SERIOUS PYTHON

BLACK-BELT ADVICE ON DEPLOYMENT, SCALABILITY, TESTING, AND MORE

JULIEN DANJOU

SERIOUS PYTHON

Black-Belt Advice on Deployment, Scalability, Testing, and More

by Julien Danjou

no starch
press

San Francisco

ISBN-10: 1-59327-878-0
ISBN-13: 978-1-59327-878-6

Publisher: William Pollock
Production Editor: Laurel Chun
Cover Illustration: Josh Ellingson
Interior Design: Octopod Studios
Developmental Editors: Liz Chadwick with Ellie Bru
Technical Reviewer: Mike Driscoll
Copyeditor: Paula L. Fleming
Compositor: Laurel Chun
Proofreader: James Fraleigh

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900; info@nostarch.com
www.nostarch.com

Library of Congress Cataloging-in-Publication Data

```
Names: Danjou, Julien, author.
Title: Serious Python : black-belt advice on deployment, scalability,
   testing, and more / Julien Danjou.
Description: San Francisco, CA : No Starch Press, Inc., [2019].
Identifiers: LCCN 2018042631 (print) | LCCN 2018050473 (ebook) | ISBN
   9781593278793 (epub) | ISBN 1593278799 (epub) | ISBN 9781593278786 (print)
   | ISBN 1593278780 (print) | ISBN 9781593278793 (ebook) | ISBN 1593278799
   (ebook)
Subjects:  LCSH: Python (Computer program language)
Classification: LCC QA76.73.P98 (ebook) | LCC QA76.73.P98 D36 2019 (print) |
   DDC 005.13/3--dc23
LC record available at https://lccn.loc.gov/2018042631
```

About the Author

Julien Danjou has been a free software hacker for close to twenty years and has been developing software with Python for twelve years. He currently works as Project Team Leader for the distributed cloud platform OpenStack, which has the largest existing open-source Python codebase at 2.5 million lines of Python. Before building clouds, Julien created the awesome window manager and contributed to various software such as Debian and GNU Emacs.

About the Technical Reviewer

Mike Driscoll has been programming with Python for more than a decade. He has been writing about Python on his blog, *The Mouse vs. The Python*, for many years. Mike is the author of several Python books including *Python 101*, *Python Interviews*, and *ReportLab: PDF Processing with Python*. You can find Mike on Twitter or GitHub via his handle: @driscollis.

BRIEF CONTENTS

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

Writing this first book has been a tremendous effort. Looking back, I had no clue how crazy this journey would be but also no idea how fulfilling it would turn out to be.

They say that if you want to go fast you should go alone, but that if you want to go far you should go together. This is the fourth edition of the original book I wrote, and I would not have made it here without the people who helped along the way. This is a team effort and I would like to thank everyone who participated.

Most of the interviewees gave me their time and trust without a second thought, and I owe a lot of what we teach in this book to them: Doug Hellmann for his great advice about building libraries, Joshua Harlow for his good humor and knowledge about distributed systems, Christophe de Vienne for his experience in building frameworks, Victor Stinner for his incredible CPython knowledge, Dimitri Fontaine for his database wisdom, Robert Collins for messing up with testing, Nick Coghlan for his work in getting Python into better shape, and Paul Tagliamonte for his amazing hacker spirit.

Thanks to the No Starch crew for working with me on bringing this book to a brand new level — especially to Liz Chadwick for her editing skills, Laurel Chun for keeping me on track, and Mike Driscoll for his technical insight.

My gratitude also goes to the free software communities who shared their knowledge and helped me grow, especially to the Python community which always has been welcoming and enthusiastic.

INTRODUCTION

If you're reading this, the odds are good you've been working with Python for some time already. Maybe you learned it using some tutorials, delved into some existing programs, or started from scratch. Whatever the case, you've *hacked* your way into learning it. That's exactly how I got familiar with Python up until I started working on big open source projects 10 years ago.

It is easy to think that you know and understand Python once you've written your first program. The language is that simple to grasp. However, it takes years to master it and to develop a deep comprehension of its advantages and shortcomings.

When I started Python, I built my own Python libraries and applications on a "garage project" scale. Things changed once I started working with hundreds of developers on software that thousands of users rely on. For example, the OpenStack platform—a project I contribute to—represents over 9 million lines of Python code, which collectively needs to be concise, efficient, and scalable to the needs of whatever cloud computing application its users require. When you have a project of this size, things like testing and documentation absolutely require automation, or else they won't get done at all.

I thought I knew a lot about Python before working on projects of this scale—a scale I could hardly imagine when I started out—but I've learned a lot more. I've also had the opportunity to meet some of the best Python hackers in the industry and learn from them. They've taught me everything from general architecture and design principles to

various helpful tips and tricks. Through this book, I hope to share the most important things I've learned so that you can build better Python programs—and build them more efficiently, too!

The first version of this book, *The Hacker's Guide to Python*, came out in 2014. Now *Serious Python* is the fourth edition, with updated and entirely new contents. I hope you enjoy it!

Who Should Read This Book and Why

This book is intended for Python coders and developers who want to take their Python skills to the next level.

In it, you'll find methods and advice that will help you get the most out of Python and build future-proof programs. If you're already working on a project, you'll be able to apply the techniques discussed right away to improve your current code. If you're starting your first project, you'll be able to create a blueprint with the best practice.

I'll introduce you to some Python internals to give you a better understanding of how to write efficient code. You will gain a greater insight into the inner workings of the language that will help you understand problems or inefficiencies.

The book also provides applicable battle-tested solutions to problems such as testing, porting, and scaling Python code, applications, and libraries. This will help you avoid making the mistakes that others have made and discover strategies that will help you maintain your software in the long run.

About This Book

This book is not necessarily designed to be read from front to back. You should feel free to skip to sections that interest you or are relevant to your work. Throughout the book, you'll find a wide range of advice and practical tips. Here's a quick breakdown of what each chapter contains.

Chapter 1 provides guidance about what to consider before you undertake a project, with advice on structuring your project, numbering

versions, setting up automated error checking, and more. At the end there's an interview with Joshua Harlow.

Chapter 2 introduces Python modules, libraries, and frameworks and talks a little about how they work under the hood. You'll find guidance on using the `sys` module, getting more from the `pip` package manager, choosing the best framework for you, and using standard and external libraries. There's also an interview with Doug Hellmann.

Chapter 3 gives advice on documenting your projects and managing your APIs as your project evolves even after publication. You'll get specific guidance on using Sphinx to automate certain documentation tasks. Here you'll find an interview with Christophe de Vienne.

Chapter 4 covers the age-old issue of time zones and how best to handle them in your programs using `datetime` objects and `tzinfo` objects.

Chapter 5 helps you get your software to users with guidance on distribution. You'll learn about packaging, distributions standards, the `distutils` and `setuptools` libraries, and how to easily discover dynamic features in a package using entry points. Nick Coghlan is interviewed.

Chapter 6 advises you on unit testing with best-practice tips and specific tutorials on automating unit tests with `pytest`. You'll also look at using virtual environments to increase the isolation of your tests. The interview is with Robert Collins.

Chapter 7 digs into methods and decorators. This is a look at using Python for functional programming, with advice on how and when to use decorators and how to create decorators *for* decorators. We'll also dig into static, class, and abstract methods and how to mix the three for a more robust program.

Chapter 8 shows you more functional programming tricks you can implement in Python. This chapter discusses generators, list comprehensions, functional functions and common tools for implementing them, and the useful `functools` library.

Chapter 9 peeks under the hood of the language itself and discusses the abstract syntax tree (AST) that is the inner structure of Python. We'll also look at extending `flake8` to work with the AST to introduce

more sophisticated automatic checks into your programs. The chapter concludes with an interview with Paul Tagliamonte.

Chapter 10 is a guide to optimizing performance by using appropriate data structures, defining functions efficiently, and applying dynamic performance analysis to identify bottlenecks in your code. We'll also touch on memoization and reducing waste in data copies. You'll find an interview with Victor Stinner.

Chapter 11 tackles the difficult subject of multithreading, including how and when to use multithreading as opposed to multiprocessing and whether to use event-oriented or service-oriented architecture to create scalable programs.

Chapter 12 covers relational databases. We'll take a look at how they work and how to use PostgreSQL to effectively manage and stream data. Dimitri Fontaine is interviewed.

Finally, **Chapter 13** offers sound advice on a range of topics: making your code compatible with both Python 2 and 3, creating functional Lisp-like code, using context managers, and reducing repetition with the `attr` library.

1
STARTING YOUR PROJECT

In this first chapter, we'll look at a few aspects of starting a project and what you should think about before you begin, such as which Python version to use, how to structure your modules, how to effectively number software versions, and how to ensure best coding practices with automatic error checking.

Versions of Python

Before beginning a project, you'll need to decide what version(s) of Python it will support. This is not as simple a decision as it may seem.

It's no secret that Python supports several versions at the same time. Each minor version of the interpreter gets bug-fix support for 18 months and security support for 5 years. For example, Python 3.7, released on June 27, 2018, will be supported until Python 3.8 is released, which should be around October 2019. Around December 2019, a last bug-fix release of Python 3.7 will occur, and everyone will be expected to switch to Python 3.8. Each new version of Python introduces new features and deprecates old ones. Figure 1-1 illustrates this timeline.

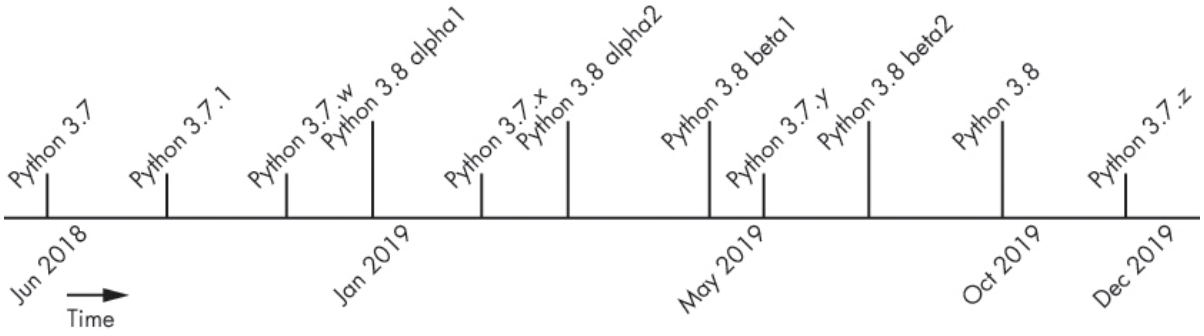


Figure 1-1: Python release timeline

On top of that, we should take into consideration the Python 2 versus Python 3 problem. People working with (very) old platforms may still require Python 2 support because Python 3 has not been made available on those platforms, but the rule of thumb is to forget Python 2 if you can.

Here is a quick way to figure out which version you need:

- Versions 2.6 and older are now obsolete, so I do not recommend you worry about supporting them at all. If you do intend to support these older versions for whatever reason, be warned that you'll have a hard time ensuring that your program supports Python 3.x as well. Having said that, you might still run into Python 2.6 on some older systems—if that's the case, sorry!
- Version 2.7 is and will remain the last version of Python 2.x. Every system is basically running or able to run Python 3 one way or the other nowadays, so unless you're doing archeology, you shouldn't need to worry about supporting Python 2.7 in new programs. Python 2.7 will cease to be supported after the year 2020, so the last thing you want to do is build a new software based on it.
- Version 3.7 is the most recent version of the Python 3 branch as of this writing, and that's the one that you should target. However, if your operating system ships version 3.6 (most operating systems, except Windows, ship with 3.6 or later), make sure your application will also work with 3.6.

Techniques for writing programs that support both Python 2.7 and 3.x will be discussed in Chapter 13.

Finally, note that this book has been written with Python 3 in mind.

Laying Out Your Project

Starting a new project is always a bit of a puzzle. You can't be sure how your project will be structured, so you might not know how to organize your files. However, once you have a proper understanding of best practices, you'll understand which basic structure to start with. Here I'll give some tips on dos and don'ts for laying out your project.

What to Do

First, consider your project structure, which should be fairly simple. Use packages and hierarchy wisely: a deep hierarchy can be a nightmare to navigate, while a flat hierarchy tends to become bloated.

Then, avoid making the common mistake of storing unit tests outside the package directory. These tests should definitely be included in a subpackage of your software so that they aren't automatically installed as a *tests* top-level module by `setuptools` (or some other packaging library) by accident. By placing them in a subpackage, you ensure they can be installed and eventually used by other packages so users can build their own unit tests.

Figure 1-2 illustrates what a standard file hierarchy should look like.

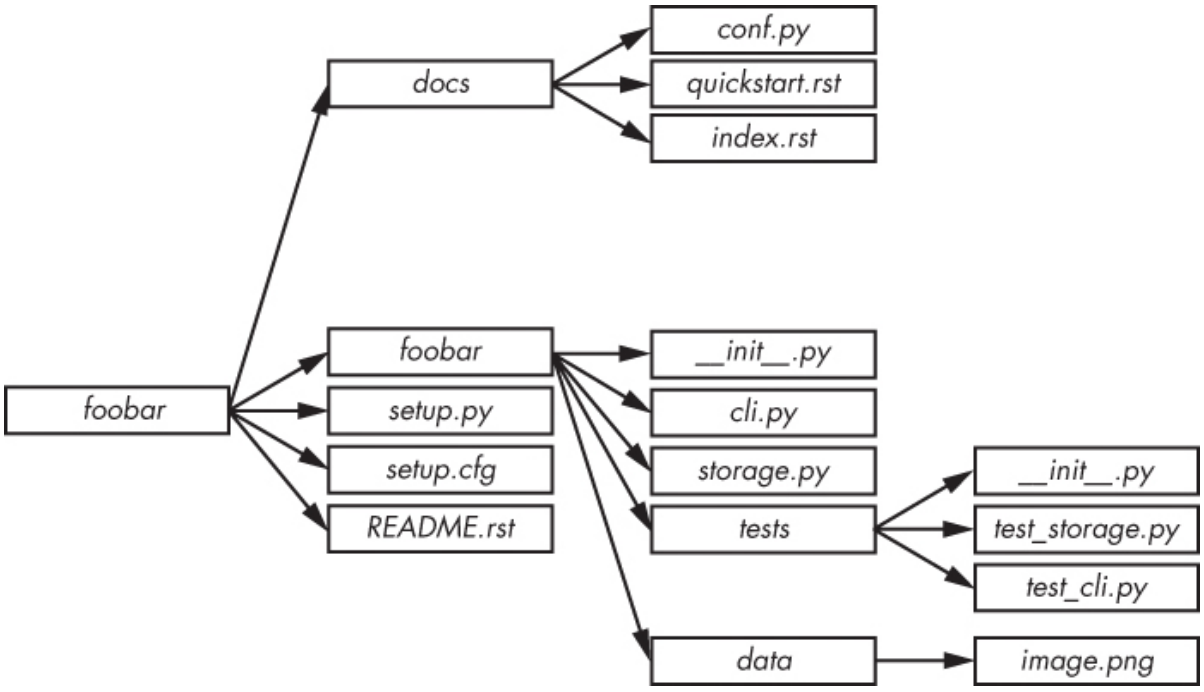


Figure 1-2: Standard package directory

The standard name for a Python installation script is *setup.py*. It comes with its companion *setup.cfg*, which should contain the installation script configuration details. When run, *setup.py* will install your package using the Python distribution utilities.

You can also provide important information to users in *README.rst* (or *README.txt*, or whatever filename suits your fancy). Finally, the *docs* directory should contain the package's documentation in *reStructuredText* format, which will be consumed by Sphinx (see Chapter 3).

Packages will often have to provide extra data for the software to use, such as images, shell scripts, and so forth. Unfortunately, there's no universally accepted standard for where these files should be stored, so you should just put them wherever makes the most sense for your project depending on their functions. For example, web application templates could go in a *templates* directory in your package root directory.

The following top-level directories also frequently appear:

- *etc* for sample configuration files

- *tools* for shell scripts or related tools
- *bin* for binary scripts you've written that will be installed by *setup.py*

What Not to Do

There is a particular design issue that I often encounter in project structures that have not been fully thought out: some developers will create files or modules based on the type of code they will store. For example, they might create *functions.py* or *exceptions.py* files. This is a *terrible* approach and doesn't help any developer when navigating the code. When reading a codebase, the developer expects a functional area of a program to be confined in a particular file. The code organization doesn't benefit from this approach, which forces readers to jump between files for no good reason.

Organize your code based on *features*, not on types.

It is also a bad idea to create a module directory that contains only an *__init__.py* file, because it's unnecessary nesting. For example, you shouldn't create a directory named *hooks* with a single file named *hooks/__init__.py* in it, where *hooks.py* would have been enough. If you create a directory, it should contain several other Python files that belong to the category the directory represents. Building a deep hierarchy unnecessarily is confusing.

You should also be very careful about the code that you put in the *__init__.py* file. This file will be called and executed the first time that a module contained in the directory is loaded. Placing the wrong things in your *__init__.py* can have unwanted side effects. In fact, *__init__.py* files should be empty most of the time, unless you know what you're doing. Don't try to remove *__init__.py* files altogether though, or you won't be able to import your Python module at all: Python requires an *__init__.py* file to be present for the directory to be considered a submodule.

Version Numbering

Software versions need to be stamped so users know which is the more recent version. For every project, users must be able to organize the timeline of the evolving code.

There is an infinite number of ways to organize your version numbers. However, PEP 440 introduces a version format that every Python package, and ideally every application, should follow so that other programs and packages can easily and reliably identify which versions of your package they require.

PEP 440 defines the following regular expression format for version numbering:

```
N[.N]+[{a|b|c|rc}N][.postN][.devN]
```

This allows for standard numbering such as `1.2` or `1.2.3`. There are a few further details to note:

- Version `1.2` is equivalent to `1.2.0`, `1.3.4` is equivalent to `1.3.4.0`, and so forth.
- Versions matching `N[.N]+` are considered *final* releases.
- Date-based versions such as `2013.06.22` are considered invalid. Automated tools designed to detect PEP 440–format version numbers will (or should) raise an error if they detect a version number greater than or equal to `1980`.
- Final components can also use the following format:
    - `N[.N]+aN` (for example, `1.2a1`) denotes an alpha release; this is a version that might be unstable and missing features.
    - `N[.N]+bN` (for example, `2.3.1b2`) denotes a beta release, a version that might be feature complete but still buggy.
    - `N[.N]+cN` or `N[.N]+rcN` (for example, `0.4rc1`) denotes a (release) candidate. This is a version that might be released as the final product unless significant bugs emerge. The `rc` and `c` suffixes have the same meaning, but if both are used, `rc` releases are considered newer than `c` releases.
- The following suffixes can also be used:

- The suffix *.postN* (for example, `1.4.post2`) indicates a post release. Post releases are typically used to address minor errors in the publication process, such as mistakes in release notes. You shouldn't use the *.postN* suffix when releasing a bug-fix version; instead, increment the minor version number.
- The suffix*.devN* (for example, `2.3.4.dev3`) indicates a developmental release. It indicates a prerelease of the version that it qualifies: for example, *2.3.4.dev3* indicates the third developmental version of the *2.3.4* release, prior to any alpha, beta, candidate, or final release. This suffix is discouraged because it is harder for humans to parse.

This scheme should be sufficient for most common use cases.

> **NOTE**
>
> *You might have heard of* Semantic Versioning, *which provides its own guidelines for version numbering. This specification partially overlaps with PEP 440, but unfortunately, they're not entirely compatible. For example, Semantic Versioning's recommendation for prerelease versioning uses a scheme such as* `1.0.0-alpha+001` *that is not compliant with PEP 440.*

Many *distributed version control system (DVCS)* platforms, such as Git and Mercurial, are able to generate version numbers using an identifying hash (for Git, refer to `git describe`). Unfortunately, this system isn't compatible with the scheme defined by PEP 440: for one thing, identifying hashes aren't orderable.

Coding Style and Automated Checks

Coding style is a touchy subject, but one we should talk about before we dive further into Python. Unlike many programming languages, Python uses *indentation* to define blocks. While this offers a simple solution to the age-old question "Where should I put my braces?" it introduces a new question: "How should I indent?"

That was one of the first questions raised in the community, so the Python folks, in their vast wisdom, came up with the *PEP 8: Style Guide for Python Code* (*https://www.python.org/dev/peps/pep-0008/*).

This document defines the standard style for writing Python code. The list of guidelines boils down to:

- Use four spaces per indentation level.
- Limit all lines to a maximum of 79 characters.
- Separate top-level function and class definitions with two blank lines.
- Encode files using ASCII or UTF-8.
- Use one module import per `import` statement and per line. Place import statements at the top of the file, after comments and docstrings, grouped first by standard, then by third party, and finally by local library imports.
- Do not use extraneous whitespaces between parentheses, square brackets, or braces or before commas.
- Write class names in camel case (e.g., `CamelCase`), suffix exceptions with `Error` (if applicable), and name functions in lowercase with words and underscores (e.g., `separated_by_underscores`). Use a leading underscore for `_private` attributes or methods.

These guidelines really aren't hard to follow, and they make a lot of sense. Most Python programmers have no trouble sticking to them as they write code.

However, *errare humanum est*, and it's still a pain to look through your code to make sure it fits the PEP 8 guidelines. Luckily, there's a `pep8` tool (found at *https://pypi.org/project/pep8/*) that can automatically check any Python file you send its way. Install `pep8` with `pip`, and then you can use it on a file like so:

```
$ pep8 hello.py
hello.py:4:1: E302 expected 2 blank lines, found 1
$ echo $?
1
```

Here I use `pep8` on my file *hello.py*, and the output indicates which lines and columns do not conform to PEP 8 and reports each issue with a code—here it's line 4 and column 1. Violations of *MUST* statements in the specification are reported as *errors*, and their error codes start with an *E*. Minor issues are reported as *warnings*, and their error codes start with a *W*. The three-digit code following that first letter indicates the exact kind of error or warning.

The hundreds digit tells you the general category of an error code: for example, errors starting with `E2` indicate issues with whitespace, errors starting with `E3` indicate issues with blank lines, and warnings starting with `W6` indicate deprecated features being used. These codes are all listed in the `pep8` readthedocs documentation (*https://pep8.readthedocs.io/*).

Tools to Catch Style Errors

The community still debates whether validating against PEP 8 code, which is not part of the Standard Library, is good practice. My advice is to consider running a PEP 8 validation tool against your source code on a regular basis. You can do this easily by integrating it into your continuous integration system. While this approach may seem a bit extreme, it's a good way to ensure that you continue to respect the PEP 8 guidelines in the long term. We'll discuss in "Using `virtualenv` with `tox`" on page 92 how you can integrate `pep8` with `tox` to automate these checks.

Most open source projects enforce PEP 8 conformance through automatic checks. Using these automatic checks from the very beginning of the project might frustrate newcomers, but it also ensures that the codebase always looks the same in every part of the project. This is very important for a project of any size where there are multiple developers with differing opinions on, for example, whitespace ordering. You know what I mean.

It's also possible to set your code to ignore certain kinds of errors and warnings by using the `--ignore` option, like so:

```
$ pep8 --ignore=E3 hello.py
$ echo $?
0
```

This will ignore any code E3 errors inside my *hello.py* file. The `--ignore` option allows you to effectively ignore parts of the PEP 8 specification that you don't want to follow. If you're running `pep8` on an existing codebase, it also allows you to ignore certain kinds of problems so you can focus on fixing issues one category at a time.

NOTE

If you write C code for Python (e.g., modules), the PEP 7 standard describes the coding style that you should follow.

Tools to Catch Coding Errors

Python also has tools that check for actual coding errors rather than style errors. Here are some notable examples:

- *Pyflakes* (*https://launchpad.net/pyflakes/*): Extendable via plugins.
- *Pylint* (*https://pypi.org/project/pylint/*): Checks PEP 8 conformance while performing code error checks by default; can be extended via plugins.

These tools all make use of static analysis—that is, they parse the code and analyze it rather than running it outright.

If you choose to use Pyflakes, note that it doesn't check PEP 8 conformance on its own, so you'd need the second `pep8` tool to cover both.

To simplify things, Python has a project named `flake8` (*https://pypi.org/project/flake8/*) that combines `pyflakes` and `pep8` into a single command. It also adds some new fancy features: for example, it can skip checks on lines containing `# noqa` and is extensible via plugins.

There are a large number of plugins available for `flake8` that you can use out of the box. For example, installing *flake8-import-order* (with `pip install flake8-import-order`) will extend `flake8` so that it also checks whether your `import` statements are sorted alphabetically in your source code. Yes, some projects want that.

In most open source projects, `flake8` is heavily used for code style verification. Some large open source projects have even written their own plugins for `flake8`, adding checks for errors such as odd usage of `except`, Python 2/3 portability issues, import style, dangerous string formatting, possible localization issues, and more.

If you're starting a new project, I strongly recommend that you use one of these tools for automatic checking of your code quality and style. If you already have a codebase that didn't implement automatic code checking, a good approach is to run your tool of choice with most of the warnings disabled and fix issues one category at a time.

Though none of these tools may be a *perfect* fit for your project or your preferences, `flake8` is a good way to improve the quality of your code and make it more durable.

NOTE

Many text editors, including the famous GNU Emacs and vim, have plugins available (such as Flycheck) that can run tools such as `pep8` or `flake8` directly in your code buffer, interactively highlighting any part of your code that isn't PEP 8 compliant. This is a handy way to fix most style errors as you write your code.

We'll talk about extending this toolset in Chapter 9 with our own plugin to verify correct method declaration.

Joshua Harlow on Python

Joshua Harlow is a Python developer. He was one of the technical leads on the OpenStack team at Yahoo! between 2012 and 2016 and now

works at GoDaddy. Josh is the author of several Python libraries such as *Taskflow*, *automaton*, and *Zake*.

What got you into using Python?

I started programming in Python 2.3 or 2.4 back in about 2004 during an internship at IBM near Poughkeepsie, New York (most of my relatives and family are from upstate NY, shout out to them!). I forget exactly what I was doing there, but it involved wxPython and some Python code that they were working on to automate some system.

After that internship I returned to school, went on to graduate school at the Rochester Institute of Technology, and ended up working at Yahoo!.

I eventually ended up in the CTO team, where I and a few others were tasked with figuring out which open source cloud platform to use. We landed on OpenStack, which is written almost entirely in Python.

What do you love and hate about the Python language?

Some of the things I love (not a comprehensive listing):

- Its simplicity—Python is really easy for beginners to engage with and for experienced developers to stay engaged with.
- Style checking—reading code you wrote later on is a big part of developing software and having consistency that can be enforced by tools such as `flake8`, `pep8`, and Pylint really helps.
- The ability to pick and choose programming styles and mix them up as you see fit.

Some of the things I dislike (not a comprehensive listing):

- The somewhat painful Python 2 to 3 transition (version 3.6 has paved over most of the issues here).
- Lambdas are too simplistic and should be made more powerful.
- The lack of a decent package installer—I feel `pip` needs some work, like developing a real dependency resolver.

- The global interpreter lock (GIL) and the need for it. It makes me sad . . . [more on the GIL in Chapter 11].
- The lack of native support for multithreading—currently you need the addition of an explicit `asyncio` model.
- The fracturing of the Python community; this is mainly around the split between CPython and PyPy (and other variants).

You work on debtcollector, a Python module for managing deprecation warnings. How is the process of starting a new library?

The simplicity mentioned above makes it really easy to get a new library going and to publish it so others can use it. Since that code came out of one of the other libraries that I work on (taskflow[1]) it was relatively easy to transplant and extend that code without having to worry about the API being badly designed. I am very glad others (inside the OpenStack community or outside of it) have found a need/use for it, and I hope that library grows to accommodate more styles of deprecation patterns that other libraries (and applications?) find useful.

What is Python missing, in your opinion?

Python could perform better under just-in-time (JIT) compilation. Most newer languages being created (such as Rust, Node.js using the Chrome V8 JavaScript engine, and others) have many of Python's capabilities but are also JIT compiled. It would be really be great if the default CPython could also be JIT compiled so that Python could compete with these newer languages on performance.

Python also really needs a strong set of concurrency patterns; not just the low level `asyncio` and threading styles of patterns, but higher-level concepts that help make applications that work performantly at larger scale. The Python library `goless` does port over some of the concepts from Go, which does provide a built-in concurrency model. I believe these higher-level patterns need to be available as first-class patterns that are built in to the Standard Library and maintained so that developers can use them where they see fit. Without these, I don't see how Python can compete with other languages that do provide them.

Until next time, keep coding and be happy!

2
MODULES, LIBRARIES, AND FRAMEWORKS

Modules are an essential part of what makes Python extensible. Without them, Python would just be a language built around a monolithic interpreter; it wouldn't flourish within a giant ecosystem that allows developers to build applications quickly and simply by combining extensions. In this chapter, I'll introduce you to some of the features that make Python modules great, from the built-in modules you need to know to externally managed frameworks.

The Import System

To use modules and libraries in your programs, you have to import them using the `import` keyword. As an example, Listing 2-1 imports the all-important Zen of Python guidelines.

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
```

```
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Listing 2-1: The Zen of Python

The import system is quite complex, and I'm assuming you already know the basics, so here I'll show you some of the internals of this system, including how the `sys` module works, how to change or add import paths, and how to use custom importers.

First, you need to know that the `import` keyword is actually a wrapper around a function named `__import__`. Here is a familiar way of importing a module:

```
>>> import itertools
>>> itertools
<module 'itertools' from '/usr/.../>
```

This is precisely equivalent to this method:

```
>>> itertools = __import__("itertools")
>>> itertools
<module 'itertools' from '/usr/.../>
```

You can also imitate the `as` keyword of `import`, as these two equivalent ways of importing show:

```
>>> import itertools as it
>>> it
<module 'itertools' from '/usr/.../>
```

And here's the second example:

```
>>> it = __import__("itertools")
>>> it
<module 'itertools' from '/usr/.../>
```

While `import` is a keyword in Python, internally it's a simple function that's accessible through the `__import__` name. The `__import__` function is

extremely useful to know, as in some (corner) cases, you might want to import a module whose name is unknown beforehand, like so:

```
>>> random = __import__("RANDOM".lower())
>>> random
<module 'random' from '/usr/.../>
```

Don't forget that modules, once imported, are essentially objects whose attributes (classes, functions, variables, and so on) are objects.

The sys Module

The `sys` module provides access to variables and functions related to Python itself and the operating system it is running on. This module also contains a lot of information about Python's import system.

First of all, you can retrieve the list of modules currently imported using the `sys.modules` variable. The `sys.modules` variable is a dictionary whose key is the module name you want to inspect and whose returned value is the module object. For example, once the `os` module is imported, we can retrieve it by entering:

```
>>> import sys
>>> import os
>>> sys.modules['os']
<module 'os' from '/usr/lib/python2.7/os.pyc'>
```

The `sys.modules` variable is a standard Python dictionary that contains all loaded modules. That means that calling `sys.modules.keys()`, for example, will return the complete list of the names of loaded modules.

You can also retrieve the list of modules that are built in by using the `sys.builtin_module_names` variable. The built-in modules compiled to your interpreter can vary depending on what compilation options were passed to the Python build system.

Import Paths

When importing modules, Python relies on a list of paths to know where to look for the module. This list is stored in the `sys.path` variable.

To check which paths your interpreter will search for modules, just enter `sys.path`.

You can change this list, adding or removing paths as necessary, or even modify the `PYTHONPATH` environment variable to add paths without writing Python code at all. Adding paths to the `sys.path` variable can be useful if you want to install Python modules to nonstandard locations, such as a test environment. In normal operations, however, it should not be necessary to change the path variable. The following approaches are almost equivalent—*almost* because the path will not be placed at the same level in the list; this difference may not matter, depending on your use case:

```
>>> import sys
>>> sys.path.append('/foo/bar')
```

This would be (almost) the same as:

```
$ PYTHONPATH=/foo/bar python
>>> import sys
>>> '/foo/bar' in sys.path
True
```

It's important to note that the list will be iterated over to find the requested module, so the order of the paths in `sys.path` is important. It's useful to put the path most likely to contain the modules you are importing early in the list to speed up search time. Doing so also ensures that if two modules with the same name are available, the first match will be picked.

This last property is especially important because one common mistake is to shadow Python built-in modules with your own. Your current directory is searched before the Python Standard Library directory. That means that if you decide to name one of your scripts *random.py* and then try using `import random`, the file from your current directory will be imported rather than the Python module.

Custom Importers

You can also extend the import mechanism using custom importers. This is the technique that the Lisp-Python dialect `Hy` uses to teach Python how to import files other than standard *.py* or *.pyc* files. (`Hy` is a Lisp implementation on top of Python, discussed later in the section "A Quick Introduction to Hy" on page 145.)

The *import hook mechanism*, as this technique is called, is defined by PEP 302. It allows you to extend the standard import mechanism, which in turn allows you to modify how Python imports modules and build your own system of import. For example, you could write an extension that imports modules from a database over the network or that does some sanity checking before importing any module.

Python offers two different but related ways to broaden the import system: the meta path finders for use with `sys.meta_path` and the path entry finders for use with `sys.path_hooks`.

Meta Path Finders

The *meta path finder* is an object that will allow you to load custom objects as well as standard *.py* files. A meta path finder object must expose a `find_module(fullname, path=None)` method that returns a loader object. The loader object must also have a `load_module(fullname)` method responsible for loading the module from a source file.

To illustrate, Listing 2-2 shows how `Hy` uses a custom meta path finder to enable Python to import source files ending with *.hy* instead of *.py*.

```
class MetaImporter(object):
    def find_on_path(self, fullname):
        fls = ["%s/__init__.hy", "%s.hy"]
        dirpath = "/".join(fullname.split("."))

        for pth in sys.path:
            pth = os.path.abspath(pth)
            for fp in fls:
                composed_path = fp % ("%s/%s" % (pth, dirpath))
                if os.path.exists(composed_path):
                    return composed_path

    def find_module(self, fullname, path=None):
        path = self.find_on_path(fullname)
        if path:
```

```
            return MetaLoader(path)

sys.meta_path.append(MetaImporter())
```

Listing 2-2: A `Hy` module importer

Once Python has determined that the path is valid and that it points to a module, a `MetaLoader` object is returned, as shown in Listing 2-3.

```
class MetaLoader(object):
    def __init__(self, path):
        self.path = path

    def is_package(self, fullname):
        dirpath = "/".join(fullname.split("."))
        for pth in sys.path:
            pth = os.path.abspath(pth)
            composed_path = "%s/%s/__init__.hy" % (pth, dirpath)
            if os.path.exists(composed_path):
                return True
        return False

    def load_module(self, fullname):
        if fullname in sys.modules:
            return sys.modules[fullname]

        if not self.path:
            return

        sys.modules[fullname] = None
     ❶ mod = import_file_to_module(fullname, self.path)

        ispkg = self.is_package(fullname)

        mod.__file__ = self.path
        mod.__loader__ = self
        mod.__name__ = fullname

        if ispkg:
            mod.__path__ = []
            mod.__package__ = fullname
        else:
            mod.__package__ = fullname.rpartition('.')[0]

        sys.modules[fullname] = mod
        return mod
```

Listing 2-3: A `Hy` module loader object

At ❶, `import_file_to_module` reads a *.hy* source file, compiles it to Python code, and returns a Python module object.

This loader is pretty straightforward: once the *.hy* file is found, it's passed to this loader, which compiles the file if necessary, registers it, sets some attributes, and then returns it to the Python interpreter.

The `uprefix` module is another good example of this feature in action. Python 3.0 through 3.2 didn't support the `u` prefix for denoting Unicode strings that was featured in Python 2; the `uprefix` module ensures compatibility between Python versions 2 and 3 by removing the `u` prefix from strings before compilation.

Useful Standard Libraries

Python comes with a huge standard library packed with tools and features for almost any purpose you can think of. Newcomers to Python who are used to having to write their own functions for basic tasks are often shocked to find that the language itself ships with so much functionality built in and ready for use.

Whenever you're tempted to write your own function to handle a simple task, first stop and look through the standard library. In fact, skim through the whole thing at least once before you begin working with Python so that next time you need a function, you have an idea of whether it already exists in the standard library.

We'll talk about some of these modules, such as `functools` and `itertools`, in later chapters, but here are a few of the standard modules that you'll definitely find useful:

- `atexit` allows you to register functions for your program to call when it exits.
- `argparse` provides functions for parsing command line arguments.
- `bisect` provides bisection algorithms for sorting lists (see Chapter 10).
- `calendar` provides a number of date-related functions.
- `codecs` provides functions for encoding and decoding data.
- `collections` provides a variety of useful data structures.

- `copy` provides functions for copying data.
- `csv` provides functions for reading and writing CSV files.
- `datetime` provides classes for handling dates and times.
- `fnmatch` provides functions for matching Unix-style filename patterns.
- `concurrent` provides asynchronous computation (native in Python 3, available for Python 2 via PyPI).
- `glob` provides functions for matching Unix-style path patterns.
- `io` provides functions for handling I/O streams. In Python 3, it also contains StringIO (inside the module of the same name in Python 2), which allows you to treat strings as files.
- `json` provides functions for reading and writing data in JSON format.
- `logging` provides access to Python's own built-in logging functionality.
- `multiprocessing` allows you to run multiple subprocesses from your application, while providing an API that makes them look like threads.
- `operator` provides functions implementing the basic Python operators, which you can use instead of having to write your own lambda expressions (see Chapter 10).
- `os` provides access to basic OS functions.
- `random` provides functions for generating pseudorandom numbers.
- `re` provides regular expression functionality.
- `sched` provides an event scheduler without using multithreading.
- `select` provides access to the `select()` and `poll()` functions for creating event loops.
- `shutil` provides access to high-level file functions.
- `signal` provides functions for handling POSIX signals.
- `tempfile` provides functions for creating temporary files and directories.
- `threading` provides access to high-level threading functionality.

- `urllib` (and `urllib2` and `urlparse` in Python 2.*x*) provides functions for handling and parsing URLs.
- `uuid` allows you to generate Universally Unique Identifiers (UUIDs).

Use this list as a quick reference for what these useful libraries modules do. If you can memorize even part of this list, all the better. The less time you have to spend looking up library modules, the more time you can spend writing the code you actually need.

Most of the standard library is written in Python, so there's nothing stopping you from looking at the source code of the modules and functions. When in doubt, crack open the code and see what it does for yourself. Even if the documentation has everything you need to know, there's always a chance you could learn something useful.

External Libraries

Python's "batteries included" philosophy is that, once you have Python installed, you should have everything you need to build whatever you want. This is to prevent the programming equivalent of unwrapping an awesome gift only to find out that whoever gave it to you forgot to buy batteries for it.

Unfortunately, there's no way the people behind Python can predict *everything* you might want to make. And even if they could, most people wouldn't want to deal with a multigigabyte download, especially if they just wanted to write a quick script for renaming files. So even with its extensive functionality, the Python Standard Library doesn't cover everything. Luckily, members of the Python community have created external libraries.

The Python Standard Library is safe, well-charted territory: its modules are heavily documented, and enough people use it on a regular basis that you can feel assured it won't break messily when you give it a try—and in the unlikely event that it *does* break, you can be confident someone will fix it in short order. External libraries, on the other hand,

are the parts of the map labeled "here there be dragons": documentation may be sparse, functionality may be buggy, and updates may be sporadic or even nonexistent. Any serious project will likely need functionality that only external libraries can provide, but you need to be mindful of the risks involved in using them.

Here's a tale of external library dangers from the trenches. OpenStack uses SQLAlchemy, a database toolkit for Python. If you're familiar with SQL, you know that database schemas can change over time, so OpenStack also made use of `sqlalchemy-migrate` to handle schema migration needs. And it worked . . . until it didn't. Bugs started piling up, and nothing was getting done about them. At this time, OpenStack was also interested in supporting Python 3, but there was no sign that `sqlalchemy-migrate` was moving toward Python 3 support. It was clear by that point that `sqlalchemy-migrate` was effectively dead for our needs and we needed to switch to something else—our needs had outlived the capabilities of the external library. At the time of this writing, OpenStack projects are migrating toward using Alembic instead, a new SQL database migrations tool with Python 3 support. This is happening not without some effort, but fortunately without much pain.

The External Libraries Safety Checklist

All of this builds up to one important question: how can you be sure you won't fall into this external libraries trap? Unfortunately, you can't: programmers are people, too, and there's no way you can know for sure whether a library that's zealously maintained today will still be in good shape in a few months. However, using such libraries may be worth the risk; it's just important to carefully assess your situation. At OpenStack, we use the following checklist when choosing whether to use an external library, and I encourage you to do the same.

Python 3 compatibility Even if you're not targeting Python 3 right now, odds are good that you will somewhere down the line, so it's a good idea to check that your chosen library is already Python 3–compatible and committed to staying that way.

Active development GitHub and Ohloh usually provide enough information to determine whether a given library is being actively developed by its maintainers.

Active maintenance Even if a library is considered finished (that is, feature complete), the maintainers should be ensuring it remains bug-free. Check the project's tracking system to see how quickly the maintainers respond to bugs.

Packaged with OS distributions If a library is packaged with major Linux distributions, that means other projects are depending on it—so if something goes wrong, you won't be the only one complaining. It's also a good idea to check this if you plan to release your software to the public: your code will be easier to distribute if its dependencies are already installed on the end user's machine.

API compatibility commitment Nothing's worse than having your software suddenly break because a library it depends on has changed its entire API. You might want to check whether your chosen library has had anything like this happen in the past.

License You need to make sure that the license is compatible with the software you're planning to write and that it allows you to do whatever you intend to do with your code in terms of distribution, modification, and execution.

Applying this checklist to dependencies is also a good idea, though that could turn out to be a huge undertaking. As a compromise, if you know your application is going to depend heavily on a particular library, you should apply this checklist to each of that library's dependencies.

Protecting Your Code with an API Wrapper

No matter what libraries you end up using, you need to treat them as useful devices that could potentially do some serious damage. For safety, libraries should be treated like any physical tool: kept in your tool shed, away from your fragile valuables but available when you actually need them.

No matter how useful an external library might be, be wary of letting it get its hooks into your actual source code. Otherwise, if something goes wrong and you need to switch libraries, you might have to rewrite huge swaths of your program. A better idea is to write your own API—a wrapper that encapsulates your external libraries and keeps them out of your source code. Your program never has to know what external libraries it's using, only what functionality your API provides. Then, if you need to use a different library, all you have to change is your wrapper. As long as the new library provides the same functionality, you won't have to touch the rest of your codebase at all. There might be exceptions, but probably not many; most libraries are designed to solve a tightly focused range of problems and can therefore be easily isolated.

Later in Chapter 5, we'll also look at how you can use entry points to build driver systems that will allow you to treat parts of your projects as modules you can switch out at will.

Package Installation: Getting More from pip

The `pip` project offers a really simple way to handle package and external library installations. It is actively developed, well maintained, and included with Python starting at version 3.4. It can install or uninstall packages from the *Python Packaging Index (PyPI)*, a tarball, or a `Wheel` archive (we'll discuss these in Chapter 5).

Its usage is simple:

```
$ pip install --user voluptuous
Downloading/unpacking voluptuous
  Downloading voluptuous-0.8.3.tar.gz
  Storing download in cache at ./.cache/pip/https%3A%2F%2Fpypi.python.org%2Fpa
ckages%2Fsource%2Fv%2Fvoluptuous%2Fvoluptuous-0.8.3.tar.gz
  Running setup.py egg_info for package voluptuous

Requirement already satisfied (use --upgrade to upgrade): distribute in /usr/
lib/python2.7/dist-packages (from voluptuous)
Installing collected packages: voluptuous
  Running setup.py install for voluptuous

Successfully installed voluptuous
Cleaning up...
```

By looking it up on the PyPI distribution index, where anyone can upload a package for distribution and installation by others, `pip install` can install any package.

You can also provide a `--user` option that makes `pip` install the package in your home directory. This avoids polluting your operating system directories with packages installed system-wide.

