1 Starting your project

1.1 Python versions

One of the first questions you're likely to ask is "which versions of Python should my software support?". It's well worth asking, since each new version of Python introduces new features and deprecates old ones. Furthermore, there's a **huge** gap between Python 2.x and Python 3.x: there are enough changes between the two branches of the language that it can be hard to keep code compatible with both, as we'll see in more detail later, and it can be hard to tell which version is more appropriate when you're starting a new project. Here are some short answers:

- Versions 2.5 and older are pretty much obsolete by now, so you don't have to worry about supporting them at all. If you're intent on supporting these older versions anyway, be warned that you'll have an even harder time ensuring that your program supports Python 3.x as well. Though you might still run into Python 2.5 on some older systems; if that's the case for you, sorry!
- Version 2.6 is still viable; you'll find it in some older versions of operating systems such as Red Hat Enterprise Linux. It's not hard to support Python 2.6 as well as newer versions, but if you don't think your program will need to run on 2.6, don't stress yourself trying to accommodate it.
- Version 2.7 is and will remain the last version of Python 2.x. It's a good idea to

make it your main target, or one of your main targets, since a lot of software, libraries, and developers still make use of it. Python 2.7 *should* continue to be supported until around 2016, so odds are it's not going away anytime soon.

- Version 3.0, 3.1, and 3.2 were released in quick succession and as such haven't seen much adoption. If your code already supports 2.7, there's not much point in supporting these versions as well.
- Version 3.3 and 3.4 are the most recent distributed editions of Python 3 and the ones you should focus on supporting. Python 3.3 and 3.4 represent the future of the language, so unless you're focusing on compatibility with older versions, you should make sure your code runs on these versions as well.

In summary: support 2.6 only if you have to (or are looking for a challenge), definitely support 2.7, and if you want to guarantee that your software will continue to run for the foreseeable future, support 3.3 and above as well. You can safely ignore other versions, though that's not to say it's impossible to support them all: the CherryPy project supports all versions of Python from 2.3 onward.

Techniques for writing programs that support both Python 2.7 and 3.3 will be discussed in Chapter 13. You might spot some of these techniques in the sample code as you read: all of the code that you'll see in this book has been written to support both major versions.

1.2 Project layout

Your project structure 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.

One common mistake is leaving unit tests outside the package directory. These tests should definitely be included in a sub-package of your software so that:

- they don't get automatically installed as a *tests* top-level module by **setuptools** (or some other packaging library).
- they can be installed and eventually used by other packages to build their own unit tests.

The following diagram illustrates what a standard file hierarchy should look like:

