# A Pragmatic Guide to Python 3 Adoption

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009) and *Python Cookbook* (3rd Edition, O'Reilly Media, 2013). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

Believe it or not, it's been more than five years since Python 3 was unleashed on the world. At the time of release, common consensus among Python core developers was that it would probably take about five years for there to be any significant adoption of Python 3. Now that the time has passed, usage of Python 3 still remains low. Does the continued dominance of Python 2 represent a failure on the part of Python 3? Should porting existing code to Python 3 be a priority for anyone? Does the slow adoption of Python 3 reflect a failure on the part of the Python developers or community? Is it something that you should be worried about?

There are no clear answers to any of these questions other than to say that "it's complicated." To be sure, almost any discussion of Python 3 on the Internet can quickly turn into a fiery debate of finger pointing and whining. Although, to be fair, much of that is coming from library writers who are trying to make their code work on Python 2 and 3 at the same time—a very different problem than that faced by most users. In this article, I'm going to try and steer clear of that and have a pragmatic discussion of how working programmers might approach the whole Python 3 puzzle.

This article is primary for those who use Python to get actual work done. In other words, I'm not talking about library and framework authors—if that applies to you and you're still not supporting Python 3, stop sitting on the sidelines and get on with it already. No, this article is for everyone else who simply uses Python and would like to keep using it after the Python 3 transition.

## Python 3 Background

If you haven't been following Python 3 very closely, it helps to review a bit of history. To my best recollection, the idea of "Python 3" originates back to the year 2000, if not earlier. At that time, it was merely known as "Python 3000"—a hypothetical future version of Python (named in honor of Mystery Science Theater 3000) where all of the really hard bugs, design faults, and pie-in-the-sky ideas would be addressed someday. It was a release reserved for language changes that couldn't be made without also breaking the entire universe of existing code. It was a stock answer that Guido van Rossum could give in a conference talk (e.g., "I'll eventually fix that problem in Python 3000").

Work on an actual Python 3000 version didn't really begin until much later—perhaps around 2005. This culminated in the eventual release of Python 3.0 in December 2008. A major aspect of Python 3 is that backward-incompatible changes were made to the core language. By far, the most visible change is the breakage of the lowly print statement, leading first-time Python 3 users to type a session similar to this:

```
>>> print "hello world"
  File "<stdin>", line 1
        print "hello world"
                       ^
SyntaxError: invalid syntax
>>>
```

This is easy to fix—simply change the print statement to `print("hello world")`. However, the fact that even the easiest example breaks causes some developers to grumble and come away with a bad first impression. In reality, the internal changes of Python 3 run much deeper than this, but you're not likely to encounter them as immediately as with `print()`. The purpose of this article isn't to dwell on Python 3 features, however—they are widely published [1] and I've written about them before [2].

## Some Assumptions

If you're using Python to solve day-to-day problems, I think there are a few underlying assumptions about software development that might apply to your work. First, it's somewhat unlikely that you're concerned about supporting every conceivable Python version. For example, I have Python 2.7 installed on my machine and I use it for a lot of projects. Although I could enter a time machine and install Python 2.3 on my system to see if my code still works with it, I honestly don't care. Seriously, why would I spend my time worrying about something like that? Even at large companies, I find that there is often an "official" Python version that almost everyone is using. It might not always be the latest version, but it's some specific version of the language. People aren't wasting their time fooling around with different interpreter versions.

I think a similar argument can be made about the choice between Python 2 and 3. If you've made a conscious choice to work on a project in Python 3, there is really no good reason to also worry about Python 2. Again, as an application programmer, why would I do that? If Python 3 works, I'm going to stick with it and use it. I've got better things to be doing with my time than trying to wrap my brain around different language versions. (To reiterate, this is not directed at grumpy library writers.)

Related to both of the above points, I also don't think many application programmers want to write code that involves weird hacks and non-idiomatic techniques—specifically, hacks aimed at making code work on two incompatible versions of the Python language. For example, if I'm trying to use Python to solve some pressing problem, I'm mostly just concerned with that problem. I want my code to be nice and readable—like the code you see in books and tutorials. I want to be able to understand my own code when I come back to read it six months later. I don't want to be sitting in a code review trying to explain some elaborate hacky workaround to a theoretical problem involving Python 2/3 compatibility.