You can list the packages you already have installed using the `pip freeze` command, like so:

```
$ pip freeze
Babel==1.3
Jinja2==2.7.1
commando=0.3.4
--snip--
```

Uninstalling packages is also supported by `pip`, using the `uninstall` command:

```
$ pip uninstall pika-pool
Uninstalling pika-pool-0.1.3:
  /usr/local/lib/python2.7/site-packages/pika_pool-0.1.3.dist-info/
DESCRIPTION.rst
  /usr/local/lib/python2.7/site-packages/pika_pool-0.1.3.dist-info/INSTALLER
  /usr/local/lib/python2.7/site-packages/pika_pool-0.1.3.dist-info/METADATA

--snip--
Proceed (y/n)? y
  Successfully uninstalled pika-pool-0.1.3
```

One very valuable feature of `pip` is its ability to install a package without copying the package's file. The typical use case for this feature is when you're actively working on a package and want to avoid the long and boring process of reinstalling it each time you need to test a change. This can be achieved by using the `-e` *`<directory>`* flag:

```
$ pip install -e .
Obtaining file:///Users/jd/Source/daiquiri
Installing collected packages: daiquiri
  Running setup.py develop for daiquiri
Successfully installed daiquiri
```

Here, `pip` does not copy the files from the local source directory but places a special file, called an `egg-link`, in your distribution path. For

example:

```
$ cat /usr/local/lib/python2.7/site-packages/daiquiri.egg-link
/Users/jd/Source/daiquiri
```

The `egg-link` file contains the path to add to `sys.path` to look for packages. The result can be easily checked by running the following command:

```
$ python -c "import sys; print('/Users/jd/Source/daiquiri' in sys.path)"
True
```

Another useful `pip` tool is the `-e` option of `pip install`, helpful for deploying code from repositories of various version control systems: git, Mercurial, Subversion, and even Bazaar are supported. For example, you can install any library directly from a git repository by passing its address as a URL after the `-e` option:

```
$ pip install -e git+https://github.com/jd/daiquiri.git\#egg=daiquiri
Obtaining daiquiri from git+https://github.com/jd/daiquiri.git#egg=daiquiri
  Cloning https://github.com/jd/daiquiri.git to ./src/daiquiri
Installing collected packages: daiquiri
  Running setup.py develop for daiquiri
Successfully installed daiquiri
```

For the installation to work correctly, you need to provide the package egg name by adding `#egg=` at the end of the URL. Then, `pip` just uses `git clone` to clone the repository inside a `src/`*`<eggname>`* and creates an `egg-link` file pointing to that same cloned directory.

This mechanism is extremely handy when depending on unreleased versions of libraries or when working in a continuous testing system. However, since there is no versioning behind it, the `-e` option can also be very nasty. You cannot know in advance that the next commit in this remote repository is not going to break everything.

Finally, all other installation tools are being deprecated in favor of `pip`, so you can confidently treat it as your one-stop shop for all your package management needs.

Using and Choosing Frameworks

Python has a variety of frameworks available for various kinds of Python applications: if you're writing a web application, you could use Django, Pylons, TurboGears, Tornado, Zope, or Plone; if you're looking for an event-driven framework, you could use Twisted or Circuits; and so on.

The main difference between frameworks and external libraries is that applications use frameworks by building on top of them: your code will extend the framework rather than vice versa. Unlike a library, which is basically an add-on you can bring in to give your code some extra oomph, a framework forms the *chassis* of your code: everything you do builds on that chassis in some way. This can be a double-edged sword. There are plenty of upsides to using frameworks, such as rapid prototyping and development, but there are also some noteworthy downsides, such as lock-in. You need to take these considerations into account when you decide whether to use a framework.

The recommendations for what to check when choosing the right framework for your Python application are largely the same as those described in "The External Libraries Safety Checklist" on page 23—which makes sense, as frameworks are distributed as bundles of Python libraries. Sometimes frameworks also include tools for creating, running, and deploying applications, but that doesn't change the criteria you should apply. We've established that replacing an external library after you've already written code that makes use of it is a pain, but replacing a framework is a thousand times worse, usually requiring a complete rewrite of your program from the ground up.

To give an example, the Twisted framework mentioned earlier still doesn't have full Python 3 support: if you wrote a program using Twisted a few years back and wanted to update it to run on Python 3, you'd be out of luck. Either you'd have to rewrite your entire program to use a different framework, or you'd have to wait until someone finally gets around to upgrading Twisted with full Python 3 support.

Some frameworks are lighter than others. For example, Django has its own built-in ORM functionality; Flask, on the other hand, has nothing of the sort. The *less* a framework tries to do for you, the fewer problems you'll have with it in the future. However, each feature a framework lacks is another problem for you to solve, either by writing

your own code or going through the hassle of handpicking another library to handle it. It's your choice which scenario you'd rather deal with, but choose wisely: migrating away from a framework when things go sour can be a Herculean task, and even with all its other features, there's nothing in Python that can help you with that.

Doug Hellmann, Python Core Developer, on Python Libraries

Doug Hellmann is a senior developer at DreamHost and a fellow contributor to the OpenStack project. He launched the website Python Module of the Week (*http://www.pymotw.com/*) and has written an excellent book called *The Python Standard Library by Example*. He is also a Python core developer. I've asked Doug a few questions about the Standard Library and designing libraries and applications around it.

When you start writing a Python application from scratch, what's your first move?

The steps for writing an application from scratch are similar to hacking an existing application, in the abstract, but the details change.

When I change existing code, I start by figuring out how it works and where my changes would need to go. I may use some debugging techniques: adding logging or print statements, or using `pdb`, and running the app with test data to make sure I understand what it's doing. I usually make the change and test it by hand, then add any automated tests before contributing a patch.

I take the same exploratory approach when I create a new application —create some code and run it by hand, and then once I have the basic functionality working, I write tests to make sure I've covered all of the edge cases. Creating the tests may also lead to some refactoring to make the code easier to work with.

That was definitely the case with smiley [a tool for spying on your Python programs and recording their activities]. I started by experimenting with Python's trace API, using some throwaway scripts,

before building the real application. Originally, I planned to have one piece to instrument and collect data from another running application, and another to collect the data sent over the network and save it. While adding a couple of reporting features, I realized that the processing for replaying the collected data was almost identical to the processing for collecting it in the first place. I refactored a few classes and was able to create a base class for the data collection, database access, and report generator. Making those classes conform to the same API allowed me to easily create a version of the data collection app that wrote directly to the database instead of sending information over the network.

While designing an app, I think about how the user interface works, but for libraries, I focus on how a developer will use the API. It can also be easier to write the tests for programs that will use the new library first, then the library code. I usually create a series of example programs in the form of tests and then build the library to work that way.

I've also found that writing documentation for a library before writing any code helps me think through the features and workflows without committing to the implementation details, and it lets me record the choices I made in the design so the reader understands not just how to use the library but the expectations I had while creating it.

What's the process for getting a module into the Python Standard Library?

The full process and guidelines for submitting a module into the standard library can be found in the Python Developer's Guide at *https://docs.python.org/devguide/stdlibchanges.html*.

Before a module can be added, the submitter needs to prove that it's stable and widely useful. The module should provide something that is either hard to implement correctly on your own or so useful that many developers have created their own variations. The API should be clear, and any module dependencies should be inside the Standard Library only.

The first step would be to run the idea of introducing the module into the standard library by the community via the *python-ideas* list to informally gauge the level of interest. Assuming the response is positive,

the next step is to create a Python Enhancement Proposal (PEP), which should include the motivation for adding the module and implementation details of how the transition will happen.

Because package management and discovery tools have become so reliable, especially `pip` and the PyPI, it may be more practical to maintain a new library outside of the Python Standard Library. A separate release allows for more frequent updates with new features and bug fixes, which can be especially important for libraries addressing new technologies or APIs.

What are the top three modules from the Standard Library that you wish people knew more about?

One really useful tool from the Standard Library is the `abc` module. I use the `abc` module to define the APIs for dynamically loaded extensions as abstract base classes, to help extension authors understand which methods of the API are required and which are optional. Abstract base classes are built into some other OOP [object-oriented programming] languages, but I've found a lot of Python programmers don't know we have them as well.

The binary search algorithm in the `bisect` module is a good example of a useful feature that's often implemented incorrectly, which makes it a great fit for the Standard Library. I especially like the fact that it can search sparse lists where the search value may not be included in the data.

There are some useful data structures in the `collections` module that aren't used as often as they could be. I like to use `namedtuple` for creating small, class-like data structures that need to hold data without any associated logic. It's very easy to convert from a `namedtuple` to a regular class if logic does need to be added later, since `namedtuple` supports accessing attributes by name. Another interesting data structure from the module is `ChainMap`, which makes a good stackable namespace. `ChainMap` can be used to create contexts for rendering templates or managing configuration settings from different sources with clearly defined precedence.

A lot of projects, including OpenStack and external libraries, roll their own abstractions on top of the Standard Library, like for date/time handling, for example. In your opinion, should programmers stick to the Standard Library, roll their own functions, switch to some external library, or start sending patches to Python?

All of the above! I prefer to avoid reinventing the wheel, so I advocate strongly for contributing fixes and enhancements upstream to projects that can be used as dependencies. On the other hand, sometimes it makes sense to create another abstraction and maintain that code separately, either within an application or as a new library.

The `timeutils` module, used in your example, is a fairly thin wrapper around Python's `datetime` module. Most of the functions are short and simple, but creating a module with the most common operations ensures they're handled consistently throughout all projects. Because a lot of the functions are application specific, in the sense that they enforce decisions about things like timestamp format strings or what "now" means, they are not good candidates for patches to Python's library or to be released as a general purpose library and adopted by other projects.

In contrast, I have been working to move the API services in OpenStack away from the WSGI [Web Server Gateway Interface] framework created in the early days of the project and onto a third-party web development framework. There are a lot of options for creating WSGI applications in Python, and while we may need to enhance one to make it completely suitable for OpenStack's API servers, contributing those reusable changes upstream is preferable to maintaining a "private" framework.

What would your advice be to developers hesitating between major Python versions?

The number of third-party libraries supporting Python 3 has reached critical mass. It's easier than ever to build new libraries and applications for Python 3, and thanks to the compatibility features added to 3.3, maintaining support for Python 2.7 is also easier. The major Linux

distributions are working on shipping releases with Python 3 installed by default. Anyone starting a new project in Python should look seriously at Python 3, unless they have a dependency that hasn't been ported. At this point, though, libraries that don't run on Python 3 could almost be classified as "unmaintained."

What are the best ways to branch code out from an application into a library in terms of design, planning ahead, migration, etc.?

Applications are collections of "glue code" holding libraries together for a specific purpose. Designing your application with the features to achieve that purpose as a library first and then building the application ensures that code is properly organized into logical units, which in turn makes testing simpler. It also means the features of an application are accessible through the library and can be remixed to create other applications. If you don't take this approach, you risk the features of the application being tightly bound to the user interface, which makes them harder to modify and reuse.

What advice would you give to people planning to design their own Python libraries?

I always recommend designing libraries and APIs from the top down, applying design criteria such as the Single Responsibility Principle (SRP) at each layer. Think about what the caller will want to do with the library and create an API that supports those features. Think about what values can be stored in an instance and used by the methods versus what needs to be passed to each method every time. Finally, think about the implementation and whether the underlying code should be organized differently than the code of the public API.

SQLAlchemy is an excellent example of applying those guidelines. The declarative ORM [object relational mapping], data mapping, and expression generation layers are all separate. A developer can decide the right level of abstraction for entering the API and using the library based on their needs rather than constraints imposed by the library's design.

What are the most common programming errors you encounter while reading Python developers' code?

One area where Python's idioms are significantly different from other languages is in looping and iteration. For example, one of the most common anti-patterns I see is the use of a `for` loop to filter a list by first appending items to a new list and then processing the result in a second loop (possibly after passing the list as an argument to a function). I almost always suggest converting filtering loops like these into generator expressions, which are more efficient and easier to understand. It's also common to see lists being combined so their contents can be processed together in some way, rather than using `itertools.chain()`.

There are other, more subtle things I often suggest in code reviews, like using a `dict()` as a lookup table instead of a long `if:then:else` block, making sure functions always return the same type of object (for example, an empty list instead of `None`), reducing the number of arguments a function requires by combining related values into an object with either a tuple or a new class, and defining classes to use in public APIs instead of relying on dictionaries.

What's your take on frameworks?

Frameworks are like any other kind of tool. They can help, but you need to take care when choosing one to make sure that it's right for the job at hand.

Pulling out the common parts of your app into a framework helps you focus your development efforts on the unique aspects of an application. Frameworks also provide a lot of bootstrapping code, for doing things like running in development mode and writing a test suite, that helps you bring an application to a useful state more quickly. They also encourage consistency in the implementation of the application, which means you end up with code that is easier to understand and more reusable.

There are some potential pitfalls too, though. The decision to use a particular framework usually implies something about the design of the application itself. Selecting the wrong framework can make an

application harder to implement if those design constraints do not align naturally with the application's requirements. You may end up fighting with the framework if you try to use patterns or idioms that differ from what it recommends.

3
DOCUMENTATION AND GOOD API PRACTICE

In this chapter, we'll discuss documentation; specifically, how to automate the trickier and more tedious aspects of documenting your project with *Sphinx*. While you will still have to write the documentation yourself, Sphinx will simplify your task. As it is common to provide features using a Python library, we'll also look at how to manage and document your public API changes. Because your API will have to evolve as you make changes to its features, it's rare to get everything built perfectly from the outset, but I'll show you a few things you can do to ensure your API is as user-friendly as possible.

We'll end this chapter with an interview with Christophe de Vienne, author of the Web Services Made Easy framework, in which he discusses best practices for developing and maintaining APIs.

Documenting with Sphinx

Documentation is one of the most important parts of writing software. Unfortunately, a lot of projects don't provide proper documentation. Writing documentation is seen as complicated and daunting, but it doesn't have to be: with the tools available to Python programmers, documenting your code can be just as easy as writing it.

One of the biggest reasons for sparse or nonexistent documentation is that many people assume the only way to document code is by hand.

Even with multiple people on a project, this means one or more of your team will end up having to juggle contributing code with maintaining documentation—and if you ask any developer which job they'd prefer, you can be sure they'll say they'd rather write software than write *about* software.

Sometimes the documentation process is completely separate from the development process, meaning that the documentation is written by people who did not write the actual code. Furthermore, any documentation produced this way is likely to be out-of-date: it's almost impossible for manual documentation to keep up with the pace of development, regardless of who handles it.

Here's the bottom line: the more degrees of separation between your code and your documentation, the harder it will be to keep the latter properly maintained. So why keep them separate at all? It's not only possible to put your documentation directly in the code itself, but it's also simple to convert that documentation into easy-to-read HTML and PDF files.

The most common format for Python documentation is *reStructuredText*, or *reST* for short. It's a lightweight markup language (like Markdown) that's as easy to read and write for humans as it is for computers. Sphinx is the most commonly used tool for working with this format; Sphinx can read reST-formatted content and output documentation in a variety of other formats.

I recommend that your project documentation always include the following:

- The problem your project is intended to solve, in one or two sentences.
- The license your project is distributed under. If your software is open source, you should also include this information in a header in each code file; just because you've uploaded your code to the Internet doesn't mean that people will know what they're allowed to do with it.
- A small example of how your code works.

- Installation instructions.
- Links to community support, mailing list, IRC, forums, and so on.
- A link to your bug tracker system.
- A link to your source code so that developers can download and start delving into it right away.

You should also include a *README.rst* file that explains what your project does. This README should be displayed on your GitHub or PyPI project page; both sites know how to handle reST formatting.

> **NOTE**
>
> *If you're using GitHub, you can also add a* CONTRIBUTING.rst *file that will be displayed when someone submits a pull request. It should provide a checklist for users to follow before they submit the request, including things like whether your code follows PEP 8 and reminders to run the unit tests. Read the Docs* (http://readthedocs.org/) *allows you to build and publish your documentation online automatically. Signing up and configuring a project is straightforward. Then Read the Docs searches for your Sphinx configuration file, builds your documentation, and makes it available for your users to access. It's a great companion to code-hosting sites.*

Getting Started with Sphinx and reST

You can get Sphinx from *http://www.sphinx-doc.org/*. There are installation instructions on the site, but the easiest method is to install with `pip install sphinx`.

Once Sphinx is installed, run `sphinx-quickstart` in your project's top-level directory. This will create the directory structure that Sphinx expects to find, along with two files in the *doc/source* folder: *conf.py*, which contains Sphinx's configuration settings (and is absolutely required for Sphinx to work), and *index.rst*, which serves as the front page of your documentation. Once you run the quick-start command,

you'll be taken through a series of steps to designate naming conventions, version conventions, and options for other useful tools and standards.

The *conf.py* file contains a few documented variables, such as the project name, the author, and the theme to use for HTML output. Feel free to edit this file at your convenience.

Once you've built your structure and set your defaults, you can build your documentation in HTML by calling `sphinx-build` with your source directory and output directory as arguments, as shown in Listing 3-1. The command `sphinx-build` reads the *conf.py* file from the source directory and parses all the *.rst* files from this directory. It renders them in HTML in the output directory.

```
$ sphinx-build doc/source doc/build
  import pkg_resources
Running Sphinx v1.2b1
loading pickled environment... done
No builder selected, using default: html
building [html]: targets for 1 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
preparing documents... done
writing output... [100%] index
writing additional files... genindex search
copying static files... done
dumping search index... done
dumping object inventory... done
build succeeded.
```

Listing 3-1: Building a basic Sphinx HTML document

Now you can open *doc/build/index.html* in your favorite browser and read your documentation.

NOTE

If you're using `setuptools` or `pbr` (see Chapter 5) for packaging, Sphinx extends them to support the command `setup.py build_sphinx`, which will run `sphinx-build` automatically. The `pbr` integration of Sphinx has some saner defaults, such as outputting the documentation in the /doc *subdirectory.*

Your documentation begins with the *index.rst* file, but it doesn't have to end there: reST supports `include` directives to include reST files from other reST files, so there's nothing stopping you from dividing your documentation into multiple files. Don't worry too much about syntax and semantics to start; reST offers a lot of formatting possibilities, but you'll have plenty of time to dive into the reference later. The complete reference (*http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html*) explains how to create titles, bulleted lists, tables, and more.

Sphinx Modules

Sphinx is highly extensible: its basic functionality supports only manual documentation, but it comes with a number of useful modules that enable automatic documentation and other features. For example, `sphinx.ext.autodoc` extracts reST-formatted docstrings from your modules and generates *.rst* files for inclusion. This is one of the options `sphinx-quickstart` will ask if you want to activate. If you didn't select that option, however, you can still edit your *conf.py* file and add it as an extension like so:

```
extensions = ['sphinx.ext.autodoc']
```

Note that `autodoc` will *not* automatically recognize and include your modules. You need to explicitly indicate which modules you want documented by adding something like Listing 3-2 to one of your *.rst* files.

```
   .. automodule:: foobar
❶     :members:
❷     :undoc-members:
❸     :show-inheritance:
```

Listing 3-2: Indicating the modules for `autodoc` to document

In Listing 3-2, we make three requests, all of which are optional: that all documented members be printed ❶, that all undocumented members be printed ❷, and that inheritance be shown ❸. Also note the following:

- If you don't include any directives, Sphinx won't generate any output.
- If you only specify `:members:`, undocumented nodes on your module, class, or method tree will be skipped, even if all their members are documented. For example, if you document the methods of a class but not the class itself, `:members:` will exclude both the class and its methods. To keep this from happening, you'd have to write a docstring for the class or specify `:undoc-members:` as well.
- Your module needs to be where Python can import it. Adding `.`, `..`, and/or `../..` to `sys.path` can help.

The `autodoc` extension gives you the power to include most of your documentation in your source code. You can even pick and choose which modules and methods to document—it's not an "all-or-nothing" solution. By maintaining your documentation directly alongside your source code, you can easily ensure it stays up to date.

Automating the Table of Contents with autosummary

If you're writing a Python library, you'll usually want to format your API documentation with a table of contents containing links to individual pages for each module.

The `sphinx.ext.autosummary` module was created specifically to handle this common use case. First, you need to enable it in your *conf.py* by adding the following line:

```
extensions = ['sphinx.ext.autosummary']
```

Then, you can add something like the following to an *.rst* file to automatically generate a table of contents for the specified modules:

```
.. autosummary::

   mymodule
   mymodule.submodule
```

This will create files called *generated/mymodule.rst* and *generated/mymodule.submodule.rst* containing the autodoc directives

described earlier. Using this same format, you can specify which parts of your module API you want included in your documentation.

> **NOTE**
>
> *The `sphinx-apidoc` command can automatically create these files for you; check out the Sphinx documentation to find out more.*

Automating Testing with doctest

Another useful feature of Sphinx is the ability to run `doctest` on your examples automatically when you build your documentation. The standard Python `doctest` module searches your documentation for code snippets and tests whether they accurately reflect what your code does. Every paragraph starting with the primary prompt `>>>` is treated as a code snippet to test. For example, if you wanted to document the standard `print` function from Python, you could write this documentation snippet and `doctest` would check the result:

```
    To print something to the standard output, use the :py:func:`print`
function:
>>> print("foobar")
    foobar
```

Having such examples in your documentation lets users understand your API. However, it's easy to put off and eventually forget to update your examples as your API evolves. Fortunately, `doctest` helps make sure this doesn't happen. If your documentation includes a step-by-step tutorial, `doctest` will help you keep it up to date throughout development by testing every line it can.

You can also use `doctest` for *documentation-driven development (DDD)*: write your documentation and examples first and then write code to match your documentation. Taking advantage of this feature is as simple as running `sphinx-build` with the special `doctest` builder, like this:

```
$ sphinx-build -b doctest doc/source doc/build
Running Sphinx v1.2b1
loading pickled environment... done
building [doctest]: targets for 1 source files that are out of date
```

```
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
running tests...

Document: index
--------------
1 items passed all tests:
   1 tests in default
1 tests in 1 items.
1 passed and 0 failed.
Test passed.

Doctest summary
===============
    1 test
    0 failures in tests
    0 failures in setup code
    0 failures in cleanup code
build succeeded.
```

When using the `doctest` builder, Sphinx reads the usual *.rst* files and executes code examples that are contained in those files.

Sphinx also provides a bevy of other features, either out of the box or through extension modules, including these:

- Linking between projects
- HTML themes
- Diagrams and formulas
- Output to Texinfo and EPUB format
- Linking to external documentation

You might not need all this functionality right away, but if you ever need it in the future, it's good to know about in advance. Again, check out the full Sphinx documentation to find out more.

Writing a Sphinx Extension

Sometimes off-the-shelf solutions just aren't enough and you need to create custom tools to deal with a situation.

Say you're writing an HTTP REST API. Sphinx will only document the Python side of your API, forcing you to write your REST API documentation by hand, with all the problems that entails. The creators

of Web Services Made Easy (WSME) (interviewed at the end of this chapter) have come up with a solution: a Sphinx extension called `sphinxcontrib-pecanwsme` that analyzes docstrings and actual Python code to generate REST API documentation automatically.

> **NOTE**
>
> *For other HTTP frameworks, such as Flask, Bottle, and Tornado, you can use `sphinxcontrib.httpdomain`.*

My point is that whenever you know you could extract information from your code to build documentation, you should, and you should also automate the process. This is better than trying to maintain manually written documentation, especially when you can leverage auto-publication tools such as Read the Docs.

We'll examine the `sphinxcontrib-pecanwsme` extension as an example of writing your own Sphinx extension. The first step is to write a module—preferably as a submodule of `sphinxcontrib`, as long as your module is generic enough—and pick a name for it. Sphinx requires this module to have one predefined function called `setup(app)`, which contains the methods you'll use to connect your code to Sphinx events and directives. The full list of methods is available in the Sphinx extension API at *http://www.sphinx-doc.org/en/master/extdev/appapi.html*.

For example, the `sphinxcontrib-pecanwsme` extension includes a single directive called `rest-controller`, added using the `setup(app)` function. This added directive needs a fully qualified controller class name to generate documentation for, as shown in Listing 3-3.

```
def setup(app):
    app.add_directive('rest-controller', RESTControllerDirective)
```

Listing 3-3: Code from `sphinxcontrib.pecanwsme.rest.setup` that adds the `rest-controller` directive

The `add_directive` method in Listing 3-3 registers the `rest-controller` directive and delegates its handling to the `RESTControllerDirective` class. This `RESTControllerDirective` class exposes certain attributes that indicate

how the directive treats content, whether it has arguments, and so on. The class also implements a `run()` method that actually extracts the documentation from your code and returns parsed data to Sphinx.

The repository at *https://bitbucket.org/birkenfeld/sphinx-contrib/src/* has many small modules that can help you develop your own extensions.

> **NOTE**
>
> *Even though Sphinx is written in Python and targets it by default, extensions are available that allow it to support other languages as well. You can use Sphinx to document your project in full, even if it uses multiple languages at once.*

As another example, in one of my projects named Gnocchi—a database for storing and indexing time series data at a large scale—I've used a custom Sphinx extension to autogenerate documentation. Gnocchi provides a REST API, and usually to document such an API, projects will manually write examples of what an API request and its response should look like. Unfortunately, this approach is error prone and out of sync with reality.

Using the unit-testing code available to test the Gnocchi API, we built a Sphinx extension to run Gnocchi and generate an *.rst* file containing HTTP requests and responses run against a real Gnocchi server. In this way, we ensure the documentation is up to date: the server responses are not manually crafted, and if a manually written request fails, then the documentation process fails, and we know that we must fix the documentation.

Including that code in the book would be too verbose, but you can check the sources of Gnocchi online and look at the `gnocchi.gendoc` module to get an idea of how it works.

Managing Changes to Your APIs

Well-documented code is a sign to other developers that the code is suitable to be imported and used to build something else. When

building a library and exporting an API for other developers to use, for example, you want to provide the reassurance of solid documentation.

This section will cover best practices for public APIs. These will be exposed to users of your library or application, and while you can do whatever you like with internal APIs, public APIs should be handled with care.

To distinguish between public and private APIs, the Python convention is to prefix the symbol for a private API with an underscore: `foo` is public, but `_bar` is private. You should use this convention both to recognize whether another API is public or private and to name your own APIs. In contrast to other languages, such as Java, Python does not enforce any restriction on accessing code marked as private or public. The naming conventions are just to facilitate understanding among programmers.

Numbering API Versions

When properly constructed, the version number of an API can give users a great deal of information. Python has no particular system or convention in place for numbering API versions, but we can take inspiration from Unix platforms, which use a complex management system for libraries with fine-grained version identifiers.

Generally, your version numbering should reflect changes in the API that will impact users. For example, when the API has a major change, the major version number might change from 1 to 2. When only a few new API calls are added, the lesser number might go from 2.2 to 2.3. If a change only involves bug fixes, the version might bump from 2.2.0 to 2.2.1. A good example of how to use version numbering is the Python `requests` library (*https://pypi.python.org/pypi/requests/*). This library increments its API numbers based on the number of changes in each new version and the impact the changes might have on consuming programs.

Version numbers hint to developers that they should look at changes between two releases of a library, but alone they are not enough to fully

guide a developer: you must provide detailed documentation to describe those changes.

Documenting Your API Changes

Whenever you make changes to an API, the first and most important thing to do is to heavily document them so that a consumer of your code can get a quick overview of what's changing. Your document should cover the following:

- New elements of the new interface
- Elements of the old interface that are deprecated
- Instructions on how to migrate to the new interface

You should also make sure that you don't remove the old interface right away. I recommend keeping the old interface until it becomes too much trouble to do so. If you have marked it as deprecated, users will know not to use it.

Listing 3-4 is an example of good API change documentation for code that provides a representation of a car object that can turn in any direction. For whatever reason, the developers decided to retract the `turn_left` method and instead provide a generic `turn` method that can take the direction as an argument.

```
class Car(object):

    def turn_left(self):
        """Turn the car left.

        .. deprecated:: 1.1
           Use :func:`turn` instead with the direction argument set to left
        """
        self.turn(direction='left')

    def turn(self, direction):
        """Turn the car in some direction.

        :param direction: The direction to turn to.
        :type direction: str
        """
        # Write actual code for the turn function here instead
        pass
```

Listing 3-4: An example of API change documentation for a car object

The triple quotes here, `"""`, indicate the start and end of the docstrings, which will be pulled into the documentation when the user enters `help(Car.turn_left)` into the terminal or extracts the documentation with an external tool such as Sphinx. The deprecation of the `car.turn_left` method is indicated by `.. deprecated 1.1`, where `1.1` refers to the first version released that ships this code as deprecated.

Using this deprecation method and making it visible via Sphinx clearly tells users that the function should not be used and gives them direct access to the new function along with an explanation of how to migrate old code.

Figure 3-1 shows Sphinx documentation that explains some deprecated functions.

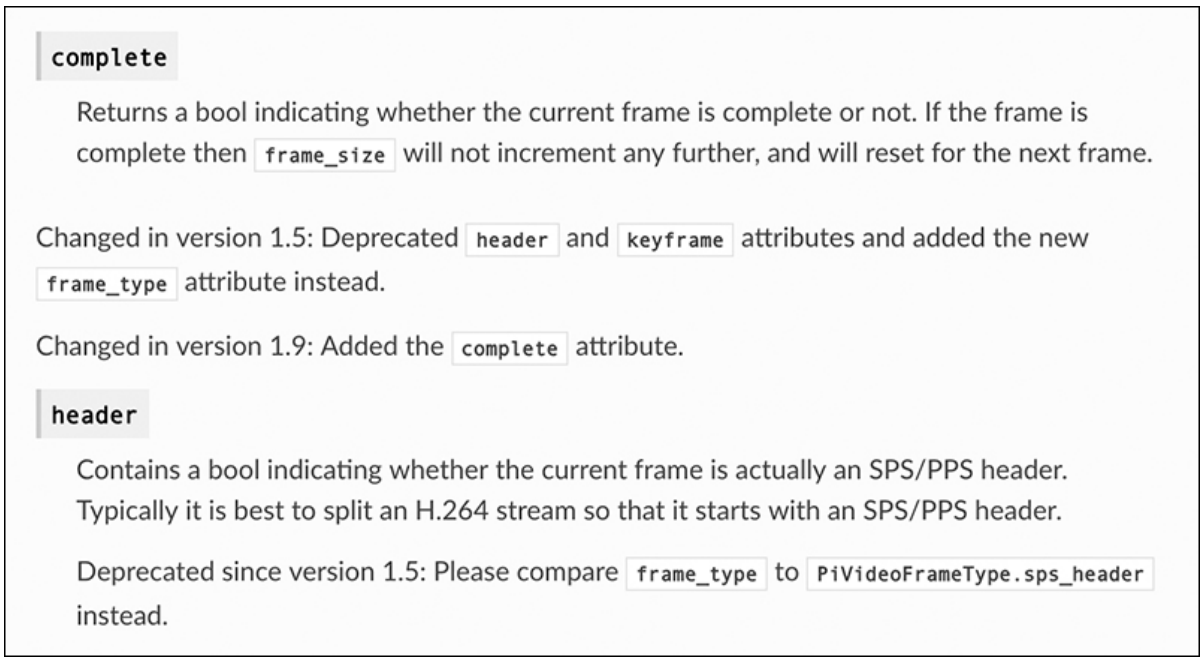


Figure 3-1: Explanation of some deprecated functions

The downside of this approach is that it relies on developers reading your changelog or documentation when they upgrade to a newer version of your Python package. However, there is a solution for that: mark your deprecated functions with the `warnings` module.

Marking Deprecated Functions with the warnings Module

Though deprecated modules should be marked well enough in documentation that users will not attempt to call them, Python also provides the `warnings` module, which allows your code to issue various kinds of warnings when a deprecated function is called. These warnings, `DeprecationWarning` and `PendingDeprecationWarning`, can be used to tell the developer that a function they're calling is deprecated or going to be deprecated, respectively.

> **NOTE**
>
> *For those who work with C, this is a handy counterpart to the `__attribute__((deprecated))` GCC extension.*

To go back to the car object example in Listing 3-4, we can use this to warn users when they are attempting to call deprecated functions, as shown in Listing 3-5.

```
import warnings

class Car(object):
    def turn_left(self):
        """Turn the car left.

     ❶ .. deprecated:: 1.1
           Use :func:`turn` instead with the direction argument set to "left".
        """
     ❷ warnings.warn("turn_left is deprecated; use turn instead",
                     DeprecationWarning)
        self.turn(direction='left')

    def turn(self, direction):
        """Turn the car in some direction.

        :param direction: The direction to turn to.
        :type direction: str
        """
        # Write actual code here instead
        pass
```

Listing 3-5: A documented change to the car object API using the `warnings` module

Here, the `turn_left` function has been deprecated ❶. By adding the `warnings.warn` line, we can write our own error message ❷. Now, if any code should call the `turn_left` function, a warning will appear that looks like this:

```
>>> Car().turn_left()
__main__:8: DeprecationWarning: turn_left is deprecated; use turn instead
```

Python 2.7 and later versions, by default, do not print any warnings emitted by the `warnings` module because the warnings are filtered. To see those warnings printed, you need to pass the `-W` option to the Python executable. The option `-W all` will print all warnings to `stderr`. See the Python man page for more information on the possible values for `-W`.

When running test suites, developers can run Python with the `-W` error option, which will raise an error every time an obsolete function is called. Developers using your library can readily find exactly where their code needs to be fixed. Listing 3-6 shows how Python transforms warnings into fatal exceptions when Python is called with the `-W` error option.

```
>>> import warnings
>>> warnings.warn("This is deprecated", DeprecationWarning)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
DeprecationWarning: This is deprecated
```

Listing 3-6: Running `Python` with the `-W` error option and getting a deprecation error

Warnings are usually missed at runtime, and running a production system with the `-W` error option is rarely a good idea. Running the test suite of a Python application with the `-W` error option, on the other hand, can be a good way to catch warnings and fix them early on.

However, manually writing all those warnings, docstring updates, and so on can become tedious, so the `debtcollector` library has been created to help automate some of that. The `debtcollector` library provides a few decorators that you can use with your functions to make sure the correct warnings are emitted and the docstring is updated correctly. Listing 3-7 shows how you can, with a simple decorator, indicate that a function has been moved to some other place.

```
from debtcollector import moves

class Car(object):
    @moves.moved_method('turn', version='1.1')
    def turn_left(self):
        """Turn the car left."""

        return self.turn(direction='left')
    def turn(self, direction):
        """Turn the car in some direction.

        :param direction: The direction to turn to.
        :type direction: str
        """

        # Write actual code here instead
        pass
```

Listing 3-7: An API change automated with `debtcollector`

Here we're using the `moves()` method from `debtcollector`, whose `moved_method` decorator makes `turn_left` emit a `DeprecationWarning` whenever it's called.

Summary

Sphinx is the de facto standard for documenting Python projects. It supports a wide variety of syntax, and it is easy to add new syntax or features if your project has particular needs. Sphinx can also automate tasks such as generating indexes or extracting documentation from your code, making it easy to maintain documentation in the long run.

Documenting changes to your API is critical, especially when you deprecate functionality, so that users are not caught unawares. Ways to document deprecations include the Sphinx deprecated keyword and the `warnings` module, and the `debtcollector` library can automate maintaining this documentation.

Christophe de Vienne on Developing APIs

Christophe is a Python developer and the author of the WSME (Web Services Made Easy) framework, which allows developers to define web

services in a Pythonic way and supports a wide variety of APIs, allowing it to be plugged into many other web frameworks.

What mistakes do developers tend to make when designing a Python API?

There are a few common mistakes I avoid when designing a Python API by following these rules:

- **Don't make it too complicated.** Keep it simple. Complicated APIs are hard to understand and hard to document. While the actual library functionality doesn't *have* to be simple as well, it's smart to make it simple so users can't easily make mistakes. For example, the library is very simple and intuitive, but it does complex things behind the scenes. The `urllib` API, by contrast, is almost as complicated as the things it does, making it hard to use.

- **Make the magic visible.** When your API does things that your documentation doesn't explain, your end users will want to crack open your code and see what's going on under the hood. It's okay if you've got some magic happening behind the scenes, but your end users should never see anything unexpected happening up front, or they could become confused or rely on a behavior that may change.

- **Don't forget use cases.** When you're so focused on writing code, it's easy to forget to think about how your library will actually be used. Thinking up good use cases makes it easier to design an API.

- **Write unit tests.** *TDD (test-driven development)* is a very efficient way to write libraries, especially in Python, because it forces the developer to assume the role of the end user from the very beginning, which leads the developer to design for usability. It's the only approach I know of that allows a programmer to completely rewrite a library, as a last resort.

What aspects of Python may affect how easy it is to design a library API?

Python has no built-in way to define which sections of the API are public and which are private, which can be both a problem and an advantage.

It's a problem because it can lead the developer to not fully consider which parts of their API are public and which parts should remain private. But with a little discipline, documentation, and (if needed) tools like `zope.interface`, it doesn't stay a problem for long.

It's an advantage when it makes it quicker and easier to refactor APIs while keeping compatibility with previous versions.

What do you consider when thinking about your API's evolution, deprecation, and removal?

There are several criteria I weigh when making any decision regarding API development:

- **How difficult will it be for users of the library to adapt their code?** Considering that there are people relying on your API, any change you make has to be worth the effort needed to adopt it. This rule is intended to prevent incompatible changes to the parts of the API that are in common use. That said, one of the advantages of Python is that it's relatively easy to refactor code to adopt an API change.

- **How easy will it be to maintain my API?** Simplifying the implementation, cleaning up the codebase, making the API easier to use, having more complete unit tests, making the API easier to understand at first glance . . . all of these things will make your life as a maintainer easier.

- **How can I keep my API consistent when applying a change?** If all the functions in your API follow a similar pattern (such as requiring the same parameter in the first position), make sure new functions follow that pattern as well. Also, doing too many things at once is a great way to end up doing none of them right: keep your API focused on what it's meant to do.

- **How will users benefit from the change?** Last but not least, always consider the users' point of view.

What advice do you have regarding API documentation in Python?

Good documentation makes it easy for newcomers to adopt your library. Neglecting it will drive away a lot of potential users—not just beginners, either. The problem is, documenting is difficult, so it gets neglected all the time!

- **Document early and include your documentation build in continuous integration.** With the Read the Docs tool for creating and hosting documentation, there's no excuse for not having documentation built and published (at least for open source software).

- **Use docstrings to document classes and functions in your API.** If you follow the PEP 257 (*https://www.python.org/dev/peps/pep-0257/*) guidelines, developers won't have to read your source to understand what your API does. Generate HTML documentation from your docstrings—and don't limit it to the API reference.

- **Give practical examples throughout.** Have at least one "startup guide" that will show newcomers how to build a working example. The first page of the documentation should give a quick overview of your API's basic and representative use case.

- **Document the evolution of your API in detail, version by version.** Version control system (VCS) logs are not enough!

- **Make your documentation accessible and, if possible, comfortable to read.** Your users need to be able to find it easily and get the information they need without feeling like they're being tortured. Publishing your documentation through PyPI is one way to achieve this; publishing on Read the Docs is also a good idea, since users will expect to find your documentation there.

- **Finally, choose a theme that is both efficient and attractive.** I chose the "Cloud" Sphinx theme for WSME, but there are plenty of other themes out there to choose from. You don't have to be a web expert to produce nice-looking documentation.

4
HANDLING TIMESTAMPS AND TIME ZONES

Time zones are complicated. Most people expect dealing with time zones to involve merely adding or subtracting a few hours from the universal time reference, UTC (Coordinated Universal Time), from –12 hours to +12 hours.

However, reality shows otherwise: time zones are not logical or predictable. There are time zones with 15-minute granularity; countries that change time zones twice a year; countries that use a custom time zone during summer, known as daylight saving time, that starts on different dates; plus tons of special and corner cases. These make the history of time zones interesting but also complicate how to handle them. All of those particularities should make you stop and think when dealing with time zones.

This chapter will outline why dealing with time zones is tricky and how to best handle them in your programs. We'll look at how to build timestamp objects, how and why to make them time zone aware, and how to deal with corner cases you might come across.

The Problem of Missing Time Zones

A timestamp without a time zone attached gives no useful information, because without the time zone, you cannot infer what point in time your application is really referring to. Without their respective time

zones, therefore, you can't compare two timestamps; that would be like comparing days of the week without accompanying dates—whether Monday is before or after Tuesday depends on what weeks they're in. Timestamps without time zones attached should be considered irrelevant.

For that reason, your application should never have to handle timestamps with no time zone. Instead, it must raise an error if no time zone is provided, or it should make clear what default time zone is assumed—for example, it's common practice to choose UTC as the default time zone.

You also must be careful of making any kind of time zone conversion before *storing* your timestamps. Imagine a user creates a recurring event every Wednesday at 10:00 AM in their local time zone, say Central European Time (CET). CET is an hour ahead of UTC, so if you convert that timestamp to UTC to store it, the event will be stored as every Wednesday at 09:00 AM. The CET time zone switches from UTC+01:00 to UTC+02:00 in the summer, so on top of that, in the summer months, your application will compute that the event starts at 11:00 AM CET every Wednesday. You can see how this program quickly becomes redundant!

Now that you understand the general problem of handling time zones, let's dig into our favorite language. Python comes with a timestamp object named `datetime.datetime` that can store date and time precise to the microsecond. The `datetime.datetime` object can be either time zone *aware*, in which case it embeds time zone information, or time zone *unaware*, in which case it does not. Unfortunately, the `datetime` API returns a time zone–unaware object by default, as you'll soon see in Listing 4-1. Let's look at how to build a default timestamp object and then how to rectify it so that it uses time zones.

Building Default datetime Objects

To build a `datetime` object with the current date and time as values, you can use the `datetime.datetime.utcnow()` function. This function retrieves the date and time for the UTC time zone right now, as shown in Listing 4-

1. To build this same object using the date and time for the time zone of the region the machine is in, you can use the `datetime.datetime.now()` method. Listing 4-1 retrieves the time and date for both UTC and my region's time zone.

```
>>> import datetime
>>> datetime.datetime.utcnow()
  ❶ datetime.datetime(2018, 6, 15, 13, 24, 48, 27631)
>>> datetime.datetime.utcnow().tzinfo is None
  ❷ True
```

Listing 4-1: Getting the time of the day with `datetime`

We import the `datetime` library and define the `datetime` object as using the UTC time zone. This returns a UTC timestamp whose values are year, month, date, hours, minutes, seconds, and microseconds ❶, respectively, in the listing. We can check whether this object has time zone information by checking the `tzinfo` object, and here we're told that it doesn't ❷.

We then create the `datetime` object using the `datetime.datetime.now()` method to retrieve the current date and time in the default time zone for the region of the machine:

```
>>> datetime.datetime.now()
  ❸ datetime.datetime(2018, 6, 15, 15, 24, 52, 276161)
```

This timestamp, too, is returned without any time zone, as we can tell from the absence of the `tzinfo` field ❸—if the time zone information had been present, it would have appeared at the end of the output as something like `tzinfo=<UTC>`.

The `datetime` API always returns unaware `datetime` objects by default, and since there is no way for you to tell what the time zone is from the output, these objects are pretty useless.

Armin Ronacher, creator of the Flask framework, suggests that an application should always assume the unaware `datetime` objects in Python are UTC. However, as we just saw, this doesn't work for objects returned by `datetime.datetime.now()`. When you are building datetime objects, I strongly recommend that you always make sure they are time

zone aware. That ensures you can always compare your objects directly and check whether they are returned correctly with the information you need. Let's see how to create time zone–aware timestamps using `tzinfo` objects.

BONUS: CONSTRUCT ING A DATETIME OB JECT FROM A DATE

You can also build your own `datetime` object with a particular date by passing the values you want for the different components of the day, as shown in Listing 4-2.

```
>>> import datetime
>>> datetime.datetime(2018, 6, 19, 19, 54, 49)
datetime.datetime(2018, 6, 19, 19, 54, 49)
```

Listing 4-2: Building your own `timestamp` object

Time Zone–Aware Timestamps with dateutil

There are already many databases of existing time zones, maintained by central authorities such as IANA (Internet Assigned Numbers Authority), which are shipped with all major operating systems. For this reason, rather than creating our own time zone classes and manually duplicating those in each Python project, Python developers rely on the `dateutil` project to obtain `tzinfo` classes. The `dateutil` project provides the Python module `tz`, which makes time zone information available directly, without much effort: the `tz` module can access the operating system's time zone information, as well as ship and embed the time zone database so it is directly accessible from Python.

You can install `dateutil` using `pip` with the command `pip install python-dateutil`. The `dateutil` API allows you to obtain a `tzinfo` object based on a time zone name, like so:

```
>>> from dateutil import tz
>>> tz.gettz("Europe/Paris")
tzfile('/usr/share/zoneinfo/Europe/Paris')
```

```
>>> tz.gettz("GMT+1")
tzstr('GMT+1')
```

The `dateutil.tz.gettz()` method returns an object implementing the `tzinfo` interface. This method accepts various string formats as argument, such as the time zone based on a location (for example, "Europe/Paris") or a time zone relative to GMT. The `dateutil` time zone objects can be used as `tzinfo` classes directly, as demonstrated in Listing 4-3.

```
>>> import datetime
>>> from dateutil import tz
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2018, 10, 16, 19, 40, 18, 279100)
>>> tz = tz.gettz("Europe/Paris")
>>> now.replace(tzinfo=tz)
datetime.datetime(2018, 10, 16, 19, 40, 18, 279100,
tzinfo=tzfile('/usr/share/zoneinfo/Europe/
Paris'))
```

Listing 4-3: Using `dateutil` objects as `tzinfo` classes

As long as you know the name of the desired time zone, you can obtain a `tzinfo` object that matches the time zone you target. The `dateutil` module can access the time zone managed by the operating system, and if that information is for some reason unavailable, will fall back on its own list of embedded time zones. If you ever need to access this embedded list, you can do so via the `datetutil.zoneinfo` module:

```
>>> from dateutil.zoneinfo import get_zonefile_instance
>>> zones = list(get_zonefile_instance().zones)
>>> sorted(zones)[:5]
['Africa/Abidjan', 'Africa/Accra', 'Africa/Addis_Ababa', 'Africa/Algiers',
'Africa/Asmara']
>>> len(zones)
592
```

In some cases, your program does not know which time zone it's running in, so you'll need to determine it yourself. The `datetutil.tz.gettz()` function will return the local time zone of your computer if you pass no argument to it, as shown in Listing 4-4.

```
>>> from dateutil import tz
>>> import datetime
```

```
>>> now = datetime.datetime.now()
>>> localzone = tz.gettz()
>>> localzone
tzfile('/etc/localtime')
>>> localzone.tzname(datetime.datetime(2018, 10, 19))
'CEST'
>>> localzone.tzname(datetime.datetime(2018, 11, 19))
'CET'
```

Listing 4-4: Obtaining your local time zone

As you can see, we pass two dates to `localzone.tzname(datetime.datetime())` separately, and `dateutil` is able to tell us that one is in Central European Summer Time (CEST) and the other is in Central European Time (no summer). If you pass in your current date, you'll get your own current time zone.

