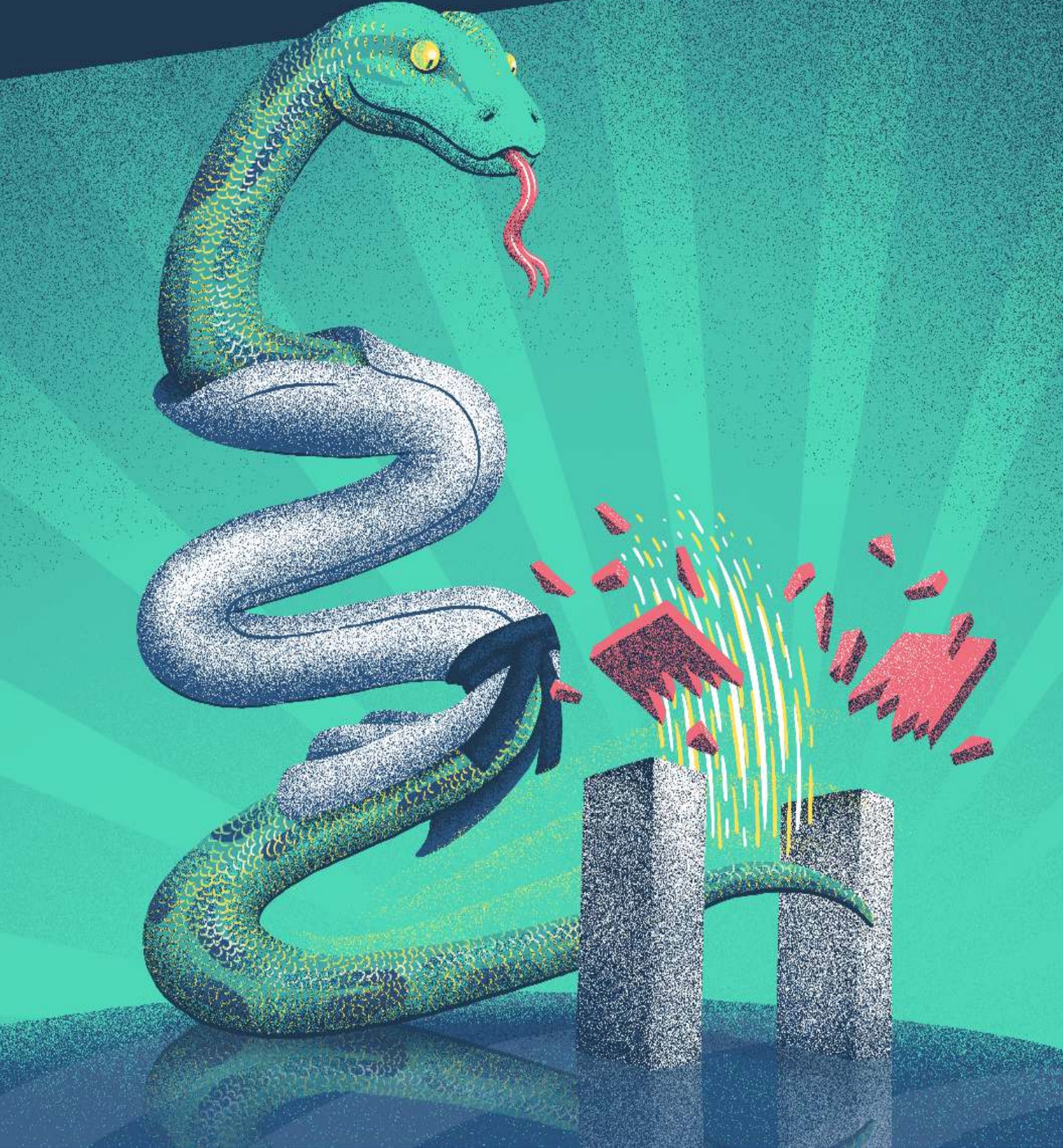
Julien Danjou

Serious Python

Black Belt Advice



Acknowledgements

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

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. As this is the fourth edition of the original book I wrote, I would not have made it without people 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 them a lot of what we teach in this book: Doug Hellmann for its great advice about building libraries, Joshua Harlow for his good mood and knowledge about distributed systems, Christophe de Vienne for his experience around building a framework, Victor Stinner for his incredible knowledge about CPython, Dimitri Fontaine for his wisdom on databases, Robert Collins for messing up with testing, Nick Coghlan for his work in getting Python in a 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 Michael Driscoll for his technical hindsight.

