Colliberty
Stockholm Kraków

Porting to Python 3 Second edition

An in-depth guide

Lennart Regebro

Foreword by Brett Cannon

Porting to Python 3

Lennart Regebro

Colliberty
Ulica Retoryka 15/5
31-108 Kraków
Poland
e-mail: info@colliberty.com

Second edition, Revision 1.2
July 12, 2013

CONTENTS

ABOUT THIS BOOK

This book is written in reStructuredText[1], typeset with Sphinx[2] and LaTeX[3] and printed on CreateSpace[4]. The typefaces used are TeX Gyre Schola for the main text, DejaVu Sans Mono for the code and Flux Bold for headings.

The cover photo is taken by **Emmanuel "Tambako" Keller**, http://www.flickr.com/photos/tambako/

Almost all the code examples are unit tested, but there are likely to be many bugs in the text, so there will be a list of errata on the books web page, http://python3porting.com/.

I would like to thank the following persons for helping making this book much better:

Brett Cannon for writing the foreword.

Martin von Löwis for indispensable feedback with his technical review of the book, as well as helping a lot on early porting work, like Distribute and zope.interface.

Godefroid Chapelle, **Jasper Spaans** and **Wyn Williams** for reviewing the book for content, grammar and errors.

Brandon Craig Rhodes for editing the first chapters of this book when the idea was that it would be a series of articles in Python Magazine.

[1] http://docutils.sourceforge.net/rst.html
[2] http://sphinx.pocoo.org/
[3] http://www.latex-project.org/
[4] https://www.createspace.com/

FOREWORD

By Brett Cannon

When I joined the python-dev mailing list back in June 2002, the term “Python 3000” was brought out every few months to represent an idea the Python development team wished they could act on, but for compatibility reasons could not. Saying we could do something “maybe for Python 3000” meant it had no chance of happening.

But then we started to say something could happen in Python 3000 more and more often. Eventually it got to the point that “Python 3000” was referenced so often on python-dev that the acronym “py3k” was created out of our inherent programmer’s laziness. Then we began to believe our own hype about what py3k would be like and how much better it would be. It got to the point where Andrew Kuchling created PEP 3100 (which was the original PEP 3000, which I eventually took responsibility for) to keep track of the various ideas we had for py3k in late 2004 as it was obvious around that time that we were actually going to go through with the “crazy” idea of making Python 3000 happen. This all led serious development starting in March 2006 and culminating in the release of Python 3.0 on December 3, 2008.

While all of this was happening, there was mixed feelings from the community about the feasibility/sanity of creating Python 3. When PEP 3100 was created in 2004, Python’s popularity took a very noticeable uptick. This trend continued and around 2006, when py3k development started in earnest, Python’s popularity exceeded that of Perl. So while Python was becoming one of the most popular dynamic programming languages in the world, the development team was beginning to create the next major version which would break compatibility with the version of the language that all of these people were now learning. Some people called us a little nuts for obvious reasons.

But we would like to think we knew what we were doing. While Python 2 is a great language, Guido and everyone on the development team knew it had its flaws (if it didn’t then we would not have been able to create a PEP with nearly 100 things we wanted to change). Guido also realized that more Python code would be written in the future then had been written to that point and into the future that would continue to be true. As a service to our community (and partly because it was fun) we decided we should try to fix some of our previous mistakes so that future Python code could be written better and faster than was possible with Python 2, hence why we created Python 3.

But while some considered us a little nuts to break compatibility with Python 2, We also realized that we didn’t want to leave our existing community behind and develop Python 3 only for new code. The development team knew as we created Python 3 that it was a superior language and so we wanted to share it with everyone by making sure they could bring their Python 2 code with them into their Python 3 work. From the beginning we made sure that changes we made could either be warned against in the worst case, and automated in the best. Techniques we learned and tools we developed were used to port Python’s extensive standard library so as to learn from our own mistakes and make sure

that other people could port their own code. We always kept in the back of our heads the goal of making porting Python 2 code as easy as possible.

The continual increase in the number of Python 3 projects available and the fact that all of the major Linux distributions ship with Python 3, or will do so in their next major release, is a testament that we didn't screw up. Guido always said it would take 3 to 5 years for Python 3 to gain traction within the community. The constant trend of Python 3 projects being released is a testament that the timeline Guido set out is turning out to be true as major libraries have already been ported, allowing their dependents to make the switch themselves.

While some might question the utility in moving to Python 2 code to Python 3, there are two things to keep in mind. One is that Python 3 is simply a nicer language than Python 2. While there are only a handful of major changes, it's all of the little changes that add up to make the experience of programming Python 3 that much more pleasant compared to Python 2. It's rather common to hear core developers say how they prefer coding in Python 3 over Python 2. I for one have simply stopped coding in Python 2 as it just feels slightly off compared to the more uniform feel of Python 3. And secondly, more code will be written in Python 3 than in Python 2 over the history of the Python language, so not porting means your project will eventually be left behind (this is already starting to happen for projects which have publicly said they will not switch, leading people to find alternatives for both their Python 2 and Python 3 code in make sure they can switch to Python 3 when ready). Sitting idly by as the world changes around you is not a good thing to do if you want to stay relevant.

I still remember the day that Python 3 was released. It was the end of the workday and I was on IRC in #python-dev waiting for Barry Warsaw, the Python 3.0 release manager, to flip the switch on the release. When it was hit, I just swivelled around in my chair and told Guido that it was done; Python 3 was no longer a dream but an actual thing. I stood up, we gave each other an ecstatic high-five, and just smiled (the next day people asked us at work what we were so giddy about that night).

At that moment, and to this day, the thought that Python 3 would flop or not be worth the amount of time and effort my fellow core developers and I put into it never crossed my mind. And the fact that people care enough about seeing Python 3 work that there is now a book dedicated to helping people get from Python 2 to Python 3 is a testament that Python 3 has not, and will not, flop.

CHAPTER

ONE

WELCOME TO PYTHON 3

On Christmas Day 1999 I sat down to write my first piece of software in Python. My experience seems to be typical for Python users. I was initially surprised that indentation was significant, it felt scary to not define variables and I was hesitant to use a dynamic language to make serious software. However, in no time at all these worries were gone and I noticed I wrote code faster than ever. 18 months later a friend hired me to his start-up to help him write a content management system and I ended up in the enviable position of being a full time Python developer. I don't even mention other languages on my CV anymore, because I don't want to use them. I've become a full fledged, fanatic, Pythonista.

I got interested in Python 3 at EuroPython 2007 in lovely Vilnius. Guido van Rossums keynote was about the upcoming changes in Python 3 and although he emphatically said that you could not run the same code under Python 2 and Python 3, I couldn't see many reasons why it couldn't be done, considering the forward compatibility that was planned for Python 2.6. So I started looking at the differences between Python 2 and Python 3 and tried to find out how to write code that would run under both versions and learned a whole lot about Python on the way.

Most surprising was how little the fundamentals have changed. Writing code with Python 3 still feels just like the old comfortable shoes, but newly shined and with new laces. The hardest change to get used to is to remember that `print` is a function. The relatively small differences doesn't necessarily mean that porting to Python 3 is easy, but it can be and hopefully this book will make it even easier.

1.1 Is it time yet?

Yes, Python 3 is a nicer language to work with. But Python 2 is also very good and the major reason for not porting yet is that Python 2 is so good that most developers feel little incentive to switch. Although it has been officially declared that Python 2.7 will be the last version of Python 2, it will receive bug-fixes for many years to come, so there is no hurry to change to Python 3 for that reason.

So when should you port? In general, I would recommend everyone to move to Python 3 as soon as you can. If the applications and modules you write are for your or your company's use only, then look into porting when it feels like you have the time. If your project is in a state of panic, moving to Python 3 is probably not the right thing to do.

If you are writing software that you sell or share as open source, then you want to move more quickly to enable your customers to move over to Python 3.

If you are writing a package that other *developers* use, every day it doesn't support Python 3 is a day when you are blocking your users from porting, and a day when Python 3 users have to look for another package than yours. In this case you should really try to port immediately, and if you have dependencies that is not ported, then help port them first.

1.2 What if I can't port right now?

In any case all the packages *you* depend on need to be ported before you can port. Most packages that have been ported to Python 3 are listed on the CheeseShop under the "Python 3 packages" heading[1]. That list is a list of all packages that includes `"Programming Language :: Python :: 3"` as a trove classifier in the package meta data. If your dependencies have not been ported it is a good idea to contact the maintainers of your dependencies to see what plans they have for porting. Perhaps they do already support Python 3, but didn't update their meta data? Maybe they just didn't know anyone was waiting for a Python 3 port? Maybe you can help porting?

It's always a good idea to publish information on your plans for porting on your software's homepage or in the description of the package on the CheeseShop. Include a list of your dependencies that aren't ported. That way your users can see if there is something they can help with. Open source all is about programmers helping each other; both using and contributing to each others software. A porting effort is no different.

And even if you aren't porting right now, there are things you can do already. Chapter 3, *Preparing for Python 3* (page 15) lists things you should change before porting, and Chapter 6 *Improving your code with modern idioms* (page 49) lists modern idioms in Python that you already can use, depending on what Python versions you need to support. To ease porting, many of the new functions and modules in Python 3 has been backported to Python 2.6 or Python 2.7, and the only thing that stops you from using this already is if you need to support older versions.

1.3 Python and its versions

Since I started writing this book, Python 2.7 and Python 3.2 has been released. For the purposes of this book, Python 2.6 and Python 2.7 can be seen as equal. So most of the times the book says Python 2.6, you can read that as Python 2.6 or Python 2.7.

Python 3.1 was released quite quickly after Python 3.0 and before any significant adoption of Python 3. Therefore it was decided to drop support for Python 3.0. As most platforms that support Python 3 already use Python 3.1 for that support it is unlikely that you ever need to care about Python 3.0. When running your tests under Python 3 you only have to run it with Python 3.1 and Python 3.2, and you can safely ignore Python 3.0. So when this book says Python 3, you can read that as Python 3.1 and later.

[1] http://pypi.python.org/pypi?:action=browse&c=533&show=all

1.4 Further resources

There is still very little documentation on how to port to Python 3. There is a short how-to in the Python 3.2 documentation at http://docs.python.org/dev/howto/pyporting.html. There is also page on the official Python wiki for porting notes at http://wiki.python.org/moin/PortingPythonToPy3k but it is still fairly empty.

If you need help, or if you want to help out, there is the `python-porting@python.org` mailing list. You can subscribe to it from http://mail.python.org/mailman/listinfo/python-porting.

CHAPTER

TWO

MIGRATION STRATEGIES

Making a new release of software that is backwards incompatible is a high risk strategy. When people need to rewrite their software, or maintain separate versions of their source code to support both versions of a language or a framework, the risk is that they never make the transition and stay on the old version forever, or worse, that they switch to another framework.

For that reason Python versions 2.6 and 2.7 include both several forward compatibility features to enable you to write code for both Python 2 and Python 3, as well as support for migrating in the form of `2to3`, a program and package that can convert code from Python 2 to Python 3. There are other techniques and strategies you can use and there are also different ways to use `2to3`. Which strategy to use depends very much on what type of software you are converting.

2.1 Only supporting Python 3

The easiest case is when you only need to support one version of Python at a time. In these cases you can just convert your code to Python 3 and forget about Python 2. With this strategy you will first use the `2to3` tool to make an automatic conversion of most of the changes and then fix every problem that remains manually in the Python 3 code. You will probably also want to go through all of the converted code and clean it up, as the `2to3` conversions may not always be the most elegant solution for your case.

2.2 Separate branches for Python 2 and Python 3

If you need to continue to support Python 2, the simplest case is having a branch in your source tree with the code for Python 2 and another branch for Python 3. You will then have to make every change on the two different branches, which is a bit more work, but feasible if the code doesn't change that often.

One problem with this strategy is that your distribution becomes complex, because you now have two distributions and you need to make sure that Python 3 users get the Python 3 version and Python 2 users get the Python 2 version of your package. Solutions for this are documented in *Distributing packages* (page 28).

2.3 Converting to Python 3 with 2to3

In complex cases you can support both Python 2 and Python 3 by maintaining the source code in a Python 2 version and converting it with `2to3` for Python 3 support. That means you will have to run `2to3` each time you have made a code change so you can test it under Python 3, but on the other hand `2to3` deals with many of the differences.

To do this you need a script that performs the `2to3` conversion, because doing all the steps manually quickly becomes really boring. Since you need to do the conversion every time you have changed something so you can run the tests under Python 3, you want to run the conversion only on the files that have been modified as the conversion can be rather slow. That means that the conversion script should compare time stamps on the files to see which ones have been modified and convert only them, which means the script can not be a trivial shell script.

You can of course write these conversion scripts yourself, but you might not need to. If you are using Distutils it has support for running `2to3` as a part of the build process. This also solves the distribution problems, as this way you can distribute only Python 2 code and `2to3` will be run on that code during install when installed on Python 3. That way you don't have to have separate packages or even two copies of the code in your package. *Distributing packages* (page 28) also has information on this.

However, the lazy coders approach here would be to use Distribute, as it includes some extensions to the `2to3`-story.

2.3.1 Using Distribute to support the 2to3 conversion

Distribute[1] is a fork of Phillip J. Eby's popular Setuptools package and provides Python 3 compatibility, as well as extensions simplifying the support of Python 2 and Python 3 from the same source. Basically what Distribute has done is to extend the principles of Distutils `build_py_2to3` command and integrated `2to3` into all parts of the packaging story.

These changes will be merged back into Setuptools during 2013, but at the time of writing Setuptools doesn't support Python 3.

With Distribute you can add a few extra parameters in the `setup.py` file to have `2to3` run the conversion at build time. This means you only need to have one version of the source in your version control system and you therefore only need to fix bugs once. You also need only one source release, so you only have to release the software once and there is only one package to download and install for both Python 2 and Python 3.

You still need to run your tests under all versions of Python that you want to support, but Distribute includes a test command that will convert your code with `2to3` before running the tests. You can easily set up your package to use that. Then testing becomes just running `python setup.py test` once for every python version you want to support.

The main drawback with this solution is that you can't use the earliest versions of `2to3`, because they are too buggy. In practice it means you need to have Python 3.1 or later installed on the target machine. This is generally not a problem, as most platforms that support Python 3 already use Python 3.1 for that support.

[1] http://pypi.python.org/pypi/distribute

You can find examples of how to set up your module or package to use Distribute for your Python 3 support under *Supporting multiple versions of Python with Distribute* (page 30) as well as in the standard Distribute documentation[2].

2.4 Python 2 and Python 3 without conversion

In many cases it's often perfectly feasible to modify the code so that it runs under both Python 2 and Python 3 without needing any conversion, although you have to apply several tricks to avoid the incompatibilities between Python 2 and Python 3.

Python 2.6 and 2.7 has a lot of forward compatibility, making supporting Python 2.6 and Python 3 much easier than supporting Python 2.5 and Python 3. Supporting 2.5 or even older versions means you have to employ more tricks. Python 3.3 also re-introduces the `u''` literal for strings, which helps with one of the major porting difficulties.

Benjamin Petersons excellent `six` module[3] also helps by wrapping much of the incompatibilities, and since the need to support older Python versions is shrinking, supporting both Python 2 and Python 3 without conversion is becoming the preferred method.

There are also cases where you can't use Distribute, or don't want to. You may need to distribute your code in a format that is not installable with Distutils and therefore not Distribute. In those cases you can't use Distributes `2to3` support and then using `2to3` is more work and not using `2to3` becomes a more attractive prospect.

Even if you do use `2to3` for your project as a whole, you still may end up with having to write some code so it runs on both Python 2 and Python 3 without conversion. This is useful for bootstrapping scripts and setup scripts or if your code generates code from strings, for example to create command line scripts. You can of course have two separate strings depending on the Python version, or even run `2to3` on the string using `lib2to3`. However, in these cases it's generally easier to make the generated code snippets run on all Python versions without `2to3`.

My recommendation for the development workflow if you want to support Python 3 without using `2to3` is to run `2to3` on the code once and then fix it up until it works on Python 3. Only then introduce Python 2 support into the Python 3 code, using `six` where needed. Add support for Python 2.7 first, and then Python 2.6. Doing it this way can sometimes result in a very quick and painless process.

There is also a tool called `python-modernize` which will do a `2to3`-type conversion of your code, but it will keep Python 2 compatibility together with the `six` library. This can be a good start.

More information on the techniques necessary to do this is in the chapter *Supporting Python 2 and 3 without 2to3 conversion* (page 57).

[2] http://packages.python.org/distribute/python3.html
[3] http://pypi.python.org/pypi/six

2.5 Using 3to2

The `2to3` tool is flexible enough for you to define what changes should be done by writing "fixers". Almost any kind of Python code conversion is imaginable here and `3to2`[4] is a set of fixers written by Joe Amenta that does the conversion from Python 3 to Python 2. This enables you to write your code for Python 3 and then convert it to Python 2 before release.

However, there is no Distribute support for `3to2` and also Python 2.5 or earlier also do not include the required `lib2to3` package. Therefore `3to2` currently remains only an interesting experiment, although this may change in the future.

2.6 Which strategy is for you?

2.6.1 Applications

Unless your code is a reusable package or framework you probably do not need to support older versions of Python, unless some of your customers are stuck on Python 2 while others demand that you support Python 3. In most cases of you can just drop Python 2 support completely.

2.6.2 Python modules and packages

If you are developing some sort of module or package that other Python developers use you would probably like to support both Python 2 and Python 3 at the same time. The majority of your users will run Python 2 for some time to come, so you want to give them access to new functionality, but if you don't support Python 3, the users of Python 3 must find another package to fulfill their need.

If the package is stable from a functional standpoint, it might be perfectly reasonable to have separate branches in your version control system and make bugfixes on both branches separately, but if your package is under active development you probably want to support both Python 2 and Python 3 at the same time from the same code base. If you want to use the official `2to3` conversion method, or if you want to try to get the code running under both Python 2 and Python 3 without a conversion step depends on what your code does and what versions of Python you need to support.

There are cases where you won't be able to run the same code under Python 2 and Python 3 without a lot of effort because it relies so much on Python internals that the code becomes too different. There are also cases where the code is so straightforward that running `2to3` on it hardly changes it. Most code is somewhere in between and the decision is not always easy. A good idea is to run `2to3` on your code and look at the differences. If `2to3` makes a lot of changes in your code, then you may want to use it to convert the code to minimize the amount of workarounds.

If you can support only Python 2.6 and later then supporting Python 3 without `2to3` conversion is probably the best option. Especially if you aren't much affected by the binary/Unicode switch, or if you only need to support Python 3.3 or later.

[4] http://pypi.python.org/pypi/3to2

If you are already releasing your package using Distutils or its descendants Setuptools and Distribute, then using Distribute's `2to3` support is easy, and that might be the path of least resistence. You can also start that way and change the code bit by bit to something that doesn't need converting.

2.6.3 Frameworks

The benefit of using frameworks when developing doesn't only come from the framework itself, but also from the plugins and extensions available to it. It is therefore important to make it easy for all developers using and extending the framework to switch to Python 3, as you otherwise risk to be stuck on Python 2 forever.

If your framework is extended by writing Python packages that uses Distutils, Setuptools or Distribute as a packaging system this means the users of your framework are already in a good position, as they can use Distutils or Distribute for the `2to3` conversion to support both new and old versions of the framework.

If extensions are packaged and distributed in some other way than with Distutils you may want to consider making your own set of support scripts to make the transition easier, or stopping support for older Python versions, so that your third-party package developers doesn't have to use `2to3`.

2.7 Summary

In general, if you write end-user software, you can just switch to Python 3, starting with a one-time run of `2to3` on your code. If you write a Python package you want to support both Python 2 and Python 3 at the same time, and you can drop Python 2.5 support, try first to support Python 2 and 3 without `2to3` conversion.

If you need to support Python 2.5 or older, using `2to3` is often the best option.

CHAPTER

THREE

PREPARING FOR PYTHON 3

Before you start the actual porting there are several things you should do to prepare your code to make the transition to Python 3 as smooth as possible, by changing things that is hard for `2to3` to port correctly. These are things you can do right now even if you don't plan to move to Python 3 yet and in some cases they will even speed up your code under Python 2.

You might also want to read the chapter on *Improving your code with modern idioms* (page 49), which contains many other improvements you can do to your code even before porting.