Last, but not least, most good programmers are motivated by a certain sense of laziness. That is, if the code is working fine already, there has to be a pretty compelling reason to want to "fix" it. In my experience, porting a code base to a new language version is just not that compelling. It usually involves a lot of grunt work and time—something that is often in short supply. Laziness also has a dark side involving testing. You know how you hacked up that magic Python data processing script on a Friday afternoon three years ago? Did you write any unit tests for it? Probably not. Yes, this can be a problem too.

So, with the understanding that you probably just want to use a single version of Python, you don't want to write a bunch of weird hacks, you may not have tests, and you're already overworked, let's jump further into the Python 3 fray.

## Starting a New Project? Try Python 3

If you're starting a brand new project, there is no reason not to try Python 3 at this point. In fact, it doesn't even have to be too significant. For example, if you find yourself needing to write a few one-off scripts, this is a perfect chance to give Python 3 a whirl without worrying if it will work in a more mission critical setting.

Python 3 can be easily installed side-by-side with any existing Python 2 installation, and it's okay for both versions to coexist on your machine. Typically, if you install Python 3 on your system, the `python` command will run Python 2 and the `python3` command will run Python 3. Similarly, if you've installed additional tools such as a package manager (e.g., setuptools, pip, etc.), you'll find that the Python 3 version includes "3" in the name. For example, `pip3`.

If you rely on third-party libraries, you may be pleasantly surprised at what packages currently work with Python 3. Most popular packages now provide some kind of Python 3 support. Although there are still some holdouts, it's worth your time to try the experiment of installing the packages you need to see if they work. From personal experience over the last couple of years, I've encountered very few packages that don't work with Python 3.

Once you've accepted the fact that you're going to use Python 3 for your new code, the only real obstacle to starting is coming to terms with the new `print()` function. Yes, you're going to screw that up a few hundred times because you're used to typing it as a statement out of habit. However, after a day of coding, adding the parentheses will become old hat. Next thing you know, you're a Python 3 programmer.

## What To Do with Your Existing Code?

Knowing what to do with existing code in a Python 3 universe is a bit more delicate. For example, is migrating your code something that you should worry about right now? If you don't

migrate, will your existing programs be left behind in the dustbin of coding history? If you take the plunge, will all your time be consumed by fixing bugs due to changes in Python 3 semantics? Are the third-party libraries used by your application available in Python 3?

These are all legitimate concerns. Thus, let's explore some concrete steps you can take with the assumption that migrating your code to Python 3 is something you might consider eventually if it's not too painful, maybe.

## Do Nothing!

Yes, you heard that right. If your programs currently work with Python 2 and you don't need any of the new functionality that Python 3 provides, there's little harm in doing nothing for now. There's often a lot of pragmatic wisdom in the old adage of "if it ain't broke, don't fix it." In fact, I would go one step further and suggest that you *NOT* try to port existing code to Python 3 unless you've first written a few small programs with Python 3 from scratch.

Currently, Python 2 is considered "end of life" with version 2.7. However, this doesn't mean that 2.7 will be unmaintained or unsupported. It simply means that changes, if any, are reserved for critical bug fixes, security patches, and similar activity. Starting in 2015, changes to Python 2.7 will be reserved to security-only fixes. Beyond that, it is expected that Python 2.7 will enter an extended maintenance mode that might last as long as another decade (yes, until the year 2025). Although it's a little hard to predict anything in technology that remote, it seems safe to say that Python 2.7 isn't going away anytime soon. Thus, it's perfectly fine to sit back and take it slow for a while.

This long-term maintenance may, in fact, have some upsides. For one, Python 2.7 is a very capable release with a wide variety of useful features and library support. Over time, it seems clear that Python 2.7 will simply become the de facto version of Python 2 found on most machines and distributions. Thus, if you need to worry about deploying and maintaining your current code base, you'll most likely converge upon only one Python version that you need to worry about. It's not unlike the fact that real programmers are still coding in Fortran 77. It will all be fine.

