Porting Python 2 Code to Python 3

Release 3.3.0

Guido van Rossum
Fred L. Drake, Jr., editor

September 29, 2012

Python Software Foundation
Email: docs@python.org

Contents

author Brett Cannon

Abstract

With Python 3 being the future of Python while Python 2 is still in active use, it is good to have your project available for both major releases of Python. This guide is meant to help you choose which strategy works best for your project to support both Python 2 & 3 along with how to execute that strategy.
If you are looking to port an extension module instead of pure Python code, please see *cporting-howto*.

1 Choosing a Strategy

When a project makes the decision that it's time to support both Python 2 & 3, a decision needs to be made as to how to go about accomplishing that goal. The chosen strategy will depend on how large the project's existing codebase is and how much divergence you want from your Python 2 codebase from your Python 3 one (e.g., starting a new version with Python 3).

If your project is brand-new or does not have a large codebase, then you may want to consider writing/porting *all of your code for Python 3 and use 3to2* to port your code for Python 2.

If you would prefer to maintain a codebase which is semantically **and** syntactically compatible with Python 2 & 3 simultaneously, you can write *Python 2/3 Compatible Source*. While this tends to lead to somewhat non-idiomatic code, it does mean you keep a rapid development process for you, the developer.

Finally, you do have the option of *using 2to3* to translate Python 2 code into Python 3 code (with some manual help). This can take the form of branching your code and using 2to3 to start a Python 3 branch. You can also have users perform the translation at installation time automatically so that you only have to maintain a Python 2 codebase.

Regardless of which approach you choose, porting is not as hard or time-consuming as you might initially think. You can also tackle the problem piece-meal as a good portion of porting is simply updating your code to follow current best practices in a Python 2/3 compatible way.

1.1 Universal Bits of Advice

Regardless of what strategy you pick, there are a few things you should consider.

One is make sure you have a robust test suite. You need to make sure everything continues to work, just like when you support a new minor version of Python. This means making sure your test suite is thorough and is ported properly between Python 2 & 3. You will also most likely want to use something like tox to automate testing between both a Python 2 and Python 3 VM.

Two, once your project has Python 3 support, make sure to add the proper classifier on the Cheeseshop (PyPI). To have your project listed as Python 3 compatible it must have the Python 3 classifier (from http://techspot.zzzeek.org/2011/01/24/zzzeek-s-guide-to-python-3-porting/):

```
setup(
  name='Your Library',
  version='1.0',
  classifiers=[
      # make sure to use :: Python *and* :: Python :: 3 so
      # that pypi can list the package on the python 3 page
      'Programming Language :: Python',
      'Programming Language :: Python :: 3'
```

```
    ],
    packages=['yourlibrary'],
    # make sure to add custom_fixers to the MANIFEST.in
    include_package_data=True,
    # ...
)
```

Doing so will cause your project to show up in the Python 3 packages list. You will know you set the classifier properly as visiting your project page on the Cheeseshop will show a Python 3 logo in the upper-left corner of the page.

Three, the six project provides a library which helps iron out differences between Python 2 & 3. If you find there is a sticky point that is a continual point of contention in your translation or maintenance of code, consider using a source-compatible solution relying on six. If you have to create your own Python 2/3 compatible solution, you can use `sys.version_info[0] >= 3` as a guard.

Four, read all the approaches. Just because some bit of advice applies to one approach more than another doesn't mean that some advice doesn't apply to other strategies.

Five, drop support for older Python versions if possible. Python 2.5 introduced a lot of useful syntax and libraries which have become idiomatic in Python 3. Python 2.6 introduced future statements which makes compatibility much easier if you are going from Python 2 to 3. Python 2.7 continues the trend in the stdlib. So choose the newest version of Python which you believe can be your minimum support version and work from there.

