JULIEN DANJOU

THE HACKER'S GUIDE TO PYTHON

Contents

List of Figures

List of Examples

About this book

Version 1.0 released in March 2014.

If you're reading this, 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, but whatever the case, you've *hacked* your way into learning it. That's exactly how I got familiar with Python up until I joined the OpenStack team over two years ago.

Before then, I was building my own Python libraries and applications on a "garage project" scale, but things change once you start working with hundreds of developers on software and libraries that thousands of users rely on. The OpenStack platform represents over half a million lines of Python code, all of which needs to be concise, efficient, and scalable to needs of whatever cloud computing application its users require. And when you have a project 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 when I first joined OpenStack, but I've learned a lot more these past two years working on projects the scale of which I could barely even imagine when I got started. I've also had the opportunity to meet some of the best Python hackers in the industry and learn from them – 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!

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:

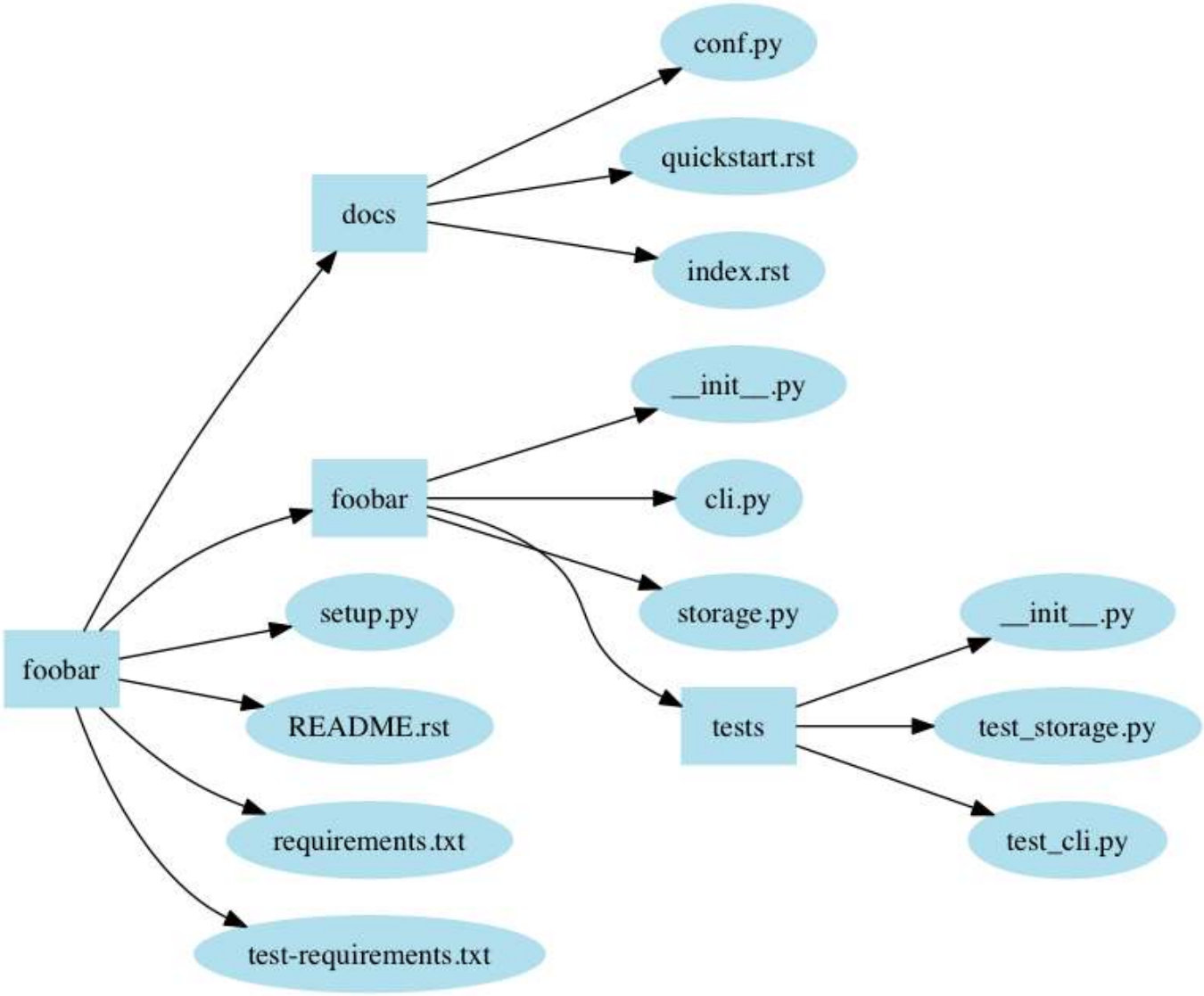


Figure 1.1: Standard package directory

`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.

2 Modules and libraries

2.1 The import system

In order to use modules and libraries, you have to import them.

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, but you probably already know the basics. Here, I'll show you some of the internals of this subsystem.

The `sys` module contains a lot of information about Python's *import* system. First of all, the list of modules currently imported is available through the `sys.modules` variable. It's a dictionary where the key is the module name and the value is the module object.

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

Some modules are built-in; these are listed in `sys.builtin_module_names`. Built-in modules can vary depending on the compilation options passed to the Python build system.

When importing modules, Python relies on a list of paths. This list is stored in the `sys.path` variable and tells Python where to look for modules to load. You can change this list in code, adding or removing paths as necessary, or you can modify the `PYTHONPATH` environment variable to add paths without writing Python code at all. The following approaches are almost equivalent [1]:

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

```
$ PYTHONPATH=/foo/bar python
>>> import sys
```

[1] *Almost* because the path will not be placed at the same level in the list, though it may not matter depending on your use case.

```
>>> '/foo/bar' in sys.path
True
```

The order in `sys.path` is important, since the list will be iterated over to find the requested module.

It is also possible to extend the import mechanism using custom importers. This is the technique that *Hy* [2] uses to teach Python how to import files other than standard `.py` or `.pyc` files.

The import hook mechanism, as it is called, is defined by PEP 302 [3]. It allows you to extend the standard import mechanism and apply preprocessing to it. You can also add a custom module finder by appending a factory class to `sys.path_hooks`.

The module finder object must have a `find_module(fullname, path=None)` method that returns a loader object. The load object also must have a `load_module(fullname)` responsible for loading the module from a source file.

To illustrate, here's how *Hy* uses a custom importer to import source files ending with `.hy` instead of `.py`:

Example 2.1 *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):
```

[2] *Hy* is a Lisp implementation on top of Python, discussed in Section 9.1
[3] *New Import Hooks*, implemented since Python 2.3

```
                    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 the path is determined to both be valid and point to a module, a `MetaLoader` object is returned:

Hy module loader

```
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
```

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

The uprefix module is another good example of this feature in action. Python 3.0 through 3.2 didn't have the `u` prefix for denoting Unicode strings featured in Python 2 [4]; this module ensures compatibility between 2.x and 3.x by removing the `u` prefix from strings before compilation.

[4] It was added back in Python 3.3.

2.2 Standard libraries

Python comes with a huge standard library packed with tools and features for 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 such functionality built in and ready for use.

Whenever you're about to write your own function to handle a simple task, please **stop** and look through the standard library first. My advice is to skim through the whole thing at least once so that next time you need a function, you'll already know whether what you need already exists in the standard library.

We'll talk about some of these modules in later sections, such as **functools** and **itertools**, but here's a few of the standard modules that you should definitely know about:

- **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 10.3).
- **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.

- **glob** provides functions for matching Unix-style path patterns.
- **io** provides functions for handling I/O streams. In Python 3, it also contains **StringIO** (which is in 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 Section 8.3).
- **os** provides access to basic OS functions.
- **random** provides functions for generating pseudo-random numbers.
- **re** provides regular expression functionality.
- **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 to help you keep track of which library modules do what. If you can memorize even part of it, 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.

Tip

The entire standard library is written in Python, so there's nothing stopping you from looking at the source code of its 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

Have you ever unwrapped an awesome birthday gift or Christmas present only to find out that whoever gave it to you forgot to buy batteries for it? Python's "batteries included" philosophy is all about keeping that from happening to you as a programmer: the idea is that, once you have Python installed, you have everything you need to make anything you want.

Unfortunately, there's no way the people behind Python can predict *everything* you might want to make. And even if they could, most people won't want to deal with a multi-gigabyte download when all they want to do is write a quick script for renaming files. The bottom line is, even with all its extensive functionality, there are some things the Python Standard Library just doesn't cover. But that doesn't mean that there are things you simply can't do with Python – it just means that there are things you'll have to do using 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 be sure it won't break messily when you try to use it – and in the unlikely event that it *does*,

you can be sure 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 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 we also made use of sqlalchemy-migrate to handle our schema migration needs. And it worked…until it didn't. Bugs started piling up, and nothing was getting done about them. Furthermore, OpenStack was getting interested in supporting Python 3 at the time, but there was no sign that sqlalchemy-migrate was going to support it as well. It was clear by that point that sqlalchemy-migrate was effectively dead and we needed to switch to something else. At the time of this writing, OpenStack projects are migrating towards using Alembic instead; not without some effort, but fortunately without much pain.

All of this builds up to one important question: "how can I be sure I won't fall into this same 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 like that in a few months. However, here at OpenStack, we use the following checklist to help tip the odds in our favor (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 still being worked on by its maintainers.
- Active maintenance. Even if a library is "finished" (i.e. feature-complete), the

maintainers should still be working on 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: it'll 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.

Applying this checklist to dependencies is also a good idea, though it might be a huge undertaking. If you know your application is going to depend heavily on a particular library, you should at least apply this checklist to each of that library's dependencies.

No matter what libraries you end up using, you need to treat them like you would any other tools: as useful devices that could potentially do some serious damage. It won't always be the case, but ask yourself: if you had a hammer, would you carry it through your entire house, possibly breaking your stuff by accident as you went along? Or would you keep it in your tool shed or garage, away from your fragile valuables and right where you actually need it?

It's the same thing with external libraries: no matter how useful they are, you need to be wary of letting them get their 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. Need to use a different library? All you have to change is your wrapper: as long as it still provides the same functionality, you won't have to touch your codebase at all. There might be exceptions, but there shouldn't be many: most libraries are designed to solve a tightly focused range of problems and can therefore be easily isolated.

Later, in Section 4.7.3, 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 that can be switched out at will.

2.4 Frameworks

There are various Python 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 make use of 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 is going to build 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 earlier for external libraries - which only makes sense, as frameworks are distributed as bundles of Python libraries. Sometimes they 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.

Just 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 want to update it to run on Python 3, you're out of luck unless either you rewrite your entire program to use a different framework or someone finally gets around to upgrading it with full Python 3 support.

Some frameworks are lighter than others. For one comparison, 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 your 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.

2.5 Interview with Doug Hellmann

I've had the chance to work with Doug Hellmann these past few months. He's a senior developer at DreamHost and a fellow contributor to the OpenStack project. He launched the website Python Module of the Week a while back, and he's also 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? Is it different from hacking an existing application?

The steps are similar in the abstract, but the details change. There tend to be more differences between my approach to working on applications and libraries than there are for new versus existing projects.

When I want to change existing code, especially when it has been created by someone else, I start by digging in to figure out how it works and where my change would need to go. I may add logging or print statements, or use *pdb*, and run the app with test data to make sure I understand what it is 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. I create some code and run it by hand, then write tests to make sure I've covered all of the edge cases after I have the basic aspect of a feature working. Creating the tests may also lead to some refactoring to make the code easier to work with.

That was definitely the case with *smiley*. I started by experimenting with Python's trace API using some throw-away scripts, before building the real application. My original vision for smiley included one piece to instrument and collect data from another running application, and a second piece to collect the data sent over the network and save it. In the course of adding a couple of different reporting features, I realized that the processing for replaying the data that had been collected 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. Thinking about how to write programs with the new library can be made easier by writing the tests first, instead of after 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 have also found that writing the documentation for a library before writing any code at all gives me a way to think through the features and workflows for using it without committing to the implementation details. It also 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. That was the approach I took with *stevedore*.

I knew I wanted *stevedore* to provide a set of classes for managing plugins for applications. During the design phase, I spent some time thinking about common patterns I had seen for consuming plugins and wrote a few pages of rough documentation describing how the classes would be used. I realized that if I put most of the complex arguments into the class constructors, the `map()` methods could be almost interchangeable. Those design notes fed directly into the introduction for stevedore's official documentation, explaining the various patterns and guidelines for using plugins in an application.

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

The full process and guidelines can be found in the Python Developer's Guide.

Before a module can be added to the Python Standard Library, it needs to be proven to be stable and widely useful. The module should provide something that is either hard to implement correctly or so useful that many developers have created their own variations. The API should be clear and the implementation should not have dependencies on modules outside the Standard Library.

The first step to proposing a new module is bringing it up within 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 includes the motivation for adding the module and some implementation details of how the transition will happen.

Because package management and discovery tools have become so reliable, especially *pip* and the Python Package Index (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 bugfixes, 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 and would start using?

I've been doing a lot of work with dynamically loaded extensions for applications recently. I use the **abc** module to define the APIs for those 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 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 feature that is widely useful and 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 just need to hold data but don't have 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 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, or external libraries, roll their own abstractions on top of the Standard Library. I'm particularly thinking about things like date/time handling, for example. What would be your advice on that? 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 example you raise, the **timeutils** module in OpenStack, is a fairly thin wrapper around Python's **datetime** module. Most of the functions are short and simple, but by creating a module with the most common oper-

ations, we can ensure they are handled consistently throughout all OpenStack 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 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.

Do you have any particular recommendations on what to do when importing and using a lot of modules, from the Standard Library or elsewhere?

I don't have a hard limit, but if I have more than a handful of imports, I reconsider the design of the module and think about splitting it up into a package. The split may happen sooner for a lower level module than for a high-level or application module, since at a higher level I expect to be joining more pieces together.

Regarding Python 3, what are the modules that are worth mentioning and might make developers more interested in looking into it?

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 maintaining support for Python 2.7 is also easier thanks to the compatibility features added to 3.3. 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."

Many developers write all their code into an application, but there are cases where it would be worth the effort to branch their code out into a Python library. In term of design, planning ahead, migration, etc., what are the best ways to do this?

Applications are collections of "glue code" holding libraries together for a specific purpose. Designing based on implementing those features 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. Failing to take this approach means the features of the application are tightly bound to the user interface, which makes them harder to modify and reuse.

What advice would you give to people planning to start 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 from the public API.

SQLAlchemy is an excellent example of applying those guidelines. The declarative ORM, 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 that you encounter while reading random Python developers' code?

A big area where Python's idioms are different from other languages is looping and iteration. For example, one of the most common anti-patterns I see is using a `for` loop to filter one list by 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 that to generator expressions because they 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 also some more subtle things I 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 (e.g., an empty list instead of None); reducing the number of arguments to a function 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.

Do you have a concrete example, something you've either done or witnessed, of picking up a "wrong" dependency?

Recently, I had a case in which a new release of *pyparsing* dropped Python 2 support and caused me a little trouble with a library I maintain. The update to pyparsing was a major revision, and was clearly labeled as such, but because I had not constrained the version of the dependency in the settings for *cliff*, the new release of pyparsing caused issues for some of

cliff's consumers. The solution was to provide different version bounds for Python 2 and Python 3 in the dependency list for *cliff*. This situation highlighted the importance of both understanding dependency management and ensuring proper test configurations for continuous integration testing.

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.

By pulling out the common parts into a framework, you can focus your development efforts on the unique aspects of an application. They also help you bring an application to a useful state more quickly than if you started from scratch by providing a lot of bootstrapping code for doing things like running in development mode and writing a test suite. They also encourage you to be consistent in the way you implement the application, which means you end up with code that is easier to understand and more reusable.

There are some potential pitfalls to watch out for when working with frameworks, 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 different patterns or idioms than it recommends.

2.6 Managing API changes

When building an API, it's rare to get everything right the first try. Your API will have to evolve, adding, removing, or changing the features it provides.

In the following paragraphs, we will discuss how to manage public API changes. Public APIs are the APIs that you expose to users of your library or application; internal APIs are another concern, and since they're internal (i.e. your users will never have to deal with them), you can do whatever you want with them: break them, twist them, or generally abuse them as you see fit.

The two types of API can be easily distinguished from each other. The Python convention is to prefix private API symbols with an underscore: `foo` is public, but `_bar` is private.

When building an API, the worst thing you can do is to break it abruptly. Linus Torvalds is (among other things) famous for having a zero tolerance policy on public API breakage for the Linux kernel. Considering how many people rely on Linux, it's safe to say he made a wise choice.

Unix platforms have a complex management system for libraries, relying on soname[http://en.wikipedia.org/wiki/Soname] and fine-grained version identifiers. Python doesn't provide such a system, nor an equivalent convention. It's up to maintainers to pick the right version numbers and policies. However, you can still take the Unix system as inspiration for how to version your own libraries or applications. Generally, your version numbering should reflect changes in the API that will impact users; most developers use major version increments to denote such changes, but depending on how you number your versions, you can also use minor version increments as well.

Whatever else you decide to do, the first thing and most important step when modifying an API is to heavily document the change. This includes:

- documenting the new interface
- documenting that the old interface is deprecated
- documenting how to migrate to the new interface

You shouldn't remove the old interface right away; in fact, you should try to keep the old interface for as long as possible. New users won't use it since it's explicitly marked as deprecated. You should only remove the old interface when it's too much trouble to keep.

Example 2.2 A documented API change

```
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 here instead
        pass
```

It's a good idea to use Sphinx markup to highlight changes. When building the documentation, it will be clear to users that the function should not be used, and direct

access to the new function will be provided along with an explanation of how to migrate old code. The downside of this approach is that you can't rely on developers to read your changelog or documentation when they upgrade to a newer version of your Python package.

Python provides an interesting module called **`warnings`** that can help in this regard. This module allows your code to issue various kinds of warnings, such as **`PendingDeprecationWarning`** and **`DeprecationWarning`**. These warnings can be used to inform the developer that a function they're calling is either deprecated or going to be deprecated. This way, developers will be able to see that they're using an old interface and should do something about it. [5]

To go back to the previous example, we can make use of this and warn the user:

Example 2.3 A documented API change with warning

```
import warnings

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".
        """
        warnings.warn("turn_left is deprecated, use turn instead",
                      DeprecationWarning)
        self.turn(direction='left')

    def turn(self, direction):
```

[5] For those who work with C, this is a handy counterpart to the `__attribute__ ((deprecated))` GCC extension.

```
        """Turn the car in some direction.

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

Should any code call the deprecated `turn_left` function, a warning will be raised:

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

Note

Since Python 2.7, `DeprecationWarning` are not displayed by default. To disable this filter, you need to call `python` with the `-W all` option. See the `python` manual page for more information on the possible values for `-W`.

Having your code tell developers that their programs are using something that will stop working eventually is a good idea because it can also be automated. When running their test suites, developers can run `python` with the `-W error` option, which transforms warnings into **exceptions**. That means that every time an obsolete function is called, an error will be raised, and it will be easy for developers using your library to know exactly where their code needs to be fixed.

Example 2.4 Running `python -W error`

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

2.7 Interview with Christophe de Vienne

Christophe is a Python developer and the author of WSME, *Web Services Made Easy*. This framework 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 are the mistakes developers often make when designing a Python API?

There are a few mistakes I try not to make when designing a Python API:

- Making it too complicated. As the saying goes, "Keep It Simple." (Some people would say "Keep It Simple Stupid," but I don't think "simple" and "stupid" are compatible.) Complicated APIs are hard to understand and hard to document. You don't have to make the actual library functionality simple as well, but it's a smart idea. A good example is the Requests library: compared to the various standard **urllib** libraries, the *Requests* API is very simple and natural, but it does complex things behind the scenes. The **urllib** API, by contrast, is almost as complicated as the things it does.
- Doing (visible) magic. When your API does things that your documentation doesn't explain, your end users are going to 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 unnatural happening up front.

- Forgetting your use cases. When writing code down in the depths of your library, it's easy to forget how your library will actually be used. Coming up with good use cases makes it easier to design an API.
- Not writing unit tests. TDD is a very efficient way to write libraries, especially in Python. It forces the developer to assume the role of the end user from the very beginning and maintain compatibility between versions. It's also the only approach I know of that allows you to completely rewrite a library. Even if it's not always necessary, it's good to have that option.

Considering the variety of frameworks WSME can sit on top of, what kinds of API does it have to support?

There actually aren't that many, since the frameworks it sits on are similar in a lot of ways. They use decorators to expose functions and methods to the outside world; they're based on the *WSGI* standard (so their request objects look very similar); and they've all more or less used each other as a source of inspiration. That said, we haven't yet attempted to plug it into an asynchronous web framework such as Twisted.

The biggest difference I've had to deal with is the way contextual information is accessed. In a web framework, the context is mainly the request and what can be deduced from or attached to it (identity, session data, data connection, etc.), as well as a few global things like the global configuration, connection pool, and so forth. Most web frameworks assume they're running on a multi-threaded server and treat all this information as TSD (Thread-Specific Data). This allows them to access the current request by simply importing a *request* proxy object from a module and working with it. While it's pretty straightforward to use, it implies a little magic and makes global objects out of context-specific data.

The Pyramid framework doesn't work like this, for example. Instead, the context is explicitly injected into the code pieces that work with it. This is why the views takes a "request" parameter, which wraps the *WSGI* environment and gives access to the global context of the application.

What are their pros and cons?

An API style like the one used in Pyramid has the big advantage that it allows a single program to run several completely distinct environments in a very natural way. The downside is that its learning curve is a little steeper.

How does Python make it easier or harder to design a library API?

The lack of a built-in way to define which parts are public and which parts aren't is both a (slight) problem and an advantage.

It's a problem when it means developers don't think as much as they should about which parts are their API and which parts aren't. 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's your rule of thumb about API evolution, deprecation, removal, etc.?

There are several criteria I weigh when making a decision:

- **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 non-compatible 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.

- **Will maintenance be easier with the change?** 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 much more (or less) consistent will my API be after the change?** If all of the API's functions follow a similar pattern (such as requiring the same parameter in the first position), it's important to make sure that 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 this change?** Last but not least, always consider the users' point of view.

What advice do you have regarding API documentation in Python?

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. Now that we have Read the Docs, 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. Follow the PEP 257[6] guidelines so that 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.

[6] *Docstring Conventions*, David Goodger, Guido van Rossum, 29 May 2001

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. (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.

3 Documentation

As I've already touched upon, documentation is one of the most important parts of writing software. Unfortunately, there are still a lot of projects out there that doesn't provide proper documentation. Writing documentation is seen as a complicated and daunting task, but it doesn't have to be: with the tools that are available to Python programmers, documenting your code can be just as easy as writing it in the first place.

One of the biggest culprits behind why documentation is either sparse or nonexistent is that many people assume that the only way to document code is by hand. Even if you have multiple people working on the same project, this means that one or more of them is going to 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 tell you they'd rather write software than write about software. Sometimes the documentation process is even completely separate from the development process, meaning that the documentation is written by people who have never written so much as a line of the actual code. Furthermore, any documentation produced this way is likely to be out-of-date: whether the documentation is handled by the programmers themselves or by dedicated writers, it's almost impossible for manual documentation to keep up with the pace of development.

The bottom line is, the more degrees of separation there are between your code and your documentation, the harder it will be to keep the latter properly maintained.

So why keep your code and documentation separate at all? It's not only possible to put your documentation directly in your code itself, but it's also easy to convert that documentation into easy-to-read HTML and PDF files.

The *de facto* standard documentation format for Python is *reStructuredText*, or *reST* for short. It's a lightweight markup language (like the famous *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: it can read *reST*-formatted content and output documentation in a variety of other formats.

Your project documentation should include:

- 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 it works.
- Installation instructions.
- Links to community support, mailing list, IRC, forums, etc.
- 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 will be displayed on your GitHub or PyPI project page; both sites know how to handle *reST* formatting.

Tip

If you're using GitHub, you can also add a `CONTRIBUTING.rst` file that will be displayed when someone creates a pull request. It should provide a checklist for them to follow before they submit the request, e.g. follow PEP 8 or don't forget to run the unit tests.

Tip

Read The Docs allows you to build and publish your documentation online automatically. Signing up and configuring a project is a straightforward process: it 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.

3.1 Getting started with Sphinx and reST

First of all, you should run `sphinx-quickstart` in your project's top-level directory. This will create the directory structure *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 will serve as the front page of your documentation.

You can then build your documentation in HTML format by calling `sphinx-build` with your source directory and output directory as arguments:

```
$ 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.
```

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

Tip

If you are using *setuptools* or *pbr* (see Section 4.2) 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.

`index.rst` is where your documentation begins, but it doesn't have to end there: *reST* supports includes, so there's nothing stopping you from dividing your documentation up into multiple files. Don't worry too much about syntax and semantics to start with: it's true that *reST* offers a lot of formatting possibilities, but you'll have plenty of time to dive into the reference later. The complete reference explains how to create titles, bulleted lists, tables, and more.

3.2 Sphinx modules

Sphinx is highly extensible: its basic functionality only supports manual documentation, but it comes with a number of useful modules which enable automatic documentation and other features. For example, `sphinx.ext.autodoc` extracts *reST*-formatted docstrings from your modules and generates `.rst` files for inclusion. `sph`

`inx-quickstart` will ask you if you want to activate this module when you run it – alternately, you can edit your `conf.py` file and add it as an extension:

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

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

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

❶ Request that all documented members be printed (optional)

❷ Request that all undocumented members be printed (optional)

❸ Show inheritance (optional)

Also note:

- If you don't include any directives, Sphinx won't generate any output.
- If you only specify `:members:`, undocumented nodes on your module/class/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 entirely. To keep this from happening, you'd either 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 with this.

autodoc gives you the power to include most of your documentation in your actual 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.

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.autogen` module was created specifically to handle this common use case. First, you need to enable it in `conf.py`:

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

Now you can add something like the following to an `.rst` file to automatically generate a TOC 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.

Tip

In large projects, it can be tedious to add modules to this list by hand. Just remember that `conf.py` is an ordinary Python source file: there's nothing stopping you from writing your own code in it, including code that automatically builds `.rst` files indicating which modules to document.

Another useful feature of *Sphinx* is the ability to run *doctest* on your examples automatically when you build your documentation. *doctest* is a standard Python mod-

ule which searches your documentation for code snippets and runs them to test whether they accurately reflect what your code actually does. Every paragraph starting with >>> (i.e. the primary prompt) is treated as a code snippet to test:

```
To print something to the standard output, use the :py:func:`print`  ↩
   function.

