Python 2-to-3 Migration Guide

Introduction

Python[1] 3 will be 10 years old in December of this year (2018). It has been mature and robust for a while now. Yet, because of inertia, Python 2 is still alive and well in many organizations.

Python 2 will no longer be supported past 2019[2] which will push some organizations to finally switch to Python 3. Is yours among them, perhaps? If so, **this document will help you to build a plan to migrate your organization's codebase to Python 3**.

This document is intended for two audiences. First, it's intended for those whose job it is to develop the **plan** for the migration to Python 3. Second, it's for those who will **perform** the migration. (In some organizations, those groups are one and the same.) The subject is first discussed at a high level, with technical details following.

This guide assumes you're currently using Python 2.7, and making a **clean break** with Python 2 instead of supporting Python 2.x and 3.x simultaneously.

The Plan for the Plan

Since Python 3 contains backwards-incompatible changes, migrating your code is like **stepping through a one-way door**. (Again, this document assumes you have no interest in supporting Python 2 and 3 simultaneously.) During the migration, either all other work will stop, or you'll maintain two syntactically incompatibility codebases. Both options are undesirable, so there's incentive to complete the transition **quickly** and **confidently**.

This document shows you how to construct a plan that will maximize your odds of an orderly, quick, and smooth transition. The key to this approach is advance preparation, and **there's a lot you can do right now** while you're still using Python 2. Tackle the tasks outlined below as you have time, and you'll be perfectly positioned to migrate when you're ready.

Acronyms are popular, so let's aim for a transition that's like a **BOSS – Brief**, **Orderly**, **Smooth**, and **Satisfying**. It's easiest to cover the individual BOSS words in reverse order.

1 "Python" and the Python logos are trademarks or registered trademarks of the Python Software Foundation.
2 https://pythonclock.org

Python 2-to-3 Migration Guide

1 – Decide Which Python 3 to Target

Migrating to Python 3 can feel like a lot of work that's forced on you, but it can be quite *satisfying* once you're done because of what you gain.

It's a good idea to migrate to **the most recent Python 3 available**. (The early versions of Python 3 – especially those before 3.3 – were still smoothing off rough edges, so definitely avoid those.) **Your migration costs are the same regardless of which Python 3 version you target, but the benefits are not**. Compared to older versions of Python 3, newer versions are more efficient, offer additional features (like coroutines and type hinting), and buy you more time before you feel the need to upgrade again.

If you're on a platform where the standard Python 3 version is relatively old (e.g. a conservative Linux distribution like Red Hat® Enterprise Linux®), consider how much it extra effort you'd have to invest to use the latest Python 3. Even if there are short term costs to installing a Python that's not supplied by the platform, the odds are that you'll recoup your investment quickly.

Python 3.7 was released in June 2018[3].

2 – Identify Test Gaps

Good test coverage ensures a *smooth* transition.

Tests identify problems before they make their way into production. If you don't already have complete coverage (hardly anyone does) and you don't have time to make it perfect (hardly anyone has that, either), **focus on areas that are most likely to be affected by the transition**.

Python 3's new (and improved) approach to Unicode is its most prominent difference from Python 2. You might run into a few places where Python 3 rejects some string manipulation that Python 2 accepts. **Testing areas of your code that handle text should be a priority.**

The technical notes in "Tests for Text Handling" and "Other Tests – Division" include some details and specific suggestions.

3 https://www.python.org/downloads/release/python-370/

3 – Review Dependencies

Identifying and upgrading dependencies that won't work under Python 3 can be done at your leisure before the migration to ensure an *orderly* transition.

Almost all 3rd party libraries have been ported to Python 3 by now. Many are compatible with both Python 2 and 3 simultaneously, so you might be able to use most of your existing dependencies (or a newer version of them) both before and after the migration.

If you've been relying on a library that has become abandonware, you might have to do some hunting to find a Python 3-compatible replacement. That can take some time, but the good news is that you don't have to wait to start that process.

The technical notes in "Identifying Python 3-Incompatible Dependencies" include some suggestions on how to identify dependencies that need attention.

4 – Start the Future Now

By making your code as Python 3-like as possible now, you minimize the changes you'll need to make at transition time, thus ensuring a *brief* transition.