3.1 Run under Python 2.7.

The first step in the porting process is to get your code running in Python 2.6 or 2.7. It isn't very important which version you use here, but obviously the last Python 2 version makes the most sense, so if you can use Python 2.7, do so.

Most code will run directly without modifications, but there are a couple of things that change from Python 2.5 to 2.6. In Python 2.6 `as` and `with` are keywords, so if you use these as variables you will need to change them. The easiest is to add an underscore to the end of the variables.

```
>>> with_ = True
>>> as_ = False
```

You also need to get rid of string exceptions. Using strings for exceptions has been recommended against for a very long time, mainly because they are very inflexible, you can't subclass them for example.

```
>>> raise "Something went wrong!"
Traceback (most recent call last):
...
Something went wrong!
```

In Python 3 string exceptions are completely gone. In Python 2.6 you can't raise them, although you can still catch them for backwards compatibility. In any case you should remove all usage of string exceptions in your code and make it run under Python 2.6 before doing anything else.

```
>>> raise Exception("Something went wrong!")
Traceback (most recent call last):
...
Exception: Something went wrong!
```

The next step is to run your code under Python 2.6 or Python 2.7 with the `-3` option. This will warn about things that are not supported in Python 3 and which `2to3` will not convert. It's mostly ways of doing things that have long been deprecated and have newer alternative ways to be done, or modules removed from the standard library. For example the support for Classic Mac OS has been removed in Python 3, only OS X is supported now and for that reasons the modules that support specific things about Classic Mac OS have been removed.

You will get warnings for many of the changes listed below, but not all, as well as for some of the library reorganization. The library reorganization changes are simple and need no explanation, the warnings will tell you the new name of the module.

3.2 Use // instead of / when dividing integers

In Python 2 dividing two integers returns an integer. That means five divided by two will return two.

```
>>> 5/2
2
```

However, under Python 3 it returns two and a half.

```
>>> 5/2
2.5
```

Many who use the division operator in Python 2 today rely on the integer division to always return integers. But the automatic conversion with `2to3` will not know what types the operands are and therefore it doesn't know if the division operator divides integers or not. Therefore it can not do any conversion here. This means that if you are using the old integer division, your code may fail after porting to Python 3.

Since this change has been planned already since Python 2.2, it and all later versions include a new operator, called *floor division*, written with two slashes. It always returns whole integers, even with floats. Any place in your code where you really do want to have the floor division that returns whole numbers, you should change the division operator to the floor division operator.

```
>>> 5//2
2

>>> 5.0//2.0
2.0
```

Often the Python 2 integer division behavior is unwanted. The most common way to get around that problem is to convert one of the integers to a float, or to add a decimal comma to one of the numbers.

```
>>> 5/2.0
2.5

>>> a = 5
>>> b = 2
>>> float(a)/b
2.5
```

However, there is a neater way to do this and that is to enable the Python 3 behavior. This is done via a `__future__` import also available since Python 2.2.

```
>>> from __future__ import division
>>> 5/2
2.5
```

Although converting one of the operands to a float before division will work fine it is unnecessary in Python 3 and by using the `__future__` import you can avoid it.

Running Python 2.6 with the `-3` option will warn you if you use the old integer division.

3.3 Use new-style classes

In Python 2 there is two types of classes, "old-style" and "new". The "old-style" classes have been removed in Python 3, so all classes now subclass from `object`, even if they don't do so explicitly.

There are many differences between new and old classes, but few of them will cause you any problems when porting. If you use multiple inheritance you are probably going to encounter problems because if the different method resolution orders.[1]

If you use multiple inheritance you should therefore switch to using new-style classes before porting. This is done by making sure all objects subclass from `object`, and you will probably have to change the order you list the super-classes in the class definitions.

3.4 Separate binary data and strings

In Python 2, you use `str` objects to hold binary data and ASCII text, while text data that needs more characters than what is available in ASCII is held in `unicode` objects. In Python 3, instead of `str` and `unicode` objects, you use `bytes` objects for binary data and `str` objects for all kinds of text data, Unicode or not. The `str` type in Python 3 is more or less the same as the `unicode` type in Python 2 and the `bytes` type is quite similar to Python 2's `str` type, even though there are significant differences.

The first step in preparing for this is to make sure you don't use the same variable name for both binary and text data. In Python 2 this will not cause you much trouble, but in Python 3 it will, so try to keep binary data and text separated as much as possible.

In Python 2 the `'t'` and `'b'` file mode flags changes how newlines are treated on some platforms, for example Windows. But the flag makes no difference on Unix, so many programs that are developed for Unix tends to ignore that flag and open binary files in text

[1] See http://www.python.org/download/releases/2.2.3/descrintro/#mro

mode. However, in Python 3 the flags also determine if you get `bytes` or `unicode` objects as results when you read from the file. So make sure you really use the text and binary flags when you open a file. Although the text flag is default, add it anyway, as you then show that the text mode is intentional and not just because you forgot to add the flag.

Running Python 2.6 with the `-3` option will *not* warn about this problem, as there simply is no way for Python 2 to know if the data is text or binary data.

3.5 When sorting, use `key` instead of `cmp`

In Python 2 sorting methods take a `cmp` parameter that should be a function that returns -1, 0 or 1 when comparing two values.

```
>>> def compare(a, b):
...     """Comparison that ignores the first letter"""
...     return cmp(a[1:], b[1:])
>>> names = ['Adam', 'Donald', 'John']
>>> names.sort(cmp=compare)
>>> names
['Adam', 'John', 'Donald']
```

Since Python 2.4 `.sort()` as well as the new `sorted()` function (see *Use sorted() instead of .sort()* (page 49)) take a `key` parameter which should be a function that returns a sorting key.

```
>>> def keyfunction(item):
...     """Key for comparison that ignores the first letter"""
...     return item[1:]
>>> names = ['Adam', 'Donald', 'John']
>>> names.sort(key=keyfunction)
>>> names
['Adam', 'John', 'Donald']
```

This is easier to use and faster to run. When using the `cmp` parameter, the sorting compares pairs of values, so the compare-function is called multiple times for every item. The larger the set of data, the more times the compare-function is called per item. With the `key` function the sorting instead keeps the key value for each item and compares those, so the key function is only called once for every item. This results in much faster sorts for large sets of data.

The `key` function is often so simple that you can replace it with a lambda:

```
>>> names = ['Adam', 'Donald', 'John']
>>> names.sort(key=lambda x: x[1:])
>>> names
['Adam', 'John', 'Donald']
```

Python 2.4 also introduced a `reverse` parameter.

```
>>> names = ['Adam', 'Donald', 'John']
>>> names.sort(key=lambda x: x[1:], reverse=True)
>>> names
['Donald', 'John', 'Adam']
```

There is one case where using `key` is less obvious than using `cmp` and that's when you need to sort on several values. Let's say we want the result to be sorted with the longest names first and names of the same length should be sorted alphabetically. Doing this with a `key` function is not immediately obvious, but the solution is usually to sort twice, with the least importing sorting first.

```
>>> names = ['Adam', 'Donald', 'John']
>>> # Alphabetical sort
>>> names.sort()
>>> # Long names should go first
>>> names.sort(key=lambda x: len(x), reverse=True)
>>> names
['Donald', 'Adam', 'John']
```

This works because since Python 2.3 the timsort sorting algorithm is used[2]. It's a stable algorithm, meaning that if two items are sorted as equal it will preserve the order of those items.

You can also make a key function that returns a value that combines the two keys and sort in one go. This is surprisingly not always faster, you will have to test which solution is faster in your case, it depends on both the data and the key function.

```
>>> def keyfunction(item):
...     """Sorting on descending length and alphabetically"""
...     return -len(item), item
>>> names = ['Adam', 'Donald', 'John']
>>> names.sort(key=keyfunction)
>>> names
['Donald', 'Adam', 'John']
```

The `key` parameter was introduced in Python 2.4, so if you need to support Python 2.3 you can't use it. If you need to do a lot of sorting using the key function, the best thing is to implement a simple `sorted()` function for Python 2.3 and use that conditionally instead of the `sorted()` builtin in with Python 2.4 and later.

```
>>> import sys
>>> if sys.version < '2.4':
...    def sorted(data, key):
...        mapping = {}
...        for x in data:
...            mapping[key(x)] = x
...        keys = mapping.keys()
...        keys.sort()
...        return [mapping[x] for x in keys]
>>> data = ['ant', 'Aardvark', 'banana', 'Dingo']
>>> sorted(data, key=str.lower)
['Aardvark', 'ant', 'banana', 'Dingo']
```

Python 2.4 is over five years old now, so it is quite unlikely that you would need to support Python 2.3.

[2] http://en.wikipedia.org/wiki/Timsort

Warning: Running Python with the `-3` option will only warn you if you use the `cmp` parameter explicitly:

```
>>> l.sort(cmp=cmpfunction)
__main__:1: DeprecationWarning: the cmp argument is not
supported in 3.x
```

But it will not warn if you use it like this:

```
>>> l.sort(cmpfunction)
```

So this syntax may slip through. In these cases you get a `TypeError: must use keyword argument for key function` when running the code under Python 3.

In Python 2.7 and Python 3.2 and later there is a function that will convert a comparison function to a key function via a wrapper class. It is very clever, but will make the compare function even slower, so use this only as a last resort.

```
>>> from functools import cmp_to_key
>>> def compare(a, b): return cmp(a[1:], b[1:])
>>> sorted(['Adam', 'Donald', 'John'], key=cmp_to_key(compare))
['Adam', 'John', 'Donald']
```

3.6 Use rich comparison operators

In Python 2 the most common way to support comparison and sorting of your objects is to implement a `__cmp__()` method that in turn uses the builtin `cmp()` function, like this class that will sort according to `lastname`:

```
>>> class Orderable(object):
...
...     def __init__(self, firstname, lastname):
...         self.first = firstname
...         self.last = lastname
...
...     def __cmp__(self, other):
...         return cmp("%s, %s" % (self.last, self.first),
...                    "%s, %s" % (other.last, other.first))
...
...     def __repr__(self):
...         return "%s %s" % (self.first, self.last)
...
>>> sorted([Orderable('Donald', 'Duck'),
...         Orderable('Paul', 'Anka')])
[Paul Anka, Donald Duck]
```

However, you can have objects, for example colors, that are neither “less than” nor “greater than”, but still can be “equal” or “not equal”, so since Python 2.1 there has also been support for rich comparison methods where each method corresponds to one comparison operator. They are `__lt__` for `<`, `__le__` for `<=`, `__eq__` for `==`, `__ne__` for `!=`, `__gt__` for `>` and `__ge__` for `>=`.

Having both the rich comparison methods and the `__cmp__()` method violates the principle that there should be only one obvious way to do it, so in Python 3 the support for `__cmp__()` has been removed. For Python 3 you therefore must implement all of the rich comparison operators if you want your objects to be comparable. You don't have to do this before porting but doing so makes the porting experience a bit smoother.

3.6.1 Comparatively tricky

Making comparison methods can be quite tricky, since you also need to handle comparing different types. The comparison methods should return the `NotImplemented` constant if it doesn't know how to compare with the other object. Returning `NotImplemented` works as a flag for Pythons comparisons that makes Python try the reverse comparison. So if your `__lt__()` method returns `NotImplemented` then Python will try to ask the other objects `__gt__()` method instead.

> **Attention:** This means that you should never in your rich comparison methods call the other objects comparison operator! You'll find several examples of rich comparison helpers that will convert a greater than call like `self.__gt__(other)` into `return other < self`. But then you are calling `other.__lt__(self)` and if it returns `NotImplemented` then Python will try `self.__gt__(other)` again and you get infinite recursion!

So implementing a good set of rich comparison operators that behave properly in all cases is not difficult once you understand all the cases, but getting to grips with that is not entirely trivial. You can do it in many different ways, my preferred way is this mixin, which works equally well in Python 2 and Python 3.

```
class ComparableMixin(object):
    def _compare(self, other, method):
        try:
            return method(self._cmpkey(), other._cmpkey())
        except (AttributeError, TypeError):
            # _cmpkey not implemented, or return different type,
            # so I can't compare with "other".
            return NotImplemented

    def __lt__(self, other):
        return self._compare(other, lambda s, o: s < o)

    def __le__(self, other):
        return self._compare(other, lambda s, o: s <= o)

    def __eq__(self, other):
        return self._compare(other, lambda s, o: s == o)

    def __ge__(self, other):
        return self._compare(other, lambda s, o: s >= o)

    def __gt__(self, other):
        return self._compare(other, lambda s, o: s > o)
```

```
    def __ne__(self, other):
        return self._compare(other, lambda s, o: s != o)
```

The previously mentioned `functools.total_ordering()` class decorator from Python 3.2 is a nice solution as well, and it can be copied and used other Python versions as well. But since it uses class decorators it will not work in version below Python 2.6.

To use the mixin above you need to implement a `_cmpkey()` method that returns a key of objects that can be compared, similar to the `key()` function used when sorting. The implementation could look like this:

```
>>> from mixin import ComparableMixin

>>> class Orderable(ComparableMixin):
...
...     def __init__(self, firstname, lastname):
...         self.first = firstname
...         self.last = lastname
...
...     def _cmpkey(self):
...         return (self.last, self.first)
...
...     def __repr__(self):
...         return "%s %s" % (self.first, self.last)
...
>>> sorted([Orderable('Donald', 'Duck'),
...         Orderable('Paul', 'Anka')])
[Paul Anka, Donald Duck]
```

The above mixin will return `NotImplemented` if the object compared with does not implement a `_cmpkey()` method, or if that method returns something that isn't comparable with the value that `self._cmpkey()` returns. This means that every object that has a `_cmpkey()` that returns a tuple will be comparable with all other objects that also has a `_cmpkey()` that returns a tuple and most importantly, if it isn't comparable, the operators will fall back to asking the other object if it knows how to compare the two objects. This way you have an object who has the maximum chance of meaningful comparisons.

3.6.2 Implementing `__hash__()`

In Python 2, if you implement `__eq__()` you should also override `__hash__()`. This is because two objects that compare equal should also have the same hash-value. If the object is mutable, you should set `__hash__` to `None`, to mark it as mutable. This will mean you can't use it as a key in dictionaries for example, and that's good, only immutable objects should be dictionary keys.

In Python 3, `__hash__` will be set to `None` automatically if you define `__eq__()`, and the object will become unhasheable, so for Python 3 you don't need to override `__hash__()` unless it is an immutable object and you want to be able to use it as a key value.

The value returned by `__hash__()` needs to be an integer, and two objects that compare equal should have the same hash value. It must stay the same over the whole lifetime

of the object, which is why mutable objects should set `__hash__ = None` to mark them as unhashable.

If you are using the `_cmpkey()` method of implementing comparison operators mentioned above, then implementing `__hash__()` is very easy:

```
>>> from mixin import ComparableMixin

>>> class Hashable(ComparableMixin):
...     def __init__(self, firstname, lastname):
...         self._first = firstname
...         self._last = lastname
...
...     def _cmpkey(self):
...         return (self._last, self._first)
...
...     def __repr__(self):
...         return "%s(%r, %r)" % (self.__class__.__name__,
...                                self._first, self._last)
...
...     def __hash__(self):
...         return hash(self._cmpkey())
...
>>> d = {Hashable('Donald', 'Duck'): 'Daisy Duck'}
>>> d
{Hashable('Donald', 'Duck'): 'Daisy Duck'}
```

The attributes of this class is marked as internal by the convention of using a leading underscore, but they are not strictly speaking immutable. If you want a truly immutable class in Python the easiest way is subclassing `collections.namedtuple`, but that is out of scope for this book.

3.7 Make sure you aren't using any removed modules

Many of the modules in the standard library have been dropped from Python 3. Most of them are specific to old operating systems that aren't supported any more and others have been supplanted by new modules with a better interface.

Running Python 2.6 with the `-3` option will warn you if you use some of the more commonly used modules. It's quite unlikely that you are using any of the modules that Python 2.6 will not warn about, but if you are and you are planning to support both Python 2 and Python 3, you should replace them with their modern counterparts, if any.

See *Removed modules* (page 100) for a list of the removed modules.

3.8 Testing `coverage` and `tox`

Having a good set of tests is always valuable for any project. When it comes to porting to Python 3, having tests is going to speed up the process a lot, because you will need to run the tests over and over and testing an application by hand takes a lot of time.

It's always a good idea to increase the test coverage with more tests. The most popular Python tool for getting a report on the test coverage of your modules is Ned Batchelder's `coverage` module.[3] Many test runner frameworks like zope.testing, nose and py.test include support for the `coverage` module, so you may have it installed already.

If you are porting a module that supports many versions of Python, running the tests for all these versions quickly becomes a chore. To solve this Holger Krekel has created a tool called `tox`[4] that will install a virtualenv for each version you want to support, and will run your tests with all these versions with one simple command. It seems like a small thing, and it is, but it makes porting just a little bit more pleasant. If you plan to support both Python 2 and Python 3 you should try it out.

3.9 Optional: Use the iterator-methods on dictionaries

Since Python 2.2 the built-in Python dictionary type has had the methods `iterkeys()`, `itervalues()` and `iteritems()`. They yield the same data as `keys()`, `values()` and `items()` do, but instead of returning lists they return iterators, which saves memory and time when using large dictionaries.

```
>>> dict = {'Adam': 'Eve', 'John': 'Yoko', 'Donald': 'Daisy'}
>>> dict.keys()
['Donald', 'John', 'Adam']

>>> dict.iterkeys()
<dictionary-keyiterator object at 0x...>
```

In Python 3 the standard `keys()`, `values()` and `items()` return dictionary views, which are very similar to the iterators of Python 2. As there is no longer any need for the iterator variations of these methods they have been removed.

`2to3` will convert the usage of the iterator methods to the standard methods. By explicitly using the iterator methods you make it clear that you don't need a list, which is helpful for the `2to3` conversion, which otherwise will replace your `dict.values()` call with a `list(dict.values())` just to be safe.

Python 2.7 also has the new view iterators available on dictionaries as `.viewitems()`, `.viewkeys()` and `.viewvalues()`, but since they don't exist in earlier Python versions they are only useful once you can drop support for Python 2.6 and earlier.

Also note that if your code is relying on lists being returned, then you are probably misusing the dictionary somehow. For example, in the code below, you can't actually rely on the order of the keys being the same every time, with the result that you can't predict exactly how the code will behave. This can lead to some troublesome debugging.

```
>>> dict = {'Adam': 'Eve', 'John': 'Yoko', 'Donald': 'Daisy'}
>>> dict.keys()[0]
'Donald'
```

Remember, if all you want to do is loop over the dictionary, use `for x in dict` and you will use iterators automatically in both Python 2 and Python 3.

[3] https://pypi.python.org/pypi/coverage
[4] http://testrun.org/tox/latest/

```
>>> dict = {'Adam': 'Eve', 'John': 'Yoko', 'Donald': 'Daisy'}
>>> for x in dict:
...     print '%s + %s == True' % (x, dict[x])
Donald + Daisy == True
John + Yoko == True
Adam + Eve == True
```

CHAPTER

FOUR

PORTING WITH 2TO3

Although it's perfectly possible to just run your Python 2 code under Python 3 and fix each problem as it turns up, this quickly becomes very tedious. You need to change every `print` statement to a `print()` function, and you need to change every `except Exception, e` to use the new `except Exception as e` syntax. These changes are tricky to do in a search and replace and stops being exciting very quickly. Luckily we can use `2to3` to do most of these boring changes automatically.

4.1 Using 2to3

`2to3` is made up of several parts. The core is a `lib2to3`, a package that contains support for refactoring Python code. It can analyze the code and build up a parse tree describing the structure of the code. You can then modify this parse tree and from it generate new code. Also in `lib2to3` is a framework for "fixers", which are modules that make specific refactorings, such as changing the `print` statement into a `print()` function. The set of standard fixers that enables `2to3` to convert Python 2 code to Python 3 code are located in `lib2to3.fixes`. Lastly is the `2to3` script itself, which you run to do the conversion.

Running `2to3` is very simple. You give a file or directory as parameter and it will convert the file or look through the directory for Python files and convert them. `2to3` will print out all the changes to the output, but unless you include the `-w` flag it will not write the changes to the files. It writes the new file to the same file name as the original one and saves the old one with a `.bak` extension, so if you want to keep the original file name without changes you must first copy the files and then convert the copy.