    >>> print("foobar")
    foobar
```

It's easy to end up leaving your examples unchanged as your API evolves; *doctest* helps you 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. You can also use *doctest* for *Documentation-Driven Development (DDD)*: write your documentation and examples first, and then write your code to match your documentation.

Taking advantage of this feature is as simple as running `sphinx-build` with the special `doctest` builder:

```
    $ 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.
```

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

- Link between projects using
- 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 in advance that there are modules that can provide it.

3.3 Extending Sphinx

Sometimes the off-the-shelf solutions just aren't enough. It's one thing if you're writing an API that's going to be used from within Python, but what if you're writing, say, 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 WSME had other ideas. They developed a *Sphinx* extension called *sphinxcontrib-pecanwsme* which analyzes docstrings and actual Python code to generate REST API documentation automatically. You can do the same thing for your own projects: if you can extract information from your code that could be useful in your documentation, it only makes sense to automate the process.

Tip

You can use *sphinxcontrib.httpdomain* for other HTTP frameworks such as Flask, Bottle, and Tornado.

My point here is that whenever you know that you could extract information from your code that could help to build documentation, you should really do that and automatize it. It is better than trying to maintain a manually written documentation, especially when you can leverage it with auto-publication tools like Read The Docs.

To write a *Sphinx* extension, first you need 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* expects this module to have one predefined function called `setup(app)`. The `app` object will contain 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.

For example, *sphinxcontrib-pecanwsme* adds a single directive called `rest-controller` using the `setup(app)` function. This added directive needs a fully qualified WSME controller class name to generate documentation for.

Example 3.1 Code from `sphinxcontrib.pecanwsme.rest.setup`

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

`RESTControllerDirective` is a directive class which has to have certain properties and methods as described in the Sphinx extension API. The main method, `run()`, will do the actual work of extracting documentation from your code.

The *sphinx-contrib* repository has a bunch of small modules that can help you develop your own.

Note

Even though *Sphinx* is written in Python and targets it by default, there are extensions 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.

4 Distribution

It's a safe bet you'll want to distribute your software at some point. As tempted as you might be to just zip up your code and upload it to the Internet, Python provides tools to help you make sure your end users will have no trouble getting your software to work. You should already be familiar with using `setup.py` to install Python applications and libraries, but you've probably never delved into how it actually works behind the scenes, or how to make a `setup.py` of your own.

4.1 A bit of history

distutils has been part of the standard Python library since 1998. It was originally developed by Greg Ward, who sought to create an easy way for developers to automate the installation process for their end users:

Example 4.1 `setup.py` using *distutils*

```
#!/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'])
```

And that's it. All users have to do to `build` or `install` your software is run `setup.py` with the appropriate command. If your distribution includes C modules in addition to native Python ones, it can even handle those automatically as well.

Development on *distutils* was abandoned in 2000; since then, other developers picked up where it left off, building their own tools based on it. One of the most notable successors to *distutils* is the packaging library known as *setuptools*, which offered 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 canonical means of packaging software included with the Python Standard Library, *setuptools* also provided a degree of backwards compatibility with it.

Example 4.2 `setup.py` using *setuptools*

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


setuptools.setup(
    name="pymunincli",
    version="0.2",
    author="Julien Danjou",
    author_email="julien@danjou.info",
    description="munin client library",
    license="GPL",
    url="http://julien.danjou.info/software/pymunincli/",
    packages=['munin'],
    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"
    ],
)
```

Eventually, development on *setuptools* slowed down, and people began to consider it a dead project like the original *distutils*. 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, and this one's no different: in March 2013, the teams behind *setuptools* and *distribute* decided to merge their code bases 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 replacing *distutils* in the Python Standard Library outright. One of its most notable differences from both *distutils* and *setuptools* was that it stored package metadata in a plain text file, `setup.cfg`, which was both easier for developers to write and easier for external tools to read. However, it also retained some of the failings of *distutils*, such as its obtuse command-based design, and lacked support for things like entry points and native script execution on Windows - both features provided by *setuptools*. For these and other reasons, plans to include *distutils2* in the Python 3.3 Standard Library as *packaging* fell through, and the project was abandoned in 2012.

However, *packaging* still has a chance to rise from the ashes through *distlib*, an up-and-coming effort to replace *distutils* which - hopefully - will become part of the Standard Library in 3.4. It includes the best features from *packaging* and 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.
- *distribute* has been merged back into *setuptools* as of version 0.7.
- *distutils2* (a.k.a. *packaging*) has been abandoned.
- *distlib might* replace *distutils* in the future.

There are other packaging libraries out there, though these five are the ones you'll encounter the most in practice. Be careful when looking up information about them on the Internet: there's plenty of documentation out there that's outdated due to the complicated history outlined above. The official documentation is, at least, up to date.

The short version of all this is, *setuptools* is the distribution library to use for the time being, but keep an eye out for *distlib* in the future.

4.2 Packaging with *pbr*

Now that I've spent some pages making your head confused with a lot of distribution tools, let's talk, about another tool and alternative, called *pbr*.

You probably already have written some package and tried to write a `setup.py`, either by copying one from some other project, or by skimming through the documentation. It isn't an obvious task, as the various problem we discussed earlier about which tool to use are usually a first obstacle. In this section I want to introduce you to *pbr*, a tool you should use to write your next *setup.py* so you'll never have to lose your time on that part again.

pbr stands for *Python Build Reasonableness*. The project has been started inside OpenStack as a set of tools around *setuptools* to facilitate installation and deployment of packages. It takes inspiration from *distutils2*, using a `setup.cfg` file to describe the packager's intents.

This is how a `setup.py` using *pbr* looks like:

```
import setuptools


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

Two lines of code – it's that simple. The actual metadata that the setup requires is stored in `setup.cfg`:

```
[metadata]
name = foobar
author = Dave Null
author-email = foobar@example.org
summary = Package doing nifty stuff
license = MIT
description-file =
    README.rst
home-page = http://pypi.python.org/pypi/foobar
requires-python = >=2.6
classifier =
    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
```

```
[files]
packages =
    foobar
```

Sound familiar? That's right – this particular way of doing things was directly inspired by *distutils2*.

pbr also offers other features such as:

- automatic dependency installation based on `requirements.txt`
- automatic documentation using Sphinx
- automatic generation of `AUTHORS` and `ChangeLog` files based on *git* history
- automatic creation of file lists for *git*
- version management based on *git* tags

And all this with little to no effort on your part. *pbr* is well-maintained and in very active development, so if you have any plans to distribute your software, you should seriously consider including *pbr* in those plans.

4.3 The *Wheel* format

For most of Python's existence, there's been no official standard distribution format. While different distribution tools still generally use some kind of 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, which 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 is automatically integrated as a command:

```
python setup.py bdist_wheel
```

This will create a `.whl` file in the `dist` directory. Like with the *Egg* format, a *Wheel* archive is just a zip file with a different extension, except *Wheel* archives don't 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}
             ...

positional arguments:
[...]
```

You might be surprised to learn this isn't a feature introduced by the *Wheel* format. Python can also run regular zip files as well, 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 automatically be imported from `__main__.py`. It's also possible to import `__main__` from a module you

specify by appending a slash followed by its name, just like 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, etc.). This is particularly useful if you need to distribute modules written in C.

4.4 Package installation

setuptools introduced the first useful command for installing packages, *easy_install*. It allows you to install Python modules from *Egg* archives with a single command; unfortunately, *easy_install* has suffered a bad reputation from the beginning due to some of its more questionable behaviors, such as ignoring best practices for system administration and its lack of uninstall functionality.

The *pip* project offers a much better way to handle package installations. It's actively developed, well-maintained, and will be included with Python starting in 3.4 [1]. It can install or uninstall packages from PyPI, a tarball, or a *Wheel* (see Section 4.3) archive.

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%2 ↩
     Fpackages%2Fsource%2Fv%2Fvoluptuous%2Fvoluptuous-0.8.3.tar.gz
```

[1] See PEP 453 and the `ensurepip` module

```
  Running setup.py egg_info for package voluptuous
    WARNING: Could not locate pandoc, using Markdown long_description.

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
    WARNING: Could not locate pandoc, using Markdown long_description.

Successfully installed voluptuous
Cleaning up...
```

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.

Tip

If you're using *pip* to install the same packages over and over, you can make it use a local cache instead of downloading the packages each time. Just set the environment variable `PIP_DOWNLOAD_CACHE` to a directory: *pip* will then use it to store downloaded tarballs and will check that location for packages before downloading them. This is very useful when using *tox* (see Section 6.7), which needs to download packages to build virtual environments. You can also add the `download-cache` option to your `~/.pip/pip.conf` file.

You can list the packages that are currently installed by using the `pip freeze` command:

```
$ pip freeze
Babel==1.3
Jinja2==2.7.1
```

```
commando=0.3.4
…
```

All other installation tools are being deprecated in favor of *pip*, so you shouldn't have any trouble if you treat it as your one-stop shop for all your package management needs.

4.5 Sharing your work with the world

Once you have a proper `setup.py` file, it's easy to build a source tarball that you can distribute. Just use the `sdist` command:

Example 4.3 Using `setup.py sdist`

```
$ python setup.py sdist
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
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)
```

This will create a tarball under the `dist` directory of your source tree that contains all your packages and can be used to install your software. As seen in Section 4.3, you can also build *Wheel* archives using the *bdist_wheel* command.

The final step is to make things easy on your end users by setting things up where your package can be installed using *pip*. This means publishing your project to PyPI.

Since you'll probably make mistakes if this is your first time, 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 it's used 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
```

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
```

Finally, you can upload a source distribution tarball:

```
% 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
```

```
[…]

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
```

As well as a *Wheel* archive:

```
$ 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

[…]
```

```
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
```

You should now be able to search for your package on the PyPi staging server and see whether it uploaded properly. You can also try installing it using *pip*, specifying the test server using the `-i` option:

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

If everything checks out, you can continue to the next step: uploading your project to the main PyPI server. Just add your credentials and the details for the server to your ~/.pypirc` file:

```
[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>
```

Running *register* and *upload* with the `-r pypi` switch will now upload your package to PyPI proper.

4.6 Interview with Nick Coghlan

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*) for which he is acting as *BDFL* [2] delegate.

The number of packaging solutions (*distutils*, *setuptools*, *distutils2*, *distlib*, *bento*, *pbr*, etc.) for Python is quite impressive. In your opinion, what are the (possibly historical) 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. The long answer can be found in the Python Packaging User Guide. In my recent talks on this, I have noted that the problem is mainly one of age and the aforementioned tools being born in a somewhat different era of software distribution.

***setuptools* is the *de facto* standard for Python distributions nowadays. Is there anything you think users should be aware of when using it (or not)?**

setuptools is quite reasonable as a build system, especially for pure Python

[2] "Benevolent Dictator For Life," title given to Guido van Rossum, author of Python

projects, or those with only simple C extensions. It also offers a powerful system for plugin registration and good cross-platform script generation.

While effective, the multi-version support in *pkg_resources* is also a bit quirky and tricky to use properly. Unless there's a really compelling reason to have conflicting versions in the same environment, it's much easier to just use *virtualenv* or *zc.buildout*.

PEP 426, which defines a new metadata format for Python packages, is still fairly recent and not yet approved. Is it on good track? What motivated it in the first place, how do you think it'll tackle the current problems?

PEP 426 originally started as part of the *Wheel* format definition, but Daniel Holth eventually 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 (like *RPM* and *npm*), and also addresses some of the frustrations encountered with existing tools (like cleanly separating different kinds of dependencies).

If PEP 426 is accepted, what kinds of tools would you to see built to take advantage of what it offers?

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 fairly recent and not widely used yet, but it seems promising. Is there any reason it isn't part of the Standard Library, or are there already plans to include it?

It turns out the Standard Library isn't really a suitable place for packaging standards: it evolves too slowly, and an addition to a later version of the

Standard Library can't 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. *python-dev* will only be involved for proposals that involve changing CPython directly (like *pip* bootstrapping).

What kind of future do you envision that would push developers to build and distribute *Wheel* packages?

pip is adopting it at 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 Mac OS X. We still have some tweaks to make before it will be suitable for use on Linux.

4.7 Entry points

You may have already used *setuptools* entry points without knowing anything about them. If you haven't yet decided to use *setuptools* (or *pbr*, see Section 4.2) to provide a `setup.py` file with your software, here are a few features that might help you make up your mind.

Software distributed using *setuptools* includes important metadata describing things such as its required dependencies and – more relevantly to this topic – a list of "entry points." These entry points can be used by other Python programs to dynamically discover features that a package provides.

In the following sections, we will discuss how we can use entry points to add extensibility to our software.

4.7.1 Visualising entry points

The easiest way to visualize the entry points available in a package is to use a package called *entry_point_inspector*.

When installed, it provides a command called `epi` that you can run from your terminal to interactively discover the entry points provided by installed packages:

Example 4.4 Result of *epi group list*

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

Example 4.4 shows that we have many different packages that provide entry points. You'll also notice this list includes *console_scripts*, which we'll discuss in Section 4.7.2.

Example 4.5 Result of *epi group show console_scripts*

```
+----------+----------+--------+--------------+-------+
| Name     | Module   | Member | Distribution | Error |
+----------+----------+--------+--------------+-------+
| coverage | coverage | main   | coverage 3.4 |       |
+----------+----------+--------+--------------+-------+
```

Example 4.5 shows us that an entry point named *coverage* refers to the member *main* of the module *coverage*. This entry point is provided by the package *coverage 3.4*. We can obtain more information by using *epi ep show*:

Example 4.6 Result of *epi ep show console_scripts coverage*

```
+--------------+----------------------------------+
| Field        | Value                            |
+--------------+----------------------------------+
| Module       | coverage                         |
| Member       | main                             |
| Distribution | coverage 3.4                     |
| Path         | /usr/lib/python2.7/dist-packages |
| Error        |                                  |
+--------------+----------------------------------+
```

The tool we're using here is just a thin layer on top of a more complete Python library which can help us discover entry points for any Python library or program. Entry points are useful for various things, including console scripts and dynamic code discovery, as we're going to see in the next few sections.

4.7.2 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 actually run. This program needs to be installed inside a directory somewhere in the system path.

Most projects will have something along the lines of this:

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

```

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

This is actually a best-case scenario: many projects have a much longer script installed in the system path. But using such scripts has some major issues:

- There's no way they can know where the Python interpreter is or which version it will be.
- They leak binary code that can't be imported by software or unit tests.
- There's no easy way to define where to install them.
- It's not obvious how to install this in a portable way (Unix vs Windows for example).

setuptools has a feature that helps us circumvent these problems: *console_scripts*. *console_scripts* is an entry point that can be used to make *setuptools* install a tiny program in the system path which then calls a specific function in one of your modules.

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

foobar/client.py

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

foobar/server.py

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

Of course, our program doesn't really do much of anything – our client and server don't even talk to each other. For the purposes of our example, though, all they need to do is print a message letting us know they've started successfully.

We can now write the following `setup.py` file in the root directory:

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": [
            "foobard = foobar.server:main",
            "foobar = foobar.client:main",
        ],
     },
)
```

We define our entry points using the format `package.subpackage:function`.

When you run `python setup.py install`, *setuptools* will create a script that will look like this:

Example 4.7 A console script generated by *setuptools*

```
#!/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')()
    )
```

This code scans the entry points of the `foobar` package and retrieves the `foobar` key from the `console_scripts` category, which is used to locate and run the corresponding function.

Using this technique will ensure that your code stays in your Python package and can be imported (and tested) by other programs.

Tip

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.

4.7.3 Using plugins and drivers

Entry points make it easy to discover and dynamically load code deployed by other packages. You can use **pkg_resources** to discover and load entry point files from within your Python programs. (You might notice that this is the same package used in the console script that *setuptools* creates, as seen in Example 4.7.)

In this section, we're going to create a *cron*-style daemon 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 this entry point points to should be an object that returns `number_of_seconds, callable`.

Here's our implementation of *pycrond* using `pkg_resources` to discover entry points:

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 is a very simple and naive implementation, but it's sufficient for our example. Now we can write another Python program that needs one of its functions called on a periodic basis:

hello.py

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


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

We register the function using the appropriate entry points:

setup.py

```
from setuptools import setup
```

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

And now if we run our *pytimed* script, we'll see "Hello, world!" printed on the screen every 2 seconds:

Example 4.8 Running pytimed

```
% python3
Python 3.3.2+ (default, Aug  4 2013, 15:50:24)
[GCC 4.8.1] on linux
Type "help", "copyright", "credits" or "license" for more  ↩
   information.
>>> import pytimed
>>> pytimed.main()
Hello, world!
Hello, world!
Hello, world!
```

The possibilities this mechanism offers are huge: it allows you to build driver systems, hook systems, and extensions in an easy and generic way. 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.

stevedore provides support for dynamic plugins based on the exact same mechanism demonstrated in our previous examples. Our use case in this example isn't very complicated, but we can still simplify it a bit using **stevedore**:

pytimed_stevedore.py

```
from stevedore.extension import ExtensionManager
import time

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

Our example is still very simple, but if you look through the *stevedore* documentation, you'll 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.

5 Virtual environments

When dealing with Python applications, there's always a time where you'll have to deploy, use and/or test your application. But doing that can be really painful, because of the external dependencies. There's a lot of reasons for which that may fail to deploy or operate on your operation system, such as:

- 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.

This can happen right at the time you deploy your application, or later on while running. Upgrading a Python library installed via your system manager might break your application in a snap without warning you.

The solution to this problem is to use a library directory per application, containing its dependencies. This directory will be used rather than the system installed ones to load the needed Python modules.

The tool `virtualenv` handles these directories automatically for you. Once installed, you just need to run it with a destination directory as argument.

```
$ virtualenv myvenv
Using base prefix '/usr'
New python executable in myvenv/bin/python3
```

```
Also creating executable in myvenv/bin/python
Installing Setuptools.....................done.
Installing Pip............................done.
```

Once ran, `virtualenv` creates a `lib/pythonX.Y` directory and uses it to install `setuptools` and `pip`, that will be necessary to install further Python packages.

You can now activate the virtualenv by "sourcing" the `activate` command:

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

Once you do that, your shell prompt will be prefixed by the name of your virtual environment. Calling `python` will call the Python that has been copied into the virtual environment. You can check that its working by reading the `sys.path` variable; it will have 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 not that you're not force 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 don't have access to any of the module installed and available on the system. That's good, but we probably need to install them. To do that, you just have to use the standard `pip` command, and that will install the packages in the right place, without changing anything to 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...
```

And voilà. We can install all the libraries we need and then run our application from this virtual environment, without breaking our system. It's then easily imaginable to script this to automatize the installation of a virtual environment based on a list of a dependency with something along these lines:

Example 5.1 Automatic virtual environment creation

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

In certain situation, it's still useful to have access to your system installed packages. You can enable them when creating your virtual environment by passing the `--system-site-packages` flag to the `virtualenv` command.

As you might guess, virtual environments are utterly useful for automated run of unit test suite. This is a really common pattern, so common that a special tool has been built to solve it, called `tox` (discussed in Section 6.7).

More recently, the PEP 405 [1] which defines a virtual environment mechanism has been accepted and implemented in Python 3.3. Indeed, the usage of virtual envi-

[1] *Python Virtual Environments*, 13th June 2011, Carl Meyer

ronment became so popular that it is now part of the standard Python library.

The `venv` module is now part of Python 3.3 and above, and allows to handle virtual environment without using the `virtualenv` package or any other one. You can call it using the `-m` flag of Python, which loads a module:

```
$ python3.3 -m venv
usage: venv [-h] [--system-site-packages] [--symlinks] [--clear] [--upgrade ↩
   ]
            ENV_DIR [ENV_DIR ...]
venv: error: the following arguments are required: ENV_DIR
```

Building virtual environment is then really simple:

```
$ python3.3 -m venv myvenv
```

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

The mechanism to activate the virtual environment is the same as described earlier, "sourcing" the activate script:

```
$ source myvenv/bin/activate
(myvenv) $
```

Also here, you can call `deactivate` to leave the virtual environment.

The downside of this `venv` module is that it doesn't install `setuptools` nor `pip` by default. We will have to bootstrap the environment by ourself, contrary to `virtualenv` that does that for us.

Example 5.2 Boostraping a `venv` environment

```
(myvenv) $ wget https://bitbucket.org/pypa/setuptools/raw/bootstrap/ ↩
   ez_setup.py -O - | python
```

```
-2013-09-02 22:26:07--  https://bitbucket.org/pypa/setuptools/raw/bootstrap ↩
   /ez_setup.py
Resolving bitbucket.org (bitbucket.org)... 131.103.20.168, 131.103.20.167
Connecting to bitbucket.org (bitbucket.org)|131.103.20.168|:443...  ↩
   connected.
HTTP request sent, awaiting response... 200 OK
Length: 11835 (12K) [text/plain]
Saving to: ‘STDOUT’

100%[=============================================>] 11,835      --.-K/s    ↩
   in 0s

2013-09-02 22:26:08 (184 MB/s) - written to stdout [11835/11835]

Downloading https://pypi.python.org/packages/source/s/setuptools/setuptools ↩
   -1.1.tar.gz
Extracting in /tmp/tmp228fqm
Now working in /tmp/tmp228fqm/setuptools-1.1
Installing Setuptools
running install
running bdist_egg
running egg_info
writing dependency_links to setuptools.egg-i
[…]
Adding setuptools 1.1 to easy-install.pth file
Installing easy_install script to /home/jd/myvenv/bin
Installing easy_install-3.3 script to /home/jd/myvenv/bin

Installed /home/jd/myvenv/lib/python3.3/site-packages/setuptools-1.1-py3.3. ↩
   egg
```

```
Processing dependencies for setuptools==1.1
Finished processing dependencies for setuptools==1.1
```

We can then install *pip* via `easy_install`:

```
(myvenv) $ easy_install pip
Searching for pip
Reading https://pypi.python.org/simple/pip/
Best match: pip 1.4.1
Downloading https://pypi.python.org/packages/source/p/pip/pip-1.4.1.tar.gz# ↩
   md5=6afbb46aeb48abac658d4df742bff714
Processing pip-1.4.1.tar.gz
Writing /tmp/easy_install-hxo3b0/pip-1.4.1/setup.cfg
Running pip-1.4.1/setup.py -q bdist_egg --dist-dir /tmp/easy_install-hxo3b0 ↩
   /pip-1.4.1/egg-dist-tmp-efgi80
warning: no files found matching '*.html' under directory 'docs'
warning: no previously-included files matching '*.rst' found under  ↩
   directory 'docs/_build'
no previously-included directories found matching 'docs/_build/_sources'
Adding pip 1.4.1 to easy-install.pth file
Installing pip script to /home/jd/myvenv/bin
Installing pip-3.3 script to /home/jd/myvenv/bin

Installed /home/jd/myvenv/lib/python3.3/site-packages/pip-1.4.1-py3.3.egg
Processing dependencies for pip
Finished processing dependencies for pip
```

We can then use `pip` to install any further package we would need.

So while Python 3.3 includes `venv` by default, one has to admit that it has this little drawback to not come with what you would expect by default. It's easy enough to write a tool using the `venv` library that would mimic the default behaviour of `virtu`

`alenv`, but on the other side, there's little point working on that unless you are only targeting Python 3.3 and above. On the other hand, the *pip* bootstrapping code has been merged into Python 3.4, meaning that this bootstrap problem is solved by the latest Python version.

Anyway, since like most projects, you probably target Python 2 and Python 3, relying only on the `venv` module isn't really an option. Sticking with `virtualenv` for now is probably the best solution. Considering that they both function in an identical manner, this shouldn't be a problem.

6 Unit testing

Breaking news! It's 2013 and there are still people who don't have a policy of testing their projects. Now, the purpose of this book is not to convince you to jump in and start unit testing. If you need to be convinced, I suggest you start by reading about the benefits of test-driven development. Writing code that is not tested is essentially useless, as there's no way to conclusively prove that it works.

This section will cover the Python tools you can use to construct a great suite of tests. We'll talk about how you can utilise them to enhance your software, making it rock-solid and regression free!

6.1 The basics

Contrary to what you may believe, the writing and running of unit tests is really simple in Python. It's not intrusive or disruptive, and it's going to help you and other developers a lot in maintaining your software.

Your tests should be stored inside a `tests` submodule of your application or library. This allows you to ship the tests as part of your module, so that they can be run or reused by anyone – even once your software is installed – without necessarily using the source package. This also prevents them from being installed by mistake in a top-level `tests` module.

It's usually simpler to use a hierarchy in your test tree that mimics the hierarchy you have in your module tree. This means that the tests covering the code of `mylib/foobar.py` should be inside `mylib/tests/test_foobar.py`; this makes things simpler when looking for the tests relating to a particular file.

Example 6.1 A really simple test in `test_true.py`

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

This is the most simple unit test that can be written. To run it, you simply need to load the `test_true.py` file and run the `test_true` function defined within.

Obviously, following these steps for all of your test files and functions would be a pain. This is where the *nose* package comes to the rescue – once installed, it provides the `nosetests` command, which loads every file whose name starts with `test_` and then executes all functions within that start with `test_`.

Therefore, with the `test_true.py` file in our source tree, running `nosetests` will give us the following output:

```
$ nosetests -v
test_true.test_true ... ok


----------------------------------------------------------------
Ran 1 test in 0.003s


OK
```

On the other hand, as soon as a test fails, the output changes to indicate the failure, accompanied by the whole traceback.