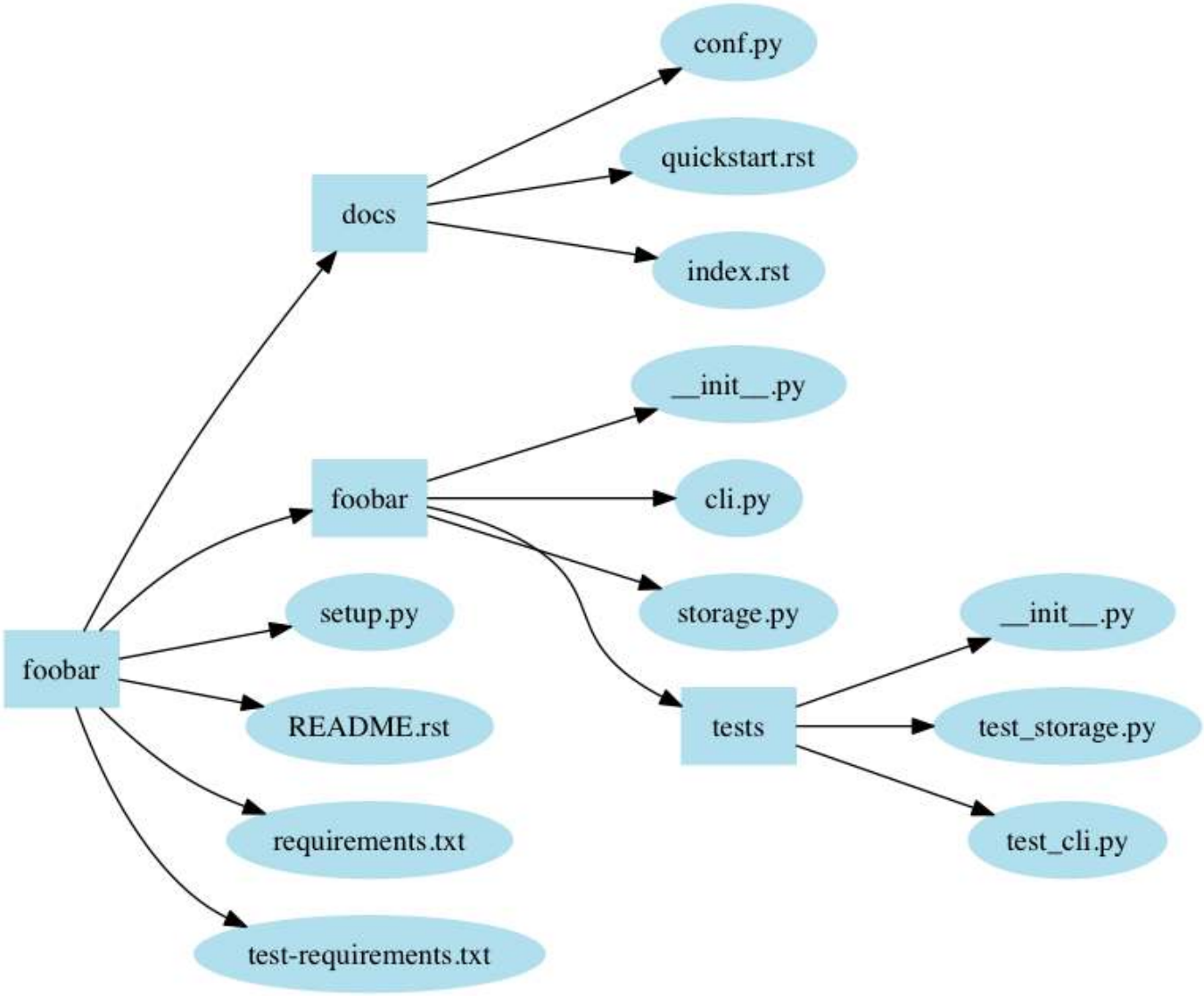

Figure 1.1: Standard package directory

`setup.py` is the standard name for Python installation script. When run, it installs your package using the Python distribution utilities (*distutils*). You can also pro-

vide important information to users in `README.rst` (or `README.txt`, or whatever filename suits your fancy). `requirements.txt` should list your Python package's dependencies – i.e., all of the packages that a tool such as `pip` should install to make your package work. You can also include `test-requirements.txt`, which lists only the dependencies required to run the test suite. Finally, the `docs` directory should contain the package's documentation in *reStructuredText* format, that will be consumed by *Sphinx* (see Section 3.1).

Packages often have to provide extra data, such as images, shell scripts, and so forth. Unfortunately, there's no universally accepted standard for where these files should be stored. Just put them wherever makes the most sense for your project.

The following top-level directories also frequently appear:

Most of the time, the following extra top level directories are used:

- `etc` is for sample configuration files.
- `tools` is for shell scripts or related tools.
- `bin` is for binary scripts you've written that will be installed by `setup.py`.
- `data` is for other kinds of data, such as media files.

A design issue I often encountered is to create files or modules based on the type of code they will store. Having a `functions.py` or `exceptions.py` file is a **terrible** approach. It doesn't help anything at all with code organization and forces a reader to jump between files for no good reason. Organize your code based on features, not type.

Also, don't create a directory and just an `__init__.py` file in it, e.g. don't create `hooks/__init__.py` where `hooks.py` would have been enough. If you create a directory, it should contains several other Python files that belongs to the category/-module the directory represents.

1.3 Version numbering

As you might already know, there's an ongoing effort to standardize package metadata in the Python ecosystem. One such piece of metadata is *version number*.

PEP 440 introduces a version format that every Python package, and ideally every application, should follow. This way, other programs and packages will be able to 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 like *1.2* or *1.2.3*. But note:

- *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* (e.g. *1.2a1*) denotes an **alpha** release, a version that might be unstable and missing features.
- *N[.N]+bN* (e.g. *2.3.1b2*) denotes a **beta** release, a version that might be feature-complete but still buggy.
- *N[.N]+cN* or *N[.N]+rcN* (e.g. *0.4rc1*) denotes a **(release) candidate**, 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.

These suffixes can also be used:

- *.postN* (e.g. 1.4.post2) indicates a **post release**. These are typically used to address minor errors in the publication process (e.g. mistakes in release notes). You shouldn't use *.postN* when releasing a bugfix version; instead, you should increment the minor version number.
- *.devN* (e.g. 2.3.4.dev3) indicates a **developmental release**. This suffix is discouraged because it is harder for humans to parse. It indicates a prerelease of the version that it qualifies: e.g. 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 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.

