13 Python 3 support strategies

As far as I'm aware, Python 3 is still not the default Python interpreter in any system that I'm aware of at the moment, despite having been released in December 2008 – five years ago!

The problem, as you know, is that Python 3 broke compatibility with Python 2. At the time that Python 3.0 arrived, the gap between it and Python 2.6 was so huge that people weren't even beginning to think about bridging it. Scared. Shrugging.

But then things changed: Python 2.7 back-ported a lot of features from Python 3.1, narrowing the gap. Much sanity returned through subsequent versions of Python, and I am happy to state that it is now possible to support both Python 2.7 and Python 3.3… almost without difficulty!

There is official documentation on porting applications, but I wouldn't recommend following it to the letter. It talks a lot about the *2to3* tool – which converts Python 2 code to Python 3 – and contains proposals like starting a special Python 3 branch for your project.

In my opinion, this is terrible advice nowadays. It may have been the most appropriate advice a few years ago, but considering the current state of "compatibility" between Python 2.7 and Python 3.3, it's better to forget about this approach.

Note

Note that a *3to2* tool also exists – but for the same reason given above, I wouldn't encourage its use.

Firstly, *2to3* doesn't do always the right thing – it's not magic. It only deals with syntax changes, which covers a lot; but it doesn't maintain backward compatibility with Python 2 – and in any case, you'll have to handle semantic changes manually. Secondly, running *2to3* is damn slow; and for this reason it's unlikely to be a good long-term solution. Some guides even suggest running it at *setup.py* time, which is somewhat hazardous.

Some documentation recommends using different project branches to support Python 2 and Python 3. Experience shows that this can be terrible to manage, and that users will get confused about which version they should use. Even worse, you will get confused when they start submitting bug reports without explicitly stating which branch they are using.

A better method is to use a single code base that is both Python 2 and Python 3 compatible. This is on what we put our effort on with OpenStack.

In the end, the only way to be sure that your code works under both Python versions is to have unit testing. Without unit testing, it is **impossible** to know if your code will work in both contexts and across versions. If you do not have any test in your application [1] the first thing to do is to increase your code coverage dramatically; you may want to jump to Chapter 6 right ahead.

Tox is a great tool for automating tests run against multiple Python versions, and we'll talk about it in Section 6.7.

Once you have unit tests and tox set up, it's easy enough to run your tests against both Python versions using:

[1] I have heard that such projects exist.

```
tox -e py27,py33
```

See what's broken, fix it, and launch *tox* again. Repeat until all tests pass. If you're doing it correctly, the number of errors will decrease slowly but steadily, to the point where all of your code base will be fully Python 2 and 3 compatible.

If you have a C module written for Python that you would like to port, I'm sorry to inform you that there's not much to say – other than to tell you to read the documentation and port your code. It may be a useful option to rewrite using cffi if possible.

In the following sections I will discuss some points you will encounter while porting between Python versions. I will assume that you already have a Python 2 code base. While most of what follows could in theory also be applied to the porting of a Python 3 project to Python 2, I have never personally encountered such a case.

13.1 Language and standard library

The language hasn't changed radically; I'm sure you've already taken a look. This book won't cover the entire list of changes – it would be much too boring, and in any case can be found online. The book Porting to Python 3 gives a pretty good overview of what you may need to change in order to support Python 3.

If you haven't yet taken a look at the language changes made in Python 3, I invite you to do so. It's a great language, with a lot less corner cases, and much cleaner interfaces on various object bases. You'll love Python 3.

But it raises strong compatibility problems. The syntax changes to some statements (e.g. exception catching) have removed old Python version compatibilities, and they can be a pain to tackle if you used them. The hacking tool that we'll discuss in section Section 1.4 can help you to fix these incompatible usages, and stop you from adding more.

When supporting multiple versions of Python, you shouldn't try to support anything older than 2.6 and 3.3 at the same time. Python 2.6 is the first version which has enough compatibility with Python 3 to be easy enough to port forward.