```
% nosetests -v
test_true.test_true ... ok
test_true.test_false ... FAIL
```

```
======================================================
FAIL: test_true.test_false

Traceback (most recent call last):
  File "/usr/lib/python2.7/dist-packages/nose/case.py", line 197, in ↩
     runTest
    self.test(*self.arg)
  File "/home/jd/test_true.py", line 5, in test_false
    assert False
AssertionError
----------------------------------------------------------
Ran 2 tests in 0.003s

FAILED (failures=1)
```

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

Simple, isn't it? While simplistic, this approach is used by a lot of small projects, and works very well. They don't require tools or libraries other than than `nose`, and relying on `assert` is good enough.

However, as you start to write more sophisticated tests, you'll start to become frustrated by things like the use of `assert`. Consider the following test:

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

When running `nosetests`, it gives the following output:

```
$ nosetests -v
test_complicated.test_key ... FAIL

======================================================================
FAIL: test_complicated.test_key
Traceback (most recent call last):
  File "/usr/lib/python2.7/dist-packages/nose/case.py", line 197, in  ↩
     runTest
    self.test(*self.arg)
  File "/home/jd/test_complicated.py", line 4, in test_key
    assert a == b
AssertionError

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Alright, so `a` and `b` are different and this test doesn't pass. But how are they different? `assert` doesn't give us this information, just states that the assertion is wrong – not particularly useful.

Also, with such a basic zero framework approach, advanced usage such as skipping tests or executing actions before or after running every test can become painful.

This is where the `unittest` package comes in handy. It provides tools that will help covering all of that – and good news is that `unittest` is part of the Python standard library.

Warning

`unittest` has been largely improved starting with Python 2.7, so if you are supporting earlier version of Python you may want to use its backport named `unittest2`. If you need to support Python 2.6, you can then use the following snippet to import the correct module for any Python versions at runtime:

```
try:
    import unittest2 as unittest
except ImportError:
    import unittest
```

If we rewrite the previous example using `unittest`, this is what it will look like:

```
import unittest

class TestKey(unittest.TestCase):
    def test_key(self):
        a = ['a', 'b']
        b = ['b']
        self.assertEqual(a, b)
```

As you can see, the implementation isn't much more complicated. All you have to do is create a class that inherits from `unittest.TestCase`, and write a method that runs a test. Instead of using `assert`, we rely on a method provided by `unittest.TestCase` that provides an equality tester. When run, it outputs the following:

```
$ nosetests -v
test_key (test_complicated.TestKey) ... FAIL

======================================================
FAIL: test_key (test_complicated.TestKey)
Traceback (most recent call last):
```

```
  File "/home/jd/Source/python-book/test_complicated.py", line 7, in  ↩
     test_key
    self.assertEqual(a, b)
AssertionError: Lists differ: ['a', 'b'] != ['b']

First differing element 0:
a
b

First list contains 1 additional elements.
First extra element 1:
b

- ['a', 'b']
+ ['b']

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

As you can see, the output is much more useful. An assertion error is still raised, and the test is still being failed, but at least we have real information about why it's failing, which can help us to fix the problem. This is why you should definitely **never** use `assert` when writing test cases. Anyone who tries to hack your code and ends up failing a test will definitely thank you for having not used `assert`, and having thereby providing him/her with debugging information right away.

`unittest` provides a few test functions that you can use to specialize your tests, such as: `assertDictEqual`, `assertEqual`, `assertTrue`, `assertFalse`, `assertGreater`, `assertGreaterEqual`, `assertIn`, `assertIs`, `assertIsIntance`, `assertIsNone`, `asser`

tIsNot, assertIsNotNone, assertItemsEqual, assertLess, assertLessEqual, assertListEqual, assertMultiLineEqual, assertNotAlmostEqual, assertNotEqual, assertTupleEqual, assertRaises, assertRaisesRegexp, assertRegexpMatches, etc. It would be a good idea to go through `pydoc unittest` and discover them all.

It's also possible to deliberately fail a test right away using the `fail(msg)` method. This can be convenient when you know that a particular part of your code will definitely raise an error if executed, but there isn't a particular assertion to check for.

Example 6.2 Failing a test

```
import unittest

class TestFail(unittest.TestCase):
    def test_range(self):
        for x in range(5):
            if x > 4:
                self.fail("Range returned a too big value: %d" % x)
```

It's sometimes useful skip a test if it can't be run – 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 raise the **`unittest.SkipTest`** exception. When the test is raised, it is simply marked as having been skipped. The convenient method `unittest.TestCase.skipTest()` can be used rather than raising the exception manually, as can the `unittest.skip` decorator:

Example 6.3 Skipping tests

```
import unittest

try:
    import mylib
except ImportError:
    mylib = None
```

```
class TestSkipped(unittest.TestCase):
    @unittest.skip("Do not run this")
    def test_fail(self):
        self.fail("This should not be run")

    @unittest.skipIf(mylib is None, "mylib is not available")
    def test_mylib(self):
        self.assertEqual(mylib.foobar(), 42)

    def test_skip_at_runtime(self):
        if True:
            self.skipTest("Finally I don't want to run it")
```

When executed, this test file will output the following:

```
$ python -m unittest -v test_skip
test_fail (test_skip.TestSkipped) ... skipped 'Do not run this'
test_mylib (test_skip.TestSkipped) ... skipped 'mylib is not available'
test_skip_at_runtime (test_skip.TestSkipped) ... skipped "Finally I don't ↩
   want to run it"

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK (skipped=3)
```

Tip

As you may have noticed in Example 6.3, the `unittest` module provides a way to execute a Python module that contains tests. It is less convenient than using `nosetests`, as it does not discover test files on its own, but it can still be useful for running a particular test module.

In many cases, there's a need to execute a set of common actions before and after running a test. `unittest` provides two particular methods called **`setUp`** and **`tearDown`** that are executed each time one of the test methods of a class is about to, or has been, called.

Example 6.4 Using `setUp` with `unittest`

```
import unittest

class TestMe(unittest.TestCase):
    def setUp(self):
        self.list = [1, 2, 3]

    def test_length(self):
        self.list.append(4)
        self.assertEqual(len(self.list), 4)

    def test_has_one(self):
        self.assertEqual(len(self.list), 3)
        self.assertIn(1, self.list)
```

In this case, `setUp` is called before running `test_length` and before running `test_has_one`. It can be really handy to create objects that are worked with during each test; but you need to be sure that they get recreated in a clean state before each test method is called. This is really useful for creating test environments, often referred

to as "fixtures" (see Section 6.2).

Tip

When using `nosetests`, you often might want to run only one particular test. You can select which test you want to run by passing it as an argument – the syntax is: *path.to.your.module:ClassOfYourTest.test_method*. Be sure that there's a colon between the module path and the class name. You can also specify *path.to.your.module:ClassOfYourTest* to execute an entire class, or *path.to.your.module* to execute an entire module.

Tip

It's possible to run tests in parallel to speed things up. Simply add the `--processes=N` option to your `nosetests` invocation to spawn several `nosetests` processes. However, `testrepository` is a better alternative – this is discussed in Section 6.5.

6.2 Fixtures

In unit testing, fixtures represent components that are set up before a test, and cleaned up after the test is done. It's usually a good idea to build a special kind of component for them, as they are reused in a lot of different places. For example, if you need an object which represents the configuration state of your application, there's a chance you may want it to be initialized before each test, and reset to its default values when the test is done. Relying on temporary file creation also requires that the file is created before the test starts, and deleted once the test is done.

`unittest` only provides the `setUp` and `tearDown` functions we already evoked. However, a mechanism exists to hook into these. The **`fixtures`** Python module (not part of the standard library) provides an easy mechanism for creating fixture classes and objects, such as the `useFixture` method.

The `fixtures` modules provides a few built-in fixtures, like `fixtures.EnvironmentVariable` – useful for adding or changing a variable in `os.environ` that will be reset upon test exit.

Example 6.5 Using `fixtures.EnvironmentVariable`

```
import fixtures
import os

class TestEnviron(fixtures.TestWithFixtures):
    def test_environ(self):
        fixture = self.useFixture(
            fixtures.EnvironmentVariable("FOOBAR", "42"))
        self.assertEqual(os.environ.get("FOOBAR"), "42")

    def test_environ_no_fixture(self):
        self.assertEqual(os.environ.get("FOOBAR"), None)
```

When you can identify common patterns like these, it's a good idea to create a fixture that you can reuse over all your test cases. This greatly simplifies the logic, and shows exactly what you are testing and in what manner.

Note

If you're wondering why the base class `unittest.TestCase` isn't used in the examples in this section, it's because `fixtures.TestWithFixtures` inherits from it.

6.3 Mocking

Mock objects are simulated objects that mimic the behaviour of real application objects, but in particular and controlled ways. These are especially useful in creat-

ing environments that describe precisely the conditions for which you would like to test code.

If you are writing an HTTP client, it's likely impossible (or at least extremely complicated) to spawn the HTTP server and test it through all scenarios, making it return every possible value. It's especially difficult to test for all failure scenarios.

A much simpler option is to build a set of mock objects that model these particular scenarios, and to use them as environment for testing your code.

The standard library for creating mock objects in Python is `mock`. Starting with Python 3.3, it has been merged into the Python standard library as `unittest.mock`. You can therefore use a snippet like:

```
try:
    from unittest import mock
except ImportError:
    import mock
```

To maintain backward compatibility between Python 3.3 and earlier versions.

Mock is pretty simple to use:

Example 6.6 Basic mock usage

```
>>> import mock
>>> m = mock.Mock()
>>> m.some_method.return_value = 42
>>> m.some_method()
42
>>> def print_hello():
...     print("hello world!")
...
>>> m.some_method.side_effect = print_hello
>>> m.some_method()
```

```
hello world!
>>> def print_hello():
...     print("hello world!")
...     return 43
...
>>> m.some_method.side_effect = print_hello
>>> m.some_method()
hello world!
43
>>> m.some_method.call_count
3
```

Even using just this set of features, you should be able to mimic a lot of your internal objects in order to fake various data scenarios.

Mock uses the action/assertion pattern: this means that once your test has run, you will have to check that the actions you are mocking were correctly executed.

Example 6.7 Checking method calls

```
>>> import mock
>>> m = mock.Mock()
>>> m.some_method('foo', 'bar')
<Mock name='mock.some_method()' id='26144272'>
>>> m.some_method.assert_called_once_with('foo', 'bar')
>>> m.some_method.assert_called_once_with('foo', mock.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')
```

As you can see, it's easy enough to pass a *mock* object to any part of your code, and to check later if the code has been called with whatever argument it was supposed to have. If you don't know what arguments may have been passed, you can use `mock.ANY` as a value; that will match any argument passed to your mock method.

Sometimes you may need to a some function, method or object from an external module. `mock` provides a set of patching functions to that end.

Example 6.8 Using `mock.patch`

```
>>> 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!
```

With the `mock.patch` method, it's possible to change any part of an external piece of code – making it behave in the required way in order to test all conditions in your software.

Example 6.9 Using `mock.patch` to test a set of behaviour

```
import requests
import unittest
import mock

class WhereIsPythonError(Exception):
    pass

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)

class TestPython(unittest.TestCase):
    @mock.patch('requests.get', get_fake_get(
```

```
        200, 'Python is a programming language for sure'))
    def test_python_is(self):
        self.assertTrue(is_python_still_a_programming_language())

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

    @mock.patch('requests.get', get_fake_get(
        404, 'Whatever'))
    def test_bad_status_code(self):
        self.assertRaises(WhereIsPythonError,
                          is_python_still_a_programming_language)

    @mock.patch('requests.get', raise_get)
    def test_ioerror(self):
        self.assertRaises(WhereIsPythonError,
                          is_python_still_a_programming_language)
```

Example 6.9 uses the decorator version of `mock.patch`, this does not change its behaviour, but is easier to use when you need to use mocking within the context of an entire test function.

By using mocking we can simulate any problem – such as a Web server returning a 404 error, or network issues arising. We can make sure that our code returns the correct values, or raises the correct exception in every case – ensuring that our code always behaves as expected.

6.4 Scenarios

When unit testing, it is common to require that a set of tests be run against different versions of an object. You may want to run the same error-handling test with a bunch of different objects that trigger that error; or you may want to run an entire test suite against different drivers.

This last case is one that we heavily relied on in *Ceilometer*. *Ceilometer* provides an abstract class that we call the storage API. Any driver can implement this base abstract class 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, what need is a class of unit tests that runs against each driver – meaning against each implementation of this storage API – to be sure that they conform to what the callers expect.

The natural way of doing this is to use mixin classes; on one side, you would have a class with unit tests, and on the other side a class with the specific driver usage setup.

```
import unittest


class MongoDBBaseTest(unittest.TestCase):
    def setUp(self):
        self.connection = connect_to_mongodb()


class MySQLBaseTest(unittest.TestCase):
    def setUp(self):
        self.connection = connect_to_mysql()


class TestDatabase(unittest.TestCase):
    def test_connected(self):
        self.assertTrue(self.connection.is_connected())
```

```
class TestMongoDB(TestDatabase, MongoDBBaseTest):
    pass

class TestMySQL(TestDatabase, MySQLBaseTest):
    pass
```

Unfortunately, in the long run this method is far from convenient or scalable.

A better technique does exist, using the **testscenarios** package. This package provides an easy way to run a class test against a different set of scenarios generated at run-time. Using `testscenarios`, I have rewritten part of Example 6.9 to illustrate mocking as covered in Section 6.3.

Example 6.10 `testscenarios` basic usage

```
import mock
import requests
import testscenarios

class WhereIsPythonError(Exception):
    pass

def is_python_still_a_programming_language():
    r = requests.get("http://python.org")
    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

class TestPythonErrorCode(testscenarios.TestWithScenarios):
    scenarios = [
        ('Not found', dict(status=404)),
        ('Client error', dict(status=400)),
        ('Server error', dict(status=500)),
    ]

    def test_python_status_code_handling(self):
        with mock.patch('requests.get',
                        get_fake_get(
                            self.status,
                            'Python is a programming language for sure')):
            self.assertRaises(WhereIsPythonError,
                              is_python_still_a_programming_language)
```

Even though only one test seems to be defined, `testscenarios` runs the test three times – because we have defined three scenarios.

```
% python -m unittest -v test_scenario
test_python_status_code_handling (test_scenario.TestPythonErrorCode) ...  ↩
   ok
test_python_status_code_handling (test_scenario.TestPythonErrorCode) ...  ↩
   ok
test_python_status_code_handling (test_scenario.TestPythonErrorCode) ...  ↩
   ok
```

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


OK
```

As can see, all we need to construct the scenario list is a tuple list that consists of the scenario name as first argument, and as a second argument the dictionary of attributes to be added to the test class for this scenario.

It is easy enough to imagine another use: where instead of storing a single value as an attribute for each test, you could instantiate a particular driver and run all the tests of the class against it.

Example 6.11 Using `testscenarios` to test drivers

```
import testscenarios
from myapp import storage

class TestPythonErrorCode(testscenarios.TestWithScenarios):
    scenarios = [
        ('MongoDB', dict(driver=storage.MongoDBStorage())),
        ('SQL', dict(driver=storage.SQLStorage())),
        ('File', dict(driver=storage.FileStorage())),
    ]

    def test_storage(self):
        self.assertTrue(self.driver.store({'foo': 'bar'}))

    def test_fetch(self):
        self.assertEqual(self.driver.fetch('foo'), 'bar')
```

Note

If you wonder why there is no need to use the base class `unittest.TestCase` in the previous examples, it's because `testscenarios.TestWithScenarios` inherits from it.

6.5 Test streaming and parallelism

When performing a lot of tests, it can be useful to analyze them as they are run. The default behaviour of tools like `nosetests` is to output the result to *stdout* – which is not really convenient to parse or analyze.

`subunit` is a Python module that provides a streaming protocol for test results. It allows for a number of interesting things, such as aggregating test results [1] or to record and archive test runs, etc.

Running a test using `subunit` is simple enough:

```
$ python -m subunit.run test_scenario
```

The output of this command is binary data, so unless you have the ability to sight-read the *subunit* protocol, it wouldn't be interesting to reproduce it's output directly here. However, `subunit` also comes with a set of tools to transform this binary stream into something smoother:

Example 6.12 Using `subunit2pyunit`

```
$ python -m subunit.run test_scenario | subunit2pyunit
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Not ↩
   found)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Not ↩
   found) ... ok
```

[1] Even from different source programs or languages

```
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Client ↩
   error)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Client ↩
   error) ... ok
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Server ↩
   error)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Server ↩
   error) ... ok

----------------------------------------------------------
Ran 3 tests in 0.061s

OK
```

Now this is something that we can understand – you should recognize the test suite with scenarios from Section 6.4. Other tools worth mentioning include `subunit2csv`, `subunit2gtk` and `subunit2junitxml`.

`subunit` is also able to automatically discover which test to run, when it is passed the `discover` argument.

```
$ python -m subunit.run discover | subunit2pyunit
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Not ↩
   found)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Not ↩
   found) ... ok
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Client ↩
   error)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Client ↩
   error) ... ok
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Server ↩
   error)
```

```
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Server ↩
   error) ... ok


----------------------------------------------------------------
Ran 3 tests in 0.061s


OK
```

You can list tests, rather than running them, by passing the argument `--list`. To view the results, you can use `subunit-ls`:

```
$ python -m subunit.run discover --list | subunit-ls --exists
test_request.TestPython.test_bad_status_code
test_request.TestPython.test_ioerror
test_request.TestPython.test_python_is
test_request.TestPython.test_python_is_not
test_scenario.TestPythonErrorCode.test_python_status_code_handling
```

Tip

You can also load a list of tests that you want to run – rather than running all tests – by using the `--load-list` option.

In large applications the number of tests can be overwhelming, so having programs to handle the stream of results is very useful. The **`testrepository`** package is intended to do just that; it provides the `testr` program, which you can use to handle a database of your test run.

```
$ testr init
$ touch .testr.conf
% python -m subunit.run test_scenario | testr load
Ran 4 tests in 0.001s
```

```
PASSED (id=0)
$ testr failing
PASSED (id=0)
$ testr last
Ran 3 tests in 0.001s
PASSED (id=0)
$ testr slowest
Test id                                          Runtime (s)
-----------------------------------------------  -----------
test_python_status_code_handling(Not found)      0.000
test_python_status_code_handling(Server error)   0.000
test_python_status_code_handling(Client error)   0.000
$ testr stats
runs=1
```

Once the *subunit* stream of tests has been run and loaded inside *testrepository*, it is possible to manipulate it easily using the `testr` command.

Obviously, this is tedious to do by hand each time you want to run tests. Instead, you should teach `testr` how it should run your tests, so that it can load the results itself. This can be accomplished by editing the `.testr.conf` file at the root of your project.

Example 6.13 A `.testr.conf` file

```
[DEFAULT]
test_command=python -m subunit.run discover . $LISTOPT $IDOPTION ❶
test_id_option=--load-list $IDFILE ❷
test_list_option=--list ❸
```

❶ Command to run when calling `testr run`

❷ Command to run to load a test list

❸ Command to run to list tests

The first line, `test_command`, is the one that is the most interesting. Now, all that we need to do to load tests into *testrepository* and perform them is to run `testr run`.

Note

If you're accustomed to running `nosetests`, `testr run` is now the equivalent command.

Two other options enable us to run the tests in parallel. This is simple enough to do – all you need to do is add the `--parallel` switch to `testr run`. Running your tests in parallel can speed up the process considerably.

Example 6.14 Running `testr run --parallel`

```
$ testr run --parallel
running=python -m subunit.run discover . --list
running=python -m subunit.run discover .  --load-list /tmp/tmpiMq5Q1
running=python -m subunit.run discover .  --load-list /tmp/tmp7hYEkP
running=python -m subunit.run discover .  --load-list /tmp/tmpP_9zBc
running=python -m subunit.run discover .  --load-list /tmp/tmpTejc5J
Ran 26 (+10) tests in 0.029s (-0.001s)
PASSED (id=7, skips=3)
```

Under the hood, `testr` runs the test listing operation, splits the test list into several sublists, and creates a separate Python process to run each sublist of test. By default, the number of sublists is equal to the number of CPUs in the machine being used. You can override the number of processes that by adding the `--concurrency` flag.

```
$ testr run --parallel --concurrency=2
```

As you can imagine, there's a lot of possibilities opened up by tools such as `subunit` and `testrepository` that have only be skimmed through in this section. I believe it's worth being familiar with them, because testing can greatly influence the quality of the software you will produce and release. Having powerful tools like these can save a lot of time.

testrepository also integrates with *setuptools* and deploys a `testr` command for it. This provides easier integration with *setup.py*-based workflows – you can, for example, document your entire project around *setup.py*. The command `setup.py testr` accepts a few options, such as `--testr-args` – which adds more options to the `testr` run, or `--coverage`, which will be covered in the next section.

6.6 Coverage

Code coverage is a tool which complements unit 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 code base 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 is the cherry on the cake.

Obviously, the first thing to do is to install the **`coverage`** Python module on your system. Once this is done you will have access to the `coverage` program command from your shell.[2]

Using `coverage` in standalone mode is straightforward, and can be useful- it could lead you to part of your programs that are never run, and which might be "dead code". In addition, using it while your unit tests are running provides an obvious benefit: you'll know which parts of the code are not being tested. The test tools

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

we've talked about so far are all integrated with `coverage`.

When using *nose*, you only need to add a few option switches to generate a nice code coverage output:

Example 6.15 Using `nosetests --with-coverage`

```
$ nosetests --cover-package=ceilometer --with-coverage tests/test_pipeline ↩
   .py
.........................................
Name                                  Stmts   Miss  Cover   Missing
ceilometer                                0      0   100%
ceilometer.pipeline                     152     20    87%   49, 59, 113,  ↩
   127-128, 188-192, 275-280, 350-362
ceilometer.publisher                     12      3    75%   32-34
ceilometer.sample                        31      4    87%   81-84
ceilometer.transformer                   15      3    80%   26-32, 35
ceilometer.transformer.accumulator       17      0   100%
ceilometer.transformer.conversions       59      0   100%

TOTAL                                   888    393    56%


----------------------------------------------------------------
Ran 46 tests in 0.170s

OK
```

Adding the `--cover-package` option is important, since otherwise you will see **every** Python package used, including standard library or third-party modules. The output includes the lines of code that are were not run – and which therefore have no tests. All you need to do now is spawn your favorite text editor and start writing some.

But you can do better, and make *coverage* generate nice HTML reports. Simply add the `--cover-html` flag, and the `cover` 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.

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`

If you want to be that guy, you can use the option `--cover-min-percentage=COVE`

R_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.

Warning

A code coverage score of 100% doesn't necessarily mean that the code is entirely tested and that 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. So while being a respectable goal, it doesn't indicate anything conclusive.

When using `testrepository`, `coverage` can be run using *setuptools* integration.

Example 6.16 Using coverage with *testrepository*

```
$ python setup.py testr --coverage
```

This will automatically run your test suite with *coverage* and generate an HTML report in the `cover` directory.

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

6.7 Using virtualenv with tox

In Chapter 5, the use of virtual environments is presented and discussed. One of their main uses is to provide a clean environment for running unit tests. It would be really sad if you thought that your tests were working, when in fact you were not, for example, respecting the dependency list.

You could 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. But this is such a common use case that an application dedicated to this task has already been built: **tox**.

Tox 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 fine.