If you have doctests in your Python files you also need to run `2to3` a second time with `-d` to convert the doctests and if you have text files that are doctests you need to run `2to3` on those explicitly. A complete conversion can therefore look something like this:

```
$ 2to3 -w .
$ 2to3 -w -d .
$ 2to3 -w -d src/mypackage/README.txt
$ 2to3 -w -d src/mypackage/tests/*.txt
```

Under Linux and OS X the 2to3 script is installed in the same folder as the Python executable. Under Windows it in installed as `2to3.py` in the `Tools\Scripts` folder in your Python installation, and you have to give the full path:

```
C:\projects\mypackage> C:\Python3.3\Tools\Scripts\2to3.py -w .
```

If you run `2to3` often you might want to add the Scripts directory to your system path.

4.1.1 Explicit fixers

By default, the conversion uses all fixers in the `lib2to3.fixers` package except `buffer`, `idioms`, `set_literal` and `ws_comma`. These will only be used if specifically added to the command line. In this case you also need to specify the default set of fixers, called `all`:

```
$ 2to3 -f all -f buffer .
```

The buffer fixer will replace all use of the `buffer` type with a `memoryview` type. The `buffer` type is gone in Python 3 and the `memoryview` type is not completely compatible. So the `buffer` fixer is not included by default as you might have to make manual changes as well.

The other three fixers will make more stylistic changes and are as such not really necessary.

The `idioms` fixer will update some outdated idioms. It will change `type(x) == SomeType` and other type-tests to using `isinstance()`, it will change the old style `while 1:` used in Python 1 into `while True:` and it will change some usage of `.sort()` into `sorted()` (See *Use sorted() instead of .sort()* (page 49).)

The `set_literal` fixer will change calls to the `set()` constructor to use the new set literal. See *Set literals* (page 52).

The `ws_comma` fixer will fix up the whitespace around commas and colons in the code.

It is possible to write your own fixers, although it is highly unlikely that you would need to. For more information on that, see *Extending 2to3 with your own fixers* (page 73).

You can also exclude some fixers while still running the default `all` set of fixers:

```
$ 2to3 -x print .
```

If you don't intend to continue to support Python 2, that's all you need to know about `2to3`. You only need to run it once and then comes the fun part, fixing the migration problems, which is discussed in *Common migration problems* (page 35).

4.2 Distributing packages

When you write Python modules or packages that are used by other developers you probably want to continue to support Python 2. Then you need to make sure that Python 3 users get the Python 3 version and Python 2 users get the Python 2 version of your package. This can be as simple as documenting on the download page, if you host your packages yourself.

Most packages for general use use `distutils` and are uploaded to the CheeseShop[1], from where they are often installed with tools like `easy_install` or `pip`. These tools will download the latest version of the package and install it, and if you have both Python 2 and Python 3 packages uploaded to CheeseShop, many of the users will then get the wrong version and will be unable to install your package.

[1] http://pypi.python.org/

One common solution for this is to have two separate package names, like `mymodule` and `mymodule3`, but then you have two packages to maintain and two releases to make. A better solution is to include both source trees in one distribution archive, for example under `src2` for Python 2 and `src3` under Python 3. You can then in your `setup.py` select which source tree should be installed depending on the Python version:

```
import sys
from distutils.core import setup

if sys.version < '3':
      package_dir = {'': 'src2'}
else:
      package_dir = {'': 'src3'}

setup(name='foo',
      version='1.0',
      package_dir = package_dir,
      )
```

This way all users of your module package will download the same distribution and the install script will install the correct version of the code. Your `setup.py` needs to run under both Python 2 and Python 3 for this to work, which is usually not a problem. See *Supporting Python 2 and 3 without 2to3 conversion* (page 57) for more help on how to do that.

If you have a very complex `setup.py` you might want to have one for each version of Python, one called `setup2.py` for Python 2 and one called `setup3.py` for Python 3. You can then make a `setup.py` that selects the correct setup-file depending on Python version:

```
import sys
if sys.version < '3':
    import setup2
else:
    import setup3
```

4.2.1 Running 2to3 on install

The official way to support both Python 2 and Python 3 is to maintain the code in a version for Python 2 and convert it to Python 3 with the `2to3` tool. If you are doing this you can simplify your distribution by running the conversion during install. That way you don't have to have separate packages or even two copies of the code in your package distribution.

Distutils supports this with a custom build command. Replace the `build_py` command class with `build_py_2to3` under Python 3:

```
try:
   from distutils.command.build_py import build_py_2to3 \
        as build_py
except ImportError:
   from distutils.command.build_py import build_py

setup(
   ...
```

```
    cmdclass = {'build_py': build_py}
    )
```

However, if you want to use this solution, you probably want to switch from Distutils to Distribute, that extends this concept further and integrates `2to3` tighter into the development process.

4.3 Supporting multiple versions of Python with Distribute

If you are using `2to3` to support both Python 2 and Python 3 you will find Distribute[2] very helpful. It is a Distutils extension that is a Python 3 compatible fork of Phillip J. Eby's popular Setuptools package. Distribute adds the same `2to3` integration to the build command as Distutils does, so it will solve the distribution problems for you, but it also will help you during the development and testing of your package.

When you use `2to3` to support both Python 2 and Python 3 you need to run `2to3` every time you have made a change, before running the tests under Python 3. Distribute integrates this process into its `test` command, which means that any files you have updated will be copied to a build directory and converted with `2to3` before the tests are run in that build directory, all by just one command. After you have made a change to your code, you just run `python setup.py test` for each version of Python you need to support to make sure that the tests run. This makes for a comfortable environment to do Python 3 porting while continuing to support Python 2.

To install Distribute you need run the Distribute setup script from http://python-distribute.org/distribute_setup.py. You then run `distribute_setup.py` with all Python versions where you want Distribute installed. Distribute is mainly compatible with Setuptools, so you can use Setuptools under Python 2 instead of Distribute but it's probably better to be consistent and use Distribute under Python 2 as well.

If you are using Distutils or Setuptools to install your software you already have a `setup.py`. To switch from Setuptools to Distribute you don't have to do anything. To switch from Distutils to Distribute you need to change where you import the `setup()` function from. In Distutils you import from `distutils.core` while in Setuptools and Distribute you import from `setuptools`.

If you don't have a `setup.py` you will have to create one. A typical example of a `setup.py` would look something like this:

```
from setuptools import setup, find_packages

readme = open('docs/README.txt', 'rt').read()
changes = open('docs/CHANGES.txt', 'rt').read()

setup(name='Porting to Python 3 examples',
      version="1.0",
      description="An example project for Porting to Python 3",
      long_description=readme + '\n' + changes,
      classifiers=[
```

[2] http://pypi.python.org/pypi/distribute

```
          "Programming Language :: Python :: 2",
          "Topic :: Software Development :: Documentation"],
      keywords='python3 porting documentation examples',
      author='Lennart Regebro',
      author_email='regebro@gmail.com',
      license='GPL',
      packages=find_packages(exclude=['ez_setup']),
      include_package_data=True)
```

Explaining all the intricacies and possibilities in Distribute is outside the scope of this book. The full documentation for Distribute is on http://packages.python.org/distribute.

4.3.1 Running tests with Distribute

Once you have Distribute set up to package your module you need to use Distribute to run your tests. You can tell Distribute where to look for tests to run by adding the parameter `test_suite` to the `setup()` call. It can either specify a module to run, a test class to run, or a function that returns a `TestSuite` object to run. Often you can set it to the same as the base package name. That will make Distribute look in the package for tests to run. If you also have separate files that should be run as `DocTests` then Distribute will not find them automatically. In those cases it's easiest to make a function that returns a `TestSuite` with all the tests.

```
import unittest, doctest, StringIO

class TestCase1(unittest.TestCase):

    def test_2to3(self):
        assert True

def test_suite():
    suite = unittest.makeSuite(TestCase1)
    return suite
```

We then specify that function in the `setup()`:

```
from setuptools import setup, find_packages

readme = open('docs/README.txt', 'rt').read()
changes = open('docs/CHANGES.txt', 'rt').read()

setup(name='Porting to Python 3 examples',
      version="1.0",
      description="An example project for Porting to Python 3",
      long_description=readme + '\n' + changes,
      classifiers=[
          "Programming Language :: Python :: 2",
          "Topic :: Software Development :: Documentation"],
      keywords='python3 porting documentation examples',
      author='Lennart Regebro',
      author_email='regebro@gmail.com',
      license='GPL',
```

```
      packages=find_packages(exclude=['ez_setup']),
      include_package_data=True,
      test_suite='py3example.tests.test_suite')
```

You can now run your tests with `python setup.py test`.

4.3.2 Running 2to3 with Distribute

Once you have the tests running under Python 2, you can add the `use_2to3` keyword options to `setup()` and start running the tests with Python 3. Also add `"Programming Language :: Python :: 3"` to the list of classifiers. This tells the CheeseShop and your users that you support Python 3.

```
from setuptools import setup, find_packages

readme = open('docs/README.txt', 'rt').read()
changes = open('docs/CHANGES.txt', 'rt').read()

setup(name='Porting to Python 3 examples',
      version="1.0",
      description="An example project for Porting to Python 3",
      long_description=readme + '\n' + changes,
      classifiers=[
          "Programming Language :: Python :: 2",
          "Programming Language :: Python :: 3",
          "Topic :: Software Development :: Documentation"],
      keywords='python3 porting documentation examples',
      author='Lennart Regebro',
      author_email='regebro@gmail.com',
      license='GPL',
      packages=find_packages(exclude=['ez_setup']),
      include_package_data=True,
      test_suite='py3example.tests.test_suite',
      use_2to3=True)
```

Under Python 3, the `test` command will now first copy the files to a build directory and run `2to3` on them. It will then run the tests from the build directory. Under Python 2, the `use_2to3` option will be ignored.

Distribute will convert all Python files and also all doctests in Python files. However, if you have doctests located in separate text files, these will not automatically be converted. By adding them to the `convert_2to3_doctests` option Distribute will convert them as well.

To use additional fixers, the parameter `use_2to3_fixers` can be set to a list of names of packages containing fixers. This can be used both for the explicit fixers included in `2to3` and external fixers, such as the fixers needed if you use the Zope Component Architecture.

```
from setuptools import setup, find_packages

readme = open('docs/README.txt', 'rt').read()
changes = open('docs/CHANGES.txt', 'rt').read()

setup(name='Porting to Python 3 examples',
```

```
version="1.0",
description="An example project for Porting to Python 3",
long_description=readme + '\n' + changes,
classifiers=[
    "Programming Language :: Python :: 2",
    "Programming Language :: Python :: 3",
    "Topic :: Software Development :: Documentation"],
keywords='python3 porting documentation examples',
author='Lennart Regebro',
author_email='regebro@gmail.com',
license='GPL',
packages=find_packages(exclude=['ez_setup']),
include_package_data=True,
test_suite='py3example.tests.test_suite',
use_2to3=True,
convert_2to3_doctests=['doc/README.txt'],
install_requires=['zope.fixers'],
use_2to3_fixers=['zope.fixers'])
```

Attention: When you make changes to `setup.py`, this may change which files get converted. The conversion process will not know which files was converted during the last run, so it doesn't know that a file which during the last run was just copied now should be copied and converted. Therefore you often have to delete the whole build directory after making changes to `setup.py`.

You should now be ready to try to run the tests under Python 3 by running `python3 setup.py test`. Most likely some of the tests will fail, but as long as the `2to3` process works and the tests run, even if they fail, then you have come a long way towards porting to Python 3. It's now time to look into fixing those failing tests. Which leads us into discussing the common migration problems.

CHAPTER

FIVE

COMMON MIGRATION PROBLEMS

If you have followed the recommendation to make sure your code runs without warnings with `python2.7 -3` many of the simpler errors you can get now will be avoided, such as having variable names that are Python 3 keywords and other easy fixes. You will also have avoided one of the more subtle errors, that integer division now can return a float. There is still a range of common errors you may run into. Some are easy to fix, other less so.

If you need to support both Python 2 and Python 3 it's quite likely that you will need to make conditional code depending on the Python version. In this book I consistently use my favourite way of testing, I compare the `sys.version` string with `'3'`. But there are many other ways of doing the same test, like `sys.version[0] != '3'` or using the version_info tuple with `sys.version_info < ('3',)` etc. Which one you use is a matter of personal preference. If you end up doing a lot of tests, setting a constant is a good idea:

```
>>> import sys
>>> PY3 = sys.version > '3'
```

Then you can just use the PY3 constant in the rest of the software.

Now, onto the most common problems.

5.1 Incorrect imports

Sometimes you'll encounter a situation where `2to3` seems to have missed changing an import. This is usually because you can import a function or class from another module than where it's defined.

For example, in Python 2 `url2pathname` is defined in `urllib`, but it is used by and imported into `urllib2`. It's common to see code that imports `url2pathname` from `urllib2` to avoid needing to import `urllib` separately. However, when you do this, the import will not be correctly changed to the new library locations, as `2to3` doesn't know about this trick, so you need to change your code to import from the correct place before running `2to3`.

5.2 Relative import problems

Python 3 changes the syntax for imports from within a package, requiring you to use the relative import syntax, saying `from . import mymodule` instead of the just `import mymodule`.

For the most part `2to3` will handle this for you, but there are a couple of cases where `2to3` will do the wrong thing.

The `import` fixer will look at your imports and look at your local modules and packages, and if the import is local, it will change it to the new syntax. However, when the local module is an extension module, that module will typically not have been built when you run `2to3`. That means the fixer will not find the local module, and not change the import.

Contrariwise, if you import a module from the standard library and you also have a folder with the same name as that library, the `import` fixer will assume that it is a local package, and change the import to a local import. It will do this even if the folder is not a package. This is a bug in the `import` fixer, but all current versions of Python has this bug, so you need to be aware of it.

The solution to these problems for Python 2.5 and later is to convert the imports to relative imports and add the `__future__` import to enable Python 3's absolute/relative import syntax.

```
from __future__ import absolute_import
from . import mymodule
```

Since the module already uses the new import behavior the `import` fixer will not make any changes to the relative imports, avoiding these problems.

If you need to support Python 2.4 or earlier you can avoid these issue by not having any relative imports at all, and excluding the `import` fixer when running `2to3`.

5.3 Unorderable types, `__cmp__` and `cmp`

Under Python 2 the most common way of making types sortable is to implement a `__cmp__()` method that in turn uses the builtin `cmp()` function, like this class that will sort according to `lastname`:

```
>>> class Orderable(object):
...
...     def __init__(self, firstname, lastname):
...         self.first = firstname
...         self.last = lastname
...
...     def __cmp__(self, other):
...         return cmp("%s, %s" % (self.last, self.first),
...                    "%s, %s" % (other.last, other.first))
...
...     def __repr__(self):
...         return "%s %s" % (self.first, self.last)
...
>>> sorted([Orderable('Donald', 'Duck'),
...         Orderable('Paul', 'Anka')])
[Paul Anka, Donald Duck]
```

Since having both `__cmp__()` and rich comparison methods violates the principle of there being only one obvious way of doing something, Python 3 ignores the `__cmp__()` method. In addition to this, the `cmp()` function is gone! This typically results in your converted code

raising a `TypeError: unorderable types` error. So you need to replace the `__cmp__()` method with rich comparison methods instead. To support sorting you only need to implement `__lt__()`, the method used for the "less then" operator, <.

```
>>> class Orderable(object):
...
...     def __init__(self, firstname, lastname):
...         self.first = firstname
...         self.last = lastname
...
...     def __lt__(self, other):
...         return ("%s, %s" % (self.last, self.first) <
...                 "%s, %s" % (other.last, other.first))
...
...     def __repr__(self):
...         return "%s %s" % (self.first, self.last)
...
>>> sorted([Orderable('Donald', 'Duck'),
...         Orderable('Paul', 'Anka')])
[Paul Anka, Donald Duck]
```

To support other comparison operators you need to implement them separately. See *Separate binary data and strings* (page 17) for an example of how to do that.

5.4 Sorting

In parallel to the removal of the `cmp()` function and `__cmp__()` method the `cmp` parameter to `list.sort()` and `sorted()` is gone in Python 3. This results in one of these following errors:

```
TypeError: 'cmp' is an invalid keyword argument for this function
TypeError: must use keyword argument for key function
```

Instead of the `cmp` parameter you need to use the `key` parameter that was introduced in Python 2.4. See *When sorting, use key instead of cmp* (page 18) for more information.

5.5 Sorting Unicode

Because the `cmp=` parameter is removed in Python 3 sorting Unicode with `locale.strcoll` no longer works. In Python 3 you can use `locale.strxfrm` instead.

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'sv_SE.UTF-8')
'sv_SE.UTF-8'
>>> corpus = ["art", "Älg", "ærgeligt", "Aardvark"]
>>> sorted(corpus, key=locale.strxfrm)
['Aardvark', 'art', 'Älg', 'ærgeligt']
```

This will not work under Python 2, where `locale.strxfrm` will expect a non-unicode string encoded with the locale encoding. If you only support Python 2.7 and Python 3.2 and later, you can still use `locale.strcoll` thanks to a convenience function in `functools`.

```
>>> from functools import cmp_to_key
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'sv_SE.UTF-8')
'sv_SE.UTF-8'
>>> corpus = ["art", "Älg", "ærgeligt", "Aardvark"]
>>> sorted(corpus, key=cmp_to_key(locale.strcoll))
['Aardvark', 'art', 'Älg', 'ærgeligt']
```

This is however much slower, and doesn't work in Python 2.6 or Python 3.1.

5.6 Bytes, strings and Unicode

The biggest problem you may encounter relates to one of the most important changes in Python 3; strings are now always Unicode. This will simplify any application that needs to use Unicode, which is almost any application that is to be used outside of English-speaking countries.

Of course, since strings are now always Unicode, we need another type for binary data. Python 3 has two new binary types, `bytes` and `bytearrays`. The `bytes` type is similar to the the string type, but instead of being a string of characters, it's a string of integers. `Bytearrays` are more like a list, but a list that can only hold integers between 0 and 255. A `bytearray` is mutable and used if you need to manipulate binary data. Because it's a new type, although it also exists in Python 2.6, I'm mostly going to ignore it in this book and concentrate on other ways of handling binary data.

5.6.1 Byte literals

The first problem we encounter is how to put binary data into the Python code. In Python 2 we used standard strings and therefore standard string literals. To check if a file really is a GIF file, we can look at the first six bytes, which should start with `GIF89a` (or `GIF87a`, but let's ignore that for now):

```
>>> file = open('maybe_a.gif', 'rb')
>>> file.read(6) == 'GIF89a'
True
```

In Python 3 the test would always fail, as you need to compare with a bytes object instead. If you don't need Python 2 support you can simply change any string literals that hold binary data to be bytes literals by adding a leading `b` to them.

```
>>> file = open('maybe_a.gif', 'rb')
>>> file.read(6) == b'GIF89a'
True
```

There are also a couple of other cases that need changing. The string type in Python 2 is a list of 8-bit characters, but the bytes type in Python 3 is a list of 8-bit integers. So taking one character of a string will return a one-character long string, but taking one byte of a bytes object will return an integer! You will therefore have to change any byte-level manipulation to use integers instead. Often this is a question of removing a lot of `ord()` and `chr()` calls, as manipulation on byte levels tend to be about integers and not characters in the first place.

```
>>> 'GIF89a'[2]
'F'
>>> b'GIF89a'[2]
70
```

5.6.2 Binary data in Python 2 and Python 3

These changes create a problem if you want to still support Python 2. `2to3` will generally assume that when you use a string in Python 2, that's what you want in Python 3 as well and in most cases this is true. So when it is not true, you need to mark the data as binary so it keeps working in Python 3.

In Python 2.6 and the upcoming 2.7 there is both a bytes literal you can use to specify that the data is binary, as well as a bytes type. Under Python 2 the bytes literal and bytes type are just aliases for `str` so the objects will not behave exactly the same as the bytes object in Python 3. Most importantly, it will be a string of characters, not a string of bytes, so in Python 2, `b'GIF89a'[2]` will not return the integer `70`, but the string `'F'`.

Luckily this is not a very common problem, as most cases of handling binary data handle the data as one block and you don't need to look at or modify separate bytes. If you need to do it there is a trick that works under both Python 2 and Python 3 and that is to make a one character long slice.

```
>>> b'GIF89a'[2:3]
b'F'
```

This will work equally well under Python 2.6 as Python 3, although you get a one character `str`-string in Python 2.6 and a one character `bytes`-string in Python 3.

However, under Python 2.5 or earlier, the `b'GIF89a'` syntax doesn't work at all, so it's only a solution if you don't need to support versions of Python before 2.6. To ensure that the data is binary you can make a Unicode string and encode it to get binary data. This code will work well in all versions of Python 2 and Python 3.