2 Python 3 and 3to2

If you are starting a new project or your codebase is small enough, you may want to consider writing your code for Python 3 and backporting to Python 2 using 3to2. Thanks to Python 3 being more strict about things than Python 2 (e.g., bytes vs. strings), the source translation can be easier and more straightforward than from Python 2 to 3. Plus it gives you more direct experience developing in Python 3 which, since it is the future of Python, is a good thing long-term.

A drawback of this approach is that 3to2 is a third-party project. This means that the Python core developers (and thus this guide) can make no promises about how well 3to2 works at any time. There is nothing to suggest, though, that 3to2 is not a high-quality project.

3 Python 2 and 2to3

Included with Python since 2.6, the 2to3 tool (and `lib2to3` module) helps with porting Python 2 to Python 3 by performing various source translations. This is a perfect solution for projects which wish to branch their Python 3 code from their Python 2 codebase and maintain them as independent codebases. You can even begin preparing to use this approach today by writing future-compatible Python code which works cleanly in Python 2 in conjunction with 2to3; all steps outlined below will work with Python 2 code up to the point when the actual use of 2to3 occurs.

Use of 2to3 as an on-demand translation step at install time is also possible, preventing the need to maintain a separate Python 3 codebase, but this approach does come with some drawbacks. While users will only have to pay the translation cost once at installation, you as a developer will need to pay the cost regularly during development. If your codebase is sufficiently large enough then the translation step ends up acting like a compilation step, robbing you of the rapid development process you are used to with Python. Obviously the time required to translate a project will vary, so do an experimental translation just to see how long it takes to evaluate whether you prefer this approach compared to using *Python 2/3 Compatible Source* or simply keeping a separate Python 3 codebase.

Below are the typical steps taken by a project which uses a 2to3-based approach to supporting Python 2 & 3.

3.1 Support Python 2.7

As a first step, make sure that your project is compatible with Python 2.7. This is just good to do as Python 2.7 is the last release of Python 2 and thus will be used for a rather long time. It also allows for use of the `-3` flag to Python to help discover places in your code which 2to3 cannot handle but are known to cause issues.

3.2 Try to Support Python 2.6 and Newer Only

While not possible for all projects, if you can support Python 2.6 and newer **only**, your life will be much easier. Various future statements, stdlib additions, etc. exist only in Python 2.6 and later which greatly assist in porting to Python 3. But if you project must keep support for Python 2.5 (or even Python 2.4) then it is still possible to port to Python 3.

Below are the benefits you gain if you only have to support Python 2.6 and newer. Some of these options are personal choice while others are **strongly** recommended (the ones that are more for personal choice are labeled as such). If you continue to support older versions of Python then you at least need to watch out for situations that these solutions fix.

```
from __future__ import print_function
```

This is a personal choice. 2to3 handles the translation from the print statement to the print function rather well so this is an optional step. This future statement does help, though, with getting used to typing `print('Hello, World')` instead of `print 'Hello, World'`.

```
from __future__ import unicode_literals
```

Another personal choice. You can always mark what you want to be a (unicode) string with a `u` prefix to get the same effect. But regardless of whether you use this future statement or not, you **must** make sure you know exactly which Python 2 strings you want to be bytes, and which are to be strings. This means you should, **at minimum** mark all strings that are meant to be text strings with a `u` prefix if you do not use this future statement.

Bytes literals

This is a **very** important one. The ability to prefix Python 2 strings that are meant to contain bytes with a `b` prefix help to very clearly delineate what is and is not a Python 3 string. When you run 2to3 on code, all Python 2 strings become Python 3 strings **unless** they are prefixed with `b`.

There are some differences between byte literals in Python 2 and those in Python 3 thanks to the bytes type just being an alias to `str` in Python 2. Probably the biggest "gotcha" is that indexing results in different values. In Python 2, the value of `b'py'[1]` is `'y'`, while in Python 3 it's `121`. You can avoid this disparity by always slicing at the size of a single element: `b'py'[1:2]` is `'y'` in Python 2 and `b'y'` in Python 3 (i.e., close enough).