You can use objects from the `dateutil` library in `tzinfo` classes without having to bother implementing those yourself in your application. This makes it easy to convert unaware `datetime` objects to aware `datetime` objects.

IMPLEMENTING YOUR OWN TIME ZONE CLASSES

A class exists in Python that allows you to implement time zone classes yourself: the `datetime.tzinfo` class is an abstract class that provides a base for implementing classes representing time zones. If you ever want to implement a class to represent a time zone, you need to use this as the parent class and implement three different methods:

- `utcoffset(dt)`, which must return an offset from UTC in minutes east of UTC for the time zone
- `dst(dt)`, which must return the daylight saving time adjustment in minutes east of UTC for the time zone
- `tzname(dt)`, which must return the name of the time zone as a string

These three methods will embed a `tzinfo object`, allowing you to translate any time zone–aware `datetime` to another time zone.

However, as mentioned, since time zone databases exist, it's impractical to implement those time zone classes oneself.

Serializing Time Zone–Aware datetime Objects

You'll often need to transport a `datetime` object from one point to another, where those different points might not be Python native. The typical case nowadays would be with an HTTP REST API, which must return `datetime` objects serialized to a client. The native Python method named `isoformat` can be used to serialize `datetime` objects for non-Python native points, as shown in Listing 4-5.

```
   >>> import datetime
   >>> from dateutil import tz
❶ >>> def utcnow():
        return datetime.datetime.now(tz=tz.tzutc())
   >>> utcnow()
❷ datetime.datetime(2018, 6, 15, 14, 45, 19, 182703, tzinfo=tzutc())
❸ >>> utcnow().isoformat()
   '2018-06-15T14:45:21.982600+00:00'
```

Listing 4-5: Serializing a time zone–aware `datetime` object

We define a new function called `utcnow` and tell it explicitly to return an object with the UTC time zone ❶. As you can see, the object returned now contains time zone information ❷. We then format the string using the ISO format ❸, ensuring the timestamp also contains some time zone information (the `+00:00` part).

You can see I've used the method `isoformat()` to format the output. I recommend that you always format your `datetime` input and output strings using ISO 8601, with the method `datetime.datetime.isoformat()`, to return timestamps formatted in a readable way that includes the time zone information.

Your ISO 8601–formatted strings can then be converted to native `datetime.datetime` objects. The `iso8601` module offers only one function, `parse_date`, which does all the hard work of parsing the string and

determining the timestamp and time zone values. The `iso8601` module is not provided as a built-in module in Python, so you need to install it using `pip install iso8601`. Listing 4-6 shows how to parse a timestamp using ISO 8601.

```
>>> import iso8601
>>> import datetime
>>> from dateutil import tz
>>> now = datetime.datetime.utcnow()
>>> now.isoformat()
'2018-06-19T09:42:00.764337'
❶ >>> parsed = iso8601.parse_date(now.isoformat())
>>> parsed
datetime.datetime(2018, 6, 19, 9, 42, 0, 764337, tzinfo=<iso8601.Utc>)
>>> parsed == now.replace(tzinfo=tz.tzutc())
True
```

Listing 4-6: Using the `iso8601` module to parse an ISO 8601–formatted timestamp

In Listing 4-6, the `iso8601` module is used to construct a `datetime` object from a string. By calling `iso8601.parse_date` on a string containing an ISO 8601–formatted timestamp ❶, the library is able to return a `datetime` object. Since that string does not contain any time zone information, the `iso8601` module assumes that the time zone is UTC. If a time zone contains correct time zone information, the `iso8601` module returns correctly.

Using time zone–aware `datetime` objects and using ISO 8601 as the format for their string representation is a perfect solution for most problems around time zone, making sure no mistakes are made and building great interoperability between your application and the outside world.

Solving Ambiguous Times

There are certain cases where the time of the day can be ambiguous; for example during the daylight saving time transition when the same "wall clock" time occurs twice a day. The `dateutil` library provides us with the `is_ambiguous` method to distinguish such timestamps. To show this in action, we'll create an ambiguous timestamp in Listing 4-7.

```
>>> import dateutil.tz
>>> localtz = dateutil.tz.gettz("Europe/Paris")
>>> confusing = datetime.datetime(2017, 10, 29, 2, 30)
>>> localtz.is_ambiguous(confusing)
True
```

Listing 4-7: A confusing timestamp, occurring during the daylight saving time crossover

On the night of October 30, 2017, Paris switched from summer to winter time. The city switched at 3:00 AM, when the time goes back to 2:00 AM. If we try to use a timestamp at 2:30 on that date, there is no way for this object to be sure whether it is after or before the daylight saving time change.

However, it is possible to specify which side of the fold a timestamp is on by using the `fold` attribute, added to `datetime` objects from Python 3.6 by PEP 495 (Local Time Disambiguation—*https://www.python.org/dev/peps/pep-0495/*). This attribute indicates which side of the fold the datetime is on, as demonstrated in Listing 4-8.

```
>>> import dateutil.tz
>>> import datetime
>>> localtz = dateutil.tz.gettz("Europe/Paris")
>>> utc = dateutil.tz.tzutc()
>>> confusing = datetime.datetime(2017, 10, 29, 2, 30, tzinfo=localtz)
>>> confusing.replace(fold=0).astime zone(utc)
datetime.datetime(2017, 10, 29, 0, 30, tzinfo=tzutc())
>>> confusing.replace(fold=1).astime zone(utc)
datetime.datetime(2017, 10, 29, 1, 30, tzinfo=tzutc())
```

Listing 4-8: Disambiguating the ambiguous timestamp

You'll need to use this in only very rare cases, since ambiguous timestamps occur only in a small window. Sticking to UTC is a great workaround to keep life simple and avoid running into time zone issues. However, it is good to know that the `fold` attribute exists and that `dateutil` is able to help in such cases.

Summary

In this chapter, we have seen how crucial it is to carry time zone information in time stamps. The built-in `datetime` module is not complete in this regard, but the `dateutil` module is a great complement: it allows us to get `tzinfo`-compatible objects that are ready to be used. The `dateutil` module also helps us solve subtle issues such as daylight saving time ambiguity.

The ISO 8601 standard format is an excellent choice for serializing and unserializing timestamps because it is readily available in Python and compatible with any other programming language.

5
DISTRIBUTING YOUR SOFTWARE

It's safe to say that at some point, you will want to distribute your software. As tempted as you might be to just zip your code and upload it to the internet, Python provides tools to make it easier for your end users to get your software to work. You should already be familiar with using *setup.py* to install Python applications and libraries, but you have probably never delved into how it works behind the scenes or how to make a *setup.py* of your own.

In this chapter, you'll learn the history of *setup.py*, how the file works, and how to create your own custom *setup.py*. We'll also take a look at some of the less well-known capabilities of the package installation tool `pip` and how to make your software downloadable via `pip`. Finally, we'll see how to use Python's entry points to make functions easy to find between programs. With these skills, you can make your published software accessible for end users.

A Bit of setup.py History

The `distutils` library, originally created by software developer Greg Ward, has been part of the standard Python library since 1998. Ward sought to create an easy way for developers to automate the installation process for their end users. Packages provide the *setup.py* file as the

standard Python script for their installation, and they can use `distutils` to install themselves, as shown in Listing 5-1.

```
#!/usr/bin/python
from distutils.core import setup

setup(name="rebuildd",
      description="Debian packages rebuild tool",
      author="Julien Danjou",
      author_email="acid@debian.org",
      url="http://julien.danjou.info/software/rebuildd.html",
      packages=['rebuildd'])
```

Listing 5-1: Building a setup.py *using* `distutils`

With the *setup.py* file as the root of a project, all users have to do to build or install your software is run that file with the appropriate command as its argument. Even if your distribution includes C modules in addition to native Python ones, `distutils` can handle them automatically.

Development of `distutils` was abandoned in 2000; since then, other developers have picked up where it left off. One of the notable successors is the packaging library known as `setuptools`, which offers more frequent updates and advanced features, such as automatic dependency handling, the `Egg` distribution format, and the `easy_install` command. Since `distutils` was still the accepted means of packaging software included with the Python Standard Library at the time of development, `setuptools` provided a degree of backward compatibility with it. Listing 5-2 shows how you'd use `setuptools` to build the same installation package as in Listing 5-1.

```
#!/usr/bin/env python
import setuptools

setuptools.setup(
    name="rebuildd",
    version="0.2",
    author="Julien Danjou",
    author_email="acid@debian.org",
    description="Debian packages rebuild tool",
    license="GPL",
    url="http://julien.danjou.info/software/rebuildd/",
    packages=['rebuildd'],
    classifiers=[
        "Development Status :: 2 - Pre-Alpha",
```

```
        "Intended Audience :: Developers",
        "Intended Audience :: Information Technology",
        "License :: OSI Approved :: GNU General Public License (GPL)",
        "Operating System :: OS Independent",
        "Programming Language :: Python"
    ],
)
```

Listing 5-2: Building a setup.py *using* `setuptools`

Eventually, development on `setuptools` slowed down too, but it wasn't long before another group of developers forked it to create a new library called `distribute`, which offered several advantages over `setuptools`, including fewer bugs and Python 3 support.

All the best stories have a twist ending, though: in March 2013, the teams behind `setuptools` and `distribute` decided to merge their codebases under the aegis of the original `setuptools` project. So `distribute` is now deprecated, and `setuptools` is once more the canonical way to handle advanced Python installations.

While all this was happening, another project, known as `distutils2`, was developed with the intention of completely replacing `distutils` in the Python Standard Library. Unlike both `distutils` and `setuptools`, it stored package metadata in a plaintext file, *setup.cfg*, which was easier both for developers to write and for external tools to read. However, `distutils2` retained some of the failings of `distutils`, such as its obtuse command-based design, and lacked support for entry points and native script execution on Windows—both features provided by `setuptools`. For these and other reasons, plans to include `distutils2`, renamed as `packaging`, in the Python 3.3 Standard Library fell through, and the project was abandoned in 2012.

There is still a chance for `packaging` to rise from the ashes through `distlib`, an up-and-coming effort to replace `distutils`. Before release, it was rumored that the `distlib` package would become part of the Standard Library in Python 3.4, but that never came to be. Including the best features from `packaging`, `distlib` implements the basic groundwork described in the packaging-related PEPs.

So, to recap:

- **`distutils`** is part of the Python Standard Library and can handle simple package installations.
- **`setuptools`**, the standard for advanced package installations, was at first deprecated but is now back in active development and the de facto standard.
- **`distribute`** has been merged back into `setuptools` as of version 0.7; **`distutils2`** (aka `packaging`) has been abandoned.
- **`distlib`** *might* replace `distutils` in the future.

There are other packaging libraries out there, but these are the five you'll encounter the most. Be careful when researching these libraries on the internet: plenty of documentation is outdated due to the complicated history outlined above. The official documentation is up-to-date, however.

In short, `setuptools` is the distribution library to use for the time being, but keep an eye out for `distlib` in the future.

Packaging with setup.cfg

You've probably already tried to write a *setup.py* for a package at some point, either by copying one from another project or by skimming through the documentation and building it yourself. Building a *setup.py* is not an intuitive task. Choosing the right tool to use is just the first challenge. In this section, I want to introduce you to one of the recent improvements to `setuptools`: the *setup.cfg* file support.

This is what a *setup.py* using a *setup.cfg* file looks like:

```
import setuptools

setuptools.setup()
```

Two lines of code—it is that simple. The actual metadata the setup requires is stored in *setup.cfg*, as in Listing 5-3.

```
[metadata]
name = foobar
author = Dave Null
```

```
author-email = foobar@example.org
license = MIT
long_description = file: README.rst
url = http://pypi.python.org/pypi/foobar
requires-python = >=2.6
classifiers =
    Development Status :: 4 - Beta
    Environment :: Console
    Intended Audience :: Developers
    Intended Audience :: Information Technology
    License :: OSI Approved :: Apache Software License
    Operating System :: OS Independent
    Programming Language :: Python
```

Listing 5-3: The setup.cfg *metadata*

As you can see, *setup.cfg* uses a format that's easy to write and read, directly inspired by `distutils2`. Many other tools, such as `Sphinx` or `Wheel`, also read configuration from this *setup.cfg* file—that alone is a good argument to start using it.

In Listing 5-3, the description of the project is read from the *README.rst* file. It's good practice to always have a README file—preferably in the RST format—so users can quickly understand what the project is about. With just these basic *setup.py* and *setup.cfg* files, your package is ready to be published and used by other developers and applications. The `setuptools` documentation provides more details if needed, for example, if you have some extra steps in your installation process or want to include extra files.

Another useful packaging tool is `pbr`, short for *Python Build Reasonableness*. The project was started in OpenStack as an extension of `setuptools` to facilitate installation and deployment of packages. The `pbr` packaging tool, used alongside `setuptools`, implements features absent from `setuptools`, including these:

- Automatic generation of Sphinx documentation
- Automatic generation of *AUTHORS* and *ChangeLog* files based on `git` history
- Automatic creation of file lists for `git`
- Version management based on `git` tags using semantic versioning

And all this with little to no effort on your part. To use `pbr`, you just need to enable it, as shown in Listing 5-4.

```
import setuptools

setuptools.setup(setup_requires=['pbr'], pbr=True)
```

Listing 5-4: setup.py *using pbr*

The `setup_requires` parameter indicates to `setuptools` that `pbr` must be installed prior to using `setuptools`. The `pbr=True` argument makes sure that the `pbr` extension for `setuptools` is loaded and called.

Once enabled, the `python setup.py` command is enhanced with the `pbr` features. Calling `python setup.py -version` will, for example, return the version number of the project based on existing `git` tags. Running `python setup.py sdist` would create a source tarball with automatically generated *ChangeLog* and *AUTHORS* files.

The Wheel Format Distribution Standard

For most of Python's existence, there's been no official standard distribution format. While different distribution tools generally use some common archive format—even the `Egg` format introduced by `setuptools` is just a zip file with a different extension—their metadata and package structures are incompatible with each other. This problem was compounded when an official installation standard was finally defined in PEP 376 that was also incompatible with existing formats.

To solve these problems, PEP 427 was written to define a new standard for Python distribution packages called `Wheel`. The reference implementation of this format is available as a tool, also called `Wheel`.

`Wheel` is supported by `pip` starting with version 1.4. If you're using `setuptools` and have the `Wheel` package installed, it automatically integrates itself as a `setuptools` command named `bdist_wheel`. If you don't have `Wheel` installed, you can install it using the command `pip install wheel`. Listing 5-5 shows some of the output when calling `bdist_wheel`, abridged for print.

```
$ python setup.py bdist_wheel
running bdist_wheel
running build
running build_py
creating build/lib
creating build/lib/daiquiri
creating build/lib/daiquiri/tests
copying daiquiri/tests/__init__.py -> build/lib/daiquiri/tests
--snip--
running egg_info
writing requirements to daiquiri.egg-info/requires.txt
writing daiquiri.egg-info/PKG-INFO
writing top-level names to daiquiri.egg-info/top_level.txt
writing dependency_links to daiquiri.egg-info/dependency_links.txt
writing pbr to daiquiri.egg-info/pbr.json
writing manifest file 'daiquiri.egg-info/SOURCES.txt'
installing to build/bdist.macosx-10.12-x86_64/wheel
running install
running install_lib
--snip--

running install_scripts
creating build/bdist.macosx-10.12-x86_64/wheel/daiquiri-1.3.0.dist-info/WHEEL
❶ creating '/Users/jd/Source/daiquiri/dist/daiquiri-1.3.0-py2.py3-none-any.whl'
and adding '.' to it
adding 'daiquiri/__init__.py'
adding 'daiquiri/formatter.py'
adding 'daiquiri/handlers.py'

--snip--
```

Listing 5-5: Calling `setup.py bdist_wheel`

The `bdist_wheel` command creates a *.whl* file in the *dist* directory ❶. As with the `Egg` format, a `Wheel` archive is just a zip file with a different extension. However, `Wheel` archives do not require installation—you can load and run your code just by adding a slash followed by the name of your module:

```
$ python wheel-0.21.0-py2.py3-none-any.whl/wheel -h
usage: wheel [-h]

             {keygen,sign,unsign,verify,unpack,install,install-
scripts,convert,help}
             --snip--

positional arguments:
--snip--
```

You might be surprised to learn this is not a feature introduced by the `Wheel` format itself. Python can also run regular zip files, just like with Java's *.jar* files:

```
python foobar.zip
```

This is equivalent to:

```
PYTHONPATH=foobar.zip python -m __main__
```

In other words, the `__main__` module for your program will be automatically imported from `__main__.py`. You can also import `__main__` from a module you specify by appending a slash followed by the module name, just as with `Wheel`:

```
python foobar.zip/mymod
```

This is equivalent to:

```
PYTHONPATH=foobar.zip python -m mymod.__main__
```

One of the advantages of `Wheel` is that its naming conventions allow you to specify whether your distribution is intended for a specific architecture and/or Python implementation (CPython, PyPy, Jython, and so on). This is particularly useful if you need to distribute modules written in C.

By default, `Wheel` packages are tied to the major version of Python that you used to build them. When called with `python2 setup.py bdist_wheel`, the pattern of a `Wheel` filename will be something like *library-version-py2-none-any.whl*.

If your code is compatible with all major Python versions (that is, Python 2 and Python 3), you can build a universal `Wheel`:

```
python setup.py bdist_wheel --universal
```

The resulting filename will be different and contains both Python major versions—something like *library-version-py2.py3-none-any.whl*.

Building a universal `Wheel` avoids ending up with two different `Wheels` when only one would cover both Python major versions.

If you don't want to pass the `--universal` flag each time you are building a `Wheel`, you can just add this to your *setup.cfg* file:

```
[wheel]
universal=1
```

If the `Wheel` you build contains binary programs or libraries (like a Python extension written in C), the binary `Wheel` might not be as portable as you imagine. It will work by default on some platforms, such as Darwin (macOS) or Microsoft Windows, but it might not work on all Linux distributions. The PEP 513 (*https://www.python.org/dev/peps/pep-0513*) targets this Linux problem by defining a new platform tag named `manylinux1` and a minimal set of libraries that are guaranteed to be available on that platform.

`Wheel` is a great format for distributing ready-to-install libraries and applications, so you are encouraged to build and upload them to PyPI as well.

Sharing Your Work with the World

Once you have a proper *setup.py* file, it is easy to build a source tarball that can be distributed. The `sdist setuptools` command does just that, as demonstrated in Listing 5-6.

```
$ python setup.py sdist
running sdist

[pbr] Generating AUTHORS
running egg_info
writing requirements to ceilometer.egg-info/requires.txt
writing ceilometer.egg-info/PKG-INFO
writing top-level names to ceilometer.egg-info/top_level.txt
writing dependency_links to ceilometer.egg-info/dependency_links.txt
writing entry points to ceilometer.egg-info/entry_points.txt
[pbr] Processing SOURCES.txt
[pbr] In git context, generating filelist from git
warning: no previously-included files matching '*.pyc' found anywhere in
distribution
writing manifest file 'ceilometer.egg-info/SOURCES.txt'
running check
```

```
copying setup.cfg -> ceilometer-2014.1.a6-g772e1a7
Writing ceilometer-2014.1.a6-g772e1a7/setup.cfg

--snip--

Creating tar archive
removing 'ceilometer-2014.1.a6.g772e1a7' (and everything under it)
```

Listing 5-6: Using `setup.py sdist` to build a source tarball

The `sdist` command creates a tarball under the *dist* directory of the source tree. The tarball contains all the Python modules that are part of the source tree. As seen in the previous section, you can also build `Wheel` archives using the `bdist_wheel` command. `Wheel` archives are a bit faster to install as they're already in the correct format for installation.

The final step to make that code accessible is to export your package somewhere users can install it via `pip`. That means publishing your project to PyPI.

If it's your first time exporting to PyPI, it pays to test out the publishing process in a safe sandbox rather than on the production server. You can use the PyPI staging server for this purpose; it replicates all the functionality of the main index but is solely for testing purposes.

The first step is to register your project on the test server. Start by opening your *~/.pypirc* file and adding these lines:

```
[distutils]
index-servers =
    testpypi
[testpypi]
username = <your username>
password = <your password>
repository = https://testpypi.python.org/pypi
```

Save the file, and now you can register your project in the index:

```
$ python setup.py register -r testpypi
running register
running egg_info
writing requirements to ceilometer.egg-info/requires.txt
writing ceilometer.egg-info/PKG-INFO
writing top-level names to ceilometer.egg-info/top_level.txt
writing dependency_links to ceilometer.egg-info/dependency_links.txt
writing entry points to ceilometer.egg-info/entry_points.txt
[pbr] Reusing existing SOURCES.txt
running check
```

```
Registering ceilometer to https://testpypi.python.org/pypi
Server response (200): OK
```

This connects to the test PyPI server instance and creates a new entry. Don't forget to use the `-r` option; otherwise, the real production PyPI instance would be used!

Obviously, if a project with the same name is already registered there, the process will fail. Retry with a new name, and once you get your program registered and receive the `OK` response, you can upload a source distribution tarball, as shown in Listing 5-7.

```
$ python setup.py sdist upload -r testpypi
running sdist
[pbr] Writing ChangeLog
[pbr] Generating AUTHORS
running egg_info
writing requirements to ceilometer.egg-info/requires.txt
writing ceilometer.egg-info/PKG-INFO
writing top-level names to ceilometer.egg-info/top_level.txt
writing dependency_links to ceilometer.egg-info/dependency_links.txt
writing entry points to ceilometer.egg-info/entry_points.txt
[pbr] Processing SOURCES.txt
[pbr] In git context, generating filelist from git
warning: no previously-included files matching '*.pyc' found anywhere in
distribution
writing manifest file 'ceilometer.egg-info/SOURCES.txt'
running check
creating ceilometer-2014.1.a6.g772e1a7

--snip--

copying setup.cfg -> ceilometer-2014.1.a6.g772e1a7
Writing ceilometer-2014.1.a6.g772e1a7/setup.cfg
Creating tar archive
removing 'ceilometer-2014.1.a6.g772e1a7' (and everything under it)
running upload
Submitting dist/ceilometer-2014.1.a6.g772e1a7.tar.gz to https://testpypi
.python.org/pypi
Server response (200): OK
```

Listing 5-7: Uploading your tarball to PyPI

Alternatively, you could upload a `Wheel` archive, as in Listing 5-8.

```
$ python setup.py bdist_wheel upload -r testpypi
running bdist_wheel
running build
running build_py
running egg_info
writing requirements to ceilometer.egg-info/requires.txt
```

```
writing ceilometer.egg-info/PKG-INFO
writing top-level names to ceilometer.egg-info/top_level.txt
writing dependency_links to ceilometer.egg-info/dependency_links.txt
writing entry points to ceilometer.egg-info/entry_points.txt
[pbr] Reusing existing SOURCES.txt
installing to build/bdist.linux-x86_64/wheel
running install
running install_lib
creating build/bdist.linux-x86_64/wheel

--snip--

creating build/bdist.linux-x86_64/wheel/ceilometer-2014.1.a6.g772e1a7
.dist-info/WHEEL
running upload
Submitting /home/jd/Source/ceilometer/dist/ceilometer-2014.1.a6
.g772e1a7-py27-none-any.whl to https://testpypi.python.org/pypi
Server response (200): OK
```

Listing 5-8: Uploading a `Wheel` archive to PyPI

Once those operations are finished, you and other users can search for the uploaded packages on the PyPI staging server, and even install those packages using `pip`, by specifying the test server using the `-i` option:

```
$ pip install -i https://testpypi.python.org/pypi ceilometer
```

If everything checks out, you can upload your project to the main PyPI server. Just make sure to add your credentials and the details for the server to your `~/.pypirc` file first, like so:

```
[distutils]
index-servers =
    pypi
    testpypi

[pypi]
username = <your username>
password = <your password>
[testpypi]
repository = https://testpypi.python.org/pypi
username = <your username>
password = <your password>
```

Now if you run `register` and `upload` with the `-r pypi` switch, your package should be uploaded to PyPI.

NOTE

PyPI can keep several versions of your software in its index, allowing you to install specific and older versions, if you ever need to. Just pass the version number to the `pip install` *command; for example,* `pip install foobar==1.0.2`.

This process is straightforward to use and allows for any number of uploads. You can release your software as often as you want, and your users can install and update as often as they need.

Entry Points

You may have already used `setuptools` entry points without knowing anything about them. Software distributed using `setuptools` includes important metadata describing features such as its required dependencies and—more relevantly to this topic—a list of *entry points*. Entry points are methods by which other Python programs can discover the dynamic features a package provides.

The following example shows how to provide an entry point named `rebuildd` in the `console_scripts` entry point group:

```
#!/usr/bin/python
from distutils.core import setup

setup(name="rebuildd",
    description="Debian packages rebuild tool",
    author="Julien Danjou",
    author_email="acid@debian.org",
    url="http://julien.danjou.info/software/rebuildd.html",
    entry_points={
        'console_scripts': [
            'rebuildd = rebuildd:main',
        ],
    },
    packages=['rebuildd'])
```

Any Python package can register entry points. Entry points are organized in groups: each group is made of a list of key and value pairs.

Those pairs use the format `path.to.module:variable_name`. In the previous example, the key is `rebuildd`, and the value is `rebuildd:main`.

The list of entry points can be manipulated using various tools, from `setuptools` to `epi`, as I'll show here. In the following sections, we discuss how we can use entry points to add extensibility to our software.

Visualizing Entry Points

The easiest way to visualize the entry points available in a package is to use a package called `entry point inspector`. You can install it by running `pip install entry-point-inspector`. When installed, it provides the command `epi` that you can run from your terminal to interactively discover the entry points provided by installed packages. Listing 5-9 shows an example of running `epi group list` on my system.

```
$ epi group list
--------------------------
| Name                   |
--------------------------
| console_scripts |
| distutils.commands |
| distutils.setup_keywords |
| egg_info.writers |
| epi.commands |
| flake8.extension |
| setuptools.file_finders |
| setuptools.installation |
--------------------------
```

Listing 5-9: Getting a list of entry point groups

The output from `epi group list` in Listing 5-9 shows the different packages on a system that provide entry points. Each item in this table is the name of an entry point group. Note that this list includes `console_scripts`, which we'll discuss shortly. We can use the `epi` command with the `show` command to show details of a particular entry point group, as in Listing 5-10.

```
$ epi group show console_scripts
-----------------------------------------------
| Name     | Module   | Member | Distribution | Error |
-----------------------------------------------
| coverage | coverage | main   | coverage 3.4 |       |
```

Listing 5-10: Showing details of an entry point group

We can see that in the group `console_scripts`, an entry point named `coverage` refers to the member `main` of the module `coverage`. This entry point in particular, provided by the package `coverage 3.4`, indicates which Python function to call when the command line script `coverage` is executed. Here, the function `coverage.main` is to be called.

The `epi` tool is just a thin layer on top of the complete Python library `pkg_resources`. This module allows us to discover entry points for any Python library or program. Entry points are valuable for various things, including console scripts and dynamic code discovery, as you'll see in the next few sections.

Using Console Scripts

When writing a Python application, you almost always have to provide a launchable program—a Python script that the end user can run—that needs to be installed inside a directory somewhere in the system path.

Most projects have a launchable program similar to this:

```
#!/usr/bin/python
import sys
import mysoftware

mysoftware.SomeClass(sys.argv).run()
```

This kind of script is a best-case scenario: many projects have a much longer script installed in the system path. However, such scripts pose some major issues:

- There's no way the user can know where the Python interpreter is or which version it uses.
- This script leaks binary code that can't be imported by software or unit tests.
- There's no easy way to define where to install this script.
- It's not obvious how to install this in a portable way (for example, on both Unix and Windows).

Helping us circumvent these problems, `setuptools` offers the `console_scripts` feature. This entry point can be used to make `setuptools` install a tiny program in the system path that calls a specific function in one of your modules. With `setuptools`, you can specify a function call to start your program by setting up a key/value pair in the `console_scripts` entry point group: the key is the script name that will be installed, and the value is the Python path to your function (something like `my_module.main`).

Let's imagine a `foobar` program that consists of a client and a server. Each part is written in its module—`foobar.client` and `foobar.server`, respectively, in *foobar/client.py*:

```
def main():
    print("Client started")
```

And in *foobar/server.py*:

```
def main():
    print("Server started")
```

Of course, this program doesn't do much of anything—our client and server don't even talk to each other. For our example, though, they just need to print a message letting us know they have started successfully.

We can now write the following *setup.py* file in the root directory with entry points defined in *setup.py*.

```
from setuptools import setup

setup(
    name="foobar",
    version="1",
    description="Foo!",
    author="Julien Danjou",
    author_email="julien@danjou.info",
    packages=["foobar"],
    entry_points={
        "console_scripts": [
         ❶ "foobard = foobar.server:main",
           "foobar = foobar.client:main",
        ],
     },
)
```

We define entry points using the format `module.submodule:function`. You can see here that we've defined an entry point each for both `client` and `server` ❶.

When `python setup.py install` is run, `setuptools` will create a script that will look like the one in Listing 5-11.

```
#!/usr/bin/python
# EASY-INSTALL-ENTRY-SCRIPT: 'foobar==1','console_scripts','foobar'
__requires__ = 'foobar==1'
import sys
from pkg_resources import load_entry_point

if __name__ == '__main__':
    sys.exit(
        load_entry_point('foobar==1', 'console_scripts', 'foobar')()
    )
```

Listing 5-11: A console script generated by `setuptools`

This code scans the entry points of the `foobar` package and retrieves the `foobar` key from the `console_scripts` group, which is used to locate and run the corresponding function. The return value of the `load_entry_point` will then be a reference to the function `foobar.client.main`, which will be called without any arguments and whose return value will be used as an exit code.

Notice that this code uses `pkg_resources` to discover and load entry point files from within your Python programs.

NOTE

If you're using `pbr` on top of `setuptools`, the generated script is simpler (and therefore faster) than the default one built by `setuptools`, as it will call the function you wrote in the entry point without having to search the entry point list dynamically at runtime.

Using console scripts is a technique that removes the burden of writing portable scripts, while ensuring that your code stays in your Python package and can be imported (and tested) by other programs.

Using Plugins and Drivers

Entry points make it easy to discover and dynamically load code deployed by other packages, but this is not their only use. Any application can propose and register entry points and groups and then use them as it wishes.

In this section, we're going to create a `cron`-style daemon `pycrond` that will allow any Python program to register a command to be run once every few seconds by registering an entry point in the group `pytimed`. The attribute indicated by this entry point should be an object that returns `number_of_seconds, callable`.

Here's our implementation of `pycrond` using `pkg_resources` to discover entry points, in a program I've named *pytimed.py*:

```
import pkg_resources
import time

def main():
    seconds_passed = 0
    while True:
        for entry_point in pkg_resources.iter_entry_points('pytimed'):
            try:
                seconds, callable = entry_point.load()()
            except:
                # Ignore failure
                pass
            else:
                if seconds_passed % seconds == 0:
                    callable()
        time.sleep(1)
        seconds_passed += 1
```

This program consists of an infinite loop that iterates over each entry point of the `pytimed` group. Each entry point is loaded using the `load()` method. The program then calls the returned method, which needs to return the number of seconds to wait before calling the callable as well as the aforementioned callable.

The program in *pytimed.py* is a very simplistic and naive implementation, but it is sufficient for our example. Now we can write another Python program, named *hello.py*, that needs one of its functions called on a periodic basis:

```
def print_hello():
    print("Hello, world!")

def say_hello():
    return 2, print_hello
```

Once we have that function defined, we register it using the appropriate entry points in *setup.py*.

```
from setuptools import setup

setup(
    name="hello",
    version="1",
    packages=["hello"],
    entry_points={
        "pytimed": [
            "hello = hello:say_hello",
        ],
     },)
```

The *setup.py* script registers an entry point in the group `pytimed` with the key `hello` and the value pointing to the function `hello.say_hello`. Once that package is installed using that *setup.py*—for example, using `pip install`—the `pytimed` script can detect the newly added entry point.

At startup, `pytimed` will scan the group `pytimed` and find the key `hello`. It will then call the `hello.say_hello` function, getting two values: the number of seconds to wait between each call and the function to call, 2 seconds and `print_hello` in this case. By running the program, as we do in Listing 5-12, you can see "Hello, world!" printed on the screen every 2 seconds.

```
>>> import pytimed
>>> pytimed.main()
Hello, world!
Hello, world!
Hello, world!
```

Listing 5-12: Running `pytimed`

The possibilities this mechanism offers are immense: you can build driver systems, hook systems, and extensions easily and generically. Implementing this mechanism by hand in every program you make would be tedious, but fortunately, there's a Python library that can take care of the boring parts for us.

The `stevedore` library provides support for dynamic plugins based on the same mechanism demonstrated in our previous examples. The use case in this example is already simplistic, but we can still simplify it further in this script, *pytimed_stevedore.py*:

```
from stevedore.extension import ExtensionManager
import time

def main():
    seconds_passed = 0
    extensions = ExtensionManager('pytimed', invoke_on_load=True)
    while True:
        for extension in extensions:
            try:
                seconds, callable = extension.obj
            except:
                # Ignore failure
                pass
            else:
                if seconds_passed % seconds == 0:
                    callable()
        time.sleep(1)
        seconds_passed += 1
```

The `ExtensionManager` class of `stevedore` provides a simple way to load all extensions of an entry point group. The name is passed as a first argument. The argument `invoke_on_load=True` makes sure that each function of the group is called once discovered. This makes the results accessible directly from the `obj` attribute of the extension.

If you look through the `stevedore` documentation, you will see that `ExtensionManager` has a variety of subclasses that can handle different situations, such as loading specific extensions based on their names or the result of a function. All of those are commonly used models you can apply to your program in order to implement those patterns directly.

For example, we might want to load and run only one extension from our entry point group. Leveraging the `stevedore.driver.DriverManager` class allows us to do that, as Listing 5-13 shows.

```
from stevedore.driver import DriverManager
import time

def main(name):
    seconds_passed = 0
    seconds, callable = DriverManager('pytimed', name, invoke_on_load=True).
driver
```

```
    while True:
        if seconds_passed % seconds == 0:
            callable()
        time.sleep(1)
        seconds_passed += 1

main("hello")
```

Listing 5-13: Using `stevedore` to run a single extension from an entry point

In this case, only one extension is loaded and selected by name. This allows us to quickly build a *driver system* in which only one extension is loaded and used by a program.

Summary

The packaging ecosystem in Python has a bumpy history; however, the situation is now settling. The `setuptools` library provides a complete solution to packaging, not only to transport your code in different formats and upload it to PyPI, but also to handle connection with other software and libraries via entry points.

Nick Coghlan on Packaging

Nick is a Python core developer working at Red Hat. He has written several PEP proposals, including PEP 426 (Metadata for Python Software Packages 2.0), and he is acting as delegate for our Benevolent Dictator for Life, Guido van Rossum, author of Python.

The number of packaging solutions (`distutils`, `setuptools`, `distutils2`, `distlib`, `bento`, `pbr`, and so on) for Python is quite extensive. In your opinion, what are the reasons for such fragmentation and divergence?

The short answer is that software publication, distribution, and integration is a complex problem with plenty of room for multiple solutions tailored for different use cases. In my recent talks on this, I have noted that the problem is mainly one of age, with the different packaging tools being born into different eras of software distribution.

PEP 426, which defines a new metadata format for Python packages, is still fairly recent and not yet approved. How do you think it will tackle current packaging problems?

PEP 426 originally started as part of the `Wheel` format definition, but Daniel Holth realized that `Wheel` could work with the existing metadata format defined by `setuptools`. PEP 426 is thus a consolidation of the existing `setuptools` metadata with some of the ideas from `distutils2` and other packaging systems (such as `RPM` and `npm`). It addresses some of the frustrations encountered with existing tools (for example, with cleanly separating different kinds of dependencies).

The main gains will be a REST API on PyPI offering full metadata access, as well as (hopefully) the ability to automatically generate distribution policy–compliant packages from upstream metadata.

The `Wheel` format is somewhat recent and not widely used yet, but it seems promising. Why is it not part of the Standard Library?

It turns out the Standard Library is not really a suitable place for packaging standards: it evolves too slowly, and an addition to a later version of the Standard Library cannot be used with earlier versions of Python. So, at the Python language summit earlier this year, we tweaked the PEP process to allow `distutils-sig` to manage the full approval cycle for packaging-related PEPs, and `python-dev` will only be involved for proposals that involve changing CPython directly (such as `pip` bootstrapping).

What is the future for Wheel packages?

We still have some tweaks to make before `Wheel` is suitable for use on Linux. However, `pip` is adopting `Wheel` as an alternative to the `Egg` format, allowing local caching of builds for fast virtual environment creation, and PyPI allows uploads of `Wheel` archives for Windows and macOS.

6
UNIT TESTING

Many find unit testing to be arduous and time-consuming, and some people and projects have no testing policy. This chapter assumes that you see the wisdom of unit testing! Writing code that is not tested is fundamentally useless, as there's no way to conclusively prove that it works. If you need convincing, I suggest you start by reading about the benefits of test-driven development.

In this chapter you'll learn about the Python tools you can use to construct a comprehensive suite of tests that will make testing simpler and more automated. We'll talk about how you can use tools to make your software rock solid and regression-free. We'll cover creating reusable test objects, running tests in parallel, revealing untested code, and using virtual environments to make sure your tests are clean, as well as some other good-practice methods and ideas.

The Basics of Testing

Writing and running unit tests is uncomplicated in Python. The process is not intrusive or disruptive, and unit testing will greatly help you and other developers in maintaining your software. Here I'll discuss some of the absolute basics of testing that will make things easier for you.

Some Simple Tests

First, you should store tests inside a `tests` submodule of the application or library they apply to. Doing so will allow you to ship the tests as part of your module so that they can be run or reused by anyone—even after your software is installed—without necessarily using the source package. Making the tests a submodule of your main module also prevents them from being installed by mistake in a top-level `tests` module.

Using a hierarchy in your test tree that mimics the hierarchy of your module tree will make the tests more manageable. This means that the tests covering the code of *mylib/foobar.py* should be stored inside *mylib/tests/test_foobar.py*. Consistent nomenclature makes things simpler when you're looking for the tests related to a particular file. Listing 6-1 shows the simplest unit test you can write.

```
def test_true():
    assert True
```

Listing 6-1: A really simple test in test_true.py

This will simply assert that the behavior of the program is what you expect. To run this test, you need to load the *test_true.py* file and run the `test_true()` function defined within.

However, writing and running an individual test for each of your test files and functions would be a pain. For small projects with simple usage, the `pytest` package comes to the rescue—once installed via `pip`, pytest provides the `pytest` command, which loads every file whose name starts with *test_* and then executes all functions within that start with `test_`.

With just the *test_true.py* file in our source tree, running `pytest` gives us the following output:

```
 $ pytest -v test_true.py
========================== test session starts ===========================
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0 --
/usr/local/opt/python/bin/python3.6
cachedir: .cache
rootdir: examples, inifile:
collected 1 item
```

```
test_true.py::test_true PASSED                                       [100%]

======================== 1 passed in 0.01 seconds ========================
```

The `-v` option tells `pytest` to be verbose and print the name of each test run on a separate line. If a test fails, the output changes to indicate the failure, accompanied by the whole traceback.

Let's add a failing test this time, as shown in Listing 6-2.

```
def test_false():
    assert False
```

Listing 6-2: A failing test in test_true.py

If we run the test file again, here's what happens:

```
 $ pytest -v test_true.py
========================== test session starts ===========================
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0 -- /usr/
local/opt/python/bin/python3.6
cachedir: .cache
rootdir: examples, inifile:
collected 2 items

test_true.py::test_true PASSED                                       [ 50%]
test_true.py::test_false FAILED                                      [100%]

================================ FAILURES ================================
_________________________________ test_false _________________________________

    def test_false():
>       assert False
E       assert False

test_true.py:5: AssertionError
=================== 1 failed, 1 passed in 0.07 seconds ===================
```

A test fails as soon as an `AssertionError` exception is raised; our `assert` test will raise an `AssertionError` when its argument is evaluated to something false (`False`, `None`, 0, etc.). If any other exception is raised, the test also errors out.

Simple, isn't it? While simplistic, a lot of small projects use this approach and it works very well. Those projects require no tools or libraries other than pytest and thus can rely on simple `assert` tests.

As you start to write more sophisticated tests, pytest will help you understand what's wrong in your failing tests. Imagine the following

test:

```
def test_key():
    a = ['a', 'b']
    b = ['b']
    assert a == b
```

When `pytest` is run, it gives the following output:

```
 $ pytest test_true.py
========================== test session starts ==========================
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: /Users/jd/Source/python-book/examples, inifile:
plugins: celery-4.1.0
collected 1 item

test_true.py F                                                      [100%]

================================ FAILURES ================================
_________________________________ test_key _________________________________

    def test_key():
        a = ['a', 'b']
        b = ['b']
>       assert a == b
E       AssertionError: assert ['a', 'b'] == ['b']
E         At index 0 diff: 'a' != 'b'
E         Left contains more items, first extra item: 'b'
E         Use -v to get the full diff

test_true.py:10: AssertionError
======================== 1 failed in 0.07 seconds ========================
```

This tells us that `a` and `b` are different and that this test does not pass. It also tells us exactly how they are different, making it easy to fix the test or code.

Skipping Tests

If a test cannot be run, you will probably want to skip that test—for example, you may wish to run a test conditionally based on the presence or absence of a particular library. To that end, you can use the `pytest.skip()` function, which will mark the test as skipped and move on to the next one. The `pytest.mark.skip` decorator skips the decorated test function unconditionally, so you'll use it when a test always needs to be skipped. Listing 6-3 shows how to skip a test using these methods.

```
import pytest

try:
    import mylib
except ImportError:
    mylib = None

@pytest.mark.skip("Do not run this")
def test_fail():
    assert False

@pytest.mark.skipif(mylib is None, reason="mylib is not available")
def test_mylib():
    assert mylib.foobar() == 42
def test_skip_at_runtime():
    if True:
        pytest.skip("Finally I don't want to run it")
```

Listing 6-3: Skipping tests

When executed, this test file will output the following:

```
 $ pytest -v examples/test_skip.py
========================== test session starts ===========================
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0 -- /usr/
local/opt/python/bin/python3.6
cachedir: .cache
rootdir: examples, inifile:
collected 3 items

examples/test_skip.py::test_fail SKIPPED
[ 33%]
examples/test_skip.py::test_mylib SKIPPED
[ 66%]
examples/test_skip.py::test_skip_at_runtime SKIPPED
[100%]

================= 3 skipped in 0.01 seconds =================
```

The output of the test run in Listing 6-3 indicates that, in this case, all the tests have been skipped. This information allows you to ensure you didn't accidentally skip a test you expected to run.

Running Particular Tests

When using `pytest`, you often want to run only a particular subset of your tests. You can select which tests you want to run by passing their directory or files as an argument to the `pytest` command line. For example, calling `pytest test_one.py` will only run the *test_one.py* test. Pytest

also accepts a directory as argument, and in that case, it will recursively scan the directory and run any file that matches the *test_*.py* pattern.