The changes that might impact you the most are in the area of string handling. In Python 3 what was called **unicode** is now **str**. That means that every string is Unicode – i.e. that *u'foobar'* [2] and *'foobar'* mean the same thing.

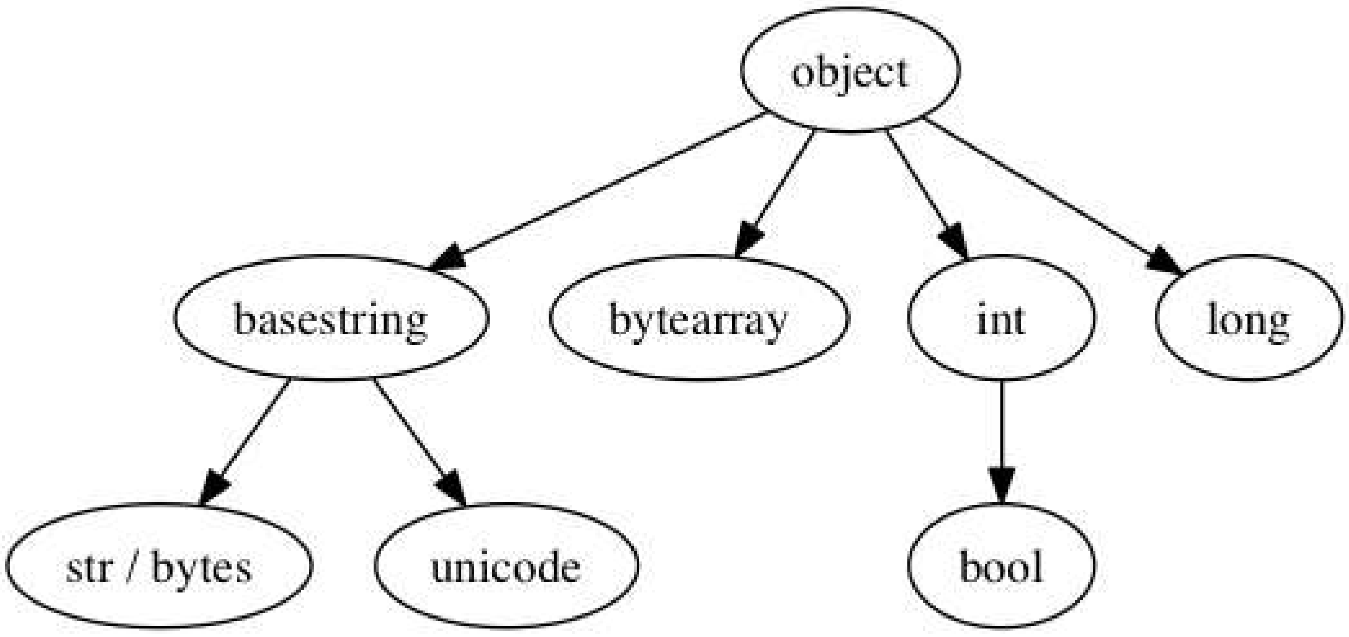

Figure 13.1: Python 2 base classes

[2] The *u* prefix was removed in Python 3.0 but reintroduced in Python 3.3 – see PEP 414