```
>>> file = open('maybe_a.gif', 'rb')
>>> file.read(6) == u'GIF89a'.encode('ISO-8859-1')
True
```

Of course, the `u'GIF89a'` isn't valid syntax in Python 3, because the separate Unicode literal is gone, but `2to3` will handle it and remove the `u` prefix.

5.6.3 Nicer solutions

If you think this is all a bunch of ugly hacks, you are correct. Let's improve things by making a special function which under Python 3 will take a string and make binary data from it and in Python 2 will simply return the string as is. This is both less ugly and gets rid of the encoding step under Python 2. You can call this function anything you want, including "`ArthurBelling`", but several early adopters of Python 3 has made their own variants of this function and they all called it "`b`" which is nice, short and looks similar to the bytes literal. We'll define it like this:

```
import sys
if sys.version < '3':
    def b(x):
        return x
else:
    import codecs
    def b(x):
        return codecs.latin_1_encode(x)[0]
```

Under Python 2 this will return a the string you pass in, ready for use as binary data:

```
>>> from makebytes import b
>>> b('GIF89a')
'GIF89a'
```

While under Python 3 it will take a string and encode it to return a bytes object.

```
>>> from makebytes import b
>>> b('GIF89a')
b'GIF89a'
```

This method uses the ISO-8859-1 encoding, also known as Latin-1, as it is the only encoding whose 256 characters are identical to the 256 first characters of Unicode. The example here would work fine with the ASCII encoding, but if you have a character value over 127 then you need to use the ISO-8859-1 and its one-to-one mapping to Unicode.

This implementation of the `b()` function picks up the encoding function directly from the `codecs` module as this is marginally faster, but it will not be noticeable in practice. You'll need to call `b()` millions of times for it to make a difference, and since you use it as a replacement for bytes literals this will not happen. Feel free to use `x.encode('ISO-8859-1')` instead if you like it better.

5.6.4 Manipulating binary data

The `b()` function makes it possible to create binary data from literals, so it solves one major problem with binary data when supporting both Python 2 and Python 3. It doesn't solve the unusual case where you need to inspect or modify the bytes one by one, as indexing or iterating over the binary data will return one-character strings under Python 2 but integers under Python 3.

If you only need to support Python 2.6 and Python 2.7 you can use the new `bytearray` type for this. It is a mutable type that has an interface that supports a useful mix of the most common list and string operations, it has for example both `.append()` and `.find()` as well as some methods of its own, like the handy `.fromhex()`.

```
>>> data = bytearray(b'Monty Python')
>>> data[5] = 33
>>> data
bytearray(b'Monty!Python')
```

If you need to support Python 2.5 or earlier you can solve this issue by introducing helper functions to iterate or get a specific index out of either a `str` or a `bytes` and return integers in both cases:

```
>>> import sys
>>> if sys.version < '3':
...     def byteindex(data, index):
...         return ord(data[index])
...
...     def iterbytes(data):
...         return (ord (char) for char in data)
...
... else:
...     byteindex = lambda x, i: x[i]
...     iterbytes = lambda x: iter(x)
...
>>> from makebytes import b
>>> byteindex(b('Test'), 2)
115
>>> print([x for x in iterbytes(b('Test'))])
[84, 101, 115, 116]
```

The above `iterbytes` example uses a generator expression, which requires Python 2.4. To support earlier Pythons you can make it a list comprehension instead, which just uses more memory.

If you don't like these helper functions you might want to introduce a special binary type that works the same under both Python 2 and Python 3. However, the standard library will assume you pass in strings under Python 2 and under Python 3 it will assume you pass in bytes, so you have to subclass it from `str` in Python 2 and `bytes` in Python 3. This solution introduces a new type that has extra functions that behave the same under all versions of Python, but leave the standard functions alone:

```
import sys
if sys.version < '3':
    class Bites(str):
        def __new__(cls, value):
            if isinstance(value[0], int):
                # It's a list of integers
                value = ''.join([chr(x) for x in value])
            return super(Bites, cls).__new__(cls, value)

        def itemint(self, index):
            return ord(self[index])

        def iterint(self):
            for x in self:
                yield ord(x)
else:

    class Bites(bytes):
        def __new__(cls, value):
            if isinstance(value, str):
                # It's a unicode string:
                value = value.encode('ISO-8859-1')
            return super(Bites, cls).__new__(cls, value)
```

```
    def itemint(self, x):
        return self[x]

    def iterint(self):
        for x in self:
            yield x
```

This new binary class, which I called `Bites` but equally well could be called anything else, including just `b`, in analogy with the method above, will accept both strings and lists of integers in the construction. As you see it subclasses `str` in Python 2 and `bytes` in Python 3 and is therefore an extension of the binary types in both versions, so it can be passed straight into standard library functions that need binary data.

You would use it like this:

```
>>> from bites import Bites
>>> Bites([71, 73, 70, 56, 57, 97]).itemint(2)
70
>>> binary_data = Bites(open('maybe_a.gif', 'rb').read())
>>> binary_data.itemint(2)
70
>>> print([x for x in Bites('GIF89a').iterint()])
[71, 73, 70, 56, 57, 97]
```

You could easily add support for taking a slice out of the class and always get a list of integers as well, or any other method you need to work the same in Python 2 and Python 3.

If you think a whole class like this seems overly complicated, then you are right. It is quite likely that you won't ever need it, even when handling binary data. Most of the time you need to handle binary data you do it by reading or writing binary data from a stream or calling functions, and in these cases you handle the data as one block and do not look at individual bytes. So for almost all handling of binary data the `Bites` class is overkill.

5.6.5 Reading from files

Another source of problems in handling binary and Unicode data is when you read and write to files or other streams. One common problem is that the file is opened in the wrong mode. Make sure you open text files with the `'t'` flag and binary files with the `'b'` flag and you have solved many problems.

Opening a file with the `'t'` flag will return a `unicode` object under Python 3 and it will be decoded from the system default encoding, which is different on different platforms. If it has any other encoding you need to pass that encoding into the `open()` function as a parameter. In Python 2 the `open()` function doesn't take an encoding parameter and will return a `str` object. As long as your file contains only ASCII characters this isn't a problem, but when it does you will have to make some changes.

Opening the file as binary and decoding the data afterward is an option, for example with `codecs.open()`. However, the translation of line endings that happens on Windows isn't going to work in that case:

```
>>> import codecs
>>> infile = codecs.open('UTF-8.txt', 'r', encoding='UTF-8')
>>> print(infile.read())
It wörks
```

Python 3's handling and open method is contained in the new `io` module. This module has been backported to Python 2.6 and 2.7, so if you don't need to support Python 2.5 or earlier, you can replace all `open()` calls with `io.open()` for Python 3 compatibility. It doesn't suffer from the line ending problem under Windows that `codec.open()` has and will in fact convert line endings in text mode under all platforms, unless explicitly told not to with the `newline=''` parameter.

```
>>> import io
>>> infile = io.open('UTF-8.txt', 'rt', encoding='UTF-8')
>>> print(infile.read())
It wörks
```

But beware that the `io` module under Python 2.6 and Python 3.0 is quite slow, so if you are handling loads of data and need to support Python 2.6 this may not be the right solution for you.

Another case where you need to open the file in binary mode is if you don't know if the file is text or binary until you have read part of it, or if the file contains both binary and text. Then you need to open it in binary mode and decode it conditionally, where in Python 2 you could often have opened it in binary mode, and skipped the decoding.

Also, if you do a lot of seeking in a file it will be very slow if you open the file in text mode, as the seeking will need to decode the data. In that case you need to open the file in binary mode, and decode the data after reading it.

5.7 Replacing UserDict

When you want to make classes that behave like dictionaries but aren't, the `UserDict` module is a popular solution because you don't have to implement all the dictionary methods yourself. However, the `UserDict` module is gone in Python 3, merged into the `collections` module. Because of the change in how dictionaries work in Python 3, where `items()`, `keys()` and `values()` now return views instead of lists as well as the changes in how sorting and comparing is done, the replacements are not completely compatible. Because of this there are no fixers that make these changes, you will have to do them manually.

In most cases it is just a question of replacing the base class. `UserDict.IterableUserDict` is replaced by `collections.UserDict` and `UserDict.DictMixin` is now `collections.MutableMapping`, `UserDict.UserDict` is gone, but `collections.UserDict` will work as a solution in most cases.

One of the common problems is that `collections.MutableMapping` requires your dictionary to implement `__len__` and `__iter__` where `DictMixin` doesn't. However, implementing them so that they work under Python 3 won't break anything under Python 2.

If you need to support both Python 3 and versions below Python 2.6 you also have to make conditional imports:

```
>>> try:
...     from UserDict import UserDict
...     from UserDict import DictMixin
... except ImportError:
...     from collections import UserDict
...     from collections import MutableMapping as DictMixin
```

5.8 CSV API changes

In Python 2, the `csv` module requires you to open files in binary mode. This is because the module needs to be able to control line-endings, as typical CSV files use DOS line-endings, and the text-mode under Python 2 can change line-endings on some platforms. The `csv` module will also return and expect data in byte-strings.

The Python 3 `csv`-module instead requires you to open the file in text-mode with `newline=''`, and it returns and expects Unicode strings.

If you need to support both Python 2 and Python 3, and you need to support Unicode, the best solution I have found is to use "wrapper" classes. The following classes work for Python 2.6 and later.

```
import sys, csv, codecs

PY3 = sys.version > '3'

class UnicodeReader:
    def __init__(self, filename, dialect=csv.excel,
                 encoding="utf-8", **kw):
        self.filename = filename
        self.dialect = dialect
        self.encoding = encoding
        self.kw = kw

    def __enter__(self):
        if PY3:
            self.f = open(self.filename, 'rt',
                          encoding=self.encoding, newline='')
        else:
            self.f = open(self.filename, 'rb')
        self.reader = csv.reader(self.f, dialect=self.dialect,
                                 **self.kw)
        return self

    def __exit__(self, type, value, traceback):
        self.f.close()

    def next(self):
        row = next(self.reader)
        if PY3:
            return row
        return [s.decode("utf-8") for s in row]
```

```
    __next__ = next

    def __iter__(self):
        return self

class UnicodeWriter:
    def __init__(self, filename, dialect=csv.excel,
                 encoding="utf-8", **kw):
        self.filename = filename
        self.dialect = dialect
        self.encoding = encoding
        self.kw = kw

    def __enter__(self):
        if PY3:
            self.f = open(self.filename, 'wt',
                          encoding=self.encoding, newline='')
        else:
            self.f = open(self.filename, 'wb')
        self.writer = csv.writer(self.f, dialect=self.dialect,
                                 **self.kw)
        return self

    def __exit__(self, type, value, traceback):
        self.f.close()

    def writerow(self, row):
        if not PY3:
            row = [s.encode(self.encoding) for s in row]
        self.writer.writerow(row)

    def writerows(self, rows):
        for row in rows:
            self.writerow(row)
```

The `DictReader` and `DictWriter` can easily be extended in the same way, by encoding and decoding both keys and values.

5.9 Running doctests

One of the more persistently annoying problems you may encounter are doctests. Personally I think doctests are brilliant for testing documentation, but there has been a recommendation in some circuits to make as many tests as possible doctests. This becomes a problem with Python 3 because doctests rely on comparing the output of the code. That means they are sensitive to changes in formatting and Python 3 has several of these. This means that if you have doctests you will get many, many failures. Don't despair! Most of them are not actual failures, but changes in the output formatting. `2to3` handles that change in the code of the doctests, but not in the output.

If you are only porting to Python 3, the solution is simple and boring. Run the doctests and look at each failure to see if it is a real failure or a change in formatting. This can sometimes be frustrating, as you can sit and stare at a failure trying to figure out what

actually is different between the expected and the actual output. On the other hand, that's normal with doctests, even when you aren't porting to Python 3, which of course is one of the reasons that they aren't suitable as the main form of testing for a project.

It gets more tricky if you need to continue to support Python 2, since you need to write output that works in both versions and that can be difficult and in some cases impossible for example when testing for exceptions, see below.

5.9.1 `write()` has a return value

One common reason a doctest fails under Python 3 is when writing to a file. The `write()` method now returns the number of bytes written. Doctests for Python 2 will not expect anything to be returned, so they will break. The workaround for that is easy and will work under Python 2 as well. Just assign a dummy variable to the result:

```
>>> ignore = open('/tmp/file.out', 'wt').write('Some text')
```

5.9.2 Types are now classes

Also, the `__repr__()` output of many types have changed. In Python 2 built-in classes represented themselves as types.

```
>>> type([])
<type 'list'>
```

In Python 3, they are classes, like everything else.

```
>>> type([])
<class 'list'>
```

Here you have two options if you want to support both Python 2 and Python 3. The first it to use `isinstance` instead:

```
>>> isinstance([], list)
True
```

The alternative is to enable the `ELLIPSIS` flag for the doctests and replace the part of the output that changes with three dots.

```
>>> type([])
<... 'list'>
```

5.9.3 Handling expected exceptions

Using the `ELLIPSIS` flag is something you can use for most differences you find when you need to support both Python 2 and Python 3, but with one exception, namely exceptions. The output of tracebacks now includes the module names of the exception. In Python 2 checking for an exception would look like this:

```
>>> import socket
>>> socket.gethostbyname("www.python.rog")
Traceback (most recent call last):
gaierror: [Errno -5] No address associated with hostname
```

However, in Python 3 that traceback will include the module name, so you have to make it look like this:

```
>>> import socket
>>> socket.gethostbyname("www.python.rog")
Traceback (most recent call last):
socket.gaierror: [Errno -5] No address associated with hostname
```

In addition to this, some Exceptions have moved as a part of the general reorganization of the standard library. You can't use the `ELLIPSIS` flag and put an ellipsis in the beginning of the exception definition, because if you add that doctests no longer recognize the output as an exception and it will stop working in all versions of Python! The solution to this is to trap the exception:

```
>>> import socket
>>> try:
...     socket.gethostbyname("www.python.rog")
...     raise AssertionError("gaierror exception was not raised")
... except socket.gaierror:
...     pass
```

It's not a pretty solution, but the only one available at the moment. Luckily the most common exceptions, the ones in the builtin module, do not change their rendering, so they will continue to work. You only need to do this change with exceptions from the modules in the standard library or any third-party modules. In Python 2.7 the `IGNORE_EXCEPTION_DETAIL` flag has been extended so that it will handle the differences in exception formatting. However, it will still not work under Python 2.6, so if you need to support Python 2.6 or earlier you need to rewrite the test to trap the exception.

If I have doctests with a lot of exception testing I usually end up using a helper function, similar to the `assertRaises` of standard unit tests:

```
import sys

def shouldRaise(eclass, method, *args, **kw):
     try:
         method(*args, **kw)
     except:
         e = sys.exc_info()[1]
         if not isinstance(e, eclass):
             raise
         return
     raise Exception("Expected exception %s not raised" %
                     str(eclass))
```

The usage would be like this:

```
>>> import socket
>>> shouldRaise(socket.gaierror,
...     socket.gethostbyname, "www.python.rog")
```

5.9.4 String representation

Output from functions that return binary data, like reading from a website, will return `str` under Python 2, while in Python 3 they will return `bytes`, which has a different representation.

To solve this I have used a helper function that makes sure the output is a string before printing it:

```
>>> def bprint(data):
...     if not isinstance(data, str):
...         data = data.decode()
...     print(data.strip())
```

It also removes leading and trailing whitespace for good measure, so that you don't have to have as many `<BLANKLINE>` statements in the code.

5.9.5 `dict` and `set` order

In Python 3.3 a random seed value is added to the hash function, for security reasons. This means that any doctest you have that tests the output of a dictionary or set will fail when you try to run it in Python 3.3, as the order will change with every run.

```
Failed example:
    {x for x in department}
Expected:
    {'a', ' ', 'i', 'k', 'l', 'S', 'W', 'y'}
Got:
    {'y', 'S', 'W', 'i', 'k', 'l', 'a', ' '}
```

This must be changed to equality testing, which unfortunately will make the failures much less informative.

```
Failed example:
    {x for x in department} == \
        {'a', ' ', 'i', 'e', 'l', 'S', 'W', 'y'}
Expected:
    True
Got:
    False
```

CHAPTER

SIX

IMPROVING YOUR CODE WITH MODERN IDIOMS

Once you have ported to Python 3 you have a chance to use the newer features of Python to improve your code. Many of the things mentioned in this chapter are in fact possible to do even before porting, as they are supported even by quite old versions of Python. But I mention them here anyway, because they aren't always used when the code could benefit from them. This includes generators, available since Python 2.2; the `sorted()` method, available since Python 2.4 and context managers, available since Python 2.5.

The rest of the new features mentioned here have in fact been backported to either Python 2.6 or Python 2.7. So if you are able to drop support for Python 2.5 and earlier, you can use almost all of these new features already before porting.

6.1 Use `sorted()` instead of `.sort()`

Lists in Python has a `.sort()` method that will sort the list in place. Quite often when `.sort()` is used is it used on a temporary variable discarded after the loop. Code like this is used because up to Python 2.3 this was the only built-in way of sorting.

```
>>> infile = open('pythons.txt')
>>> pythons = infile.readlines()
>>> pythons.sort()
>>> [x.strip() for x in pythons]
['Eric', 'Graham', 'John', 'Michael', 'Terry', 'Terry']
```

Python 2.4 has a new built-in, `sorted()`, that will return a sorted list and takes the same parameters as `.sort()`. With `sorted()` you often can avoid the temporary variable. It also will accept any iterable as input, not just lists, which can make your code more flexible and readable.

```
>>> infile = open('pythons.txt')
>>> [x.strip() for x in sorted(infile)]
['Eric', 'Graham', 'John', 'Michael', 'Terry', 'Terry']
```

There is however no benefit in replacing a `mylist.sort()` with `mylist = sorted(mylist)`, in fact it will use more memory.

The `2to3` fixer `"idioms"` will change some usage of `.sort()` into `sorted()`.

6.2 Coding `with` context managers

Since Python 2.5 you can create context managers, which allows you to create and manage a runtime context. If you think that sounds rather abstract, you are completely right. Context managers are abstract and flexible beasts that can be used and misused in various ways, but this chapter is about how to improve your existing code so I'll take up some examples of typical usage where they simplify life.

Context managers are used as a part of the `with` statement. The context manager is created and entered in the `with` statement and available during the `with` statements code block. At the end of the code block the context manager is exited. This may not sound very exiting until you realize that you can use it for resource allocation. The resource manager then allocates the resource when you enter the context and deallocates it when you exit.

The most common example of this type of resource allocation are open files. In most low level languages you have to remember to close files that you open, while in Python the file is closed once the file object gets garbage collected. However, that can take a long time and sometimes you may have to make sure you close the file explicitly, for example when you open many files in a loop as you may otherwise run out of file handles. You also have to make sure you close the file even if an exception happens. The result is code like this:

```
>>> f = open('/tmp/afile.txt', 'w')
>>> try:
...     n = f.write('sometext')
... finally:
...     f.close()
```

Since files are context managers, they can be used in a `with`-statement, simplifying the code significantly:

```
>>> with open('/tmp/afile.txt', 'w') as f:
...     n = f.write('sometext')
```

When used as a context manager, the file will close when the code block is finished, even if an exception occurs. As you see the amount of code is much smaller, but more importantly it's much clearer and easier to read and understand.

Another example is if you want to redirect standard out. Again you would normally make a `try/except` block as above. That's OK if you only do it once in your program, but if you do this repeatedly you will make it much cleaner by making a context manager that handles the redirection and also resets it.

```
>>> import sys
>>> from StringIO import StringIO
>>> class redirect_stdout:
...     def __init__(self, target):
...         self.stdout = sys.stdout
...         self.target = target
...
...     def __enter__(self):
...         sys.stdout = self.target
...
...     def __exit__(self, type, value, tb):
```

```
...         sys.stdout = self.stdout
...
>>> out = StringIO()
>>> with redirect_stdout(out):
...     print 'Test'
...
>>> out.getvalue() == 'Test\n'
True
```

The `__enter__()` method is called when the indented block after the `with` statement is reached and the `__exit___()` method is called when the block is exited, including after an error was raised.

Context managers can be used in many other ways and they are generic enough to be abused in various ways as well. Any code you have that uses exception handling to make sure an allocated resource or a global setting is unallocated or unset will be a good candidate for a context manager.

There are various functions to help you out in making context managers in the `contextlib` module. For example, if you have objects that have a `.close()` method but aren't context managers you can use the `closing()` function to automatically close them at the end of the `with`-block.