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

```
$ touch tox.ini
```

You can now 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 :)
```

Obviously this alone is not very useful. In this instance, `tox` creates a virtual environment in *.tox/python* using its default Python version, uses `setup.py` to create a distribution of your package and then installs it inside this virtual environment. No commands are then run, because we didn't specify any in the configuration file.

We can change this default behaviour by adding a command that will be run inside our test environment. Editing tox.ini to include the following:

```
[testenv]
commands=nosetests
```

will run the command `nosetests` will likely fail, since we don't have `nosetests` installed in the virtual environment. We need to list it as part of the dependencies to be installed.

```
[testenv]
deps=nose
commands=nosetests
```

When run, `tox` will now recreate the environment, install the new dependency and run the command `nosetests`, which will execute all of our unit tests. Obviously, we might want to add more dependencies – you can list them in the *deps* configuration option, but you can also use the `-rfile` syntax to read from a file. If you're using *pbr* to manage your *setup.py* file, you know that it reads the dependencies from a file called `requirements.txt`. It is therefore a good idea to tell *tox* to use that file too:

```
[testenv]
deps=nose
     -rrequirements.txt
commands=nosetests
```

The `[testenv]` section of the file defines the parameters for all virtual environments managed by `tox`. But as mentioned previously, *tox* can manage multiple Python virtual environments – indeed, it's possible to run our tests under a Python version other than the default one by passing the `-e` flag to `tox`:

```
% 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] | nosetests
.......

----------------------------------------------------------
```

```
Ran 7 tests in 0.029s


OK
_____________________ summary _____________________
  py26: commands succeeded
  congratulations :)
```

By default, tox can simulate many environments: *py24*, *py25*, *py26*, *py27*, *py30*, *py31*, *py32*, *py33*, *jython* and *pypy*! You can even add your own. To add an environment or to create a new one, you just need to add another section named `[testenv:_envname_]`. If we want to run a different command for one of the environments, it's easy with the following `tox.ini` file:

```
[testenv]
deps=nose
commands=nosetests

[testenv:py27]
commands=pytest
```

This only overrides the commands for the `py27` environment; so *nose* will still be installed as part of the dependencies when running `tox -e py27`, but the command `pytest` will be run instead.

We can create a new environment with an unsupported version of Python right away:

```
[testenv]
deps=nose
commands=nosetests

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

We can now (attempt to) use Python 2.1 to run our test suite – although I don't think it will work.

Now, it is likely that you will want to support multiple Python versions. So it would be great to have `tox` run all the tests for all the Python versions you want to support by default. This can be done by specifying the environment list you want to use when `tox` is run without arguments:

```
[tox]
envlist=py26,py27,py33,pypy

[testenv]
deps=nose
commands=nosetests
```

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

We can also use `tox` to integrate other tests like `flake8`, as discussed in Section 1.4.

```
[tox]
envlist=py26,py27,py33,pypy,pep8

[testenv]
deps=nose
commands=nosetests

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

In this case, the *pep8* environment will be run using the default version of Python,

which is probably fine.[3]

Tip

When running `tox`, you will spot that all of the environments are built and run in sequence. This can often make the process very long. Since the 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 which runs all of the default environments from *envlist* in parallel. You should *pip install* it!

6.8 Testing policy

Having testing code embedded in your project is wonderful, but how you run it is also extremely important. There are too many projects that have test code which lays around, but which fails to be run for some reason.

While this topic is not strictly limited to Python, I consider it important enough to emphasize here: you should have a zero tolerance policy on untested code. No code should be merged unless there is a proper set of unit tests to cover it.

The minimum that you should aim for is to be sure that each of the commits you push pass all the tests. Having an automated way to do that is even better.

For example, *OpenStack* relies on a specific workflow based on Gerrit, Jenkins and Zuul. Each commit pushed goes through the code review system provided by Gerrit, and Zuul is in charge of running a set of testing jobs against it using Jenkins. Jenkins runs the unit testing, and various higher-level functional tests for each project. This ensures that the submitted patches pass all tests. Code reviewing by a couple of developers makes sure that all code that is committed has associated unit tests.

If you are using the popular GitHub hosting service, Travis CI provides a way to run a test after each push or merge, or against pull requests that are submitted. While it is

[3] You can still specify the `basepython` option if you want to change that

unfortunate that this done post-push, it's still a fantastic way to track regressions. Travis supports all significant Python versions out of the box, and it's possible to customize it to a high degree. Once you've activated Travis on your project via their Web interface, adding a file is simple: `.travis.yml` does the job for you.

Example 6.17 A `.travis.yml` example file

```
language: python
python:
  - "2.7"
  - "3.3"
# command to install dependencies
install: "pip install -r requirements.txt --use-mirrors"
# command to run tests
script: nosetests
```

Wherever your code is hosted, these days it is always possible to aim for some sort of automatic testing of your software, and to make sure that you are going forward with your project – not going backward by adding more bugs.

6.9 Interview with Robert Collins

You may have already used one of Robert's programs, without knowing – he 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 has written a lot of the Python tools described in this book, such as *fixtures*, *testscenarios*, *testrespository* and even *python-subunit*.

What kind of testing policy would you advise using? When is it acceptable not to test code?

I think it's an engineering trade-off – considering the likelihood of failure slipping through to production undetected, the cost of an undetected failure of that component, the size and cohesion of the team doing the work… Take OpenStack – 1600 contributors – a nuanced policy is very hard to work with there, as so many people have opinions. Generally speaking, there should be some automated check as part of landing in trunk 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 speaks to requiring functional tests that might be in different code bases. Unit tests are great for speed and pinning down corner cases. I think it's ok 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's a relatively rare situation: most things can be tested fairly cheaply, and the benefit of catching errors early is usually quite high.

What are the best strategies to put in place when writing Python code in order to make testing easier, and improve its quality?

Separate out concerns – don't do multiple things in one place; this makes reuse easier, and that makes it easier to put test doubles in place. Take a pure functional approach when you can (e.g. in a single method either calculate something, or change some state, but where possible avoid doing

both). That way you can test all of the calculating behaviour without dealing with state changes – such as writing to a database, talking to an HTTP server, etc. The benefit works the other way around too – you can replace the calculation logic for tests to provoke corner case behaviour and detect via mocks / test doubles that the expected state propagation happens as desired. The most heinous stuff to test IME is deeply layered stacks with complex cross-layer behavioural dependencies. There you want to evolve the code so that the contract between layers is simple, predictable, and most usefully for testing – replaceable.

In your opinion, what's the best way to organize unit tests in source code?

Having a hierarchy like `$ROOT/$PACKAGE/tests` – but I do just one for a whole source tree (vs e.g. `$ROOT/$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`.

There should be no imports from tests by the rest of the tree except perhaps a test_suite/load_tests function in the top level `__init__`. This permits easily detaching the tests for small footprint installations.

What are the tools that can be employed to build functional tests in Python?

I just use whichever flavour of `unittest` is in use in the project: it's sufficiently flexible (particularly with things like `testresources` and parallel runners) to cater for most needs.

How do you envision the future of unit testing libraries and frameworks in Python?

The big challenges I see are:

- the continued expansion of parallel capabilities in new machines – 4 CPU phones now. Existing unit test internal APIs aren't optimised for parallel workloads. My *StreamResult* work is aimed directly at 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 large variety of frameworks we have today: it would be great to be able to get a consolidated view across multiple projects – for integration testing – that have different test runners in use.

7 Methods and decorators

Python provides decorators as a handy way to modify functions. They were first introduced with `classmethod()` and `staticmethod()` in Python 2.2, but were overhauled through PEP 318 into something more flexible and readable. Python provides a few decorators (including the two mentioned above) right out of the box, but it seems that most developers don't understand how they actually work behind the scenes. This chapter aims to change that.

7.1 Creating decorators

A decorator is essentially a function that takes another function as an argument and replaces it with a new, modified function. Odds are good you've already used decorators to make your own wrapper functions. The simplest possible decorator is the identity function, which does nothing except return the original function:

```
def identity(f):
    return f
```

You can then use your decorator like this:

```
@identity
def foo():
    return 'bar'
```

Which is the same as:

```
def foo():
    return 'bar'


foo = identity(foo)
```

This decorator is useless, but it works. It just does nothing.

Example 7.1 A registering decorator

```
_functions = {}
def register(f):
    global _functions
    _functions[f.__name__] = f
    return f
```

@register def foo(): return *bar*

In this example, we register and store functions in a dictionary so we can retrieve them by their name later from that dictionary.

In the following sections, I'll explain the standard decorators that Python provides and how (and when) to use them.

The primary use case for decorators is factoring common code that needs to be called before, after, or around multiple function. If you ever wrote Emacs Lisp code you may have used *defadvice* that allows you to define code called around a function. Same things apply for developers having used the fabulous method combinations brought by CLOS [1].

Consider a set of functions that are called and need to check that the user name that they receive as argument:

```
class Store(object):
    def get_food(self, username, food):
```

[1] The Common Lisp Object System

```
        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 get food")
        self.storage.put(food)
```

The obvious first step here is to factor the checking code:

```
def check_is_admin(username):
    if username != 'admin':
        raise Exception("This user is not allowed to get 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)
```

Now our code looks a bit cleaner. But we can do even better if we use a decorator:

```
def check_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
```

```
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)
```

Using decorators like this makes it easier to manage common functionality. This is probably old hat to you if you have any serious Python experience, but what you might not realize is that this naive approach to implementing decorators has some major drawbacks.

As mentioned before, 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:

```
>>> 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'
```

Fortunately, the **functools** module included in Python solves this problem with the `update_wrapper` function, which copies these attributes to the wrapper itself. The source code of `update_wrapper` is self-explanatory:

Example 7.2 Source code of `functools.update_wrapper` in Python 3.3

```
WRAPPER_ASSIGNMENTS = ('__module__', '__name__', '__qualname__', '__doc__',
                       '__annotations__')
WRAPPER_UPDATES = ('__dict__',)
def update_wrapper(wrapper,
                   wrapped,
                   assigned = WRAPPER_ASSIGNMENTS,
                   updated = WRAPPER_UPDATES):
    wrapper.__wrapped__ = wrapped
    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, {}))
    # Return the wrapper so this can be used as a decorator via partial()
    return wrapper
```

If we take our previous example and use this function to update our wrapper, things work much more nicely:

```
>>> def foobar(username="someone"):
...     """Do crazy stuff."""
...     pass
...
>>> foobar = functools.update_wrapper(is_admin, foobar)
>>> foobar.__name__
'foobar'
>>> foobar.__doc__
'Do crazy stuff.'
```

It can get tedious to use `update_wrapper` manually when creating decorators, so **functools** provides a decorator for decorators called `wraps`:

Example 7.3 Using `functools.wraps`

```
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):
        return self.storage.get(food)
```

In our examples so far, we've always assumed that the decorated function would have a `username` passed to it as a keyword argument, but that might not always be the case. With this in mind, it's a better idea to build a smarter version of our decorator that can look at the decorated function's arguments and pull out what it needs.

To that end, the **inspect** module allows us to retrieve a function's signature and operate on it:

Example 7.4 Retrieving function arguments using **inspect**

```
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!"
```

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'}`. This means that our decorator doesn't have to check if the `username` parameter is a positional or a keyword argument: all it has to do is look for it in the dictionary.

7.2 How methods work in Python

You've probably written dozens of methods and thought nothing of them before now, 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:

Example 7.5 A Python 2 method

```
>>> class Pizza(object):
...     def __init__(self, size):
...         self.size = size
...     def get_size(self):
...         return self.size
...
>>> Pizza.get_size
<unbound method Pizza.get_size>
```

Python 2 tells us that the `get_size` attribute of the `Pizza` class is an **unbound** method.

Example 7.6 A Python 3 method

```
>>> 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>
```

In Python 3, the concept of unbound method has been removed entirely, and we're told `get_size` is a function.

The principle is the same in both cases: `get_size` is a function that is not tied to any particular object, and Python will raise an error if we try to call it:

Example 7.7 Calling unbound get_size in Python 2

```
>>> Pizza.get_size()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unbound method get_size() must be called with Pizza instance as ↩
   first argument (got nothing instead)
```

Example 7.8 Calling unbound get_size in Python 3

```
>>> Pizza.get_size()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: get_size() missing 1 required positional argument: 'self'
```

Python 2 rejects the method call because it's unbound; Python 3 permits the call, but complains that we haven't provided the necessary `self` argument. This makes Python 3 a bit more flexible: not only can we pass an arbitrary instance of the class to the method if we want to, but we can pass *any* object as long as it has the properties that the method expects to find:

```
>>> Pizza.get_size(Pizza(42))
42
```

And it works, just as promised, though it's 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`, and better still, Python will automatically pass the object itself to the method's `self` parameter:

Example 7.9 Calling bound `get_size`

```
>>> Pizza(42).get_size
<bound method Pizza.get_size of <__main__.Pizza object at 0x7f3138827910>>
>>> Pizza(42).get_size()
42
```

As expected, we don't 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's a even better example:

```
>>> m = Pizza(42).get_size
>>> m()
42
```

You don't even have to keep a reference to your `Pizza` object as long as you have a reference to the bound method. And if you have a reference to a method but you want to find out which object it's bound to, you can just check the method's `__self__` property:

```
>>> 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 back if we want.

7.3 Static methods

Static methods are methods which belong to a class, but don't actually operate on class instances. For example:

Example 7.10 `@staticmethod` usage

```
class Pizza(object):
    @staticmethod
    def mix_ingredients(x, y):
        return x + y

    def cook(self):
        return self.mix_ingredients(self.cheese, self.vegetables)
```

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. The `@staticmethod` decorator gives us several things:

- Python doesn't have to instantiate a bound method for each `Pizza` object we create. Bound methods are objects, too, and creating them has a cost. Using a static method lets us avoid that:

```
>>> Pizza().cook is Pizza().cook
False
>>> Pizza().mix_ingredients is Pizza.mix_ingredients
True
>>> Pizza().mix_ingredients is Pizza().mix_ingredients
True
```

- It improves the readability of the code: when we see `@staticmethod`, we know that the method does not depend on the state of the object.

- We can override our static methods in subclasses. If 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.

7.4 Class method

Class methods are methods that are bound directly to a class rather than its instances:

```
>>> 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
```

However you choose to access this method, it will be always bound to the class it is attached to, and its first argument will be the class itself (remember, classes are objects too!)

Class methods are mostly useful for creating *factory methods* – methods which instantiate objects in a specific fashion. If we used a `@staticmethod` instead, we would

have to hard-code the `Pizza` class name in our method, making any class inheriting from `Pizza` unable to use our factory for its own purposes.

```
class Pizza(object):
    def __init__(self, ingredients):
        self.ingredients = ingredients

    @classmethod
    def from_fridge(cls, fridge):
        return cls(fridge.get_cheese() + fridge.get_vegetables())
```

In this case, 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 will return a brand-new `Pizza` with ingredients taken from what's available in `myfridge`.

7.5 Abstract methods

An abstract method is a method defined in a base class which may or may not actually provide any implementation. The simplest way to write an abstract method in Python is:

```
class Pizza(object):
    @staticmethod
    def get_radius():
        raise NotImplementedError
```

Any class inheriting from `Pizza` should implement and override the `get_radius` method; otherwise, calling the method will raise an exception.

This particular way of implementing abstract methods has a drawback: if you write a class that inherits from `Pizza` and forget to implement `get_radius`, the error will only be raised if you try to use that method at runtime.

Example 7.11 Implementing an abstract method

```
>>> 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
```

If you implement your abstract methods using Python's built-in abc module instead, you'll get an early warning if you try to instantiate an object with abstract methods:

Example 7.12 Implementing an abstract method using `abc`

```
import abc


class BasePizza(object):
    __metaclass__  = abc.ABCMeta


    @abc.abstractmethod
    def get_radius(self):
         """Method that should do something."""
```

When you use `abc` and its special class, if you try to instantiate a `BasePizza` or a class inheriting from it that doesn't override `get_radius`, you'll get 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
```

Note

The metaclass declaration changed between Python 2 and Python 3. The previous examples only work with Python 2 for this reason.

7.6 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. Here are some tips that will help you with that.

An abstract method's prototype isn't set in stone. When you actually implement the method, there's nothing stopping you from extending the argument list as you see fit:

```
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]
```

We can define `Calzone`'s methods any way we like, as long as they still support the interface we define in the `BasePizza` class. This includes implementing them as class or static methods:

```
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 still supports our abstract `BasePizza` class's interface, so it's still valid.

Starting with Python 3 (this won't work as expected in Python 2; see issue 5867), it's also possible to use the `@staticmethod` and `@classmethod` decorators on top of `@abstractmethod`:

Example 7.13 Mixing `@classmethod` and `@abstractmethod`

```
import abc

class BasePizza(object):
    __metaclass__  = abc.ABCMeta

    ingredients = ['cheese']

    @classmethod
```

```
    @abc.abstractmethod
    def get_ingredients(cls):
         """Returns the ingredient list."""
         return cls.ingredients
```

Note that defining `get_ingredients` as a class method in `BasePizza` like this doesn't force its subclasses to define it as a class method as well. The same would apply if we'd defined it as a static method: there's no way to force subclasses to implement abstract methods as a specific kind of method.

But wait – here we have an implementation *in* an abstract method. Can we *do* that? Yep – Python doesn't have a problem with it! Unlike Java, you can put code in your abstract methods and call it using `super()`:

Example 7.14 Using `super()` with abstract methods

```
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()
```

In this example, every `Pizza` you make that inherits from `BasePizza` will have to override the `get_ingredients` method, but it will have access to the base class's default mechanism for getting the ingredients list.

7.7 The truth about `super`

From the earliest days of Python, developers have been able to use both single and multiple inheritance to extend their classes. However, many developers don't seem to understand how these mechanisms actually work, and the associated `super()` method that is associated with it.

There is pros and cons of single and multiple inheritance, composition or even duck typing would be out of topic for this book, though if you are not familiar with these notions I suggest that you read about them to have a view – and build your own opinion.

Multiple inheritance is still used in many places, and especially in code where the mixin pattern is involved. That's why it's still important to know about it, and because it is part of Python's core.

Note

A mixin is a class that inherits from two or more other classes, combining their features together.

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 part in parentheses is a Python expression that returns the list of class objects to be used as the class's parents. Normally you'd specify them directly, but you could also write something like:

```
>>> def parent():
...     return object
...
>>> class A(parent()):
...     pass
...
>>> A.mro()
[<class '__main__.A'>, <type 'object'>]
```

And it works as expected: class `A` is defined with `object` as its parent class. The class method `mro()` returns the *method resolution order* used to resolve attributes. The current MRO system was first implemented in Python 2.3, and its internal workings are described in the Python 2.3 release notes.

You already know that the canonical way to call a method in a parent class is by using 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 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:

```
>>> 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
```

When requesting an attribute of the super object of an instance of `C`, it walks through the MRO list and return the attribute from the first class having it.

In the previous example, we used a bound `super` object; i.e., we called `super` with two arguments. If we call `super()` with only one argument, it returns an unbound `super` object instead:

```
>>> super(C)
<super: <class 'C'>, NULL>
```

Since this object is unbound, you can't use it to access class attributes:

```
>>> 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 kind of `super` object is useless, but the `super` class implements the descriptor protocol (i.e. `__get__`) in a way that 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 and the attribute name as arguments (`super(C).__get__(D(), 'foo')`), allowing it to find and resolve `foo`.

Note

Even if you've never heard of the descriptor protocol, you've probably used it through the `@property` decorator without knowing it. It's the mechanism in Python that allows an object that's stored as an attribute to return something other than itself. This protocol isn't covered in this book, but you can find out more about it in the Python data model documentation.

There are plenty of situations where using `super` can be tricky, such as handling

different method signatures along the inheritance chain. Unfortunately, there's no silver bullet for that, apart from using tricks like having all your methods accept their arguments using `*args, **kwargs`.

In Python 3, `super()` picked up a little 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 them:

```
class B(A):
      def foo(self):
          super().foo()
```

`super` is the standard way of accessing parent attributes in subclasses, 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 using multiple inheritance.

8 Functional programming

Functional programming might not be the first thing you think of when you think of Python, but the support is there, and it's quite extensive. Many Python developers don't seem to realize this, though, which is a shame: with few exceptions, functional programming allows you to write more concise and efficient code.

When you write code using functional style, your functions are designed not to have side effects: 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*:

A non-pure function

```
def remove_last_item(mylist):
    """Removes the last item from a list."""
    mylist.pop(-1)  # This modifies mylist
```

A pure function

```
def butlast(mylist):
    """Like butlast in Lisp; returns the list without the last element."""
    return mylist[:-1]  # This returns a copy of mylist
```

The practical advantages of functional programming include:

- **Formal provability**; admittedly, this is a pure theoretical advantages, nobody is going to mathematically prove a Python program.

- **Modularity**; writing functionally forces a certain degree of separation in solving your problems and eases reuse in other contexts.
- **Brevity.** Functional programming is often less verbose than other paradigms.
- **Concurrency.** Purely functional functions are thread-safe and can run concurrently. While it's not yet the case in Python, some functional languages do this automatically, which can be a big help if you ever need to scale your application.
- **Testability.** It's a simple matter to test a functional program: all you need is a set of inputs and an expected set of outputs. They are idempotent.

Tip

If you want to get serious about functional programming, take my advice: take a break from Python and learn Lisp. I know it might sound strange to talk about Lisp in a Python book, but playing with Lisp for several years is what taught me how to "think functional." You simply won't 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 there's more focus on functional programming than you'll find in Python.

8.1 Generators

A generator is an object that returns a value on each call of its `next()` method until it raises `StopIteration`. They were first introduced in PEP 255 and offer an easy way to create objects that implement the iterator protocol.

All you have to do to create a generator is write a normal Python function that contains a `yield` statement. Python will detect the use of `yield` and tag the function as a generator. When the function's execution reaches a `yield` statement, it returns a value as with a `return` statement, but with one notable difference: the interpreter

will save a stack reference, which will be used to resume the function's execution the next time `next` is called.

Creating a generator

```
>> 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
```

You can check whether a function is a generator or not yourself by using `inspect.isgeneratorfunction`:

```
>>> import inspect
>>> def mygenerator():
...     yield 1
...
>>> inspect.isgeneratorfunction(mygenerator)
True
```

```
>>> inspect.isgeneratorfunction(sum)
False
```

Reading the source code of `inspect.isgeneratorfunction` gives us some insight into the tagging mentioned earlier:

Source code of inspect.isgeneratorfunction

```
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)
```

Python 3 provides another useful function, `inspect.getgeneratorstate`:

```
>>> import inspect
>>> def mygenerator():
...     yield 1
...
>>> gen = mygenerator()
>>> gen
<generator object mygenerator at 0x7f94b44fec30>
>>> inspect.getgeneratorstate(gen)
'GEN_CREATED'
>>> next(gen)
1
>>> inspect.getgeneratorstate(gen)
'GEN_SUSPENDED'
>>> next(gen)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> inspect.getgeneratorstate(gen)
'GEN_CLOSED'
```

This function gives us the current state of a generator, allowing us to determine whether it's waiting to be run for the first time (`GEN_CREATED`), currently being executed by the interpreter (`GEN_RUNNING`), waiting to be resumed by a call to `next()` (`GEN_SUSPENDED`), or finished running (`GEN_CLOSED`).

In Python, generators are built by keeping a reference of the stack when a function *yield* something, resuming this stack when needed, i.e. when a call to `next()` is executed again.

When you iterate over any kind of data, the obvious approach is to build the entire list first, which is often wasteful in terms of memory consumption. 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 128 MB:

```
$ ulimit -v 131072
$ python
>>> a = list(range(10000000))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
```

Uh-oh. Turns out we can't build a list of ten million items with only 128MB of memory!

Warning

In Python 3, `range()` returns a generator; to get a generator in Python 2, you have to use `xrange()` instead. (This function doesn't exist in Python 3, since it's redundant.)

Let's try using a generator instead:

```
$ python
>>> for value in xrange(10000000):
...     if value == 50000:
...             print("Found it")
...             break
...
Found it
```

This time, our program executes without issue. The `range()` function returns an iterable object that dynamically generates our list of integers. Better still, since we were only interested in the 50,000th number, the generator only had to generate 50,000 numbers.

Generators allow you to handle large data sets with minimal consumption of memory and processing cycles by generating values on-the-fly. Whenever you need to work with a huge number of values, generators can help ensure you handle them efficiently.

`yield` also has a less commonly used feature: it can return a value like a function call. This allows us to pass a value to a generator by calling its `send()` method:

Example 8.1 `yield` returning a value

```
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
```

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 string is determined by 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.

Tip

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']
```

8.2 List comprehensions

List comprehension, or *listcomp* for short, allows you to define a list's contents inline with its declaration:

Without list comprehension

```
>>> x = []
>>> for i in (1, 2, 3):
...     x.append(i)
...
>>> x
[1, 2, 3]
```

With list comprehension

```
>>> x = [i for i in (1, 2, 3)]
>>> x
[1, 2, 3]
```

You can use multiple `for` statements together and use `if` statements to filter out items:

```
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']
```

Using list comprehension rather than *for* loops is a neat way to quickly define lists. Since we're still talking about functional programming, it's worth noting that lists built through list comprehension can't rely on the program's state. [1] This generally makes them more concise and easier to read than lists made without list comprehension.