You cannot concatenate bytes and strings in Python 3. But since Python 2 has bytes aliased to `str`, it will succeed: `b'a' + u'b'` works in Python 2, but `b'a' + 'b'` in Python 3 is a `TypeError`. A similar issue also comes about when doing comparisons between bytes and strings.

3.3 Supporting Python 2.5 and Newer Only

If you are supporting Python 2.5 and newer there are still some features of Python that you can utilize.

`from __future__ import absolute_import`

Implicit relative imports (e.g., importing `spam.bacon` from within `spam.eggs` with the statement `import bacon`) does not work in Python 3. This future statement moves away from that and allows the use of explicit relative imports (e.g., `from . import bacon`).

In Python 2.5 you must use the __future__ statement to get to use explicit relative imports and prevent implicit ones. In Python 2.6 explicit relative imports are available without the statement, but you still want the __future__ statement to prevent implicit relative imports. In Python 2.7 the __future__ statement is not needed. In other words, unless you are only supporting Python 2.7 or a version earlier than Python 2.5, use the __future__ statement.

3.4 Handle Common "Gotchas"

There are a few things that just consistently come up as sticking points for people which 2to3 cannot handle automatically or can easily be done in Python 2 to help modernize your code.

`from __future__ import division`

While the exact same outcome can be had by using the `-Qnew` argument to Python, using this future statement lifts the requirement that your users use the flag to get the expected behavior of division in Python 3 (e.g., `1/2 == 0.5; 1//2 == 0`).

Specify when opening a file as binary

Unless you have been working on Windows, there is a chance you have not always bothered to add the `b` mode when opening a binary file (e.g., `rb` for binary reading). Under Python 3, binary files and text files are clearly distinct and mutually incompatible; see the `io` module for details. Therefore, you **must** make a decision of whether a file will be used for binary access (allowing to read and/or write bytes data) or text access (allowing to read and/or write unicode data).

Text files

Text files created using `open()` under Python 2 return byte strings, while under Python 3 they return unicode strings. Depending on your porting strategy, this can be an issue.

If you want text files to return unicode strings in Python 2, you have two possibilities:

- Under Python 2.6 and higher, use `io.open()`. Since `io.open()` is essentially the same function in both Python 2 and Python 3, it will help iron out any issues that might arise.
- If pre-2.6 compatibility is needed, then you should use `codecs.open()` instead. This will make sure that you get back unicode strings in Python 2.

Subclass `object`

New-style classes have been around since Python 2.2. You need to make sure you are subclassing from `object` to avoid odd edge cases involving method resolution order, etc. This continues to be totally valid in Python 3 (although unneeded as all classes implicitly inherit from `object`).

Deal With the Bytes/String Dichotomy

One of the biggest issues people have when porting code to Python 3 is handling the bytes/string dichotomy. Because Python 2 allowed the `str` type to hold textual data, people have over the years been rather loose in their delineation of what `str` instances held text compared to bytes. In Python 3 you cannot be so care-free anymore and need to properly handle the difference. The key handling this issue is to make sure that **every** string literal

in your Python 2 code is either syntactically of functionally marked as either bytes or text data. After this is done you then need to make sure your APIs are designed to either handle a specific type or made to be properly polymorphic.

Mark Up Python 2 String Literals

First thing you must do is designate every single string literal in Python 2 as either textual or bytes data. If you are only supporting Python 2.6 or newer, this can be accomplished by marking bytes literals with a `b` prefix and then designating textual data with a `u` prefix or using the `unicode_literals` future statement.

If your project supports versions of Python predating 2.6, then you should use the six project and its `b()` function to denote bytes literals. For text literals you can either use six's `u()` function or use a `u` prefix.

Decide what APIs Will Accept

In Python 2 it was very easy to accidentally create an API that accepted both bytes and textual data. But in Python 3, thanks to the more strict handling of disparate types, this loose usage of bytes and text together tends to fail.

