line argument.
- The *function annotations* are exploited to provide an inline help description of each parameter.
- The default value of each parameter (here, `0.0`) is used both as a default value, as well as to indicate the required datatype (in this case, a `float` number value).
- The `auto_convert=True` is used to enforce type coercion from a string to the target parameter type.
- The docstring for the *function* now becomes the help documentation of the program itself.

Also, you may have noticed that this example lacks the usual `if __name__ == '__main__'` boilerplate: that's because `begins` is smart enough to work with Python's stack frames so that your target function becomes the starting point.

For completeness, here is the help for the `begins`-version, produced with `-h`:

```
$ python beginsdemo.py -h
usage: beginsdemo.py [-h] [--a A] [--b B]

Add two numbers

optional arguments:
  -h, --help   show this help message and exit
  --a A, -a A  First value (default: 0.0)
  --b B, -b B  Second value (default: 0.0)
```

There are a bunch more tricks that `begins` makes available, and you are encouraged to read the documentation; for instance, if the parameters are words:

```
@begin.start
def main(directory: 'Target dir'):
    ...
```

then both `-d VALUE` and `--directory VALUE` will work for specifying the value of the `directory` parameter on the command line. A sequence of positional arguments of unknown length is easily defined with unpacking:

```
#demo.py
@begin.start
def main(directory: 'Target dir', *filters):
    ...
```

When called with:

```
$ python demo.py -d /home/user tmp temp TEMP
```

the `filters` argument would be the list `['tmp', 'temp', 'TEMP']`.

If that were all that `begins` supported, it would already be sufficient for the vast majority of simple programs; however, `begins` also provides support for *subcommands*:

```
import begin

@begin.subcommand
def status(compact: 'Short or long format' = True):
    """ Report the current status. """
```

```
    if compact:
        print('ok.')
    else:
        print('Very well, thank-you.')

@begin.subcommand
def fetch(url: 'Data source' = 'http://google.com'):
    """ Fetch data from the URL. """
    print('Work goes here')

@begin.start(auto_convert=True)
def main(a: 'First value' = 0.0, b: 'Second value' = 0.0):
    """ Add two numbers """
    print(a + b)
```

The `main` function is the same as before, but we've added two subcommands:

- The first, `status`, represents a subcommand that could be used to provide some kind of *system status* message (think "git status").
- The second, `fetch`, represents a subcommand for some kind of work function (think "git fetch").

Each subcommand has its own set of parameters, and the rules work in the same way as before with the `main` function. For instance, observe the updated help (obtained with the `-h` parameter) for the program:

```
$ python beginssubdemo.py -h
usage: beginssubdemo.py [-h] [--a A] [--b B] {fetch,status} ...

Add two numbers

optional arguments:
  -h, --help      show this help message and exit
  --a A, -a A     First value (default: 0.0)
  --b B, -b B     Second value (default: 0.0)

Available subcommands:
  {fetch,status}
    fetch         Fetch data from the URL.
    status        Report the current status.
```

We still have the same documentation for the main program, but now additional help for the subcommands have been added. Note that the function docstrings for `status` and `fetch` have also been recycled into CLI help descriptions. Here is an example of how our program might be called:

```
$ python beginssubdemo.py -a 7 -b 7 status --compact
14.0
ok.

$ python beginssubdemo.py -a 7 -b 7 status --no-compact
14.0
Very well, thank-you.
```

You can also see how *boolean* parameters get some special treatment: they evaluate `True` if present and `False` when `no-` is prefixed to the parameter name, as shown for the parameter `compact`.

`begins` has even more tricks up its sleeve, such as automatic handling for environment variables, config files, error handling, and logging, and I once again urge you to check out the project documentation.

If `begins` seems too extreme for you, there are a bunch of other tools for easing the creation of command-line interfaces. One particular option that has been growing in popularity is click.

Graphical User Interfaces

Python offers a wealth of options for creating graphical user interfaces (GUIs) including PyQt, wxPython, and tkinter, which is also available directly in the standard library. In this chapter we will describe two significant, but largely undiscovered, additions to the lineup. The first, `pyqtgraph`, is much more than simply a chart-plotting library, while `pywebview` gives you a full-featured web-technology interface for your desktop Python applications.

pyqtgraph

The most popular chart-plotting library in Python is matplotlib, but you may not yet have heard of the wonderful alternative, pyqtgraph. Pyqtgraph is not a one-to-one replacement for matplotlib; rather, it offers a different selection of features and in particular is excellent for real-time and interactive visualization.

It is very easy to get an idea of what pyqtgraph offers: simply run the built-in examples application:

```
$ pip install pyqtgraph
$ python -m pyqtgraph.examples
```

This will install and run the examples browser. You can see a screenshot of one of the examples in Figure 1-4.

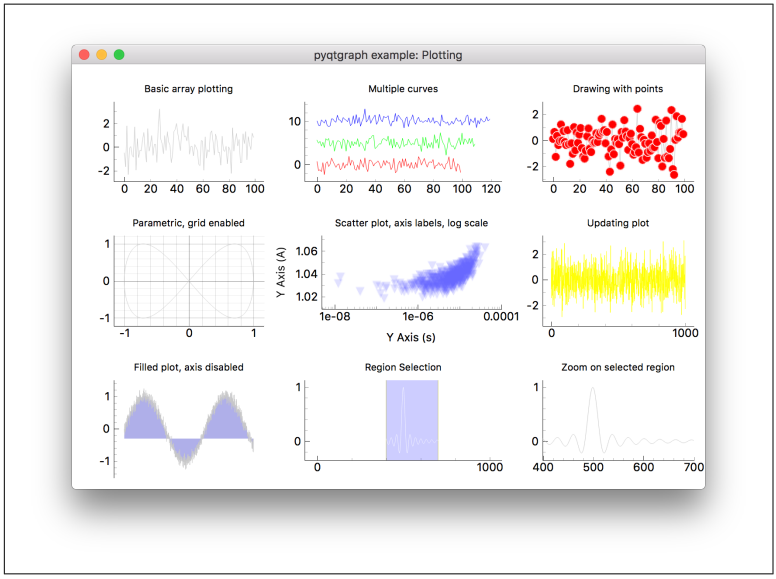

Figure 1-4. Interactive pyqtgraph window.

This example shows an array of interactive plots. You can't tell from the screenshot, but when you run this example, the yellow chart on the right is a high-speed animation. Each plot, including the animated one (in yellow), can be panned and scaled in real-time with your cursor. If you've used Python GUI frameworks in the past, and in particular ones with free-form graphics, you would probably expect sluggish performance. This is not the case with `pyqtgraph`: because `pyqtgraph` uses, unsurprisingly, `PyQt` for the UI. This allows `pyqt`

graph itself to be a pure-python package, making installation and distribution easy.[7]

By default, the style configuration for graphs in pyqtgraph uses a black background with a white foreground. In Figure 1-4, I sneakily used a style configuration change to reverse the colors, since dark backgrounds look terrible in print. This information is also available in the documentation:

```
import pyqtgraph as pg
# Switch to using white background and black foreground
pg.setConfigOption('background', 'w')
pg.setConfigOption('foreground', 'k')
```

There is more on offer than simply plotting graphs: pyqtgraph also has features for 3D visualization, widget docking, and automatic data widgets with two-way binding. In Figure 1-5, the data-entry and manipulation widgets were generated automatically from a Numpy table array, and UI interaction with these widgets changes the graph automatically.

pyqtgraph is typically used as a direct visualizer, much like how matplotlib is used, but with better interactivity. However, it is also quite easy to *embed* pyqtgraph into another separate PyQt application and the documentation for this is easy to follow. pyqtgraph also provides a few useful extras, like edit widgets that are units-aware (kilograms, meters and so on), and tree widgets that can be built automatically from (nested!) standard Python data structures like lists, dictionaries, and arrays.

The author of pyqtgraph is now working with the authors of other visualization packages on a new high-performance visualization package: vispy. Based on how useful pyqtgraph has been for me, I've no doubt that vispy is likely to become another indispensable tool for data visualization.

---

7 Unfortunately, PyQt itself can be either trivial or very tricky to install, depending on your platform. At the time of writing, the stable release of pyqtgraph requires PyQt4 for which no prebuilt installer is available on PyPI; however, the development branch of pyqtgraph works with PyQt5, for which a prebuilt, pip-installable version *does* exist on PyPI. With any luck, by the time you read this a new version of pyqtgraph will have been released!