## Start Writing Code in a Modern Style

Even if you're still using Python 2, there are certain small steps you can take to start modernizing your code now. For example, make sure you're always using new-style classes by inheriting from `object`:

```
class Point(object):
    def __init__(self, x, y):
            self.x = x
            self.y = y
```

Similarly, make sure you use the modern style of exception handling with the "as" keyword:

```
try:
    x = int(val)
except ValueError as exc: # Not: except ValueError, exc:
...
```

Make sure you use the more modern approaches to certain built-in operations. For example, sorting data using key functions instead of the older compare functions:

```
names = ['paula', 'Dave', 'Thomas', 'lewis']
names.sort(lambda n1, n2: cmp(n1.upper(), n2.upper()))    # OLD
names.sort(key=lambda n: n.upper())                       # NEW
```

Make sure you're using proper file modes when performing I/O. For example, using mode `'t'` for text and mode `'b'` for binary:

```
f = open('sometext.txt', 'rt')
g = open('somebin.bin', 'rb')
```

These aren't major changes, but a lot of little details like this come into play if you're ever going to make the jump to Python 3 later on. Plus, they are things that you can do now without breaking your existing code on Python 2.

## Embrace the New Printing

As noted earlier, in Python 3, the print statement turns into a function:

```
>>> print('hello', 'world')
hello world
>>>
```

You can turn this feature on in Python 2 by including the following statement at the top of each file that uses `print()` as a function:

```
from __future__ import print_function
```

Although it's not much of a change, mistakes with `print` will almost certainly be the most annoying thing encountered if you switch Python versions. It's not that the new print function is any harder to type or work with—it's just that you're not used to typing it. As such, you'll repeatedly make mistakes with it for some time. In my case, I even found myself repeatedly typing `printf()` in my programs as some kind of muscle-memory holdover from C programming.

## Run Code with the -3 Option

Python 2.7 has a command line switch `-3` that can warn you about more serious and subtle matters of Python 3 compatibility. If you enable it, you'll get warning messages about your usage of incompatible features. For example:

```
bash % python2.7 -3
>>> names = ['Paula', 'Dave', 'Thomas', 'lewis']
>>> names.sort(lambda n1, n2: cmp(n1.upper(), n2.upper()))
__main__:1: DeprecationWarning: the cmp argument is not
supported in 3.x
>>>
```

With this option, you can take steps to find an alternative implementation that eliminates the warning. Chances are, it will improve the quality of your Python 2 code, so there are really no downsides.

## Future Built-ins

A number of built-in functions change their behavior in Python 3. For example, `zip()` returns an iterator instead of a list. You can include the following statement in your program to turn on some of these features:

```
from future_builtins import *
```

If your program still works afterwards, there's a pretty good chance it will continue to work in Python 3. So it's usually a useful idea to try this experiment and see if anything breaks.

## The Unicode Apocalypse

By far, the hardest problem in modernizing code for Python 3 concerns Unicode [3]. In Python 3, all strings are Unicode. Moreover, automatic conversions between Unicode and byte strings are strictly forbidden. For example:

```
>>> # Python 2 (works)
>>> 'Hello' + u'World'
u'HelloWorld'
>>>

>>> # Python 3 (fails)
>>> b'Hello' + u'World'
Traceback (most recent call last):
   File "<stdin>", line 1, in
TypeError: can't concat bytes to str
>>>
```

Python 2 programs are often extremely sloppy in their treatment of Unicode and bytes, interchanging them freely. Even if you don't think that you're using Unicode, it still might show up in your program. For example, if you're working with databases, JSON, XML, or anything else that's similar, Unicode almost always creeps into your program.