My gratitude also goes to the Free Software communities who help me grow and gave me the willingness to share my knowledge back, especially to the Python community which always has been welcoming and enthusiastic.

Contents

List of Figures

List of Examples

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 will provide 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.

Chapter 2 will look at Python modules, libraries, and frameworks and talk 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 will help 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 will show 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.

In **Chapter 12**, we'll tackle 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, in **Chapter 13**, you'll find 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.

CHAPTER 1

Starting Your Project

In this first chapter, we'll look at a few aspects of starting out 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.

1.1 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, around October 2019 (15 months later). 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.

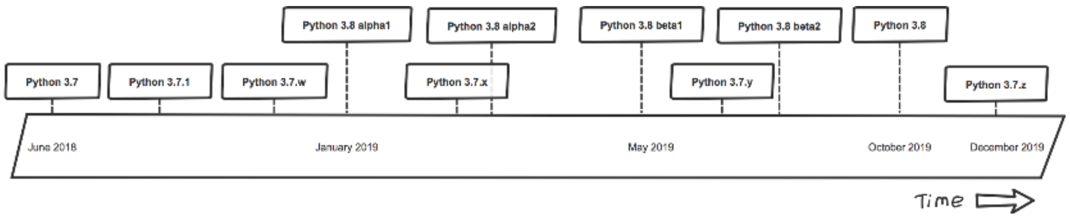


Figure 1.1: Python release timeline

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.

- Versions 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. Most recent operating systems ship at least 3.6, so in the case where you'd target those, you can make sure your application also work with 3.7.

Techniques for writing programs that support both Python 2.7 and 3.x will be discussed in Section 13.1.

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

1.2 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.

1.2.1 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.