You can also add a filter with the `-k` argument on the command line in order to execute only the test matching a name, as shown in Listing 6-4.

```
$ pytest -v examples/test_skip.py -k test_fail
========================== test session starts ==========================
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0 -- /usr/
local/opt/python/bin/python3.6
cachedir: .cache
rootdir: examples, inifile:
collected 3 items

examples/test_skip.py::test_fail SKIPPED
[100%]
=== 2 tests deselected ===
=== 1 skipped, 2 deselected in 0.04 seconds ===
```

Listing 6-4: Filtering tests run by name

Names are not always the best way to filter which tests will run. Commonly, a developer would group tests by functionalities or types instead. Pytest provides a dynamic marking system that allows you to mark tests with a keyword that can be used as a filter. To mark tests in this way, use the `-m` option. If we set up a couple of tests like this:

```
import pytest

@pytest.mark.dicttest
def test_something():
    a = ['a', 'b']
    assert a == a

def test_something_else():
    assert False
```

we can use the `-m` argument with `pytest` to run only one of those tests:

```
$ pytest -v test_mark.py -m dicttest
=== test session starts ===
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0 -- /usr/
local/opt/python/bin/python3.6
cachedir: .cache
rootdir: examples, inifile:
collected 2 items

test_mark.py::test_something PASSED
```

```
[100%]

=== 1 tests deselected ===
=== 1 passed, 1 deselected in 0.01 seconds ===
```

The `-m` marker accepts more complex queries, so we can also run all tests that are *not* marked:

```
$ pytest test_mark.py -m 'not dicttest'
=== test session starts ===
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: examples, inifile:
collected 2 items

test_mark.py F
[100%]

=== FAILURES ===
test_something_else
    def test_something_else():
>       assert False
E       assert False

test_mark.py:10: AssertionError
=== 1 tests deselected ===
=== 1 failed, 1 deselected in 0.07 seconds ===
```

Here pytest executed every test that was not marked as `dicttest`—in this case, the `test_something_else` test, which failed. The remaining marked test, `test_something`, was not executed and so is listed as `deselected`.

Pytest accepts complex expressions composed of the `or`, `and`, and `not` keywords, allowing you to do more advanced filtering.

Running Tests in Parallel

Test suites can take a long time to run. It's not uncommon for a full suite of unit tests to take tens of minutes to run in large software projects. By default, pytest runs all tests serially, in an undefined order. Since most computers have several CPUs, you can usually speed things up if you split the list of tests and run them on multiple CPUs.

To handle this approach, pytest provides the plugin `pytest-xdist`, which you can install with `pip`. This plugin extends the pytest command line with the `--numprocesses` argument (shortened as `-n`), which accepts as its argument the number of CPUs to use. Running `pytest -n 4` would run

your test suite using four parallel processes, balancing the load across the available CPUs.

Because the number of CPUs can change from one computer to another, the plugin also accepts the `auto` keyword as a value. In this case, it will probe the machine to retrieve the number of CPUs available and start this number of processes.

Creating Objects Used in Tests with Fixtures

In unit testing, you'll often need to execute a set of common instructions before and after running a test, and those instructions will use certain components. For example, you might need an object that represents the configuration state of your application, and you'll likely want that object to be initialized before each test, then reset to its default values when the test is achieved. Similarly, if your test relies on the temporary creation of a file, the file must be created before the test starts and deleted once the test is done. These components, known as *fixtures*, are set up before a test and cleaned up after the test has finished.

With pytest, fixtures are defined as simple functions. The fixture function should return the desired object(s) so that a test using that fixture can use that object.

Here's a simple fixture:

```
import pytest

@pytest.fixture
def database():
    return <some database connection>

def test_insert(database):
    database.insert(123)
```

The database fixture is automatically used by any test that has `database` in its argument list. The `test_insert()` function will receive the result of the `database()` function as its first argument and use that result as it wants. When we use a fixture this way, we don't need to repeat the database initialization code several times.

Another common feature of code testing is tearing down after a test has used a fixture. For example, you may need to close a database connection. Implementing the fixture as a generator allows us to add teardown functionality, as shown in Listing 6-5.

```
import pytest

@pytest.fixture
def database():
    db = <some database connection>
    yield db
    db.close()

def test_insert(database):
    database.insert(123)
```

Listing 6-5: Teardown functionality

Because we used the `yield` keyword and made `database` a generator, the code after the `yield` statement runs when the test is done. That code will close the database connection at the end of the test.

However, closing a database connection for each test might impose an unnecessary runtime cost, as tests may be able to reuse that same connection. In that case, you can pass the `scope` argument to the fixture decorator, specifying the scope of the fixture:

```
import pytest

@pytest.fixture(scope="module")
def database():
    db = <some database connection>
    yield db
    db.close()

def test_insert(database):
    database.insert(123)
```

By specifying the `scope="module"` parameter, you initialize the fixture once for the whole module, and the same database connection will be passed to all test functions requesting a database connection.

Finally, you can run some common code before and after your tests by marking fixtures as *automatically used* with the `autouse` keyword, rather than specifying them as an argument for each of the test functions. Specifying the `autouse=True` keyword argument to the `pytest.fixture()`

function will make sure the fixture is called before running any test in the module or class it is defined in, as in this example:

```
import os

import pytest

@pytest.fixture(autouse=True)
def change_user_env():
    curuser = os.environ.get("USER")
    os.environ["USER"] = "foobar"
    yield
    os.environ["USER"] = curuser

def test_user():
    assert os.getenv("USER") == "foobar"
```

Such automatically enabled features are handy, but make sure not to abuse fixtures: they are run before each and every test covered by their scope, so they can slow down a test run significantly.

Running Test Scenarios

When unit testing, you may want to run the same error-handling test with several different objects that trigger that error, or you may want to run an entire test suite against different drivers.

We relied heavily on this latter approach when developing *Gnocchi*, a time series database. Gnocchi provides an abstract class that we call the *storage API*. Any Python class can implement this abstract base and register itself to become a driver. The software loads the configured storage driver when required and uses the implemented storage API to store or retrieve data. In this case, we need a class of unit tests that runs against each driver—thus running against each implementation of this storage API—to be sure all drivers conform to what the callers expect.

An easy way to achieve this is by using *parameterized fixtures*, which will run all the tests that use them several times, once for each of the defined parameters. Listing 6-6 shows an example of using parameterized fixtures to run a single test twice with different parameters: once for `mysql` and once for `postgresql`.

```
import pytest
import myapp
@pytest.fixture(params=["mysql", "postgresql"])
def database(request):
    d = myapp.driver(request.param)
    d.start()
    yield d
    d.stop()

def test_insert(database):
    database.insert("somedata")
```

Listing 6-6: Running a test using parameterized fixtures

In Listing 6-6, the `driver` fixture is parameterized with two different values, each the name of a database driver that is supported by the application. When `test_insert` is run, it is actually run twice: once with a MySQL database connection and once with a PostgreSQL database connection. This allows us to easily reuse the same test with different scenarios, without adding many lines of code.

Controlled Tests Using Mocking

Mock objects are simulated objects that mimic the behavior of real application objects, but in particular and controlled ways. These are especially useful in creating environments that describe precisely the conditions for which you would like to test code. You can replace all objects but one with mock objects to isolate the behavior of your focus object and create an enviroment for testing your code.

One use case is in writing an HTTP client, since it is likely impossible (or at least extremely complicated) to spawn the HTTP server and test it through all scenarios to return every possible value. HTTP clients are especially difficult to test for all failure scenarios.

The standard library for creating mock objects in Python is `mock`. Starting with Python 3.3, `mock` has been merged into the Python Standard Library as `unittest.mock`. You can, therefore, use a snippet like the following to maintain backward compatibility between Python 3.3 and earlier versions:

```
try:
    from unittest import mock
```

```
except ImportError:
    import mock
```

The `mock` library is pretty simple to use. Any attribute accessed on a `mock.Mock` object is dynamically created at runtime. Any value can be set to such an attribute. Listing 6-7 shows `mock` being used to create a fake object with a fake attribute.

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.some_attribute = "hello world"
>>> m.some_attribute
"hello world"
```

Listing 6-7: Accessing the `mock.Mock` attribute

You can also dynamically create a method on a malleable object, as in Listing 6-8 where we create a fake method that always returns 42 and accepts anything as an argument.

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.some_method.return_value = 42
>>> m.some_method()
42
>>> m.some_method("with", "arguments")
42
```

Listing 6-8: Creating methods on a `mock.Mock` object

In just a few lines, your `mock.Mock` object now has a `some_method()` method that returns 42. It accepts any kind of argument, and there is no check on what the values are—yet.

Dynamically created methods can also have (intentional) side effects. Rather than being boilerplate methods that just return a value, they can be defined to execute useful code.

Listing 6-9 creates a fake method that has the side effect of printing the `"hello world"` string.

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> def print_hello():
...     print("hello world!")
...     return 43
...
```

```
❶ >>> m.some_method.side_effect = print_hello
   >>> m.some_method()
   hello world!
   43
❷ >>> m.some_method.call_count
   1
```

Listing 6-9: Creating methods on a `mock.Mock` object with side effects

We assign an entire function to the `some_method` attribute ❶. This technique allows us to implement more complex scenarios in a test because we can plug any code needed for testing into a mock object. We then just need to pass this mock object to whichever function expects it.

The `call_count` attribute ❷ is a simple way of checking the number of times a method has been called.

The `mock` library uses the action/assertion pattern: this means that once your test has run, it's up to you to check that the actions you are mocking were correctly executed. Listing 6-10 applies the `assert()` method to our mock objects to perform these checks.

```
   >>> from unittest import mock
   >>> m = mock.Mock()
❶ >>> m.some_method('foo', 'bar')
   <Mock name='mock.some_method()' id='26144272'>
❷ >>> m.some_method.assert_called_once_with('foo', 'bar')
   >>> m.some_method.assert_called_once_with('foo', ❸mock.ANY)
   >>> m.some_method.assert_called_once_with('foo', 'baz')
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
     File "/usr/lib/python2.7/dist-packages/mock.py", line 846, in assert_called_
   once_with
       return self.assert_called_with(*args, **kwargs)
     File "/usr/lib/python2.7/dist-packages/mock.py", line 835, in assert_called_
   with
       raise AssertionError(msg)
   AssertionError: Expected call: some_method('foo', 'baz')
   Actual call: some_method('foo', 'bar')
```

Listing 6-10: Checking method calls

We create a method with the arguments `foo` and `bar` to stand in as our tests by calling the method ❶. The usual way to check calls to a mock object is to use the `assert_called()` methods, such as `assert_called_once_with()` ❷. To these methods, you need to pass the values

that you expect callers to use when calling your mock method. If the values passed are not the ones being used, then `mock` raises an `AssertionError`. If you don't know what arguments may be passed, you can use `mock.ANY` as a value ❸; that will match any argument passed to your mock method.

Th `mock` library can also be used to patch some function, method, or object from an external module. In Listing 6-11, we replace the `os.unlink()` function with a fake function we provide.

```
>>> from unittest import mock
>>> import os
>>> def fake_os_unlink(path):
...     raise IOError("Testing!")
...
>>> with mock.patch('os.unlink', fake_os_unlink):
...     os.unlink('foobar')
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in fake_os_unlink
IOError: Testing!
```

Listing 6-11: Using `mock.patch`

When used as a context manager, `mock.patch()` replaces the target function with the function we provide so the code executed inside the context uses that patched method. With the `mock.patch()` method, it's possible to change any part of an external piece of code, making it behave in a way that lets you test all conditions in your application, as shown in Listing 6-12.

```
   from unittest import mock

   import pytest
   import requests

   class WhereIsPythonError(Exception):
       pass

❶ def is_python_still_a_programming_language():
       try:
           r = requests.get("http://python.org")
       except IOError:
           pass
       else:
           if r.status_code == 200:
```

```
            return 'Python is a programming language' in r.content
        raise WhereIsPythonError("Something bad happened")

    def get_fake_get(status_code, content):
        m = mock.Mock()
        m.status_code = status_code
        m.content = content

        def fake_get(url):
            return m

        return fake_get

    def raise_get(url):
        raise IOError("Unable to fetch url %s" % url)

❷ @mock.patch('requests.get', get_fake_get(
        200, 'Python is a programming language for sure'))
    def test_python_is():
        assert is_python_still_a_programming_language() is True

    @mock.patch('requests.get', get_fake_get(
        200, 'Python is no more a programming language'))
    def test_python_is_not():
        assert is_python_still_a_programming_language() is False

    @mock.patch('requests.get', get_fake_get(404, 'Whatever'))
    def test_bad_status_code():
        with pytest.raises(WhereIsPythonError):
            is_python_still_a_programming_language()

    @mock.patch('requests.get', raise_get)
    def test_ioerror():
        with pytest.raises(WhereIsPythonError):
            is_python_still_a_programming_language()
```

Listing 6-12: Using `mock.patch()` to test a set of behaviors

Listing 6-12 implements a test suite that searches for all instances of the string "Python is a programming language" on the *http://python.org/* web page ❶. There is no way to test negative scenarios (where this sentence is not on the web page) without modifying the page itself—something we're not able to do, obviously. In this case, we're using `mock` to cheat and change the behavior of the request so it returns a mocked reply with a fake page that doesn't contain that string. This allows us to test the negative scenario in which *http://python.org/* does not contain this sentence, making sure the program handles that case correctly.

This example uses the decorator version of `mock.patch()` ❷. Using the decorator does not change the mocking behavior, but it is simpler when

you need to use mocking within the context of an entire test function.

Using mocking, we can simulate any problem, such as a web server returning a 404 error, an I/O error, or a network latency issue. We can make sure code returns the correct values or raises the correct exception in every case, ensuring our code always behaves as expected.

Revealing Untested Code with coverage

A great complement to unit testing, the `coverage` tool identifies whether any of your code has been missed during testing. It uses code analysis tools and tracing hooks to determine which lines of your code have been executed; when used during a unit test run, it can show you which parts of your codebase have been crossed over and which parts have not. Writing tests is useful, but having a way to know what part of your code you may have missed during the testing process is the cherry on the cake.

Install the `coverage` Python module on your system via `pip` to have access to the `coverage` program command from your shell.

> **NOTE**
>
> *The command may also be named `python-coverage`, if you install `coverage` through your operating system installation software. This is the case on Debian, for example.*

Using `coverage` in stand-alone mode is straightforward. It can show you parts of your programs that are never run and which code might be "dead code," that is, code that could be removed without modifying the normal workflow of the program. All the test tools we've talked about so far in this chapter are integrated with `coverage`.

When using `pytest`, just install the `pytest-cov` plugin via `pip install pytest-pycov` and add a few option switches to generate a detailed code coverage output, as shown in Listing 6-13.

```
$ pytest --cov=gnocchiclient gnocchiclient/tests/unit
---------- coverage: platform darwin, python 3.6.4-final-0 -----------
```

```
Name                                      Stmts   Miss Branch BrPart  Cover
---------------------------
gnocchiclient/__init__.py                     0      0      0      0   100%
gnocchiclient/auth.py                        51     23      6      0    49%
gnocchiclient/benchmark.py                  175    175     36      0     0%
--snip--
---------------------------
TOTAL                                      2040   1868    424      6     8%

=== passed in 5.00 seconds ===
```

Listing 6-13: Using coverage with `pytest`

The `--cov` option enables the coverage report at the end of the test run. You need to pass the package name as an argument for the plugin to filter the coverage report properly. The output includes the lines of code that were not run and therefore have no tests. All you need to do now is spawn your favorite text editor and start writing tests for that code.

However, `coverage` goes one better, allowing you to generate clear HTML reports. Simply add the `--cov-report=html` flag, and the *htmlcov* directory from which you ran the command will be populated with HTML pages. Each page will show you which parts of your source code were or were not run.

If you want to be *that* person, you can use the option `--cover-fail-under=COVER_MIN_PERCENTAGE`, which will make the test suite fail if a minimum percentage of the code is not executed when the test suite is run. While having a good coverage percentage is a decent goal, and while the tool is useful to gain insight into the state of your test coverage, defining an arbitrary percentage value does not provide much insight. Figure 6-1 shows an example of a coverage report with the percentage at the top.

For example, a code coverage score of 100 percent is a respectable goal, but it does not necessarily mean the code is entirely tested and you can rest. It only proves that your whole code path has been run; there is no indication that every possible condition has been tested.

You should use coverage information to consolidate your test suite and add tests for any code that is currently not being run. This facilitates later project maintenance and increases your code's overall quality.

Coverage for **ceilometer.publisher** : 75%

12 statements | 9 run | 3 missing | 0 excluded

```
1  # -*- encoding: utf-8 -*-
2  #
3  # Copyright © 2013 Intel Corp.
4  # Copyright © 2013 eNovance
5  #
6  # Author: Yunhong Jiang <yunhong.jiang@intel.com>
7  #         Julien Danjou <julien@danjou.info>
8  #
9  # Licensed under the Apache License, Version 2.0 (the "License"); you may
10 # not use this file except in compliance with the License. You may obtain
11 # a copy of the License at
12 #
13 #      http://www.apache.org/licenses/LICENSE-2.0
14 #
15 # Unless required by applicable law or agreed to in writing, software
16 # distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
17 # WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
18 # License for the specific language governing permissions and limitations
19 # under the License.
20
21 import abc
22 from stevedore import driver
23 from ceilometer.openstack.common import network_utils
24
25
26 def get_publisher(url, namespace='ceilometer.publisher'):
27     """Get publisher driver and load it.
28
29     :param URL: URL for the publisher
30     :param namespace: Namespace to use to look for drivers.
31     """
32     parse_result = network_utils.urlsplit(url)
33     loaded_driver = driver.DriverManager(namespace, parse_result.scheme)
34     return loaded_driver.driver(parse_result)
35
36
37 class PublisherBase(object):
38     """Base class for plugins that publish the sampler."""
39
40     __metaclass__ = abc.ABCMeta
41
42     def __init__(self, parsed_url):
43         pass
44
45     @abc.abstractmethod
46     def publish_samples(self, context, samples):
47         "Publish samples into final conduit."
48
```

Figure 6-1: Coverage of `ceilometer.publisher`

Virtual Environments

Earlier we mentioned the danger that your tests may not capture the absence of dependencies. Any application of significant size inevitably depends on external libraries to provide features the application needs, but there are many ways external libraries might cause issues on your operating system. Here are a few:

- Your system does not have the library you need packaged.
- Your system does not have the right *version* of the library you need packaged.
- You need two different versions of the same library for two different applications.

These problems can happen when you first deploy your application or later on, while it's running. Upgrading a Python library installed via your system manager might break your application in a snap without warning, for reasons as simple as an API change in the library being used by the application.

The solution is for each application to use a library directory that contains all the application's dependencies. This directory is then used to load the needed Python modules rather than the system-installed ones.

Such a directory is known as a *virtual environment*.

Setting Up a Virtual Environment

The tool `virtualenv` handles virtual environments automatically for you. Until Python 3.2, you'll find it in the `virtualenv` package that you can install using `pip install virtualenv`. If you use Python 3.3 or later, it's available directly via Python under the `venv` name.

To use the module, load it as the main program with a destination directory as its argument, like so:

```
$ python3 -m venv myvenv
$ ls foobar
bin       include    lib       pyvenv.cfg
```

Once run, `venv` creates a *lib/pythonX.Y* directory and uses it to install `pip` into the virtual environment, which will be useful to install further Python packages.

You can then activate the virtual environment by "sourcing" the `activate` command. Use the following on Posix systems:

```
$ source myvenv/bin/activate
```

On Windows systems, use this code:

```
> \myvenv\Scripts\activate
```

Once you do that, your shell prompt should appear prefixed by the name of your virtual environment. Executing `python` will call the version of Python that has been copied into the virtual environment. You can check that it's working by reading the `sys.path` variable and checking that it has your virtual environment directory as its first component.

You can stop and leave the virtual environment at any time by calling the `deactivate` command:

```
$ deactivate
```

That's it. Also note that you are not forced to run `activate` if you want to use the Python installed in your virtual environment just once. Calling the `python` binary will also work:

```
$ myvenv/bin/python
```

Now, while we're in our activated virtual environment, we do not have access to any of the modules installed and available on the main system. That is the point of using a virtual environment, but it does mean we probably need to install the packages we need. To do that, use the standard `pip` command to install each package, and the packages will install in the right place, without changing anything about your system:

```
$ source myvenv/bin/activate
(myvenv) $ pip install six
Downloading/unpacking six
  Downloading six-1.4.1.tar.gz
  Running setup.py egg_info for package six
```

```
Installing collected packages: six
  Running setup.py install for six

Successfully installed six
Cleaning up...
```

Voilà! We can install all the libraries we need and then run our application from this virtual environment, without breaking our system. It's easy to see how we can script this to automate the installation of a virtual environment based on a list of dependencies, as in Listing 6-14.

```
virtualenv myappvenv
source myappvenv/bin/activate
pip install -r requirements.txt
deactivate
```

Listing 6-14: Automatic virtual environment creation

It can still be useful to have access to your system-installed packages, so `virtualenv` allows you to enable them when creating your virtual environment by passing the `--system-site-packages` flag to the `virtualenv` command.

Inside `myvenv`, you will find a *pyvenv.cfg*, the configuration file for this environment. It doesn't have a lot of configuration options by default. You should recognize `include-system-site-package`, whose purpose is the same as the `--system-site-packages` of `virtualenv` that we described earlier.

As you might guess, virtual environments are incredibly useful for automated runs of unit test suites. Their use is so widespread that a particular tool has been built to address it.

Using virtualenv with tox

One of the central uses of virtual environments is to provide a clean environment for running unit tests. It would be detrimental if you were under the impression that your tests were working, when they were not, for example, respecting the dependency list.

One way to ensure you're accounting for all the dependencies would be to write a script to deploy a virtual environment, install `setuptools`, and then install all of the dependencies required for both your

application/library runtime and unit tests. Luckily, this is such a popular use case that an application dedicated to this task has already been built: `tox`.

The `tox` management tool aims to automate and standardize how tests are run in Python. To that end, it provides everything needed to run an entire test suite in a clean virtual environment, while also installing your application to check that the installation works.

Before using `tox`, you need to provide a configuration file named *tox.ini* that should be placed in the root directory of your project, beside your *setup.py* file:

```
$ touch tox.ini
```

You can then run `tox` successfully:

```
% tox
GLOB sdist-make: /home/jd/project/setup.py
python create: /home/jd/project/.tox/python
python inst: /home/jd/project/.tox/dist/project-1.zip
____________________ summary ____________________
  python: commands succeeded
  congratulations :)
```

In this instance, `tox` creates a virtual environment in *.tox/python* using the default Python version. It uses *setup.py* to create a distribution of your package, which it then installs inside this virtual environment. No commands are run, because we did not specify any in the configuration file. This alone is not particularly useful.

We can change this default behavior by adding a command to run inside our test environment. Edit *tox.ini* to include the following:

```
[testenv]
commands=pytest
```

Now `tox` runs the command `pytest`. However, since we do not have `pytest` installed in the virtual environment, this command will likely fail. We need to list `pytest` as a dependency to be installed:

```
[testenv]
deps=pytest
commands=pytest
```

When run now, `tox` re-creates the environment, installs the new dependency, and runs the command `pytest`, which executes all of the unit tests. To add more dependencies, you can either list them in the `deps` configuration option, as is done here, or use the `-rfile` syntax to read from a file.

Re-creating an Environment

Sometimes you'll need to re-create an environment to, for example, ensure things work as expected when a new developer clones the source code repository and runs `tox` for the first time. For this, `tox` accepts a `--recreate` option that will rebuild the virtual environment from scratch based on parameters you lay out.

You define the parameters for all virtual environments managed by `tox` in the `[testenv]` section of *tox.ini*. And, as mentioned, `tox` can manage multiple Python virtual environments—indeed, it is possible to run our tests under a Python version other than the default one by passing the `-e` flag to `tox`, like so:

```
 % tox -e py26
 GLOB sdist-make: /home/jd/project/setup.py
 py26 create: /home/jd/project/.tox/py26
 py26 installdeps: nose
 py26 inst: /home/jd/project/.tox/dist/rebuildd-1.zip
 py26 runtests: commands[0] | pytests
 --snip--
== test session starts ==
=== 5 passed in 4.87 seconds ====
```

By default, `tox` simulates any environment that matches an existing Python version: `py24`, `py25`, `py26`, `py27`, `py30`, `py31`, `py32`, `py33`, `py34`, `py35`, `py36`, `py37`, `jython`, and `pypy`! Furthermore, you can define your own environments. You just need to add another section named `[testenv:_envname_]`. If you want to run a particular command for just one of the environments, you can do so easily by listing the following in the *tox.ini* file:

```
[testenv]
deps=pytest
commands=pytest
```

```
[testenv:py36-coverage]
deps={[testenv]deps}
      pytest-cov
commands=pytest --cov=myproject
```

By using `pytest --cov=myproject` under the `py36-coverage` section as shown here, you override the commands for the `py36-coverage` environment, meaning when you run `tox -e py36-coverage`, `pytest` is installed as part of the dependencies, but the command `pytest` is actually run instead with the coverage option. For that to work, the `pytest-cov` extension must be installed: to this end, we replace the `deps` value with the `deps` from `testenv` and add the `pytest-cov` dependency. Variable interpolation is also supported by `tox`, so you can refer to any other field from the *tox.ini* file and use it as a variable, the syntax being `{[env_name]variable_name}`. This allows us to avoid repeating the same things over and over again.

Using Different Python Versions

We can also create a new environment with an unsupported version of Python right away with the following in *tox.ini*:

```
[testenv]
deps=pytest
commands=pytest

[testenv:py21]
basepython=python2.1
```

When we run this, it will now (attempt to) use Python 2.1 to run the test suite—although since it is very unlikely you have this ancient Python version installed on your system, I doubt this would work for you!

It's likely that you'll want to support multiple Python versions, in which case it would be useful to have `tox` run all the tests for all the Python versions you want to support by default. You can do this by specifying the environment list you want to use when `tox` is run without arguments:

```
[tox]
envlist=py35,py36,pypy
```

```
[testenv]
deps=pytest
commands=pytest
```

When `tox` is launched without any further arguments, all four environments listed are created, populated with the dependencies and the application, and then run with the command `pytest`.

Integrating Other Tests

We can also use `tox` to integrate tests like `flake8`, as discussed in Chapter 1. The following *tox.ini* file provides a PEP 8 environment that will install `flake8` and run it:

```
[tox]
envlist=py35,py36,pypy,pep8

[testenv]
deps=pytest
commands=pytest

[testenv:pep8]
deps=flake8
commands=flake8
```

In this case, the `pep8` environment is run using the default version of Python, which is probably fine, though you can still specify the `basepython` option if you want to change that.

When running `tox`, you'll notice that all the environments are built and run sequentially. This can make the process very long, but since virtual environments are isolated, nothing prevents you from running `tox` commands in parallel. This is exactly what the `detox` package does, by providing a `detox` command that runs all of the default environments from *envlist* in parallel. You should `pip install` it!

Testing Policy

Embedding testing code in your project is an excellent idea, but how that code is run is also extremely important. Too many projects have test code lying around that fails to run for some reason or other. This

topic is not strictly limited to Python, but I consider it important enough to emphasize here: you should have a zero-tolerance policy regarding untested code. No code should be merged without a proper set of unit tests to cover it.

The minimum you should aim for is that each of the commits you push passes all the tests. Automating this process is even better. For example, OpenStack relies on a specific workflow based on *Gerrit* (a web-based code review service) and *Zuul* (a continuous integration and delivery service). Each commit pushed goes through the code review system provided by Gerrit, and Zuul is in charge of running a set of testing jobs. Zuul runs the unit tests and various higher-level functional tests for each project. This code review, which is executed by a couple of developers, makes sure all code committed has associated unit tests.

If you're using the popular GitHub hosting service, *Travis CI* is a tool that allows you to run tests after each push or merge or against pull requests that are submitted. While it is unfortunate that this testing is done post-push, it's still a fantastic way to track regressions. Travis supports all significant Python versions out of the box, and it can be customized significantly. Once you've activated Travis on your project via the web interface at *https://www.travis-ci.org/*, just add a *.travis.yml* file that will determine how the tests are run. Listing 6-15 shows an example of a *.travis.yml* file.

```
language: python
python:
  - "2.7"
  - "3.6"
# command to install dependencies
install: "pip install -r requirements.txt --use-mirrors"
# command to run tests
script: pytest
```

Listing 6-15: A .travis.yml *example file*

With this file in place in your code repository and Travis enabled, the latter will spawn a set of jobs to test your code with the associated unit tests. It's easy to see how you can customize this by simply adding dependencies and tests. Travis is a paid service, but the good news is that for open source projects, it's entirely free!

The `tox-travis` package (*https://pypi.python.org/pypi/tox-travis/*) is also worth looking into, as it will polish the integration between `tox` and Travis by running the correct `tox` target depending on the Travis environment being used. Listing 6-16 shows an example of a *.travis.yml* file that will install `tox-travis` before running `tox`.

```
sudo: false
language: python
python:
  - "2.7"
  - "3.4"
install: pip install tox-travis
script: tox
```

Listing 6-16: A .travis.yml *example file with* `tox-travis`

Using `tox-travis`, you can simply call `tox` as the script on Travis, and it will call `tox` with the environment you specify here in the *.travis.yml* file, building the necessary virtual environment, installing the dependency, and running the commands you specified in *tox.ini*. This makes it easy to use the same workflow both on your local development machine and on the Travis continuous integration platform.

These days, wherever your code is hosted, it is always possible to apply some automatic testing of your software and to make sure your project is moving forward, not being held back by the addition of bugs.

Robert Collins on Testing

Robert Collins is, among other things, the original author of the *Bazaar* distributed version control system. Today, he is a Distinguished Technologist at HP Cloud Services, where he works on OpenStack. Robert is also the author of many of the Python tools described in this book, such as fixtures, `testscenarios`, `testrepository`, and even `python-subunit`—you may have used one of his programs without knowing it!

What kind of testing policy would you advise using? Is it ever acceptable not to test code?

I think testing is an engineering trade-off: you must consider the likelihood of a failure slipping through to production undetected, the

cost and size of an undetected failure, and cohesion of the team doing the work. Take OpenStack, which has 1,600 contributors: it's difficult to work with a nuanced policy with so many people with their own opinions. Generally speaking, a project needs some automated testing to check that the code will do what it is intended to do, and that what it is intended to do is what is needed. Often that requires functional tests that might be in different codebases. Unit tests are excellent for speed and pinning down corner cases. I think it is okay to vary the balance between styles of testing, as long as there is testing.

Where the cost of testing is very high and the returns are very low, I think it's fine to make an informed decision not to test, but that situation is relatively rare: most things can be tested reasonably cheaply, and the benefit of catching errors early is usually quite high.

What are the best strategies when writing Python code to make testing manageable and improve the quality of the code?

Separate out concerns and don't do multiple things in one place; this makes reuse natural, and that makes it easier to put test doubles in place. Take a purely functional approach when possible; for example, in a single method either calculate something or change some state, but avoid doing both. That way you can test all of the calculating behaviors without dealing with state changes, such as writing to a database or talking to an HTTP server. The benefit works the other way around too—you can replace the calculation logic for tests to provoke corner case behavior and use mocks and test doubles to check that the expected state propagation happens as desired. The most heinous things to test are deeply layered stacks with complex cross-layer behavioral dependencies. There you want to evolve the code so that the contract between layers is simple, predictable, and—most usefully for testing—replaceable.

What's the best way to organize unit tests in source code?

Have a clear hierarchy, like *$ROOT/$PACKAGE/tests*. I tend to do just one hierarchy for a whole source tree, for example *$ROOT/$PACKAGE/$SUBPACKAGE/tests*.

Within tests, I often mirror the structure of the rest of the source tree: *$ROOT/$PACKAGE/foo.py* would be tested in *$ROOT/$PACKAGE/tests/test_foo.py*.

The rest of the tree should not import from the tests tree, except perhaps in the case of a `test_suite/load_tests` function in the top level `__init__`. This permits you to easily detach the tests for small-footprint installations.

What do you see as the future of unit-testing libraries and frameworks in Python?

The significant challenges I see are these:

- The continued expansion of parallel capabilities in new machines, like phones with four CPUs. Existing unit test internal APIs are not optimized for parallel workloads. My work on the StreamResult Java class is aimed directly at resolving this.
- More complex scheduling support—a less ugly solution for the problems that class and module-scoped setup aim at.
- Finding some way to consolidate the vast variety of frameworks we have today: for integration testing, it would be great to be able to get a consolidated view across multiple projects that have different test runners in use.

7
METHODS AND DECORATORS

Python's decorators are a handy way to modify functions. Decorators were first introduced in Python 2.2, with the `classmethod()` and `staticmethod()` decorators, but were overhauled to become more flexible and readable. Along with these two original decorators, Python now provides a few right out of the box and supports the simple creation of custom decorators. But it seems as though most developers do not understand how they work behind the scenes.

This chapter aims to change that—we'll cover what a decorator is and how to use it, as well as how to create your own decorators. Then we'll look at using decorators to create static, class, and abstract methods and take a close look at the `super()` function, which allows you to place implementable code inside an abstract method.

Decorators and When to Use Them

A *decorator* is a function that takes another function as an argument and replaces it with a new, modified function. The primary use case for decorators is in factoring common code that needs to be called before, after, or around multiple functions. If you've ever written Emacs Lisp code, you may have used the `defadvice` decorator, which allows you to define code called around a function. If you've used method combinations in the Common Lisp Object System (CLOS), Python

decorators follow the same concepts. We'll look at some simple decorator definitions, and then we'll examine some common situations in which you'd use decorators.

Creating Decorators

The odds are good that you've already used decorators to make your own wrapper functions. The dullest possible decorator, and the simplest example, is the `identity()` function, which does nothing except return the original function. Here is its definition:

```
def identity(f):
    return f
```

You would then use your decorator like this:

```
@identity
def foo():
    return 'bar'
```

You enter the name of the decorator preceded by an `@` symbol and then enter the function you want to use it on. This is the same as writing the following:

```
def foo():
    return 'bar'
foo = identity(foo)
```

This decorator is useless, but it works. Let's look at another, more useful example in Listing 7-1.

```
_functions = {}
def register(f):
    global _functions
    _functions[f.__name__] = f
    return f
@register
def foo():
    return 'bar'
```

Listing 7-1: A decorator to organize functions in a dictionary

In Listing 7-1, the `register` decorator stores the decorated function name into a dictionary. The `_functions` dictionary can then be used and accessed using the function name to retrieve a function: `_functions['foo']` points to the `foo()` function.

In the following sections, I will explain how to write your own decorators. Then I'll cover how the built-in decorators provided by Python work and explain how (and when) to use them.

Writing Decorators

As mentioned, decorators are often used when refactoring repeated code around functions. Consider the following set of functions that need to check whether the username they receive as an argument is the admin or not and, if the user is not an admin, raise an exception:

```
class Store(object):
    def get_food(self, username, food):
        if username != 'admin':
            raise Exception("This user is not allowed to get food")
        return self.storage.get(food)

    def put_food(self, username, food):
        if username != 'admin':
            raise Exception("This user is not allowed to put food")
        self.storage.put(food)
```

We can see there's some repeated code here. The obvious first step to making this code more efficient is to factor the code that checks for admin status:

```
❶ def check_is_admin(username):
    if username != 'admin':
        raise Exception("This user is not allowed to get or put food")

class Store(object):
    def get_food(self, username, food):
        check_is_admin(username)
        return self.storage.get(food)

    def put_food(self, username, food):
        check_is_admin(username)
        self.storage.put(food)
```

We've moved the checking code into its own function ❶. Now our code looks a bit cleaner, but we can do even better if we use a decorator, as shown in Listing 7-2.

```
def check_is_admin(f):
 ❶ def wrapper(*args, **kwargs):
        if kwargs.get('username') != 'admin':
            raise Exception("This user is not allowed to get or put food")
        return f(*args, **kwargs)
    return wrapper

class Store(object):
    @check_is_admin
    def get_food(self, username, food):
        return self.storage.get(food)

    @check_is_admin
    def put_food(self, username, food):
        self.storage.put(food)
```

Listing 7-2: Adding a decorator to the factored code

We define our `check_is_admin` decorator ❶ and then call it whenever we need to check for access rights. The decorator inspects the arguments passed to the function using the `kwargs` variable and retrieves the `username` argument, performing the username check before calling the actual function. Using decorators like this makes it easier to manage common functionality. To anyone with much Python experience, this is probably old hat, but what you might not realize is that this naive approach to implementing decorators has some major drawbacks.

Stacking Decorators

You can also use several decorators on top of a single function or method, as shown in Listing 7-3.

```
def check_user_is_not(username):
    def user_check_decorator(f):
        def wrapper(*args, **kwargs):
            if kwargs.get('username') == username:
                raise Exception("This user is not allowed to get food")
            return f(*args, **kwargs)
        return wrapper
    return user_check_decorator
```

```
class Store(object):
    @check_user_is_not("admin")
    @check_user_is_not("user123")
    def get_food(self, username, food):
        return self.storage.get(food)
```

Listing 7-3: Using more than one decorator with a single function

Here, `check_user_is_not()` is a factory function for our decorator `user_check_decorator()`. It creates a function decorator that depends on the `username` variable and then returns that variable. The function `user_check_decorator()` will serve as a function decorator for `get_food()`.

The function `get_food()` gets decorated twice using `check_user_is_not()`. The question here is which username should be checked first—`admin` or `user123`? The answer is in the following code, where I translated Listing 7-3 into equivalent code without using a decorator.

```
class Store(object):
    def get_food(self, username, food):
        return self.storage.get(food)

Store.get_food = check_user_is_not("user123")(Store.get_food)
Store.get_food = check_user_is_not("admin")(Store.get_food)
```

The decorator list is applied from top to bottom, so the decorators closest to the `def` keyword will be applied first and executed last. In the example above, the program will check for `admin` first and then for `user123`.

Writing Class Decorators

It's also possible to implement class decorators, though these are less often used in the wild. *Class decorators* work in the same way as function decorators, but they act on classes rather than functions. The following is an example of a class decorator that sets attributes for two classes:

```
import uuid

def set_class_name_and_id(klass):
    klass.name = str(klass)
    klass.random_id = uuid.uuid4()
    return klass

@set_class_name_and_id
```

```
class SomeClass(object):
    pass
```

When the class is loaded and defined, it will set the `name` and `random_id` attributes, like so:

```
>>> SomeClass.name
"<class '__main__.SomeClass'>"
>>> SomeClass.random_id
UUID('d244dc42-f0ca-451c-9670-732dc32417cd')
```

As with function decorators, this can be handy for factorizing common code that manipulates classes.

Another possible use for class decorators is to wrap a function or class with classes. For example, class decorators are often used for wrapping a function that's storing a state. The following example wraps the `print()` function to check how many times it has been called in a session:

```
class CountCalls(object):
    def __init__(self, f):
        self.f = f
        self.called = 0
    def __call__(self, *args, **kwargs):
        self.called += 1
        return self.f(*args, **kwargs)

@CountCalls
def print_hello():
    print("hello")
```

We can then use this to check how many times the function `print_hello()` has been called:

```
>>> print_hello.called
0
>>> print_hello()
hello
>>> print_hello.called
1
```

Retrieving Original Attributes with the update_wrapper Decorator

As mentioned, a decorator replaces the original function with a new one built on the fly. However, this new function lacks many of the attributes

of the original function, such as its docstring and its name. Listing 7-4 shows how the function `foobar()` loses its docstring and its name attribute once it is decorated with the `is_admin` decorator.

```
>>> def is_admin(f):
...     def wrapper(*args, **kwargs):
...         if kwargs.get('username') != 'admin':
...             raise Exception("This user is not allowed to get food")
...         return f(*args, **kwargs)
...     return wrapper
...
>>> def foobar(username="someone"):
...     """Do crazy stuff."""
...     pass
...
>>> foobar.func_doc
'Do crazy stuff.'
>>> foobar.__name__
'foobar'
>>> @is_admin
... def foobar(username="someone"):
...     """Do crazy stuff."""
...     pass
...
>>> foobar.__doc__
>>> foobar.__name__
'wrapper'
```

Listing 7-4: A decorated function loses its docstring and name attributes.

Not having the correct docstring and name attribute for a function can be problematic in various situations, such as when generating the source code documentation.

Fortunately, the `functools` module in the Python Standard Library solves this problem with the `update_wrapper()` function, which copies the attributes from the original function that were lost to the wrapper itself. The source code of `update_wrapper()` is shown in Listing 7-5.

```
WRAPPER_ASSIGNMENTS = ('__module__', '__name__', '__qualname__', '__doc__',
                       '__annotations__')
WRAPPER_UPDATES = ('__dict__',)
def update_wrapper(wrapper,
                   wrapped,
                   assigned = WRAPPER_ASSIGNMENTS,
                   updated = WRAPPER_UPDATES):
    for attr in assigned:
        try:
            value = getattr(wrapped, attr)
        except AttributeError:
```

```
            pass
        else:
            setattr(wrapper, attr, value)
    for attr in updated:
        getattr(wrapper, attr).update(getattr(wrapped, attr, {}))
    # Issue #17482: set __wrapped__ last so we don't inadvertently copy it
    # from the wrapped function when updating __dict__
    wrapper.__wrapped__ = wrapped
    # Return the wrapper so this can be used as a decorator via partial()
    return wrapper
```

Listing 7-5: The `update_wrapper()` source code

In Listing 7-5, the `update_wrapper()` source code highlights which attributes are worth saving when wrapping a function with a decorator. By default, the `__name__` attribute, `__doc__` attribute, and some other attributes are copied. You can also personalize which attributes of a function are copied to the decorated function. When we use `update_wrapper()` to rewrite our example from Listing 7-4, things are much nicer:

```
>>> def foobar(username="someone"):
...     """Do crazy stuff."""
...     pass
...
>>> foobar = functools.update_wrapper(is_admin, foobar)
>>> foobar.__name__
'foobar'
>>> foobar.__doc__
'Do crazy stuff.'
```

Now the `foobar()` function has the correct name and docstring even when decorated by `is_admin`.

wraps: A Decorator for Decorators

It can get tedious to use `update_wrapper()` manually when creating decorators, so `functools` provides a decorator for decorators called `wraps`. Listing 7-6 shows the `wraps` decorator in use.

```
import functools

def check_is_admin(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        if kwargs.get('username') != 'admin':
            raise Exception("This user is not allowed to get food")
```

```
        return f(*args, **kwargs)
    return wrapper

class Store(object):
    @check_is_admin
    def get_food(self, username, food):
        """Get food from storage."""
        return self.storage.get(food)
```

Listing 7-6: Updating our decorator with `wraps` from `functools`

With `functools.wrap`, the decorator function `check_is_admin()` that returns the `wrapper()` function takes care of copying the docstring, name function, and other information from the function `f` passed as argument. Thus, the decorated function (`get_food()`, in this case) still sees its unchanged signature.

Extracting Relevant Information with inspect

In our examples so far, we have assumed that the decorated function will always have a `username` passed to it as a keyword argument, but that might not be the case. It might instead have a bunch of information from which we need to extract the username to check. With this in mind, we'll build a smarter version of our decorator that can look at the decorated function's arguments and pull out what it needs.

For this, Python has the `inspect` module, which allows us to retrieve a function's signature and operate on it, as shown in Listing 7-7.

```
import functools
import inspect

def check_is_admin(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        func_args = inspect.getcallargs(f, *args, **kwargs)
        if func_args.get('username') != 'admin':
            raise Exception("This user is not allowed to get food")
        return f(*args, **kwargs)
    return wrapper
@check_is_admin
def get_food(username, type='chocolate'):
    return type + " nom nom nom!"
```

Listing 7-7: Using tools from the `inspect` module to extract information

The function that does the heavy lifting here is `inspect.getcallargs()`, which returns a dictionary containing the names and values of the arguments as key-value pairs. In our example, this function returns `{'username': 'admin','type': 'chocolate'}`. That means that our decorator does not have to check whether the `username` parameter is a positional or a keyword argument; all the decorator has to do is look for `username` in the dictionary.

Using `functools.wraps` and the `inspect` module, you should be able to write any custom decorator that you would ever need. However, do not abuse the `inspect` module: while being able to guess what the function will accept as an argument sounds handy, this capability can be fragile, breaking easily when function signatures change. Decorators are a terrific way to implement the *Don't Repeat Yourself* mantra so cherished by developers.

How Methods Work in Python

Methods are pretty simple to use and understand, and you've likely just used them correctly without delving in much deeper than you needed to. But to understand what certain decorators do, you need to know how methods work behind the scenes.

A *method* is a function that is stored as a class attribute. Let's have a look at what happens when we try to access such an attribute directly:

```
>>> class Pizza(object):
...     def __init__(self, size):
...         self.size = size
...     def get_size(self):
...         return self.size
...
>>> Pizza.get_size
<function Pizza.get_size at 0x7fdbfd1a8b90>
```

We are told that `get_size()` is a function—but why is that? The reason is that at this stage, `get_size()` is not tied to any particular object. Therefore, it is treated as a normal function. Python will raise an error if we try to call it directly, like so:

```
>>> Pizza.get_size()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: get_size() missing 1 required positional argument: 'self'
```

Python complains that we have not provided the necessary `self` argument. Indeed, as it is not bound to any object, the `self` argument cannot be set automatically. However, we are able to use the `get_size()` function not only by passing an arbitrary instance of the class to the method if we want to but also by passing *any* object, as long as it has the properties that the method expects to find. Here's an example:

```
>>> Pizza.get_size(Pizza(42))
42
```

This call works, just as promised. It is, however, not very convenient: we have to refer to the class every time we want to call one of its methods.

So Python goes the extra mile for us by binding a class's methods to its instances. In other words, we can access `get_size()` from any `Pizza` instance, and, better still, Python will automatically pass the object itself to the method's `self` parameter, like so:

```
>>> Pizza(42).get_size
<bound method Pizza.get_size of <__main__.Pizza object at 0x7f3138827910>>
>>> Pizza(42).get_size()
42
```

As expected, we do not have to provide any argument to `get_size()`, since it's a bound method: its `self` argument is automatically set to our `Pizza` instance. Here is an even clearer example:

```
>>> m = Pizza(42).get_size
>>> m()
42
```

As long as you have a reference to the bound method, you do not even have to keep a reference to your `Pizza` object. Moreover, if you have a reference to a method but you want to find out which object it is bound to, you can just check the method's `__self__` property, like so:

```
>>> m = Pizza(42).get_size
>>> m.__self__
<__main__.Pizza object at 0x7f3138827910>
>>> m == m.__self__.get_size
True
```

Obviously, we still have a reference to our object, and we can find it if we want.

Static Methods

Static methods belong to a class, rather than an instance of a class, so they don't actually operate on or affect class instances. Instead, a static method operates on the parameters it takes. Static methods are generally used to create utility functions, because they do not depend on the state of the class or its objects.

For example, in Listing 7-8, the static `mix_ingredients()` method belongs to the `Pizza` class but could actually be used to mix ingredients for any other food.