```
>>> from contextlib import closing
>>> import urllib
>>>
>>> book_url = 'http://python3porting.com/'
>>> with closing(urllib.urlopen(book_url)) as page:
...     print len(page.readlines())
148
```

6.3 Advanced string formatting

In Python 3 and also Python 2.6 a new string formatting support was introduced. It is more flexible and has a clearer syntax than the old string formatting.

```
Old style formatting:
>>> 'I %s Python %i' % ('like', 2)
'I like Python 2'

New style formatting:
>>> 'I {0} Python {1}'.format('□', 3)
'I □ Python 3'
```

It is in fact a mini-language, and you can do some crazy stuff, but when you go overboard you lose the benefit of the more readable syntax:

```
>>> import sys
>>> 'Python {0.version_info[0]:!<9.1%}'.format(sys)
'Python 300.0%!!!'
```

For a full specification of the advanced string formatting syntax see the `Common String Operations` section of the Python documentation[1].

The old string formatting based on % is planned to be eventually removed, but there is no decided timeline for this.

6.4 Class decorators

Decorators have been around since Python 2.4 and have become commonplace thanks to the builtin decorators like `@property` and `@classmethod`. Python 2.6 introduces class decorators, that work similarly.

Class decorators can both be used to wrap classes and modify the class that should be decorated. An example of the later is `functools.total_ordering`, that will let you implements a minimum of rich comparison operators, and then add the missing ones to your class. They can often do the job of metaclasses, and examples of class decorators are decorators that make the class into a singleton class, or the `zope.interface` class decorators that register a class as implementing a particular interface.

If you have code that modify classes, take a look at class decorators, they may help you to make your code more readable.

6.5 Set literals

There is a new literal syntax for sets available in Python 3. Instead of `set([1, 2, 3])` you can now write the cleaner `{1, 2, 3}`. Both syntaxes work in Python 3, but the new one is the recommended and the representation of sets in Python 3 has changed accordingly:

```
>>> set([1,2,3])
{1, 2, 3}
```

The set literal has been back-ported to Python 2.7, but the representation has not.

6.6 `yield` to the generators

Like the floor division operators and the `key`-parameter to `.sort()`, generators have been around for long time, but you still don't see them that much. But they are immensely practical and save you from creating temporary lists and thereby both save memory and simplify the code. As an example we take a simple function with two loops:

```
>>> def allcombinations(starters, endings):
...     result = []
...     for s in starters:
...         for e in endings:
...             result.append(s+e)
...     return result
```

This becomes more elegant by using `yield` and thereby a generator:

[1] http://docs.python.org/library/string.html#format-string-syntax

```
>>> def allcombinations(starters, endings):
...     for s in starters:
...         for e in endings:
...             yield s+e
```

Although this is a rather trivial case, making complicated loops into generators can sometimes make them much simpler, resulting in cleaner and more readable code. They can be a bit tricky to debug though, since they reverse the normal program flow. If you have a chain of generators, you can't step "up" in the call stack to see what the function that created the generator did. "Up" in the call stack is instead the function that will *use the result* of the generator. This feels backwards or upside down until you get used to it, and can be a bit of a brain teaser. If you are used to functional programming you will feel right at home though.

6.7 More comprehensions

Generators have been around since Python 2.2, but a new way to make generators appeared in Python 2.4, namely generator expressions. These are like list comprehesions, but instead of returning a list, they return a generator. They can be used in many places where list comprehensions are normally used:

```
>>> sum([x*x for x in xrange(2000000)])
2666664666667000000L
```

Becomes:

```
>>> sum(x*x for x in xrange(2000000))
2666664666667000000L
```

Thereby saving you from creating a list with 2 million items and then immediately throwing it away. You can use a generator expression anywhere you can have any expression, but quite often you need to put parentheses around it:

```
>>> (x for x in 'Silly Walk')
<generator object <genexpr> at ...>
```

In Python 3 the generator expression is not just a new nice feature, but a fundamental change as the generator expression is now the base around which all the comprehensions are built. In Python 3 a list comprehension is only syntactic sugar for giving a generator expression to the `list` types constructor:

```
>>> list(x for x in 'Silly Walk')
['S', 'i', 'l', 'l', 'y', ' ', 'W', 'a', 'l', 'k']

>>> [x for x in 'Silly Walk']
['S', 'i', 'l', 'l', 'y', ' ', 'W', 'a', 'l', 'k']
```

This also means that the loop variable no longer leaks into the surrounding namespace.

The new generator expressions can be given to the `dict()` and `set()` constructors in Python 2.6 and later, but in Python 3 and also in Python 2.7 you have new syntax for dictionary and set comprehensions:

```
>>> department = 'Silly Walk'
>>> {x: department.count(x) for x in department}
{'a': 1, ' ': 1, 'i': 1, 'k': 1, 'l': 3, 'S': 1, 'W': 1, 'y': 1}

>>> {x for x in department}
{'a', ' ', 'i', 'k', 'l', 'S', 'W', 'y'}
```

6.8 The next `next()`

In Python 2 iterators have a `.next()` method you use to get the next value from the iterator.

```
>>> i = iter(range(5))
>>> i.next()
0
>>> i.next()
1
```

This special method has in Python 3 been renamed to `.__next__()` to be consistent with the naming of special attributes elswhere in Python. However, you should generally not call it directly, but instead use the builtin `next()` function. This function is also available from Python 2.6, so unless you are supporting Python 2.5 or earlier you can switch.

```
>>> i = iter(range(5))
>>> next(i)
0
>>> next(i)
1
```

6.9 New modules

There is several new modules that you should also take a look at to see if they can be of use for you. I won't take them up in detail here, since most of them are hard to benefit from without refactoring your software completely, but you should know they exist. For more information on them, you can look at the Python documentation.

6.9.1 Abstract base classes

The `abc` module contains support for making abstract base classes, where you can mark a method or property on a base class as "abstract", which means you must implement it in a subclass. Classes that do not implement all abstract methods or properties can not be instantiated.

The abstract base classes can also be used to define interfaces by creating classes that have no concrete methods. The class would then work only as an interface, and subclassing from it guarantees that it implements the interface. The new hierarchy of mathematical classes introduced in Python 2.6 and Python 3.0 is a good example of this.

The `abc` module is included in Python 2.6 and later.

6.9.2 `multiprocessing` and `futures`

`multiprocessing` is a new module that helps you if you are using Python do to concurrent processing, letting you have process queues and use locks and semaphores for synchronizing the processes.

`multiprocessing` is included in Python 2.6 and later. It is also available for Python 2.4 and Python 2.5 on the CheeseShop[2].

If you do concurrency you may also want to take a look at the `futures` module which will be included in Python 3.2, and exists on the CheeseShop in a version that supports Python 2.5 and later[3].

6.9.3 `numbers` and `fractions`

Python 3 has a new class hierarchy of mathematical classes. For the most part you will not notice this, but one of the interesting results of this is the `fractions` module, available in Python 2.6 and later.

```
>>> from fractions import Fraction
>>> Fraction(3,4) / Fraction('2/3')
Fraction(9, 8)
```

There is also a `numbers` module that contains the abstract base classes for all the number types which is useful if you are implementing your own numeric types.

[2] http://pypi.python.org/pypi/multiprocessing
[3] http://pypi.python.org/pypi/futures/

CHAPTER

SEVEN

SUPPORTING PYTHON 2 AND 3 WITHOUT 2TO3 CONVERSION

Although the official documentation for Python 3 discourages writing code for both Python 2 and Python 3, in some cases it is desirable. Especially if you can drop support for Python 2.5 and earlier, since Python 2.6 introduces quite a lot of forwards compatibility.

It's possible to make the same code run under earlier versions of Python as well, but then you start getting into the "contorted" writing style the Python 3 documentation mentions. I'll take up tricks to do this and the `six` module I mention at the end of this chapter will help a lot. It has been done even for some quite big projects, but I would in general not recommend it for a large project. For small projects or parts of bigger projects, for example bootstrapping scripts, supporting old versions of Python without using `2to3` is often the best solution.

Python 2.7 has some small improvements on Python 3 compatibility, but it's likely that if you want to run the same code under both Python 2 and Python 3 you will have to support Python 2.6 for some time to come.

Many of the changes you need will be done by `2to3`, so to start converting your code you actually want to first run `2to3` on your code and make your code run under Python 3. It is generally easier, or at least less monotonous, to introduce Python 2 compatibility in Python 3 code, than to introduce Python 3 compatibility in Python 2 code.

Once you have the project running under Python 3, try to run it under Python 2.6. At this stage you may run into syntax errors. They should come from only changes in the `print` statement. Once you have fixed them you can fix the remaining errors and then lastly you do the same for earlier versions of Python, if you need to support them as well.

7.1 Supporting the `print()` function

One of the major causes of syntax errors is the change of `print` from a statement to a function. The simple cases are not problematic, you can simply put parentheses around the text that should be printed. The following will print exactly the same in all versions of Python:

```
>>> print("This works in all versions of Python!")
This works in all versions of Python!
```

However, if you use any more advanced feature of `print` you either end up with a syntax error or not printing what you intended. Python 2's trailing comma has in Python 3 become a parameter, so if you use trailing commas to avoid the newline after a print, this will in Python 3 look like `print('Text to print', end=' ')` which is a syntax error under Python 2.

Under Python 2.6 there is a `__future__` import to make `print` into a function. So to avoid any syntax errors and other differences you should start any file where you use `print()` with `from __future__ import print_function`. The `__future__` import only works under Python 2.6 and later, so for Python 2.5 and earlier you have two options. You can either convert the more complex `print` to something simpler, or you can use a separate print function that works under both Python 2 and Python 3.

Writing your own print function sounds more complicated than it is. The trick is to use `sys.stdout.write()` and formatting according to the parameters passed in to the function. However, it is even easier to use the `print_()` function from the `six` module that I will present at the end of this chapter.

7.2 Handling exceptions

If you in your exception handling need access to the exception itself you need to use an exception variable. In Python 2 the syntax for this is:

```
>>> try:
...     a = 1/0
... except ZeroDivisionError, e:
...     print e.args[0]
integer division or modulo by zero
```

However, this syntax is confusing when you need to catch more than one exception. You need to have parentheses around the list of exceptions:

```
>>> try:
...     a = 1/"Dinsdale"
... except (ZeroDivisionError, TypeError):
...     print "You can't divide by zero or by strings!"
You can't divide by zero or by strings!
```

If you forget the parentheses only `ZeroDivisionError` is caught and the raised exception will be stored in a variable named `TypeError`. That's not what you expect. Therefore, in Python 3, the syntax has changed to use `as` instead of a comma, which avoids this mistake. It will also give you a syntax error if you do not have parentheses when catching more than one exception:

```
>>> try:
...     a = 1/'0'
... except (ZeroDivisionError, TypeError) as e:
...     print(e.args[0])
unsupported operand type(s) for /: 'int' and 'str'
```

The above syntax also works in Python 2.6 where both the old syntax with a comma and the new syntax using `as` works. If you need to support Python versions lower than Python 2.6

and you need access to the exception that was raised, you can get that in all versions through the `exc_info()` function:

```
>>> import sys
>>> try:
...     a = 1/'0'
... except (ZeroDivisionError, TypeError):
...     e = sys.exc_info()[1]
...     print(e.args[0])
unsupported operand type(s) for /: 'int' and 'str'
```

Another difference is that Exceptions are no longer iterable. In Python 2 the arguments to the exception was available by iterating over the exception or subscripting the exception.

```
>>> try:
...    f = 1/0
... except ZeroDivisionError, e:
...    for m in e:
...        print m
...    print e[0]
integer division or modulo by zero
integer division or modulo by zero
```

In Python 3 you need to use the exception attribute `args` instead:

```
>>> try:
...    f = "1" + 1
... except TypeError as e:
...    for m in e.args:
...        print(m)
...    print(e.args[0])
Can't convert 'int' object to str implicitly
Can't convert 'int' object to str implicitly
```

A `message` attribute was added to the built-in exceptions in Python 2.5. It was however deprecated already in Python 2.6 and removed in Python 3. Python 2.6 and Python 2.7 will warn you about this when using the `-3` flag.

7.3 Import errors

One of the big changes is the reorganization of the standard library and as a result the typical errors you will get at this stage are mostly import errors. Getting around it is very easy. You simply try to import from the Python 3 locations and if that fails you import from the Python 2 locations. For modules that have been renamed, you can just import them as the new name.

```
>>> try:
...     import configparser
... except ImportError:
...     import ConfigParser as configparser
```

Some of the new modules are mergers of several old modules and in that case the above will not work if you need things from several of the old modules. You can also not do this

with modules that have sub-modules. `import httplib as http.client` is a syntax error. The `urllib` and `urllib2` modules have not only been merged, but reorganized into several sub-modules. So there you need to import each name you use separately. This often means you need to make changes to your code as well. In Python 2 retrieving a web page looks like this:

```
>>> import urllib
>>> import urlparse
>>>
>>> url = 'http://docs.python.org/library/'
>>> parts = urlparse.urlparse(url)
>>> parts = parts._replace(path='/3.0'+parts.path)
>>> page = urllib.urlopen(parts.geturl())
```

After conversion with `2to3` it will look like this:

```
>>> import urllib.request, urllib.parse, urllib.error
>>> import urllib.parse
>>>
>>> url = 'http://docs.python.org/library/'
>>> parts = urllib.parse.urlparse(url)
>>> parts = parts._replace(path='/3.0'+parts.path)
>>> page = urllib.request.urlopen(parts.geturl())
```

Yes, urllib.parse will be imported twice and urllib.error imported even though it isn't used. That's how this fixer does it and solving that would be a lot of extra effort, so it imports more than is needed. We need to fix up the code to import the names we use directly instead of the modules they are located in, so the version that runs under both Python 2 and Python 3 ends up like this:

```
>>> try:
...     from urllib.request import urlopen
...     from urllib.parse import urlparse
... except ImportError:
...     from urlparse import urlparse
...     from urllib import urlopen
...
>>> url = 'http://docs.python.org/library/'
>>> parts = urlparse(url)
>>> parts = parts._replace(path='/3.0'+parts.path)
>>> page = urlopen(parts.geturl())
```

7.4 Integer incompatibilities

There are two big changes in integer handling in Python 3. The first one is that the `int` and the `long` types have been merged. That means that you can't specify that an integer should be long by adding the `L` suffix any more. `1L` is a syntax error in Python 3.

If you have to have an integer being a `long` in Python 2 and still be compatible with Python 3, the following code will solve it. It defines up a `long` variable to be the same as the `int` class under Python 3, and it can then be used explicitly to make sure the integer is a `long`.

```
>>> import sys
>>> if sys.version > '3':
...     long = int
>>> long(1)
1L
```

Another change is that the syntax for octal literals has changed. In Python 2 any number starting with a `0` is an octal, which can be confusing if you decide to format your numbers by starting them with zeros. In Python 3 the syntax has instead changed to the less confusing `0o`, similar to `0x` used for hex numbers. Starting numbers with `0` is a syntax error to prevent you from trying the old octal syntax by mistake.

Octal is used almost exclusively when setting permissions under Unix, but that in turn is quite a common task. There is an easy way that works under both Python 2 and Python 3: Use the decimal or hex value and put the octal value in a comment:

```
>>> f = 420 # 644 in octal, 'rw-r--r--'
```

7.5 More bytes, strings and Unicode

It's no surprise that the trickiest issue we have in supporting Python 2 and Python 3 without `2to3` is handling binary data, just as it is when using `2to3`. When we choose not to use `2to3` the problem is made more tricky by making Unicode an issue as well. When using `2to3` to support Python 2 and Python 3, `2to3` will convert any Unicode literals to straight string literals. Without `2to3` we have no such luxury and since the Unicode literal `u''` is gone in Python 3 we need to find a portable way to say that we want a Unicode string.

Here, only supporting Python 3.3 will make things much easier for you, because in Python 3.3, the `u''` literal is back! In that case you can mostly ignore this section.

But if you need to support Python 3.1 or 3.2, the best way to do this is to make a Unicode string maker function just like the `b()` function in *Common migration problems* but for Unicode strings instead of binary `bytes`. The natural name for this function is of course `u()`. We would then use `b()` instead of the `b''` literal, and `u()` instead of the `u''` literal.

```
import sys
if sys.version < '3':
    import codecs
    def u(x):
        return codecs.unicode_escape_decode(x)[0]
else:
    def u(x):
        return x
```

This will return a unicode object in Python 2:

```
>>> from makeunicode import u
>>> u('GIF89a')
u'GIF89a'
```

While it will return a string object in Python 3:

```
>>> from makeunicode import u
>>> u('GIF89a')
'GIF89a'
```

Here I use the `unicode_escape` encoding, because other encodings could fail if you save the file with a different encoding than the one specified in the function. Using `unicode_escape` is a bit more work that just typing in the characters and saving the file but it will be portable both between different versions of Python and different platforms.

The `unicode_escape` encoding will convert all the various ways of entering unicode characters. The `'\x00'` syntax, the `'\u0000'` and even the `'\N{name}'` syntax:

```
>>> from makeunicode import u
>>> print(u('\u00dcnic\u00f6de'))
Ünicöde
>>> print(u('\xdcnic\N{Latin Small Letter O with diaeresis}de'))
Ünicöde
```

If you only need to support Python 2.6 or later there is also `from __future__ import unicode_literals`. This will turn all string literals in the file into Unicode literals:

```
>>> from __future__ import unicode_literals
>>> type("A standard string literal")
<type 'unicode'>
>>> type(b"A binary literal")
<type 'str'>
```

Both with the `__future__` import and the `u()` function the the binary data type is still called `str` and the text type is still called `unicode` under Python 2, while under Python 3 they are called `bytes` and `str`.

The best way around this is to define two variables; `text_type` and `binary_type`, depending on Python version and then we test against that variable.