Python's designers wisely made many of Python 3's features available in Python 2.7. **You can make many changes now** that will reduce your work at transition time, **without giving up Python 2**.

The technical notes contain extensive advice on how to make as many of these changes as possible in advance.

5 – Work in Parallel

The tasks above are independent of one another. If your team has the capacity, you can work on all of them at once.

6 – Ready, Set, Migrate!

Once you've completed all of the tasks above, the actual migration of your code should be anti-climactic, which is just how we want it to be. **Boring is good.**

The technical notes describe code changes necessary at this stage. Don't forget to allow time for changes specific to your organization, such as deploying new dependencies and changing the default Python version in your environment.

When you're done with all of these changes, deploy to a staging environment (if you have one) and test. If all goes well, you're ready to **enjoy Python 3 in production**.

Python 2-to-3 Migration Guide

Technical Notes

These technical notes are intended for those who will perform the actual migration.

If you want to refresh yourself on the main changes in Python 3, the Python documentation has a good summary[4].

1 – Tests for Text Handling

You might find that Python 3 rejects text handling that Python 2 found perfectly acceptable. **This is good. Python 3 clearly distinguishes between bytes and strings; Python 2 doesn't.** Python 3 is warning you about implicit byte/string conversions in your code. For instance, this is acceptable in Python 2 –

```
>>> '' + b''
''
```

It's not acceptable in Python 3 –

```
>>> '' + b''
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't concat bytes to str
```

Once you're familiar with it, Python 3's more restrictive behavior is less likely to surprise you. Under Python 2, combining bytes and strings can fail if the example isn't as simple as the one above. For instance, in Python 2 –

```
>>> u'' + b''     # Python promotes the byte string (str) to unicode
u''
>>> u'' + b'año'  # Python tries but fails to promote the byte string
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 0: ordinal not
in range(128)
```

4 https://docs.python.org/3.0/whatsnew/3.0.html

It's easier to test byte/string handling under Python 3 because it's concatenation rules are *type* dependent rather than *data* dependent. Python 3 consistently raises an error when combining bytes and strings, whereas Python 2 might or might not, depending on the data in the byte string.

Byte strings most often show up in relation to I/O, for instance when reading data from a file or an HTTP response body. Make sure these areas of your code don't go untested, and Python 3 will give you loud and clear messages if you're doing anything it doesn't like.

2 – Other Tests – Division

Python 3 changed the default behavior of the division operator[5]. Under Python 2, "floor division" is the default when both arguments are integers, and "true division" takes over if at least one is a float –

```
>>> 4 / 3
1
>>> 4 / 3.0
1.3333333333333333
```

Under Python 3, the behavior is consistent regardless of the argument type –

```
>>> 4 / 3
1.3333333333333333
>>> 4 / 3.0
1.3333333333333333
```

The behavior of the "floor division" operator remains consistent in both Python 2 and 3 –

```
>>> 4 // 3
1
```

This change is subtle. It can easily sneak past a unit test if the test wasn't constructed with this change in mind, in particular because Python considers integers and floats equal if their values are the same. (That is, `int(1) == float(1)`, and `1 == 1.0`.)

For instance, a binning algorithm[6] that's expected to always return an integral index value might rely on the behavior of Python 2's division operator to generate that integer return value. Under Python 3 that behavior will change, but if your test's inputs to the algorithm

5 https://www.python.org/dev/peps/pep-0238/
6 https://en.wikipedia.org/wiki/Data_binning

result in a non-fractional value like 3.0, your unit test will probably accept the value without noticing that the type is unexpected. Python won't complain until you try to use the value –

```
>>> my_bins[index]   # index has been set to 3.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not float
```

Unit tests and other code that uses division deserves review to ensure it's robust.

3 – Identifying Python 3-Incompatible Dependencies

The tool `caniusepython3`[7] is **the easiest way to identify potential Python 3 incompatibilities in 3rd party libraries**. If you're lucky, you won't find any incompatibilities. But if some turn up, keep in mind that `caniusepython3` trusts (relies on) package metadata. Most package authors populate this metadata and keep it current, but not all of them do.

If a package doesn't advertise Python 3 compatibility in its metadata, `caniusepython3` assumes it isn't compatible. That might be true, or it might be that the package supports Python 3 but the metadata isn't current (or doesn't contain any descriptive qualifiers at all).