If you need to handle more advanced version numbers, you should note that PEP 426 defines **source label**, a field that you can use to carry any version string, and then build a version number consistent with PEP requirements.

Many DVCS [1] platforms, such as Git and Mercurial, are able to generate version numbers using an identifying hash [2]. Unfortunately, this system isn't compatible with the scheme defined by PEP 440: for one thing, identifying hashes aren't orderable. However, it's possible to use a source label field to hold such a version number and use it to build a PEP 440-compliant version number.

[1] Distributed Version Control System
[2] For Git, refer to *git-describe(1)*.

Tip

pbr [a], which will be discussed in Section 4.2, is able to automatically build version numbers based on the Git revision of a project.

[a] *Python Build Reasonableness*

1.4 Coding style & automated checks

Yes, coding style is a touchy subject, but we still need to talk about it.

Python has an amazing quality [3] that few other languages have: it uses indentation to define blocks. At first glance, it seems to offer a solution to the age-old question of "where should I put my curly braces?"; unfortunately, it introduces a new question in the process: "how should I indent?"

And so the Python community, in their vast wisdom, came up with the PEP 8 [4] standard for writing Python code. The list of guidelines boils down to:

- Use 4 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, at the top of the file, after comments and docstrings, grouped first by standard, then third-party, and finally local library imports.
- No extraneous whitespaces between parentheses, brackets, or braces, or before commas.

[3] Your mileage may vary.
[4] *PEP 8 Style Guide for Python Code*, 5th July 2001, Guido van Rossum, Barry Warsaw, Nick Coghlan

- Name classes in `CamelCase`; suffix exceptions with `Error` (if applicable); name functions in lowercase with words `separated_by_underscores`; and use a leading underscore for `_private` attributes or methods.

These guidelines really aren't hard to follow, and furthermore, 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. That's what the **pep8** tool is there for: it can automatically check any Python file you send its way.

Example 1.1 A *pep8* run

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

pep8 indicates which lines and columns do not conform to PEP 8 and reports each issue with a code. Violations of *MUST* statements in the specification are reported as **errors** (starting with *E*), while minor problems are reported as **warnings** (starting with *W*). The three-digit code following the letter indicates the exact kind of error or warning; you can tell the general category at a glance by looking at the hundreds digit. 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.

The community still debates whether validating against PEP 8 code that is not part of the standard library is a good practice. I advise you to consider it and run a PEP 8 validation tool against your source code on a regular basis. An easy way to do this is to integrate it into your test suite. While it 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 Section 6.7 how you can integrate *pep8* with *tox* to automate these checks.

The OpenStack project has enforced PEP 8 conformance through automatic checks since the beginning. While it sometimes frustrates newcomers, it ensures that the codebase – which has grown to over 1.67 *million* lines of code – 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 whitespace ordering.

It's also possible to ignore certain kinds of errors and warnings by using the *--ignore* option:

Example 1.2 Running *pep8* with *--ignore*

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

This allows you to effectively ignore parts of the PEP 8 standard that you don't want to follow. If you're running *pep8* on a existing code base, 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.

Other tools also exist that check for actual coding errors rather than style errors. Some notable examples include:

- pyflakes, which supports plugins
- pylint, which also checks PEP 8 conformance, performs more checks by default, and supports 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 – you'll still need to run *pep8* as well. To simplify things, a project called *flake8* combines *pyflakes* and *pep8* into a single command. It also adds some new features such as skipping checks on lines containing `#noqa` and extensibility via entry points.

In its quest for beautiful and unified code, the OpenStack project chose *flake8* for all of its code checks. However, as time passed, the hackers took advantage of *flake8*'s extensibility to test for even more potential issues with submitted code. The end result of all this is a *flake8* extension called *hacking*. It checks for errors such as odd usage of `except`, Python 2/3 portability issues, import style, dangerous string formatting, and possible localization issues.

If you're starting a new project, I strongly recommend you use one of these tools and rely on it for automatic checking of your code quality and style. If you already have a codebase, a good approach is to run them with most of the warnings disabled and fix issues one category at a time.

While none of these tools may be a perfect fit for your project or your preferences, using *flake8* and *hacking* together is a good way to improve the quality of your code and make it more durable. If nothing else, it's a good start toward that goal.

Tip

Many text editors, including the famous GNU Emacs and vim, have plugins available (such as *Flymake*) 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.