```
>>> from __future__ import unicode_literals
>>> import sys
>>> if sys.version < '3':
...     text_type = unicode
...     binary_type = str
... else:
...     text_type = str
...     binary_type = bytes
>>> isinstance('U\xf1ic\xf6de', text_type)
True
```

For the handling of binary data you can use the same techniques as discussed in *Common migration problems* (page 35).

7.6 Two times three is “six”

There are many many more unusual and sometimes subtle differences between Python 2 and Python 3. Although the techniques mentioned here works for most of them, I definitely

recommend you to look at Benjamin Petersons module "six"[1] It contains a `PY3` constant to use when checking for the version of Python, and it contains the above mentioned `b()` and `u()` functions, although the `u()` function doesn't specify an encoding, so you are restricted to using ASCII characters. It also has many helpful constants like `text_type` and `binary_type` and a `print_()` function that works both under Python 2 and Python 3.

It also contains imports of much of the reorganized standard library, so instead of the `try/except` construct from the beginning of this chapter you can import the module from the six module instead. However it notably doesn't support the reorganization of the urllib and urllib2 modules, so there you still need to use the `try/except` import technique.

The six module even contains helpers for unusual problems, like using metaclasses and the attributes of functions, which also have been renamed. Although it requires Python 2.4 or later you can use many of the techniques in it even under earlier versions of Python, if you need to support them.

If you are attempting to support Python 2 and Python 3 without conversion you will definitely find it very helpful.

[1] http://pypi.python.org/pypi/six

CHAPTER

EIGHT

MIGRATING C EXTENSIONS

Python 3 has many changes to the C API, including in the API for defining classes and initializing modules. This means that every single C extension has to be modified to run under Python 3. Some of the changes are simple and some are not, but you will have to do them all by hand as `2to3` only handles Python code. That also means that you can't support Python 2 and Python 3 by `2to3` conversion. Luckily the the C pre-processor make it fairly easy to support both Python 2 and Python 3 from the same code. This is a standard way of supporting different versions of API's in C it will be standard fare for C-programmers. So there are no ugly hacks involved, just some less than pretty `#if` and `#ifndef` statements.

8.1 Before you start

There are some things you can do before you start the actual porting. The first one is to remove any usage of some old aliases you don't need any more. For example the `RO` macro has been removed. It was only a shorthand for `READONLY`, so if you used `RO` in your code you can replace it with `READONLY` instead.

Other common redefinitions are `statichere` and `staticforward`. They are workarounds for compatibility with certain compilers. For well behaving compilers they are just redefinitions of `static` so in Python 3 they are now gone since there are now have well behaving compilers on all platforms that CPython supports. If you use them in your code, you can replace them with `static`.

Another change you can do before porting is to move away from `PyClassObject`. That's the long deprecated "old-style classes" which are removed in Python 3. You should be able to move over to `PyTypeObject` with no big problem.

8.2 Object initialization

One of the less obvious errors encountered when porting C extensions to Python 3 is the error `"missing braces around initializer"`, which you will get when you initialize a Python class. You can indeed get around that by placing a couple of braces in the correct places, but a better solution is to replace the `PyObject_HEAD_INIT` macro to the `PyVarObject_HEAD_INIT` macro. The change was done to conform with C's strict aliasing rules, you can find more information in PEP 3123[1] if you are interested.

[1] http://www.python.org/dev/peps/pep-3123/

Code that previously looked like this:

```
static PyTypeObject mytype = {
    PyObject_HEAD_INIT(NULL)
    0,
    ...
};
```

Should now look like this:

```
static PyTypeObject mytype = {
    PyVarObject_HEAD_INIT(NULL, 0)
    ...
};
```

This will work in Python 2.6 and 2.7 as well. If you need to support Python 2.5 or earlier, you can simply define the `PyVarObject_HEAD_INIT` macro if it's missing:

```
#ifndef PyVarObject_HEAD_INIT
    #define PyVarObject_HEAD_INIT(type, size) \
        PyObject_HEAD_INIT(type) size,
#endif
```

Another change in the object header is that the `PyObject_HEAD` macro has changed so that `ob_type` is now in a nested structure. This means you no longer can pick the `ob_type` directly from the struct, so code like `ob->ob_type` stops working. You should replace this with `Py_TYPE(ob)`. The `Py_TYPE` macro doesn't appear until Python 2.6, so to support earlier versions we make another `#ifndef`:

```
#ifndef Py_TYPE
    #define Py_TYPE(ob) (((PyObject*)(ob))->ob_type)
#endif
```

In both cases the definitions above are taken directly from the Python 2.6 headers, where they are defined for forward compatibility purposes with Python 3. They work well in earlier Python versions as well, so this is a trick you can use as a general rule; if you need to use a macro that is defined in Python 3 and Python 2.6, just steal the Python 2.6 or Python 2.7 definition and put it inside an `#ifndef`.

8.3 Module initialization

Another big change is in the module initialization. There are many changes here, and as a result the module initialization usually ends up as part of the C extension with the most pre-processor commands or macros.

The family of functions to initialize modules, such as `Py_InitModule3` are gone. Instead, you should use `PyModule_Create`. Where `Py_InitModule3` took a couple of parameters `PyModule_Create` needs a `PyModuleDef` struct. If you want to support Python 2 you need to wrap this code with an `#if PY_MAJOR_VERSION >= 3`, both when you define the struct and when you use it.

```
#if PY_MAJOR_VERSION >= 3
    static struct PyModuleDef moduledef = {
        PyModuleDef_HEAD_INIT,
        "themodulename",     /* m_name */
        "This is a module",  /* m_doc */
        -1,                  /* m_size */
        module_functions,    /* m_methods */
        NULL,                /* m_reload */
        NULL,                /* m_traverse */
        NULL,                /* m_clear */
        NULL,                /* m_free */
    };
#endif

...

#if PY_MAJOR_VERSION >= 3
    m = PyModule_Create(&moduledef);
#else
    m = Py_InitModule3("themodulename",
        module_functions, "This is a module");
#endif
```

If you want to separate the `#if` statements from the code you can make a macro definition. I've used this one, although it doesn't support the extra functions like reload and traverse:

```
#if PY_MAJOR_VERSION >= 3
    #define MOD_DEF(ob, name, doc, methods) \
        static struct PyModuleDef moduledef = { \
            PyModuleDef_HEAD_INIT, name, doc, -1, methods, }; \
        ob = PyModule_Create(&moduledef);
#else
    #define MOD_DEF(ob, name, doc, methods) \
        ob = Py_InitModule3(name, methods, doc);
#endif
```

The definition of the module initialization function has also changed. In Python 2 you declared a function to initialize the module like this:

```
PyMODINIT_FUNC init<yourmodulename>(void)
```

In Python 3 this has changed to:

```
PyMODINIT_FUNC PyInit_<yourmodulename>(void)
```

It's not just the name that has changed; it's also the value of `PyMODINIT_FUNC`. In Python 2 it's typically `void` while in Python 3 it now returns a `PyObject*`. You have to return `NULL` if an error happened and you need to return the module object if initialization succeeded. There are various ways of dealing with this if you need both Python 3 and Python 2 support, starting with using multiple `#if PY_MAJOR_VERSION >= 3` in the function. However, that gets ugly, especially in the function definition:

```
PyMODINIT_FUNC
#if PY_MAJOR_VERSION >= 3
PyInit_<yourmodulename>(void)
#else
init<yourmodulename>(void)
#endif
{
...
```

It works, but it is not very readable code. It gets slightly better by using a macro:g

```
#if PY_MAJOR_VERSION >= 3
    #define MOD_INIT(name) PyMODINIT_FUNC PyInit_##name(void)
#else
    #define MOD_INIT(name) PyMODINIT_FUNC init##name(void)
#endif

MODINIT(themodulename)
{
...
}
```

But you still have to either have `#if` statements in the function to determine if you should return a value or not, or make yet another macro for that.

Another option is to define three functions. Firstly the actual module initialization function, returning a `PyObject*` and then two wrappers. One for Python 3 that calls the first and returns the value and one for Python 2 that calls the module initizaliation without returning a value:

```
// Python 3 module initialization
static PyObject *
moduleinit(void)
{
    MOD_DEF(m, "themodulename",
            "This is the module docstring",
    module_methods)

    if (m == NULL)
        return NULL;

    if (PyModule_AddObject(m, "hookable",
           (PyObject *)&hookabletype) < 0)
        return NULL;

    return m;
}

#if PY_MAJOR_VERSION < 3
    PyMODINIT_FUNC initthemodulename(void)
    {
        moduleinit();
    }
#else
```

```
    PyMODINIT_FUNC PyInit_themodulename(void)
    {
        return moduleinit();
    }
#endif
```

As you see the module initialization will in any case end up with a lot of `#if PY_MAJOR_VERSION >= 3`. A complete example of all these `#if` statements is this, taken from `zope.proxy`:

```
#if PY_MAJOR_VERSION >= 3
  static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "_zope_proxy_proxy", /* m_name */
    module___doc__,      /* m_doc */
    -1,                  /* m_size */
    module_functions,    /* m_methods */
    NULL,                /* m_reload */
    NULL,                /* m_traverse */
    NULL,                /* m_clear */
    NULL,                /* m_free */
  };
#endif

static PyObject *
moduleinit(void)
{
    PyObject *m;

#if PY_MAJOR_VERSION >= 3
    m = PyModule_Create(&moduledef);
#else
    m = Py_InitModule3("_zope_proxy_proxy",
                       module_functions, module___doc__);
#endif

    if (m == NULL)
        return NULL;

    if (empty_tuple == NULL)
        empty_tuple = PyTuple_New(0);

    ProxyType.tp_free = _PyObject_GC_Del;

    if (PyType_Ready(&ProxyType) < 0)
        return NULL;

    Py_INCREF(&ProxyType);
    PyModule_AddObject(m, "ProxyBase", (PyObject *)&ProxyType);

    if (api_object == NULL) {
        api_object = PyCObject_FromVoidPtr(&wrapper_capi, NULL);
        if (api_object == NULL)
```

```
        return NULL;
    }
    Py_INCREF(api_object);
    PyModule_AddObject(m, "_CAPI", api_object);

  return m;
}

#if PY_MAJOR_VERSION < 3
    PyMODINIT_FUNC
    init_zope_proxy_proxy(void)
    {
        moduleinit();
    }
#else
    PyMODINIT_FUNC
    PyInit__zope_proxy_proxy(void)
    {
        return moduleinit();
    }
#endif
```

If you don't like all the version tests, you can put all of these together before the function definition and use macros for anything that differs. Here is the same `zope.proxy` module, after I replaced all the `#if` tests with one block of definitions in the beginning:

```
#if PY_MAJOR_VERSION >= 3
  #define MOD_ERROR_VAL NULL
  #define MOD_SUCCESS_VAL(val) val
  #define MOD_INIT(name) PyMODINIT_FUNC PyInit_##name(void)
  #define MOD_DEF(ob, name, doc, methods) \
          static struct PyModuleDef moduledef = { \
            PyModuleDef_HEAD_INIT, name, doc, -1, methods, }; \
          ob = PyModule_Create(&moduledef);
#else
  #define MOD_ERROR_VAL
  #define MOD_SUCCESS_VAL(val)
  #define MOD_INIT(name) void init##name(void)
  #define MOD_DEF(ob, name, doc, methods) \
          ob = Py_InitModule3(name, methods, doc);
#endif

MOD_INIT(_zope_proxy_proxy)
{
    PyObject *m;

    MOD_DEF(m, "_zope_proxy_proxy", module___doc__,
            module_functions)

    if (m == NULL)
        return MOD_ERROR_VAL;

    if (empty_tuple == NULL)
```

```
        empty_tuple = PyTuple_New(0);

    ProxyType.tp_free = _PyObject_GC_Del;

    if (PyType_Ready(&ProxyType) < 0)
        return MOD_ERROR_VAL;

    Py_INCREF(&ProxyType);
    PyModule_AddObject(m, "ProxyBase", (PyObject *)&ProxyType);

    if (api_object == NULL) {
        api_object = PyCObject_FromVoidPtr(&wrapper_capi, NULL);
        if (api_object == NULL)
        return MOD_ERROR_VAL;
    }
    Py_INCREF(api_object);
    PyModule_AddObject(m, "_CAPI", api_object);

    return MOD_SUCCESS_VAL(m);

}
```

This is by far my preferred version, for stylistic reasons, but ultimately it's a matter of taste and coding style if you prefer the in-line `#if` statements or if you like to use many `#define` macros. So you choose what fits best with your coding style.

8.4 Changes in Python

The changes in Python are of course reflected in the C API. These are usually easy to handle. A typical example here is the unification of `int` and `long` types. Although in Python it behaves like the `long` type is gone, it's actually the `int` type that has been removed and the `long` type renamed. However, in the C API it hasn't been renamed. That means that all the functions that returned Python `int` objects are now gone and you need to replace them with the functions that returns Python `long` objects. This means that `PyInt_FromLong` must be replaced with `PyLong_FromLong`, `PyInt_FromString` with `PyLong_FromString` etc. If you need to keep Python 2 compatibility you have to replace it conditionally:

```
#if PY_MAJOR_VERSION >= 3
    PyModule_AddObject(m, "val", PyLong_FromLong(2));
#else
    PyModule_AddObject(m, "val", PyInt_FromLong(2));
#endif
```

Also in this case a `#define` makes for cleaner code if you need to do it more than once:

```
#if PY_MAJOR_VERSION < 3
    #define PyInt_FromLong PyLong_FromLong
#endif

PyModule_AddObject(m, "val", PyInt_FromLong(2));
```

In Python 3.2 the CObject API was removed. It was replaced with the Capsule API, which is also available for Python 2.7 and 3.1. For the simple usecase of just wrapping a C value the change is simple:

```
#if PY_MAJOR_VERSION < 3
    c_api = PyCObject_FromVoidPtr ((void *)pointer_to_value, NULL);
#else
    c_api = PyCapsule_New((void *)pointer_to_value, NULL, NULL);
#endif
```

Other things changed in Python that you are likely to encounter is the removal of the support for `__cmp__()` methods. The `_typeobject` structure used for defining a new type includes a place for a `__cmp__()` method definition. It's still there in Python 3 for compatibility but it's now ignored. The `cmpfunc` type definition and the `PyObject_Compare` functions have been removed as well. The only way to get full Python 3 compatibility here is to implement rich comparison functionality. There is support for that back to Python 2.1, so there is no problem with backwards compatibility.

8.5 Strings and Unicode

The changes in strings, Unicode and bytes are of course one of the biggest changes also when writing C extensions. In the C API, as with integers, there has been no renaming amongst the strings and the `unicode` type is still called `unicode`. The `str` type and all accompanying support functions are gone and the new `bytes` type has replaced it.

This means that if your extension returns or handles binary data you will in Python 2 get and return `PyString` objects, while you in Python 3 will handle `PyBytes` objects. Where you handle text data you should in Python 2 accept both `PyString` and `PyUnicode` while in Python 3 only `PyUnicode` is relevant. This can be handled with the same techniques as for `int` and `long` above, you can either make two versions of the code and choose between them with an `#if`, or you can redefine the missing `PyString` functions in Python 3 as either `PyBytes` or `PyUnicode` depending on what you need.

CHAPTER

NINE

EXTENDING 2TO3 WITH YOUR OWN FIXERS

The `2to3` command is a wrapper around a standard library package, called `lib2to3`. It contains a code parser, a framework for setting up fixers that modify the parse tree and a large set of fixers. The fixers that are included in `lib2to3` are enough to do all of the conversion you need for any normal porting. There are cases, however, where things are slightly beyond normal and you may have to write your own fixers. I first want to reassure you that these cases are very rare and you are unlikely to ever need this chapter and that you can skip it without feeling bad.

9.1 When fixers are necessary

It is strongly recommended that you don't change the API when you port your module or package to Python 3, but sometimes you have to. For example, the Zope Component Architecture, a collection of packages to help you componentize your system, had to change it's API. With the ZCA[1] you define interfaces that define the behavior of components and then make components that implement these interfaces. A simple example looks like this:

```
>>> from zope.interface import Interface, implements
>>>
>>> class IMyInterface(Interface):
...     def amethod():
...         '''This is just an example'''
...
>>> class MyClass(object):
...
...     implements(IMyInterface)
...
...     def amethod(self):
...         return True
```

The important line here is the `implements(IMyInterface)` line. It uses the way meta-classes are done in Python 2 for it's extensions, by using the `__metaclass__` attribute. However, in Python 3, there is no `__metaclass__` attribute and this technique doesn't work any longer. Thankfully class decorators arrived in Python 2.6, a new and better technique to do similar things. This is supported in the latest versions of `zope.interface`, with another syntax:

[1] http://www.muthukadan.net/docs/zca.html

```
>>> from zope.interface import Interface, implementer
>>>
>>> class IMyInterface(Interface):
...     def amethod():
...         '''This is just an example'''
...
>>> @implementer(IMyInterface)
... class MyClass(object):
...
...     def amethod(self):
...         return True
```

Since the first syntax no longer works in Python 3, `zope.interfaces` needs a custom fixer to change this syntax and the package `zope.fixers` contains just such a fixer. It is these types of advanced techniques that (ab)use Python internals that may force you to change the API of your package and if you change the API, you should write a fixer to make that change automatic, or you will cause a lot of pain for the users of your package.

So writing a fixer is a very unusual task. However, if you should need to write a fixer, you need any help you can get, because it is extremely confusing. So I have put down my experiences from writing `zope.fixers`, to try to remove some of the confusion and lead you to the right path from the start.

9.2 The Parse Tree

The `2to3` package contains support for parsing code into a parse tree. This may seem superfluous, as Python already has two modules for that, namely `parser` and `ast`, but the `parser` module uses Python's internal code parser, which is optimized to generate byte code and too low level for porting, while the `ast` module is designed to generate an abstract syntax tree and ignores all comments and formatting.

The parsing module of `2to3` is both high level and contains all formatting, but that doesn't mean it's easy to use. It can be highly confusing and the objects generated by parsed code may not be what you would expect at first glance. In general, the best hint I can give you when making fixers is to debug and step through the fixing process, looking closely at the parse tree until you start getting a feeling for how it works and then start manipulating it to see exactly what effects that has on the output. Having many unit tests is crucial to you make sure all the edge cases work.

The parse tree is made up of two types of objects; `Node` and `Leaf`. `Node` objects are containers that contain a series of objects, both `Node` and `Leaf`, while `Leaf` objects have no sub objects and contain the actual code.

`Leaf` objects have a type, telling you what it contains. Examples are `INDENT`, which means the indentation increased, `STRING` which is used for all strings, including docstrings, `NUMBER` for any kind of number, integers, floats, hexadecimal, etc, `RPAR` and `LPAR` for parentheses, `NAME` for any keyword or variable name and so on. The resulting parse tree does not contain much information about how the code relates to the Python language. It will not tell you if a `NAME` is a keyword or a variable, nor if a `NUMBER` is a an integer or a floating point value. However, the parser itself cares very much about Python grammar and will in general raise an error if it is fed invalid Python code.

One of the bigger surprises are that `Leaf` objects have a prefix and a suffix. These contain anything that isn't strictly code, including comments and white space. So even though there is a node type for comments, I haven't seen it in actual use by the parser. Indentation and dedentation are separate `Leaf` objects, but this will just tell you that indentation changed, not how much. Not that you need to know, the structure of the code is held by the hierarchy of `Node` objects, but if you do want to find out the indentation you will have to look at the prefix of the nodes. The suffix of a node is the same as the prefix of the next node and can be ignored.

9.3 Creating a fixer

To simplify the task of making a fixer there is a `BaseFix` class you can use. If you subclass from `BaseFix` you only need to override two methods, `match()` and `transform()`. `match()` should return a result that evaluates to false if the fixer doesn't care about the node and it should return a value that is not false when the node should be transformed by the fixer.

If `match()` returns a non-false result, `2to3` will then call the `transform()` method. It takes two values, the first one being the node and the second one being whatever `match()` returned. In the simplest case you can have `match()` return just `True` or `False` and in that case the second parameter sent to `transform()` will always be `True`. However, the parameter can be useful for more complex behavior. You can for example let the `match()` method return a list of sub-nodes to be transformed.

By default all nodes will be sent to `match()`. To speed up the fixer the refactoring methods will look at a fixer attribute called `_accept_type`, and only check the node for matching if it is of the same type. `_accept_type` defaults to `None`, meaning that it accepts all types. The types you can accept are listed in `lib2to3.pgen2.token`.

A fixer should have an `order` attribute that should be set to `"pre"` or `"post"`. This attribute decides in which order you should get the nodes, if you should get the leaves before their containing node (`"pre"`) or if the fixer should receive the leaves after it gets the containing node (`"post"`). The examples in this chapter are all based on `BaseFix`, which defaults to `"post"`.

You should follow a certain name convention when making fixers. If you want to call your fixer "something", the fixer module should be called `fix_something` and the fixer class should be called `FixSomething`. If you don't follow that convention, `2to3` may not be able to find your fixer.

9.4 Modifying the Parse Tree

The purpose of the fixer is for you to modify the parse tree so it generates code compatible with Python 3. In simple cases, this is easier than it sounds, while in complex cases it can be more tricky than expected. One of the main problems with modifying the parse tree directly is that if you replace some part of the parse tree the new replacement has to not only generate the correct output on its own but it has to be organized correctly. Otherwise the replacement can fail and you will not get the correct output when rendering the complete tree. Although the parse tree looks fairly straightforward at first glance, it can be quite convoluted. To help you generate parse trees that will generate valid code there is several helper functions in `lib2to3.fixer_util`. They range from the trivial ones as `Dot()` that

just returns a `Leaf` that generates a dot, to `ListComp()` that will help you generate a list comprehension. Another way is to look at what the parser generates when fed the correct code and replicate that.

A minimal example is a fixer that changes any mention of `oldname` to `newname`. This fixer does require the name to be reasonably unique, as it will change any reference to `oldname` even if it is not the one imported in the beginning of the fixed code.

```
from lib2to3.fixer_base import BaseFix
from lib2to3.pgen2 import token

class FixName1(BaseFix):

    _accept_type = token.NAME

    def match(self, node):
        if node.value == 'oldname':
            return True
        return False

    def transform(self, node, results):
        node.value = 'newname'
        node.changed()
```

Here we see that we only accept `NAME` nodes, which is the node for almost any bit of text that refers to an object, function, class etc. Only `NAME` nodes gets passed to the `match()` method and there we then check if the value is `oldname` in which case True is returned and the node is passed to the `transform()` method.

As a more complex example I have a fixer that changes the indentation to 4 spaces. This is a fairly simple use case, but as you can see it's not entirely trivial to implement. Although it is basically just a matter of keeping track of the indentation level and replacing any new line with the current level of indentation there are still several special cases. The indentation change is also done on the prefix value of the node and this may contain several lines, but only the last line is the actual indentation.