Note

There's also syntax for building dictionaries or sets in the same fashion:

```
>>> {x:x.upper() for x in ['hello', 'world']}
{'world': 'WORLD', 'hello': 'HELLO'}
>>> {x.upper() for x in ['hello', 'world']}
set(['WORLD', 'HELLO'])
```

Note that this only works in Python 2.7 and onward.

8.3 Functional functions functioning

Python includes a number of tools for functional programming. These built-in functions cover the basics:

- `map(function, iterable)` applies `function` to each item in `iterable` and returns either a list in Python 2 or an iterable `map` object in Python 3:

[1] Technically they *can,* but that's really not how they're supposed to work.

map usage in Python 3

```
>>> 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!"]
```

- `filter(function or None, iterable)` filters the items in `iterable` based on the result returned by `function`, and returns either a list in Python 2, or better, an iterable `filter` object in Python 3:

Example 8.2 *filter* usage 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']
```

Tip

You can write a function equivalent to `filter` or `map` using generators and list comprehension:

Equivalent of map using list comprehension

```
>>> (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!"]
```

Equivalent of filter using list comprehension

```
>>> (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']
```

Using generators like this in Python 2 will give you an iterable object rather than a list, just like the `map` and `filter` functions in Python 3.

- `enumerate(iterable[, start])` returns an iterable `enumerate` object that yields a sequence of tuples, each consisting of an integer index (starting with `start`, if provided) and the corresponding item in `iterable`. It's 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 write this:

```
for i, item in enumerate(mylist):
    print("Item %d: %s" % (i, item))
```

- `sorted(iterable, key=None, reverse=False)` returns a sorted version of `iterable`. The `key` argument allows you to provide a function that returns the value to sort on.
- `any(iterable)` and `all(iterable)` both return a boolean depending on the values returned by `iterable`. These functions are equivalent to:

```
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:

```
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")
```

- `zip(iter1 [,iter2 [...]])` takes multiple sequences and combines them into tuples. It's useful when you need to combine a list of keys and a list of values into a dict. Like the other functions described above, it returns a list in Python 2 and an iterable in Python 3:

```
>>> 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}
```

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 to Python 2, don't despair yet: the `itertools` module from the standard library provides an iterator based version of many of these functions (`itertools.izip`, `itertoolz.imap`, `itertools.ifilter`, etc).

There's still one important tool missing from this list, however. One common task when working with lists is finding the first item that satisfies a specific condition. This is usually accomplished with a function like this:

```
def first_positive_number(numbers):
    for n in numbers:
        if n > 0:
            return n
```

We can also write this in functional style:

```
def first(predicate, items):
    for item in items:
```

```
        if predicate(item):
            return item


first(lambda x: x > 0, [-1, 0, 1, 2])
```

Or more concisely:

```
# Less efficient
list(filter(lambda x: x > 0, [-1, 0, 1, 2]))[0] ❶
# Efficient but for Python 3
next(filter(lambda x: x > 0, [-1, 0, 1, 2]))
# Efficient but for Python 2
next(itertools.ifilter(lambda x: x > 0, [-1, 0, 1, 2]))
```

❶ Note that this may raise an IndexError if no items satisfy the condition, causing `list(filter())` to return an empty list.

Instead of writing this same function in every program you make, you can include the small but very useful Python package **first**:

Example 8.3 Using `first`

```
>>> 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
```

The `key` argument can be used to provide a function which receives each item as an argument and returns a boolean indicating whether it satisfies the condition.

You'll notice that we've used `lambda` in a good portion of the examples so far in this chapter. `lambda` was actually added to Python in the first place 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:

```
import operator
from first import first


def greater_than_zero(number):
    return number > 0


first([-1, 0, 1, 2], key=greater_than_zero)
```

This code works identically to the previous example, but it's a good deal more cumbersome: if we wanted to get the first number in the sequence that's greater than, say, 42, then we'd need to `def` an appropriate function rather than defining it in-line with our call to `first`.

But despite its usefulness in helping us avoid situations like this, `lambda` still has its problems. First and most obviously, we can't pass a `key` function using `lambda` if it would require more than a single line of code. In this event, we're back to the cumbersome pattern of writing new function definitions for each `key` we need. Or are we?

`functools.partial` is our first step towards replacing `lambda` with a more flexible alternative. It 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:

```
from functools import partial
from first import first


def greater_than(number, min=0):
```

```
    return number > min


first([-1, 0, 1, 2], key=partial(greater_than, min=42))
```

Our new `greater_than` function works just like the old `greater_than_zero` by default, but now we can specify the value we want to compare our numbers to. In this case, we pass `functools.partial` our function and the value we want for `min`, and we get back a new function that has `min` set to 42, just like we want. In other words, we can write a function and use `functools.partial` to customize what it does to our needs in any given situation.

This is still a couple lines more than we strictly need in this case, though. All we're doing in this example is comparing two numbers; what if Python had built-in functions for these kinds of comparisons? As it turns out, the **operator** module has just what we're looking for:

```
import operator
from functools import partial
from first import first


first([-1, 0, 1, 2], key=partial(operator.le, 0))
```

Here we see that `functools.partial` also works with positional arguments. In this case, `operator.le(a, b)` takes two numbers and returns whether the first is less than or equal to the second: the 0 we pass to `functools.partial` gets sent to `a`, and the argument passed to the function returned by `functools.partial` gets sent to `b`. So this works identically to our initial example, without using `lambda` or defining any additional functions.

Note

`functools.partial` is typically useful in replacement of lambda, and is to be considered as a superior alternative. `lambda` is to be considered an anomaly in Python language [a], due to its limited body size of one line long single expression. On the other hand, `functools.partial` is built as a nice wrapper around the original function.

[a] And was once even planned to be removed in Python 3, but finally escaped from its fate.

The **itertools** module in the Python Standard Library also provides a bunch of useful functions that you'll want to keep in mind. I've seen too many programmers end up writing their own versions of these functions even though Python itself provides them out-of-the-box:

- `chain(*iterables)` iterates over multiple iterables one after each other without building an intermediate list of all items.
- `combinations(iterable, r)` generates all combination 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 from `start` and incrementing by `step` with each call.
- `cycle(iterable)` loops repeatedly over the values in `iterable`.
- `dropwhile(predicate, iterable)` filters elements of an iterable starting from the beginning until `predicate` is false.
- `groupby(iterable, keyfunc)` creates an iterator grouping 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:

Example 8.4 Using the `operator` module with `itertools.groupby`

```
>>> 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 list(itertools.groupby(a,  ↩
   operator.itemgetter('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.

9 The AST

AST stands for *Abstract Syntax Tree*. It is a tree representation of the abstract structure of the source code of any programming language, including Python. Python as its own AST that is built upon parsing a Python source file.

This area of Python is not heavily documented, and not easy to deal with at first glance. Still, its is very interesting to know and understand some deeper construction of Python as a programming language to masterize its usage.

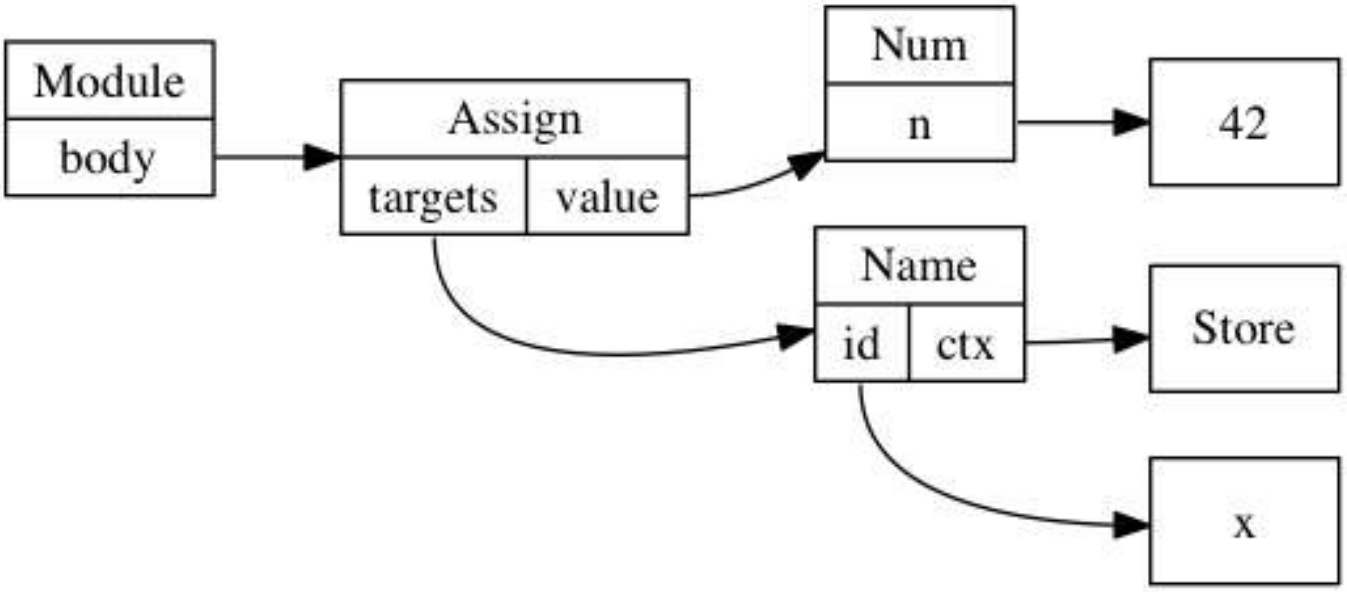
The easiest way to have a view of what the Python AST looks like is to parse a Python code and dumps the generated AST. To do that, the Python `ast` module provides everything you need for.

Example 9.1 Parsing Python code to AST

```
>>> 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)) ↩
   ])"
```

The `ast.parse` function returns a `_ast.Module` object that is the root of the tree.

The tree can be entirely dumped using the `ast.dump` module, and in this case is the following:



An AST construction always starts with a root element, which is usually an `ast.Module` object. This object contains a list of statements or expressions to evaluate in its `body` attribute. It usually represents the content of a file.

As you can guess, the `ast.Assign` object represents an assignment, that is mapped to the = sign in the Python syntax. Assign has a list of targets, and a value it assignates to it. The list of target in this case consists of one object, `ast.Name`, which represents a variable named `x`. The value is a number with value being 42.

This AST can be passed to Python to be compiled and then evaluated. The `compile` function provided as a Python built-in allows that.

```
>>> 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
```

An abstract syntax tree can be built manually using the classes provided in the `ast`

module. Obviously, this is a very long way to write Python code, not a method I would recommend! But it's still interesting to use.

Let's write a good old "Hello world!" in Python using the AST.

Example 9.2 Hello world using Python AST

```
>>> hello_world = ast.Str(s='hello world!', lineno=1, col_offset=1)
>>> print_call = ast.Print(values=[hello_world], lineno=1, col_offset=1, nl ↩
   =True)
>>> module = ast.Module(body=[print_call])
>>> code = compile(module, '', 'exec')
>>> eval(code)
hello world!
```

Note

`lineno` and `col_offset` represents the line number and column offset of the source code that has been used to generate the AST. This doesn't have much sense to set them in this context since we are not parsing any source file, but it's useful to find back the position of the code that generated this AST. It's for example used by Python when generating backtraces. Anyway, Python refused to compile any AST object that doesn't provide this information, this is why we pass it fake values of 1 here. The `ast.fix_missing_locations()` function can fix it for you by setting the missing values to the ones set on the parent node.

The whole list of objects that are available in the AST is easily available by reading the `_ast` module documentation (note the underscore).

The first two categories you should consider are statement and expressions. Statements cover types like *assert*, *assign* (=), augmented assigned (+=, /=, etc), *global*, *def*, *if*, *return*, *for*, *class*, *pass*, *import*, etc. They all inherit from `ast.stmt`. Expressions cover types like *lambda*, *number*, *yield*, *name* (variable), *compare* or *call*. They all inherit from `ast.expr`.

There's also a few other categories, such as `ast.operator` defining standard operator such as *add* (+), *div* (/), *right shift* (>>), etc, or `ast.cmpop` defining comparisons operator.

You can easily imagine that it is then possible to leverage this AST to construct a compiler that would parse strings and generate code by building a Python AST. This is exactly what led to the Hy project discussed in Section 9.1.

In case you need to walk through your tree, the `ast.walk` function will help you with that. But the `ast` module also provides `NodeTransformer`, a class that can be subclassed to walk an AST to modify some nodes. It's therefore easy to use it to change code dynamically.

Example 9.3 Changing all binary operation to addition

```
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)


tree = ast.parse("x = 1/3")
ast.fix_missing_locations(tree)
eval(compile(tree, '', 'exec'))
print(ast.dump(tree))
print(x)
tree = ReplaceBinOp().visit(tree)
ast.fix_missing_locations(tree)
print(ast.dump(tree))
eval(compile(tree, '', 'exec'))
```

```
print(x)
```

Which executes to the following:

```
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
```

Tip

If you need to evaluate a string of Python that should return a simple data type, you can use `ast.literal_eval`. Contrary to `eval`, it disallows the input string to execute any code. It's a safer alternative to `eval`.

9.1 Hy

Now that you know about the AST, you can easily dream of creating a new syntax for Python that you would parse and compile down to a standard Python AST. The Hy programming language is doing exactly that. It is a Lisp dialect that parses a Lisp like language and converts it to regular Python AST. It is therefore fully compatible with the Python ecosystem. You could compare it to what Clojure is to Java. Hy could deserve a book for itself, so we will only fly over it in this section.

If you already wrote Lisp [1], the Hy syntax will really look familiar. Once installed, launching the `hy` interpreter will give you a standard REPL prompt where you can start interact with the interpreter.

[1] If not, you should consider it.

```
% hy
hy 0.9.10
=> (+ 1 1)
2
```

For those not familiar with the Lisp syntax, the parentheses denote a list, the first element is a function, and the rest of the list are the arguments. Here the code is equivalent to Python `1 + 1`.

Most constructs are mapped from Python directly, such as function definition. Setting a variable relies on the `setv` function.

```
=> (defn hello [name]
...  (print "Hello world!")
...  (print (% "Nice to meet you %s" name)))
=> (hello "jd")
Hello world!
Nice to meet you jd
```

Internally, *Hy* parses the code that is provided and compiles it down to Python AST. Luckily, Lisp is an easy to parse tree, as each pair of parentheses represents a node of the list tree. All is needed to be done is to convert this Lisp tree to a Python abstract syntax tree.

Class definition is supported through the `defclass` construct, that is inspired from CLOS [2].

```
(defclass A [object]
  [[x 42]
   [y (fn [self value]
        (+ self.x value))]])
```

[2] Common Lisp Object System

This defines a class named *A*, inheriting from `object`, with a class attribute `x` whose value is 42 and a method `y` that returns the `x` attribute plus the 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.

```
=> (import uuid)
=> (uuid.uuid4)
UUID('f823a749-a65a-4a62-b853-2687c69d0e1e')
=> (str (uuid.uuid4))
'4efa60f2-23a4-4fc1-8134-00f5c271f809'
```

Hy also has more advanced construct and macros. If you ever wanted to have a case or switch statement in Python, admire what `cond` can do for you:

```
(cond
 ((> somevar 50)
  (print "That variable is too big!"))
 ((< somevar 10)
  (print "That variable is too small!"))
 (true
  (print "That variable is jusssst right!")))
```

Hy is a very nice project that allows you to jump into Lisp world without leaving your comfort zone too far behind you, as you are still writing Python. The `hy2py` tool can even show you what your Hy code would look like once translated into Python [3].

9.2 Interview with Paul Tagliamonte

Paul is a Debian developer, who's working at Sunlight Foundation. He created Hy in 2013 and, as a Lisp lover, I joined him in this fabulous adventure some time later.

[3] Though it has some restrictions.

Why did you create Hy in the first place?

Initially, I created Hy following a conversation about how someone should write 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 – something which resembled a lisp, and even worked like a proper lisp, but it was slow. I mean, really slow. It took about an order of magnitude slower than native Python, since the Lisp runtime itself was implemented in Python.

Frustrated, I almost gave up, only to be pushed forward by a coworker the promise of using AST to implement the runtime, rather than implement the runtime in Python. This insane idea started to really spark the entire project. This set in shortly before the holidays in 2012, leading me to spend my entire break from work hacking on Hy. A week or so later, and I ended up with something that resembled the current Hy codebase quite closely – most Hy devs would even know their way around the compiler.

Just after getting enough working to implement a basic Flask app, I gave a talk at Boston Python about this project, and the reception was incredibly warm – so warm, in fact, that I'd started to view Hy as a good way to teach people about Python internals, such as how the REPL works [4], PEP 302 import hooks, and Python AST – a good introduction to the concept of code that writes code.

After the talk, I was a bit disappointed in a few sections, so 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!

[4] *code.InteractiveConsole*

In addition, Hy (the Language) is a good way to get people to understand how to read Lisp, since they can get comfortable with s-expressions in an environment they know (even using libraries they have lying around), easing the transition to other ("real") Lisps, such as Common Lisp, Scheme or Clojure, as well as experiment with new ideas (such as macro systems, homoiconicity, and working without the concept of a statement).

How did you find out about using the AST correctly? What are the tips and tricks, advice you can give to people looking at it?

Python's AST is quite interesting. It's not quite private (in fact, it's explicitly not private), but it's also 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 Python AST in the same way.

For instance, CPython can deal with slightly out of order AST entries (by the *lineno* and *col_offset*), whereas PyPy will throw an assertion error. While sometimes annoying, the AST is generally sane. It's not impossible to build AST that works on a vast number of Python instances. With a conditional or two, it's only mildly annoying to create AST that works on CPython 2.6 through 3.3 and PyPy, making this tool quite handy.

The AST is extremely under-documented, so most knowledge comes from reverse engineering generated AST. By writing up simple Python scripts, one can use something similar to `import ast;ast.dump(ast.parse("print foo"))` to generate 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.

What's the current status, and future goals of Hy?

Hy is currently in development. It has a few subtle issues that need to be addressed, and fixing the bugs to make Hy virtually indistinguishable from any other LISP-1 variant. This is a monumental task, but it's one that it's ripe for hacking.

I'm also interested in keeping Hy efficient, in so far as it can be.

I hope, in the long run, that Hy will become a sort of teaching tool – one way to explain some of the concepts that are quite foreign to even experienced Pythonistas. I hope it also proves interesting enough to Pythonistas that they take an interest in these tools at our disposal, and continue pushing the bounds of what I think Hy is.

My hope is that people see Hy for what it is – an amazing teaching tool. A way to get people interested in Common Lisp, Clojure or Scheme. I want people to go home and read about why Lisp variants do things the way they do, and how they can borrow this philosophy in their day-to-day coding.

How interoperable with Python is Hy? What about code distribution and packaging?

Amazingly interoperable. Stunningly interoperable, really. So well, in fact, that *pdb* can properly debug Hy without any changes at all. To really drive this point home, I've written Flask apps, Django apps and modules of all sorts. Python can import Python, Hy can import Hy, Hy can import Python and Python can import Hy. This is what really makes Hy unique – even variants like Clojure can't do this, the interop is purely unidirectional (Clojure can import Java, but Java has one hell of a time importing Clojure). This was done to really bring home how powerful these tools we have are.

Hy works by translating Hy code (in s-expressions) into Python AST almost directly. This compilation step means the generated bytecode is fairly sane stuff (so much so that debugging Hy by looking at Python source generated from Python AST is a good way of tracking down pesky AST errors), 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 to a Python equivalent (in this case, `*earmuffs*` becomes `EARMUFFS`, and `using-dashes` becomes `using_dashes`), which means Python doesn't have a hard time of 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 upside and downside of choosing Hy over Python?

This is an interesting question. I'm quite partial, so take this with a grain of salt!

Hy outshines Python in a few special ways because we've taken a bit of effort to smooth behavior over Python versions to allow the new Python 3 future happen sooner. This was done by doing things like using future division in Python 2, and ensuring the syntax is normalized between the two versions.

In addition, Hy has something Python has a very hard time with (even with the outstanding AST module), which is a full macro system. Macros are very special functions that alter the code during it's compile step – not unlike having `ast.NodeVisitor` as a first-class function of the language. This leads to easy creation of new domain-specific languages, which is composed of the base language (in this case, Hy / Python), with the addition of many macros which allow uniquely expressive and succinct code.

Often times, clever DSLs can replace languages designed to perform this role, such as Lua.

As for downsides, what gives Hy it's power can also hurt it. Not technically, but socially. 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 due to the fear of Hy being extremely complex.

Hy is the Lisp everyone loves to hate – Python folks tend to not enjoy its syntax, and Lispers tend to avoid Hy due to, well, being Python. Hy uses Python objects directly, so the behavior of fundamental objects can sometimes be surprising to the seasoned Lisper.

Hopefully people will look past it's syntax and consider using it for a project to expand one's horizons, and explore parts of Python previously untouched.

10 Performances and optimizations

> Premature optimization is the root of all evil.
>
> --- Donald Knuth *Structured Programming with go to Statements*

10.1 Data structures

Most computer problems can be solved in an elegant and simple manner, provided that you use the right data structures – and Python provides many data structures to choose from.

Often, there is a temptation to code your own custom data structures – this is invariably a vain, useless, doomed idea. Python almost always has better data structures and code to offer – learn to use them.

For example, everybody uses `dict`, but how many times have you seen code like this:

```
def get_fruits(basket, fruit):
    # A variation is to use "if fruit in basket:"
    try:
        return basket[fruit]
    except KeyError:
```

```
        return set()
```

It's much more easy to use the `get` method already provided by the `dict` structure:

```
def get_fruits(basket, fruit):
    return basket.get(fruit, set())
```

It's not uncommon for people to use basic Python data structures without being aware of all the methods they provide. This is also true for sets – for example:

```
def has_invalid_fields(fields):
    for field in fields:
        if field not in ['foo', 'bar']:
            return True
    return False
```

This can be written without a loop:

```
def has_invalid_fields(fields):
    return bool(set(fields) - set(['foo', 'bar']))
```

The `set` data structures have methods which can solve many problems that would otherwise need to be addressed by writing nested for/if blocks.

There are also more advanced data structures that can greatly reduce the burden of code maintenance. For example, take a look at the following code:

```
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')
```

Sure, this code is perfectly valid, but how many times will your program require a variation of the above? 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 that you try to access a non-existent item from your *dict*, the `defaultdict` will use the function that was passed as argument to its constructor to build a new value – instead than raising a `KeyError`. In this case, the `set` function is used to build a new *set* each time we need it.

By the way, the `collections` module offers a few useful data structures that can solve other kinds of problems, such as `OrderedDict` or `Counter`.

It's really important to look for the right data structure in Python, as the correct choice will save you time, and lessen code maintenance.

10.2 Profiling

Python provides a few tools to profile your program. The standard one is `cProfile` and is easy enough to use.

Example 10.1 Using the `cProfile` module

```
$ 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)
```

The results list indicates the number of calls each function was called, and the time spent on its execution. You can use the `-s` option to sort by other fields; e.g. `-s time` will sort by internal time.

If you've coded in C, as I did years ago, you probably already know the fantastic Valgrind tool, that – among other things – is able to provide profiling data for C programs. The data that it provides can then be visualized by another great tool named KCacheGrind.

You'll be happy to know that the profiling information generated by *cProfile* can easily be converted to a call tree that can be read by *KCacheGrind*. The `cProfile` module has a `-o` option that allows you to save the profiling data, and pyprof2calltree can convert from one format to the other.

Example 10.2 Using *KCacheGrind* to visualize Python profiling data