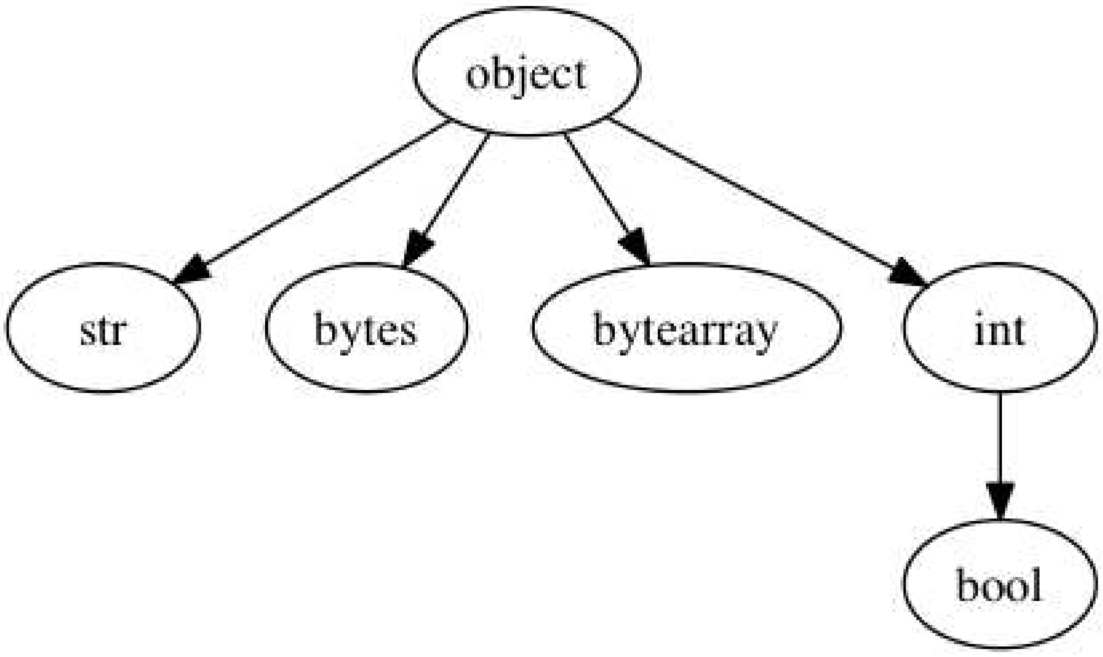

Figure 13.2: Python 3 base classes

Classes implementing *unicode* should rename that function to *str*, since the former isn't used anymore; you can automate this with a class decorator along these lines:

```python
# -*- encoding: utf-8 -*-
import six

# This backports your Python 3 __str__ for Python 2
def unicode_compat(klass):
    if not six.PY3:
        klass.__unicode__ = klass.__str__
        klass.__str__ = lambda self: self.__unicode__().encode('utf-8')
    return klass


@unicode_compat
class Square(object):
    def __str__(self):
        return u"■ " + str(id(self))
```

That way you implement just one method for all Python versions returning Unicode, and the decorator handles the compatibility issue.

Another trick that can be handy when dealing with Python and Unicode is to use the *unicode_literals* function, which is available starting with Python 2.6 [3].

```
>>> 'foobar'
'foobar'
>>> from __future__ import unicode_literals
>>> 'foobar'
u'foobar'
```

Various functions no longer return lists, instead returning iterable objects (such as *range*); in addition, dictionary methods like *keys* or *items* now return iterable objects, and functions like *iterkeys* and *iteritems* have been removed. This is a big change, but *six* (discussed in Section 13.3) can help you with handling it.

Obviously, the standard library has evolved between Python 2 and Python 3, but that shouldn't be a huge concern. Some modules have been renamed or moved, but in the end the result is a clearer layout. There's no official listing that I'm aware of, but you can find a pretty good list here, or use a search engine.

The six module, which we will discuss in Section 13.3, will also help a lot when trying to maintain compatibility between Python 2 & 3.

13.2 External libraries

Your first enemies are the external libraries you depend on. If you read my advice in Section 2.3 and followed my check-list, you won't have a problem here – since

[3] Another reason not to support older versions?

that check-list included a Python 3 support requirement. However, you may have started a project earlier and have already made the mistake.

Unfortunately there isn't any magic trick than can resolve the problem. Luckily, if you followed my other advice, you isolated this library enough that it is not spread across your whole code base; so you can think about replacing it. Indeed, this may be your best move if the library does not show a strong possibility of supporting Python 3. However, small and medium-sized libraries might be more easily ported to Python 3 than big frameworks, so you may want to cut your teeth on them.

When looking for packages on PyPI, you can check for the trove classifiers *"Programming Language :: Python :: 2"* and *"Programming Language :: Python :: 3"*, which indicate which version of Python the package supports. However, be careful that these may not be up to date.