```
from lib2to3.fixer_base import BaseFix
from lib2to3.fixer_util import Leaf
from lib2to3.pgen2 import token

class FixIndent(BaseFix):

    indents = []
    line = 0

    def match(self, node):
        if isinstance(node, Leaf):
            return True
        return False

    def transform(self, node, results):
        if node.type == token.INDENT:
            self.line = node.lineno
```

```
            # Tabs count like 8 spaces.
            indent = len(node.value.replace('\t', ' ' * 8))
            self.indents.append(indent)
            # Replace this indentation with 4 spaces per level:
            new_indent = ' ' * 4 * len(self.indents)
            if node.value != new_indent:
                node.value = new_indent
                # Return the modified node:
                return node
        elif node.type == token.DEDENT:
            self.line = node.lineno
            if node.column == 0:
                # Complete outdent, reset:
                self.indents = []
            else:
                # Partial outdent, we find the indentation
                # level and drop higher indents.
                level = self.indents.index(node.column)
                self.indents = self.indents[:level+1]
                if node.prefix:
                    # During INDENT's the indentation level is
                    # in the value. However, during OUTDENT's
                    # the value is an empty string and then
                    # indentation level is instead in the last
                    # line of the prefix. So we remove the last
                    # line of the prefix and add the correct
                    # indententation as a new last line.
                    prefix_lines = node.prefix.split('\n')[:-1]
                    prefix_lines.append(' ' * 4 *
                                        len(self.indents))
                    new_prefix = '\n'.join(prefix_lines)
                    if node.prefix != new_prefix:
                        node.prefix = new_prefix
                        # Return the modified node:
                        return node
        elif self.line != node.lineno:
            self.line = node.lineno
            # New line!
            if not self.indents:
                # First line. Do nothing:
                return None
            else:
                # Continues the same indentation
                if node.prefix:
                    # This lines intentation is the last line
                    # of the prefix, as during DEDENTS. Remove
                    # the old indentation and add the correct
                    # indententation as a new last line.
                    prefix_lines = node.prefix.split('\n')[:-1]
                    prefix_lines.append(' ' * 4 *
                                        len(self.indents))
                    new_prefix = '\n'.join(prefix_lines)
                    if node.prefix != new_prefix:
```

```
                    node.prefix = new_prefix
                    # Return the modified node:
                    return node

        # Nothing was modified: Return None
        return None
```

This fixer is not really useful in practice and is only an example. This is partly because some things are hard to automate. For example it will not indent multi-line string constants, because that would change the formatting of the string constant. However, docstrings probably should be re-indented, but the parser doesn't separate docstrings from other strings. That's a language feature and the `2to3` parser only parses the syntax, so I would have to add code to figure out if a string is a docstring or not.

Also it doesn't change the indentation of comments, because they are a part of the prefix. It would be possible to go through the prefix, look for comments and re-indent them too, but we would then have to assume that the comments should have the same indentation as the following code, which is not always true.

9.5 Finding the nodes with Patterns

In the above cases finding the nodes in the `match()` method is relatively simple, but in most cases you are looking for something more specific. The renaming fixer above will for example rename all cases of `oldname`, even if it is a method on an object and not the imported function at all. Writing matching code that will find exactly what you want can be quite complex, so to help you `lib2to3` has a module that will do a grammatical pattern matching on the parse tree. As a minimal example we can take a pattern based version of the fixer that renamed `oldname` to `newname`.

You'll note that here I don't replace the value of the node, but make a new node and replace the old one. This is only to show both techniques, there is no functional difference.

```
from lib2to3.fixer_base import BaseFix
from lib2to3.fixer_util import Name

class FixName2(BaseFix):

    PATTERN = "fixnode='oldname'"

    def transform(self, node, results):
        fixnode = results['fixnode']
        fixnode.replace(Name('newname', prefix=fixnode.prefix))
```

When we set the `PATTERN` attribute of our fixer class to the above pattern `BaseFix` will compile this pattern into matcher and use it for matching. You don't need to override `match()` anymore and `BaseFix` will also set `_accept_type` correctly, so this simplifies making fixers in many cases.

The difficult part of using pattern fixers is to find what pattern to use. This is usually far from obvious, so the best way is to feed example code to the parser and then convert that tree to a pattern via the code parser. This is not trivial, but thankfully Collin Winter has

written a script called `find_pattern.py`[2] that does this. This makes finding the correct pattern a lot easier and really helps to simplify the making of fixes.

To get an example that is closer to real world cases, let us change the API of a module, so that what previously was a constant now is a function call. We want to change this:

```
from foo import CONSTANT

def afunction(alist):
    return [x * CONSTANT for x in alist]
```

into this:

```
from foo import get_constant

def afunction(alist):
    return [x * get_constant() for x in alist]
```

In this case changing every instance of `CONSTANT` into `get_constant` will not work as that would also change the import of the name to a function call which would be a syntax error. We need to treat the import and the usage separately. We'll use `find_pattern.py` to look for patterns to use.

The user interface of `find_pattern.py` is not the most verbose, but it is easy enough to use once you know it. If we run:

```
$ find_pattern.py -f example.py
```

it will parse that file and print out the various nodes it finds. You press enter for each code snipped you don't want and you press `y` for the code snippet you do want. It will then print out a pattern that matches that code snippet. You can also type in a code snippet as an argument, but that becomes fiddly for multi-line snippets.

If we look at the first line of our example, it's pattern is:

```
import_from< 'from' 'foo' 'import' 'CONSTANT' >
```

Although this will be enough to match the import line we would then have to find the `CONSTANT` node by looking through the tree that matches. What we want is for the transformer to get a special handle on the `CONSTANT` part so we can replace it with `get_constant` easily. We can do that by assigning a name to it. The finished pattern then becomes:

```
import_from< 'from' 'foo' 'import' importname='CONSTANT' >
```

The `transform()` method will now get a dictionary as the `results` parameter. That dictionary will have the key `'node'` which contains the node that matches all of the pattern and it will also contain they key `'importname'` which contains just the `CONSTANT` node.

We also need to match the usage and here we match `'CONSTANT'` and assign it to a name, like in the renaming example above. To include both patterns in the same fixer we separate them with a `|` character:

[2] http://svn.python.org/projects/sandbox/trunk/2to3/scripts/find_pattern.py

```
import_from< 'from' 'foo' 'import' importname='CONSTANT'>
|
constant='CONSTANT'
```

We then need to replace the `importname` value with `get_constant` and replace the `constant` node with a call. We construct that call from the helper classes `Call` and `Name`. When you replace a node you need to make sure to preserve the prefix, or both white-space and comments may disappear:

```
node.replace(Call(Name(node.value), prefix=node.prefix))
```

This example is still too simple. The patterns above will only fix the import when it is imported with `from foo import CONSTANT`. You can also `import foo` and you can rename either `foo` or `CONSTANT` with an `import as`. You also don't want to change every usage of `CONSTANT` in the file, it may be that you also have another module that also have something called `CONSTANT` and you don't want to change that.

As you see, the principles are quite simple, while in practice it can become complex very quickly. A complete fixer that makes a function out of the constant would therefore look like this:

```
from lib2to3.fixer_base import BaseFix
from lib2to3.fixer_util import Call, Name, is_probably_builtin
from lib2to3.patcomp import PatternCompiler

class FixConstant(BaseFix):

    PATTERN = """
        import_name< 'import' modulename='foo' >
        |
        import_name< 'import' dotted_as_name< 'foo' 'as'
           modulename=any > >
        |
        import_from< 'from' 'foo' 'import'
           importname='CONSTANT' >
        |
        import_from< 'from' 'foo' 'import' import_as_name<
           importname='CONSTANT' 'as' constantname=any > >
        |
        any
        """

    def start_tree(self, tree, filename):
        super(FixConstant, self).start_tree(tree, filename)
        # Reset the patterns attribute for every file:
        self.usage_patterns = []

    def match(self, node):
        # Match the import patterns:
        results = {"node": node}
        match = self.pattern.match(node, results)

        if match and 'constantname' in results:
```

```
            # This is an "from import as"
            constantname = results['constantname'].value
            # Add a pattern to fix the usage of the constant
            # under this name:
            self.usage_patterns.append(
                PatternCompiler().compile_pattern(
                    "constant='%s'"%constantname))
            return results

        if match and 'importname' in results:
            # This is a "from import" without "as".
            # Add a pattern to fix the usage of the constant
            # under it's standard name:
            self.usage_patterns.append(
                PatternCompiler().compile_pattern(
                    "constant='CONSTANT'"))
            return results

        if match and 'modulename' in results:
            # This is a "import as"
            modulename = results['modulename'].value
            # Add a pattern to fix the usage as an attribute:
            self.usage_patterns.append(
                PatternCompiler().compile_pattern(
                "power< '%s' trailer< '.' " \
                "attribute='CONSTANT' > >" % modulename))
            return results

        # Now do the usage patterns
        for pattern in self.usage_patterns:
            if pattern.match(node, results):
                return results

    def transform(self, node, results):
        if 'importname' in results:
            # Change the import from CONSTANT to get_constant:
            node = results['importname']
            node.value = 'get_constant'
            node.changed()

        if 'constant' in results or 'attribute' in results:
            if 'attribute' in results:
                # Here it's used as an attribute.
                node = results['attribute']
            else:
                # Here it's used standalone.
                node = results['constant']
                # Assert that it really is standalone and not
                # an attribute of something else, or an
                # assignment etc:
                if not is_probably_builtin(node):
                    return None
```

```
        # Now we replace the earlier constant name with the
        # new function call. If it was renamed on import
        # from 'CONSTANT' we keep the renaming else we
        # replace it with the new 'get_constant' name:
        name = node.value
        if name == 'CONSTANT':
            name = 'get_constant'
        node.replace(Call(Name(name), prefix=node.prefix))
```

The trick here is in the match function. We have a `PATTERN` attribute that will match all imports, but it also contains the pattern `any` to make sure we get to handle all nodes. This makes it slower, but is necessary in this case. The alternative would be to have separate fixers for each of the four import cases, which may very well be a better solution in your case.

In general, any real world fixer you need to write will be very complex. If the fix you need to do is simple, you are certainly better off making sure your Python 3 module and Python 2 module are compatible. However, I hope the examples provided here will be helpful. The fixers in `lib2to3` are also good examples, even though they unfortunately are not very well documented.

APPENDIX

A

LANGUAGE DIFFERENCES AND WORKAROUNDS

This appendix contains a listing of the differences between Python 2 and Python 3 and example code that will run both in Python 2 and Python 3 without `2to3` conversion.

This listing is incomplete. What is listed here is only the intentional changes that are not bug fixes and even so there may be accidental ommissions.

A.1 `apply()`

2to3 fixer ☑ six support ☐

The Python 2 builtin `apply()` has been removed in Python 3. It's used to call a function, but since you can call the function directly it serves no purpose and has been deprecated since Python 2.3. There is no replacement.

A.2 `buffer()`

2to3 fixer ☑ six support ☐

The Python 2 `buffer()` builtin is replaced by the `memoryview` class in Python 3. They are not fully compatible, so `2to3` does not change this unless you explicitly specify the `buffer` fixer.

This code will run in both Python 2 and Python 3 without `2to3` conversion:

```
>>> import sys
>>> if sys.version > '3':
...     buffer = memoryview
>>> b = buffer('yay!'.encode())
>>> len(b)
4
```

A.3 `callable()`

2to3 fixer ☑ six support ☑

The Python 2 builtin `callable()` was removed in Python 3.0, but reintroduced in Python 3.2. If you need to support Python 3.1 you can try to call the object under scrutiny and catch the `TypeError` if it is not callable.

If you need to know if something is callable without calling it, there are several solutions for Python 3:

```
>>> def afunction():
...     pass

>>> any("__call__" in klass.__dict__ for
...     klass in type(afunction).__mro__)
True

>>> import collections
>>> isinstance(afunction, collections.Callable)
True
```

If you need code that runs in both Python 2 and Python 3 without `2to3` conversion, you can use this:

```
>>> hasattr(bool, '__call__')
True
```

The `six` module also defines a `callable` function for use under Python 3.

A.4 Classes

2to3 fixer ☐ six support ☐

In Python 2 there is two types of classes, “old-style” and “new”. The “old-style” classes have been removed in Python 3.

See also *Use new-style classes* (page 17)

A.5 Comparisons

2to3 fixer ☐ six support ☐

The Python 2 builtin `cmp()` has been removed in Python 3.0.1, although it remained in Python 3.0 by mistake. It is mostly used when defining the `__cmp__` comparison method or functions to pass as `cmp` parameters to `.sort()` and the support for this has been removed in Python 3 as well.

Should you need `cmp()` you can define it like this:

```
def cmp(a, b):
    return (a > b) - (a < b)
```

See *Unorderable types, __cmp__ and cmp* (page 36) for more information.

A.6 `coerce()` and `__coerce__`

2to3 fixer ☐ six support ☐

The `coerce()` builtin function and the support for the `__coerce__` method has been removed in Python 3. `coerce()` would convert the numeric arguments to have the same type according to the coercion rules for Pythons arithmetic operators and was only useful in early versions of Python when implementing new numeric types. There is no replacement in Python 3; coercion should instead be done by the numeric operator methods.

A.7 Dictionary methods

2to3 fixer ☑ six support ☐

In Python 2 dictionaries have the methods `iterkeys()`, `itervalues()` and `iteritems()` that return iterators instead of lists. In Python 3 the standard `keys()`, `values()` and `items()` return dictionary views, which are iterators, so the iterator variants become pointless and are removed.

If you need to support both Python 2 and Python 3 without `2to3` conversion and you must use the iterator methods, you can access it via a `try/except`:

```
>>> d = {'key1': 'value1',
...      'key2': 'value2',
...      'key3': 'value3',
... }

>>> try:
...     values = d.itervalues()
... except AttributeError:
...     values = d.values()

>>> isinstance(values, list)
False

>>> for value in values:
...     print(value)
value3
value2
value1
```

Also, the `has_key()` method on dictionaries is gone. Use the `in` operator instead.

See also *Make sure you aren't using any removed modules* (page 23)

A.8 except

2to3 fixer ☑ six support ☐

In Python 2 the syntax to catch exceptions have changed from:

```
except (Exception1, Exception2), target:
```

to the clearer Python 3 syntax:

```
except (Exception1, Exception2) as target:
```

Other differences is that the target no longer can be a tuple and that string exceptions are gone. `2to3` will convert all this, except string exceptions.

Both syntaxes work in Python 2.6 and Python 2.7, but if you need code that is to run in earlier versions as well as Python 3 without `2to3` conversion you can get the exception object through `sys.exc_info()`:

```
>>> import sys
>>> try:
...     raise Exception("Something happened")
... except Exception:
...     e = sys.exc_info()[1]
...     print(e.args[0])
Something happened
```

A.9 Exception objects

2to3 fixer ☑ six support ☐

In Python 2 the exception object is iterable and indexable:

```
>>> e = Exception('arg1', 'arg2')
>>> e[1]
'arg2'
>>> for a in e:
...   print a
...
arg1
arg2
```

In Python 3 you must use the `args` attribute, which will work under Python 2 as well.

```
>>> e = Exception('arg1', 'arg2')
>>> e.args[1]
'arg2'
>>> for a in e.args:
...   print a
...
arg1
arg2
```

There was also a `message` attribute on exceptions introduced in Python 2.5, but it was deprecated already in Python 2.6, so it's unlikely that you will use it.

A.10 `exec`

2to3 fixer ☑ six support ☑

In Python 2 `exec` is a statement:

```
>>> g_dict={}
>>> l_dict={}
>>> exec "v = 3" in g_dict, l_dict
>>> l_dict['v']
3
```

In Python 3 `exec` is a function:

```
>>> g_dict={}
>>> l_dict={}
>>> exec("v = 3", g_dict, l_dict)
>>> l_dict['v']
3
```

The Python 3 syntax without the global and local dictionaries will work in Python 2 as well:

```
>>> exec("v = 3")
>>> v
3
```

If you need to pass in the global or local dictionaries you will need to define a custom function with two different implementations, one for Python 2 and one for Python 3. As usual `six` includes an excellent implementation of this called `exec_()`.

A.11 `execfile`

2to3 fixer ☑ six support ☐

The Python 2 `execfile` statement is gone on Python 3. As a replacement you can open the file and read the contents:

```
exec(open(thefile).read())
```

This works in all versions of Python.

A.12 `file`

2to3 fixer ☐ six support ☐

In Python 2 there is a `file` type builtin. This is replaced with various file types in Python 3. You commonly see code in Python 2 that uses `file(pathname)` which will fail in Python 3. Replace this usage with `open(pathname)`.

If you need to test for types you can in Python 3 check for `io.IOBase` instead of `file`.

A.13 `filter()`

2to3 fixer ☑ six support ☐

In Python 2 `filter()` returns a list while in Python 3 it returns an iterator. `2to3` will in some cases place a `list()` call around the call to `filter()` to ensure that the result is still a list. If you need code that runs in both Python 2 and Python 3 without `2to3` conversion and you need the result to be a list, you can do the same.

A.14 Imports

2to3 fixer ☑ six support ☐

In Python 2, if you have a package called `mypackage` and that contains a module called `csv.py`, it would hide the `csv` module from the standard library. The code `import csv` would within `mypackage` import the local file, and importing from the standard library would become tricky.

In Python 3, this has changed so that `import csv` would import from the standard library, and to import the local `csv.py` file you need to write `from . import csv` and `from csv import my_csv` needs to be changed to `from .csv import my_csv`. These are called "relative imports", and there is also a syntax to import from one level up module above; `from .. import csv`.

If you to support both Python 2 and Python 3 without `2to3` the `from .` and `from ..` syntax has been available since Python 2.5, together with a `from __future__ import absolute_import` statement that changes the behavior to the Python 3 behavior.

If you need to support Python 2.4 or earlier you have to spell out the whole package name so `import csv` becomes `from mypkg import csv` and `from csv import my_csv` becomes `from mypckg.csv import my_csv`. For clarity and readability I would avoid relative imports if you can and always spell out the whole path.

`2to3` will check if your imports are local and change them.

A.15 Indentation

2to3 fixer ☐ six support ☐

In Python 2 a tab will be equal to eight spaces as indentation, so you can indent one line with a tab, and the next line with eight spaces. This is confusing if you are using an editor that expands tabs to another number than eight spaces.

In Python 3 a tab is only equal to another tab. This means that each indentation level has to be consistent in its use of tabs and spaces. If you have a file where an indented block sometimes uses spaces and sometimes tabs, you will get the error `TabError: inconsistent use of tabs and spaces in indentation`.

The solution is of course to remove the inconsistency.

A.16 `input()` and `raw_input()`

2to3 fixer ☑ six support ☐

In Python 2 there is `raw_input()` that takes a string from `stdin` and `input()` that takes a string from `stdin` and evaluates it. That last function is not very useful and has been removed in Python 3, while `raw_input()` has been renamed to `input()`.

If you need to evaluate the input string you can use `eval()`:

```
>>> eval(input('Type in an expression: '))
'Type in an expression: ' 1+2
3
```

If you need code that runs in both Python 2 and Python 3 without `2to3` conversion you can conditionally set `input()` to be `raw_input()`:

```
>>> try:
...     input = raw_input
... except NameError:
...     pass

>>> input('Type in a string: ')
Type in a string: It works!
'It works!'
```

A.17 Integer division

2to3 fixer ☐ six support ☐

In Python 2, the result of dividing two integers will itself be an integer; in other words `1/2` returns `0`. In Python 3 integer division will return an integer only if the result is a whole number. So `1/2` will return `0.5`.

If you want the old behavior you should instead use the floor division operator `//`, available since Python 2.2. If you need to support both Python 2 and Python 3 without `2to3` conversion the following `__future__` import works since Python 2.2 and enables the new behavior:

```
>>> from __future__ import division
>>> 1/2
0.5
```

See also: *Use / / instead of / when dividing integers* (page 16)

A.18 `long`

2to3 fixer ☐ six support ☑ (partial)

Python 2 has two integer types `int` and `long`. These have been unified in Python 3, so there is now only one type, `int`. This means that the following code fails in Python 3:

```
>>> 1L
1L
>>> long(1)
1L
```

It's quite unusual that you would need to specify that an integer should be a long in Python 2. If you do and you need code that runs in both Python 2 and Python 3 without `2to3` conversion, the following code works:

```
>>> import sys
>>> if sys.version > '3':
...     long = int
>>> long(1)
1L
```

However, the representation is still different, so doctests will fail.

If you need to check if something is a number you need to check against both `int` and `long` under Python 2, but only `int` in Python 3. The best way to do that is to set up a `integer_types` tuple depending on Python version and test against that. `six` includes this:

```
>>> import sys
>>> if sys.version < '3':
...     integer_types = (int, long,)
... else:
...     integer_types = (int,)
>>> isinstance(1, integer_types)
True
```

A.19 `map()`

2to3 fixer ☐ six support ☐

In Python 2 `map()` returns a list while in Python 3 it returns an iterator. `2to3` will in some cases place a `list()` call around the call to `map()` to ensure that the result is still a list. If you need code that runs in both Python 2 and Python 3 without `2to3` conversion and you need the result to be a list, you can do the same.

In Python 2 `map()` will continue until the longest of the argument iterables are exhausted, extending the other arguments with `None`.