```
$ python -m cProfile -o myscript.cprof myscript.py
$ pyprof2calltree -k -i myscript.cprof
```

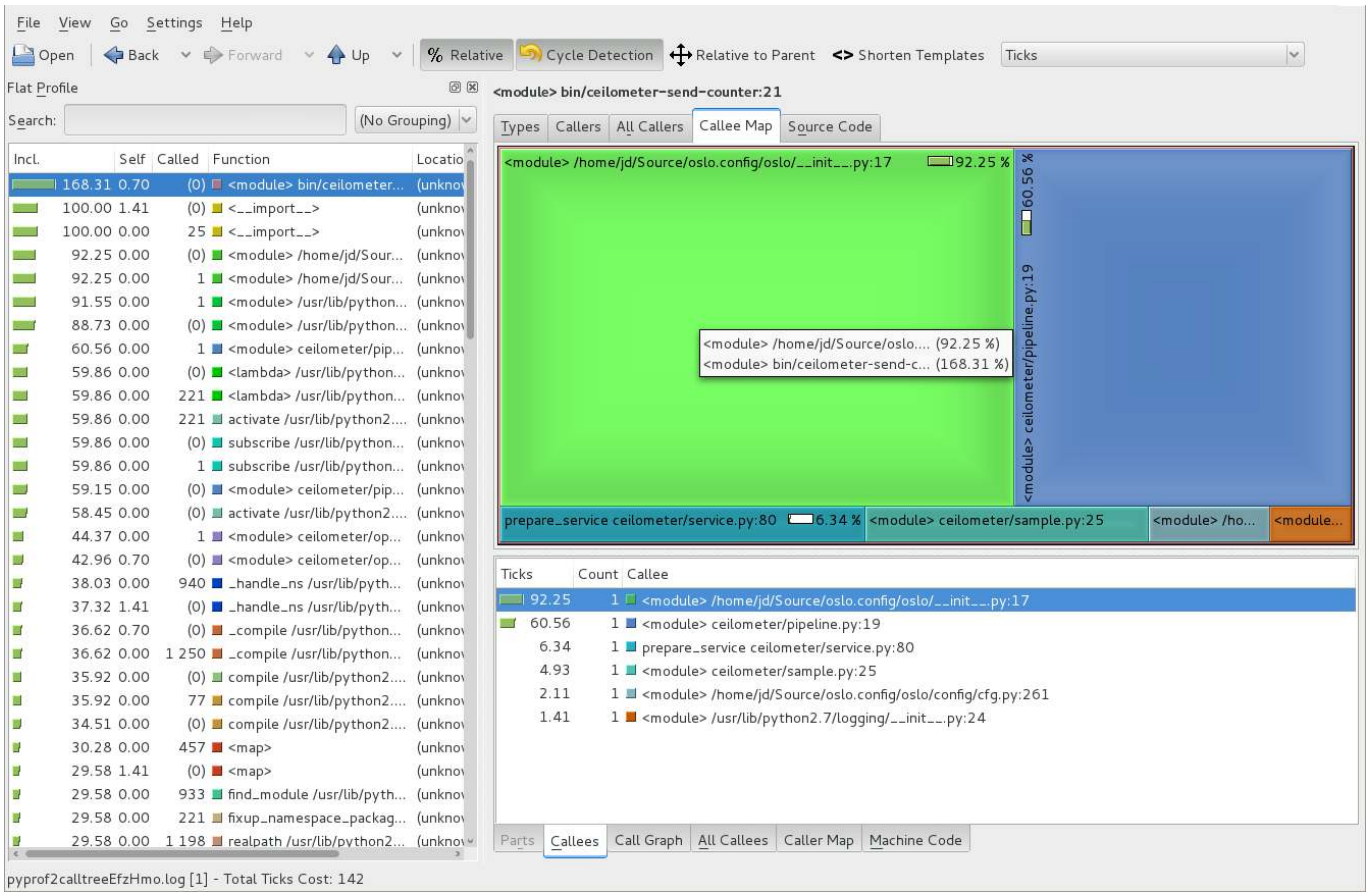


Figure 10.1: KCacheGrind example

This provides a lot of information that will allow you to determine what part of your program might be consuming too much resources.

While this clearly works well for a macroscopic view of your program, it sometimes helps to have a microscopic view of some part of the code. In such a context, I find it better to rely on the `dis` module to find out what's going on behind the scenes. The `dis` module is a disassembler of Python byte code. It's simple enough to use:

```
>>> def x():
...     return 42
...
>>> import dis
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (42)
```

```
              3 RETURN_VALUE
```

The `dis.dis` function disassembles the function that you passed as a parameter, and prints the list of bytecode instructions that are run by the function. It can be useful to understand what's really behind each line of code that you write, in order to be able to properly optimize your code.

The following code defines two functions, each of which does the same thing – concatenates three letters:

```
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 appear to do exactly the same thing, but if we disassemble them, 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
```

As you can see, in the second version we store `abc[0]` in a temporary variable be-

fore running the loop. This makes the bytecode executed inside the loop a little smaller, as we avoid having to do the `abc[0]` lookup for each iteration. Measured using `timeit`, the second version is 10% faster than the first one; it takes a whole microsecond less to execute! Obviously this microsecond is not worth the optimization unless you call this function millions of times – but this is kind of insight that the `dis` module can provide.

Whether you should need to rely on such "tricks" as storing the value outside the loop is debatable – ultimately, it should be the compiler's work to optimize this kind of thing. On the other hand, as the language is heavily dynamic, it's difficult for the compiler to be sure that optimization wouldn't result in negative side effects. So be careful when writing your code!

Another wrong habit I've often encountered when reviewing code is the defining of functions inside functions for no reason. This has a cost – as the function is going to be redefined over and over for no reason.

Example 10.3 A function defined in a function, disassembled

```
>> 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 0x100ce7e30,  ↩
```

```
    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
```

We can see here that it is needlessly complicated, calling `MAKE_FUNCTION`, `STORE_FAST`, `LOAD_FAST` and `CALL_FUNCTION` instead of just `LOAD_CONST`. That requires many more opcodes for no good reason – and function calling in Python is already inefficient.

The only case in which it is required 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.

Example 10.4 Disassembling a closure

```
>>> 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 0x100d139b0,  ↩
```

```
                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
```

10.3 Ordered list and bisect

When manipulating large lists, the use of sorted lists has a few advantages over non-sorted lists – for example, sorted lists have a retrieve time of *O(log n)*.

A couple of times, however, I've seen people trying to implement their own data structures and algorithms to handle such cases. This is a bad idea – you shouldn't spend time on problems already solved.

Firstly, Python provides a `bisect` module which contains a bisection algorithm. It's easy enough to use:

Example 10.5 Usage of `bisect`

```
>>> 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
```

The `bisect` function allows you to retrieve the index where a new list element should be inserted, while keeping the list sorted.

If you wish to insert the element immediately, the `bisect` module provides the `insort_left` and `insort_right` functions that do exactly that.

Example 10.6 Usage of `bisect.insort`

```
>>> 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']
```

You can then use these functions to create a list that is always sorted:

Example 10.7 A SortedList implementation

```
import bisect


class SortedList(list):
    def __init__(self, iterable):
        super(SortedList, self).__init__(sorted(iterable))

    def insort(self, item):
        bisect.insort(self, item)
```

```
    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)
```

Obviously, one shouldn't use the direct functions `append` or `extend` on this list – or the list will no longer be sorted.

Many Python libraries exist which implement various versions of the above code – and many more data types, such as binary or red-black tree structures. The `blist` and `bintree` Python packages contain code that you can be use for these purposes, rather than implementing and debugging your own version.

10.4 Namedtuple and slots

Sometimes it's useful to have the ability to create very simple objects which only possess a few fixed attributes. A simple implementation would 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: it creates a class which inherits from object. In using this *Point* class, you be instantiating objects.

One property of such *objects* in Python, is that they store all of their attributes inside a dictionary; this dictionary is itself stored in the `__dict__` attribute:

```
>>> p = Point(1, 2)
>>> p.__dict__
{'y': 2, 'x': 1}
>>> p.z = 42
>>> p.z
42
>>> p.__dict__
{'y': 2, 'x': 1, 'z': 42}
```

The advantage is that you can add as many attributes as you want to an object. The drawback, however, is that using a dictionary to store these attributes is quite expensive in terms of memory – you need to store the object, the keys, the value references, etc. It's slow to create and slow to manipulate, with a high memory cost. Consider the following simple class:

```
[source,python]
class Foobar(object):
    def __init__(self, x):
        self.x = x
```

Let's check the memory usage using the `memory_profiler` Python package:

```
$ 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) ]
```

Therefore, it exists a way to use objects without this default behaviour. Classes in Python can define a `__slots__` attribute that will list the only attributes allowed for instances of this class. The power of this is that instead of allocating a whole dictionary object to store all of the object attributes, they can now be stored in a `list` object. If you go through the CPython source code and take a look at the `Objects/typeobject.c` file, it is quite easy to understand what Python does in this case. Here is a cut down version of the function which handles this:

```
static PyObject *
type_new(PyTypeObject *metatype, PyObject *args, PyObject *kwds)
{
    […]
    /* 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.a
        */
        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);
    […]
    /* 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;
    […]
    return (PyObject *)type;
```

As you can see, Python converts the content of `__slots__` into a tuple, then a `list` that it builds and sorts, before converting it back into a tuple to use and store it in the class. This way, Python can retrieve the values quickly, without having to allocate and use an entire dictionary.

It's easy enough to declare such a class:

Example 10.8 A class declaration using `__slots__`

```
class Foobar(object):
    __slots__ = 'x'

    def __init__(self, x):
        self.x = x
```

We can easily compare the memory usage of the two approaches using the `memory_profiler` Python package:

Example 10.9 Memory usage of objects using `__slots__`

```
% 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) ]
```

So it seems that by using the `__slots__` attribute of Python classes, we can halve our memory usage – this means that when creating a large amount of simple objects, the `__slots__` attribute is an effective and efficient choice. However, the technique shouldn't be misused in order to perform static typing or the like. This isn't in the spirit of Python programs.

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.

That's exactly the nature of the **`namedtuple`** class from the **`collection`** module. It allows us to dynamically create a class that will inherit from **`tuple`**, therefore sharing its characteristics – such as being immutable, and having a fixed number of entries. What `namedtuple` provides is the ability to retrieve the tuple elements by referencing a named attribute, rather than just referencing by index.

Example 10.10 Declaring a class using **`namedtuple`**

```
>>> 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]
```

Since a class like this would inherit from `tuple`, we can easily convert it to a list. We can't change or add any attributes on objects of this class, because on one hand it inherits from `tuple`, and also because the `__slots__` value is set to an empty tuple – thereby avoiding the creating of the `__dict__`.

Example 10.11 Memory usage of a class built from `collections.namedtuple`

```
% 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) ]
```

Therefore, usage of the `namedtuple` class factory is as almost as efficient as using an object with `__slots__`, the only difference being that it is compatible with the `tuple` class. It can therefore be passed to many native Python functions and libraries that expect an iterable type as an argument. It also enjoys the various optimizations that exist for tuples [1].

`namedtuple` also provides a few extra methods that, even if prefixed by an underscore, are actually intended to be public. `_asdict` can convert the `namedtuple` to

[1] For example, tuples smaller than `PyTuple_MAXSAVESIZE` (20 by default) will use a faster memory allocator in CPython

a `dict` instance, `_make` allows us to convert an existing iterable object to this class, and `_replace` returns a new instance of the object with some fields replaced.

10.5 Memoization

Memoization is a technique used to speed up function calls by caching their result. The results can be cached only if the function is pure – meaning that it has no side effects or outputs, and that it does not depend on any global state.

A trivial function that can be memoized is the sine function `sin`.

Example 10.12 A basic memoization technique

```
>>> 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}
```

The first time that `memoized_sin` is called with an argument that is not stored in `_SIN_MEMOIZED_VALUES`, the value will be computed and stored in this dictionary. Later on, if we call the function with the same value again, the result will be retrieved from the dictionary rather than computed again. While `sin` is a function which computes very quickly, this may not be true of some advanced functions which involve more complicated computations.

If you've already read about decorators (if not, go to Section 7.1), you must be thinking that there is 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 LRU (Least-Recently-Used) cache decorator. This provides the same functionality as the memoization described here, but with the benefit that it limits the number of entries in the cache, removing the least recently used one when the cache size reaches its maximum size.

The module also provides statistics on cache hits, misses, etc. In my opinion, these are a must-haves when implementing such a cache. There's no point in using memoization – or any caching technique – if you are unable to meter its usage and usefulness.

Here's an example of the `memoized_sin` function above, using `functools.lru_cache`:

Example 10.13 Using `functools.lru_cache`

```
>>> 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)
```

10.6 PyPy

PyPy is an efficient implementation of the Python language which complies with standards. 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 in such a way that a variable's type can be inferred at compile time. The RPython code is translated to C code that is compiled to build the interpreter – RPython could of course be used to implement other languages 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 JIT (Just-In-Time) compiler built-in – long story short, it allows the code to be run in a faster way 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 3 times the speed, most of the time. Though don't start dreaming too much about it yet – PyPy also has some of the CPython limitations, such as the hated GIL. [2]

While not being a strict optimization technique, targeting PyPy as one of your supported Python implementations is probably a good idea. Achieving this goal requires the same kind of coding policy that is required for support of other Python versions – basically, you need to make sure that you are testing your software under PyPy like you do under CPython. *tox* (see Section 6.7) supports the building of virtual environments using PyPy, just as it does for CPython 2 or CPython 3, so it should be pretty straightforward to put this in place.

Doing so at the beginning of the project will make sure that there's not too much work to do at a later stage if you wish to be able to run your software with PyPy.

[2] Global Interpreter Lock

Note

For the Hy project, we successfully adopted such a strategy from the beginning. Hy always has supported PyPy and all CPython versions without much trouble. On the other hand, we failed to do so in all of our OpenStack projects, and we are now blocked by various code paths and dependencies that don't work on PyPy for various reasons, as they weren't fully tested in the early stages.

PyPy is compatible with Python 2.7, and its JIT compiler works on 32- and 64-bit, x86 and ARM architectures, and under various operating systems (Linux, Windows, Mac OS X…). Support for Python 3 is underway.

10.7 Achieving zero copy with the buffer protocol

Often programs have to deal with a huge amount of data in the form of large arrays of bytes. Handling such a large amount of data in strings can be very ineffective once you start manipulating it by copying, slicing, and modifying them.

Let's consider a small program which reads a large file of binary data, and copies it partially into another file. To examine out our memory usage, we will use memory_profiler, a nice Python package that allows us to see the memory usage of a program line by line.

```
@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()
```

We then run the above program using *memory_profiler*:

```
$ 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)
```

❶ We are reading 10 MB from */dev/urandom* and not doing much with it. Python needs to allocate around 10 MB of memory to store this data as a string.

❷ We copy the entire block of data minus the first KB – because we won't be writing to that first KB to the target file.

What's interesting in this example is that, as you can see, the memory usage of the program is increased by about 10 MB 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.

When dealing with large data, performing this kind of operation on large byte arrays is going to be a disaster. If you happen to have written C code already, you know that using *memcpy()* has a significant cost, both in term of memory usage and in terms of general performance: copying memory is slow.

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 this array without copying it, through the use of basic pointer arithmetic [3].

This is possible in Python using objects which implement the **buffer protocol**. The buffer protocol is defined in PEP 3118, which explains the C API used to provide this protocol to various types, such as strings.

When an object implements this protocol, you can use the **memoryview** class constructor on it to build a new *memoryview* object that will reference the original object memory.

Here's an example:

```
>>> s = b"abcdefgh"
>>> view = memoryview(s)
>>> view[1]
98 ❶
>>> limited = view[1:3]
<memory at 0x7fca18b8d460>
>>> bytes(view[1:3])
b'bc'
```

❶ This is the ASCII code for the letter *b*.

[3] Assuming that the entire string is in a contiguous memory area.

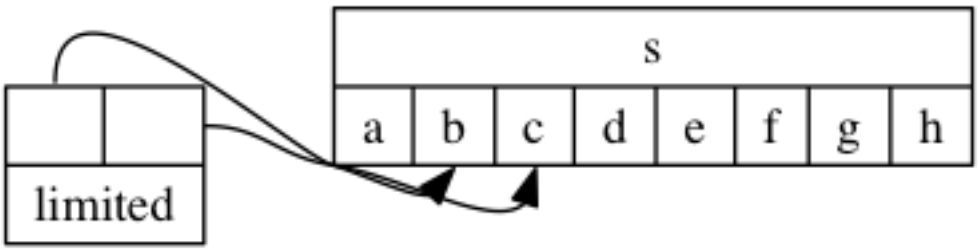


Figure 10.2: Using slice on *memoryview* objects

In this case, we are going to make 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.

With this in mind, we now can rewrite the program, this time referencing the data we want to write using a *memoryview* object.

```
@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()
```

And this program will have half the memory usage of the first version:

```
$ 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       with open("/dev/urandom", "rb") as source:
19.660 MB     9.770 MB           content = source.read(1024 * 10000) ❶
19.660 MB     0.000 MB           content_to_write = memoryview(content) ↩
   [1024:] ❷
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)
```

❶ We are reading 10 MB from */dev/urandom* and not doing much with it. Python needs to allocate around 10 MB 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 to that first KB to the target file. No copying means that no more memory is used!

This kind of trick is especially useful when dealing with sockets. As you may know, when data is sent over a socket, it might not send all the data in a single call. A simple implementation would be to write:

```
import socket
s = socket.socket(…)
s.connect(…)
data = b"a" * (1024 * 100000) ❶
while data:
```

```
    sent = s.send(data)
    data = data[sent:] ❷
```

❶ Build a bytes object with more than 100 millions times the letter *a*.

❷ Remove the first *sent* bytes sent.

Obviously, using such a mechanism, you are going to copy the data over and over until the socket has sent everything. Using *memoryview*, we can achieve the same functionality without copying data – hence, zero copy:

```
import socket
s = socket.socket(…)
s.connect(…)
data = b"a" * (1024 * 100000) ❶
mv = memoryview(data)
while mv:
    sent = s.send(mv)
    mv = mv[sent:] ❷
```

❶ Build a bytes object with more than 100 millions times the letter *a*.

❷ Build a new memoryview object pointing to the data which remains to be sent.

This won't copy anything, and won't use any more memory than the 100 MB initially needed for our *data* variable.

We've now seen memoryview objects used to write data efficiently, but the same method can also 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 it, but also write to it. In this case, we don't need *memoryview* objects – we can just ask an I/O function to write into our pre-allocated object:

```
>>> 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')
```

With such techniques, it's easy to pre-allocate 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 even place data at any point in the memory area:

```
>>> ba = bytearray(8)
>>> ba_at_4 = memoryview(ba)[4:] ❶
>>> with open("/dev/urandom", "rb") as source:
...     source.readinto(ba_at_4) ❷
...
4
>>> ba
bytearray(b'\x00\x00\x00\x00\x0b\x19\xae\xb2')
```

❶ We reference the *bytearray* from offset 4 to its end.

❷ We write the content of /dev/urandom from offset 4 to the end of the *bytearray*, effectively reading 4 bytes only.

Tip

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 copy.

10.8 Interview with Victor Stinner

Victor is a long time Python hacker, a core contributor and the author of many Python modules. He recently authored PEP 454, which proposes a new `tracemalloc` module to trace memory block allocation inside Python, and also wrote a simple AST optimizer.

What's a good starting strategy to optimize Python code?

Well, 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 a wasting time and premature optimizations. Useless optimizations may make the code worse, less readable, or even slower. A useful optimization must speed the program up by at least 5%.

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%, maybe 25%, to justify an optimization on a micro-benchmark.

It may be interesting to run a benchmark on different computers, different operating systems, different compilers. For example, performances of *realloc()* may vary between Linux and Windows. Even if it should be avoided, sometimes, the implementation may depend on the platform.

There's a lot of different tools around for profiling or optimizing Python code; what are your weapons of choice?

Python 3.3 has a new `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; 3 times is a minimum, 5 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 which use the most CPU power. To find these functions, Python has `cProfile` and profile modules which record the amount of time spent in each function.

What are the interesting Python tricks to know that could improve performance?

The standard library should be reused as much as possible – it's well tested, and also usually efficient. Python built-in 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, etc.

There are some hacks to optimize Python, but they should be avoided because they make the code less readable in exchange for only a minor speed-up.

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.

In which areas does Python have poor performance? Which areas should

be used with care?

In general, I prefer not to worry about performance while developing a new application. Premature optimization is the root of all evil. When slow functions are identified, the algorithm should be changed. If the algorithm and the container types are well chosen, it's possible to rewrite short functions in C to get best performances.

A 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 workaround 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 "mistakes" that contribute to poor performance do you see most often?

When Python is not well understood, inefficient code can be written. For example, I have seen `copy.deepcopy()` misused, when no copy was required.

Another performance-killer is an inefficient data structure. With less than one hundred items, the container type has no impact on performance. With more items, the complexity of each operation (add, get, delete) and it's effects must be known.

11 Scaling and architecture

Nowadays all the hype is about resiliency and scalability, so I assume this is something that your development process is going to have to take into account sooner or later. Many sides of the issue are not particularly tied to Python itself, while some are only relevant to its main implementation, CPython.

The scalability, concurrency and parallelism of an application largely depend on the choices made about its initial architecture and design. As you'll see, there are some paradigms – like multi-threading – that don't apply correctly to Python, whereas other techniques, such as service oriented architecture, work better.

11.1 A note on multi-threading

What is multi-threading? It's the ability to run code on separate processors [1] inside a single Python process. This means that different parts of your code will be run in parallel.

Why is this needed? The most common cases are:

1. You need to run background tasks without stopping your main thread's execution, e.g. in the case of a graphical user interface where the main loop is waiting for events.

[1] Or sequentially on one, if multiple CPUs aren't present

2. You need to spread your work-load across several CPUs.

So at first, it may seem that multi-threading looks like a good way to scale and parallelize your application, solving these problems. When you want to spread a work-load, you start a new thread for each new request instead of handling them one at a time.

Wonderful. Job done. We can move on.

No – sorry! First, if you've been in the Python world for a long time, you've probably encountered the word *GIL*, and know how hated it is. The GIL is the Python Global Interpreter Lock, a lock that must be acquired each time *CPython* [2] needs to execute byte-code. Unfortunately, this 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, in fact most applications I've seen running requests in multiple threads struggle to attain 150% CPU usage – i.e. 1.5 cores used. With computing nodes nowadays not usually having less than 2 or 4 cores, it's a shame. Blame the GIL.

There isn't currently any work being done to remove the GIL in *CPython*, because nobody thinks the solution is worth the difficulty of implementing and maintaining it.

However, *CPython* is just one [3] of the available Python implementations. Jython, for example, doesn't have a global interpreter lock, which means that it can run multiple threads in parallel efficiently. Unfortunately, these projects by their very nature lag behind *CPython*, and so are not really useful targets.

[2] The reference implementation of Python written in C that you run by typing *python* in your shell.
[3] although the most commonly used.

Note

PyPy is another Python implementation, but is written in Python (see Section 10.6). *PyPy* has a GIL too, but very interesting work is happening right now to replace it with a STM (Software Transactional Memory)-based implementation. This is something very exciting that's going to change how we build and run multi-threading software in the future. Hardware support is starting to appear in some processors, and Linux kernel developers are looking at ways to suppress kernel locks too. These are good signs.

So are we back to our initial use cases, with no good solutions on offer? Not true – there's (at least) two solutions you can use:

1. If you need to run background tasks, the easiest way to do that is to build your application around an event loop. There's a lot of different Python modules which provide for this, even a standard one called `asyncore`, which is an effort to standardize this functionality as part of PEP 3156. Some frameworks such as Twisted are built around this concept. The most advanced ones should give you access to events based on signals, timers and file descriptors activity – we'll talk about this in Section 11.3.
2. If you need to spread the work-load, using multiple processes is going to be more efficient and easier. See Section 11.2.

For us developers, mere mortals, it all means that we should think twice before using multi-threading. I've used multi-threading to dispatch jobs in rebuildd, a Debian build daemon I wrote a few years ago. While it seemed handy to have a thread to control each running build job, I very quickly fell into the concurrency trap. If I had the chance to begin again, I'd build something based on asynchronous events handling or multi-processing, and not have to worry about this problem.

Getting multi-threaded applications right is hard. The level of complexity means that it is a larger source of bugs than most others – and considering the little to be

gained generally, it's better not to waste too much effort on it.

11.2 Multiprocessing vs multithreading

As explained earlier, multi-threading is not a good scalability solution because of the *GIL*. A better solution is the **multiprocessing** package that is provided with Python. It provides the same kind of interface that you would have using the **multithreading** module, except that it starts new processes (via *fork(2)*) rather than new system threads.

The below program is a simple example, which sums one million random integers 8 times, spread across 8 threads at the same time.

Worker using multithreading

```
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)
```

Running this program returns the following:

Example 11.1 Result of *time python worker.py*

```
$ 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 4 cores CPU, which means that Python could have used up to 400% CPU power. But it was clearly unable to do that, even with 8 threads running in parallel – it stuck at 129%, which is just 32% of the hardware's capabilities.

Now, let's rewrite this implementation using **multiprocessing**. For a simple case like this, it's pretty straightforward:

Example 11.2 Worker using multiprocessing

```
import multiprocessing
import random

def compute(n):
    return sum(
        [random.randint(1, 100) for i in range(1000000)])

# Start 8 workers
pool = multiprocessing.Pool(8)
print("Results: %s" % pool.map(compute, range(8)))
```

Running this program under the exact same conditions gives the following result:

Example 11.3 Result of *time python worker.py*

```
$ time python workermp.py
Results: [50495989, 50566997, 50474532, 50531418, 50522470,  ↩
   50488087, 50498016, 50537899]
```

```
python workermp.py  16.53s user 0.12s system 363% cpu 4.581 total
```

The execution time has been reduced by 60%; this time, we have been able to consume up to 363% of CPU power, which is more than 90% of the computer's CPU capacity.

A further note – the **multithreading** module is not only able to efficiently spread a work-loads over several local processors, but can also do so over a network, through its **multithreading.managers** objects. It also provides bi-directional communication transports so your processes can exchange information with each other.

Each time you think that you can **parallelize** some work for a certain amount of time, it's much better to rely on multi-processing and to fork your jobs, in order to spread the workload among several CPU cores.

11.3 Asynchronous and event-driven architecture

Event-driven programming is a good solution to organize program flow in a way which listens for various events at once, without using a multi-threaded approach.