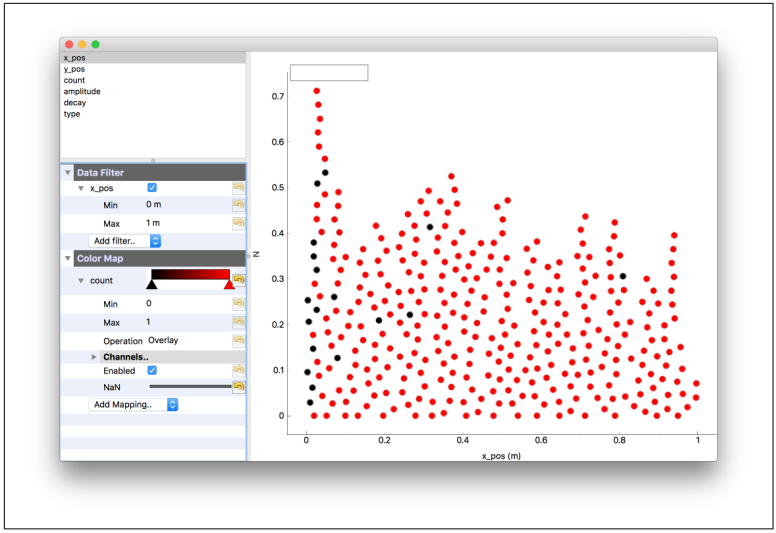

Figure 1-5. Another pyqtgraph example: the data fields and widgets on the left are automatically created from a Numpy table array.

pywebview

There are a huge number of ways to make *desktop* GUI applications with Python, but in recent years the idea of using a browser-like interface as a desktop client interface has become popular. This approach is based around tools like cefpython (using the Chrome-embedded framework) and Electron, which has been used to build many popular tools such as the Atom text editor and Slack social messaging application.

Usually, such an approach involves bundling a browser engine with your application, but Pywebview gives you a one-line command to create a GUI window that wraps a *system native* "web view" window. By combining this with a Python web abb like Flask or Bottle, it becomes very easy to create a local application with a GUI, while also taking advantage of the latest GUI technologies that have been developed in the browser space. The benefit to using the system native browser widget is that you don't have to distribute a potentially large application bundle to your users.

Let's begin with an example. Since our application will be built up as a web-page template in HTML, it might be interesting to use a Python tool to build the HTML rather than writing it out by hand.

```
from string import ascii_letters
from random import choice, randint
import webview ❶
import dominate ❷
from dominate.tags import *
from bottle import route, run, template ❸

bootstrap = 'https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/'
bootswatch = 'https://maxcdn.bootstrapcdn.com/bootswatch/3.3.6/'

doc = dominate.document()

with doc.head:
  link(rel='stylesheet',
       href=bootstrap + 'css/bootstrap.min.css')
  link(rel='stylesheet',
       href=bootswatch + 'paper/bootstrap.min.css')
  script(src='https://code.jquery.com/jquery-2.1.1.min.js')
  [script(src=bootstrap + 'js/' + x) ❹
    for x in ['bootstrap.min.js', 'bootstrap-dropdown.js']]

with doc:
  with div(cls='container'):
    with h1():
      span(cls='glyphicon glyphicon-map-marker') ❺
      span('My Heading')
    with div(cls='row'):
      with div(cls='col-sm-6'):
        p('{{body}}') ❻
      with div(cls='col-sm-6'):
        p('Evaluate an expression:')
        input_(id='expression', type='text')
        button('Evaluate', cls='btn btn-primary')
        div(style='margin-top: 10px;')
        with div(cls='dropdown'):
          with button(cls="btn btn-default dropdown-toggle",
                      type="button", data_toggle='dropdown'):
            span('Dropdown')
            span(cls='caret')
          items = ('Action', 'Another action',
                   'Yet another action')
          ul((li(a(x, href='#')) for x in items),
             cls='dropdown-menu') ❹

    with div(cls='row'):
      h3('Progress:')
      with div(cls='progress'): ❼
        with div(cls='progress-bar', role='progressbar',
                 style='width: 60%;'):
          span('60%')
```

```
        with div(cls='row'):  ❽
          for vid in ['4vuW6tQ0218', 'wZZ7oFKsKzY', 'NfnMJMkhDoQ']:
            with div(cls='col-sm-4'):
              with div(cls='embed-responsive embed-responsive-16by9'):
                iframe(
                    cls='embed-responsive-item',
                    src='https://www.youtube.com/embed/' + vid,
                    frameborder='0')


@route('/')
def root():
    word = lambda: ''.join(
        choice(ascii_letters) for i in range(randint(2, 10)))
    nih_lorem = ' '.join(word() for i in range(50))
    return template(str(doc), body=nih_lorem)


if __name__ == '__main__':
    import threading
    thread = threading.Thread(
        target=run, kwargs=dict(host='localhost', port=8080),
        daemon=True)
    thread.start()

    webview.create_window(
        "Not a browser, honest!", "http://localhost:8080",
        width=800, height=600, resizable=False)
```

❶ `webview`, the star of the show!

❷ The other star of the show. `dominate` lets you create HTML with a series of nested context handlers.

❸ The third star of the show! The `bottle` framework provides a very simple interface for building a basic web app with templates and routing.

❹ Building up HTML in Python has the tremendous advantage of using all the syntax tools the language has to offer.

❺ By using Bootstrap, one of the oldest and most robust front-end frameworks, we get access to fun things like glyph icons.

❻ Template variables can still be entered into our HTML and substituted later using the web framework's tools.

❼ To demonstrate that we really do have Bootstrap, here is a progress bar widget. With `pywebview`, you can, of course, use any Bootstrap widget, and indeed any other tools that would normally be used in web browser user interfaces.

❽ I've embedded a few fun videos from YouTube.

Running this program produces a great-looking interface, as shown in Figure 1-6.

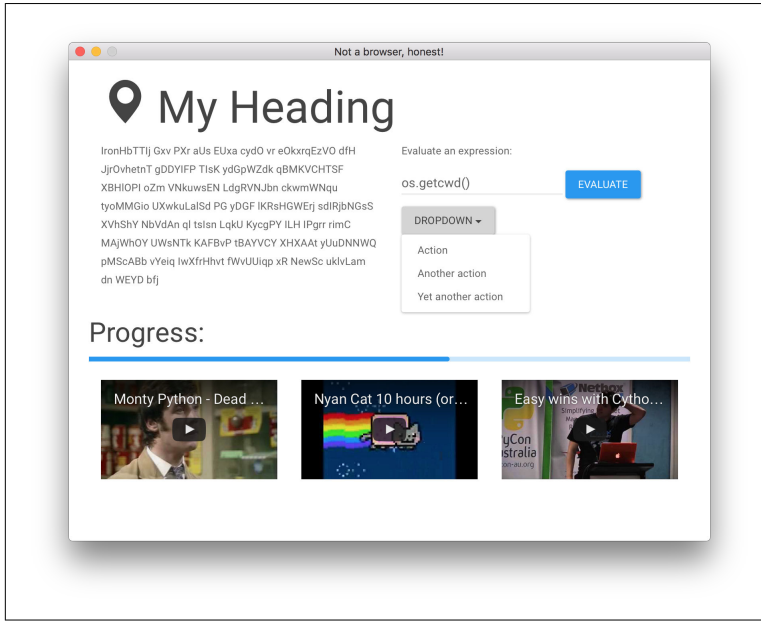

Figure 1-6. A beautiful and pleasing interface with very little effort.

For this demonstration, I've used the *Paper* Bootstrap theme, but by changing a single word you can use an entirely different one! The *Superhero* Bootstrap theme is shown in Figure 1-7.

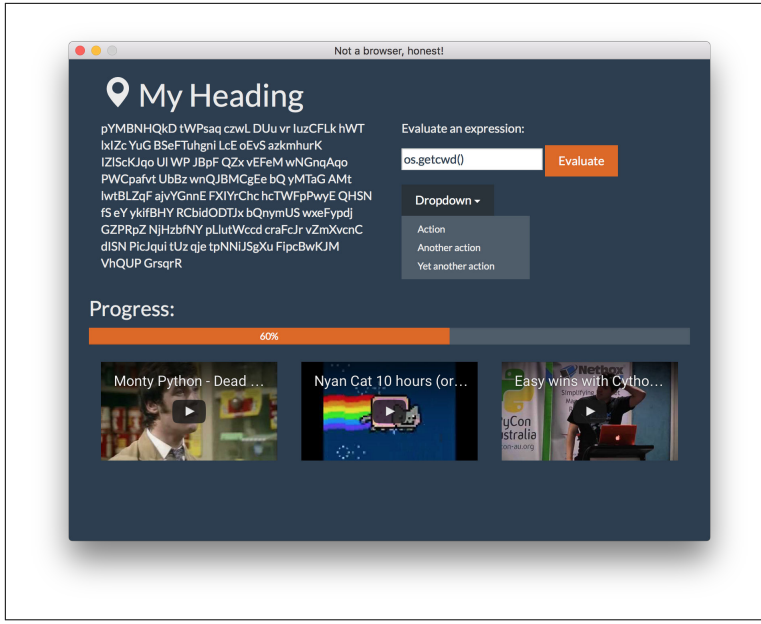

Figure 1-7. The same application, but using the Superhero theme from Bootswatch.