```
>>> def fun(a, b):
...     if b is not None:
...         return a - b
...     return -a
>>> map(fun, range(5), [3,2,1])
[-3, -1, 1, -3, -4]
```

In Python 3 `map()` will instead stop at the shortest of the arguments. If you want the Python 2 behaviour in Python 3 you can use a combination of `starmap()` and `zip_longest()`.

```
>>> from itertools import starmap, zip_longest
>>> def fun(a, b):
...     if b is not None:
...         return a - b
...     return -a
>>> list(starmap(fun, zip_longest(range(5), [3,2,1])))
[-3, -1, 1, -3, -4]
```

The Python 2 `map()` will accept `None` as it's function argument, where it will just return the object(s) passed in. As this transforms `map()` into `zip()` it's not particularily useful, and in Python 3 this no longer works. However, some code dependes on this behavior, and you can use the following function as a full replacement for the Python 2 map.

```
from itertools import starmap, zip_longest

def map(func, *iterables):
    zipped = zip_longest(*iterables)
    if func is None:
        # No need for a NOOP lambda here
        return zipped
    return starmap(func, zipped)
```

A.20 Metaclasses

2to3 fixer ☑ six support ☑

In Python 2 you specified the metaclass with the `__metaclass__` attribute. In Python 3 you instead pass in a `metaclass` parameter in the class definition. Supporting metaclasses in Python 2 and Python 3 without using `2to3` requires you to create classes on the fly.

If you want this, I highly recommend to use the `six` module, which has a very clever `with_metaclass()` function.

A.21 `.next()`

2to3 fixer ☑ six support ☑

In Python 2 you get the next result from an iterator by calling the iterators `.next()` method. In Python 3 there is instead a `next()` builtin.

If you need code that runs in both Python 2 and Python 3 without `2to3` conversion you can make a function that under Python 2 calls `iterator.next()` and under Python 3 calls `next(iterator)`. The `six` module contains such a function, called `advance_iterator()`.

A.22 Parameter unpacking

2to3 fixer ☐ six support ☐

In Python 2 you have parameter unpacking:

```
>>> def unpacks(a, (b, c)):
...     return a,b,c

>>> unpacks(1, (2,3))
(1, 2, 3)
```

Python 3 does not support this, so you need to do your own unpacking:

```
>>> def unpacks(a, b):
...     return a,b[0],b[1]

>>> unpacks(1, (2,3))
(1, 2, 3)
```

A.23 `print`

2to3 fixer ☑ six support ☑

The Python 2 `print` statement is in Python 3 a function. If you need to run the same code in both Python 2 and Python 3 without `2to3` conversion there are various techniques for this. This is discussed in detail in *Supporting the print() function* (page 57).

A.24 `raise`

2to3 fixer ☑ six support ☑

In Python 2 the syntax for the `raise` statement is:

```
raise E, V, T
```

Where `E` is a string, an exception class or an exception instance, `V` the an optional exception value in the case that `E` is a class or a string and `T` is a `traceback` object if you want to supply a traceback from a different place than the current code. In Python 3 this has changed to:

```
raise E(V).with_traceback(T)
```

As with the Python 2 syntax, `value` and `traceback` are optional. The syntax without the traceback variable is:

```
raise E(V)
```

This works in all versions of Python. It's very unusual that you need the traceback parameter, but if you do and you also need to write code that runs under Python 2 and Python 3 without using `2to3` you need to create different a function that takes `E`, `V` and `T` as parameters and have different implementations under Python 2 and Python 3 for that function. The `six` module has a nice implementation of that, called `reraise()`.

A.25 `range()` and `xrange()`

2to3 fixer ☑ six support ☑

In Python 2 `range()` returns a list, and `xrange()` returns an object that will only generate the items in the range when needed, saving memory.

In Python 3, the `range()` function is gone, and `xrange()` has been renamed `range()`. In addition the `range()` object support slicing in Python 3.2 and later .

`2to3` will in some cases place a `list()` call around the call to `range()`, to ensure that the result is still a list. If you need code that runs in both Python 2 and Python 3 without `2to3` conversion and you need the result to be a list, you can do the same.

You can import `xrange()` from the `six` module to be sure you get the iterator variation under both Python 2 and Python 3.

A.26 `repr()` as backticks.

2to3 fixer ☑ six support ☐

In Python 2 you can generate a string representation of an expression by enclosing it with backticks:

```
>>> `sorted`
'<built-in function sorted>'

>>> `2+3`
'5'
```

The only purpose with this syntax is to confuse newbies and make obfuscated Python. It has been removed in Python 3, since the `repr()` builtin does exactly the same.

```
>>> repr(sorted)
'<built-in function sorted>'

>>> repr(2+3)
'5'
```

A.27 Rounding behavior

2to3 fixer ☐ six support ☐

The behavior of `round` has changed in Python 3. In Python 2, rounding of halfway cases was away from zero, and `round()` would always return a float.

```
>>> round(1.5)
2.0
>>> round(2.5)
3.0
>>> round(10.0/3, 0)
3.0
```

In Python 3 rounding of halfway cases are now always towards the nearest even. This is standard practice, as it will make a set of evenly distributed roundings average out.

When called without the second parameter, which determines the number of decimals, `round()` will in Python 3 return an integer. If you pass in a parameter to set the number of decimals to round to, the returned value will be of the same type as the unrounded value. This is true even if you pass in zero.

```
>>> round(1.5)
2
>>> round(2.5)
2
>>> round(10.0/3, 0)
3.0
```

If you need the Python 2 behavior, you can use the following method:

```
>>> import math
>>> def my_round(x, d=0):
...     p = 10 ** d
...     return float(math.floor((x * p) + math.copysign(0.5, x)))/p

>>> my_round(1.5)
2.0
>>> my_round(2.5)
3.0
>>> my_round(10.0/3, 0)
3.0
```

A.28 Slice operator methods

2to3 fixer ☐ six support ☐

In Python 1 you used `__getslice__` and `__setslice__` to support slice methods like `foo[3:7]` on your object. These were deprecated in Python 2.0 but still supported. Python 3 removes the support for the slice methods, so you need to instead extend `__getitem__`, `__setitem__` and `__delitem__` with slice object support.

```
>>> class StrawberryTart(object):
...
...    def __getitem__(self, n):
...        """An example of how to use slice objects"""
...        if isinstance(n, slice):
...            # Expand the slice object using range()
...            # to a maximum of eight items.
...            return [self[x] for x in
...                    range(*n.indices(8))]
...
...        # Return one item of the tart
...        return 'A slice of StrawberryTart with ' \
...               'not so much rat in it.'
...
>>> tart = StrawberryTart()
>>> tart[5:6]
['A slice of StrawberryTart with not so much rat in it.']
```

A.29 Sorting

2to3 fixer ☐ six support ☐

In Python 2 the `.sort()` method on lists as well as the `sorted()` builtin takes two parameters, `cmp` and `key`. In Python 3 only the `key` parameter is supported. There are no fixers for this, so you need to change that in the Python 2 code.

See *When sorting, use key instead of cmp* (page 18) for more information.

A.30 `StandardError`

2to3 fixer ☑ six support ☐

Python 2 has an exception class called `StandardError` that has been removed in Python 3. Use `Exception` instead.

A.31 String types

2to3 fixer ☑ six support ☑

Python 2 had two string types; `str` and `unicode`. Python 3 has only one; `str`, but instead it also has a `bytes` type made to handle binary data. For more information on this, see *Bytes, strings and Unicode* (page 38) and *More bytes, strings and Unicode* (page 61).

APPENDIX

B

REORGANIZATIONS AND RENAMINGS

B.1 The standard library

The Python standard library has been reorganized in Python 3 to be more consistent and easier to use. All the module names now conform to the style guide for Python code, PEP 8[1], and several modules have been merged.

`2to3` contains fixers for all of this, so this sectin is mostly of interest if you need to support both Python 2 and Python 3 without `2to3` conversion.

The `six` module[2] has support for most of the standard library reorganization. You can import the reorganized modules from `six.moves`:

```
>>> from six.moves import cStringIO
```

In Python 2 this will be equivalent to:

```
>>> from cStringIO import StringIO
```

While in Python 3 this will be equivalent to:

```
>>> from io import StringIO
```

If you want to support Python 2 and Python 3 without conversion and you don't want to use the `six` module, this is also very easy. You just try to import from one location, catch the error and then import from the other location. It doesn't matter of you put the Python 3 location first or last, both works equally well:

```
>>> try:
...     from io import StringIO
... except ImportError:
...     from cStringIO import StringIO
```

This table contains the renamings and reorganizations of the standard library, except for the `urllib`, `urllib2` and `urlparse` reorganization, which has a separate table:

[1] http://www.python.org/dev/peps/pep-0008/
[2] http://pypi.python.org/pypi/six

Python 2 name	Python 3 name	six name
anydbm	dbm	
BaseHTTPServer	http.server	BaseHTTPServer
__builtin__	builtins	builtins
CGIHTTPServer	http.server	CGIHTTPServer
ConfigParser	configparser	configparser
copy_reg	copyreg	copyreg
cPickle	pickle	cPickle
cProfile	profile	
cStringIO.StringIO	io.StringIO	cStringIO
Cookie	http.cookies	http_cookies
cookielib	http.cookiejar	http_cookiejar
dbhash	dbm.bsd	
dbm	dbm.ndbm	
dumbdb	dbm.dumb	
Dialog	tkinter.dialog	tkinter_dialog
DocXMLRPCServer	xmlrpc.server	
FileDialog	tkinter.FileDialog	tkinter_filedialog
FixTk	tkinter._fix	
gdbm	dbm.gnu	
htmlentitydefs	html.entities	html_entities
HTMLParser	html.parser	html_parser
httplib	http.client	http_client
markupbase	_markupbase	
Queue	queue	queue
repr	reprlib	reprlib
robotparser	urllib.robotparser	urllib_robotparser
ScrolledText	tkinter.scolledtext	tkinter_scrolledtext
SimpleDialog	tkinter.simpledialog	tkinter_simpledialog
SimpleHTTPServer	http.server	SimpleHTTPServer
SimpleXMLRPCServer	xmlrpc.server	
StringIO.StringIO	io.StringIO	
SocketServer	socketserver	socketserver
test.test_support	test.support	tkinter
Tkinter	tkinter	tkinter
Tix	tkinter.tix	tkinter_tix
Tkconstants	tkinter.constants	tkinter_constants
tkColorChooser	tkinter.colorchooser	tkinter_colorchooser
tkCommonDialog	tkinter.commondialog	tkinter_commondialog
Tkdnd	tkinter.dnd	tkinter_dnd
tkFileDialog	tkinter.filedialog	tkinter_tkfiledialog
tkFont	tkinter.font	tkinter_font
tkMessageBox	tkinter.messagebox	tkinter_messagebox
tkSimpleDialog	tkinter.simpledialog	tkinter_tksimpledialog
turtle	tkinter.turtle	
UserList	collections	

Continued on next page

Table B.1 – continued from previous page

Python 2 name	Python 3 name	six name
UserString	collections	
whichdb	dbm	
_winreg	winreg	winreg
xmlrpclib	xmlrpc.client	

B.1.1 **urllib**, **urllib2** and **urlparse**

The three modules urllib, urllib2 and urlparse has been reorganized into three new modules, urllib.request, urllib.parse and urllib.error. Since there is no six support for these renames you have to use the try/except technique above.

Python 2 name	Moved to
urllib._urlopener	urllib.request
urllib.ContentTooShortError	urllib.error
urllib.FancyURLOpener	urllib.request
urllib.pathname2url	urllib.request
urllib.quote	urllib.parse
urllib.quote_plus	urllib.parse
urllib.splitattr	urllib.parse
urllib.splithost	urllib.parse
urllib.splitnport	urllib.parse
urllib.splitpasswd	urllib.parse
urllib.splitport	urllib.parse
urllib.splitquery	urllib.parse
urllib.splittag	urllib.parse
urllib.splittype	urllib.parse
urllib.splituser	urllib.parse
urllib.splitvalue	urllib.parse
urllib.unquote	urllib.parse
urllib.unquote_plus	urllib.parse
urllib.urlcleanup	urllib.request
urllib.urlencode	urllib.parse
urllib.urlopen	urllib.request
urllib.URLOpener	urllib.request
urllib.urlretrieve	urllib.request
urllib2.AbstractBasicAuthHandler	urllib.request
urllib2.AbstractDigestAuthHandler	urllib.request
urllib2.BaseHandler	urllib.request
urllib2.build_opener	urllib.request
urllib2.CacheFTPHandler	urllib.request
urllib2.FileHandler	urllib.request
urllib2.FTPHandler	urllib.request
urllib2.HTTPBasicAuthHandler	urllib.request
urllib2.HTTPCookieProcessor	urllib.request
urllib2.HTTPDefaultErrorHandler	urllib.request

Continued on next page

Table B.2 – continued from previous page

Python 2 name	Moved to
urllib2.HTTPDigestAuthHandler	urllib.request
urllib2.HTTPError	urllib.request
urllib2.HTTPHandler	urllib.request
urllib2.HTTPPasswordMgr	urllib.request
urllib2.HTTPPasswordMgrWithDefaultRealm	urllib.request
urllib2.HTTPRedirectHandler	urllib.request
urllib2.HTTPSHandler	urllib.request
urllib2.install_opener	urllib.request
urllib2.OpenerDirector	urllib.request
urllib2.ProxyBasicAuthHandler	urllib.request
urllib2.ProxyDigestAuthHandler	urllib.request
urllib2.ProxyHandler	urllib.request
urllib2.Request	urllib.request
urllib2.UnknownHandler	urllib.request
urllib2.URLError	urllib.request
urllib2.urlopen	urllib.request
urlparse.parse_qs	urllib.parse
urlparse.parse_qsl	urllib.parse
urlparse.urldefrag	urllib.parse
urlparse.urljoin	urllib.parse
urlparse.urlparse	urllib.parse
urlparse.urlsplit	urllib.parse
urlparse.urlunparse	urllib.parse
urlparse.urlunsplit	urllib.parse

B.2 Removed modules

Some of the standard library modules have been dropped. One is `UserDict`, but some of the classes have replacements that are almost, but not quite completely compatible. See *Replacing UserDict* (page 43) for more information on that.

Most of the other modules that have been dropped are modules that have long been supplanted by other, improved modules, or modules that are specific to platforms that is no longer supported. Fittingly, and exception to this rule is the `exception` module. It contains a hierarchy of exceptions, but all of them are also built-ins, so you never need to import anything from the `exception` module. It has therefore been removed from Python 3 completely.

Except for the modules specific for Solaris, IRIX and Mac OS 9, this is the list of modules that has been removed in Python 3:

Module name	Comment
audiodev	
Bastion	
bsddb185	Supplanted by bsddb3

Continued on next page

Table B.3 – continued from previous page

Module name	Comment
bsddb3	Available on the CheeseShop
Canvas	
cfmfile	
cl	
commands	
compiler	
dircache	
dl	Supplanted by ctypes
exception	See above
fpformat	
htmllib	Supplanted html.parser
ihooks	
imageop	
imputil	
linuxaudiodev	Supplanted by ossaudiodev
md5	Supplanted by hashlib
mhlib	
mimetools	Supplanted by email
MimeWriter	Supplanted by email
mimify	Supplanted by email
multifile	Supplanted by email
mutex	
new	
popen2	Supplanted by subprocess
posixfile	
pure	
rexec	
rfc822	Supplanted by email
sha	Supplanted by hashlib
sgmllib	
sre	Supplanted by re
stat	Supplanted by os.stat()
stringold	
sunaudio	
sv	
test.testall	
thread	Supplanted by threading
timing	
toaiff	
user	

B.3 Moved builtins

There are also a couple of builtin functions that have been moved to standard library modules. You handle them in a similar way, by trying to import them from the Python 3 location and if this fails you just do nothing:

```
>>> try:
...     from imp import reload
... except ImportError:
...     pass
```

The moved builtins are:

Python 2 name	Python 3 name	six name
`intern()`	`sys.intern()`	
`reduce()`	`functools.reduce()`	`reduce`
`reload()`	`imp.reload()`	`reload_module`

B.4 `string` module removals

Several functions have existed both in the `string` module and as methods on the `str` type and instances. These have now been removed from the `string` module. You can use them either on string instances or from the `str` type itself. So what in Python 2 could be written:

```
>>> import string
>>> string.upper('Dinsdale!')
'DINSDALE!'
```

Should now be written in one of the following ways:

```
>>> 'Dinsdale!'.upper()
'DINSDALE!'

>>> str.upper('Dinsdale!')
'DINSDALE!'
```

The first way of doing it is the most common one, but moving to the second way can be done with a simple search and replace.

The removed functions are `capitalize()`, `center()`, `count()`, `expandtabs()`, `find()`, `index()`, `join()`, `ljust()`, `lower()`, `lstrip()`, `maketrans()`, `replace()`, `rfind()`, `rindex()`, `rjust()`, `rsplit()`, `rstrip()`, `split()`, `strip()`, `swapcase()`, `translate()`, `upper()` and `zfill()`.

In addition the functions `atof()`, `atoi()` and `atol()` has been removed, and have been replaced with passing the string value into the `float` and `int` constructors. Since these functions have been deprecated since Python 2.0 it's highly unlikely that you actually use them.

B.5 Function and method attribute renamings

Many of the special attribute names on functions and methods were named before it was decided that Python should use the "double underscore" method for special names used by Python. They have been renamed in Python 3.

The easiest way of handling this if you are not using `2to3` is to define a variable with the attribute name depending on the Python version and using `getattr` to access the attribute. This doesn't work in the case of the renaming of `im_class`, though, so there you need a function to fetch the result:

```
>>> import sys
>>> if sys.version < '3':
...     defaults_attr = 'func_defaults'
...     get_method_class = lambda x: x.im_class
... else:
...     defaults_attr = '__defaults__'
...     get_method_class = lambda x: x.__self__.__class__
>>> class Test(object):
...     def hasdefaults(a=1, b=2):
...         pass

>>> method = Test().hasdefaults
>>> getattr(method, defaults_attr)
(1, 2)

>>> get_method_class(method)
<class 'Test'>
```

Six has defined functions to retrieve the most common attribute names:

Python 2 name	Python 3 name	`six` function
`func_closure`	`__closure__`	
`func_doc`	`__doc__`	
`func_globals`	`__globals__`	
`func_name`	`__name__`	
`func_defaults`	`__defaults__`	`get_function_defaults()`
`func_code`	`__code__`	`get_function_code()`
`func_dict`	`__dict__`	
`im_func`	`__func__`	`get_method_function()`
`im_self`	`__self__`	`get_method_self()`
`im_class`	`__self__.__class__`	

INDEX

U

W

X

Y

Z

This book guides you through the process of porting your Python 2 code to Python 3, from choosing a porting strategy to solving your distribution issues. Using plenty of code examples it takes you cross the hurdles and shows you the new Python features. The second edition is updated for Python 3.3.

- Migration strategies
- Preparing for Python 3
- Porting with 2to3
- Common migration problems
- Improving your code with modern idioms
- Supporting Python 2 and Python 3 without 2to3 conversion
- Migrating C extensions
- Extending 2to3 with your own fixers
- Language differences and workarounds
- Reorganizations and renamings

Lennart Regebro has been a full time Python developer since 2001 and has been porting to Python 3 since early 2008.

Published by Colliberty