Take the dict `{b'a': 'bytes', u'a': 'text'}` in Python 2.6. It creates the dict `{u'a': 'text'}` since `b'a' == u'a'`. But in Python 3 the equivalent dict creates `{b'a': 'bytes', 'a': 'text'}`, i.e., no lost data. Similar issues can crop up when transitioning Python 2 code to Python 3.

This means you need to choose what an API is going to accept and create and consistently stick to that API in both Python 2 and 3.

Bytes / Unicode Comparison

In Python 3, mixing bytes and unicode is forbidden in most situations; it will raise a `TypeError` where Python 2 would have attempted an implicit coercion between types. However, there is one case where it doesn't and it can be very misleading:

```
>>> b"" == ""
False
```

This is because an equality comparison is required by the language to always succeed (and return `False` for incompatible types). However, this also means that code incorrectly ported to Python 3 can display buggy behaviour if such comparisons are silently executed. To detect such situations, Python 3 has a `-b` flag that will display a warning:

```
$ python3 -b
>>> b"" == ""
__main__:1: BytesWarning: Comparison between bytes and string
False
```

To turn the warning into an exception, use the `-bb` flag instead:

```
$ python3 -bb
>>> b"" == ""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
BytesWarning: Comparison between bytes and string
```

Indexing bytes objects

Another potentially surprising change is the indexing behaviour of bytes objects in Python 3:

```
>>> b"xyz"[0]
120
```

Indeed, Python 3 bytes objects (as well as `bytearray` objects) are sequences of integers. But code converted from Python 2 will often assume that indexing a bytestring produces another bytestring, not an integer. To reconcile both behaviours, use slicing:

```
>>> b"xyz"[0:1]
b'x'
>>> n = 1
>>> b"xyz"[n:n+1]
b'y'
```

The only remaining gotcha is that an out-of-bounds slice returns an empty bytes object instead of raising `IndexError`:

```
>>> b"xyz"[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index out of range
>>> b"xyz"[3:4]
b''
```

`__str__()/__unicode__()`

In Python 2, objects can specify both a string and unicode representation of themselves. In Python 3, though, there is only a string representation. This becomes an issue as people can inadvertently do things in their `__str__()` methods which have unpredictable results (e.g., infinite recursion if you happen to use the `unicode(self).encode('utf8')` idiom as the body of your `__str__()` method).

There are two ways to solve this issue. One is to use a custom 2to3 fixer. The blog post at http://lucumr.pocoo.org/2011/1/22/forwards-compatible-python/ specifies how to do this. That will allow 2to3 to change all instances of `def __unicode(self): ...` to `def __str__(self): ...`. This does require that you define your `__str__()` method in Python 2 before your `__unicode__()` method.

The other option is to use a mixin class. This allows you to only define a `__unicode__()` method for your class and let the mixin derive `__str__()` for you (code from http://lucumr.pocoo.org/2011/1/22/forwards-compatible-python/):

```
import sys

class UnicodeMixin(object):

  """Mixin class to handle defining the proper __str__/__unicode__
  methods in Python 2 or 3."""

  if sys.version_info[0] >= 3: # Python 3
      def __str__(self):
          return self.__unicode__()
  else:  # Python 2
      def __str__(self):
          return self.__unicode__().encode('utf8')


class Spam(UnicodeMixin):

  def __unicode__(self):
      return u'spam-spam-bacon-spam'  # 2to3 will remove the 'u' prefix
```

Don't Index on Exceptions

In Python 2, the following worked:

```
>>> exc = Exception(1, 2, 3)
>>> exc.args[1]
2
>>> exc[1]  # Python 2 only!
2
```

But in Python 3, indexing directly on an exception is an error. You need to make sure to only index on the `BaseException.args` attribute which is a sequence containing all arguments passed to the `__init__()` method.

Even better is to use the documented attributes the exception provides.

Don't use `__getslice__` & Friends

Been deprecated for a while, but Python 3 finally drops support for `__getslice__()`, etc. Move completely over to `__getitem__()` and friends.