Consider an application that wants to listen for connection on a socket and then process the connection it receives. There are basically three ways to approach the problem:

1. Fork a new process each time a new connection is established, relying on something like the *multiprocessing* module.
2. Start a new thread each time a new connection is established, relying on something like the *threading* module.
3. Add this new connection to your event loop, and react to the event it will generate when it occurs.

It is (now) well known that listening to hundreds of event sources is going to scale much better when using an event-driven approach than, say, a thread-per-event approach [4]. This doesn't mean that the two techniques are not compatible, but it does mean that you can usually get rid of multiple threads by using an event-driven mechanism.

We've already covered the pros and cons of the first options; in this section, only the event-driven mechanism will be discussed.

The technique behind event-driven architecture is the building of an event loop. Your program calls a function that blocks until an event is received. The idea behind this is that your program can be kept busy while waiting for inputs and outputs to complete; the most basic events are "I have data ready to be read" or "I can now write data without blocking".

In Unix, the standard functions used to build such an event loop are the system calls `select(2)` or `poll(2)`. They expect a few file descriptors to listen for, and will react when one of them is ready to be read from or written to.

In Python, these system calls are exposed through the `select` module. It's easy enough to build an event-driven system with them, though it can be tedious.

Example 11.4 Basic example of using `select`

```
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
```

[4] For further reading on this, take a look at the C10K problem.

```
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")
```

A wrapper around these low-level interfaces was added to Python in the early days, called `asyncore`. It is not widely used, and hasn't evolved much.

Alternatively, there are many frameworks which provide this kind of functionality in a more integrated manner, such as Twisted or Tornado. Twisted has been almost a de-facto standard for years in this regard. C libraries that export Python interfaces, such as libevent, libev or libuv, also provides very efficient event loops.

While they all solve the same problem, the downside is that nowadays there are too many choices, and most of them are not interoperable. Also, most of them are callback based – which means that the program flow is not really clear when reading the code.

What about gevent or Greenlet? They avoid the use of callback, but the implementation details are scary, and include CPython x86 specific code and monkey-patching of standard functions. Not something you want to use and maintain on the long term, really.

Recently, Guido Van Rossum started to work on a solution code-named *tulip*, which

is documented under PEP 3156.[5] The goal of this package is to provide a standard event loop interface. In the future, all frameworks and libraries would be compatible with it and would be able to interoperate.

tulip has been renamed and merged into Python 3.4 as the *asyncio* package. If you don't plan to depend on Python 3.4, it's also possible to install it for Python 3.3 using the version provided on PyPI – simply running `pip install asyncio` will do the job. Victor Stinner started a backport of *tulip* named *trollius*, which aims to be compatible with Python 2.6 and superior versions.

Now that you've got all the cards in your hand, no doubt you're wondering: but **what should I use to build an event loop in my event-driven application?**

At this point in Python's development, it's a really tough question. The language is still in a transition phase. As of the time of this writing, nothing yet uses the `asyncio` module. That means that using is going to be a real challenge.

Here are my recommendations at this point:

- If you target Python 2 only, `asyncio` is out of reach for you. For me, the next best choice would be something based on `libev`, like pyev.
- If you target both major Python versions – 2 and 3 – you'd better use something that is compatible with both, such as pyev. However, I would strongly advise you to keep in mind that you might have to transition later to `asyncio`. It may be useful to have a minimal abstraction layer, and not to spread the internal guts of your eventing-dependency over the entire program. If you're adventurous, trying to mix `asyncio`/`trollius` can be a nice solution too.
- If you only target version 3, go ahead with `asyncio`. It'll be a pain to start with, as there are still not a lot of examples or documentation, but it's a safe bet. You'll be a pioneer.

[5] *Asynchronous IO Support Rebooted: the "asyncio" Module*, Guido van Rossum, 2012

Example 11.5 Example with pyev

```
import pyev
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)

def server_activity(watcher, revents):
    connection, client_address = server.accept()
    connection.send("hello!\n")
    connection.close()

loop = pyev.default_loop()
watcher = pyev.Io(server, pyev.EV_READ, loop, server_activity)
watcher.start()
loop.start()
```

As you can see here, the `pyev` interface is pretty easy to grasp. Via its `libev` usage, it supports an `Io` object for input/output, but also the tracking of child processes, timers, signals and even callbacks to call when idle. `libev` also automatically relies on the best interface for polling – `epoll(2)` on Linux or `kqueue(2)` on BSD.

11.4 Service-oriented architecture

Considering the previously stated problems and solutions, the shortcomings of Python in terms of scalability and usage in large, complex applications can seem tricky to circumvent. However it appears that Python is really good at implementing Service-Oriented Architecture (SOA) – if you're not yet familiar with this, there's plenty of documentation and opinions that you can read online.

SOA is the architecture type used by OpenStack in all its components. Components use HTTP REST to communicate with external clients (end-users) and an abstracted RPC mechanism that can support several wire protocols, the most commonly used one being AMQP.

In your own case, the choice of which communication channels to use between those blocks is mainly a matter of knowing with whom you will be communicating.

When exposing an API to the outside world, the preferred channel nowadays is HTTP, and especially stateless designs such as REST [6] style architectures. These kinds of architectures are easy to implement, scale, deploy and comprehend.

However, when exposing and using your API internally, using HTTP may be not the best protocol. A large panel of communication protocols for applications exist, and a full description of any of them would likely fill an entire book.

In Python, there's plenty of libraries to build RPC [7] systems. Kombu – among others – is interesting because it provides an RPC mechanism on top of a lot of back-ends; AMQ protocol being the main one. But support for Redis, MongoDB, BeanStalk, Amazon SQS, CouchDB, or ZooKeeper are also provided.

In the end, there's a huge amount to be gained indirectly from using such loosely coupled architecture. If we consider that each module provides and exposes an API,

[6] Representational state transfer
[7] Remote Procedure Call

we can run multiple daemons exposing this API. For example, *Apache httpd* would create a new worker using a new system process that handles new connections; we can then dispatch our connection to a different worker running on the same compute node. All we need to have is a system of dispatching the work between our workers, which provides this API. Each block will be a different Python process, and as we've seen above, this is better than multi-threading to spread your work-load. You'll be able to start multiple workers on each computing node you have. Even if not strictly necessary, using stateless blocks should be favored any time you have the choice.

ZeroMQ is a socket library that can act as a concurrency framework. The following example implements the same worker seen in the previous examples, but uses ZeroMQ as a way to dispatch and communicate.

Workers using ZeroMQ

```
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
work_sender = context.socket(zmq.PUSH)
work_sender.bind("tcp://0.0.0.0:5555")
# Build a channel to receive computed results
result_receiver = context.socket(zmq.PULL)
result_receiver.bind("tcp://0.0.0.0:5556")
# Start 8 workers
processes = []
for x in range(8):
    p = multiprocessing.Process(target=worker)
    p.start()
    processes.append(p)
# Start 8 jobs
for x in range(8):
    work_sender.send_pyobj(compute)
# Read 8 results
results = []
for x in range(8):
    results.append(result_receiver.recv_pyobj())
# Terminate all processes
for p in processes:
    p.terminate()
```

```
print("Results: %s" % results)
```

As you can see, ZeroMQ provides an easy way to build communication channels. I've chosen 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 a *inproc* communication channel that works by using Unix sockets. Obviously the communication protocol built upon ZeroMQ in this example is very simplistic – in order to keep this book's examples clear and concise; but it shouldn't be hard to imagine building a more sophisticated communication layer on top of it.

With such a protocol, it's easy to imagine building a entirely distributed application communication with a network message bus – ZeroMQ, AMQP, or something else.

Note also that protocols like HTTP, ZeroMQ or 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 is a good option. It allows you to build both synchronous and asynchronous APIs that can be spread from one computer to several thousand. It doesn't tie you to a particular technology or language – and these days, there's no longer a reason not to be ready to distribute your software, or to be constrained by any particular language.

12 RDBMS and ORM

RDBMSs [1] and ORM [2] are touchy subjects, but there's no way to avoid having to deal with them sooner or later. Many applications have to store data of some kind, and developers often choose to do so using relational databases. And when a developer chooses to use a relational database, the tool they almost always choose to use for it is an ORM library of some kind.

Note

This chapter will be a little less Python-centric than others; bear with me. I'll only be talking about relational databases here, but many of the things we'll cover here can also apply to other kinds of databases.

RDBMSs are about storing relational data using normal form, while SQL is about dealing with relational algebra. Together, they allow you to store data and answer questions about that data. However, there are a number of common difficulties with using ORM in object-oriented programs, known collectively as the object-relational impedance mismatch. The bottom line is, relational databases and object-oriented programs have different representations of data which don't map properly to one another: mapping SQL tables to Python classes won't give you optimal results, no matter what you do.

[1] Relational database management systems
[2] Object-relational mapping

ORM is supposed to make database systems easier to access: these tools abstract the process of creating queries, generating SQL so you don't have to. Unfortunately, more likely sooner than later, you'll want to do something with your database only to discover that the abstraction layer simply won't allow it. To make the most efficient use of your database, you absolutely have to have an understanding of SQL and RDBMSs so that you can write your own queries directly without having to rely on the abstraction layer for everything.

But that's not to say you should avoid ORM entirely. ORM libraries can help with rapid prototyping of your application model, and some even provide useful tools such as schema upgrades/downgrades. The important thing is that you understand that it's not a substitute for a proper grasp 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.

Imagine a SQL table for keeping track of messages. It has a single column named "id," which is the primary key, and a string containing the message:

```
CREATE TABLE message (
  id serial PRIMARY KEY,
  content text
);
```

We want to avoid duplicates when receiving a message, so a typical developer would write something like this:

```
if message_table.select_by_id(message.id):
    # We already have the message, it's a duplicate, ignore and raise
    raise DuplicateMessage(message)
else:
    # Insert the message
    message_table.insert(message)
```

This would definitely work in most cases, but it has some major drawbacks:

- It implements a constraint already expressed in the SQL schema, so it is a sort of code duplication.
- It execute 2 SQL queries; executing SQL query might be long and requires round-trip to the SQL server, introducing extraneous delay.
- It doesn't take into account the possibility of someone else inserting a duplicate message after we call *select_by_id* but before we call *insert*, which would cause the program to raise an exception.

There's a much better way to write this code, but it requires cooperation with the RDBMS server rather than treating it like dumb storage:

```
try:
    # Insert the message
    message_table.insert(message)
except UniqueViolationError:
    # Duplicate
    raise DuplicateMessage(message)
```

This achieves the exact same effect in a more efficient fashion and without any race condition. This is a very simple pattern, and it doesn't conflict with ORM in any way. The problem is that developers tend to treat SQL databases as dumb storage and duplicate the constraints they wrote (or could write) 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.

Another point that needs to be raised about ORM is support for multiple database backends. Many ORM libraries tout it as a feature, but it's really a trap waiting to

ensnare unsuspecting developers. No ORM library provides a complete abstraction of all RDBMS features, so you'll have to dumb down your code to the most basic RDBMS available (or that you want to put up with), and you'll be unable to use any advanced RDBMS functions without breaking the abstraction layer.

Simple things that aren't standardized in SQL, such as handling timestamp operations, are a pain to deal with when using an ORM; even more so if your code is written to be RDBMS-agnostic. With this in mind, be sure to choose an RDBMS that suits your application well [3].

A good way to mitigate the problems with ORM libraries is to isolate them as prescribed in Section 2.3. This approach not only allows you to easily swap your ORM library for a different one should the need arise, but it also allows you to optimize your SQL usage by identifying places with inefficient usage of queries, bypassing most of the ORM boilerplate.

An easy way to build such isolation is to for example only use your ORM in a module of your application, for example `myapp.storage`. This module should only exports functions and methods that allow you to manipulate the data at a high level of abstraction. The ORM should be only used from that module. At any point later, you will be able to drop in any module providing the same API to replace `myapp.storage`.

In the end, this section's goal isn't to take a side in the debate over whether to use ORM; there's already plenty of discussion on the Internet arguing over the pros and cons. The point of this section is to help you understand how important it is to know enough about SQL and RDBMS to make use of their full potential in your application.

The most commonly used ORM library in Python (and arguably the *de facto* standard) is SQLAlchemy. It supports a huge number of different backends and provides abstraction for most common operations. Schema upgrades can be handled by third-party packages such as alembic.

[3] When in doubt, pick PostgreSQL.

Some frameworks, such as Django, provide their own ORM libraries. If you choose to use a framework, it's a smart idea to use the built-in library, which will (obviously) have better integration with the framework than an external one.

Warning

The MVC [a] architecture that most frameworks rely on can be easily misused. They implement (or make it easy to implement) ORM in their model directly, but without abstracting enough of it: any code you have in your view and controllers that uses the model will *also* be using ORM directly. This is something that you need to avoid. You should write a data model that *includes* the ORM library rather than *consists* of it: this will provide better testability and better isolation, as well as make it easier to swap out with another storage technology should the need arise.

[a] Model View Controller

12.1 Streaming data with Flask and PostgreSQL

In the previous section, we talked about how important it can be to masterize your data storage system. Here, I'll show you how you can use one of *PostgreSQL*'s advanced features to build an HTTP event streaming system.

The purpose of this micro-application 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. The code that creates this table is quite simple:

Example 12.1 Creating the message table

```
CREATE TABLE message (
  id SERIAL PRIMARY KEY,
  channel INTEGER NOT NULL,
  source TEXT NOT NULL,
```

```
  content TEXT NOT NULL
);
```

What we also want to do is 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:

Example 12.2 The *notify_on_insert* function

```
CREATE OR REPLACE FUNCTION notify_on_insert() RETURNS trigger AS $$
BEGIN
  PERFORM pg_notify('channel_' || NEW.channel,
                    CAST(row_to_json(NEW) AS TEXT));
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

This 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* provides a *pl/python* language by embedding the Python interpreter.

This function performs a call to *pg_notify*. This 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!

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:

Example 12.3 The trigger for *notify_on_insert*

```
CREATE TRIGGER notify_on_message_insert AFTER INSERT ON message
FOR EACH ROW EXECUTE PROCEDURE notify_on_insert();
```

And we're done: 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:

Example 12.4 Receiving notifications in Python

```
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)
```

The above code connects to PostgreSQL using the *psycopg2* library. We could have used a library that provides an abstraction layer, such as *SQLAlchemy*, but none of them provide access to the LISTEN/NOTIFY functionality of PostgreSQL. It's still possible to access the underlying database connection to execute the code, 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 would provide.

The program listens on *channel_1*. As soon as it receives a notification, it prints it to the screen. If we run the program and insert a row in the *message* table, we get this output:

```
$ python3 listen.py
Got NOTIFY: 28797 channel_1
{"id":10,"channel":1,"source":"jd","content":"hello world"}
```

Now, we'll use Flask, a simple HTTP micro-framework, to build our application.

We're going to send the data using the **Server-Sent Events** message protocol defined by HTML5 [4].

Example 12.5 Flask streamer application

```
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),
```

[4] An alternative would be to use *Transfer-Encoding: chunked* defined by HTTP/1.1.

```
                            mimetype='text/event-stream')


if __name__ == "__main__":
    app.run()
```

This application is quite simple and only supports streaming for the sake of the example. We use Flask to route a request to `GET /message/<channel>`; as soon as the code is called, it returns a response with the mimetype *text/event-stream*, sending back a generator function instead of a string. Flask will then 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 it, since we're fine with sending JSON data to the HTTP client.

Note

For the sake of simplicity, this example application has been written in a single file. It isn't easy to depict examples spanning multiple modules in a book. If this were a real application, it would be a good idea to 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:

```
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
```

Data starts coming in through the terminal where *curl* is running:

```
data: {"id":71,"channel":1,"source":"jd","content":"hello world"}


data: {"id":72,"channel":1,"source":"jd","content":"it works"}
```

A naive and arguably more portable implementation of this application [5] would in-

[5] It would be compatible with other RDBMS servers, such as MySQL

stead loop over a *SELECT* statement over and over to poll for new data inserted in the table. However, there's no need to demonstrate that a push system like this one is much more efficient than constantly polling the database.

12.2 Interview with Dimitri Fontaine

I first met Dimitri a decade ago. He is a skilled PostgreSQL Major Contributor who works at 2ndQuadrant 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 RDBMS as their storage backends? What should they know about?*

That's a very good question, mainly because it offers more than one opportunity to clarify assumptions that I want to highlight as very wrong here. If you think the question as asked makes sense, you really need to be reading my answer now!

Let's start with something really boring: RDBMS stands for Relational DataBase Management System. Those beasts have been invented in the 70s to answer some common needs that every application developer needed to solve themselves at that time, and the main services RDBMS have been implementing are not data storage, as everyone knew how to implement that already.

The main services offered by a RDBMS are the following:

- Concurrency: access your data for read or write with as many concurrent **threads of execution** as you want to, the RDBMS is there to handle that correctly for you. That's the main feature you want out of a RDBMS.
- Concurrency semantics: the details about the concurrency behaviour when using a RDBMS are proposed with a high-level specification in terms of **Atomicity** and **Isolation**, that are maybe the most crucial parts of **ACID**. **Atomicity** is the property that in 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 that includes **Data Definition Language** (or `DDL`, e.g. `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 4 level of isolation, as described in transaction isolation documentation

The RDBMS takes full responsibility for your data. So it allows the developer to **describe** its own rules for consistency and then it will check that those rules are valid at crucial times such as transaction **commit** or statements boundaries, depending on the **deferability** of your constraints declarations.

The first constraint you can place on your data is about its expected input and output formatting, using the proper **data type**. A proper 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 (**Julian** is not huge nowadays, you probably want **Gregorian** unless doing history).

Data Types are not just about input and output formats, though. They also allow to implement behaviours 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, points boxes and lines, booleans and circles, UUIDs and XML, Arrays and Ranges in the same way, to name but a few.

Protecting your data also means that the only choice for a proper RDBMS is to actively refuse data that won't match with your consistency rules, the first of which is the data type you've chosen. If you think it's OK 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 though as a user level extension of the data type definition and behavior, the main difference being that you can choose to `DEFER` checking those constraints from being enforced at the end of each statement to being enforced at the end of the current transaction.

The **relational** bits of an RDBMS is 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 a process 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 given by the result of your **Normalization** process. Common wisdom is to first normalize your data schema and only then see about how to **denormalize** it in order to get back the flexibility you think you need. Chances are that you realize you actually don't need any.

When you do need more flexibility, using PostgreSQL you can pick from

a number of **denormalisation** options: composite types, records, arrays, hstore, json or XML, for starters.

There's a very important drawback to **denormalisation** 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 **denormalisation** as possible when using composite types, arrays or hstore, and even json in recent releases.

The RDBMS knows very much about your data and can help you implement a very fined 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 procedure, allowing you to offer access to sensible data in a very controlled way, much the same as with using `suid` programs.

The RDBMS offers you to access your data using a **Structured Query Language** which became a **de-facto** standard in the 80s and is now driven by a commitee. In the case of PostgreSQL, lots of extensions are being added with each and every major release each year allowing you to have access to a very rich **DSL** language. All the work of query planning and optimisation is done for you by the RDBMS so that you can focus on a **declarative** query where you only describe 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 just removing the **Structured Query Language** out of the offering, but a whole lot of other foundations that you've been trained to expect.

My advice to developers is to remember the differences between a **storage backend** and a RDBMS. Those are very different services, and if all you need actually is a storage backend, maybe consider not using a 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, see the list of data types, operators, functions, features and extensions it provides. Read some usage examples on blog posts.

Then consider PostgreSQL as 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 ORM?

SQL stands for **Structured Query Language** and in the case of PostgreSQL has been proven to be **Turing Complete**. Its implementation and optimizer are far from trivial.

As ORM stands for **Object Relational Mapper**, the idea is that you can deal with a one-to-one mapping of database relations with classes and database tuples with objects, or class instances.

Even when a RDBMS, like PostgreSQL, implements strong static typing, relation definitions are built on the fly: each query result is a new relation. Each subquery result is a new relation that might exists only for the duration of the subquery. Each `JOIN`, either **INNER** or **OUTER**, will result in a new relation dynamically built for solving just that `JOIN`.

As a direct consequence of that, it's easy to understand that where the **ORM** will be able to best work for you is for what's called **CRUD** applications: **Create**, **Read**, **Update** and **Delete**. The **Read** part should then only be limited to a very simple `SELECT` statement targeting a single table. If you compare non-trivial **output lists** you can measure the impact of retrieving more columns than necessary on query performances. Now,

if your **ORM** is including all the known fields in its **projections** (or output list), then it will force your RDBMS to fetch external data (and decompress) it before sending it, maybe only to compress it again if you're using **SSL** in between the RDBMS and your application. Also, just think about network bandwidth usage and remember than we're measuring simple **primary key** based lookup queries in fractions of a **millisecond**.

So 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** of choice is well able to only fetch the data you're asking for, then you have to somehow manage the exact list of columns you want in each situation, and avoid using a simple abstract magic method that will automatically compute the fields list for you.

The next part of the **CRUD** queries are simple `INSERT`, `UPDATE` and `DELETE` statements. First, all those commands accept joins and sub-select when you're using an advanced RDBMS, such as PostgreSQL. Then again, for example PostgreSQL implements the `RETURNING` clause, allowing you to return to the client any data that's just been edited, such as **default** (typically sequence numbers for surrogate keys) and other values **computed** automatically on the RDBMS (typically with `BEFORE <action>` triggers).

Is your **ORM** aware of that? What's the syntax there to benefit from that?

In the general case, a relation is either a table, the result of calling a **Set REturning Function**, or the result of any query. It's common practice when using an **ORM** to build a **relational mapping** in 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 legend of the Sufficiently Smart Compiler applies to ORMs too. For more details about what that legend is, read On Being Sufficiently Smart by James Hague.

The idea when applied to our very 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 even work out the exact set of data you are interested into.

It's true that at times, SQL can get quite complex. You're not going to get anywhere near simpler by using an API to SQL generator that you can't control, though.

After having said all that against the typical **ORM**, something needs to be said against the alternatives.

Building **SQL** queries as a string is not scalable. You want to be able to compose several **restrictions** (the `WHERE` clauses) and dynamically add some joins right into a subquery just so that you can optionally fetch some more detailed data, etc.

My current thinking is that the tool you really want to have is not an **ORM**, it's a nice way to compose a **SQL** query from a programmatic interface.

There's a PostgreSQL driver proposing exactly the right abstraction to that problem, it's the **Common Lisp** library Postmodern with the S-SQL solution. Of course, **Lisp** lends itself really well to allow for easy to program **composable** components.

Actually in two cases you can relax and use your ORM, provided that you're willing to accept the following compromise: as soon as possible you will need to edit your ORM usage out of your code base.

- **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 when compared to hand crafting 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, and your usage of the ORM having painted you into a corner and bad code design decisions. But if you're there, you're successful enough to spend some refactoring money and remove any dependency toward the ORM, right?

- **CRUD Application**; the real thing, where you are only editing a single tuple at a time, and you don't really care about performances. Like for the basic admin application interface.

Are there any pros or cons to choosing PostgreSQL over other databases when working with Python?

Here are my top reasons for choosing PostgreSQL as a developer:

- Community support: the PostgreSQL community really is welcoming to new users, and will typically spend the time it takes to fully understand your question before to answer the best possible answer. The mailing lists are still the best way to communicate with the community. See PostgreSQL Mailing Lists for details.
- 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, function, operators, arrays and ranges: PostgreSQL has a very rich set of data types that are really useful and come with a host of operators and functions to process them. It's even possible to denormalize using **arrays** or **JSON** data types, and still be able to write

advanced queries including joins against those. For example, did you know about the ~ regular expression operator? and the `regexp_split_to_array` and `regexp_split_to_table` functions?

- The planner and optimizer: you have to try to push the limits you know about those to really understand how complex and powerful they are. I've repeatedly seen 2 to 3 pages long queries run to complement in a small number of milliseconds.
- 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?
- `INSERT INTO ...RETURNING`: you can return anything from the `INSERT` statement directly, like for example the `id` value that got derived from a sequence. You win a network round-trip and get the result with the same protocol and tools as when issuing a `SELECT` statement.
- `WITH (DELETE FROM ...RETURNING *) INSERT INTO ...SELECT`: PostgreSQL support **Common Table Expression** in queries, which are known as **WITH queries**, and thanks to its support for the `RETURNING` clause, it also supports **DML** commands there. That's just awesome, rith?
- Window Functions, `CREATE AGGREGATE`: if you don't know what a window function is, go read about it in the PostgreSQL Manual or in my blog at Understanding Window Functions. Then you have to realise that PostgreSQL allows you to use any existing **aggregate** as a window function, and allows you to dynamically define new aggregates online in SQL.
- PL/Python (and others such as C, SQL, Javascript or Lua): you can run your own code on the server, right where the data is, so that you don't have to fetch it over the network just to process it then send it back in a query to do the next level of JOIN. Whatever it is, you can do it all on the

server.

- Specific Indexing (GiST, GIN, SP-GiST, partial & functional): did you know that you can create Python functions to process your data from within PostgreSQL, then index the result of calling that function? So that 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? PostgreSQL implements index frameworks for non sortable data types, like 2 dimensional types (ranges, geometry, etc); and for container data types. Lots of cases are already supported out of the box, and a host more thanks to the **Extension** system. Have a look at the Additional Supplied Modules and the PostgreSQL Extension Network.
- Extensions: such extensions include **hstore**, a full blown key value store with flexible indexing, **ltree** for indexing nested tags, **pg_trgm** as a poor man's full text search solution, that supports indexing regular expression searches and unanchored **LIKE** queries, **ip4r** for quick searches of an IP address in a range, and a lot more.
- Foreign Data Wrappers: the **foreign data wrappers** are a whole class of extensions, implementing the SQL/MED standard (Management of External Data). The idea is to embed a connection driver right into the PostgreSQL server then expose it through the `CREATE SERVER` command. PostgreSQL provides an API to **foreign data wrapper** authors that allows them to implement read and write access to the remote data, and also `where` clauses push-down for efficient joining capabilities. You can even use the advanced SQL capabilities of PostgreSQL against data that you maintain with another piece of technology!
- `LISTEN/NOTIFY`: PostgreSQL implements an asynchronous server-to-client protocol called `LISTEN/NOTIFY`. The application may receive unsolicited

messages from the server when something interesting happens, for example an UPDATE of some data. The `NOTIFY` command accepts a data payload so that you can e.g. notify your cache application the object id's to purge when the object just has been removed or updated. Of course, the notification only happens if the transaction actually did a successful `COMMIT`.

- `COPY` Streaming protocol: PostgreSQL implements a **streaming** protocol and uses it to implement its fully integrated replication solution. Now, that protocol is quite easy to use from an application and allows impressive performance boosts. As soon as you're working on more than a dozen row at a time, sometimes before, thing about using `COPY` against a **temporary table** then issuing a single statement joining to that temporary table: PostgreSQL knows how to join against other tables in all data modifying statements (**insert**, **update**, **delete**), and batch operation usually are way faster.

13 Python 3 support strategies

As far as I'm aware, Python 3 is still not the default Python interpreter in any system that I'm aware of at the moment, despite having been released in December 2008 – five years ago!

The problem, as you know, is that Python 3 broke compatibility with Python 2. At the time that Python 3.0 arrived, the gap between it and Python 2.6 was so huge that people weren't even beginning to think about bridging it. Scared. Shrugging.

But then things changed: Python 2.7 back-ported a lot of features from Python 3.1, narrowing the gap. Much sanity returned through subsequent versions of Python, and I am happy to state that it is now possible to support both Python 2.7 and Python 3.3… almost without difficulty!

There is official documentation on porting applications, but I wouldn't recommend following it to the letter. It talks a lot about the *2to3* tool – which converts Python 2 code to Python 3 – and contains proposals like starting a special Python 3 branch for your project.

In my opinion, this is terrible advice nowadays. It may have been the most appropriate advice a few years ago, but considering the current state of "compatibility" between Python 2.7 and Python 3.3, it's better to forget about this approach.

Note

Note that a *3to2* tool also exists – but for the same reason given above, I wouldn't encourage its use.

Firstly, *2to3* doesn't do always the right thing – it's not magic. It only deals with syntax changes, which covers a lot; but it doesn't maintain backward compatibility with Python 2 – and in any case, you'll have to handle semantic changes manually. Secondly, running *2to3* is damn slow; and for this reason it's unlikely to be a good long-term solution. Some guides even suggest running it at *setup.py* time, which is somewhat hazardous.

Some documentation recommends using different project branches to support Python 2 and Python 3. Experience shows that this can be terrible to manage, and that users will get confused about which version they should use. Even worse, you will get confused when they start submitting bug reports without explicitly stating which branch they are using.

A better method is to use a single code base that is both Python 2 and Python 3 compatible. This is on what we put our effort on with OpenStack.

In the end, the only way to be sure that your code works under both Python versions is to have unit testing. Without unit testing, it is **impossible** to know if your code will work in both contexts and across versions. If you do not have any test in your application [1] the first thing to do is to increase your code coverage dramatically; you may want to jump to Chapter 6 right ahead.

Tox is a great tool for automating tests run against multiple Python versions, and we'll talk about it in Section 6.7.

Once you have unit tests and tox set up, it's easy enough to run your tests against both Python versions using:

[1] I have heard that such projects exist.