To be completely correct about treatment of Unicode, you need to make strict use of the `encode()` and `decode()` methods in any conversions between bytes and Unicode. For example:

```
>>> 'Hello'.decode('utf-8') + u'World'    # Result is Unicode
u'HelloWorld'
>>> 'Hello' + u'World'.encode('utf-8')    # Result is bytes
'HelloWorld'
>>>
```

However, it's really a bit more nuanced than this. If you know that you're working with proper text, you can probably ignore all of these explicit conversions and just let Python 2 implicitly convert as it does now—your code will work fine when ported to Python 3. It's the case in which you know that you're working with byte-oriented non-text data that things get tricky (e.g., images, videos, network protocols, and so forth).

In particular, you need to be wary of any "text" operation being applied to byte data. For example, suppose you had some code like this:

```
f = open('data.bin', 'rb')    # File in binary mode
data = f.read(32)             # Read some data
parts = data.split(',')       # Split into parts
```

Here, the problem concerns the `split()` operation. Is it splitting on a text string or is it splitting on a byte string? If you try the above example in Python 2 it works, but if you try it in Python 3 it crashes. The reason it crashes is that the `data.split(',')` operation is mixing bytes and Unicode together. You would either need to change it to bytes:

```
parts = data.split(b',')
```

or you would need to decode the data into text:

```
parts = data.decode('utf-8').split(',')
```

Either way, it requires careful attention on your part. In addition to core operations, you also must focus your attention on the edges of your program and, in particular, on its use of I/O. If you are performing any kind of operation on files or the network, you need to pay careful attention to the distinction between bytes and Unicode. For example, if you're reading from a network socket, that data is always going to arrive as uninterpreted bytes. To convert it to text, you need to explicitly decode it according to a known encoding. For example:

```
data = sock.recv(8192)
text = data.decode('ascii')

import urllib
u = urllib.urlopen('http://www.python.org')
text = u.read().decode('utf-8')
```

Likewise, if you're writing text out to the network, you need to encode it:

```
text = 'Hello World'
sock.send(text.encode('ascii'))
```

Again, Python 2 is very sloppy in its treatment of bytes—you can write a lot of code that never performs these steps. However, if you move that code to Python 3, you'll find that it breaks.

Even if you don't port, resolving potential problems with Unicode is often beneficial even in a Python 2 codebase. At the very least, you'll find yourself resolving a lot of mysterious `UnicodeError` exceptions. Your code will probably be a bit more reliable. So it's a good idea.

## Taking the Plunge

Assuming that you've taken all of these steps of modernizing code, paying careful attention to Unicode and I/O, adopting the `print()` function, and so forth, you might actually be ready to attempt a Python 3 port, maybe.

Keep in mind that there are still minor things that you might need to fix. For example, certain library modules get renamed and the behavior of certain built-in operations may vary slightly. However, you can try running your program through the 2to3 tool and see what happens. If you haven't used 2to3, it simply identifies the parts of your code that will have to be modified to work on Python 3. You can either use its output as a guide for making the changes yourself, or you can instruct it to automatically rewrite your code for you. If you're lucky, adapting your code to Python 3 may be much less work than you thought.

## What About Compatibility Libraries?

If you do a bit a research, you might come across some compatibility libraries that aim to make code compatible with both Python 2 and 3 (e.g., "six," "python-modernize," etc.). As an application programmer, I'm somewhat reluctant to recommend the use of such libraries. In part, this is because they sometimes translate code into a form that is not at all idiomatic or easy to understand. They also might introduce new library dependencies. For library writers who are trying to support a wide range of Python versions, such tools can be helpful. However, if you're just trying to use Python as a normal programmer, it's often best to just keep your code simple. It's okay to write code that only works with one Python version.

### *References*

[1] Nick Coghlan's "Python 3 Q&A" (http://ncoghlan-devs-python-notes.readthedocs.org/en/latest/python3/questions_and_answers.html) is a great read concerning the status of Python 3 along with its goals.

[2] David Beazley, "Three Years of Python 3," *;login:*, vol. 37, no. 1, February 2012: beazley12-02_0.pdf.

[3] For the purposes of modernizing code, I recommend Ned Batchelder's "Pragmatic Unicode" presentation (http://nedbatchelder.com/blog/201203/pragmatic_unicode.html) for details on sorting out Unicode issues in Python 2 and preparing your mind for work in Python 3.