NOTE

In the preceding example we also used the `dominate` library, which is a handy utility for generating HTML procedurally. It's not only for simple tasks: this demo shows how it can handle Bootstrap attributes and widgets quite successfully. `dominate` works with nested context managers to manage the scope of HTML tags, while HTML attributes are assigned with keyword arguments.

For the kind of application that `Pywebview` is aimed at, it would be very interesting to think about how `dominate` could be used to make an abstraction that completely hides browser technologies like HTML and JavaScript, and lets you create (e.g., button-click handlers) entirely in Python without worrying about the intermediate event processing within the JavaScript engine. This technique is already explored in existing tools like flexx.

System Tools

Python is heavily used to make tools that work closely with the operating system, and it should be no surprise to discover that there are excellent libraries waiting to be discovered. The `psutil` library gives you all the access to operating-system and process information you could possibly hope for, while the `Watchdog` library gives you hooks into system file-event notifications. Finally, we close out this chapter with a look at `ptpython`, which gives you a significantly enriched interactive Python prompt.

psutil

Psutil provides complete access to system information. Here's a simple example of a basic report for CPU load sampled over 5 seconds:

```
import psutil

cpu = psutil.cpu_percent(interval=5, percpu=True)
print(cpu)
```

Output:

```
[21.4, 1.2, 18.0, 1.4, 15.6, 1.8, 17.4, 1.6]
```

It produces one value for each logical CPU: on my computer, eight. The rest of the `psutil` API is as simple and clean as this example shows, and the API also provides access to memory, disk, and network information. It is *very* extensive.

There is also detailed information about processes. To demonstrate, here is a program that monitors its own memory consumption and throws in the towel when a limit is reached:

```
import psutil
import os, sys, time

pid = os.getpid()  ❶
p = psutil.Process(pid)  ❷
print('Process info:')
print('  name  :', p.name())
print('  exe   :', p.exe())

data = []
while True:
    data += list(range(100000))  ❸
    info = p.memory_full_info()
    # Convert to MB
```

```
    memory = info.uss / 1024 / 1024
    print('Memory used: {:.2f} MB'.format(memory))
    if memory > 40:
        print('Memory too big! Exiting.')
        sys.exit()
    time.sleep(1)
```

❶ The `os` module in the standard library provides our process ID (PID).

❷ `psutil` constructs a `Process()` object based on the PID. In this case the PID is our own, but it could be any running process on the system.

❸ This looks bad: an infinite loop *and* an ever-growing list!

Output:

```
Process info:
  name  : Python
  exe   : /usr/local/Cellar/.../Python  ❶
Memory used: 11.82 MB
Memory used: 14.91 MB
Memory used: 18.77 MB
Memory used: 22.63 MB
Memory used: 26.48 MB
Memory used: 30.34 MB
Memory used: 34.19 MB
Memory used: 38.05 MB
Memory used: 41.90 MB
Memory too big! Exiting.
```

❶ The full path has been shortened here to improve the appearance of the snippet. There are many more methods besides `name` and `exe` available on `psutil.Process()` instances.

The type of memory shown here is the *unique set size*, which is the real memory released when that process terminates. The reporting of the unique set size is a new feature in version 4 of `psutil`, which also works on Windows.

There is an extensive set of process properties you can interrogate with `psutil`, and I encourage you to read the documentation. Once you begin using `psutil`, you will discover more and more scenarios in which it can make sense to capture process properties. For example, it might be useful, depending on your application, to capture

process information inside exception handlers so that your error logging can include information about CPU and memory load.

Watchdog

`Watchdog` is a high-quality, cross-platform library for receiving notifications of changes in the file system. Such file-system notifications is a fundamental requirement in many automation applications, and `Watchdog` handles all the low-level and cross-platform details of notifications for system events. And, the great thing about `Watchdog` is that it doesn't use *polling*.

The problem with polling is that having many such processes running can sometimes consume more resources than you may be willing to part with, particularly on lower-spec systems like the Raspberry Pi. By using the native notification systems on each platform, the operating system *tells you* immediately when something has changed, rather than you having to ask. On Linux, the `inotify` API is used; on Mac OS X, either `kqueue` or `FSEvents` are used; and on Windows, the `ReadDirectoryChangesW` API is used. `Watchdog` allows you to write cross-platform code and not worry too much about how the sausage is made.

`Watchdog` has a mature API, but one way to get immediate use out of it is to use the included `watchmedo` command-line utility to run a shell command when something changes. Here are a few ideas for inspiration:

- Compiling template and markup languages:

```
# Compile Jade template language into HTML
$ watchmedo shell-command \
      --patterns="*.jade" \
      --command='pyjade -c jinja "${watch_src_path}"' \  ❶
      --ignore-directories

# Convert an asciidoc to HTML
$ watchmedo shell-command \
      --patterns="*.asciidoc" \
      --command='asciidoctor "${watch_src_path}"' \  ❶
      --ignore-directories
```

❶ Watchdog will substitute the name of the specific changed file using this template name.

- Maintenance tasks, like backups or mirroring:

```
# Synchronize files to a server
$ watchmedo shell-command \
      --patterns="*" \
      --command='rsync -avz mydir/ host:/home/ubuntu"'
```

- Convenience tasks, like automatically running formatters, linters, and tests:

```
# Automatically format Python source code to PEP8(!)
$ watchmedo shell-command \
      --patterns="*.py" \
      --command='pyfmt -i "${watch_src_path}"' \
      --ignore-directories
```

- Calling API endpoints on the Web, or even sending emails and SMS messages:

```
# Mail a file every time it changes!
$ watchmedo shell-command \
      --patterns="*" \
      --command='cat "${watch_src_path}" | ↲
      mail -s "New version" me@domain.com' \
      --ignore-directories
```

The API of `Watchdog` as a library is fairly similar to the command-line interface introduced earlier. There are the usual quirks with thread-based programming that you have to be aware of, but the typical idioms are sufficient:

```
from watchdog.observers import Observer
from watchdog.events import (
    PatternMatchingEventHandler, FileModifiedEvent,
    FileCreatedEvent)

observer = Observer()  ❶

class Handler(PatternMatchingEventHandler):
    def on_created(self, event: FileCreatedEvent):  ❷
        print('File Created: ', event.src_path)

    def on_modified(self, event: FileModifiedEvent):  ❷
        print('File Modified: %s [%s]' % (
                event.src_path, event.event_type))

observer.schedule(event_handler=Handler('*'), path='.')  ❸
observer.daemon = False
observer.start()
```

```
try:
    observer.join() ❹
except KeyboardInterrupt:
    print('Stopped.')
    observer.stop()
    observer.join()
```

❶ Create an *observer* instance.

❷ You have to subclass one of the *handler* classes, and override the methods for events that you want to process. Here I've implemented handlers for creation and modification, but there are several other methods you can supply, as explained in the documentation.

❸ You *schedule* the event handler, and tell `watchdog` what it should be watching. In this case I've asked for notifications on *all files* (`*`) n the *current directory* (`.`).

❹ `Watchdog` runs in a separate thread. By calling the `join()` method, you can force the program flow to block at this point.

With that code running, I cunningly executed a few of these:

```
$ touch secrets.txt
$ touch secrets.txt
$ touch secrets.txt
[etc]
```

This is the console output from the running `Watchdog` demo:

```
File Created:  ./secrets.txt
File Modified: . [modified]
File Modified: ./secrets.txt [modified]
File Modified: ./secrets.txt [modified]
File Modified: ./secrets.txt [modified]
File Modified: ./secrets.txt [modified]
Stopped. ❶

Process finished with exit code 0
```

❶ Here I terminated the program by pressing Ctrl-C.

As we can see, `Watchdog` saw all the modifications I made to a file in the directory being watched.

ptpython

`ptpython` is an alternative interpreter interface, offering a beautiful interactive Python experience. As such, `ptpython` is more like a tool than a library to include in your own projects, but what a tool! Figure 1-8 shows a screenshot showing the basic user interface.

When you need to work with blocks in the interpreter, like classes and functions, you will find that accessing the command history is much more convenient than the standard interpreter interface: when you scroll to a previous command, the entire block is shown, not only single lines, as shown in Figure 1-9.

Here, I retrieved the code for the function declaration by pressing the up arrow (or Ctrl-p) and the entire block is shown, rather than having to scroll through lines separately.