To summarize, `caniusepython3` will list all of your questionable dependencies. Review that list, find the dependencies that truly need attention, and decide whether you need a version upgrade or another library entirely.

If you find that you're using an abandoned library, the good news is that you're likely not alone. Someone may have either written a replacement or forked the abandoned library to produce a Python 3 version. If not, you might need to refactor your code to use a different library, or take up the Python 3 port yourself.

4 – Start the Future Now (for New Code)

You can enable some Python 3 syntax in your Python 2 code via imports created for just this purpose.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals
```

7 https://pypi.python.org/pypi/caniusepython3

There is only benefit to adding these imports to new code, so I suggest adopting them immediately. Every file in which they're present will be easier to port to Python 3 when the time comes.

They can break code if added to existing modules because they change behavior, sometimes in very subtle ways. Add them to existing code only if you're prepared to do adequate testing before committing.

The potential for subtly breaking existing code is mostly limited to the `division` and `unicode_literals` imports. Errors resulting from the `print_function` import will be loud and fatal which makes finding and correcting problems easy. However, changing the syntax of `print` is easy to deal with automatically, so although adding this import to existing code is easy, it doesn't buy you much either. (See "Review Your Code With 2to3" below for details on how to fix this automatically.)

The same caveats that apply to the `print_function` import apply to the `absolute_import` import, although in some very rare edge cases it can result in subtle changes.

5 – Start the Future Now (for Existing Code)

Python ships with a very useful tool called `2to3` that automates the process of converting your Python 2 code to Python 3 syntax[8]. Used carefully, this tool can be a huge help towards making the awkward transition *brief*. The section below explains how to get the most out of this very useful tool.

6 – Review Your Code With 2to3

Adapting your code is usually the biggest part of any Python 2-to-3 migration. Python provides the tool `2to3` to help automate this process. It can quickly give you an idea of how much porting work you'll have to do.

`2to3` runs a set of approximately 50 named "fixers", each of which handles a specific aspect of making Python 2 code Python 3-compatible. A very simple example is the "`ne`" (not-equals) fixer. Python 2 recognizes two different operators as not-equals: `<>` and `!=`. Python 3 recognizes only the latter; the former raises a `SyntaxError`. The "`ne`" fixer[9] changes `<>` to `!=`.

8 https://docs.python.org/2/library/2to3.html
9 Not to be confused with The Knights Who Say "Ni".

Most fixers are more complex, and some of their fixes deserve review.

The changes suggested by `2to3`'s fixers fall into 3 categories –

1. **Now**: These are Python 2-compatible changes that can be made immediately with no ill effects.
2. **Now, With Review**: These are also Python 2-compatible changes, but with more ambiguous goals, so they might be suboptimal, or, in some unusual cases, incorrect.
3. **Later**: These are changes that need to wait until you're ready to abandon Python 2 for good.

Depending on the size of your project, you might want to run these fixers one at a time, in small groups, or all at once.

6.1 – Now Fixers

These `2to3` fixers can be applied immediately without breaking Python 2.7 compatibility. For organizational purposes, you might want to apply each fixer's changes in a separate commit.

- `apply`
- `asserts`
- `except`
- `execfile`
- `exitfunc`
- `funcattrs`
- `has_key`
- `input`
- `isinstance`
- `methodattrs`
- `ne`
- `next`
- `nonzero`
- `numliterals`
- `paren`
- `reduce`
- `repr`
- `set_literal`
- `standarderror`
- `sys_exc`
- `tuple_params`
- `types`
- `ws_comma`
- `xreadlines`

6.1.1 – The Now or Never Fixer

Most organizations won't need the lone fixer in this special category. It's the `callable` fixer, and it's specific to Python 3.1 support. If you only need to support Python ≥ 3.2, you don't need the `callable` fixer.

6.2 – Now Fixers, With Review and Testing

These `2to3` fixers also make changes that are Python 2.7-compatible. **They operate on more ambiguous parts of Python's 2/3 syntax changes** where the best choice is not always clear. **On rare occasions, they can even be incorrect. Each fixer's changes deserve review.**

- `buffer`
- `dict`
- `filter`
- `idioms`
- `import`
- `map`
- `raise`
- `renames`
- `xrange`
- `zip`

To emphasize, I recommend review for *all* of the changes suggested by each of the fixers above. Here's details of three *specific* fixers from that list that provide examples of why you would want to review their changes.

6.2.1 – The `dict` Fixer

The `dict` fixer's changes are **harmless** because they're very **conservative**, but the conservative behavior also creates many changes that **aren't necessary**. The fixer wraps the result of many `dict` methods with `list()`.

Here's an example of code before and after the `dict` fixer has run. The added call to `list()` is unnecessary visual clutter so your code clarity suffers a little. The code is also slightly more efficient without it.

Before (valid in both Python 2 and 3) –