```
class Pizza(object):
    @staticmethod
    def mix_ingredients(x, y):
        return x + y

    def cook(self):
        return self.mix_ingredients(self.cheese, self.vegetables)
```

Listing 7-8: Creating a static method as part of a class

You could write `mix_ingredients()` as a non-static method if you wanted to, but it would take a `self` argument that would never actually be used. Using the `@staticmethod` decorator gives us several things.

The first is speed: Python does not have to instantiate a bound method for each `Pizza` object we create. Bound methods are objects, too, and creating them has a CPU and memory cost—even if it's low. Using a static method lets us avoid that, like so:

```
>>> Pizza().cook is Pizza().cook
False
>>> Pizza().mix_ingredients is Pizza.mix_ingredients
```

```
True
>>> Pizza().mix_ingredients is Pizza().mix_ingredients
True
```

Second, static methods improve the readability of the code. When we see `@staticmethod`, we know that the method does not depend on the state of the object.

Third, static methods can be overridden in subclasses. If instead of a static method, we used a `mix_ingredients()` function defined at the top level of our module, a class inheriting from `Pizza` wouldn't be able to change the way we mix ingredients for our pizza without overriding the `cook()` method itself. With static methods, the subclasses can override the method for their own purposes.

Unfortunately, Python is not always able to detect for itself whether a method is static or not—I call that a defect of the language design. One possible approach is to add a check that detects such pattern and emits a warning using `flake8`. We will look into how to do this in "Extending `flake8` with AST Checks" on page 140.

Class Methods

Class methods are bound to a class rather than its instances. That means that those methods cannot access the state of the object but only the state and methods of the class. Listing 7-9 shows how to write a class method.

```
>>> class Pizza(object):
...     radius = 42
...     @classmethod
...     def get_radius(cls):
...         return cls.radius
...
>>> Pizza.get_radius
<bound method type.get_radius of <class '__main__.Pizza'>>
>>> Pizza().get_radius
<bound method type.get_radius of <class '__main__.Pizza'>>
>>> Pizza.get_radius is Pizza().get_radius
True
>>> Pizza.get_radius()
42
```

Listing 7-9: Binding a class method to its class

As you can see, there are various ways to access the `get_radius()` class method, but however you choose to access it, the method is always bound to the class it is attached to. Also, its first argument must be the class itself. Remember: classes are objects too!

Class methods are principally useful for creating *factory methods*, which instantiate objects using a different signature than `__init__`:

```
class Pizza(object):
    def __init__(self, ingredients):
        self.ingredients = ingredients

    @classmethod
    def from_fridge(cls, fridge):
        return cls(fridge.get_cheese() + fridge.get_vegetables())
```

If we used a `@staticmethod` here instead of a `@classmethod`, we would have to hardcode the `Pizza` class name in our method, making any class inheriting from `Pizza` unable to use our factory for its own purposes. In this case, however, we provide a `from_fridge()` factory method that we can pass a `Fridge` object to. If we call this method with something like `Pizza.from_fridge(myfridge)`, it returns a brand-new `Pizza` with ingredients taken from what's available in `myfridge`.

Any time you write a method that cares only about the class of the object and not about the object's state, it should be declared as a class method.

Abstract Methods

An *abstract method* is defined in an abstract base class that may not itself provide any implementation. When a class has an abstract method, it cannot be instantiated. As a consequence, an *abstract class* (defined as a class that has at least one abstract method) must be used as a parent class by another class. This subclass will be in charge of implementing the abstract method, making it possible to instantiate the parent class.

We can use abstract base classes to make clear the relationships between other, connected classes derived from the base class but make the abstract base class itself impossible to instantiate. By using abstract

base classes, you can ensure the classes derived from the base class implement particular methods from the base class, or an exception will be raised. The following example shows the simplest way to write an abstract method in Python:

```
class Pizza(object):
    @staticmethod
    def get_radius():
        raise NotImplementedError
```

With this definition, any class inheriting from `Pizza` must implement and override the `get_radius()` method; otherwise, calling the method raises the exception shown here. This is handy for making sure that each subclass of `Pizza` implements its own way of computing and returning its radius.

This way of implementing abstract methods has a drawback: if you write a class that inherits from `Pizza` but forget to implement `get_radius()`, the error is raised only if you try to use that method at runtime. Here's an example:

```
>>> Pizza()
<__main__.Pizza object at 0x7fb747353d90>
>>> Pizza().get_radius()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in get_radius
NotImplementedError
```

As `Pizza` is directly instantiable, there's no way to prevent this from happening. One way to make sure you get an early warning about forgetting to implement and override the method, or trying to instantiate an object with abstract methods, is to use Python's built-in `abc` (abstract base classes) module instead, like so:

```
import abc

class BasePizza(object, metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def get_radius(self):
         """Method that should do something."""
```

The `abc` module provides a set of decorators to use on top of methods that will be defined as abstracts and a metaclass to enable this. When you use `abc` and its special `metaclass`, as shown above, instantiating a `BasePizza` or a class inheriting from it that doesn't override `get_radius()` causes a `TypeError`:

```
>>> BasePizza()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class BasePizza with abstract methods
get_radius
```

We try to instantiate the abstract `BasePizza` class and are immediately told it can't be done!

While using abstract methods doesn't guarantee that the method is implemented by the user, this decorator helps you catch the error earlier. This is especially handy when you are providing interfaces that must be implemented by other developers; it's a good documentation hint.

Mixing Static, Class, and Abstract Methods

Each of these decorators is useful on its own, but the time may come when you'll have to use them together.

For example, you could define a factory method as a class method while forcing the implementation to be made in a subclass. In that case, you'd need to have a class method defined as both an abstract method and a class method. This section gives some tips that will help you with that.

First, an abstract method's prototype is not set in stone. When you implement the method, there is nothing stopping you from extending the argument list as you see fit. Listing 7-10 is an example of code in which a subclass extends the signature of the abstract method of its parent.

```
import abc

class BasePizza(object, metaclass=abc.ABCMeta):
```

```
    @abc.abstractmethod
    def get_ingredients(self):
         """Returns the ingredient list."""

class Calzone(BasePizza):
    def get_ingredients(self, with_egg=False):
        egg = Egg() if with_egg else None
        return self.ingredients + [egg]
```

Listing 7-10: Using a subclass to extend the signature of the abstract method of its parent

We define the `Calzone` subclass to inherit from the `BasePizza` class. We can define the `Calzone` subclass's methods any way we like, as long as they support the interface we define in `BasePizza`. This includes implementing the methods as either class or static methods. The following code defines an abstract `get_ingredients()` method in the base class and a static `get_ingredients()` method in the `DietPizza` subclass:

```
import abc

class BasePizza(object, metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def get_ingredients(self):
         """Returns the ingredient list."""

class DietPizza(BasePizza):
    @staticmethod
    def get_ingredients():
        return None
```

Even though our static `get_ingredients()` method doesn't return a result based on the object's state, it supports our abstract `BasePizza` class's interface, so it's still valid.

It is also possible to use the `@staticmethod` and `@classmethod` decorators on top of `@abstractmethod` in order to indicate that a method is, for example, both static and abstract, as shown in Listing 7-11.

```
import abc

class BasePizza(object, metaclass=abc.ABCMeta):

    ingredients = ['cheese']

    @classmethod
    @abc.abstractmethod
    def get_ingredients(cls):
```

```
        """Returns the ingredient list."""
        return cls.ingredients
```

Listing 7-11: Using a class method decorator with abstract methods

The abstract method `get_ingredients()` needs to be implemented by a subclass, but it's also a class method, meaning the first argument it will receive will be a class (not an object).

Note that by defining `get_ingredients()` as a class method in `BasePizza` like this, you are not forcing any subclasses to define `get_ingredients()` as a class method—it could be a regular method. The same would apply if we had defined it as a static method: there's no way to force subclasses to implement abstract methods as a specific kind of method. As we have seen, you can change the signature of an abstract method when implementing it in a subclass in any way you like.

Putting Implementations in Abstract Methods

Hold the phone: in Listing 7-12, we have an implementation *in* an abstract method. Can we *do* that? The answer is yes. Python does not have a problem with it! You can put code in your abstract methods and call it using `super()`, as demonstrated in Listing 7-12.

```
import abc

class BasePizza(object, metaclass=abc.ABCMeta):

    default_ingredients = ['cheese']

    @classmethod
    @abc.abstractmethod
    def get_ingredients(cls):
         """Returns the default ingredient list."""
         return cls.default_ingredients

class DietPizza(BasePizza):
    def get_ingredients(self):
        return [Egg()] + super(DietPizza, self).get_ingredients()
```

Listing 7-12: Using an implementation in an abstract method

In this example, every `Pizza` you make that inherits from `BasePizza` has to override the `get_ingredients()` method, but every `Pizza` also has access to

the base class's default mechanism for getting the ingredients list. This mechanism is especially useful when providing an interface to implement while also providing base code that might be useful to all inheriting classes.

The Truth About super

Python has always allowed developers to use both single and multiple inheritances to extend their classes, but even today, many developers do not seem to understand how these mechanisms, and the `super()` method that is associated with them, work. To fully understand your code, you need to understand the trade-offs.

Multiple inheritances are used in many places, particularly in code involving a mixin pattern. A *mixin* is a class that inherits from two or more other classes, combining their features.

> **NOTE**
>
> *Many of the pros and cons of single and multiple inheritances, composition, or even duck typing are out of scope for this book, so we won't cover everything here. If you are not familiar with these notions, I suggest you read about them to form your own opinions.*

As you should know by now, classes are objects in Python. The construct used to create a class is a special statement that you should be well familiar with: *`class classname(expression of inheritance)`*.

The code in parentheses is a Python expression that returns the list of class objects to be used as the class's parents. Ordinarily, you would specify them directly, but you could also write something like this to specify the list of parent objects:

```
>>> def parent():
...     return object
...
>>> class A(parent()):
...     pass
...
```

```
>>> A.mro()
[<class '__main__.A'>, <type 'object'>]
```

This code works as expected: we declare class `A` with `object` as its parent class. The class method `mro()` returns the *method resolution order* used to resolve attributes—it defines how the next method to call is found via the tree of inheritance between classes. The current MRO system was first implemented in Python 2.3, and its internal workings are described in the Python 2.3 release notes. It defines how the system browses the tree of inheritance between classes to find the method to call.

We already saw that the canonical way to call a method in a parent class is to use the `super()` function, but what you probably don't know is that `super()` is actually a constructor and you instantiate a `super` object each time you call it. It takes either one or two arguments: the first argument is a class, and the second, optional argument is either a subclass or an instance of the first argument.

The object returned by the constructor functions as a proxy for the parent classes of the first argument. It has its own `__getattribute__` method that iterates over the classes in the MRO list and returns the first matching attribute it finds. The `__getattribute__` method is called when an attribute of the `super()` object is retrieved, as shown in Listing 7-13.

```
>>> class A(object):
...     bar = 42
...     def foo(self):
...             pass
...
>>> class B(object):
...     bar = 0
...
>>> class C(A, B):
...     xyz = 'abc'
...
>>> C.mro()
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <type
'object'>]
>>> super(C, C()).bar
42
>>> super(C, C()).foo
<bound method C.foo of <__main__.C object at 0x7f0299255a90>>
>>> super(B).__self__
```

```
>>> super(B, B()).__self__
<__main__.B object at 0x1096717f0>
```

Listing 7-13: The `super()` function is a constructor that instantiates a `super` object.

When requesting an attribute of the `super` object of an instance of `C`, the `__getattribute__` method of the `super()` object walks through the MRO list and returns the attribute from the first class it finds that has the `super` attribute.

In Listing 7-13, we called `super()` with two arguments, meaning we used a *bound* `super` object. If we call `super()` with only one argument, it returns an *unbound* `super` object instead:

```
>>> super(C)
<super: <class 'C'>, NULL>
```

Since no instance has been provided as the second argument, the `super` object cannot be bound to any instance. Therefore, you cannot use this unbound object to access class attributes. If you try, you'll get the following errors:

```
>>> super(C).foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'super' object has no attribute 'foo'
>>> super(C).bar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'super' object has no attribute 'bar'
>>> super(C).xyz
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'super' object has no attribute 'xyz'
```

At first glance, it might seem like this unbound kind of `super` object is useless, but actually the way the `super` class implements the descriptor protocol `__get__` makes unbound `super` objects useful as class attributes:

```
>>> class D(C):
...     sup = super(C)
...
>>> D().sup
<super: <class 'C'>, <D object>>
>>> D().sup.foo
<bound method D.foo of <__main__.D object at 0x7f0299255bd0>>
```

```
>>> D().sup.bar
42
```

The unbound `super` object's `__get__` method is called using the instance `super(C).__get__(D())` and the attribute name `'foo'` as arguments, allowing it to find and resolve `foo`.

NOTE

Even if you've never heard of the descriptor protocol, it's likely you've used it through the `@property` decorator without knowing it. The descriptor protocol *is the mechanism in Python that allows an object stored as an attribute to return something other than itself. This protocol is not covered in this book, but you can find out more about it in the Python data model documentation.*

There are plenty of situations in which using `super()` can be tricky, such as when handling different method signatures along the inheritance chain. Unfortunately, there's no silver bullet for all occasions. The best precaution is to use tricks such as having all your methods accept their arguments using `*args, **kwargs`.

Since Python 3, `super()` has picked up a bit of magic: it can now be called from within a method without any arguments. When no arguments are passed to `super()`, it automatically searches the stack frame for arguments:

```
class B(A):
      def foo(self):
          super().foo()
```

The standard way of accessing parent attributes in subclasses is `super()`, and you should always use it. It allows cooperative calls of parent methods without any surprises, such as parent methods not being called or being called twice when multiple inheritances are used.

Summary

Equipped with what you learned in this chapter, you should be unbeatable on everything that concerns methods definition in Python. Decorators are essential when it comes to code factorization, and proper use of the built-in decorators provided by Python can vastly improve the neatness of your Python code. Abstract classes are especially useful when providing an API to other developers and services.

Class inheritance is not often fully understood, and having an overview of the internal machinery of the language is a good way to fully apprehend how this works. There should be no secrets left on this topic for you now!

8
FUNCTIONAL PROGRAMMING

Many Python developers are unaware of the extent to which you can use functional programming in Python, which is a shame: with few exceptions, functional programming allows you to write more concise and efficient code. Moreover, Python's support for functional programming is extensive.

This chapter will cover some of the functional programming aspects of Python, including creating and using generators. You'll learn about the most useful functional packages and functions available and how to use them in combination to get the most efficient code.

NOTE

If you want to get serious about functional programming, here's my advice: take a break from Python and learn a hugely functional programming language, such as Lisp. I know it might sound strange to talk about Lisp in a Python book, but playing with Lisp for several years taught me how to "think functional." You may not develop the thought processes necessary to make full use of functional programming if all your experience comes from imperative and object-oriented programming. Lisp isn't purely functional itself, but it has more focus on functional programming than you'll find in Python.

Creating Pure Functions

When you write code using a functional style, your functions are designed to have no side effects: instead, they take an input and produce an output without keeping state or modifying anything not reflected in the return value. Functions that follow this ideal are referred to as *purely functional*.

Let's start with an example of a regular, non-pure function that removes the last item in a list:

```
def remove_last_item(mylist):
    """Removes the last item from a list."""
    mylist.pop(-1)  # This modifies mylist
```

The following is a pure version of the same function:

```
def butlast(mylist):

    return mylist[:-1]  # This returns a copy of mylist
```

We define a `butlast()` function to work like `butlast` in Lisp, in that it returns the list without the last element *without* modifying the original list. Instead, it returns a copy of the list that has the modifications in place, allowing us to keep the original.

The practical advantages of functional programming include the following:

Modularity Writing with a functional style forces a certain degree of separation in solving your individual problems and makes sections of code easier to reuse in other contexts. Since the function does not depend on any external variable or state, calling it from a different piece of code is straightforward.

Brevity Functional programming is often less verbose than other paradigms.

Concurrency Purely functional functions are thread-safe and can run concurrently. Some functional languages do this automatically, which can be a big help if you ever need to scale your application, though this is not quite the case yet in Python.

Testability Testing a functional program is incredibly easy: all you need is a set of inputs and an expected set of outputs. They are *idempotent*, meaning that calling the same function over and over with the same arguments will always return the same result.

Generators

A *generator* is an object that behaves like an iterator, in that it generates and returns a value on each call of its `next()` method until a `StopIteration` is raised. Generators, first introduced in PEP 255, offer an easy way to create objects that implement the *iterator protocol*. While writing generators in a functional style is not strictly necessary, doing so makes them easier to write and debug and is a common practice.

To create a generator, just write a regular Python function that contains a `yield` statement. Python will detect the use of `yield` and tag the function as a generator. When execution reaches the `yield` statement, the function returns a value as with a `return` statement, but with one notable difference: the interpreter will save a stack reference, and this will be used to resume the function's execution when the `next()` function is called again.

When functions are executed, the chaining of their execution produces a *stack*—function calls are said to be stacked on each other. When a function returns, it's removed from the stack, and the value it returns is passed to the calling function. In the case of a generator, the function does not really return but *yields* instead. Python therefore saves the state of the function as a stack reference, resuming the execution of the generator at the point it saved when the next iteration of the generator is needed.

Creating a Generator

As mentioned, you create a generator by writing a normal function and including `yield` in the function's body. Listing 8-1 creates a generator called `mygenerator()` that includes three yields, meaning it will iterate with the next three calls to `next()`.

```
>> def mygenerator():
...     yield 1
...     yield 2
...     yield 'a'
...
>>> mygenerator()
<generator object mygenerator at 0x10d77fa50>
>>> g = mygenerator()
>>> next(g)
1
>>> next(g)
2
>>> next(g)
'a'
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Listing 8-1: Creating a generator with three iterations

When it runs out of `yield` statements, `StopIteration` is raised at the next call to `next()`.

In Python, generators keep a reference to the stack when a function yields something, and they resume this stack when a call to `next()` is executed again.

The naive approach when iterating over any data without using generators is to build the entire list first, which often consumes memory wastefully.

Say we want to find the first number between 1 and 10,000,000 that's equal to 50,000. Sounds easy, doesn't it? Let's make this a challenge. We'll run Python with a memory constraint of 128MB and try the naive approach of first building the entire list:

```
$ ulimit -v 131072
$ python3
>>> a = list(range(10000000))
```

This naive method first tries to build the list, but if we run the program so far:

```
Traceback (most recent call last): File "<stdin>", line 1, in <module>
MemoryError
```

Uh-oh. Turns out we can't build a list of 10 million items with only 128MB of memory!

> **WARNING**
>
> *In Python 3, `range()` returns a generator when iterated. To get a generator in Python 2, you have to use `xrange()` instead. This function doesn't exist in Python 3 anymore, since it's redundant.*

Let's try using a generator instead, with the same 128MB restriction:

```
$ ulimit -v 131072
$ python3
>>> for value in range(10000000):
...     if value == 50000:
...             print("Found it")
...             break
...
Found it
```

This time, our program executes without issue. When it is iterated over, the `range()` class returns a generator that dynamically generates our list of integers. Better still, since we are only interested in the 50,000th number, instead of building the full list, the generator only had to generate 50,000 numbers before it stopped.

By generating values on the fly, generators allow you to handle large data sets with minimal consumption of memory and processing cycles. Whenever you need to work with a huge number of values, generators can help you handle them efficiently.

Returning and Passing Values with yield

A `yield` statement also has a less commonly used feature: it can return a value in the same way as a function call. This allows us to pass a value to a generator by calling its `send()` method. As an example of using `send()`, we'll write a function called `shorten()` that takes a list of strings and returns a list consisting of those same strings, only truncated (Listing 8-2).

```
def shorten(string_list):
    length = len(string_list[0])
    for s in string_list:
        length = yield s[:length]

mystringlist = ['loremipsum', 'dolorsit', 'ametfoobar']
shortstringlist = shorten(mystringlist)
result = []
try:
    s = next(shortstringlist)
    result.append(s)
    while True:
        number_of_vowels = len(filter(lambda letter: letter in 'aeiou', s))
        # Truncate the next string depending
        # on the number of vowels in the previous one
        s = shortstringlist.send(number_of_vowels)
        result.append(s)
except StopIteration:
    pass
```

Listing 8-2: Returning and using a value with `send()`

In this example, we've written a function called `shorten()` that takes a list of strings and returns a list consisting of those same strings, only truncated. The length of each truncated string is equal to the number of vowels in the previous string: *loremipsum* has four vowels, so the second value returned by the generator will be the first four letters of *dolorsit*; *dolo* has only two vowels, so *ametfoobar* will be truncated to its first two letters *am*. The generator then stops and raises `StopIteration`. Our generator thus returns:

```
['loremipsum', 'dolo', 'am']
```

Using `yield` and `send()` in this fashion allows Python generators to function like *coroutines* seen in Lua and other languages.

PEP 289 introduced generator expressions, making it possible to build one-line generators using a syntax similar to list comprehension:

```
>>> (x.upper() for x in ['hello', 'world'])
<generator object <genexpr> at 0x7ffab3832fa0>
>>> gen = (x.upper() for x in ['hello', 'world'])
>>> list(gen)
['HELLO', 'WORLD']
```

In this example, `gen` is a generator, just as if we had used the `yield` statement. The `yield` in this case is implicit.

Inspecting Generators

To determine whether a function is considered a generator, use `inspect.isgeneratorfunction()`. In Listing 8-3, we create a simple generator and inspect it.

```
>>> import inspect
>>> def mygenerator():
...     yield 1
...
>>> inspect.isgeneratorfunction(mygenerator)
True
>>> inspect.isgeneratorfunction(sum)
False
```

Listing 8-3: Checking whether a function is a generator

Import the `inspect` package to use `isgeneratorfunction()` and then just pass it the name of the function to inspect. Reading the source code of `inspect.isgeneratorfunction()` gives us some insight into how Python marks functions as being generators (see Listing 8-4).

```
def isgeneratorfunction(object):
    """Return true if the object is a user-defined generator function.

    Generator function objects provides same attributes as functions.

    See help(isfunction) for attributes listing."""

  return bool((isfunction(object) or ismethod(object)) and
              object.func_code.co_flags & CO_GENERATOR)
```

Listing 8-4: Source code of `inspect.isgeneratorfunction()`

The `isgeneratorfunction()` function checks that the object is a function or a method and that its code has the `CO_GENERATOR` flag set. This example shows how easy it is to understand how Python works under the hood.

The `inspect` package provides the `inspect.getgeneratorstate()` function, which gives the current state of the generator. We'll use it on `mygenerator()` here at different points of execution:

```
>>> import inspect
>>> def mygenerator():
...     yield 1
...
>>> gen = mygenerator()
```

```
>>> gen
<generator object mygenerator at 0x7f94b44fec30>
>>> inspect.getgeneratorstate(gen)
❶ 'GEN_CREATED'
>>> next(gen)
1
>>> inspect.getgeneratorstate(gen)
❷ 'GEN_SUSPENDED'
>>> next(gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> inspect.getgeneratorstate(gen)
❸ 'GEN_CLOSED'
```

This allows us to determine whether the generator is waiting to be run for the first time (`GEN_CREATED`) ❶, waiting to be resumed by a call to `next()` (`GEN_SUSPENDED`) ❷, or finished running (`GEN_CLOSED`) ❸. This might come in handy to debug your generators.

List Comprehensions

List comprehension, or *listcomp* for short, allows you to define a list's contents inline with its declaration. To make a list into a listcomp, you must wrap it in square brackets as usual, but also include an expression that will generate the items in the list and a `for` loop to loop through them.

The following example creates a list without using list comprehension:

```
>>> x = []
>>> for i in (1, 2, 3):
...     x.append(i)
...
>>> x
[1, 2, 3]
```

And this next example uses list comprehension to make the same list with a single line:

```
>>> x = [i for i in (1, 2, 3)]
>>> x
[1, 2, 3]
```

Using a list comprehension presents two advantages: code written using listcomps is usually shorter and therefore compiles down to fewer operations for Python to perform. Rather than creating a list and calling `append` over and over, Python can just create the list of items and move them into a new list in a single operation.

You can use multiple `for` statements together and use `if` statements to filter out items. Here we create a list of words and use list comprehension to capitalize each item, split up items with multiple words into single words, and delete the extraneous *or* :

```
x = [word.capitalize()
     for line in ("hello world?", "world!", "or not")
     for word in line.split()
     if not word.startswith("or")]
>>> x
['Hello', 'World?', 'World!', 'Not']
```

This code has two `for` loops: the first iterates over the text lines, while the second iterates over words in each of those lines. The final `if` statement filters out words that start with *or* to exclude them from the final list.

Using list comprehension rather than `for` loops is a neat way to define lists quickly. Since we're still talking about functional programming, it's worth noting that lists built through list comprehension shouldn't rely on changing the program's state: you are not expected to modify any variable while building the list. This usually makes the lists more concise and easier to read than lists made without listcomp.

Note that there's also syntax for building dictionaries or sets in the same fashion, like so:

```
>>> {x:x.upper() for x in ['hello', 'world']}
{'world': 'WORLD', 'hello': 'HELLO'}
>>> {x.upper() for x in ['hello', 'world']}
set(['WORLD', 'HELLO'])
```

Functional Functions Functioning

You might repeatedly encounter the same set of problems when manipulating data using functional programming. To help you deal with this situation efficiently, Python includes a number of functions for functional programming. This section will give you a quick overview of some of these built-in functions that allow you to build fully functional programs. Once you have an idea of what's available, I encourage you to research further and try out functions where they might apply in your own code.

Applying Functions to Items with map()

The `map()` function takes the form `map(function, iterable)` and applies `function` to each item in `iterable` to return a list in Python 2 or an iterable `map` object in Python 3, as shown in Listing 8-5.

```
>>> map(lambda x: x + "bzz!", ["I think", "I'm good"])
<map object at 0x7fe7101abdd0>
>>> list(map(lambda x: x + "bzz!", ["I think", "I'm good"]))
['I thinkbzz!', "I'm goodbzz!"]
```

Listing 8-5: Using `map()` in Python 3

You could write an equivalent of `map()` using list comprehension, like this:

```
>>> (x + "bzz!" for x in ["I think", "I'm good"])
<generator object <genexpr> at 0x7f9a0d697dc0>
>>> [x + "bzz!" for x in ["I think", "I'm good"]]
['I thinkbzz!', "I'm goodbzz!"]
```

Filtering Lists with filter()

The `filter()` function takes the form `filter(function or None, iterable)` and filters the items in `iterable` based on the result returned by `function`. This will return a list in Python 2 or an iterable `filter` object in Python 3:

```
>>> filter(lambda x: x.startswith("I "), ["I think", "I'm good"])
<filter object at 0x7f9a0d636dd0>
>>> list(filter(lambda x: x.startswith("I "), ["I think", "I'm good"]))
['I think']
```

You could also write an equivalent of `filter()` using list comprehension, like so:

```
>>> (x for x in ["I think", "I'm good"] if x.startswith("I "))
<generator object <genexpr> at 0x7f9a0d697dc0>
>>> [x for x in ["I think", "I'm good"] if x.startswith("I ")]
['I think']
```

Getting Indexes with enumerate()

The `enumerate()` function takes the form `enumerate(`*`iterable`*`[, `*`start`*`])` and returns an iterable object that provides a sequence of tuples, each consisting of an integer index (starting with `start`, if provided) and the corresponding item in `iterable`. This function is useful when you need to write code that refers to array indexes. For example, instead of writing this:

```
i = 0
while i < len(mylist):
    print("Item %d: %s" % (i, mylist[i]))
    i += 1
```

you could accomplish the same thing more efficiently with `enumerate()`, like so:

```
for i, item in enumerate(mylist):
    print("Item %d: %s" % (i, item))
```

Sorting a List with sorted()

The `sorted()` function takes the form `sorted(`*`iterable`*`, key=None, reverse=False)` and returns a sorted version of `iterable`. The `key` argument allows you to provide a function that returns the value to sort on, as shown here:

```
>>> sorted([("a", 2), ("c", 1), ("d", 4)])
[('a', 2), ('c', 1), ('d', 4)]
>>> sorted([("a", 2), ("c", 1), ("d", 4)], key=lambda x: x[1])
[('c', 1), ('a', 2), ('d', 4)]
```

Finding Items That Satisfy Conditions with any() and all()

The `any(iterable)` and `all(iterable)` functions return a Boolean depending on the values returned by `iterable`. These simple functions are equivalent to the following full Python code:

```
def all(iterable):
    for x in iterable:
        if not x:
            return False
    return True

def any(iterable):
    for x in iterable:
        if x:
            return True
    return False
```

These functions are useful for checking whether any or all of the values in an iterable satisfy a given condition. For example, the following checks a list for two conditions:

```
mylist = [0, 1, 3, -1]
if all(map(lambda x: x > 0, mylist)):
    print("All items are greater than 0")
if any(map(lambda x: x > 0, mylist)):
    print("At least one item is greater than 0")
```

The difference here is that `any()` returns `True` when at least one element meets the condition, while `all()` returns `True` only if every element meets the condition. The `all()` function will also return `True` for an empty iterable, since none of the elements is `False`.

Combining Lists with zip()

The `zip()` function takes the form `zip(iter1 [,iter2 [...]])`. It takes multiple sequences and combines them into tuples. This is useful when you need to combine a list of keys and a list of values into a dict. As with the other functions described here, `zip()` returns a list in Python 2 and an iterable in Python 3. Here we map a list of keys to a list of values to create a dictionary:

```
>>> keys = ["foobar", "barzz", "ba!"]
>>> map(len, keys)
<map object at 0x7fc1686100d0>
>>> zip(keys, map(len, keys))
<zip object at 0x7fc16860d440>
>>> list(zip(keys, map(len, keys)))
[('foobar', 6), ('barzz', 5), ('ba!', 3)]
>>> dict(zip(keys, map(len, keys)))
{'foobar': 6, 'barzz': 5, 'ba!': 3}
```

FUNCTIONAL FUNCTIONS IN PYTHON 2 AND 3

You might have noticed by now how the return types differ between Python 2 and Python 3. Most of Python's purely functional built-in functions return a list rather than an iterable in Python 2, making them less memory efficient than their Python 3.*x* equivalents. If you're planning to write code using these functions, keep in mind that you'll get the most benefit out of them in Python 3. If you're stuck with Python 2, don't despair: the `itertools` module from the Standard Library provides an iterator-based version of many of these functions (`itertools.izip()`, `itertools.imap()`, `itertools.ifilter()`, and so on).

A Common Problem Solved

There's one important tool still to cover. Often when working with lists we want to find the first item that satisfies a specific condition. We'll look at the many ways to accomplish this and then see the most efficient way: the `first` package.

Finding the Item with Simple Code

We might be able to find the first item to satisfy a condition with a function like this:

```
def first_positive_number(numbers):
    for n in numbers:
        if n > 0:
            return n
```

We could rewrite the `first_positive_number()` function in functional style like this:

```
def first(predicate, items):
    for item in items:
        if predicate(item):
            return item

first(lambda x: x > 0, [-1, 0, 1, 2])
```

By using a functional approach where the predicate is passed as argument, the function becomes easily reusable. We could even write it more concisely, like so:

```
# Less efficient
list(filter(lambda x: x > 0, [-1, 0, 1, 2]))[0]
# Efficient
next(filter(lambda x: x > 0, [-1, 0, 1, 2]))
```

Note that this may raise an `IndexError` if no items satisfy the condition, causing `list(filter())` to return an empty list.

For simple cases, you can rely on `next()` to prevent `IndexError` from occurring, like so:

```
>>> a = range(10)
>>> next(x for x in a if x > 3)
4
```

Listing 8-6 will raise `StopIteration` if a condition can never be satisfied. This too can be solved by adding a second argument of `next()`, like so.

```
>>> a = range(10)
>>> next((x for x in a if x > 10), 'default')
'default'
```

Listing 8-6: Returning a default value when the condition is not met

This will return a default value rather than an error when a condition cannot be met. Lucky for us, Python provides a package to handle all of this for us.

Finding the Item Using first()

Rather than writing out the function from Listing 8-6 in all of your programs, you can include the small Python package `first`. Listing 8-7 shows how this package lets you find the first element of an iterable matching a condition.

```
>>> from first import first
>>> first([0, False, None, [], (), 42])
42
>>> first([-1, 0, 1, 2])
-1
>>> first([-1, 0, 1, 2], key=lambda x: x > 0)
1
```

Listing 8-7: Finding the first item in a list that satisfies a condition

You see that the `first()` function returns the first valid, non-empty item in a list.

Using lambda() with functools

You'll notice that we've used `lambda()` in a good portion of the examples so far in this chapter. The `lambda()` function was added to Python to facilitate functional programming functions such as `map()` and `filter()`, which otherwise would have required writing an entirely new function every time you wanted to check a different condition. Listing 8-8 is equivalent to Listing 8-7 but is written without using `lambda()`.

```
import operator
from first import first

def greater_than_zero(number):
    return number > 0

first([-1, 0, 1, 2], key=greater_than_zero)
```

Listing 8-8: Finding the first item to meet the condition, without using `lambda()`

This code works identically to that in Listing 8-7, returning the first non-empty value in a list to meet the condition, but it's a good deal more cumbersome: if we wanted to get the first number in the sequence that's longer than, say, 42 items, we'd need to define an appropriate function via `def` rather than defining it inline with our call to `first()`.

But despite its usefulness in helping us avoid situations like this, `lambda` still has its problems. The `first` module contains a `key` argument that can be used to provide a function that receives each item as an argument and returns a Boolean indicating whether it satisfies the condition. However, we can't pass a `key` function, as it would require more than a single line of code: a `lambda` statement cannot be written on more than one line. That is a significant limitation of `lambda`.

Instead, we would have to go back to the cumbersome pattern of writing new function definitions for each `key` we need. Or would we?

The `functools` package comes to the rescue with its `partial()` method, which provides us with a more flexible alternative to `lambda`. The `functools.partial()` method allows us to create a wrapper function with a twist: rather than changing the behavior of a function, it instead changes the *arguments* it receives, like so:

```
   from functools import partial
   from first import first

❶ def greater_than(number, min=0):
       return number > min

❷ first([-1, 0, 1, 2], key=partial(greater_than, min=42))
```

Here we create a new `greater_than()` function that works just like the old `greater_than_zero()` from Listing 8-8 by default, but this version allows us to specify the value we want to compare our numbers to, whereas before it was hardcoded. Here, we pass `functools.partial()` to our function and the value we want for `min` ❶, and we get back a new function that has `min` set to 42, just as we want ❷. In other words, we can write a function and use `functools.partial()` to customize the behavior of our new functions to suit our needs in any given situation.

Even this version can be pared down. All we're doing in this example is comparing two numbers, and as it turns out, the `operator` module has built-in functions for exactly that:

```
import operator
from functools import partial
from first import first
```

```
first([-1, 0, 1, 2], key=partial(operator.le, 0))
```

This is a good example of `functools.partial()` working with positional arguments. In this case, the function `operator.le(a, b)`, which takes two numbers and returns a Boolean that tells us whether the first number is less than or equal to the second, is passed to `functools.partial()`. The 0 we pass to `functools.partial()` gets assigned to `a`, and the argument passed to the function returned by `functools.partial()` gets assigned to `b`. So this works identically to Listing 8-8 but without using `lambda` or defining any additional functions.

> **NOTE**
>
> *The `functools.partial()` method is typically useful in place of `lambda` and should be considered a superior alternative. The `lambda` function is something of an anomaly in the Python language, and dropping it altogether was considered for Python 3 due to the function's limited body size of a single line.*

Useful itertools Functions

Finally, we'll look at some useful functions in the `itertools` module in the Python Standard Library that you should be aware of. Too many programmers end up writing their own versions of these functions simply because they aren't aware that Python provides them out of the box. They are all designed to help you manipulate `iterator` (that's why the module is called *iter-tools*) and therefore are all purely functional. Here I'll list a few of them and give a brief overview of what they do, and I encourage you to look into them further if they seem of use.

- `accumulate(`*`iterable`*`[, `*`func`*`])` returns a series of accumulated sums of items from iterables.
- `chain(*`*`iterables`*`)` iterates over multiple iterables, one after another, without building an intermediate list of all items.

- `combinations(iterable, r)` generates all combinations of length `r` from the given `iterable`.
- `compress(data, selectors)` applies a Boolean mask from `selectors` to `data` and returns only the values from `data` where the corresponding element of `selectors` is `True`.
- `count(start, step)` generates an endless sequence of values, starting with `start` and incrementing `step` at a time with each call.
- `cycle(iterable)` loops repeatedly over the values in `iterable`.
- `repeat(elem[, n])` repeats an element `n` times.
- `dropwhile(predicate, iterable)` filters elements of an iterable starting from the beginning until `predicate` is `False`.
- `groupby(iterable, keyfunc)` creates an iterator that groups items by the result returned by the `keyfunc()` function.
- `permutations(iterable[, r])` returns successive `r`-length permutations of the items in `iterable`.
- `product(*iterables)` returns an iterable of the Cartesian product of `iterables` without using a nested `for` loop.
- `takewhile(predicate, iterable)` returns elements of an iterable starting from the beginning until `predicate` is `False`.

These functions are particularly useful in conjunction with the `operator` module. When used together, `itertools` and `operator` can handle most situations that programmers typically rely on `lambda` for. Here's an example of using `operator.itemgetter()` instead of writing `lambda x: x['foo']`:

```
>>> import itertools
>>> a = [{'foo': 'bar'}, {'foo': 'bar', 'x': 42}, {'foo': 'baz', 'y': 43}]
>>> import operator
>>> list(itertools.groupby(a, operator.itemgetter('foo')))
[('bar', <itertools._grouper object at 0xb000d0>), ('baz', <itertools._grouper
object at
0xb00110>)]
>>> [(key, list(group)) for key, group in itertools.groupby(a,
operator.itemgetter('foo'))]
[('bar', [{'foo': 'bar'}, {'x': 42, 'foo': 'bar'}]), ('baz', [{'y': 43, 'foo':
'baz'}])]
```

In this case, we could have also written `lambda x: x['foo']`, but using `operator` lets us avoid having to use `lambda` at all.

Summary

While Python is often advertised as being object oriented, it can be used in a very functional manner. A lot of its built-in concepts, such as generators and list comprehension, are functionally oriented and don't conflict with an object-oriented approach. They also limit the reliance on a program's global state, for your own good.

Using functional programming as a paradigm in Python can help you make your program more reusable and easier to test and debug, supporting the Don't Repeat Yourself (DRY) mantra. In this spirit, the standard Python modules `itertools` and `operator` are good tools to improve the readability of your functional code.

9
THE ABSTRACT SYNTAX TREE, HY, AND LISP-LIKE ATTRIBUTES

The *abstract syntax tree (AST)* is a representation of the structure of the source code of any programming language. Every language, including Python, has a specific AST; Python's AST is built by parsing a Python source file. Like any tree, this one is made of nodes linked together. A node can represent an operation, a statement, an expression, or even a module. Each node can contain references to other nodes that make up the tree.

Python's AST is not heavily documented and is thus hard to deal with at first glance, but understanding some deeper aspects of how Python is constructed can help you master its usage.

This chapter will examine the AST of some simple Python commands to get you familiar with the structure and how it's used. Once you're familiar with the AST, we'll build a program that can check for wrongly declared methods using `flake8` and the AST. Finally, we'll look at Hy, a Python-Lisp hybrid language built on the Python AST.

Looking at the AST

The easiest way to view the Python AST is to parse some Python code and dump the generated AST. For that, the Python `ast` module provides

everything you need, as shown in Listing 9-1.

```
>>> import ast
>>> ast.parse
<function parse at 0x7f062731d950>
>>> ast.parse("x = 42")
<_ast.Module object at 0x7f0628a5ad10>
>>> ast.dump(ast.parse("x = 42"))
"Module(body=[Assign(targets=[Name(id='x', ctx=Store())], value=Num(n=42))])"
```

Listing 9-1: Using the `ast` module to dump the AST generated by parsing code

The `ast.parse()` function parses any string that contains Python code and returns an `_ast.Module` object. That object is actually the root of the tree: you can browse it to discover every node making up the tree. To visualize what the tree looks like, you can use the `ast.dump()` function, which will return a string representation of the whole tree.

In Listing 9-1, the code `x = 42` is parsed with `ast.parse()`, and the result is printed using `ast.dump()`. This abstract syntax tree can be rendered as shown in Figure 9-1, which shows the structure of the Python `assign` command.

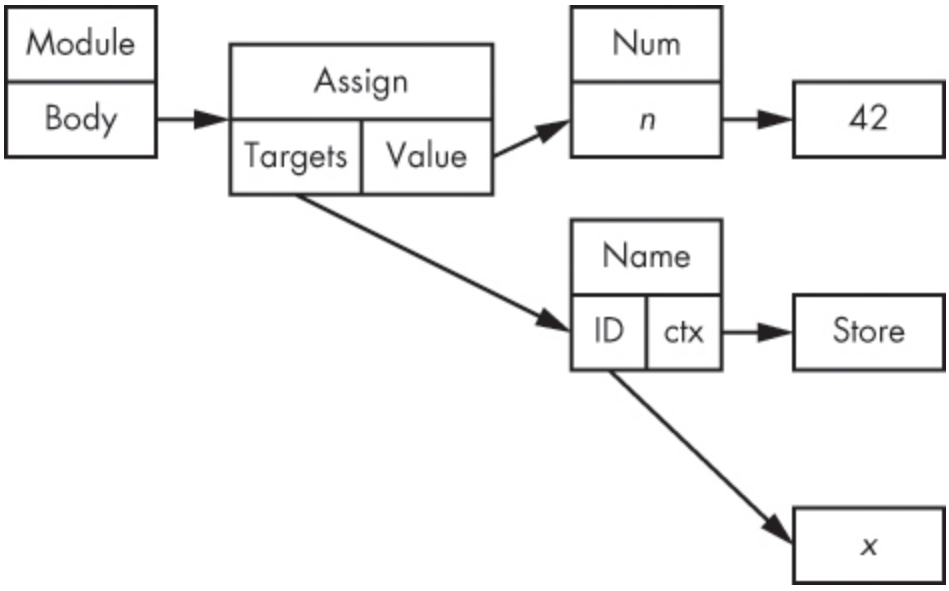


Figure 9-1: The AST of the `assign` command in Python

The AST always starts with a root element, which is usually an `_ast.Module` object. This module object contains a list of statements or

expressions to evaluate in its *body* attribute and usually represents the content of a file.

As you can probably guess, the `ast.Assign` object shown in Figure 9-1 represents an *assignment*, which is mapped to the `=` sign in the Python syntax. An `ast.Assign` object has a list of *targets* and a *value* to set the targets to. The list of targets in this case consists of one object, `ast.Name`, which represents a variable whose ID is *x*. The value is a number *n* with a value (in this case) 42. The ctx attribute stores a *context*, either `ast.Store` or `ast.Load`, depending on whether the variable is being used for reading or writing. In this case, the variable is being assigned a value, so an `ast.Store` context is used.