The additional features include *vi* and *Emacs* keys, theming support, docstring hinting, and input validation, which checks a command (or block) for syntax errors before allowing evaluation. To see the full set of configurable options, press F2, as seen in Figure 1-10.

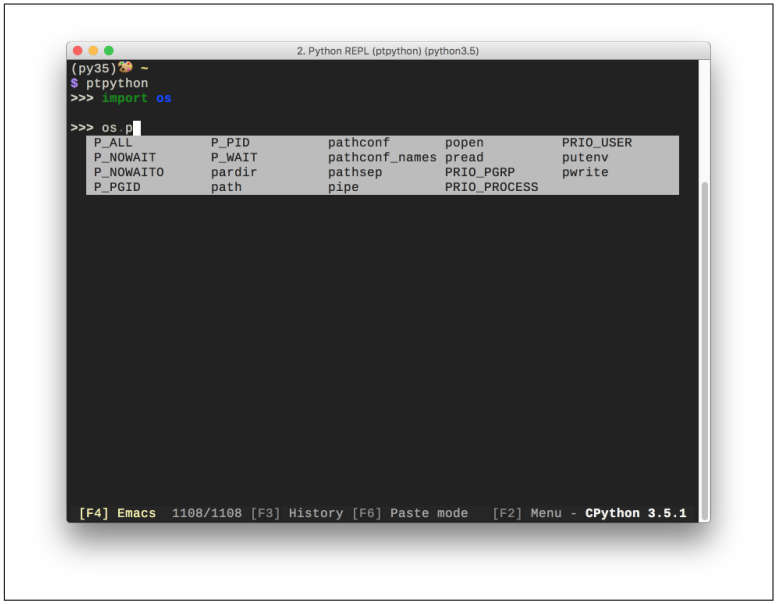

Figure 1-8. The code-completion suggestions pop up automatically as you type.

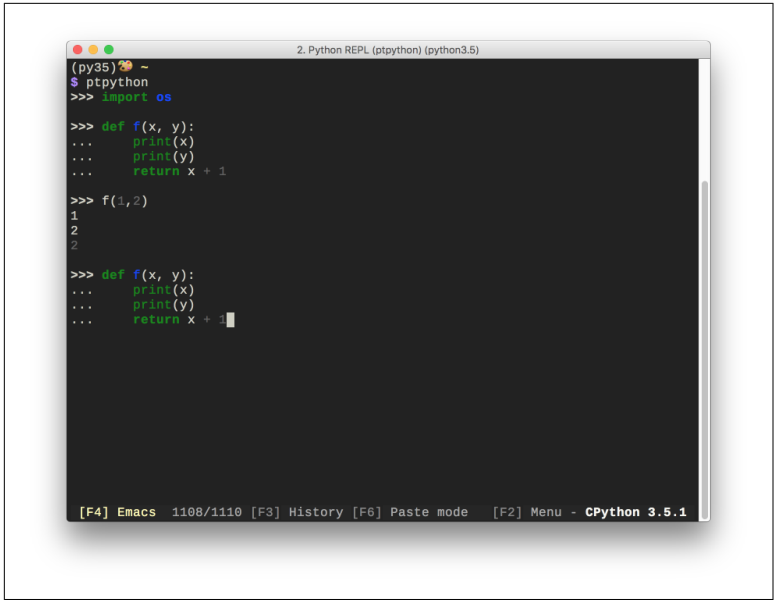

Figure 1-9. When you scroll to previous entries, entire blocks are suggested (e.g., function f()), not only single lines.

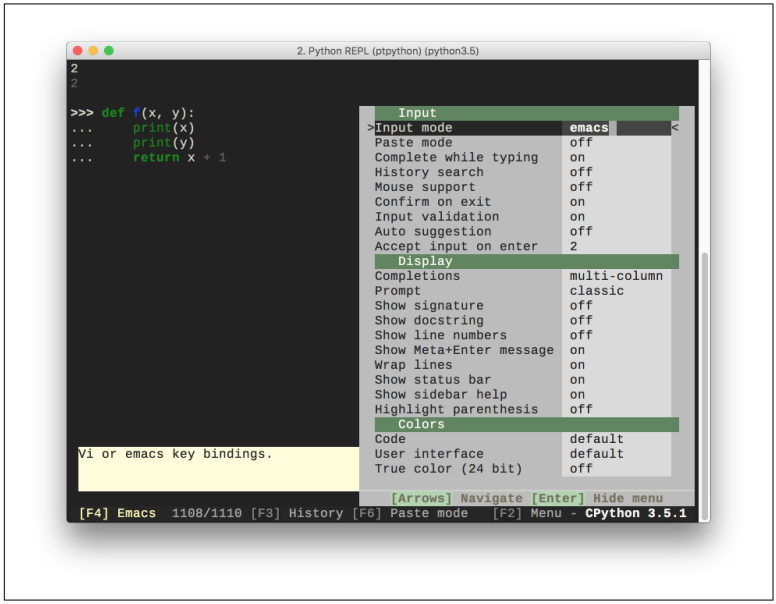

Figure 1-10. The settings menu accessible via F2.

On the command line, `ptpython` offers highly productive features that are sure to enhance your workflow. I have found that `ptpython` is the fastest way to quickly test some syntax or a library call, without having to open an editor or integrated development environment (IDE). And if you prefer *IPython*, you'll be glad to hear that `ptpython` includes an integration which you can launch with `ptipython`. This makes available the shell integration that *IPython* offers, as well as its wealth of magic commands.

Web APIs with hug

Python is famously used in a large number of web frameworks, and an extension of this area is the web services domain in which APIs are exposed for other users' programs to consume. In this domain, Django REST framework and Flask are very popular choices, but you may not yet have heard of `hug`.

hug is a library that provides an extremely simple way to create Internet APIs for web services. By exploiting some of the language features of Python, much of the usual boilerplate normally required when creating APIs for web services is removed.

Here is a small web service that converts between the hex value of a color and its CSS3 name:

```
import hug
import webcolors

@hug.get()
def hextoname(hex: hug.types.text):
    return webcolors.hex_to_name('#' + hex)

@hug.get()
def nametohex(name: hug.types.text):
    return webcolors.name_to_hex(name)
```

NOTE

We are also using the webcolors library here: your one-stop shop for converting web-safe colors between various formats like name, hex, and `rgb`. Don't forget that you can also convert between `rgb` and other formats like `hsl` with the colorsys library that's already included in the Python standard library!

In the `hug` example, we've done little more than wrap two functions from the `webcolors` package: one to convert from hex to name and one to do the opposite. It's hard to believe at first, but those `@hug.get` decorators are all you need to start a basic API. The server is launched by using the included `hug` command-line tool (Figure 1-11).

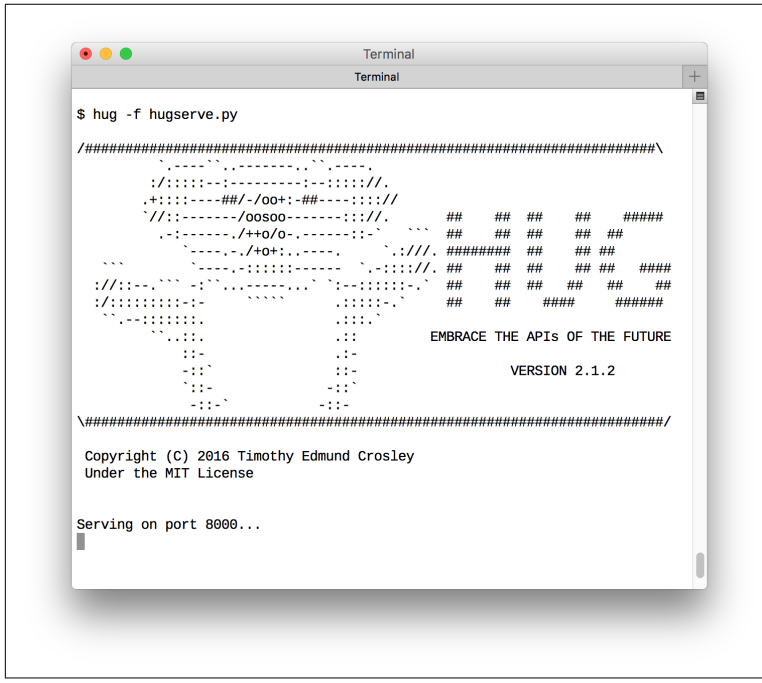

Figure 1-11. Hug goes a long way to make you feel welcome.

And that's it! We can immediately test our API. You could use a web browser for this, but it's easy enough to use a tool like `cURL`. You call the URL endpoint with the correct parameters, and `hug` returns the answer:

```
$ curl http://localhost:8000/hextoname?hex=ff0000  ❶

"red"

$ curl http://localhost:8000/nametohex?name=lightskyblue  ❷

"#87cefa"
```

❶ The function name was `hextoname`, and the parameter name was `hex`. Note that we don't supply the hash, `#`, as part of the request data because it interferes with the parsing of the HTTP request.