One of the external library choices made at the beginning of the OpenStack project was eventlet, a concurrent networking library. It has no support for Python 3, and still tries to support Python 2.5 – which, as you imagine, does not facilitate any transition. This choice was made a long time ago in OpenStack, before any kind of checks for Python 3 compatibility were done; and we already know that this module is going a big issue in the months ahead. As of yet, we have no concrete plan on how to fix it.

Don't make the same mistake!

13.3 Using six

As we have seen, Python 3 breaks compatibility with earlier versions and shifts things around. However, the basics of the language haven't changed, so it is possible to have a sort of transition layer; a module that can implement forward and backward compatibility – a bridge between Python 2 and Python 3.

This module exists, and it's called **six** – because two times three equals six.

The first thing that *six* provides is the **six.PY3** variable. This is a boolean which indicates whether we are running Python 3 or not. This is the pivot variable for any of your code base that has two versions, one for Python 2 and one for Python 3. However, be careful not to abuse it; scattering your code base with *if six.PY3* is going to be difficult to work with later.

As we discussed in Section 8.1, which concerned generators, Python 3 has a great feature whereby iterable objects are returned instead of lists. That means that methods like *dict.iteritems* are gone, and that *dict.items* returns an iterator rather than a list. Obviously this can break your code. *six* provides *six.iteritems* for such cases, so that all you have to do is to replace the following code:

```
for k, v in mydict.iteritems():
    print(k, v)
```

with:

```
import six


for k, v in six.iteritems(mydict):
    print(k, v)
```

And voilà, Python 3 compliance achieved in a snap! *six* provides a lot of similar helper functions that can increase compatibility across Python versions.

The **raise** syntax also changed in Python 3 [4], so re-raising exceptions should be done using *six.reraise*.

If you are using metaclasses, Python 3 has also changed this completely. Six has a nice trick for handling the transition – for example, if you are using the **abc** abstract base classes metaclass, here's how you would use *six*:

```
import abc
from six import with_metaclass
```

[4] It now only accepts one argument, an exception.

```python
class MyClass(with_metaclass(abc.ABCMeta, object)):
    pass
```

One cannot discuss Python 3 without touching on the string and unicode mess that it solved. In Python 2, the basic type for string is `str` which can handle only basic ASCII strings, and the type `unicode`, added later, handles real string of text. In Python 3, the basic type is still `str`, but it shares the properties of the Python 2 `unicode` class and can handle advanced encodings. The `bytes` type replaces the `str` type for handling basic characters stream.

six provides a nice set of functions and constants to handle the transition, such as *six.u* and *six.string_types*. The same compatibility is provided for integers, with *six.integer_types* that will handle the *long* type that has been removed from Python 3.

As discussed in Section 13.1, some modules have moved, and *six* provides a nice module called *six.moves* that handles a lot of these moves transparently.

For example, the *ConfigParser* module in Python 3 has been renamed to *configparser*. Code using *ConfigParser* under Python 2:

```python
from ConfigParser import ConfigParser

conf = ConfigParser()
```

can be ported and made compatible with both major Python versions:

```python
from six.moves.configparser import ConfigParser

conf = ConfigParser()
```

Tip

It is also possible to add your own moves via *six.add_move* to handle other transitions.

The *six* library might not be enough or cover all your use case. In this case, building a compatibility module encapsulating *six* itself might be worth it. By isolating the this in one particular module, you are assuring that you'll be able to enhance it for future version of Python, or dispose (part of) it when you'll want to stop supporting a particular version of Python. Also note that *six* is open source and that you can contribute to it rather than maintaining your own hacks.

The last thing I'll mention, is the modernize module. It's a thin wrapper around *2to3* that "modernizes" code by porting to Python 3; but rather than convert the syntax to Python 3 code only, it uses the *six* module. It's a better choice than the standard *2to3* tool, and get your port off to a strong start by carrying out most of the grunt work for you. It's worth a shot.