We could pass this AST to Python to be compiled and evaluated via the built-in `compile()` function. This function takes an AST as argument, the source filename, and a mode (either `'exec'`, `'eval'`, or `'single'`). The source filename can be any name that you want your AST to appear to be from; it is common to use the string `<input>` as the source filename if the data does not come from a stored file, as shown in Listing 9-2.

```
>>> compile(ast.parse("x = 42"), '<input>', 'exec')
<code object <module> at 0x111b3b0, file "<input>", line 1>
>>> eval(compile(ast.parse("x = 42"), '<input>', 'exec'))
>>> x
42
```

Listing 9-2: Using the `compile()` function to compile data that is not from a stored file

The modes stand for execute (`exec`), evaluate (`eval`), and single statement (`single`). The mode should match what has been given to `ast.parse()`, whose default is `exec`.

- The `exec` mode is the normal Python mode, used when an `_ast.Module` is the root of the tree.
- The `eval` mode is a special mode that expects a single `ast.Expression` as the tree.
- Finally, `single` is another special mode that expects a single statement or expression. If it gets an expression, `sys.displayhook()` will be called with the result, as when code is run in the interactive shell.

The root of the AST is `ast.Interactive`, and its `body` attribute is a list of nodes.

We could build an AST manually using the classes provided in the `ast` module. Obviously, this is a very long way to write Python code and not a method I would recommend! Nonetheless, it's fun to do and helpful for learning about the AST. Let's see what programming with the AST would look like.

Writing a Program Using the AST

Let's write a good old `"Hello world!"` program in Python by building an abstract syntax tree manually.

```
❶ >>> hello_world = ast.Str(s='hello world!', lineno=1, col_offset=1)
❷ >>> print_name = ast.Name(id='print', ctx=ast.Load(), lineno=1, col_offset=1)
❸ >>> print_call = ast.Call(func=print_name, ctx=ast.Load(),
  ... args=[hello_world], keywords=[], lineno=1, col_offset=1)
❹ >>> module = ast.Module(body=[ast.Expr(print_call,
  ... lineno=1, col_offset=1)], lineno=1, col_offset=1)
❺ >>> code = compile(module, '', 'exec')
  >>> eval(code)
  hello world!
```

Listing 9-3: Writing `hello world!` using the AST

In Listing 9-3, we build the tree one leaf at a time, where each leaf is an element (whether a value or an instruction) of the program.

The first leaf is a simple string ❶: the `ast.Str` represents a literal string, which here contains the `hello world!` text. The `print_name` variable ❷ contains an `ast.Name` object, which refers to a variable—in this case, the `print` variable that points to the `print()` function.

The `print_call` variable ❸ contains a function call. It refers to the function name to call, the regular arguments to pass to the function call, and the keyword arguments. Which arguments are used depend on the functions being called. In this case, since it's the `print()` function, we'll pass the string we made and stored in `hello_world`.

At last, we create an `_ast.Module` object ❹ to contain all this code as a list of one expression. We can compile `_ast.Module` objects using the `compile()` function ❺, which parses the tree and generates a native `code` object. These `code` objects are compiled Python code and can finally be executed by a Python virtual machine using `eval`!

This whole process is exactly what happens when you run Python on a *.py* file: once the text tokens are parsed, they are converted into a tree of `ast` objects, compiled, and evaluated.

NOTE

The arguments `lineno` and `col_offset` represent the line number and column offset, respectively, of the source code that has been used to generate the AST. It doesn't make much sense to set these values in this context since we are not parsing a source file, but it can be useful to be able to find the position of the code that generated the AST. For example, Python uses this information when generating backtraces. Indeed, Python refuses to compile an AST object that doesn't provide this information, so we pass fake values to these. You could also use the `ast.fix_missing_locations()` function to set the missing values to the ones set on the parent node.

The AST Objects

You can view the whole list of objects available in the AST by reading the `_ast` module documentation (note the underscore).

The objects are organized into two main categories: statements and expressions. *Statements* include types such as `assert`, assignment (`=`), augmented assignment (`+=`, `/=`, etc.), `global`, `def`, `if`, `return`, `for`, `class`, `pass`, `import`, `raise`, and so forth. Statements inherit from `ast.stmt`; they influence the control flow of a program and are often composed of expressions.

Expressions include types such as `lambda`, `number`, `yield`, `name` (variable), `compare`, and `call`. Expressions inherit from `ast.expr`; they differ from

statements in that they usually produce a value and have no impact on the program flow.

There are also a few smaller categories, such as the `ast.operator` class, which defines standard operators such as *add* (`+`), *div* (`/`), and *right shift* (`>>`), and the `ast.cmpop` module, which defines comparisons operators.

The simple example here should give you an idea of how to build an AST from scratch. It's easy to then imagine how you might leverage this AST to construct a compiler that would parse strings and generate code, allowing you to implement your own syntax to Python! This is exactly what led to the development of the Hy project, which we'll discuss later in this chapter.

Walking Through an AST

To follow how a tree is built or access particular nodes, you sometimes need to walk through your tree, browsing it and iterating over the nodes. You can do this with the `ast.walk()` function. Alternatively, the `ast` module also provides `NodeTransformer`, a class that you can subclass to walk through an AST and modify particular nodes. Using `NodeTransformer` makes it easy to change code dynamically, as shown in Listing 9-4.

```
   import ast

   class ReplaceBinOp(ast.NodeTransformer):
       """Replace operation by addition in binary operation"""
       def visit_BinOp(self, node):
           return ast.BinOp(left=node.left,
                            op=ast.Add(),
                            right=node.right)

❶ tree = ast.parse("x = 1/3")
   ast.fix_missing_locations(tree)
   eval(compile(tree, '', 'exec'))
   print(ast.dump(tree))
❷ print(x)

❸ tree = ReplaceBinOp().visit(tree)
   ast.fix_missing_locations(tree)
   print(ast.dump(tree))
   eval(compile(tree, '', 'exec'))
❹ print(x)
```

Listing 9-4: Walking a tree with `NodeTransformer` to alter a node

The first `tree` object built ❶ is an AST that represents the expression `x = 1/3`. Once this is compiled and evaluated, the result of printing `x` at the end of the function ❷ is `0.33333`, the expected result of `1/3`.

The second `tree` object ❸ is an instance of `ReplaceBinOp`, which inherits from `ast.NodeTransformer`. It implements its own version of the `ast.NodeTransformer.visit()` method and changes any `ast.BinOp` operation to an `ast.BinOp` that executes `ast.Add`. Concretely, this changes any binary operator (`+`, `-`, `/`, and so on) to the `+` operator. When this second tree is compiled and evaluated ❹, the result is now `4`, which is the result of `1 + 3`, because the `/` in the first object is replaced with `+`.

You can see the execution of the program here:

```
Module(body=[Assign(targets=[Name(id='x', ctx=Store())],
                    value=BinOp(left=Num(n=1), op=Div(), right=Num(n=3)))])
0.3333333333333333
Module(body=[Assign(targets=[Name(id='x', ctx=Store())],
                    value=BinOp(left=Num(n=1), op=Add(), right=Num(n=3)))])
4
```

NOTE

If you need to evaluate a string that should return a simple data type, you can use `ast.literal_eval`. As a safer alternative to `eval`, it prevents the input string from executing any code.

Extending flake8 with AST Checks

In Chapter 7, you learned that methods that do not rely on the object state should be declared static with the `@staticmethod` decorator. The problem is that a lot of developers simply forget to do so. I've personally spent too much time reviewing code and asking people to fix this problem.

We've seen how to use `flake8` to do some automatic checking in the code. In fact, `flake8` is extensible and can provide even more checks.

We'll write a `flake8` extension that checks for static method declaration omission by analyzing the AST.

Listing 9-5 shows an example of one class that omits the static declaration and one that correctly includes it. Write this program out and save it as *ast_ext.py*; we'll use it in a moment to write our extension.

```
class Bad(object):
    # self is not used, the method does not need
    # to be bound, it should be declared static
    def foo(self, a, b, c):
        return a + b - c

class OK(object):
    # This is correct
    @staticmethod
    def foo(a, b, c):
        return a + b - c
```

Listing 9-5: Omitting and including `@staticmethod`

Though the `Bad.foo` method works fine, strictly speaking it is more correct to write it as `OK.foo` (turn back to Chapter 7 for more detail on why). To check whether all the methods in a Python file are correctly declared, we need to do the following:

- Iterate over all the statement nodes of the AST.
- Check that the statement is a class definition (`ast.ClassDef`).
- Iterate over all the function definitions (`ast.FunctionDef`) of that class statement to check whether it is already declared with `@staticmethod`.
- If the method is not declared static, check whether the first argument (`self`) is used somewhere in the method. If `self` is not used, the method can be tagged as potentially miswritten.

The name of our project will be `ast_ext`. To register a new plugin in `flake8`, we need to create a packaged project with the usual *setup.py* and *setup.cfg* files. Then, we just need to add an entry point in the *setup.cfg* of our `ast_ext` project.

```
[entry_points]
flake8.extension =
    --snip--
```

```
    H904 = ast_ext:StaticmethodChecker
    H905 = ast_ext:StaticmethodChecker
```

Listing 9-6: Allowing `flake8` plugins for our chapter

In Listing 9-6, we also register two `flake8` error codes. As you'll notice later, we are actually going to add an extra check to our code while we're at it!

The next step is to write the plugin.

Writing the Class

Since we are writing a `flake8` check of the AST, the plugin needs to be a class following a certain signature, as shown in Listing 9-7.

```
class StaticmethodChecker(object):
    def __init__(self, tree, filename):
        self.tree = tree

    def run(self):
        pass
```

Listing 9-7: The class for checking the AST

The default template is easy to understand: it stores the tree locally for use in the `run()` method, which will *yield* the problems that are discovered. The value that will be yielded must follow the expected PEP 8 signature: a tuple of the form (*`lineno`*, *`col_offset`*, *`error_string`*, *`code`*).

Ignoring Irrelevant Code

As indicated earlier, the `ast` module provides the `walk()` function, which allows you to iterate easily on a tree. We'll use that to walk through the AST and find out what to check and what not to check.

First, let's write a loop that ignores the statements that are not class definitions. Add this to your `ast_ext` project, as shown in Listing 9-8; code that should stay the same is grayed out.

```
class StaticmethodChecker(object):
    def __init__(self, tree, filename):
        self.tree = tree
```

```
    def run(self):
        for stmt in ast.walk(self.tree):
            # Ignore non-class
            if not isinstance(stmt, ast.ClassDef):
                continue
```

Listing 9-8: Ignoring statements that are not class definitions

The code in Listing 9-8 is still not checking for anything, but now it knows how to ignore statements that are not class definitions. The next step is to set our checker to ignore anything that is not a function definition.

```
for stmt in ast.walk(self.tree):
    # Ignore non-class
    if not isinstance(stmt, ast.ClassDef):
        continue
    # If it's a class, iterate over its body member to find methods
    for body_item in stmt.body:
        # Not a method, skip
        if not isinstance(body_item, ast.FunctionDef):
            continue
```

Listing 9-9: Ignoring statements that are not function definitions

In Listing 9-9, we ignore irrelevant statements by iterating over the attributes of the class definition.

Checking for the Correct Decorator

We're all set to write the checking method, which is stored in the `body_item` attribute. First, we need to check whether the method that's being checked is already declared as static. If it is, we don't have to do any further checking and can bail out.

```
for stmt in ast.walk(self.tree):
    # Ignore non-class
    if not isinstance(stmt, ast.ClassDef):
        continue
    # If it's a class, iterate over its body member to find methods
    for body_item in stmt.body:
        # Not a method, skip
        if not isinstance(body_item, ast.FunctionDef):
            continue
        # Check that it has a decorator
        for decorator in body_item.decorator_list:
            if (isinstance(decorator, ast.Name)
```

```
               and decorator.id == 'staticmethod'):
                # It's a static function, it's OK
                break
        else:
            # Function is not static, we do nothing for now
            Pass
```

Listing 9-10: Checking for the static decorator

Note that in Listing 9-10, we use the special `for/else` form of Python, where the `else` is evaluated unless we use `break` to exit the `for` loop. At this point, we're able to detect whether a method is declared static.

Looking for self

The next step is to check whether the method that *isn't* declared as static uses the `self` argument. First, check whether the method includes any arguments at all, as shown in Listing 9-11.

```
--snip--
        # Check that it has a decorator
        for decorator in body_item.decorator_list:
            if (isinstance(decorator, ast.Name)
               and decorator.id == 'staticmethod'):
                # It's a static function, it's OK
                break
        else:
            try:
                first_arg = body_item.args.args[0]
            except IndexError:
                yield (
                    body_item.lineno,
                    body_item.col_offset,
                    "H905: method misses first argument",
                    "H905",
                )
                # Check next method
                Continue
```

Listing 9-11: Checking the method for arguments

We finally added a check! This `try` statement in Listing 9-11 grabs the first argument from the method signature. If the code fails to retrieve the first argument from the signature because a first argument doesn't exist, we already know there's a problem: you can't have a bound method without the `self` argument. If the plugin detects that case, it

raises the `H905` error code we set earlier, signaling a method that misses its first argument.

> **NOTE**
>
> *PEP 8 codes follow a particular format for error codes (a letter followed by a number), but there are no rules as to which code to pick. You could come up with any other code for this error, as long as it's not already used by PEP 8 or another extension.*

Now you know why we registered two error codes in *setup.cfg*: we had a good opportunity to kill two birds with one stone.

The next step is to check whether the `self` argument is used in the code of the method.

```
--snip--
            try:
                first_arg = body_item.args.args[0]
            except IndexError:
                yield (
                    body_item.lineno,
                    body_item.col_offset,
                    "H905: method misses first argument",
                    "H905",
                )
                # Check next method
                continue
            for func_stmt in ast.walk(body_item):
                # The checking method must differ between Python 2 and Python 3
                if six.PY3:
                    if (isinstance(func_stmt, ast.Name)
                       and first_arg.arg == func_stmt.id):
                        # The first argument is used, it's OK
                        break
                else:
                    if (func_stmt != first_arg
                       and isinstance(func_stmt, ast.Name)
                       and func_stmt.id == first_arg.id):
                        # The first argument is used, it's OK
                        break
            else:
                yield (
                    body_item.lineno,
                    body_item.col_offset,
                    "H904: method should be declared static",
                    "H904",
                )
```

Listing 9-12: Checking the method for the `self` argument

To check whether the `self` argument is used in the method's body, the plugin in Listing 9-12 iterates recursively, using `ast.walk` on the body and looking for the use of the variable named `self`. If the variable isn't found, the program finally yields the `H904` error code. Otherwise, nothing happens, and the code is considered sane.

> **NOTE**
>
> *As you may have noticed, the code walks over the module AST definition several times. There might be some degree of optimization to browsing the AST in only one pass, but I'm not sure it's worth it, given how the tool is actually used. I'll leave that exercise to you, dear reader.*

Knowing the Python AST is not strictly necessary for using Python, but it does give powerful insight into how the language is built and how it works. It thus gives you a better understanding of how the code you write is being used under the hood.

A Quick Introduction to Hy

Now that you have a good understanding of how Python AST works, you can start dreaming of creating a new syntax for Python. You could parse this new syntax, build an AST out of it, and compile it down to Python code.

This is exactly what Hy does. *Hy* is a Lisp dialect that parses a Lisp-like language and converts it to regular Python AST, making it fully compatible with the Python ecosystem. You could compare it to what Clojure is to Java. Hy could fill a book by itself, so we will only skim over it. Hy uses the syntax and some features of the Lisp family of languages: it's functionally oriented, provides macros, and is easily extensible.

If you're not already familiar with Lisp—and you should be—the Hy syntax will look familiar. Once you install Hy (by running `pip install hy`),

launching the `hy` interpreter will give you a standard REPL prompt from which you can start to interact with the interpreter, as shown in Listing 9-13.

```
% hy
hy 0.9.10
=> (+ 1 2)
3
```

Listing 9-13: Interacting with the Hy interpreter

For those not familiar with the Lisp syntax, parentheses are used to construct lists. If a list is unquoted, it is evaluated: the first element must be a function, and the rest of the items from the list are passed as arguments. Here the code `(+ 1 2)` is equivalent to `1 + 2` in Python.

In Hy, most constructs, such as function definitions, are mapped from Python directly.

```
=> (defn hello [name]
...  (print "Hello world!")
...  (print (% "Nice to meet you %s" name)))
=> (hello "jd")
Hello world!
Nice to meet you jd
```

Listing 9-14: Mapping a function definition from Python

As shown in Listing 9-14, internally Hy parses the code provided, converts it to a Python AST, compiles it, and evaluates it. Fortunately, Lisp is an easy tree to parse: each pair of parentheses represents a node of the tree, meaning the conversion is actually easier than for the native Python syntax!

Class definition is supported through the `defclass` construct, which is inspired by the Common Lisp Object System (CLOS).

```
(defclass A [object]
  [[x 42]
   [y (fn [self value]
        (+ self.x value))]])
```

Listing 9-15: Defining a class with `defclass`

Listing 9-15 defines a class named `A`, which inherits from `object`, with a class attribute `x` whose value is 42; then a method `y` returns the `x` attribute plus a value passed as argument.

What's really wonderful is that you can import *any Python library* directly into Hy and use it with no penalty. Use the `import()` function to import a module, as shown in Listing 9-16, just as you would with regular Python.

```
=> (import uuid)
=> (uuid.uuid4)
UUID('f823a749-a65a-4a62-b853-2687c69d0e1e')
=> (str (uuid.uuid4))
'4efa60f2-23a4-4fc1-8134-00f5c271f809'
```

Listing 9-16: Importing regular Python modules

Hy also has more advanced constructs and macros. In Listing 9-17, admire what the `cond()` function can do for you instead of the classic but verbose `if/elif/else`.

```
(cond
 [(> somevar 50)
  (print "That variable is too big!")]
 [(< somevar 10)
  (print "That variable is too small!")]
 [true
  (print "That variable is jusssst right!")])
```

Listing 9-17: Using `cond` instead of `if/elif/else`

The `cond` macro has the following signature: `(cond [`*`condition_expression return_expression`*`] ...)`. Each condition expression is evaluated, starting with the first: as soon as one of the condition expressions returns a true value, the return expression is evaluated and returned. If no return expression is provided, then the value of the condition expression is returned. Thus, `cond` is equivalent to an `if/elif` construct, except that it can return the value of the condition expression without having to evaluate it twice or store it in a temporary variable!

Hy allows you to jump into the Lisp world without leaving your comfort zone too far behind you, since you're still writing Python. The `hy2py` tool can even show you what your Hy code would look like once

translated into Python. While Hy is not widely used, it is a great tool to show the potential of the Python language. If you're interested in learning more, I suggest you check out the online documentation and join the community.

Summary

Just like any other programming language, Python source code can be represented using an abstract tree. You'll rarely use the AST directly, but when you understand how it works, it can provide a helpful perspective.

Paul Tagliamonte on the AST and Hy

Paul created Hy in 2013, and, as a Lisp lover, I joined him in this fabulous adventure. Paul is currently a developer at Sunlight Foundation.

How did you learn to use the AST correctly, and do you have any advice for people looking at it?

The AST is extremely underdocumented, so most knowledge comes from generated ASTs that have been reverse engineered. By writing up simple Python scripts, one can use something similar to `import ast; ast.dump(ast.parse("print foo"))` to generate an equivalent AST to help with the task. With a bit of guesswork, and some persistence, it's not untenable to build up a basic understanding this way.

At some point, I'll take on the task of documenting my understanding of the AST module, but I find writing code is the best way to learn the AST.

How does Python's AST differ between versions and uses?

Python's AST is not private, but it's not a public interface either. No stability is guaranteed from version to version—in fact, there are some rather annoying differences between Python 2 and 3 and even within different Python 3 releases. In addition, different implementations may

interpret the AST differently or even have a unique AST. Nothing says Jython, PyPy, or CPython must deal with the Python AST in the same way.

For instance, CPython can handle slightly out-of-order AST entries (by the `lineno` and `col_offset`), whereas PyPy will throw an assertion error. Though sometimes annoying, the AST is generally sane. It's not impossible to build an AST that works on a vast number of Python instances. With a conditional or two, it's only mildly annoying to create an AST that works on CPython 2.6 through 3.3 and PyPy, making this tool quite handy.

What was your process in creating Hy?

I started on Hy following a conversation about how useful it would be to have a Lisp that compiles to Python rather than Java's JVM (Clojure). A few short days later, and I had the first version of Hy. This version resembled a Lisp and even worked like a proper Lisp in some ways, but it was slow. I mean, really slow. It was about an order of magnitude slower than native Python, since the Lisp runtime itself was implemented in Python.

Frustrated, I almost gave up, but then a coworker suggested using the AST to implement the runtime, rather than implementing the runtime in Python. This suggestion was the catalyst for the entire project. I spent my entire holiday break in 2012 hacking on Hy. A week or so later, I had something that resembled the current Hy codebase.

Just after getting enough of Hy working to implement a basic Flask app, I gave a talk at Boston Python about the project, and the reception was incredibly warm—so warm, in fact, that I start to view Hy as a good way to teach people about Python internals, such as how the REPL works, PEP 302 import hooks, and the Python AST. This was a good introduction to the concept of code that writes code.

I rewrote chunks of the compiler to fix some philosophical issues in the process, leading us to the current iteration of the codebase—which has stood up quite well!

Learning Hy is also a good way to begin understanding how to read Lisp. Users can get comfortable with s-expressions in an environment

they know and even use libraries they're already using, easing the transition to other Lisps, such as Common Lisp, Scheme, or Clojure.

How interoperable with Python is Hy?

Hy is amazingly interoperable. So much so that `pdb` can properly debug Hy without you having to make any changes at all. I've written Flask apps, Django apps, and modules of all sorts with Hy. Python can import Python, Hy can import Hy, Hy can import Python, and Python can import Hy. This is what really makes Hy unique; other Lisp variants like Clojure are purely unidirectional. Clojure can import Java, but Java has one hell of a time importing Clojure.

Hy works by translating Hy code (in s-expressions) into the Python AST almost directly. This compilation step means the generated bytecode is fairly sane stuff, which means Python has a very hard time of even telling the module isn't written in Python at all.

Common Lisp-isms, such as `*earmuffs*` or `using-dashes` are fully supported by translating them into a Python equivalent (in this case, `*earmuffs*` becomes `EARMUFFS`, and `using-dashes` becomes `using_dashes`), which means Python doesn't have a hard time using them at all.

Ensuring that we have really good interoperability is one of our highest priorities, so if you see any bugs—file them!

What are the advantages and disadvantages of choosing Hy?

One advantage of Hy is that it has a full macro system, which Python struggles with. Macros are special functions that alter the code during the compile step. This makes it easy to create new domain-specific languages, which are composed of the base language (in this case, Hy/Python) along with many macros that allow uniquely expressive and succinct code.

As for downsides, Hy, by virtue of being a Lisp written in s-expressions, suffers from the stigma of being hard to learn, read, or maintain. People might be averse to working on projects using Hy for fear of its complexity.

Hy is the Lisp everyone loves to hate. Python folks may not enjoy its syntax, and Lispers may avoid it because Hy uses Python objects

directly, meaning the behavior of fundamental objects can sometimes be surprising to the seasoned Lisper.

Hopefully people will look past its syntax and consider exploring parts of Python previously untouched.

10
PERFORMANCES AND OPTIMIZATIONS

Optimizing is rarely the first thing you think about when developing, but there always comes a time when optimizing for better performance will be appropriate. That's not to say you should write a program with the idea that it will be slow, but thinking about optimization without first figuring out the right tools to use and doing the proper profiling is a waste of time. As Donald Knuth wrote, "Premature optimization is the root of all evil."[1]

Here, I'll show you how to use the right approach to write fast code and where to look when more optimization is needed. Many developers try to guess where Python might be slower or faster. Rather than speculating, this chapter will help you understand how to profile your application so you'll know what part of your program is slowing things down and where the bottlenecks are.

Data Structures

Most programming problems can be solved in an elegant and simple manner with the right data structures—and Python provides many data structures to choose from. Learning to leverage those existing data structures results in cleaner and more stable solutions than coding custom data structures.

For example, everybody uses `dict`, but how many times have you seen code trying to access a dictionary by catching the `KeyError` exception, as shown here:

```
def get_fruits(basket, fruit):
    try:
        return basket[fruit]
    except KeyError:
        return None
```

Or by checking whether the key is present first:

```
def get_fruits(basket, fruit):
    if fruit in basket:
        return basket[fruit]
```

If you use the `get()` method already provided by the `dict` class, you can avoid having to catch an exception or checking the key's presence in the first place:

```
def get_fruits(basket, fruit):
    return basket.get(fruit)
```

The method `dict.get()` can also return a default value instead of `None`; just call it with a second argument:

```
def get_fruits(basket, fruit):
    # Return the fruit, or Banana if the fruit cannot be found.
    return basket.get(fruit, Banana())
```

Many developers are guilty of using basic Python data structures without being aware of all the methods they provide. This is also true for sets; methods in set data structures can solve many problems that would otherwise need to be addressed by writing nested `for`/`if` blocks. For example, developers often use `for`/`if` loops to determine whether an item is in a list, like this:

```
def has_invalid_fields(fields):
    for field in fields:
        if field not in ['foo', 'bar']:
            return True
    return False
```

The loop iterates over each item in the list and checks that all items are either `foo` or `bar`. But you can write this more efficiently, removing the need for a loop:

```
def has_invalid_fields(fields):
    return bool(set(fields) - set(['foo', 'bar']))
```

This changes the code to convert the fields to a set, and it gets the rest of the set by subtracting the `set(['foo', 'bar'])`. It then converts the set to a Boolean value, which indicates whether any items that aren't `foo` and `bar` are left over. By using sets, there is no need to iterate over any list and to check items one by one. A single operation on two sets, done internally by Python, is faster.

Python also has more advanced data structures that can greatly reduce the burden of code maintenance. For example, take a look at Listing 10-1.

```
def add_animal_in_family(species, animal, family):
    if family not in species:
        species[family] = set()
    species[family].add(animal)

species = {}
add_animal_in_family(species, 'cat', 'felidea')
```

Listing 10-1: Adding an entry in a dictionary of sets

This code is perfectly valid, but how many times will your programs require a variation of Listing 10-1? Tens? Hundreds?

Python provides the `collections.defaultdict` structure, which solves the problem in an elegant way:

```
import collections

def add_animal_in_family(species, animal, family):
    species[family].add(animal)

species = collections.defaultdict(set)
add_animal_in_family(species, 'cat', 'felidea')
```

Each time you try to access a nonexistent item from your `dict`, the `defaultdict` will use the function that was passed as argument to its

constructor to build a new value, instead of raising a `KeyError`. In this case, the `set()` function is used to build a new `set` each time we need it.

The `collections` module offers a few more data structures that you can use to solve other kinds of problems. For example, imagine that you want to count the number of distinct items in an iterable. Let's take a look at the `collections.Counter()` method, which provides methods that solve this problem:

```
>>> import collections
>>> c = collections.Counter("Premature optimization is the root of all evil.")
>>> c
>>> c['P']  # Returns the name of occurrence of the letter 'P'
1
>>> c['e']  # Returns the name of occurrence of the letter 'e'
4
>>> c.most_common(2)  # Returns the 2 most common letters
[(' ', 7), ('i', 5)]
```

The `collections.Counter` object works with any iterable that has hashable items, removing the need to write your own counting functions. It can easily count the number of letters in a string and return the top *n* most common items of an iterable. You might have tried to implement something like this on your own if you were not aware it was already provided by Python's Standard Library.

With the right data structure, the correct methods, and—obviously—an adequate algorithm, your program should perform well. However, if it is not performing well enough, the best way to get clues about where it might be slow and need optimization is to profile your code.

Understanding Behavior Through Profiling

Profiling is a form of dynamic program analysis that allows us to understand how a program behaves. It allows us to determine where there might be bottlenecks and a need for optimization. A profile of a program takes the form of a set of statistics that describe how often parts of the program execute and for how long.

Python provides a few tools for profiling your program. One, `cProfile`, is part of the Python Standard Library and does not require

installation. We'll also look at the `dis` module, which can disassemble Python code into smaller parts, making it easier to understand what is happening under the hood.

cProfile

Python has included `cProfile` by default since Python 2.5. To use `cProfile`, call it with your program using the syntax `python -m cProfile <program>`. This should load and enable the `cProfile` module, then run the regular program with instrumentation enabled, as shown in Listing 10-2.

```
$ python -m cProfile myscript.py
         343 function calls (342 primitive calls) in 0.000 seconds

   Ordered by: standard name
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(_getframe)
        1    0.000    0.000    0.000    0.000 :0(len)
      104    0.000    0.000    0.000    0.000 :0(setattr)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 :0(startswith)
      2/1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 StringIO.py:30(<module>)
        1    0.000    0.000    0.000    0.000 StringIO.py:42(StringIO)
```

Listing 10-2: Default output of `cProfile` used against a Python script

Listing 10-2 shows the output of running a simple script with `cProfile`. This tells you the number of times each function in the program was called and the time spent on its execution. You can also use the `-s` option to sort by other fields; for example, `-s time` would sort the results by internal time.

We can visualize the information generated by `cProfile` using a great tool called KCacheGrind. This tool was created to deal with programs written in C, but luckily we can use it with Python data by converting the data to a call tree.

The `cProfile` module has an `-o` option that allows you to save the profiling data, and `pyprof2calltree` can convert data from one format to the other. First, install the converter with the following:

```
$ pip install pyprof2calltree
```

Then run the converter as shown in Listing 10-3 to both convert the data (`-i` option) and run KCacheGrind with the converted data (`-k` option).

```
$ python -m cProfile -o myscript.cprof myscript.py
$ pyprof2calltree -k -i myscript.cprof
```

Listing 10-3: Running `cProfile` and launching KCacheGrind

Once KCacheGrind opens, it will display information that looks like that in Figure 10-1. With these visual results, you can use the call graph to follow the percentage of time spent in each function, allowing you to determine what part of your program might be consuming too many resources.

The easiest way to read KCacheGrind is to start with the table on the left of the screen, which lists all the functions and methods executed by your program. You can sort these by execution time, then identify the one that consumes the most CPU time and click on it.

The right panels of KCacheGrind can show you which functions have called that function and how many times, as well as which other functions are being called by the function. The call graph of your program, including the execution time of each part, is easy to navigate.

This should allow you to better understand which parts of your code might need optimization. The way to optimize the code is up to you and depends on what your program is trying to achieve!

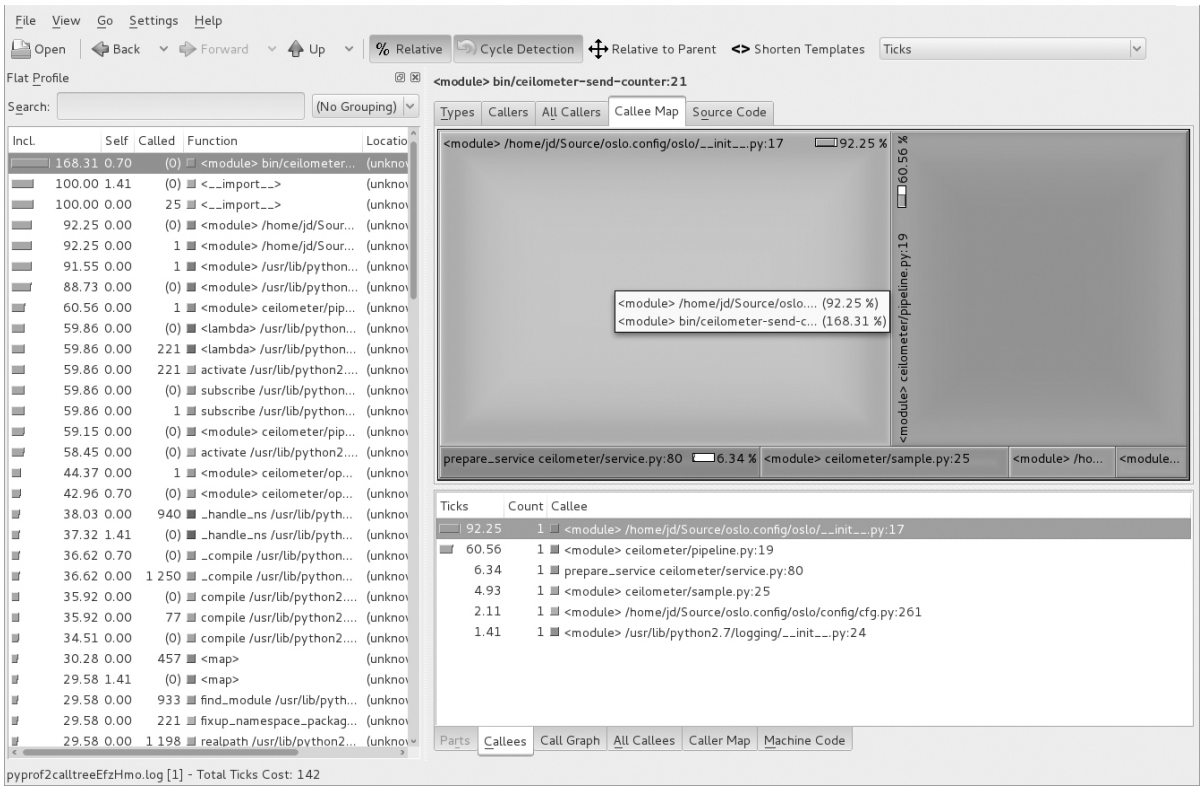


Figure 10-1: Example of KCacheGrind output

While retrieving information about how your program runs and visualizing it works well to get a macroscopic view of your program, you might need a more microscopic view of some parts of the code to inspect its elements more closely. In such a case, I find it better to rely on the `dis` module to find out what's going on behind the scenes.

Disassembling with the dis Module

The `dis` module is a disassembler of Python bytecode. Taking code apart can be useful to understand what's going on behind each line so you can properly optimize it. For example, Listing 10-4 shows the `dis.dis()` function, which disassembles whichever function you pass as a parameter and prints the list of bytecode instructions that are run by the function.

```
>>> def x():
...     return 42
...
>>> import dis
```

```
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (42)
              3 RETURN_VALUE
```

Listing 10-4: Disassembling a function

In Listing 10-4, the function `x` is disassembled and its constituents, made of bytecode instructions, are printed. There are only two operations here: loading a constant (`LOAD_CONST`), which is `42`, and returning that value (`RETURN_VALUE`).

To see `dis` in action and how it can be useful, we'll define two functions that do the same thing—concatenate three letters—and disassemble them to see how they do their tasks in different ways:

```
abc = ('a', 'b', 'c')

def concat_a_1():
    for letter in abc:
            abc[0] + letter

def concat_a_2():
    a = abc[0]
    for letter in abc:
            a + letter
```

Both functions appear to do the same thing, but if we disassemble them using `dis.dis`, as shown in Listing 10-5, we'll see that the generated bytecode is a bit different.

```
>>> dis.dis(concat_a_1)
  2           0 SETUP_LOOP              26 (to 29)
              3 LOAD_GLOBAL              0 (abc)
              6 GET_ITER
        >>    7 FOR_ITER                18 (to 28)
             10 STORE_FAST               0 (letter)

  3          13 LOAD_GLOBAL              0 (abc)
             16 LOAD_CONST               1 (0)
             19 BINARY_SUBSCR
             20 LOAD_FAST                0 (letter)
             23 BINARY_ADD
             24 POP_TOP
             25 JUMP_ABSOLUTE            7
        >>   28 POP_BLOCK
        >>   29 LOAD_CONST               0 (None)
             32 RETURN_VALUE
>>> dis.dis(concat_a_2)
  2           0 LOAD_GLOBAL              0 (abc)
              3 LOAD_CONST               1 (0)
```

```
              6 BINARY_SUBSCR
              7 STORE_FAST               0 (a)

  3           10 SETUP_LOOP              22 (to 35)
              13 LOAD_GLOBAL              0 (abc)
              16 GET_ITER
        >>    17 FOR_ITER                14 (to 34)
              20 STORE_FAST               1 (letter)

  4           23 LOAD_FAST                0 (a)
              26 LOAD_FAST                1 (letter)
              29 BINARY_ADD
              30 POP_TOP
              31 JUMP_ABSOLUTE           17
        >>    34 POP_BLOCK
        >>    35 LOAD_CONST               0 (None)
              38 RETURN_VALUE
```

Listing 10-5: Disassembling functions that concatenate strings

In the second function in Listing 10-5, we store `abc[0]` in a temporary variable before running the loop. This makes the bytecode that's executed inside the loop a little smaller than the bytecode for the first function, as we avoid having to do the `abc[0]` lookup for each iteration. Measured using `timeit`, the second version is 10 percent faster than the first function; it takes a whole microsecond less to execute! Obviously this microsecond is not worth optimizing for unless you call this function billions of times, but this is the kind of insight that the `dis` module can provide.

Whether you rely on "tricks" such as storing the value outside the loop depends on the situation—ultimately, it should be the compiler's work to optimize this kind of thing. On the other hand, it's difficult for the compiler to be sure that optimization wouldn't have negative side effects because Python is heavily dynamic. In Listing 10-5, using `abc[0]` will call `abc.__getitem__`, which could have side effects if it has been overridden by inheritance. Depending on the version of the function you use, the `abc.__getitem__` method will be called once or several times, which might make a difference. Therefore, be careful when writing and optimizing your code!

Defining Functions Efficiently

One common mistake I have found when reviewing code is definitions of functions within functions. This is inefficient because the function is then redefined repeatedly and needlessly. For example, Listing 10-6 shows the `y()` function being defined multiple times.

```
>> import dis
>>> def x():
...     return 42
...
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (42)
              3 RETURN_VALUE
>>> def x():
...     def y():
...             return 42
...     return y()
...
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (<code object y at
x100ce7e30, file "<stdin>", line 2>)
              3 MAKE_FUNCTION            0
              6 STORE_FAST               0 (y)
  4           9 LOAD_FAST                0 (y)
             12 CALL_FUNCTION            0
             15 RETURN_VALUE
```

Listing 10-6: Function redefinition

Listing 10-6 shows the calling of `MAKE_FUNCTION`, `STORE_FAST`, `LOAD_FAST`, and `CALL_FUNCTION`, which requires many more opcodes than those needed to return `42`, as seen in Listing 10-4.

The only case in which you'd need to define a function within a function is when building a function closure, and this is a perfectly identified use case in Python's opcodes with `LOAD_CLOSURE`, as shown in Listing 10-7.

```
>>> def x():
...     a = 42
...     def y():
...             return a
...     return y()
...
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (42)
              3 STORE_DEREF              0 (a)

  3           6 LOAD_CLOSURE             0 (a)
              9 BUILD_TUPLE              1
             12 LOAD_CONST               2 (<code object y at
```

```
x100d139b0, file "<stdin>", line 3>)
             15 MAKE_CLOSURE             0
             18 STORE_FAST               0 (y)

  5          21 LOAD_FAST                0 (y)
             24 CALL_FUNCTION            0
             27 RETURN_VALUE
```

Listing 10-7: Defining a closure

While you probably won't need to use it every day, disassembling code is a handy tool for when you want a closer look at what happens under the hood.

Ordered Lists and bisect

Next, let's look at optimizing lists. If a list is unsorted, the worst-case scenario for finding a particular item's position in the list has a complexity of *O(n)*, meaning that in the worst case, you'll find your item after iterating over every item of the list.

The usual solution for optimizing this problem is to use a *sorted* list instead. Sorted lists use a bisecting algorithm for lookup to achieve a retrieve time of *O(log n)*. The idea is to recursively split the list in half and look on which side, left or right, the item must appear in and so which side should be searched next.

Python provides the `bisect` module, which contains a bisection algorithm, as shown in Listing 10-8.

```
>>> farm = sorted(['haystack', 'needle', 'cow', 'pig'])
>>> bisect.bisect(farm, 'needle')
3
>>> bisect.bisect_left(farm, 'needle')
2
>>> bisect.bisect(farm, 'chicken')
0
>>> bisect.bisect_left(farm, 'chicken')
0
>>> bisect.bisect(farm, 'eggs')
1
>>> bisect.bisect_left(farm, 'eggs')
1
```

Listing 10-8: Using `bisect` to find a needle in a haystack

As shown in Listing 10-8, the `bisect.bisect()` function returns the position where an element should be inserted to keep the list sorted. Obviously, this only works if the list is properly sorted to begin with. Initial sorting allows to us get the *theoretical* index of an item: `bisect()` does not return whether the item is in the list but where the item should be if it is in the list. Retrieving the item at this index will answer the question about whether the item is in the list.

If you wish to insert the element into the correct sorted position immediately, the `bisect` module provides the `insort_left()` and `insort_right()` functions, as shown in Listing 10-9.

```
>>> farm
['cow', 'haystack', 'needle', 'pig']
>>> bisect.insort(farm, 'eggs')
>>> farm
['cow', 'eggs', 'haystack', 'needle', 'pig']
>>> bisect.insort(farm, 'turkey')
>>> farm
['cow', 'eggs', 'haystack', 'needle', 'pig', 'turkey']
```

Listing 10-9: Inserting an item in a sorted list

Using the `bisect` module, you could also create a special `SortedList` class inheriting from `list` to create a list that is always sorted, as shown in Listing 10-10:

```
import bisect
import unittest

class SortedList(list):
    def __init__(self, iterable):
        super(SortedList, self).__init__(sorted(iterable))

    def insort(self, item):
        bisect.insort(self, item)
    def extend(self, other):
        for item in other:
            self.insort(item)

    @staticmethod
    def append(o):
        raise RuntimeError("Cannot append to a sorted list")

    def index(self, value, start=None, stop=None):
        place = bisect.bisect_left(self[start:stop], value)
        if start:
            place += start
        end = stop or len(self)
```

```
        if place < end and self[place] == value:
            return place
        raise ValueError("%s is not in list" % value)

class TestSortedList(unittest.TestCase):
    def setUp(self):
        self.mylist = SortedList(
            ['a', 'c', 'd', 'x', 'f', 'g', 'w']
        )

    def test_sorted_init(self):
        self.assertEqual(sorted(['a', 'c', 'd', 'x', 'f', 'g', 'w']),
                         self.mylist)

    def test_sorted_insort(self):
        self.mylist.insort('z')
        self.assertEqual(['a', 'c', 'd', 'f', 'g', 'w', 'x', 'z'],
                         self.mylist)
        self.mylist.insort('b')
        self.assertEqual(['a', 'b', 'c', 'd', 'f', 'g', 'w', 'x', 'z'],
                         self.mylist)

    def test_index(self):
        self.assertEqual(0, self.mylist.index('a'))
        self.assertEqual(1, self.mylist.index('c'))
        self.assertEqual(5, self.mylist.index('w'))
        self.assertEqual(0, self.mylist.index('a', stop=0))
        self.assertEqual(0, self.mylist.index('a', stop=2))
        self.assertEqual(0, self.mylist.index('a', stop=20))
        self.assertRaises(ValueError, self.mylist.index, 'w', stop=3)
        self.assertRaises(ValueError, self.mylist.index, 'a', start=3)
        self.assertRaises(ValueError, self.mylist.index, 'a', start=333)

    def test_extend(self):
        self.mylist.extend(['b', 'h', 'j', 'c'])
        self.assertEqual(
            ['a', 'b', 'c', 'c', 'd', 'f', 'g', 'h', 'j', 'w', 'x']
            self.mylist)
```

Listing 10-10: A `SortedList` object implementation

Using a `list` class like this is slightly slower when it comes to inserting the item, because the program has to look for the right spot to insert it. However, this class is faster at using the `index()` method than its parent. Obviously, one shouldn't use the `list.append()` method on this class: you can't append an item at the end of the list or it could end up unsorted!

Many Python libraries implement various versions of Listing 10-10 for many more data types, such as binary or red-black tree structures. The `blist` and `bintree` Python packages contain code that can be used for

these purposes and are a handy alternative to implementing and debugging your own version.

In the next section, we'll see how the native tuple data type provided by Python can be leveraged to make your Python code a little faster.

namedtuple and Slots

Often in programming, you'll need to create simple objects that possess only a few fixed attributes. A simple implementation might be something along these lines:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

This definitely gets the job done. However, there is a downside to this approach. Here we're creating a class that inherits from the object class, so by using this `Point` class, you are instantiating full objects and allocating a lot of memory.

In Python, regular objects store all of their attributes inside a dictionary, and this dictionary is itself stored in the `__dict__` attribute, as shown in Listing 10-11.

```
>>> p = Point(1, 2)
>>> p.__dict__
{'y': 2, 'x': 1}
>>> p.z = 42
>>> p.z
42
>>> p.__dict__
{'y': 2, 'x': 1, 'z': 42}
```

Listing 10-11: How attributes are stored internally in a Python object

For Python, the advantage of using a `dict` is that it allows you to add as many attributes as you want to an object. The drawback is that using a dictionary to store these attributes is expensive in terms of memory—you need to store the object, the keys, the value references, and

everything else. That makes it slow to create and slow to manipulate, with a high memory cost.