❷ This API has a different endpoint function, `nametohex`, and its parameter is called `name`.

It's quite impressive to get a live web API up with so little work. And the features don't stop there: `hug` automatically generates API documentation, which is what you get by omitting an endpoint:

```
$ curl http://localhost:8000/
{
  "404": "The API call you tried to make was not defined. Here's
      a definition of the API to help you get going :)",
  "documentation": {
    "handlers": {
      "/hextoname": {
        "GET": {
          "outputs": {
            "format": "JSON (Javascript Serialized
                Object Notation)",
            "content_type": "application/json"
          },
          "inputs": {
            "hex": {
              "type": "Basic text / string value"
            }
          }
        }
      },
      "/nametohex": {
        "GET": {
          "outputs": {
            "format": "JSON (Javascript Serialized
                Object Notation)",
            "content_type": "application/json"
          },
          "inputs": {
            "name": {
            "type": "Basic text / string value"
            }
          }
        }
      }
    }
  }
}
```

The types of the parameters we specified with `hug.types.text` are used not only for the documentation, but also for type conversions.

If these were all the features provided by `hug`, it would already be enough for many API tasks; however, one of hug's best features is that it automatically handles versions. Consider the following example:

```
import hug
import inflect  ❶
engine = inflect.engine()

@hug.get(versions=1)
def singular(word: hug.types.text):
    """ Return the singular version of the word"""
    return engine.singular_noun(word).lower()  ❷

@hug.get(versions=1)
def plural(word: hug.types.text):
    """ Return the plural version of the word"""
    return engine.plural(word).lower()  ❷

@hug.get(versions=2)
def singular(word: hug.types.text):
    """ Return the singular of word, preserving case """
    return engine.singular_noun(word)  ❸

@hug.get(versions=2)
def plural(word: hug.types.text):
    """ Return the plural of word, preserving case """
    return engine.plural(word)
```

❶ This new `hug` API wraps the `inflect` package, which provides tools for word manipulation.

❷ The first version of the API returns lowercased results.

❸ The second, newer version of the *same API* returns the result as calculated by `inflect`, without any alteration.

NOTE

We are also using the inflect library, which makes an enormous number of word transformations possible, including pluralization, gender transformation of pronouns, counting ("was no error" versus "were no errors"), correct article selection ("a," "an," and "the"), Roman numerals, and many more!

Imagine that we've created this web service to provide an API to transform words into their singular or plural version: one apple, many apples, and so on. For some reason that now escapes us, in the first release we lowercased the results before returning them:

```
$ curl http://localhost:8000/v1/singular/?word=Silly%20Walks  ❶

"silly walk"


$ curl http://localhost:8000/v1/plural?word=Crisis

"crises"
```

❶ Note: the URL now has a "v1" specifier for "version 1."

It didn't occur to us that some users may *prefer* to have the case of input words preserved: for example, if a word at the beginning of a sentence is being altered, it would be best to preserve the capitalization. Fortunately, the `inflect` library already does this by default, and `hug` provides versioning that allows us to provide a second version of the API (so as not to hurt existing users who may expect the lower-case transformation):

```
$ curl http://localhost:8000/v2/singular/?word=Silly%20Walks

"Silly Walk"


$ curl http://localhost:8000/v2/plural/?word=Crisis

"Crises"
```

These API calls use version 2 of our web service. And finally, the documentation is also versioned, and for this example, you can see how the function *docstrings* are also incorporated as usage text:

```
$ curl http://localhost:8000/v2/  ❶
{
  "404": "The API call you tried to make was not defined. Here's
      a definition of the API to help you get going :)",
  "documentation": {
    "version": 2,  ❶
    "versions": [
      1,
      2
    ],
    "handlers": {
      "/singular": {
```

```
            "GET": {
              "usage": " Return the singular of word,
                  preserving case ",  ❷
              "outputs": {
                "format": "JSON (Javascript Serialized
                    Object Notation)",
                "content_type": "application/json"
              },
              "inputs": {
                "word": {
                  "type": "Basic text / string value"
                }
              }
            }
          },

    [snip...]
```

❶ Each version has separate documentation, and calling that version endpoint, i.e., `v2`, returns the documentation for that version.

❷ The *docstring* of each function is reused as the usage text for that API call.

`hug` has extensive documentation in which there are even more features to discover. With the exploding interest in internet-of-things applications, it is likely that simple and powerful libraries like `hug` will enjoy widespread popularity.

Dates and Times

Many Python users cite two major pain points: the first is *packaging*, and for this we covered `flit` in an earlier chapter. The second is working with dates and times. In this chapter we cover two libraries that will make an enormous difference in how you deal with temporal matters. `arrow` is a reimagined library for working with `datetime` objects in which timezones are always present, which helps to minimize a large class of errors that new Python programmers encounter frequently. `parsedatetime` is a library that lets your code parse natural-language inputs for dates and times, which can make it vastly easier for your users to provide such information.

arrow

Though opinions differ, there are those who have found the standard library's `datetime` module confusing to use. The module documentation itself in the first few paragraphs explains why: it provides both *naive* and *aware* objects to represent dates and times. The *naive* ones are the source of confusion, because application developers increasingly find themselves working on projects in which timezones are critically important. The dramatic rise of cloud infrastructure and software-as-a-service applications have contributed to the reality that your application will frequently run in a different timezone (e.g., on a server) than where developers are located, and different again to where your users are located.

The *naive* `datetime` objects are the ones you typically see in demos and tutorials:

```
import datetime

dt = datetime.datetime.now()
dt_utc = datetime.datetime.utcnow()
difference = (dt - dt_utc).total_seconds()

print('Total difference: %.2f seconds' % difference)
```

This code could not be any simpler: we create two `datetime` objects, and calculate the difference. The problem is that we probably intended both `now()` and `utcnow()` to mean *now* as in "this moment in time," but perhaps in different timezones. When the difference is calculated, we get what seems an absurdly large result:

```
Total difference: 36000.00 seconds
```

The difference is, in fact, the timezone difference between my current location and UTC: +10 hours. This result is the worst kind of "incorrect" because it is, in fact, *correct* if timezones are not taken into account; and here lies the root of the problem: a misunderstanding of what the functions actually do. The `now()` and `utcnow()` functions *do* return the current time in local and UTC timezones, but unfortunately both results are *naive* `datetime` objects and therefore lack timezone data.

Note that it *is* possible to create `datetime` objects with timezone data attached, albeit in a subjectively cumbersome manner: you create an *aware* `datetime` by setting the `tzinfo` attribute to the correct timezone. In our example, we could (and should!) have created a current

`datetime` object in the following way, by passing the timezone into the `now()` function:

```
dt = datetime.datetime.now(tz=datetime.timezone.utc)
```

Of course, if you make this change on line 3 and try to rerun the code, the following error appears:

```
TypeError: can't subtract offset-naive and
offset-aware datetimes
```

This happens because even though the `utcnow()` function produces a `datetime` for the UTC timezone, the result is a *naive* `datetime` object, and Python prevents you from mixing naive and aware `datetime` objects. If you are suddenly scared to work with dates and times, that's an understandable response, but I'd rather encourage you to instead be afraid of `datetimes` that lack timezone data. The `TypeError` just shown is the kind of error we really do want: it forces us to ensure that all our `datetime` instances are *aware*.

Naturally, to handle this in the general case requires some kind of database of timezones. Unfortunately, except for UTC, the Python standard library does not include timezone data; instead, the recommendation is to use a separately maintained library called pytz, and if you need to work with dates and times, I strongly encourage you to investigate that library in more detail.

Now we can finally delve into the main topic of this section, which is `arrow`: a third-party library that offers a much simpler API for working with dates and times. With `arrow`, *everything* has a timezone attached (and is thus "aware"):

```
import arrow

t0 = arrow.now()
print(t0)

t1 = arrow.utcnow()
print(t1)

difference = (t0 - t1).total_seconds()

print('Total difference: %.2f seconds' % difference)
```

Output:

```
2016-06-26T18:43:55.328561+10:00
2016-06-26T08:43:55.328630+00:00
Total difference: -0.00 seconds
```

As you can see, the `now()` function produces the current date and time *with* my local timezone attached (UTC+10), while the `utc now()` function *also* produces the current date and time, but with the UTC timezone attached. As a consequence, the actual difference between the two times is as it should be: zero.