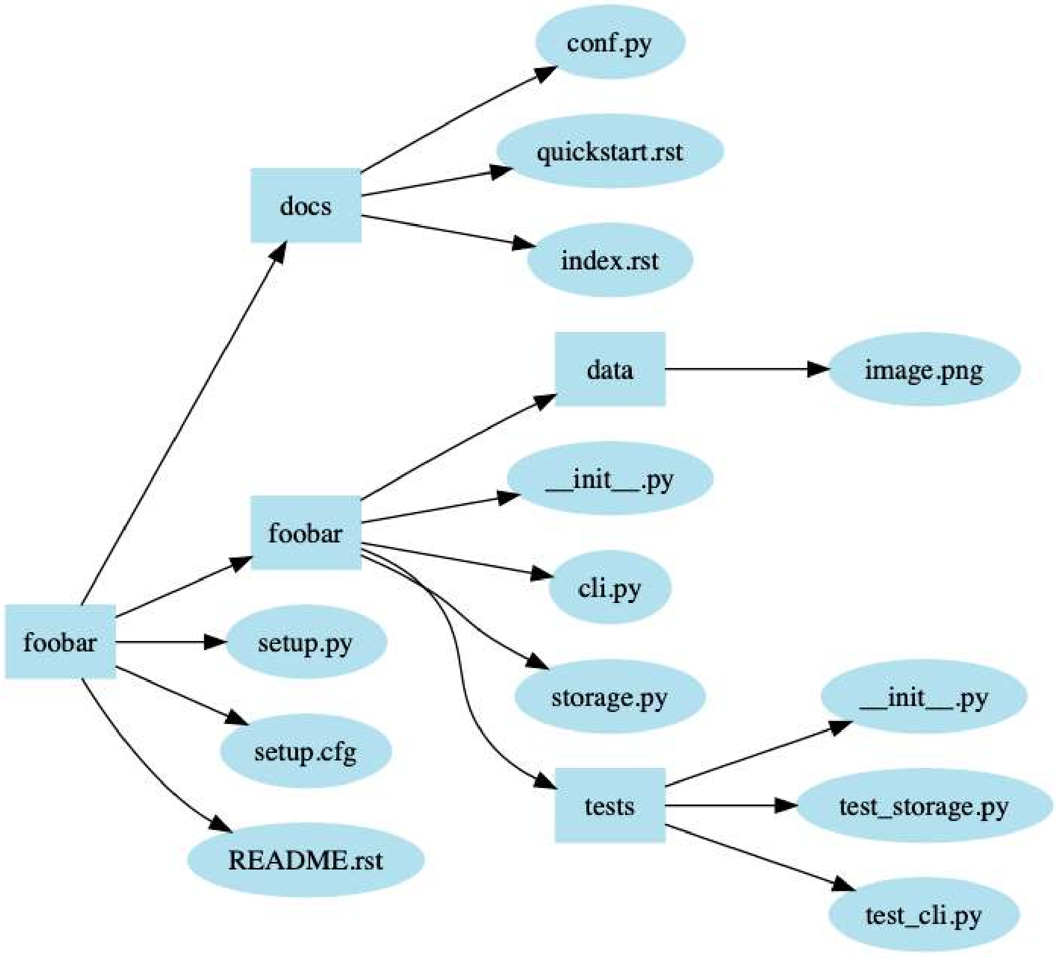


Figure 1.2: Standard package directory

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 Section 3.2).

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`

1.2.2 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 code base, 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 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.

1.3 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. While the `rc` and `c` suffixes have the same meaning, if both are used, `rc` releases are considered to be newer than `c` releases.

The following suffixes can also be used:

- `.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 `.postN` when releasing a bug-fix version; instead but should increment the minor version number.
- `.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.

1.4 Coding Style & 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.

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.
- 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, 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 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:

Example 1.1 A *pep8* run

```
$ 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` documentation.

1.4.1 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" in Section 6.3 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:

Example 1.2 Running `pep8` with `--ignore`

```
$ 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.

1.4.2 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: extendable via plugins.
- pylint: Checks PEP 8 conformance while performing code error checks by default; can be extended via plugin.

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 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 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.

Tip
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 Section 9.3 with our own plugin to verify correct method declaration.

1.5 Interview with Joshua Harlow

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 the 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 (RIT), 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 it 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.
- 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) 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 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!

”

CHAPTER 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 have 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.

2.1 The Import System

To use modules and libraries in your programs, you have to import them using the `imp ort` keyword. As an example, Example 2.1 imports the all-important Zen of Python guidelines:.

Example 2.1 The Zen of Python

```
>>> 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!
```

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 the workings of the sys module, how to change or add import paths, and using 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 those 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/…/>
```

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 module, once imported, are essentially object whose attributes (classes, functions, variables, etc) are objects.

2.1.1 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.

2.1.2 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 non-standard 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 and not the Python module.

2.1.3 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 Section 9.4.

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 can allow 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 would do 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`.

2.1.4 Meta Path Finders

The meta path finder is an object that will allow you to load custom objects as well as the regular and standard `.py` files that Python knows how to load. 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, Example 2.2 shows how Hy uses a custom meta path finder to enable Python to import source files ending with `.hy` instead of `.py`.

Example 2.2 *Hy* module importer

```
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())
```

Once Python has determined that the path is valid and that it points to a module, a `MetaLoader` object is returned as shown in Example 2.3.

Example 2.3 *Hy* module loader

```python
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
        mod = import_file_to_module(fullname,
                                    self.path) ❶