As an example of this unnecessary memory usage, consider the following simple class:

```
class Foobar(object):
    def __init__(self, x):
        self.x = x
```

This creates a simple `Point` object with a single attribute named `x`. Let's check the memory usage of this class using the `memory_profiler`, a nice Python package that allows us to see the memory usage of a program line by line, and a small script that creates 100,000 objects, as shown in Listing 10-12.

```
$ python -m memory_profiler object.py
Filename: object.py

Line #    Mem usage    Increment   Line Contents
     5                             @profile
     6     9.879 MB     0.000 MB   def main():
     7    50.289 MB    40.410 MB       f = [ Foobar(42) for i in range(100000) ]
```

Listing 10-12: Using `memory_profiler` on a script using objects

Listing 10-12 demonstrates that creating 100,000 of the objects of the `Foobar` class would consume 40MB of memory. Although 400 bytes per object might not sound that big, when you are creating thousands of objects, the memory adds up.

There is a way to use objects while avoiding this default behavior of `dict`: classes in Python can define a `__slots__` attribute that will list only the attributes allowed for instances of this class. Instead of allocating a whole dictionary object to store the object attributes, you can use a *list* object to store them.

If you go through CPython source code and take a look at the *Objects/typeobject.c* file, it is quite easy to understand what Python does when `__slots__` is set on a class. Listing 10-13 is an abbreviated version of the function that handles this:

```
static PyObject *
type_new(PyTypeObject *metatype, PyObject *args, PyObject *kwds)
```

```
{
    --snip--
    /* Check for a __slots__ sequence variable in dict, and count it */
    slots = _PyDict_GetItemId(dict, &PyId___slots__);
    nslots = 0;
    if (slots == NULL) {
        if (may_add_dict)
            add_dict++;
        if (may_add_weak)
            add_weak++;
    }
    else {
        /* Have slots */
        /* Make it into a tuple */
        if (PyUnicode_Check(slots))
            slots = PyTuple_Pack(1, slots);
        else
            slots = PySequence_Tuple(slots);
        /* Are slots allowed? */
        nslots = PyTuple_GET_SIZE(slots);
        if (nslots > 0 && base->tp_itemsize != 0) {
            PyErr_Format(PyExc_TypeError,
                         "nonempty __slots__ "
                         "not supported for subtype of '%s'",
                         base->tp_name);
            goto error;
        }
        /* Copy slots into a list, mangle names and sort them.
           Sorted names are needed for __class__ assignment.
           Convert them back to tuple at the end.
        */
        newslots = PyList_New(nslots - add_dict - add_weak);
        if (newslots == NULL)
            goto error;
        if (PyList_Sort(newslots) == -1) {
            Py_DECREF(newslots);
            goto error;
        }
        slots = PyList_AsTuple(newslots);
        Py_DECREF(newslots);
        if (slots == NULL)
            goto error;
    }
    /* Allocate the type object */
    type = (PyTypeObject *)metatype->tp_alloc(metatype, nslots);
    --snip--
    /* Keep name and slots alive in the extended type object */
    et = (PyHeapTypeObject *)type;
    Py_INCREF(name);
    et->ht_name = name;
    et->ht_slots = slots;
    slots = NULL;
    --snip--
    return (PyObject *)type;
```

Listing 10-13: An extract from Objects/typeobject.c

As you can see in Listing 10-13, Python converts the content of `__slots__` into a tuple and then into a list, which it builds and sorts before converting the list back into a tuple to use and store in the class. In this way, Python can retrieve the values quickly, without having to allocate and use an entire dictionary.

It's easy enough to declare and use such a class. All you need to do is to set the `__slots__` attribute to a list of the attributes that will be defined in the class:

```
class Foobar(object):
    __slots__ = ('x',)

    def __init__(self, x):
        self.x = x
```

We can compare the memory usage of the two approaches using the `memory_profiler` Python package, as shown in Listing 10-14.

```
% python -m memory_profiler slots.py
Filename: slots.py

Line #    Mem usage    Increment   Line Contents
     7                             @profile
     8     9.879 MB     0.000 MB   def main():
     9    21.609 MB    11.730 MB       f = [ Foobar(42) for i in range(100000) ]
```

Listing 10-14: Running `memory_profiler` on the script using `__slots__`

Listing 10-14 shows that this time, less than 12MB of memory was needed to create 100,000 objects—or fewer than 120 bytes per object. Thus, by using the `__slots__` attribute of Python classes, we can reduce memory usage, so when we are creating a large number of simple objects, the `__slots__` attribute is an effective and efficient choice. However, this technique shouldn't be used for performing static typing by hardcoding the list of attributes of every class: doing so wouldn't be in the spirit of Python programs.

The drawback here is that the list of attributes is now fixed. No new attribute can be added to the `Foobar` class at runtime. Due to the fixed nature of the attribute list, it's easy enough to imagine classes where the attributes listed would always have a value and where the fields would always be sorted in some way.

This is exactly what occurs in the `namedtuple` class from the `collection` module. This `namedtuple` class allows us to dynamically create a class that will inherit from the tuple class, thus sharing characteristics such as being immutable and having a fixed number of entries.

Rather than having to reference them by index, `namedtuple` provides the ability to retrieve tuple elements by referencing a named attribute. This makes the tuple easier to access for humans, as shown in Listing 10-15.

```
>>> import collections
>>> Foobar = collections.namedtuple('Foobar', ['x'])
>>> Foobar = collections.namedtuple('Foobar', ['x', 'y'])
>>> Foobar(42, 43)
Foobar(x=42, y=43)
>>> Foobar(42, 43).x
42
>>> Foobar(42, 43).x = 44
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> Foobar(42, 43).z = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Foobar' object has no attribute 'z'
>>> list(Foobar(42, 43))
[42, 43]
```

Listing 10-15: Using `namedtuple` to reference tuple elements

Listing 10-15 shows how you can create a simple class with just one line of code and then instantiate it. We can't change any attributes of objects of this class or add attributes to them, both because the class inherits from `namedtuple` and because the `__slots__` value is set to an empty tuple, avoiding the creation of the `__dict__`. Since a class like this would inherit from tuple, we can easily convert it to a list.

Listing 10-16 demonstrates the memory usage of the `namedtuple` class factory.

```
% python -m memory_profiler namedtuple.py
Filename: namedtuple.py

Line #    Mem usage    Increment   Line Contents
     4                             @profile
     5     9.895 MB     0.000 MB   def main():
     6    23.184 MB    13.289 MB       f = [ Foobar(42) for i in range(100000) ]
```

Listing 10-16: Using `namedtuple` to run `memory_profiler` on a script

At around 13MB for 100,000 objects, using `namedtuple` is slightly less efficient than using an object with `__slots__`, but the bonus is that it is compatible with the tuple class. It can therefore be passed to many native Python functions and libraries that expect an iterable as an argument. A `namedtuple` class factory also enjoys the various optimizations that exist for tuples: for example, tuples with fewer items than `PyTuple_MAXSAVESIZE` (20 by default) will use a faster memory allocator in CPython.

The `namedtuple` class also provides a few extra methods that, even if prefixed by an underscore, are actually intended to be public. The `_asdict()` method can convert the `namedtuple` to a `dict` instance, the `_make()` method allows you to convert an existing iterable object to this class, and `_replace()` returns a new instance of the object with some fields replaced.

Named tuples are a great replacement for small objects that consists of only a few attributes and do not require any custom methods—consider using them rather than dictionaries, for example. If your data type needs methods, has a fixed list of attributes, and might be instantiated thousands of times, then creating a custom class using `__slots__` might be a good idea to save some memory.

Memoization

Memoization is an optimization technique used to speed up function calls by caching their results. The results of a function can be cached only if the function is *pure*, meaning that it has no side effects and does not depend on any global state. (See Chapter 8 for more on pure functions.)

One trivial function that can be memoized is `sin()`, shown in Listing 10-17.

```
>>> import math
>>> _SIN_MEMOIZED_VALUES = {}
>>> def memoized_sin(x):
...     if x not in _SIN_MEMOIZED_VALUES:
```

```
...         _SIN_MEMOIZED_VALUES[x] = math.sin(x)
...     return _SIN_MEMOIZED_VALUES[x]
>>> memoized_sin(1)
0.8414709848078965
>>> _SIN_MEMOIZED_VALUES
{1: 0.8414709848078965}
>>> memoized_sin(2)
0.9092974268256817
>>> memoized_sin(2)
0.9092974268256817
>>> _SIN_MEMOIZED_VALUES
{1: 0.8414709848078965, 2: 0.9092974268256817}
>>> memoized_sin(1)
0.8414709848078965
>>> _SIN_MEMOIZED_VALUES
{1: 0.8414709848078965, 2: 0.9092974268256817}
```

Listing 10-17: A memoized `sin()` function

In Listing 10-17, the first time that `memoized_sin()` is called with an argument that is not stored in `_SIN_MEMOIZED_VALUES`, the value is computed and stored in this dictionary. If we call the function with the same value again, the result will be retrieved from the dictionary rather than recomputed. While `sin()` computes very quickly, some advanced functions involving more complicated computations may take longer, and this is where memoization really shines.

If you've already read about decorators (if not, see "Decorators and When to Use Them" on page 100), you might see a perfect opportunity to use them here, and you'd be right. PyPI lists a few implementations of memoization through decorators, from very simple cases to the most complex and complete.

Starting with Python 3.3, the `functools` module provides a *least recently used (LRU)* cache decorator. This provides the same functionality as memoization, but with the benefit that it limits the number of entries in the cache, removing the least recently used one when the cache reaches its maximum size. The module also provides statistics on cache hits and misses (whether something was in the accessed cache or not), among other data. In my opinion, these statistics are must-haves when implementing such a cache. The strength of using memoization, or any caching technique, is in the ability to meter its usage and usefulness.

Listing 10-18 demonstrates how to use the `functools.lru_cache()` method to implement the memoization of a function. When decorated,

the function gets a `cache_info()` method that can be called to get statistics about the cache usage.

```
>>> import functools
>>> import math
>>> @functools.lru_cache(maxsize=2)
... def memoized_sin(x):
...     return math.sin(x)
...
>>> memoized_sin(2)
0.9092974268256817
>>> memoized_sin.cache_info()
CacheInfo(hits=0, misses=1, maxsize=2, currsize=1)
>>> memoized_sin(2)
0.9092974268256817
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=1, maxsize=2, currsize=1)
>>> memoized_sin(3)
0.1411200080598672
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=2, maxsize=2, currsize=2)
>>> memoized_sin(4)
-0.7568024953079282
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=3, maxsize=2, currsize=2)
>>> memoized_sin(3)
0.1411200080598672
>>> memoized_sin.cache_info()
CacheInfo(hits=2, misses=3, maxsize=2, currsize=2)
>>> memoized_sin.cache_clear()
>>> memoized_sin.cache_info()
CacheInfo(hits=0, misses=0, maxsize=2, currsize=0)
```

Listing 10-18: Inspecting cache statistics

Listing 10-18 demonstrates how your cache is being used and how to tell whether there are optimizations to be made. For example, if the number of misses is high when the cache is not full, then the cache may be useless because the arguments passed to the function are never identical. This will help determine what should or should not be memoized!

Faster Python with PyPy

PyPy is an efficient implementation of the Python language that complies with standards: you should be able to run any Python program with it. Indeed, the canonical implementation of Python, CPython—so

called because it's written in C—can be very slow. The idea behind PyPy was to write a Python interpreter in Python itself. In time, it evolved to be written in RPython, which is a restricted subset of the Python language.

RPython places constraints on the Python language such that a variable's type can be inferred at compile time. The RPython code is translated into C code, which is compiled to build the interpreter. RPython could of course be used to implement languages other than Python.

What's interesting in PyPy, besides the technical challenge, is that it is now at a stage where it can act as a faster replacement for CPython. PyPy has a *just-in-time (JIT)* compiler built-in; in other words, it allows the code to run faster by combining the speed of compiled code with the flexibility of interpretation.

How fast? That depends, but for pure algorithmic code, it is much faster. For more general code, PyPy claims to achieve three times the speed of CPython most of the time. Unfortunately, PyPy also has some of the limitations of CPython, including the *global interpreter lock (GIL)*, which allows only one thread to execute at a time.

Though it's not strictly an optimization technique, targeting PyPy as one of your supported Python implementations might be a good idea. To make PyPy a support implementation, you need to make sure that you are testing your software under PyPy as you would under CPython. In Chapter 6, we discussed `tox` (see "Using `virtualenv` with `tox`" on page 92), which supports the building of virtual environments using PyPy, just as it does for any version of CPython, so putting PyPy support in place should be pretty straightforward.

Testing PyPy support right at the beginning of the project will ensure that there's not too much work to do at a later stage if you decide that you want to be able to run your software with PyPy.

NOTE

For the Hy project discussed in Chapter 9, we successfully adopted this strategy from the beginning. Hy always has supported PyPy and all other

CPython versions without much trouble. On the other hand, OpenStack failed to do so for its projects and, as a result, is now blocked by various code paths and dependencies that don't work on PyPy for various reasons; they weren't required to be fully tested in the early stages.

PyPy is compatible with Python 2.7 and Python 3.5, and its JIT compiler works on 32- and 64-bit, x86, and ARM architectures and under various operating systems (Linux, Windows, and Mac OS X). PyPy often lags behind CPython in features, but it regularly catches up. Unless your project is reliant on the latest CPython features, this lag might not be a problem.

Achieving Zero Copy with the Buffer Protocol

Often programs have to deal with huge amounts of data in the form of large arrays of bytes. Handling such a large quantity of input in strings can be very ineffective once you start manipulating the data by copying, slicing, and modifying it.

Let's consider a small program that reads a large file of binary data and copies it partially into another file. To examine the memory usage of this program, we will use `memory_profiler`, as we did earlier. The script to partially copy the file is shown in Listing 10-19.

```
@profile
def read_random():
    with open("/dev/urandom", "rb") as source:
        content = source.read(1024 * 10000)
        content_to_write = content[1024:]
    print("Content length: %d, content to write length %d" %
          (len(content), len(content_to_write)))
    with open("/dev/null", "wb") as target:
        target.write(content_to_write)

if __name__ == '__main__':
    read_random()
```

Listing 10-19: Partially copying a file

Running the program in Listing 10-19 using `memory_profiler` produces the output shown in Listing 10-20.

```
$ python -m memory_profiler memoryview/copy.py
Content length: 10240000, content to write length 10238976
Filename: memoryview/copy.py

Mem usage    Increment   Line Contents
                         @profile
 9.883 MB     0.000 MB   def read_random():
 9.887 MB     0.004 MB       with open("/dev/urandom", "rb") as source:
19.656 MB     9.770 MB           content = source.read(1024 * 10000)❶
29.422 MB     9.766 MB           content_to_write = content[1024:]❷
29.422 MB     0.000 MB       print("Content length: %d, content to write length
%d" %
29.434 MB     0.012 MB             (len(content), len(content_to_write)))
29.434 MB     0.000 MB       with open("/dev/null", "wb") as target:
29.434 MB     0.000 MB           target.write(content_to_write)
```

Listing 10-20: Memory profiling of partial file copy

According to the output, the program reads 10MB from *_/dev/urandom* ❶. Python needs to allocate around 10MB of memory to store this data as a string. It then copies the entire block of data, minus the first KB ❷.

What's interesting in Listing 10-20 is that the program's memory usage is increased by about 10MB when building the variable `content_to_write`. In fact, the `slice` operator is copying the entirety of content, minus the first KB, into a new string object, allocating a large chunk of the 10MB.

Performing this kind of operation on large byte arrays is going to be a disaster since large pieces of memory will be allocated and copied. If you have experience writing in C code, you know that using the `memcpy()` function has a significant cost in terms of both memory usage and general performance.

But as a C programmer, you'll also know that strings are arrays of characters and that nothing stops you from looking at only *part* of an array without copying it. You can do this through the use of basic pointer arithmetic, assuming that the entire string is in a contiguous memory area.

This is also possible in Python using objects that implement the *buffer protocol*. The buffer protocol is defined in PEP 3118, as a C API

that needs to be implemented on various types for them to provide this protocol. The `string` class, for example, implements this protocol.

When you implement this protocol on an object, you can then use the `memoryview` class constructor to build a new `memoryview` object that will reference the original object memory. For example, Listing 10-21 shows how to use `memoryview` to access slice of a string without doing any copying:

```
   >>> s = b"abcdefgh"
   >>> view = memoryview(s)
   >>> view[1]
❶ 98 <1>
   >>> limited = view[1:3]
   >>> limited
   <memory at 0x7fca18b8d460>
   >>> bytes(view[1:3])
   b'bc'
```

Listing 10-21: Using `memoryview` to avoid copying data

At ❶, you find the ASCII code for the letter *b*. In Listing 10-21, we are making use of the fact that the `memoryview` object's `slice` operator itself returns a `memoryview` object. That means it does *not* copy any data but merely references a particular slice of it, saving the memory that would be used by a copy. Figure 10-2 illustrates what happens in Listing 10-21.

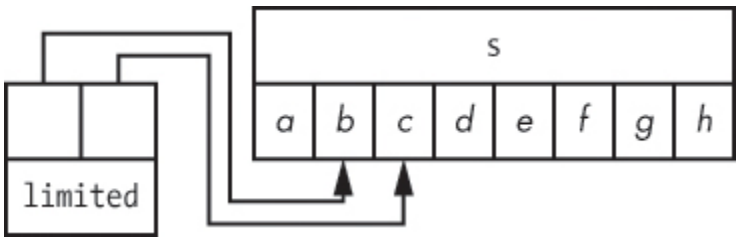


Figure 10-2: Using `slice` on `memoryview` objects

We can rewrite the program from Listing 10-19, this time referencing the data we want to write using a `memoryview` object rather than allocating a new string.

```
@profile
def read_random():
    with open("/dev/urandom", "rb") as source:
```

```
        content = source.read(1024 * 10000)
        content_to_write = memoryview(content)[1024:]
    print("Content length: %d, content to write length %d" %
          (len(content), len(content_to_write)))
    with open("/dev/null", "wb") as target:
        target.write(content_to_write)

if __name__ == '__main__':
    read_random()
```

Listing 10-22: Partially copying a file using `memoryview`

The program in Listing 10-22 uses half the memory of the first version in Listing 10-19. We can see this by testing it with `memory_profiler` again, like so:

```
$ python -m memory_profiler memoryview/copy-memoryview.py
Content length: 10240000, content to write length 10238976
Filename: memoryview/copy-memoryview.py

Mem usage    Increment   Line Contents
                         @profile
 9.887 MB     0.000 MB   def read_random():
 9.891 MB     0.004 MB ❶     with open("/dev/urandom", "rb") as source:
19.660 MB     9.770 MB ❷         content = source.read(1024 * 10000)
19.660 MB     0.000 MB           content_to_write = memoryview(content)[1024:]
19.660 MB     0.000 MB       print("Content length: %d, content to write length
%d" %
19.672 MB     0.012 MB             (len(content), len(content_to_write)))
19.672 MB     0.000 MB       with open("/dev/null", "wb") as target:
19.672 MB     0.000 MB           target.write(content_to_write)
```

These results show that we are reading 10,000KB from */dev/urandom* and not doing much with it ❶. Python needs to allocate 9.77MB of memory to store this data as a string ❷.

We reference the entire block of data minus the first KB, because we won't be writing that first KB to the target file. Because we aren't copying, no more memory is used!

This kind of trick is especially useful when dealing with sockets. When sending data over a socket, it's possible that the data might split between calls rather than be sent in a single call: the `socket.send` methods return the actual data length that was able to be sent by the network, which might be smaller than the data that was intended to be sent. Listing 10-23 shows how the situation is usually handled.

```
   import socket
   s = socket.socket(...)
   s.connect(...)
❶ data = b"a" * (1024 * 100000) <1>
   while data:
       sent = s.send(data)
    ❷ data = data[sent:] <2>
```

Listing 10-23: Sending data over a socket

First, we build a `bytes` object that contains the letter *a* more than 100 million times ❶. Then we remove the first `sent` bytes ❷.

Using a mechanism that implemented in Listing 10-23, a program will copy the data over and over until the socket has sent everything.

We can alter the program in Listing 10-23 to use `memoryview` to achieve the same functionality with zero copying, and therefore higher performance, as shown in Listing 10-24.

```
   import socket
   s = socket.socket(...)
   s.connect(...)
❶ data = b"a" * (1024 * 100000) <1>
   mv = memoryview(data)
   while mv:
       sent = s.send(mv)
    ❷ mv = mv[sent:] <2>
```

Listing 10-24: Sending data over a socket using `memoryview`

First, we build a `bytes` object that contains the letter *a* more than 100 million times ❶. Then, we build a new `memoryview` object pointing to the data that remains to be sent, rather than copying that data ❷. This program won't copy anything, so it won't use any more memory than the 100MB initially needed for the `data` variable.

We've seen how `memoryview` objects can be used to write data efficiently, and this same method can be used to *read* data. Most I/O operations in Python know how to deal with objects implementing the buffer protocol: they can read from those, and also write to those. In this case, we don't need `memoryview` objects; we can just ask an I/O function to write into our preallocated object, as shown in Listing 10-25.

```
>>> ba = bytearray(8)
>>> ba
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00')
>>> with open("/dev/urandom", "rb") as source:
...     source.readinto(ba)
...
8
>>> ba
bytearray(b'`m.z\x8d\x0fp\xa1')
```

Listing 10-25: Writing into a preallocated `bytearray`

In Listing 10-25, by using the `readinto()` method of the opened file, Python can directly read the data from the file and write it to a preallocated `bytearray`. With such techniques, it's easy to preallocate a buffer (as you would do in C to mitigate the number of calls to `malloc()`) and fill it at your convenience. Using `memoryview`, you can place data at any point in the memory area, as shown in Listing 10-26.

```
   >>> ba = bytearray(8)
❶ >>> ba_at_4 = memoryview(ba)[4:]
   >>> with open("/dev/urandom", "rb") as source:
❷ ...     source.readinto(ba_at_4)
   ...
   4
   >>> ba
   bytearray(b'\x00\x00\x00\x00\x0b\x19\xae\xb2')
```

Listing 10-26: Writing into an arbitrary position of `bytearray`

We reference the `bytearray` from offset 4 to its end ❶. Then, we write the content of */dev/urandom* from offset 4 to the end of `bytearray`, effectively reading just 4 bytes ❷.

The buffer protocol is extremely important for achieving low memory overhead and great performances. As Python hides all the memory allocations, developers tend to forget what happens under the hood, at great cost to the speed of their programs!

Both the objects in the `array` module and the functions in the `struct` module can handle the buffer protocol correctly and can therefore perform efficiently when targeting zero copying.

Summary

As we've seen in this chapter, there are plenty of ways to make Python code faster. Choosing the right data structure and using the correct methods for manipulating the data can have a huge impact in terms of CPU and memory usage. That's why it's important to understand what happens in Python internally.

However, optimization should never be done prematurely, without first performing a proper profiling. It is too easy to waste time rewriting some barely used code with a faster variant while missing central pain points. Don't miss the big picture.

Victor Stinner on Optimization

Victor is a longtime Python hacker, a core contributor, and the author of many Python modules. He authored PEP 454 in 2013, which proposed a new `tracemalloc` module to trace memory block allocation inside Python, and he wrote a simple AST optimizer called FAT. He also regularly contributes to the improvement of CPython performance.

What's a good starting strategy for optimizing Python code?

The strategy is the same in Python as in other languages. First, you need a well-defined use case in order to get a stable and reproducible benchmark. Without a reliable benchmark, trying different optimizations may result in wasted time and premature optimization. Useless optimizations may make the code worse, less readable, or even slower. A useful optimization must speed the program up by at least 5 percent if it's to be worth pursuing.

If a specific part of the code is identified as being "slow," a benchmark should be prepared on this code. A benchmark on a short function is usually called a *micro-benchmark*. The speedup should be at least 20 percent, maybe 25 percent, to justify an optimization on a micro-benchmark.

It may be interesting to run a benchmark on different computers, different operating systems, or different compilers. For example,

performances of `realloc()` may vary between Linux and Windows.

What are your recommended tools for profiling or optimizing Python code?

Python 3.3 has a `time.perf_counter()` function to measure elapsed time for a benchmark. It has the best resolution available.

A test should be run more than once; three times is a minimum, and five may be enough. Repeating a test fills disk cache and CPU caches. I prefer to keep the minimum timing; other developers prefer the geometric mean.

For micro-benchmarks, the `timeit` module is easy to use and gives results quickly, but the results are not reliable using default parameters. Tests should be repeated manually to get stable results.

Optimizing can take a lot of time, so it's better to focus on functions that use the most CPU power. To find these functions, Python has `cProfile` and `profile` modules to record the amount of time spent in each function.

Do you have any Python tricks that could improve performance?

You should reuse the Standard Library as much as possible—it's well tested and also usually efficient. Built-in Python types are implemented in C and have good performance. Use the correct container to get the best performance; Python provides many different kind of containers: `dict`, `list`, `deque`, `set`, and so on.

There are some hacks for optimizing Python, but you should avoid these because they make the code less readable in exchange for a minor speedup.

The Zen of Python (PEP 20) says, "There should be one—and preferably only one—obvious way to do it." In practice, there are different ways to write Python code, and performances are not the same. Only trust benchmarks on your use case.

Which areas of Python have the poorest performance and should be watched out for?

In general, I prefer not to worry about performance while developing a new application. Premature optimization is the root of all evil. When you identify slow functions, change the algorithm. If the algorithm and the container types are well chosen, you might rewrite short functions in C to get the best performance.

One bottleneck in CPython is the global interpreter lock, known as the GIL. Two threads cannot execute Python bytecode at the same time. However, this limitation only matters if two threads are executing pure Python code. If most processing time is spent in function calls, and these functions release the GIL, then the GIL is not the bottleneck. For example, most I/O functions release the GIL.

The multiprocessing module can easily be used to work around the GIL. Another option, more complex to implement, is to write asynchronous code. Twisted, Tornado, and Tulip projects, which are network-oriented libraries, make use of this technique.

What are some often-seen performance mistakes?

When Python is not well understood, inefficient code can be written. For example, I have seen `copy.deepcopy()` misused, when no copying was required.

Another performance killer is an inefficient data structure. With less than 100 items, the container type has no impact on performance. With more items, the complexity of each operation (`add`, `get`, `delete`) and its effects must be known.

11
SCALING AND ARCHITECTURE

Sooner or later, your development process will have to consider resiliency and scalability. An application's scalability, concurrency, and parallelism depend largely on its initial architecture and design. As we'll see in this chapter, there are some paradigms—such as multithreading—that don't apply correctly to Python, whereas other techniques, such as service-oriented architecture, work better.

Covering scalability in its entirety would take an entire book, and has in fact been covered by many books. This chapter covers the essential scaling fundamentals, even if you're not planning to build applications with millions of users.

Multithreading in Python and Its Limitations

By default, Python processes run on only one thread, called the *main thread*. This thread executes code on a single processor. *Multithreading* is a programming technique that allows code to run concurrently inside a single Python process by running several threads simultaneously. This is the primary mechanism through which we can introduce concurrency in Python. If the computer is equipped with multiple processors, you can even use *parallelism*, running threads in parallel over several processors, to make code execution faster.

Multithreading is most commonly used (though not always appropriately) when:

- You need to run background or I/O-oriented tasks without stopping your main thread's execution. For example, the main loop of a graphical user interface is busy waiting for an event (e.g., a user click or keyboard input), but the code needs to execute other tasks.
- You need to spread your workload across several CPUs.

The first scenario is a good general case for multithreading. Though implementing multithreading in this circumstance would introduce extra complexity, controlling multithreading would be manageable, and performance likely wouldn't suffer unless the CPU workload was intensive. The performance gain from using concurrency with workloads that are I/O intensive gets more interesting when the I/O has high latency: the more often you have to wait to read or write, the more beneficial it is to do something else in the meantime.

In the second scenario, you might want to start a new thread for each new request instead of handling them one at a time. This may seem like a good use for multithreading. However, if you spread your workload out like this, you will encounter the Python *global interpreter lock (GIL)*, a lock that must be acquired each time CPython needs to execute bytecode. The lock means that only one thread can have control of the Python interpreter at any one time. This rule was introduced originally to prevent race conditions, but it unfortunately means that if you try to scale your application by making it run multiple threads, you'll always be limited by this global lock.

So, while using threads seems like the ideal solution, most applications running requests in multiple threads struggle to attain 150 percent CPU usage, or usage of the equivalent of 1.5 cores. Most computers have 4 or 8 cores, and servers offer 24 or 48 cores, but the GIL prevents Python from using the full CPU. There are some initiatives underway to remove the GIL, but the effort is extremely complex because it requires performance and backward compatibility trade-offs.

Although CPython is the most commonly used implementation of the Python language, there are others that do not have a GIL. *Jython*, for example, can efficiently run multiple threads in parallel. Unfortunately, projects such as Jython by their very nature lag behind CPython and so are not really useful targets; innovation happens in CPython, and the other implementations are just following in CPython's footsteps.

So, let's revisit our two use cases with what we now know and figure out a better solution:

- When you need to run background tasks, you *can* use multithreading, but the easier solution is to build your application around an event loop. There are a lot of Python modules that provide for this, and the standard is now `asyncio`. There are also frameworks, such as Twisted, built around the same concept. The most advanced frameworks will give you access to events based on signals, timers, and file descriptor activity—we'll talk about this later in the chapter in "Event-Driven Architecture" on page 181.
- When you need to spread the workload, using multiple processes is the most efficient method. We'll look at this technique in the next section.

Developers should always think twice before using multithreading. As one example, I once used multithreading to dispatch jobs in rebuildd, a Debian-build daemon I wrote a few years ago. While it seemed handy to have a different thread to control each running build job, I very quickly fell into the threading-parallelism trap in Python. If I had the chance to begin again, I'd build something based on asynchronous event handling or multiprocessing and not have to worry about the GIL.

Multithreading is complex, and it's hard to get multithreaded applications right. You need to handle thread synchronization and locking, which means there are a lot of opportunities to introduce bugs. Considering the small overall gain, it's better to think twice before spending too much effort on it.

Multiprocessing vs. Multithreading

Since the GIL prevents multithreading from being a good scalability solution, look to the alternative solution offered by Python's *multiprocessing* package. The package exposes the same kind of interface you'd achieve using the multithreading module, except that it starts new *processes* (via `os.fork()`) instead of new system threads.

Listing 11-1 shows a simple example in which one million random integers are summed eight times, with this activity spread across eight threads at the same time.

```
import random
import threading
results = []
def compute():
    results.append(sum(
        [random.randint(1, 100) for i in range(1000000)]))
workers = [threading.Thread(target=compute) for x in range(8)]
for worker in workers:
    worker.start()
for worker in workers:
    worker.join()
print("Results: %s" % results)
```

Listing 11-1: Using multithreading for concurrent activity

In Listing 11-1, we create eight threads using the `threading.Thread` class and store them in the `workers` array. Those threads will execute the `compute()` function. They then use the `start()` method to start. The `join()` method only returns once the thread has terminated its execution. At this stage, the result can be printed.

Running this program returns the following:

```
$ time python worker.py
Results: [50517927, 50496846, 50494093, 50503078, 50512047, 50482863,
50543387, 50511493]
python worker.py  13.04s user 2.11s system 129% cpu 11.662 total
```

This has been run on an idle four-core CPU, which means that Python could potentially have used up to 400 percent of CPU. However, these results show that it was clearly unable to do that, even with eight threads running in parallel. Instead, its CPU usage maxed

out at 129 percent, which is just 32 percent of the hardware's capabilities (129/400).

Now, let's rewrite this implementation using *multiprocessing*. For a simple case like this, switching to multiprocessing is pretty straightforward, as shown in Listing 11-2.

```
import multiprocessing
import random

def compute(n):
    return sum(
        [random.randint(1, 100) for i in range(1000000)])

# Start 8 workers
pool = multiprocessing.Pool(processes=8)
print("Results: %s" % pool.map(compute, range(8)))
```

Listing 11-2: Using multiprocessing for concurrent activity

The `multiprocessing` module offers a `Pool` object that accepts as an argument the number of processes to start. Its `map()` method works in the same way as the native `map()` method, except that a different Python process will be responsible for the execution of the `compute()` function.

Running the program in Listing 11-2 under the same conditions as Listing 11-1 gives the following result:

```
$ time python workermp.py
Results: [50495989, 50566997, 50474532, 50531418, 50522470, 50488087,
0498016, 50537899]
python workermp.py  16.53s user 0.12s system 363% cpu 4.581 total
```

Multiprocessing reduces the exectution time by 60 percent. Moreover, we've been able to consume up to 363 percent of CPU power, which is more than 90 percent (363/400) of the computer's CPU capacity.

Each time you think that you can parallelize some work, it's almost always better to rely on multiprocessing and to fork your jobs in order to spread the workload across several CPU cores. This wouldn't be a good solution for very small execution times, as the cost of the `fork()` call would be too big, but for larger computing needs, it works well.

Event-Driven Architecture

Event-driven programming is characterized by the use of events, such as user input, to dictate how control flows through a program, and it is a good solution for organizing program flow. The event-driven program listens for various events happening on a queue and reacts based on those incoming events.

Let's say you want to build an application that listens for a connection on a socket and then processes the connection it receives. There are basically three ways to approach the problem:

- Fork a new process each time a new connection is established, relying on something like the `multiprocessing` module.
- Start a new thread each time a new connection is established, relying on something like the `threading` module.
- Add this new connection to your event loop and react to the event it will generate when it occurs.

Determining how a modern computer should handle tens of thousands of connections simultaneously is known as the *C10K problem*. Among other things, the C10K resolution strategies explain how using an event loop to listen to hundreds of event sources is going to scale much better than, say, a one-thread-per-connection approach. This doesn't mean that the two techniques are not compatible, but it does mean that you can usually replace the multiple-threads approach with an event-driven mechanism.

Event-driven architecture uses an event loop: the program calls a function that blocks execution until an event is received and ready to be processed. The idea is that your program can be kept busy doing other tasks while waiting for inputs and outputs to complete. The most basic events are "data ready to be read" and "data ready to be written."

In Unix, the standard functions for building such an event loop are the system calls `select(2)` or `poll(2)`. These functions expect a list of file descriptors to listen for, and they will return as soon as at least one of the file descriptors is ready to be read from or written to.

In Python, we can access these system calls through the `select` module. It's easy enough to build an event-driven system with these calls, though doing so can be tedious. Listing 11-3 shows an event-driven system that does our specified task: listening on a socket and processing any connections it receives.

```
import select
import socket

server = socket.socket(socket.AF_INET,
                       socket.SOCK_STREAM)
# Never block on read/write operations
server.setblocking(0)

# Bind the socket to the port
server.bind(('localhost', 10000))
server.listen(8)

while True:
    # select() returns 3 arrays containing the object (sockets, files...)

    # that are ready to be read, written to or raised an error
inputs,
outputs, excepts = select.select([server], [], [server])
    if server in inputs:
        connection, client_address = server.accept()
        connection.send("hello!\n")
```

Listing 11-3: Event-driven program that listens for and processes connections

In Listing 11-3, a server socket is created and set to *non-blocking*, meaning that any read or write operation attempted on that socket won't block the program. If the program tries to read from the socket when there is no data ready to be read, the socket `recv()` method will raise an OSError indicating that the socket is not ready. If we did not call `setblocking(0)`, the socket would stay in blocking mode rather than raise an error, which is not what we want here. The socket is then bound to a port and listens with a maximum backlog of eight connections.

The main loop is built using `select()`, which receives the list of file descriptors we want to read (the socket in this case), the list of file descriptors we want to write to (none in this case), and the list of file descriptors we want to get exceptions from (the socket in this case). The `select()` function returns as soon as one of the selected file descriptors is

ready to read, is ready to write, or has raised an exception. The returned values are lists of file descriptors that match the requests. It's then easy to check whether our socket is in the ready-to-be-read list and, if so, accept the connection and send a message.

Other Options and asyncio

Alternatively, there are many frameworks, such as Twisted or Tornado, that provide this kind of functionality in a more integrated manner; Twisted has been the de facto standard for years in this regard. C libraries that export Python interfaces, such as `libevent`, `libev`, or `libuv`, also provide very efficient event loops.

These options all solve the same problem. The downside is that, while there are a wide variety of choices, most of them are not interoperable. Many are also *callback based*, meaning that the program flow is not very clear when reading the code; you have to jump to a lot of different places to read through the program.

Another option would be the `gevent` or `greenlet` libraries, which avoid callback use. However, the implementation details include CPython x86–specific code and dynamic modification of standard functions at runtime, meaning you wouldn't want to use and maintain code using these libraries over the long term.

In 2012, Guido Van Rossum began work on a solution code-named *tulip*, documented under PEP 3156 (*https://www.python.org/dev/peps/pep-3156*). The goal of this package was to provide a standard event loop interface that would be compatible with all frameworks and libraries and be interoperable.

The tulip code has since been renamed and merged into Python 3.4 as the `asyncio` module, and it is now the de facto standard. Not all libraries are compatible with `asyncio`, and most existing bindings need to be rewritten.

As of Python 3.6, `asyncio` has been so well integrated that it has its own `await` and `async` keywords, making it straightforward to use. Listing 11-4 shows how the `aiohttp` library, which provides an asynchronous

HTTP binding, can be used with `asyncio` to run several web page retrievals concurrently.

```
import aiohttp
import asyncio


async def get(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return response


loop = asyncio.get_event_loop()

coroutines = [get("http://example.com") for _ in range(8)]

results = loop.run_until_complete(asyncio.gather(*coroutines))

print("Results: %s" % results)
```

Listing 11-4: Retrieving web pages concurrently with `aiohttp`

We define the `get()` function as asynchronous, so it is technically a coroutine. The `get()` function's two steps, the connection and the page retrieval, are defined as asynchronous operations that yield control to the caller until they are ready. That makes it possible for `asyncio` to schedule another coroutine at any point. The module resumes the execution of a coroutine when the connection is established or the page is ready to be read. The eight coroutines are started and provided to the event loop at the same time, and it is `asyncio`'s job to schedule them efficiently.

The `asyncio` module is a great framework for writing asynchronous code and leveraging event loops. It supports files, sockets, and more, and a lot of third-party libraries are available to support various protocols. Don't hesitate to use it!

Service-Oriented Architecture

Circumventing Python's scaling shortcomings can seem tricky. However, Python *is* very good at implementing *service-oriented architecture (SOA)*, a style of software design in which different

components provide a set of services through a communication protocol. For example, OpenStack uses SOA architecture in all of its components. The components use HTTP REST to communicate with external clients (end users) and an abstracted remote procedure call (RPC) mechanism that is built on top of the Advanced Message Queuing Protocol (AMQP).

In your development situations, knowing which communication channels to use between those blocks is mainly a matter of knowing with whom you will be communicating.

When exposing a service to the outside world, the preferred channel is HTTP, especially for stateless designs such as REST-style (REpresentational State Transfer–style) architectures. These kinds of architectures make it easier to implement, scale, deploy, and comprehend services.

However, when exposing and using your API internally, HTTP may be not the best protocol. There are many other communication protocols and fully describing even one would likely fill an entire book.

In Python, there are plenty of libraries for building RPC systems. Kombu is interesting because it provides an RPC mechanism on top of a lot of backends, AMQ protocol being the main one. It also supports Redis, MongoDB, Beanstalk, Amazon SQS, CouchDB, or ZooKeeper.

In the end, you can indirectly gain a huge amount of performance from using such loosely coupled architecture. If we consider that each module provides and exposes an API, we can run multiple daemons that can also expose that API, allowing multiple processes—and therefore CPUs—to handle the workload. For example, *Apache httpd* would create a new `worker` using a new system process that handles new connections; we could then dispatch a connection to a different `worker` running on the same node. To do so, we just need a system for dispatching the work to our various `workers`, which this API provides. Each block will be a different Python process, and as we've seen previously, this approach is better than multithreading for spreading out your workload. You'll be able to start multiple `workers` on each node. Even if stateless blocks are not strictly necessary, you should favor their use anytime you have the choice.

Interprocess Communication with ZeroMQ

As we've just discussed, a messaging bus is always needed when building distributed systems. Your processes need to communicate with each other in order to pass messages. `ZeroMQ` is a socket library that can act as a concurrency framework. Listing 11-5 implements the same `worker` seen in Listing 11-1 but uses `ZeroMQ` as a way to dispatch work and communicate between processes.

```
   import multiprocessing
   import random
   import zmq

   def compute():
       return sum(
           [random.randint(1, 100) for i in range(1000000)])

   def worker():
       context = zmq.Context()
       work_receiver = context.socket(zmq.PULL)
       work_receiver.connect("tcp://0.0.0.0:5555")
       result_sender = context.socket(zmq.PUSH)
       result_sender.connect("tcp://0.0.0.0:5556")
       poller = zmq.Poller()
       poller.register(work_receiver, zmq.POLLIN)

       while True:
           socks = dict(poller.poll())
           if socks.get(work_receiver) == zmq.POLLIN:
               obj = work_receiver.recv_pyobj()
               result_sender.send_pyobj(obj())

   context = zmq.Context()
   # Build a channel to send work to be done
❶ work_sender = context.socket(zmq.PUSH)
   work_sender.bind("tcp://0.0.0.0:5555")
   # Build a channel to receive computed results
❷ result_receiver = context.socket(zmq.PULL)
   result_receiver.bind("tcp://0.0.0.0:5556")
   # Start 8 workers
   processes = []
   for x in range(8):
❸      p = multiprocessing.Process(target=worker)
       p.start()
       processes.append(p)
   # Send 8 jobs
   for x in range(8):
       work_sender.send_pyobj(compute)
   # Read 8 results

   results = []
```

```
    for x in range(8):
❹       results.append(result_receiver.recv_pyobj())
    # Terminate all processes
    for p in processes:
        p.terminate()
    print("Results: %s" % results)
```

Listing 11-5: `workers` using `ZeroMQ`

We create two sockets, one to send the function (`work_sender`) ❶ and one to receive the job (`result_receiver`) ❷. Each `worker` started by `multiprocessing.Process` ❸ creates its own set of sockets and connects them to the master process. The `worker` then executes whatever function is sent to it and sends back the result. The master process just has to send eight jobs over its sender socket and wait for eight results to be sent back via the receiver socket ❹.

As you can see, `ZeroMQ` provides an easy way to build communication channels. I've chosen to use the TCP transport layer here to illustrate the fact that we could run this over a network. It should be noted that `ZeroMQ` also provides an interprocess communication channel that works locally (without any network layer involved) by using Unix sockets. Obviously, the communication protocol built upon `ZeroMQ` in this example is very simple for the sake of being clear and concise, but it shouldn't be hard to imagine building a more sophisticated communication layer on top of it. It's also easy to imagine building an entirely distributed application communication with a network message bus such as ZeroMQ or AMQP.

Note that protocols such as HTTP, ZeroMQ, and AMQP are language agnostic: you can use different languages and platforms to implement each part of your system. While we all agree that Python is a good language, other teams might have other preferences, or another language might be a better solution for some part of a problem.

In the end, using a transport bus to decouple your application into several parts is a good option. This approach allows you to build both synchronous and asynchronous APIs that can be distributed from one computer to several thousand. It doesn't tie you to a particular

technology or language, so you can evolve everything in the right direction.

Summary

The rule of thumb in Python is to use threads only for I/O-intensive workloads and to switch to multiple processes as soon as a CPU-intensive workload is on the table. Distributing workloads on a wider scale—such as when building a distributed system over a network—requires external libraries and protocols. These are supported by Python, though provided externally.

12
MANAGING RELATIONAL DATABASES

Applications will almost always have to store data of some kind, and developers will often combine a relational database management system (RDBMS) with some type of object relational mapping tool (ORM). RDBMSs and ORMs can be tricky and are not a favorite topic for many developers, but sooner or later, they must be addressed.

RDBMSs, ORMs, and When to Use Them

An RDBMS is the database that stores an application's relational data. Developers will use a language like SQL (Structured Query Language) to deal with the relational algebra, meaning that a language like this handles the data management and the relationships between the data. Used together, they allow you to both store data and query that data to get specific information as efficiently as possible. Having a good understanding of relational database structures, such as how to use proper normalization or the different types of serializability, might keep you from falling into many traps. Obviously, such subjects deserve an entire book and won't be covered in their entirety in this chapter; instead, we'll focus on using the database via its usual programming language, SQL.

Developers may not want to invest in learning a whole new programming language to interact with the RDBMS. If so, they tend to

avoid writing SQL queries entirely, relying instead on a library to do the work for them. ORM libraries are commonly found in programming language ecosystems, and Python is no exception.

The purpose of an ORM is to make database systems easier to access by abstracting the process of creating queries: it generates the SQL so you don't have to. Unfortunately, this abstraction layer can prevent you from performing more specific or low-level tasks that the ORM is simply not capable of doing, such as writing complex queries.

There is also a particular set of difficulties with using ORMs in object-oriented programs that are so common, they are known collectively as the *object-relational impedance mismatch*. This impedance mismatch occurs because relational databases and object-oriented programs have different representations of data that don't map properly to one another: mapping SQL tables to Python classes won't give you optimal results, no matter what you do.