And the delightful `arrow` library just gets better from there. The objects themselves have convenient attributes and methods you would expect:

```
>>> t0 = arrow.now()
>>> t0
<Arrow [2016-06-26T18:59:07.691803+10:00]>

>>> t0.date()
datetime.date(2016, 6, 26)

>>> t0.time()
datetime.time(18, 59, 7, 691803)

>>> t0.timestamp
1466931547

>>> t0.year
2016

>>> t0.month
6

>>> t0.day
26

>>> t0.datetime
datetime.datetime(2016, 6, 26, 18, 59, 7, 691803,
    tzinfo=tzlocal())
```

Note that the `datetime` produced from the same attribute correctly carries the essential timezone information.

There are several other features of the module that you can read more about in the documentation, but this last feature provides an appropriate segue to the next section:

```
>>> t0 = arrow.now()
>>> t0.humanize()
'just now'
>>> t0.humanize()
'seconds ago'

>>> t0 = t0.replace(hours=-3,minutes=10)
>>> t0.humanize()
'2 hours ago'
```

The humanization even has built-in support for several locales!

```
>>> t0.humanize(locale='ko')
'2시간 전'
>>> t0.humanize(locale='ja')
'2 時間前'
>>> t0.humanize(locale='hu')
'2 órával ezelőtt'
>>> t0.humanize(locale='de')
'vor 2 Stunden'
>>> t0.humanize(locale='fr')
'il y a 2 heures'
>>> t0.humanize(locale='el')
'2 ώρες πριν'
>>> t0.humanize(locale='hi')
'2 घंटे पहले'
>>> t0.humanize(locale='zh')
'2 小时前'
```

Alternative libraries for dates and times

There are several other excellent libraries for dealing with dates and times in Python, and it would be worth checking out both Delorean and the very new Pendulum library.

parsedatetime

`parsedatetime` is a wonderful library with a dedicated focus: parsing text into dates and times. As you'd expect, it can be obtained with `pip install parsedatetime`. The official documentation is very API-like, which makes it harder than it should be to get a quick overview of what the library offers, but you can get a pretty good idea of what's available by browsing the extensive test suite.

The minimum you should expect of a `datetime`-parsing library is to handle the more common formats, and the following code sample demonstrates this:

```
import parsedatetime as pdt

cal = pdt.Calendar()

examples = [
    "2016-07-16",
    "2016/07/16",
    "2016-7-16",
    "2016/7/16",
    "07-16-2016",
    "7-16-2016",
    "7-16-16",
    "7/16/16",
]

print('{:30s}{:>30s}'.format('Input', 'Result'))
print('=' * 60)
for e in examples:
    dt, result = cal.parseDT(e)
    print('{:<30s}{:>30}'.format('"' + e + '"', dt.ctime()))
```

This produces the following, and unsurprising, output:

```
Input                                                  Result
============================================================
"2016-07-16"                          Sat Jul 16 16:25:20 2016
"2016/07/16"                          Sat Jul 16 16:25:20 2016
"2016-7-16"                           Sat Jul 16 16:25:20 2016
"2016/7/16"                           Sat Jul 16 16:25:20 2016
"07-16-2016"                          Sat Jul 16 16:25:20 2016
"7-16-2016"                           Sat Jul 16 16:25:20 2016
"7-16-16"                             Sat Jul 16 16:25:20 2016
"7/16/16"                             Sat Jul 16 16:25:20 2016
```

By default, if the year is given *last*, then *month-day-year* is assumed, and the library also conveniently handles the presence or absence of leading zeros, as well as whether hyphens (`-`) or slashes (`/`) are used as delimiters.

Significantly more impressive, however, is how `parsedatetime` handles more complicated, "natural language" inputs:

```
import parsedatetime as pdt
from datetime import datetime

cal = pdt.Calendar()

examples = [
    "19 November 1975",
    "19 November 75",
    "19 Nov 75",
```

```
    "tomorrow",
    "yesterday",
    "10 minutes from now",
    "the first of January, 2001",
    "3 days ago",
    "in four days' time",
    "two weeks from now",
    "three months ago",
    "2 weeks and 3 days in the future",
]

print('Now: {}'.format(datetime.now().ctime()), end='\n\n')
print('{:40s}{:>30s}'.format('Input', 'Result'))
print('=' * 70)
for e in examples:
    dt, result = cal.parseDT(e)
    print('{:<40s}{:>30}'.format('"' + e + '"', dt.ctime()))
```

Incredibly, this all works just as you'd hope:

```
Now: Mon Jun 20 08:41:38 2016

Input                                                          Result
======================================================================
"19 November 1975"                          Wed Nov 19 08:41:38 1975
"19 November 75"                            Wed Nov 19 08:41:38 1975
"19 Nov 75"                                 Wed Nov 19 08:41:38 1975
"tomorrow"                                  Tue Jun 21 09:00:00 2016
"yesterday"                                 Sun Jun 19 09:00:00 2016
"10 minutes from now"                       Mon Jun 20 08:51:38 2016
"the first of January, 2001"                Mon Jan  1 08:41:38 2001
"3 days ago"                                Fri Jun 17 08:41:38 2016
"in four days' time"                        Fri Jun 24 08:41:38 2016
"two weeks from now"                        Mon Jul  4 08:41:38 2016
"three months ago"                          Sun Mar 20 08:41:38 2016
"2 weeks and 3 days in the future"          Thu Jul  7 08:41:38 2016
```

The urge to combine this with a speech-to-text package like SpeechRecognition or watson-word-watcher (which provides confidence values per word) is almost irresistible, but of course you don't need complex projects to make use of `parsedatetime`: even allowing a user to type in a friendly and natural description of a date or time interval might be much more convenient than the usual but frequently clumsy *DateTimePicker* widgets we've become accustomed to.

NOTE

Another library featuring excellent datetime parsing abilities is Chronyk.

General-Purpose Libraries

In this chapter we take a look at a few batteries that have not yet been included in the Python standard library, but which would make excellent additions.

General-purpose libraries are quite rare in the Python world because the standard library covers most areas sufficiently well that library authors usually focus on very specific areas. Here we discuss `boltons` (a play on the word *builtins*), which provides a large number of useful additions to the standard library. We also cover the `Cython` library, which provides facilities for *both* massively speeding up Python code, as well as bypassing Python's famous *global interpreter lock* (GIL) to enable true multi-CPU multi-threading.

boltons

The `boltons` library is a general-purpose collection of Python modules that covers a wide range of situations you may encounter. The library is well-maintained and high-quality; it's well worth adding to your toolset.

As a general-purpose library, `boltons` does not have a specific focus. Instead, it contains several smaller libraries that focus on specific areas. In this section I will describe a few of these libraries that `boltons` offers.

boltons.cacheutils

`boltons.cacheutils` provides tools for using a *cache* inside your code. Caches are very useful for saving the results of expensive operations and reusing those previously calculated results.

The `functools` module in the standard library already provides a decorator called `lru_cache`, which can be used to *memoize* calls: this means that the function remembers the parameters from previous calls, and when the same parameter values appear in a new call, the previous answer is returned directly, bypassing any calculation.

`boltons` provides similar caching functionality, but with a few convenient tweaks. Consider the following sample, in which we attempt to rewrite some lyrics from Taylor Swift's *1989* juggernaut record. We will use tools from `boltons.cacheutils` to speed up processing time:

```
import json
import shelve
import atexit
from random import choice
from string import punctuation
from vocabulary import Vocabulary as vb


blank_space = """
Nice to meet you, where you been?
I could show you incredible things
Magic, madness, heaven, sin
Saw you there and I thought
Oh my God, look at that face
You look like my next mistake
Love's a game, wanna play?

New money, suit and tie
I can read you like a magazine
Ain't it funny, rumors fly
And I know you heard about me
So hey, let's be friends
I'm dying to see how this one ends
Grab your passport and my hand
I can make the bad guys good for a weekend
"""

from boltons.cacheutils import LRI, LRU, cached

# Persistent LRU cache for the parts of speech
cached_data = shelve.open('cached_data', writeback=True)  ❶
atexit.register(cached_data.close)  ❷

# Retrieve or create the "parts of speech" cache
cache_POS = cached_data.setdefault(
    'parts_of_speech', LRU(max_size=5000))  ❸

@cached(cache_POS)  ❹
def part_of_speech(word):
    items = vb.part_of_speech(word.lower())
    if items:
        return json.loads(items)[0]['text']

# Temporary LRI cache for word substitutions
```

```
cache = LRI(max_size=30)

@cached(cache)  ❺
def synonym(word):
    items = vb.synonym(word)
    if items:
        return choice(json.loads(items))['text']

@cached(cache)  ❺
def antonym(word):
    items = vb.antonym(word)
    if items:
        return choice(items['text'])

for raw_word in blank_space.strip().split(' '):
    if raw_word == '\n':
        print(raw_word)
        continue
    alternate = raw_word  # default is the original word.
    # Remove punctuation
    word = raw_word.translate(
        {ord(x): None for x in punctuation})
    if part_of_speech(word) in ['noun', 'verb',
                                'adjective', 'adverb']:  ❻
        alternate = choice((synonym, antonym))(word) or raw_word
    print(alternate, end=' ')
```

❶ Our code detects "parts of speech" in order to know which lyrics to change. Looking up words online is slow, so we create a small database using the `shelve` module in the standard library to save the cache data between runs.

