# 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 operations, 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

```python
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

```python
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.