        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
```

1 `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 actually: 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.

2.2 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, My advice is to 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 in later sections, 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 to call when your program exits.

- **argparse** provides functions for parsing command line arguments.
- **bisect** provides bisection algorithms for sorting lists (see Section 10.4).
- **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. It also contains **StringIO** 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 Section 8.4).
- **os** provides access to basic OS functions.

- **random** provides functions for generating pseudo-random numbers.
- **re** provides regular expression functionality.
- **sched** provides an event scheduler without using multi-threading.
- **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 UUIDs (Universally Unique Identifiers).

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.

2.3 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 multi-gigabyte download, especially if they just wanted to write a quick script for renaming files. So, even with all 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 that 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 towards 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. OpenStack also had to adopt `sqlalchemy-migrate` in the meantime so it would be able to fix some of its bugs. That means even more work for the OpenStack community.

2.3.1 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. The following checklist helps when choosing whether to use an external library.

- **Python 3 compatibility** – There are still libraries that only supports only Python 2, and that's probably not a good sign of health nor a good option if you write your code using Python 3.
- **Active development** – GitHub and Ohloh usually provide enough information to determine whether a given library is still being worked on by its maintainers.
- **Active maintenance** – Even if a library is "finished" (i.e. 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 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.

2.3.2 Protecting Your Code with an API Wrapper

No matter what libraries you end up using, you need to treat them as you would any other tools: like useful devices that could potentially do some serious damage. Therefore, 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, you need to 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 Section 5.5.3, 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.

2.4 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 Section 5.3).

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%2Fpackages%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. PyPI is the *Python Package Index* where anyone can upload a package for distribution and installation by others.

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 that are currently installed by using the pip freeze command, like so:

```
$ pip freeze
Babel==1.3
Jinja2==2.7.1
commando=0.3.4
…
```

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
  /usr/local/lib/python2.7/site-packages/pika_pool-0.1.3.dist-info/ ↩
      RECORD
  /usr/local/lib/python2.7/site-packages/pika_pool-0.1.3.dist-info/ ↩
      WHEEL
  /usr/local/lib/python2.7/site-packages/pika_pool-0.1.3.dist-info/ ↩
      metadata.json
  /usr/local/lib/python2.7/site-packages/pika_pool-0.1.3.dist-info/ ↩
      top_level.txt
  /usr/local/lib/python2.7/site-packages/pika_pool.py
  /usr/local/lib/python2.7/site-packages/pika_pool.pyc
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 version of libraries or when working in a continuous testing system. However, since there is no versioning behind, 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 can confidently treat it as your one-stop shop for all your package management needs.

2.5 Using and Choosing Frameworks

Python has a variety of frameworks available for various kinds of 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, which 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 recommended method for choosing a framework for a Python application is largely the same as the one described in Section 2.3.1 - 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 already 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 it 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 hand-picking 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.

2.6 Doug Hellmann 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 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.

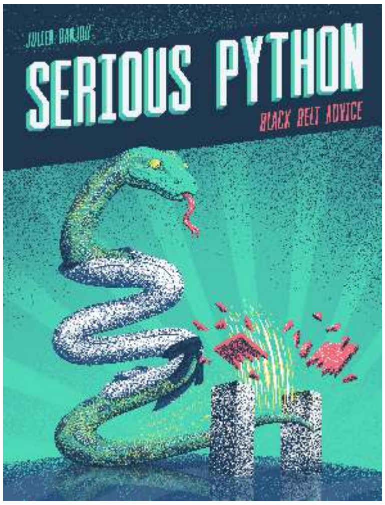
Hey, this was only a sample chapter!

I hope that you did like the sample! It includes the complete table of contents and a full chapter with its examples.

The full version of Serious Python includes:

- 13 chapters
- 8 interviews
- 330 pages
- 100+ code snippets
- Practical examples
- Available in PDF, HTML, EPUB and MOBI formats
- And a few more bonuses such as Docker images!

Buy the Book!



Now that you've read the sample, you might be interested in buying the whole book. It's available online at serious-python.com in different formats and packages. Go check it out!