❷ We use the `atexit` module, also in the standard library, to make sure that our "parts of speech" cache data will get saved when the program exits.

❸ Here we obtain the `LRU` cache provided by `boltons.cacheutils` that we saved from a previous run.

❹ Here we use the `@cache` decorator provided by `boltons.cacheutils` to enable caching of the `part_of_speech()` function call. If the `word` argument has been used in a previous call to this function, the answer will be obtained from the cache rather than a slow call to the Internet.

❺ For synonyms and antonyms, we used a different kind of cache, called a *least recently inserted* cache (this choice is explained later in this section). An LRI cache is not provided in the Python Standard Library.

❻ Here we restrict which kinds of words will be substituted.

NOTE

The excellent `vocabulary` package is used here to provide access to synonyms and antonyms. Install it with `pip install vocabulary`.

For brevity, I've included only the first verse and chorus. The plan is staggeringly unsophisticated: we're going to simply swap words with either a synonym or antonym, and which is decided randomly! Iteration over the words is straightforward, but we obtain synonyms and antonyms using the `vocabulary` package, which internally calls APIs on the Internet to fetch the data. Naturally, this can be slow since the lookup is going to be performed for every word, and this is why a cache will be used. In fact, in this code sample we use two different kinds of caching strategies.

`boltons.cacheutils` offers two kinds of caches: the *least recently used* (LRU) version, which is the same as `functools.lru_cache`, and a simpler *least recently inserted* (LRI) version, which expires entries based on their insertion order.

In our code, we use an *LRU* cache to keep a record of the *parts of speech* lookups, and we even save this cache to disk so that it can be reused in successive runs.

We also use an *LRI* cache to keep a record of word substitutions. For example, if a word is to be swapped with its antonym, the replacement will be stored in the LRI cache so that it can be reused. However, we apply a very small limit to the setting for maximum size on the LRI cache, so that words will fall out of the cache quite regularly. Using an LRI cache with a small maximum size means that the same word will be replaced with the same substitution only locally, say within the same verse; but if that same word appears later in the song (and that word has been dropped from the LRI cache), it might get a different substitution entirely.

The design of the caches in `boltons.cacheutils` is great in that it is easy to use the same cache for multiple functions, as we do here for the `synonym()` and `antonym()` functions. This means that once a word substitution appears in the cache, a call to *either* function returns the predetermined result from the same cache.

Here is an example of the output:

```
Nice to meet you, wherever you been?
I indeed conduct you astonishing things
Magic, madness, Hell sin
Saw you be and I thought
Oh my God, seek at who face
You seek same my following mistake
Love's a game, wanna play?

New financial satisfy both tie
I be able read you like a magazine
Ain't it funny, rumors fly
And gladly can you heard substantially me
So hey, let's inclination friends
I'm nascent to visit whatever that one ends
Grab your passport in addition my hand
I can take the bad guys ill in exchange for member weekend
```

On second thought, perhaps the original was best after all! It is worth noting just how much functionality is possible with a tiny amount of code, as long as the abstractions available to you are powerful enough.

`boltons` has many features and we cannot cover everything here; however, we can do a whirlwind tour and pick out a few notable APIs that solve problems frequently encountered, e.g., in StackOverflow questions.

boltons.iterutils

`boltons.iterutils.chunked_iter(src, size)` returns pieces of the source iterable in `size`-sized chunks (this example was copied from the docs):

```
>>> list(chunked_iter(range(10), 3))
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]
```

A similar requirement that often comes up is to have a moving window (of a particular size) slide over a sequence of data, and you can use `boltons.iterutils.windowed_iter` for that:

```
>>> list(windowed_iter(range(7), 3))
[(0, 1, 2), (1, 2, 3), (2, 3, 4), (3, 4, 5), (4, 5, 6)]
```

Note that both `chunked_iter()` and `windowed_iter()` can operate on *iterables*, which means that very large sequences of data can be processed while keeping memory requirements tolerable for your usage scenario.

boltons.fileutils

The copytree() function alleviates a particularly irritating behavior of the standard library's `shutil.copytree()` function: Boltons' `copytree()` will not complain if some or all of the destination file-system tree already exists.

The `boltons.fileutils.AtomicSaver` context manager helps to make sure that file-writes are protected against corruption. It achieves this by writing file data to temporary, or intermediate files, and then using an atomic renaming function to guarantee that the data is consistent. This is particularly valuable if there are multiple readers of a large file, and you want to ensure that the readers only ever see a consistent state, even though you have a (single!) writer changing the file data.

boltons.debugutils

If you've ever had a long-running python application, and wished that you could drop into an interactive debugger session to see what was happening, `boltons.debugutils.pdb_on_signal()` can make that happen. By default, a `KeyboardInterrupt` handler is automatically set up, which means that by pressing Ctrl-C you can drop immediately into the debugger prompt. This is a really great way to deal with infinite loops if your application is difficult to debug from a fresh start otherwise.

boltons.strutils

There are several functions in `boltons.strutils` that are enormously useful:

- `slugify()`: modify a string to be suitable, e.g., for use as a filename, by removing characters and symbols that would be invalid in a filename.

- `ordinalize()`: given a numerical value, create a string referring to its position:

```
>>> print(ordinalize(1))
1st
>>> print(ordinalize(2))
2nd
```

- `cardinalize`: given a word and a count, change the word for plurality and *preserve case*:

```
>>> cardinalize('python', 99)
'pythons'
>>> cardinalize('foot', 6)
'feet'
>>> cardinalize('Foot', 6)
'Feet'
>>> cardinalize('FOOT', 6)
'FEET'
>>> 'blind ' + cardinalize('mouse', 3)
'blind mice'
```

- `singularize` and `pluralize`:

```
>>> pluralize('theory')
'theories'
>>> singularize('mice')
'mouse'
```

- `bytes2human`: convert data sizes into friendler forms:

```
>>> bytes2human(1e6)
'977K'
>>> bytes2human(20)
'20B'
>>> bytes2human(1024 * 1024)
'1024K'
>>> bytes2human(2e4, ndigits=2)
'19.53K'
```

There are several other useful `boltons` libraries not mentioned here, and I encourage you to at least skim the documentation to learn about features you can use in your next project.

Cython

Cython is a magical tool! As with most magical devices, it is difficult to describe exactly what it is. Cython is a tool that converts Python

source code into C source code; this new code is then compiled into a native binary that is linked to the CPython runtime.

That sounds complicated, but basically Cython lets you convert your Python modules into compiled extension modules. There are two main reasons you might need to do this:

- You want to wrap a C/C++ library, and use it from Python.
- You want to speed up Python code.

The *second* reason is the one I'm going to focus on. By adding a few type declarations to your Python source code, you can dramatically speed up certain kinds of functions.

Consider the following code, which is as simple as I could possibly make it for this example:

```
import array  ❶

n = int(1e8) ❷
a = array.array('d', [0.0]) * n  ❸

for i in range(n): ❹
    a[i] = i % 3

print(a[:5])  ❺
```

❶ We're using the built-in `array` module.

❷ Set the size of our data.

❸ Create fictional data: a sequence of double-precision numbers; in reality your data would come from another source such as an image for image-processing applications, or numerical data for science or engineering applications.

❹ A very simple loop that modifies our data.

❺ Print the modified data; here, we only show the first five entries.

This code represents the most basic computer processing: data comes in, is transformed, and goes out. The specific code we're using is quite silly, but I hope it is clear enough so that it will be easy to understand how we implement this in Cython later.

We can run this program on the command line in the following way:

```
$ time python cythondemoslow.py
array('d', [0.0, 1.0, 2.0, 0.0, 1.0])

real    0m27.622s
user    0m27.109s
sys     0m0.443s
```

I've include the `time` command to get some performance measurements. Here we can see that this simple program takes around 30 seconds to run.

In order to use Cython, we need to modify the code slightly to take advantage of the Cython compiler's features:

```
import array  ❶

cdef int n = int(1e8)  ❷
cdef object a = array.array('d', [0.0]) * n
cdef double[:] mv = a  ❸

cdef int i  ❹
for i in range(n):
    mv[i] = i % 3  ❺

print(a[:5])
```

❶ We import the array module as before.

❷ The variable for the data size, `n`, now gets a specific datatype.

❸ This line is new: we create a *memory view* of the data inside the array `a`. This allows Cython to generate code that can access the data inside the array directly.