```
for key in my_dict.keys():
    print(key)
```

After (valid in both Python 2 and 3) –

```
for key in list(my_dict.keys()):
    print(key)
```

Here's another example of code before and after the `dict` fixer has run. Here, the fixer has absolutely done the right thing. The added call to `list()` is necessary to avoid a `TypeError`.

Before (valid in Python 2 only) –

```
foo = my_dict.keys() + ['a', 'b', 'c']
```

After (valid in both Python 2 and 3) –

```
foo = list(my_dict.keys()) + ['a', 'b', 'c']
```

6.2.2 – The `idioms` Fixer

The `idioms` fixer can change meaning. For example, it treats `isinstance()` as a drop-in replacement for `type is SomeClass`, which isn't always the case. Here's some code where the two give different results.

```
>>> type('') is basestring
False
>>> isinstance('', basestring)
True
```

You're unlikely to encounter a breaking change from this fixer, but review is still warranted.

6.2.3 – The `xrange` Fixer

The changes suggested by the `xrange` fixer are always syntactically correct, but in some cases you might not like the change. This fixer replaces `xrange()` with `range()`. The Python documentation states that the differences between the two are minimal, but names some specific exceptions where they're not[10]. If your code falls into one of those exceptional cases, you might want to move this fixer to the "Later" category.

10 https://docs.python.org/2.7/library/functions.html#xrange

6.3 – Later Fixers

These fixers make Python 2-incompatible changes. Most are very narrowly targeted (e.g. `getcwdu`), yawningly straightforward (`intern`), or both. The `unicode` fixer is an exception and merits its own section.

- `basestring`
- `exec`
- `future`
- `getcwdu`
- `imports and imports2`
- `intern`
- `itertools`
- `itertools_imports`
- `long`
- `metaclass`
- `print`
- `raw_input`
- `throw`
- `unicode`
- `urllib`

6.3.1 – Later Fixers – The `unicode` Fixer

The `unicode` fixer performs double duty. First, it strips the `'u'` prefix from all string literals. That prefix is only meaningful in Python 2, so removing it from Python 3 code is harmless and reduces visual clutter.

The `unicode` fixer's second duty is to replace calls to `unicode()` with calls to `str()`. This is where subtle changes can arise. If the objects being passed in those calls implement `__unicode__()` methods, then this fixer will change behavior because those `__unicode__()` methods won't get invoked anymore.

Under Python 2, calling `unicode()` invokes `my_object.__unicode__()`. Under Python 3, there is no `unicode()` builtin method, so it's not possible to invoke `my_object.__unicode__()` unless you call it via that explicit syntax.

Fortunately, there's a straightforward solution that you can apply in advance. The decorator `@python_2_unicode_compatible` (available under an MIT license from the `six` project and under a BSD license from the Django project) allows you to convert your Python 2 classes to use `__str__()` methods that return `unicode`. This idea takes a little getting used to under Python 2, but it works just fine. The advantage is that it makes the `unicode` fixer's changes always correct under Python 3.

Python 2-to-3 Migration Guide

About the Author

Philip Semanchuk has been developing software professionally for over 30 years. If you'd like advice or help with migrating from Python 2 to 3, any other aspect of Python, or software development in general, feel free to get in touch: philip@pyspoken.com

License

Feedback

Questions, comments, errata, and other feedback are all welcome at philip@pyspoken.com

Legal

Python 2-to-3 Migration Guide

Document History

Version 1.0.0 (2018-02-13)
Original version

Version 1.0.1 (2018-07-22)
Updated to reflect official EOL date for Python 2 and the release of Python 3.