```
tox -e py27,py33
```

See what's broken, fix it, and launch *tox* again. Repeat until all tests pass. If you're doing it correctly, the number of errors will decrease slowly but steadily, to the point where all of your code base will be fully Python 2 and 3 compatible.

If you have a C module written for Python that you would like to port, I'm sorry to inform you that there's not much to say – other than to tell you to read the documentation and port your code. It may be a useful option to rewrite using cffi if possible.

In the following sections I will discuss some points you will encounter while porting between Python versions. I will assume that you already have a Python 2 code base. While most of what follows could in theory also be applied to the porting of a Python 3 project to Python 2, I have never personally encountered such a case.

13.1 Language and standard library

The language hasn't changed radically; I'm sure you've already taken a look. This book won't cover the entire list of changes – it would be much too boring, and in any case can be found online. The book Porting to Python 3 gives a pretty good overview of what you may need to change in order to support Python 3.

If you haven't yet taken a look at the language changes made in Python 3, I invite you to do so. It's a great language, with a lot less corner cases, and much cleaner interfaces on various object bases. You'll love Python 3.

But it raises strong compatibility problems. The syntax changes to some statements (e.g. exception catching) have removed old Python version compatibilities, and they can be a pain to tackle if you used them. The hacking tool that we'll discuss in section Section 1.4 can help you to fix these incompatible usages, and stop

you from adding more.

When supporting multiple versions of Python, you shouldn't try to support anything older than 2.6 and 3.3 at the same time. Python 2.6 is the first version which has enough compatibility with Python 3 to be easy enough to port forward.

The changes that might impact you the most are in the area of string handling. In Python 3 what was called **unicode** is now **str**. That means that every string is Unicode – i.e. that *u'foobar'* [2] and *'foobar'* mean the same thing.

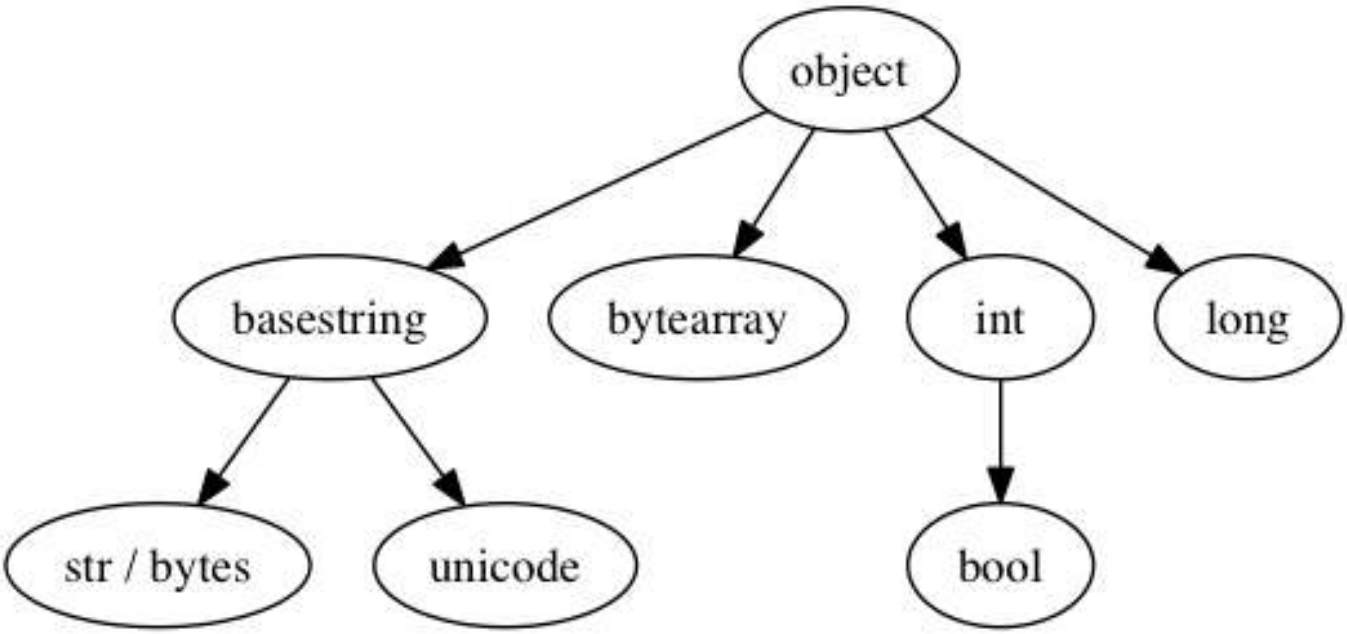


Figure 13.1: Python 2 base classes

[2]The *u* prefix was removed in Python 3.0 but reintroduced in Python 3.3 – see PEP 414

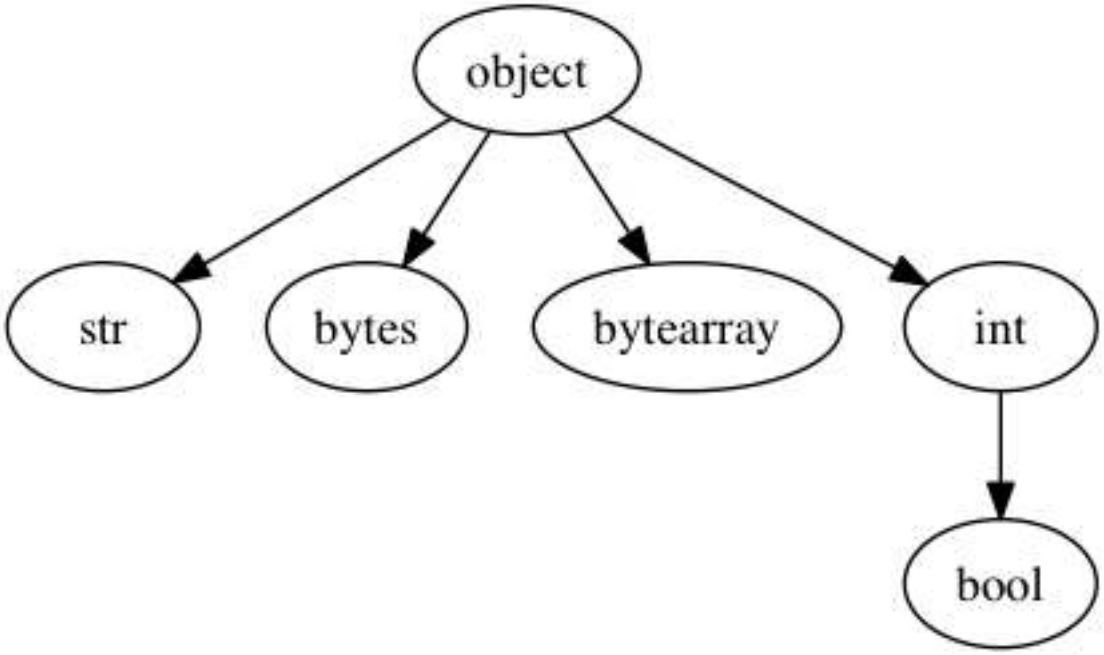


Figure 13.2: Python 3 base classes

Classes implementing *unicode* should rename that function to *str*, since the former isn't used anymore; you can automate this with a class decorator along these lines:

```
# -*- encoding: utf-8 -*-
import six

# This backports your Python 3 __str__ for Python 2
def unicode_compat(klass):
    if not six.PY3:
        klass.__unicode__ = klass.__str__
        klass.__str__ = lambda self: self.__unicode__().encode('utf-8')
    return klass


@unicode_compat
class Square(object):
    def __str__(self):
```

```
        return u"■ " + str(id(self))
```

That way you implement just one method for all Python versions returning Unicode, and the decorator handles the compatibility issue.

Another trick that can be handy when dealing with Python and Unicode is to use the *unicode_literals* function, which is available starting with Python 2.6 [3].

```
>>> 'foobar'
'foobar'
>>> from __future__ import unicode_literals
>>> 'foobar'
u'foobar'
```

Various functions no longer return lists, instead returning iterable objects (such as *range*); in addition, dictionary methods like *keys* or *items* now return iterable objects, and functions like *iterkeys* and *iteritems* have been removed. This is a big change, but *six* (discussed in Section 13.3) can help you with handling it.

Obviously, the standard library has evolved between Python 2 and Python 3, but that shouldn't be a huge concern. Some modules have been renamed or moved, but in the end the result is a clearer layout. There's no official listing that I'm aware of, but you can find a pretty good list here, or use a search engine.

The six module, which we will discuss in Section 13.3, will also help a lot when trying to maintain compatibility between Python 2 & 3.

13.2 External libraries

Your first enemies are the external libraries you depend on. If you read my advice in Section 2.3 and followed my check-list, you won't have a problem here – since

[3] Another reason not to support older versions?

that check-list included a Python 3 support requirement. However, you may have started a project earlier and have already made the mistake.

Unfortunately there isn't any magic trick than can resolve the problem. Luckily, if you followed my other advice, you isolated this library enough that it is not spread across your whole code base; so you can think about replacing it. Indeed, this may be your best move if the library does not show a strong possibility of supporting Python 3. However, small and medium-sized libraries might be more easily ported to Python 3 than big frameworks, so you may want to cut your teeth on them.

When looking for packages on PyPI, you can check for the trove classifiers *"Programming Language :: Python :: 2"* and *"Programming Language :: Python :: 3"*, which indicate which version of Python the package supports. However, be careful that these may not be up to date.

One of the external library choices made at the beginning of the OpenStack project was eventlet, a concurrent networking library. It has no support for Python 3, and still tries to support Python 2.5 – which, as you imagine, does not facilitate any transition. This choice was made a long time ago in OpenStack, before any kind of checks for Python 3 compatibility were done; and we already know that this module is going a big issue in the months ahead. As of yet, we have no concrete plan on how to fix it.

Don't make the same mistake!

13.3 Using six

As we have seen, Python 3 breaks compatibility with earlier versions and shifts things around. However, the basics of the language haven't changed, so it is possible to have a sort of transition layer; a module that can implement forward and backward compatibility – a bridge between Python 2 and Python 3.

This module exists, and it's called **six** – because two times three equals six.

The first thing that *six* provides is the **six.PY3** variable. This is a boolean which indicates whether we are running Python 3 or not. This is the pivot variable for any of your code base that has two versions, one for Python 2 and one for Python 3. However, be careful not to abuse it; scattering your code base with *if six.PY3* is going to be difficult to work with later.

As we discussed in Section 8.1, which concerned generators, Python 3 has a great feature whereby iterable objects are returned instead of lists. That means that methods like *dict.iteritems* are gone, and that *dict.items* returns an iterator rather than a list. Obviously this can break your code. *six* provides *six.iteritems* for such cases, so that all you have to do is to replace the following code:

```
for k, v in mydict.iteritems():
    print(k, v)
```

with:

```
import six

for k, v in six.iteritems(mydict):
    print(k, v)
```

And voilà, Python 3 compliance achieved in a snap! *six* provides a lot of similar helper functions that can increase compatibility across Python versions.

The **raise** syntax also changed in Python 3 [4], so re-raising exceptions should be done using *six.reraise*.

If you are using metaclasses, Python 3 has also changed this completely. Six has a nice trick for handling the transition – for example, if you are using the **abc** abstract base classes metaclass, here's how you would use *six*:

```
import abc
from six import with_metaclass
```

[4] It now only accepts one argument, an exception.

```
class MyClass(with_metaclass(abc.ABCMeta, object)):
    pass
```

One cannot discuss Python 3 without touching on the string and unicode mess that it solved. In Python 2, the basic type for string is `str` which can handle only basic ASCII strings, and the type `unicode`, added later, handles real string of text. In Python 3, the basic 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 characters stream.

six provides a nice set of functions and constants to handle the transition, such as *six.u* and *six.string_types*. The same compatibility is provided for integers, with *six.integer_types* that will handle the *long* type that has been removed from Python 3.

As discussed in Section 13.1, some modules have moved, and *six* provides a nice module called *six.moves* that handles a lot of these moves transparently.

For example, the *ConfigParser* module in Python 3 has been renamed to *configparser*. Code using *ConfigParser* under Python 2:

```
from ConfigParser import ConfigParser

conf = ConfigParser()
```

can be ported and made compatible with both major Python versions:

```
from six.moves.configparser import ConfigParser

conf = ConfigParser()
```

Tip

It is also possible to add your own moves via *six.add_move* to handle other transitions.

The *six* library might not be enough or cover all your use case. In this case, building a compatibility module encapsulating *six* itself might be worth it. By isolating the this in one particular module, you are assuring that you'll be able to enhance it for future version of Python, or dispose (part of) it when you'll want to stop supporting a particular version of Python. Also note that *six* is open source and that you can contribute to it rather than maintaining your own hacks.

The last thing I'll mention, is the modernize module. It's a thin wrapper around *2to3* that "modernizes" code by porting to Python 3; but rather than convert the syntax to Python 3 code only, it uses the *six* module. It's a better choice than the standard *2to3* tool, and get your port off to a strong start by carrying out most of the grunt work for you. It's worth a shot.

14 Write less, code more

In this section I've compiled a few of the more advanced features that I find interesting – they'll help you to write better code.

14.1 Single dispatcher

I often like to say that Python is a good subset of Lisp, and as time passes I find this to be more and more true. Recently I stumbled across the PEP 443, which describes a way to dispatch generic functions in a similar manner to that provided by CLOS, the Common Lisp Object System.

If you're familiar with Lisp, this won't be new to you. The Lisp object system, which is one of the basic components of Common Lisp, provides a good way to define and handle method dispatching. I'll show you how generic methods work in Lisp first – even if only for the pleasure of including Lisp code in a book on Python!

To begin with, let's define a few very simple classes, without any parent classes or attributes

```
(defclass snare-drum ()
  ())

(defclass cymbal ()
  ())
```

```
(defclass stick ()
  ())

(defclass brushes ()
  ())
```

This defines a few classes: `snare-drum`, `symbal`, `stick` and `brushes`, without any parent class nor attribute. These classes compose a drum kit, and we can combine them to play sound. So 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: it 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 can be called with various arguments. Now 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!")
```

Now we've defined concrete methods in code. They take two arguments: `instrument`, which is an instance of `snare-drum`; 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 any class can implement them.

Let's try it.

```
* (play (make-instance 'snare-drum) (make-instance 'stick))
"POC!"

* (play (make-instance 'snare-drum) (make-instance 'brushes))
"SHHHH!"

* (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]
```

As you can see, which function is called depends on the class of the arguments – the object systems **dispatch** the function calls to the right function for us, depending which classes we pass as arguments. If we call `play` with instances that are not known to the object system, an error will be thrown.

Inheritance is also supported and, the (more powerful and less error prone) equivalent of Python's `super()` is available via `(call-next-method)`.

```
(defclass snare-drum () ())
(defclass cymbal () ())

(defclass accessory () ())
(defclass stick (accessory) ())
(defclass brushes (accessory) ())

(defmethod play ((c cymbal) (a accessory))
  "BIIING!")

(defmethod play ((c cymbal) (b brushes))
  (concatenate 'string "SSHHHH!" (call-next-method)))
```

In this example, we define the `stick` and `brushes` classes as subclasses of `accessory`. The `play` method defined will return the sound *BIIING!*, regardless of what kind of accessory instance is used to play the cymbal – except if it's a `brushes` instance; the most precise method is always called. The `(call-next-method)` function is used to call the closest parent method, and in this case that would be the method which returns *"BIIING!"*.

```
* (play (make-instance 'cymbal) (make-instance 'stick))
"BIIING!"

* (play (make-instance 'cymbal) (make-instance 'brushes))
"SSHHHH!BIIING!"
```

Note that CLOS can define specialized methods which apply to only one instance of a class- using the `eql` specializer.

But if you're really curious about the many features CLOS provides, I suggest that you read the brief guide to CLOS by Jeff Dalton as a starter.

Python implements a simpler version of this workflow with the `singledispatch` function, which will is with Python 3.4 as part of the `functools` module. Here's the rough equivalent of the Lisp program above:

```
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")

@play.register(SnareDrum)
def _(instrument, accessory):
    if isinstance(accessory, Stick):
        return "POC!"
    if isinstance(accessory, Brushes):
        return "SHHHH!"
    raise NotImplementedError("Cannot play these")
```

We define our four classes, and a base `play` function that raises `NotImplementedError`, indicating that by default we don't know what to do. We can then write a specialized version of this function for a specific instrument, the `SnareDrum`. This function checks which accessory type has been passed, and returns the appropriate sound – or raises `NotImplementedError` again if it doesn't recognise the accessory.

If we run the program, it should work as follows:

```
>>> play(SnareDrum(), Stick())
'POC!'
```

```
>>> play(SnareDrum(), Brushes())
'SHHHH!'
>>> play(Cymbal(), Brushes())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jd/Source/cpython/Lib/functools.py", line 562, in wrapper
    return dispatch(args[0].__class__)(*args, **kw)
  File "/home/jd/sd.py", line 10, in play
    raise NotImplementedError("Cannot play these")
NotImplementedError: Cannot play these
>>> play(SnareDrum(), Cymbal())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jd/Source/cpython/Lib/functools.py", line 562, in wrapper
    return dispatch(args[0].__class__)(*args, **kw)
  File "/home/jd/sd.py", line 18, in _
    raise NotImplementedError("Cannot play these")
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 which is run – so, if our instrument is an instance of a class that we did not register, this base function will be called.

For those eager to try it out, the `singledispatch` function is provided in Python 2.6 to 3.3, through the Python Package Index.

As we saw in the Lisp version of the code, CLOS provides a multiple dispatcher that can dispatch depending on the type of **any of the arguments** defined in the method prototype, not just the first one. Unfortunately, the Python dispatcher is named *singledispatch* for a good reason: it only knows how to dispatch based on the first

argument. Guido van Rossum wrote a short article called multimethod about this subject a few years ago.

In addition, there's no way to call the parent function directly – no equivalent of either `(call-next-method)` from Lisp, or the Python `super()` function. You'll have to use various tricks to bypass this limitation.

To conclude: while I am really glad that Python is heading in this direction, as it's a really powerful way to enhance an object system, it still lacks a lot of the more advanced features that CLOS provides out of the box.

14.2 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 the language. Python provides a similar-looking mechanism, with the use of objects which 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 are called at the start of the `with` block and at the end of it, respectively.

A simple implementation of a context object would be:

Example 14.1 Simple implementation of a context object

```
class MyContext(object):
    def __enter__(self):
        pass
```

```
    def __exit__(self, exc_type, exc_value, traceback):
        pass
```

It wouldn't do anything, but is valid.

When do you want to use context managers? The use of context management protocol might be appropriate if you identify the following pattern in your object:

1. Call method A;
2. Execute some code;
3. Call method B.

Where it is expected that a **call to method B** must **always** be done after a **call to A**. The `open` function illustrates this pattern well – in this case, 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.

The **`contextlib`** standard library provides **`contextmanager`** to ease the implementation of such a mechanism, by relying on a generator to construct the `__enter__` and `__exit__` methods for you. We can use this to implement our simple context manager:

Example 14.2 Simplest usage of `contextlib.contextmanager`

```
import contextlib


@contextlib.contextmanager
def MyContext():
    yield
```

For example, I've been using this design pattern in Ceilometer for the pipeline infrastructure we set up. Basically, a pipeline is a tube into which objects are put, and

from which they are dispatched to various places. The steps to send data this way are as follows:

1. Call the `publish(objects)` method of a pipeline, with your objects as arguments – as many times as you need.
2. Once done, call the `flush()` method to indicate that you're done publishing for now.

Note that if you never call the `flush()` method, your objects will never be sent down the tube; or at least not completely. It can be very easy for a programmer to forget about a `flush()` call, which breaks the program without giving any clues as to what might be wrong.

It's much better if your API provides a context manager object that will not allow the API user to make this mistake. This can be done pretty easily with the following code:

Example 14.3 Using a context manager on a pipeline object

```
import contextlib

class Pipeline(object):
    def _publish(self, objects):
        # Imagine publication code here
        pass

    def _flush(self):
        # Imagine flushing code here
        pass

    @contextlib.contextmanager
    def publisher(self):
```

```
        try:
            yield self._publish
        finally:
            self._flush()
```

Now, when users of our API wants to publish something in our pipeline, they don't have to use `_publish` or `_flush`. They just request a publisher using the eponym function, and uses it.

```
pipeline = Pipeline()
with pipeline.publisher() as publisher:
    publisher([1, 2, 3, 4])
```

When you provide an API like this, there's no place for user error. Always use context managers when you see that it suits the design pattern.

In some contexts, it might be useful to use several context managers at the same time – for example, opening two files at the same time to copy their content:

Example 14.4 Opening two files at the same time

```
with open("file1", "r") as source:
    with open("file2", "w") as destination:
        destination.write(source.read())
```

Remember that the `with` statement supports having multiple arguments – so you should write:

Example 14.5 Opening two files at the same time with one `with` statement

```
with open("file1", "r") as source, open("file2", "w") as destination:
    destination.write(source.read())
```