❹ As with `n`, we also specify a type for the loop index `i`.

❺ The work inside the loop is identical to before, except that we manipulate elements of the *memory view* rather than `a` itself.

Having modified our source code by adding information about native datatypes, we need to make three further departures from the normal Python workflow necessary before running our Cython code.

The first is that, by convention, we change the file extension of our source-code file to *pyx* instead of *py*, to reflect the fact that our source code is no longer normal Python.

The second is that we must use Cython to *compile* our source code into a native machine binary file. There are many ways to do this depending on your situation, but here we're going to go with the simple option and use a command-line tool provided by Cython itself:

```
$ cythonize -b -i cythondemofast.pyx
```

Running this command produces many lines of output messages from the compiler, but when the smoke clears you should find a new binary file in the same place as the `.pyx` file:

```
$ ls -l cythondemofast.cpython-35m-darwin.so
-rwxr-xr-x@ calebhattingh 140228 3 Jul 15:51
                              cythondemofast.cpython-35m-darwin.so
```

This is a native binary that Cython produced from our slightly modified Python source code! Now we need to run it, and this brings us to the third departure from the normal Python workflow: by default, Cython makes *native extensions* (as shared libraries), which means you have to import these in the same way you might import other Python extensions that use shared libraries.

With the first version of our example in ordinary Python, we could run the program easily with `python cythondemoslow.py`. We can run the code in our compiled *Cython* version simply by *importing* the native extension. As before, we include the `time` for measurement:

```
$ time python -c "import cythondemofast"
array('d', [0.0, 1.0, 2.0, 0.0, 1.0])

real    0m0.751s
user    0m0.478s
sys     0m0.270s
```

The Cython program gives us a speed-up over the plain Python program of almost 40 times! In larger numerical programs where the time cost of start-up and other initialization is a much smaller part of the overall execution time, the speed-up is usually more than 100 times!

In the example shown here, all our code was set out in the module itself, but usually you would write functions and after compiling

with Cython, and then import these functions into your main Python program. Here's how Cython can easily integrate into your normal Python workflow:

1. Begin your project with normal Python.
2. Benchmark your performance.
3. If you need more speed, *profile* your code to find the functions that consume most of the time.
4. Convert these functions to Cython functions, and compile the new `.pyx` Cython modules into native extensions.
5. Import the new Cython functions into your main Python program.

Multithreading with Cython

It won't take long for a newcomer to the Python world to hear about Python's so-called GIL, a safety mechanism Python uses to decrease the possibility of problems when using threads.

Threading is a tool that lets you execute different sections of code in *parallel*, allowing the operating system to run each section on a separate CPU. The "GIL problem" is that the safety lock that prevents the interpreter state from being clobbered by parallel execution *also* has the unfortunate effect of limiting the ability of threads to actually run on different CPUs. The net effect is that Python threads do not achieve the parallel performance one would expect based on the availability of CPU resources.

Cython gives us a way out of this dilemma, and enables multithreading at full performance. This is because *native extensions* (which is what Cython makes) are allowed to tell the main Python interpreter that they will be well-behaved and don't need to be protected with the global safety lock. This means that threads containing Cython code can run in a fully parallel way on multiple CPUs; we just need to ask Python for permission.

In the following code snippet, we demonstrate how to use normal Python threading to speed up the same nonsense calculation I used in previous examples:

```
# cython: boundscheck=False, cdivision=True
import array
import threading  ❶
```

```
cpdef void target(double[:] piece) nogil:  ❷
    cdef int i, n = piece.shape[0]
    with nogil:  ❸
        for i in range(n):
            piece[i] = i % 3


cdef int n = int(1e8)  ❹
cdef object a = array.array('d', [0.0]) * n


view = memoryview(a)  ❺
piece_size = int(n / 2)  ❻

thread1 = threading.Thread(  ❼
    target=target,
    args=(view[:piece_size],)
)

thread2 = threading.Thread(
    target=target,
    args=(view[piece_size:],)
)

thread1.start()  ❽
thread2.start()

thread1.join()  ❾
thread2.join()

print(a[:5])
```

❶ The threading module in the standard library.

❷ `Thread` objects want a target function to execute, so we wrap our calculation inside a function. We declare (with the `nogil` keyword) that our function may want to release the GIL.

❸ The actual point where the GIL is released. The rest of the function is identical to before.

❹ Exactly the same as before.

❺ We create a *memory view* of the data inside the array. Cython is optimized to work with these kinds of memory views efficiently. (Did you know that `memoryview()` is a built-in *Python* function?)

❻ We're going to split up our big data array into two parts.

❼ Create normal Python threads: we must pass both the target function and the *view section* as the argument for the function. Note how each thread gets a different part of the view!

❽ Threads are started.

❾ We wait for the threads to complete.

I've also sneakily added a few small optimization options such as disabling bounds checking and enabling the faster "C division." Cython is very configurable in how it generates C code behind the scenes and the documentation is well worth investigating.

As before, we must compile our program:

```
$ cythonize -b -i -a cythondemopll.pyx
```

Then we can test the impact of our changes:

```
$ time python -c "import cythondemopll"
array('d', [0.0, 1.0, 2.0, 0.0, 1.0])

real    0m0.593s
user    0m0.390s
sys     0m0.276s
```

The use of threading has given us around 30% improvement over the previous, single-threaded version, and we're about 50 times faster than the original Python version in this example. For a longer-running program the speedup factor would be even more significant because the startup time for the Python interpreter would account for a smaller portion of the time cost.

Executables with Cython

One final trick with Cython is creating executables. So far we've been compiling our Cython code for use as a native extension module, which we then import to run. However, Cython also makes it possible to create a native binary executable directly. The key is to invoke `cython` directly with the `--embed` option:

```
$ cython --embed cythondemopll.pyx
```

This produces a C source file that will compile to an executable rather than a shared library.

The next step depends on your platform because you must invoke the C compiler directly, but the main thing you need to provide is the path to the Python header file and linking library. This is how it looks on my Mac:

```
$ gcc `python3.5-config --cflags` cythondemopll.c \
      `python3.5-config --ldflags` -o cythondemopll
```

Here I've used a utility called `python3.5-config` that conveniently returns the path to the header file and the Python library, but you could also provide the paths directly.

The compilation step using `gcc` produces a *native binary* executable that can be run directly on the command line:

```
$ ./cythondemopll
array('d', [0.0, 1.0, 2.0, 0.0, 1.0])
```

There is much more to learn about Cython, and I've made a comprehensive video series, *Learning Cython* (O'Reilly) that covers all the details. Cython's online documentation is also an excellent reference.

awesome-python

Finally, we have awesome-python. It's not a library, but rather a huge, curated list of a high-quality Python libraries covering a large number of domains. If you have not seen this list before, make sure to reserve some time before browsing because once you begin, you'll have a hard time tearing yourself away!

Conclusion

There is much more to discover than what you've seen in this report. One of the best things about the Python world is its enormous repository of high-quality libraries.

You have seen a few of the very special features of the standard library like the `collections` module, `contextlib`, the `concurrent.futures` module, and the `logging` module. If you do not yet use these heavily, I sincerely hope you try them out in your next project.

In addition to those standard library modules, we also covered several excellent libraries that are also available to you on the PyPI. You've seen how:

- *flit* makes it easy for you to create your own Python packages, and submit them to the PyPI.
- libraries like *colorama* and *begins* improve your command-line applications.
- tools like *pyqtgraph* and *pywebview* can save you lots of time when creating modern user interfaces, including *hug*, which can give your applications an easily created web API.
- system libraries like *psutil* and *watchdog* can give you a clean integration with the host operating system.
- temporal libraries like *arrow* and *parsedatetime* can simplify the tangled mess that working with dates and times often becomes.
- general-purpose libraries like *boltons* and *Cython* can further enrich the already powerful facilities in the Python standard library.

Hopefully you will be able to use one or more of the great libraries in your next project, and I wish you the best of luck!

About the Author

Caleb Hattingh is passionate about coding and has been programming for over 15 years, specializing in Python. He holds a master's degree in chemical engineering and has consequently written a great deal of scientific software within chemical engineering, from dynamic chemical reactor models all the way through to data analysis. He is very experienced with the Python scientific software stack, CRM, financial software development in the hotels and hospitality industry, frontend web experience using HTML, Sass, JavaScript (loves RactiveJS), and backend experience with Django and web2py. Caleb is a regular speaker at PyCon Australia and is actively engaged in the community as a CoderDojo Mentor, Software Carpentry helper, Govhacker, Djangogirls helper, and even Railsgirls helper. Caleb is the founder of Codermoji, and posts infrequent idle rants and half-baked ideas to his blog at *pythonomicon.com*.