Understanding SQL and RDBMSs will allow you to write your own queries, without having to rely on the abstraction layer for everything.

But that's not to say you should avoid ORMs entirely. ORM libraries can help with rapid prototyping of your application model, and some libraries even provide useful tools such as schema upgrades and downgrades. It's important to understand that using an ORM is not a substitute for gaining a real understanding of RDBMSs: many developers try to solve problems in the language of their choice rather than using their model API, and the solutions they come up with are inelegant at best.

NOTE

This chapter assumes you know basic SQL. Introducing SQL queries and discussing how tables work is beyond the scope of this book. If you're new to SQL, I recommend learning the basics before continuing. Practical SQL *by Anthony DeBarros (No Starch Press, 2018) is a good place to start.*

Let's look at an example that demonstrates why understanding RDBMSs can help you write better code. Say you have a SQL table for keeping track of messages. This table has a single column named `id` representing the ID of the message sender, which is the primary key, and a string containing the content of the message, like so:

```
CREATE TABLE message (
  id serial PRIMARY KEY,
  content text
);
```

We want to detect any duplicate messages received and exclude them from the database. To do this, a typical developer might write SQL using an ORM, as shown in Listing 12-1.

```
if query.select(Message).filter(Message.id == some_id):
    # We already have the message, it's a duplicate, ignore and raise
    raise DuplicateMessage(message)
else:
    # Insert the message
    query.insert(message)
```

Listing 12-1: Detecting and excluding duplicate messages with an ORM

This code works for most cases, but it has some major drawbacks:

- The duplicate constraint is already expressed in the SQL schema, so there is a sort of code duplication: using `PRIMARY KEY` implicitly defines the uniqueness of the `id` field.
- If the message is not yet in the database, this code executes two SQL queries: a `SELECT` statement and then an `INSERT` statement. Executing a SQL query might take a long time and require a round-trip to the SQL server, introducing extraneous delay.
- The code doesn't account for the possibility that someone else might insert a duplicate message after we call `select_by_id()` but before we call `insert()`, which would cause the program to raise an exception. This vulnerability is called a *race condition*.

There's a much better way to write this code, but it requires cooperation with the RDBMS server. Rather than checking for the

message's existence and then inserting it, we can insert it right away and use a `try...except` block to catch a duplicate conflict:

```
try:
    # Insert the message
    message_table.insert(message)
except UniqueViolationError:
    # Duplicate
    raise DuplicateMessage(message)
```

In this case, inserting the message directly into the table works flawlessly if the message is not already present. If it is, the ORM raises an exception indicating the violation of the uniqueness constraint. This method achieves the same effect as Listing 12-1 but in a more efficient fashion and without any race condition. This is a very simple pattern, and it doesn't conflict with any ORM in any way. The problem is that developers tend to treat SQL databases as dumb storage rather than as a tool they can use to get proper data integrity and consistency; consequently, they may duplicate the constraints written in SQL in their controller code rather than in their model.

Treating your SQL backend as a model API is good way to make efficient use of it. You can manipulate the data stored in your RDBMS with simple function calls programmed in its own procedural language.

Database Backends

ORM supports multiple database backends. No ORM library provides a complete abstraction of all RDBMS features, and simplifying the code to the most basic RDBMS available will make using any advanced RDBMS functions impossible without breaking the abstraction layer. Even simple things that aren't standardized in SQL, such as handling timestamp operations, are a pain to deal with when using an ORM. This is even more true if your code is RDBMS agnostic. It is important to keep this in mind when you choose your application's RDBMS.

Isolating ORM libraries (as described in "External Libraries" on page 22) helps mitigate potential problems. This approach allows you to easily swap your ORM library for a different one should the need arise

and to optimize your SQL usage by identifying places with inefficient query usage, which lets you bypass most of the ORM boilerplate.

For example, you can use your ORM in a module of your application, such as `myapp.storage`, to easily build in such isolation. This module should export only functions and methods that allow you to manipulate the data at a high level of abstraction. The ORM should be used only from that module. At any point, you will be able to drop in any module providing the same API to replace `myapp.storage`.

The most commonly used ORM library in Python (and arguably the de facto standard) is `sqlalchemy`. This library supports a huge number of backends and provides abstraction for most common operations. Schema upgrades can be handled by third-party packages such as `alembic` (*https://pypi.python.org/pypi/alembic/*).

Some frameworks, such as Django (*https://www.djangoproject.com*), provide their own ORM libraries. If you choose to use a framework, it's smart to use the built-in library because it will often integrate better with the framework than an external one.

WARNING

The Module View Controller (MVC) architecture that most frameworks rely on can be easily misused. These frameworks implement (or make it easy to implement) ORM in their models directly, but without abstracting enough of it: any code you have in your view and controllers that use the model will also be using ORM directly. You need to avoid this. You should write a data model that includes *the ORM library rather than* consists of *it. Doing so provides better testability and isolation, and makes swapping out the ORM with another storage technology much easier.*

Streaming Data with Flask and PostgreSQL

Here, I'll show you how you can use one of *PostgreSQL*'s advanced features to build an HTTP event-streaming system to help master your data storage.

Writing the Data-Streaming Application

The purpose of the micro-application in Listing 12-2 is to store messages in a SQL table and provide access to those messages via an HTTP REST API. Each message consists of a channel number, a source string, and a content string.

```
CREATE TABLE message (
  id SERIAL PRIMARY KEY,
  channel INTEGER NOT NULL,
  source TEXT NOT NULL,
  content TEXT NOT NULL
);
```

Listing 12-2: SQL table schema for storing messages

We also want to stream these messages to the client so that it can process them in real time. To do this, we're going to use the `LISTEN` and `NOTIFY` features of PostgreSQL. These features allow us to listen for messages sent by a function we provide that PostgreSQL will execute:

```
❶ CREATE OR REPLACE FUNCTION notify_on_insert() RETURNS trigger AS $$
❷ BEGIN
    PERFORM pg_notify('channel_' || NEW.channel,
                      CAST(row_to_json(NEW) AS TEXT));
    RETURN NULL;
  END;
  $$ LANGUAGE plpgsql;
```

This code creates a trigger function written in pl/pgsql, a language that only PostgreSQL understands. Note that we could also write this function in other languages, such as Python itself, as PostgreSQL embeds the Python interpreter in order to provide a pl/python language. The single simple operation we'll be performing here does not necessitate leveraging Python, so sticking to pl/pgsql is a wise choice.

The function `notify_on_insert()` ❶ performs a call to `pg_notify()` ❷, which is the function that actually sends the notification. The first argument is a string that represents a *channel*, while the second is a string carrying the actual *payload*. We define the channel dynamically based on the value of the channel column in the row. In this case, the

payload will be the entire row in JSON format. Yes, PostgreSQL knows how to convert a row to JSON natively!

Next, we want to send a notification message on each `INSERT` performed in the message table, so we need to trigger this function on such events:

```
CREATE TRIGGER notify_on_message_insert AFTER INSERT ON message
FOR EACH ROW EXECUTE PROCEDURE notify_on_insert();
```

The function is now plugged in and will be executed upon each successful `INSERT` performed in the message table.

We can check that it works by using the `LISTEN` operation in `psql`:

```
$ psql
psql (9.3rc1)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.

mydatabase=> LISTEN channel_1;
LISTEN
mydatabase=> INSERT INTO message(channel, source, content)
mydatabase-> VALUES(1, 'jd', 'hello world');
INSERT 0 1
Asynchronous notification "channel_1" with payload
"{"id":1,"channel":1,"source":"jd","content":"hello world"}"
received from server process with PID 26393.
```

As soon as the row is inserted, the notification is sent, and we're able to receive it through the PostgreSQL client. Now all we have to do is build the Python application that streams this event, shown in Listing 12-3.

```
import psycopg2
import psycopg2.extensions
import select

conn = psycopg2.connect(database='mydatabase', user='myuser',
                        password='idkfa', host='localhost')

conn.set_isolation_level(
    psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)

curs = conn.cursor()
curs.execute("LISTEN channel_1;")

while True:
    select.select([conn], [], [])
```

```
    conn.poll()
    while conn.notifies:
        notify = conn.notifies.pop()
        print("Got NOTIFY:", notify.pid, notify.channel,
notify.payload)
```

Listing 12-3: Listening and receiving the stream of notifications

Listing 12-3 connects to PostgreSQL using the `psycopg2` library. The `psycopg2` library is a Python module that implements the PostgreSQL network protocol and allows us to connect to a PostgreSQL server to send SQL requests and receive results. We could have used a library that provides an abstraction layer, such as `sqlalchemy`, but abstracted libraries don't provide access to the `LISTEN` and `NOTIFY` functionality of PostgreSQL. It's important to note that it is still possible to access the underlying database connection to execute the code when using a library like `sqlalchemy`, but there would be no point in doing that for this example, since we don't need any of the other features the ORM library provides.

The program listens on `channel_1`, and as soon as it receives a notification, prints it to the screen. If we run the program and insert a row in the `message` table, we get the following output:

```
$ python listen.py
Got NOTIFY: 28797 channel_1
{"id":10,"channel":1,"source":"jd","content":"hello world"}
```

As soon as we insert the row, PostgreSQL runs the trigger and sends a notification. Our program receives it and prints the notification payload; here, that's the row serialized to JSON. We now have the basic ability to receive data as it is inserted into the database, without doing any extra requests or work.

Building the Application

Next, we'll use *Flask*, a simple HTTP micro-framework, to build our application. We're going to build an HTTP server that streams the flux of `insert` using the *Server-Sent Events* message protocol defined by

HTML5. An alternative would be to use *Transfer-Encoding: chunked* defined by HTTP/1.1:

```
import flask
import psycopg2
import psycopg2.extensions
import select

app = flask.Flask(__name__)

def stream_messages(channel):
    conn = psycopg2.connect(database='mydatabase', user='mydatabase',
                            password='mydatabase', host='localhost')
    conn.set_isolation_level(
        psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)

    curs = conn.cursor()
    curs.execute("LISTEN channel_%d;" % int(channel))

    while True:
        select.select([conn], [], [])
        conn.poll()
        while conn.notifies:
            notify = conn.notifies.pop()
            yield "data: " + notify.payload + "\n\n"

@app.route("/message/<channel>", methods=['GET'])
def get_messages(channel):
    return flask.Response(stream_messages(channel),
                          mimetype='text/event-stream')

if __name__ == "__main__":
    app.run()
```

This application is simple enough that it supports streaming but not any other data retrieval operation. We use Flask to route the HTTP request `GET /message/`*`channel`* to our streaming code. As soon as the code is called, the application returns a response with the mimetype `text/event-stream` and sends back a generator function instead of a string. Flask will call this function and send results each time the generator yields something.

The generator, `stream_messages()`, reuses the code we wrote earlier to listen to PostgreSQL notifications. It receives the channel identifier as an argument, listens to that channel, and then yields the payload. Remember that we used PostgreSQL's JSON encoding function in the trigger function, so we're already receiving JSON data from

PostgreSQL. There's no need for us to transcode the data since it's fine to send JSON data to the HTTP client.

> **NOTE**
>
> *For the sake of simplicity, this example application has been written in a single file. If this were a real application, I would move the storage-handling implementation into its own Python module.*

We can now run the server:

```
$ python listen+http.py
 * Running on http://127.0.0.1:5000/
```

On another terminal, we can connect and retrieve the events as they're entered. Upon connection, no data is received, and the connection is kept open:

```
$ curl -v http://127.0.0.1:5000/message/1
* About to connect() to 127.0.0.1 port 5000 (#0)
*   Trying 127.0.0.1...
* Adding handle: conn: 0x1d46e90
* Adding handle: send: 0
* Adding handle: recv: 0
* Curl_addHandleToPipeline: length: 1
* - Conn 0 (0x1d46e90) send_pipe: 1, recv_pipe: 0
* Connected to 127.0.0.1 (127.0.0.1) port 5000 (#0)
> GET /message/1 HTTP/1.1
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:5000
> Accept: */*
>
```

But as soon as we insert some rows in the `message` table, we'll start seeing data coming in through the terminal running `curl`. In a third terminal, we insert a message in the database:

```
mydatabase=> INSERT INTO message(channel, source, content)
mydatabase-> VALUES(1, 'jd', 'hello world');
INSERT 0 1
mydatabase=> INSERT INTO message(channel, source, content)
mydatabase-> VALUES(1, 'jd', 'it works');
INSERT 0 1
```

Here's the data output:

```
data: {"id":71,"channel":1,"source":"jd","content":"hello world"}
data: {"id":72,"channel":1,"source":"jd","content":"it works"}
```

This data is printed to the terminal running `curl`. This keeps `curl` connected to the HTTP server while it waits for the next flux of messages. We created a streaming service without doing any kind of polling here, building an entirely *push-based* system where information flows from one point to another seamlessly.

A naive and arguably more portable implementation of this application would instead repeatedly loop over a `SELECT` statement to poll for new data inserted in the table. This would work with any other storage system that does not support a publish-subscribe pattern as this one does.

Dimitri Fontaine on Databases

Dimitri is a skilled PostgreSQL Major Contributor who works at Citus Data and argues with other database gurus on the *pgsql-hackers* mailing list. We've shared a lot of open source adventures, and he's been kind enough to answer some questions about what you should do when dealing with databases.

What advice would you give to developers using RDBMSs as their storage backends?

RDBMSs were invented in the '70s to solve some common problems plaguing every application developer at that time, and the main services implemented by RDBMSs were not simply data storage.

The main services offered by an RDBMS are actually the following:

- **Concurrency:** Access your data for read or write with as many concurrent threads of execution as you want—the RDBMS is there to handle that correctly for you. That's the main feature you want out of an RDBMS.

- **Concurrency semantics:** The details about the concurrency behavior when using an RDBMS are proposed with a high-level specification in terms of atomicity and isolation, which are maybe the most crucial parts of ACID (atomicity, consistency, isolation, durability). *Atomicity* is the property that between the time you `BEGIN` a transaction and the time you're done with it (either `COMMIT` or `ROLLBACK`), no other concurrent activity on the system is allowed to know what you're doing—whatever that is. When using a proper RDBMS, also include the Data Definition Language (DDL), for example, `CREATE TABLE` or `ALTER TABLE`. *Isolation* is all about what you're allowed to notice of the concurrent activity of the system from within your own transaction. The SQL standard defines four levels of isolation, as described in the PostgreSQL documentation (*http://www.postgresql.org/docs/9.2/static/transaction-iso.html*).

The RDBMS takes full responsibility for your data. So it allows the developer to describe their own rules for consistency, and then it will check that those rules are valid at crucial times such as at transaction commit or at statement boundaries, depending on the deferability of your constraints declarations.

The first constraint you can place on your data is its expected input and output formatting, using the proper data type. An RDBMS will know how to work with much more than text, numbers, and dates and will properly handle dates that actually appear in a calendar in use today.

Data types are not just about input and output formats, though. They also implement behaviors and some level of polymorphism, as we all expect the basic equality tests to be data type specific: we don't compare text and numbers, dates and IP addresses, arrays and ranges, and so on in the same way.

Protecting your data also means that the only choice for an RDBMS is to actively refuse data that doesn't match your consistency rules, the first of which is the data type you've chosen. If you think it's okay to have to deal with a date such as 0000-00-00 that never existed in the calendar, then you need to rethink.

The other part of the consistency guarantees is expressed in terms of constraints as in `CHECK` constraints, `NOT NULL` constraints, and constraint triggers, one of which is known as foreign key. All of that can be thought of as a user-level extension of the data type definition and behavior, the main difference being that you can choose to `DEFER` the enforcement of checking those constraints from the end of each statement to the end of the current transaction.

The relational bits of an RDBMS are all about modeling your data and the guarantee that all tuples found in a relation share a common set of rules: structure and constraints. When enforcing that, we are enforcing the use of a proper explicit schema to handle our data.

Working on a proper schema for your data is known as *normalization*, and you can aim for a number of subtly different normal forms in your design. Sometimes though, you need more flexibility than what is given by the result of your normalization process. Common wisdom is to first normalize your data schema and only then modify it to regain some flexibility. Chances are you actually don't need more flexibility.

When you do need more flexibility, you can use PostgreSQL to try out a number of denormalization options: composite types, records, arrays, H-Store, JSON, or XML to name a few.

There's a very important drawback to denormalization though, which is that the query language we're going to talk about next is designed to handle rather normalized data. With PostgreSQL, of course, the query language has been extended to support as much denormalization as possible when using composite types, arrays or H-Store, and even JSON in recent releases.

The RDBMS knows a lot about your data and can help you implement a very fine-grain security model, should you need to do so. The access patterns are managed at the relation and column level, and PostgreSQL also implements `SECURITY DEFINER` stored procedures, allowing you to offer access to sensible data in a very controlled way, much the same as with using saved user ID (SUID) programs.

The RDBMS offers to access your data using a SQL, which became the de facto standard in the '80s and is now driven by a committee. In the case of PostgreSQL, lots of extensions are being added, with each

and every major release allowing you to access a very rich DSL language. All the work of query planning and optimization is done for you by the RDBMS so that you can focus on a declarative query where you describe only the result you want from the data you have.

And that's also why you need to pay close attention to the NoSQL offerings here, as most of those trendy products are in fact not removing just the SQL from the offering but a whole lot of other foundations that you've been trained to expect.

What advice would you give to developers using RDBMSs as their storage backends?

My advice is to remember the differences between a *storage backend* and an RDBMS. Those are very different services, and if all you need is a storage backend, maybe consider using something other than an RDBMS.

Most often, though, what you really need is a full-blown RDBMS. In that case, the best option you have is PostgreSQL. Go read its documentation (*https://www.postgresql.org/docs/*); see the list of data types, operators, functions, features, and extensions it provides. Read some usage examples on blog posts.

Then consider PostgreSQL a tool you can leverage in your development and include it in your application architecture. Parts of the services you need to implement are best offered at the RDBMS layer, and PostgreSQL excels at being that trustworthy part of your whole implementation.

What's the best way to use or not use an ORM?

The ORM will best work for *CRUD* applications: create, read, update, and delete. The read part should be limited to a very simple SELECT statement targeting a single table, as retrieving more columns than necessary has a significant impact on query performances and resources used.

Any column you retrieve from the RDBMS and that you end up not using is pure waste of precious resources, a first scalability killer. Even when your ORM is able to fetch only the data you're asking for, you still

then have to somehow manage the exact list of columns you want in each situation, without using a simple abstract method that will automatically compute the fields list for you.

The create, update, and delete queries are simple `INSERT`, `UPDATE`, and `DELETE` statements. Many RDBMSs offer optimizations that are not leveraged by ORMs, such as returning data after an `INSERT`.

Furthermore, in the general case, a relation is either a table or the result of any query. It's common practice when using an ORM to build relational mapping between defined tables and some model classes, or some other helper stubs.

If you consider the whole SQL semantics in their generalities, then the relational mapper should really be able to map any query against a class. You would then presumably have to build a new class for each query you want to run.

The idea when applied to our case is that you trust your ORM to do a better job than you at writing efficient SQL queries, even when you're not giving it enough information to work out the exact set of data you are interested in.

It's true that, at times, SQL can get quite complex, though you're not going to get anywhere near simplicity by using an API-to-SQL generator that you can't control.

However, there are two cases where you can relax and use your ORM, provided that you're willing to accept the following compromise: at a later point, you may need to edit your ORM usage out of your codebase.

- **Time to market:** When you're really in a hurry and want to gain market share as soon as possible, the only way to get there is to release a first version of your application and idea. If your team is more proficient at using an ORM than handcrafting SQL queries, then by all means just do that. You have to realize, though, that as soon as you're successful with your application, one of the first scalability problems you will have to solve is going to be related to your ORM producing really bad queries. Also, your usage of the ORM will have painted you into a corner and resulted in bad code

design decisions. But if you're there, you're successful enough to spend some refactoring money and remove any dependency on the ORM, right?

- **CRUD application:** This is the real thing, where you are only editing a single tuple at a time and you don't really care about performance, like for the basic admin application interface.

What are the pros of using PostgreSQL over other databases when working with Python?

Here are my top reasons for choosing PostgreSQL as a developer:

- **Community support:** The PostgreSQL community is vast and welcoming to new users, and folks will typically take the time to provide the best possible answer. The mailing lists are still the best way to communicate with the community.
- **Data integrity and durability:** Any data you send to PostgreSQL is safe in its definition and your ability to fetch it again later.
- **Data types, functions, operators, arrays, and ranges:** PostgreSQL has a very rich set of data types that come with a host of operators and functions. It's even possible to denormalize using arrays or JSON data types and still be able to write advanced queries, including joins, against those.
- **The planner and optimizer:** It's worth taking the time to understand how complex and powerful these are.
- **Transactional DDL:** It's possible to `ROLLBACK` almost any command. Try it now: just open your `psql` shell against a database you have and type in `BEGIN; DROP TABLE foo; ROLLBACK;`, where you replace *`foo`* with the name of a table that exists in your local instance. Amazing, right?
- **PL/Python (and others such as C, SQL, Javascript, or Lua):** You can run your own Python code on the server, right where the data is, so you don't have to fetch it over the network just to process it and then send it back in a query to do the next level of `JOIN`.

- **Specific indexing (GiST, GIN, SP-GiST, partial and functional):** You can create Python functions to process your data from within PostgreSQL and then index the result of calling that function. When you issue a query with a `WHERE` clause calling that function, it's called only once with the data from the query; then it's matched directly with the contents of the index.

13
WRITE LESS, CODE MORE

In this final chapter, I've compiled a few of Python's more advanced features that I use to write better code. These are not limited to the Python Standard Library. We'll cover how to make your code compatible with both Python 2 and 3, how to create a Lisp-like method dispatcher, how to use context managers, and how to create a boilerplate for classes with the `attr` module.

Using six for Python 2 and 3 Support

As you likely know, Python 3 breaks compatibility with Python 2 and shifts things around. However, the basics of the language haven't changed between versions, which makes it possible to implement forward and backward compatibility, creating a bridge between Python 2 and Python 3.

Lucky for us, this module already exists! It's called `six`—because 2 × 3 = 6.

The `six` module provides the useful `six.PY3` variable, which is a Boolean that indicates whether you are running Python 3 or not. This is the pivot variable for any of your codebase that has two versions: one for Python 2 and one for Python 3. However, be careful not to abuse it; scattering your codebase with `if six.PY3` is going to make it difficult for people to read and understand.

When we discussed generators in “Generators” on page 121, we saw that Python 3 has a great property whereby iterable objects are returned instead of lists in various built-in functions, such as `map()` or `filter()`. Python 3 therefore got rid of methods like `dict.iteritems()`, which was the iterable version of `dict.items()` in Python 2, in favor of making `dict.items()` return an iterator rather than a list. This change in methods and their return types can break your Python 2 code.

The `six` module provides `six.iteritems()` for such cases, which can be used to replace Python 2–specific code like this:

```
for k, v in mydict.iteritems():
    print(k, v)
```

Using `six`, you would replace the `mydict.iteritems()` code with Python 2- and 3-compliant code like so:

```
import six

for k, v in six.iteritems(mydict):
    print(k, v)
```

And *voilà*, both Python 2 and Python 3 compliance achieved in a snap! The `six.iteritems()` function will use either `dict.iteritems()` or `dict.items()` to return a generator, depending on the version of Python you're using. The `six` module provides a lot of similar helper functions that can make it easy to support multiple Python versions.

Another example would be the `six` solution to the `raise` keyword, whose syntax is different between Python 2 and Python 3. In Python 2, `raise` will accept multiple arguments, but in Python 3, `raise` accepts an exception as its only argument and nothing else. Writing a `raise` statement with two or three arguments in Python 3 would result in a `SyntaxError`.

The `six` module provides a workaround here in the form of the function `six.reraise()`, which allows you to reraise an exception in whichever version of Python you use.

Strings and Unicode

Python 3's enhanced ability to handle advanced encodings solved the string and unicode issues of Python 2. In Python 2, the basic string type is `str`, which can only handle basic ASCII strings. The type `unicode`, added later in Python 2.5, handles real strings of text.

In Python 3, the basic string type is still `str`, but it shares the properties of the Python 2 `unicode` class and can handle advanced encodings. The `bytes` type replaces the `str` type for handling basic character streams.

The `six` module again provides functions and constants, such as `six.u` and `six.string_types`, to handle the transition. The same compatibility is provided for integers, with `six.integer_types` that will handle the `long` type that has been removed from Python 3.

Handling Python Modules Moves

In the Python Standard Library, some modules have moved or have been renamed between Python 2 and 3. The `six` module provides a module called `six.moves` that handles a lot of these moves transparently.

For example, the `ConfigParser` module from Python 2 has been renamed to `configparser` in Python 3. Listing 13-1 shows how code can be ported and made compatible with both major Python versions using `six.moves`:

```
from six.moves.configparser import ConfigParser

conf = ConfigParser()
```

Listing 13-1: Using `six.moves` to use `ConfigParser()` with Python 2 and Python 3

You can also add your own moves via `six.add_move` to handle code transitions that `six` doesn't handle natively.

In the event that the `six` library doesn't cover all your use cases, it may be worth building a compatibility module encapsulating `six` itself, thereby ensuring that you will be able to enhance the module to fit future versions of Python or dispose of (part of) it when you want to stop supporting a particular version of the language. Also note that `six` is

open source and that you can contribute to it rather than maintain your own hacks!

The modernize Module

Lastly, there is a tool named `modernize` that uses the `six` module to "modernize" your code by porting it to Python 3, rather than simply converting Python 2 syntax to Python 3 syntax. This provides support for both Python 2 and Python 3. The `modernize` tool helps to get your port off to a strong start by doing most of the grunt work for you, making this tool a better choice than the standard `2to3` tool.

Using Python Like Lisp to Make a Single Dispatcher

I like to say that Python is a good subset of the Lisp programming language, and as time passes, I find that this is more and more true. The PEP 443 proves that point: it describes a way to dispatch generic functions in a similar manner to what the Common Lisp Object System (CLOS) provides.

If you're familiar with Lisp, this won't be news to you. The Lisp object system, which is one of the basic components of Common Lisp, provides a simple, efficient way to define and handle method dispatching. I'll show you how generic methods work in Lisp first.

Creating Generic Methods in Lisp

To begin with, let's define a few very simple classes, without any parent classes or attributes, in Lisp:

```
(defclass snare-drum ()
  ())

(defclass cymbal ()
  ())

(defclass stick ()
  ())
```

```
(defclass brushes ()
  ())
```

This defines the classes `snare-drum`, `cymbal`, `stick`, and `brushes` without any parent class or attributes. These classes compose a drum kit, and we can combine them to play sound. For this, we define a `play()` method that takes two arguments and returns a sound as a string:

```
(defgeneric play (instrument accessory)
  (:documentation "Play sound with instrument and accessory."))
```

This only defines a generic method that isn't attached to any class and so cannot yet be called. At this stage, we've only informed the object system that the method is generic and might be called with two arguments named `instrument` and `accessory`. In Listing 13-2, we'll implement versions of this method that simulate playing our snare drum.

```
(defmethod play ((instrument snare-drum) (accessory stick))
  "POC!")

(defmethod play ((instrument snare-drum) (accessory brushes))
  "SHHHH!")

(defmethod play ((instrument cymbal) (accessory brushes))
  "FRCCCHHT!")
```

Listing 13-2: Defining generic methods in Lisp, independent of classes

Now we've defined concrete methods in code. Each method takes two arguments: `instrument`, which is an instance of `snare-drum` or `cymbal`, and `accessory`, which is an instance of `stick` or `brushes`.

At this stage, you should see the first major difference between this system and the Python (or similar) object systems: the method isn't tied to any particular class. The methods are *generic*, and they can be implemented for any class.

Let's try it. We can call our `play()` method with some objects:

```
* (play (make-instance 'snare-drum) (make-instance 'stick))
"POC!"

* (play (make-instance 'snare-drum) (make-instance 'brushes))
"SHHHH!"
```

As you can see, which function is called depends on the class of the arguments—the object system *dispatches* the function calls to the right function for us, based on the type of the arguments we pass. If we call `play()` with an object whose classes do not have a method defined, an error will be thrown.

In Listing 13-3, the `play()` method is called with a `cymbal` and a `stick` instance; however, the `play()` method has never been defined for those arguments, so it raises an error.

```
* (play (make-instance 'cymbal) (make-instance 'stick))
debugger invoked on a SIMPLE-ERROR in thread
#<THREAD "main thread" RUNNING {1002ADAF23}>:
  There is no applicable method for the generic function
    #<STANDARD-GENERIC-FUNCTION PLAY (2)>
  when called with arguments
    (#<CYMBAL {1002B801D3}> #<STICK {1002B82763}>).

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly abbreviated name):
  0: [RETRY] Retry calling the generic function.
  1: [ABORT] Exit debugger, returning to top level.

((:METHOD NO-APPLICABLE-METHOD (T)) #<STANDARD-GENERIC-FUNCTION PLAY (2)>
#<CYMBAL {1002B801D3}> #<STICK {1002B82763}>) [fast-method]
```

Listing 13-3: Calling a method with an unavailable signature

CLOS provides even more features, such as method inheritance or object-based dispatching, rather than using classes. If you're really curious about the many features CLOS provides, I suggest reading "A Brief Guide to CLOS" by Jeff Dalton (*http://www.aiai.ed.ac.uk/~jeff/clos-guide.html*) as a starting point.

Generic Methods with Python

Python implements a simpler version of this workflow with the `singledispatch()` function, which has been distributed as part of the `functools` module since Python 3.4. In versions 2.6 to 3.3, the `singledispatch()` function is provided through the Python Package Index; for those eager to try it out, just run `pip install singledispatch`.

Listing 13-4 shows a rough equivalent of the Lisp program we built in Listing 13-2.

```
import functools

class SnareDrum(object): pass
class Cymbal(object): pass
class Stick(object): pass
class Brushes(object): pass

@functools.singledispatch
def play(instrument, accessory):
    raise NotImplementedError("Cannot play these")

❶ @play.register(SnareDrum)
def _(instrument, accessory):
    if isinstance(accessory, Stick):
        return "POC!"
    if isinstance(accessory, Brushes):
        return "SHHHH!"
    raise NotImplementedError("Cannot play these")

@play.register(Cymbal)
def _(instrument, accessory):
    if isinstance(accessory, Brushes):
        return "FRCCCHHT!"
    raise NotImplementedError("Cannot play these")
```

Listing 13-4: Using `singledispatch` to dispatch method calls

This listing defines our four classes and a base `play()` function that raises `NotImplementedError`, indicating that by default we don't know what to do.

We then write a specialized version of the `play()` function for a specific instrument, the `SnareDrum` ❶. This function checks which accessory type has been passed and returns the appropriate sound or raises `NotImplementedError` again if the accessory isn't recognized.

If we run the program, it works as follows:

```
>>> play(SnareDrum(), Stick())
'POC!'
>>> play(SnareDrum(), Brushes())
'SHHHH!'
>>> play(Cymbal(), Stick())
Traceback (most recent call last):
NotImplementedError: Cannot play these
>>> play(SnareDrum(), Cymbal())
NotImplementedError: Cannot play these
```

The `singledispatch` module checks the class of the first argument passed and calls the appropriate version of the `play()` function. For the `object` class, the first defined version of the function is always the one that is run. Therefore, if our instrument is an instance of a class that we did not register, this base function will be called.

As we saw in the Lisp version of the code, CLOS provides a multiple dispatcher that can dispatch based on the type of *any of the arguments* defined in the method prototype, not just the first one. The Python dispatcher is named `singledispatch` for a good reason: it only knows how to dispatch based on the first argument.

In addition, `singledispatch` offers no way to call the parent function directly. There is no equivalent of the Python `super()` function; you'll have to use various tricks to bypass this limitation.

While Python is improving its object system and dispatch mechanism, it still lacks a lot of the more advanced features that something like CLOS provides out of the box. That makes encountering `singledispatch` in the wild pretty rare. It's still interesting to know it exists, as you may end up implementing such a mechanism yourself at some point.

Context Managers

The `with` statement introduced in Python 2.6 is likely to remind old-time Lispers of the various `with-*` macros that are often used in that language. Python provides a similar-looking mechanism with the use of objects that implement the *context management protocol*.

If you've never used the context management protocol, here's how it works. The code block contained inside the `with` statement is surrounded by two function calls. The object being used in the `with` statement determines the two calls. Those objects are said to implement the context management protocol.

Objects like those returned by `open()` support this protocol; that's why you can write code along these lines:

```
with open("myfile", "r") as f:
   line = f.readline()
```

The object returned by `open()` has two methods: one called `__enter__` and one called `__exit__`. These methods are called at the start of the `with` block and at the end of it, respectively.

A simple implementation of a context object is shown in Listing 13-5.

```
class MyContext(object):
    def __enter__(self):
        pass

    def __exit__(self, exc_type, exc_value, traceback):
        pass
```

Listing 13-5: A simple implementation of a context object

This implementation does not do anything, but it is valid and shows the signature of the methods that need to be defined to provide a class following the context protocol.

The context management protocol might be appropriate to use when you identify the following pattern in your code, where it is expected that a call to method `B` must *always* be done after a call to `A`:

1. Call method `A`.
2. Execute some code.
3. Call method `B`.

The `open()` function illustrates this pattern well: the constructor that opens the file and allocates a file descriptor internally is method `A`. The `close()` method that releases the file descriptor corresponds to method `B`. Obviously, the `close()` function is always meant to be called *after* you instantiate the file object.

It can be tedious to implement this protocol manually, so the `contextlib` standard library provides the `contextmanager` decorator to make implementation easier. The `contextmanager` decorator should be used on a generator function. The `__enter__` and `__exit__` methods will be

dynamically implemented for you based on the code that wraps the `yield` statement of the generator.

In Listing 13-6, `MyContext` is defined as a context manager.

```
import contextlib

@contextlib.contextmanager
def MyContext():
    print("do something first")
    yield
    print("do something else")


with MyContext():
    print("hello world")
```

Listing 13-6: Using `contextlib.contextmanager`

The code before the `yield` statement will be executed before the `with` statement body is run; the code after the `yield` statement will be executed once the body of the `with` statement is over. When run, this program outputs the following:

```
do something first
hello world
do something else
```

There are a couple of things to handle here though. First, it's possible to yield something inside our generator that can be used as part of the `with` block.

Listing 13-7 shows how to yield a value to the caller. The keyword `as` is used to store this value in a variable.

```
import contextlib

@contextlib.contextmanager
def MyContext():
    print("do something first")
    yield 42
    print("do something else")


with MyContext() as value:
    print(value)
```

Listing 13-7: Defining a context manager yielding a value

Listing 13-7 shows how to yield a value to the caller. The keyword `as` is used to store this value in a variable. When executed, the code outputs the following:

```
do something first
42
do something else
```

When using a context manager, you might need to handle exceptions that can be raised within the `with` code block. This can be done by surrounding the `yield` statement with a `try...except` block, as shown in Listing 13-8.

```
import contextlib

@contextlib.contextmanager
def MyContext():
    print("do something first")
    try:
        yield 42
    finally:
        print("do something else")


with MyContext() as value:
    print("about to raise")
❶   raise ValueError("let's try it")
    print(value)
```

Listing 13-8: Handling exceptions in a context manager

Here, a `ValueError` is raised at the beginning of the `with` code block ❶; Python will propagate this error back to the context manager, and the `yield` statement will appear to raise the exception itself. We enclose the `yield` statement in `try` and `finally` to make sure the final `print()` is run.

When executed, Listing 13-8 outputs the following:

```
do something first
about to raise
do something else
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ValueError: let's try it
```

As you can see, the error is raised back to the context manager, and the program resumes and finishes execution because it ignored the exception using a `try...finally` block.

In some contexts, it can be useful to use several context managers at the same time, for example, when opening two files at the same time to copy their content, as shown in Listing 13-9.

```
with open("file1", "r") as source:
    with open("file2", "w") as destination:
        destination.write(source.read())
```

Listing 13-9: Opening two files at the same time to copy content

That being said, since the `with` statement supports multiple arguments, it's actually more efficient to write a version using a single `with`, as shown in Listing 13-10.

```
with open("file1", "r") as source, open("file2", "w") as destination:
    destination.write(source.read())
```

Listing 13-10: Opening two files at the same time using only one `with` statement

Context managers are extremely powerful design patterns that help to ensure your code flow is always correct, no matter what exception might occur. They can help to provide a consistent and clean programming interface in many situations in which code should be wrapped by other code and `contextlib.contextmanager`.

Less Boilerplate with attr

Writing Python classes can be cumbersome. You'll often find yourself repeating just a few patterns because there are no other options. One of the most common examples, as illustrated in Listing 13-11, is when initializing an object with a few attributes passed to the constructor.

```
class Car(object):
    def __init__(self, color, speed=0):
        self.color = color
        self.speed = speed
```

Listing 13-11: Common class initialization boilerplate

The process is always the same: you copy the value of the argument passed to the `__init__` function to a few attributes stored in the object. Sometimes you'll also have to check the value that is passed, compute a default, and so on.

Obviously, you also want your object to be represented correctly if printed, so you'll have to implement a `__repr__` method. There's a chance some of your classes are simple enough to be converted to dictionaries for serialization. Things become even more complicated when talking about comparison and hashability (the ability to use `hash` on an object and store it in a `set`).

In reality, most Python programmers do none of this, because the burden of writing all those checks and methods is too heavy, especially when you're not always sure you'll need them. For example, you might find that `__repr__` is useful in your program only that one time you're trying to debug or trace it and decide to print objects in the standard output—and no other times.

The `attr` library aims for a straightforward solution by providing a generic boilerplate for all your classes and generating much of the code for you. You can install `attr` using `pip` with the command `pip install attr`. Get ready to enjoy!

Once installed, the `attr.s` decorator is your entry point into the wonderful world of `attr`. Use it above a class declaration and then use the function `attr.ib()` to declare attributes in your classes. Listing 13-12 shows a way to rewrite Listing 13-11 using `attr`.

```
import attr

@attr.s
class Car(object):
    color = attr.ib()
    speed = attr.ib(default=0)
```

Listing 13-12: Using `attr.ib()` to declare attributes

When declared this way, the class automatically gains a few useful methods for free, such as `__repr__`, which is called to represent objects when they are printed on `stdout` in the Python interpreter:

```
>>> Car("blue")
Car(color='blue', speed=0)
```

This output is cleaner than the default that `__repr__` would have printed:

```
<__main__.Car object at 0x104ba4cf8>.
```

You can also add more validation on your attributes by using the `validator` and `converter` keyword arguments.

Listing 13-13 shows how the `attr.ib()` function can be used to declare an attribute with some constraints.

```
import attr

@attr.s
class Car(object):
    color = attr.ib(converter=str)
    speed = attr.ib(default=0)

    @speed.validator
    def speed_validator(self, attribute, value):
        if value < 0:
            raise ValueError("Value cannot be negative")
```

Listing 13-13: Using `attr.ib()` with its `converter` argument

The `converter` argument manages the conversion of whatever is passed to the constructor. The `validator()` function can be passed as an argument to `attr.ib()` or used as a decorator, as shown in Listing 13-13.

The `attr` module provides a few validators of its own (for example, `attr.validators.instance_of()` to check the type of the attribute), so be sure to check them out before wasting your time building your own.

The `attr` module also provides tweaks to make your object hashable so it can be used in a set or a dictionary key: just pass `frozen=True` to `attr.s()` to make the class instances immutable.

Listing 13-14 shows how using the `frozen` parameter changes the behavior of the class.

```
>>> import attr
>>> @attr.s(frozen=True)
... class Car(object):
...     color = attr.ib()
```

```
...
>>> {Car("blue"), Car("blue"), Car("red")}
{Car(color='red'), Car(color='blue')}
>>> Car("blue").color = "red"
attr.exceptions.FrozenInstanceError
```

Listing 13-14: Using `frozen=True`

Listing 13-14 shows how using the `frozen` parameter changes the behavior of the `Car` class: it can be hashed and therefore stored in a set, but objects cannot be modified anymore.

In summary, `attr` provides the implementation for a ton of useful methods, thereby saving you from writing them yourself. I highly recommend leveraging `attr` for its efficiency when building your classes and modeling your software.

Summary

Congratulations! You made it to the end of the book. You've just upped your Python game and have a better idea of how to write efficient and productive Python code. I hope you enjoyed reading this book as much as I enjoyed writing it.

Python is a wonderful language and can be used in many different fields, and there are many more areas of Python that we did not touch on in this book. But every book needs an ending, right?

I highly recommend profiting from open source projects by reading the available source code out there and contributing to it. Having your code reviewed and discussed by other developers is often a great way to learn.

Happy hacking!

INDEX

Symbols & Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z

Serious Python is set in New Baskerville, Futura, Dogma, and The Sans Mono Condensed.

UPDATES

Visit *https://nostarch.com/seriouspython/* for updates, errata, and other information.

More no-nonsense books from **NO STARCH PRESS**

ELOQUENT JAVASCRIPT, 3RD EDITION
A Modern Introduction to Programming
by MARIJN HAVERBEKE
DECEMBER 2018, 472 PP., $39.95
ISBN 978-1-59327-950-9

PRACTICAL SQL

A Beginner's Guide to Storytelling with Data

by ANTHONY DEBARROS

MAY 2018, 392 PP., $39.95

ISBN 978-1-59327-827-4

PYTHON FLASH CARDS

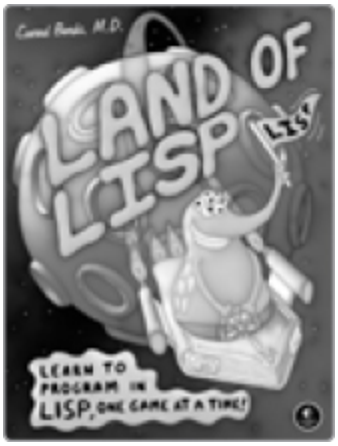
Syntax, Concepts, and Examples

by ERIC MATTHES

JANUARY 2019, 101 CARDS, $27.95

ISBN 978-1-59327-896-0

PYTHON CRASH COURSE
A Hands-On, Project-Based Introduction to Programming
by ERIC MATTHES
NOVEMBER 2015, 560 PP., $39.95
ISBN 978-1-59327-603-4



LAND OF LISP
Learn to Program in Lisp, One Game at a Time!
by CONRAD BARSKI
OCTOBER 2010, 504 PP., $49.95
ISBN 978-1-59327-281-4

THE RUST PROGRAMMING LANGUAGE

by STEVE KLABNIK *and* CAROL NICHOLS

JUNE 2018, 552 PP., $39.95
ISBN 978-1-59327-828-1

PHONE:
1.800.420.7240 OR
1.415.863.9900

EMAIL:
SALES@NOSTARCH.COM

WEB:
WWW.NOSTARCH.COM

The Electronic Frontier Foundation (EFF) is the leading organization defending civil liberties in the digital world. We defend free speech on the Internet, fight illegal surveillance, promote the rights of innovators to develop new digital technologies, and work to ensure that the rights and freedoms we enjoy are enhanced – rather than eroded – as our use of technology grows.

WRITE LESS. CODE MORE. BUILD BETTER PROGRAMS.

COVERS PYTHON 2 AND 3

Sharpen your Python skills as you dive deep into the Python programming language with *Serious Python*. Written for developers and experienced programmers, *Serious Python* brings together more than 15 years of Python experience to teach you how to avoid common mistakes, write code more efficiently, and build better programs in less time. You'll cover a range of advanced topics like multithreading and memoization, get advice from experts on things like designing APIs and dealing with databases, and learn Python internals to give you a deeper understanding of the language itself.

You'll first learn how to start a project and tackle topics like versioning, coding style, and automated checks. Then you'll look at how to define functions efficiently, pick the right data structures and libraries, build

future-proof programs, package your software for distribution, and optimize your programs down to the bytecode. You'll also learn how to:

- Create and use effective decorators and methods, including abstract, static, and class methods
- Employ Python for functional programming using generators, pure functions, and functional functions
- Extend flake8 to work with the abstract syntax tree (AST) to introduce more sophisticated automatic checks
- Apply dynamic performance analysis to identify bottlenecks in your code
- Work with relational databases and effectively manage and stream data with PostgreSQL

Take your Python skills from good to great. Learn from the experts and get seriously good at Python with *Serious Python*!

ABOUT THE AUTHOR

Julien Danjou is a principal software engineer at Red Hat and a contributor to OpenStack, the largest existing open source project written in Python. He has been a free software and open source hacker for the past 15 years.

THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com

FOOTNOTES

1 STARTING YOUR PROJECT

1. Contributors to this project are always welcome. Feel free to jump on IRC and get involved at *irc://chat.freenode.net/openstack-state-management*.

10 PERFORMANCES AND OPTIMIZATIONS

1. Donald Knuth, "Structured Programming with `go to` Statements," *ACM Computing Surveys* 6, no. 4 (1974): 261–301.