Updating doctests

2to3 will attempt to generate fixes for doctests that it comes across. It's not perfect, though. If you wrote a monolithic set of doctests (e.g., a single docstring containing all of your doctests), you should at least consider breaking the doctests up into smaller pieces to make it more manageable to fix. Otherwise it might very well be worth your time and effort to port your tests to `unittest`.

Update *map* for imbalanced input sequences

With Python 2, *map* would pad input sequences of unequal length with *None* values, returning a sequence as long as the longest input sequence.

With Python 3, if the input sequences to *map* are of unequal length, *map* will stop at the termination of the shortest of the sequences. For full compatibility with *map* from Python 2.x, also wrap the sequences in `itertools.zip_longest()`, e.g. `map(func, *sequences)` becomes `list(map(func, itertools.zip_longest(*sequences)))`.

3.5 Eliminate `-3` Warnings

When you run your application's test suite, run it using the `-3` flag passed to Python. This will cause various warnings to be raised during execution about things that 2to3 cannot handle automatically (e.g., modules that have been removed). Try to eliminate those warnings to make your code even more portable to Python 3.

3.6 Run 2to3

Once you have made your Python 2 code future-compatible with Python 3, it's time to use 2to3 to actually port your code.

Manually

To manually convert source code using 2to3, you use the `2to3` script that is installed with Python 2.6 and later.:

```
2to3 <directory or file to convert>
```

This will cause 2to3 to write out a diff with all of the fixers applied for the converted source code. If you would like 2to3 to go ahead and apply the changes you can pass it the `-w` flag:

```
2to3 -w <stuff to convert>
```

There are other flags available to control exactly which fixers are applied, etc.

During Installation

When a user installs your project for Python 3, you can have either `distutils` or Distribute run 2to3 on your behalf. For distutils, use the following idiom:

```
try:  # Python 3
  from distutils.command.build_py import build_py_2to3 as build_py
except ImportError:  # Python 2
  from distutils.command.build_py import build_py

setup(cmdclass = {'build_py': build_py},
  # ...
)
```

For Distribute:

```
setup(use_2to3=True,
  # ...
)
```

This will allow you to not have to distribute a separate Python 3 version of your project. It does require, though, that when you perform development that you at least build your project and use the built Python 3 source for testing.

3.7 Verify & Test

At this point you should (hopefully) have your project converted in such a way that it works in Python 3. Verify it by running your unit tests and making sure nothing has gone awry. If you miss something then figure out how to fix it in Python 3, backport to your Python 2 code, and run your code through 2to3 again to verify the fix transforms properly.

4 Python 2/3 Compatible Source

While it may seem counter-intuitive, you can write Python code which is source-compatible between Python 2 & 3. It does lead to code that is not entirely idiomatic Python (e.g., having to extract the currently raised exception from `sys.exc_info()[1]`), but it can be run under Python 2 **and** Python 3 without using 2to3 as a translation step (although the tool should be used to help find potential portability problems). This allows you to continue to have a rapid development process regardless of whether you are developing under Python 2 or Python 3. Whether this approach or using *Python 2 and 2to3* works best for you will be a per-project decision.

To get a complete idea of what issues you will need to deal with, see the What's New in Python 3.0. Others have reorganized the data in other formats such as http://docs.pythonsprints.com/python3_porting/py-porting.html .

The following are some steps to take to try to support both Python 2 & 3 from the same source code.

4.1 Follow The Steps for Using 2to3

All of the steps outlined in how to *port Python 2 code with 2to3* apply to creating a Python 2/3 codebase. This includes trying only support Python 2.6 or newer (the `__future__` statements work in Python 3 without issue), eliminating warnings that are triggered by `-3`, etc.

You should even consider running 2to3 over your code (without committing the changes). This will let you know where potential pain points are within your code so that you can fix them properly before they become an issue.

4.2 Use six

The six project contains many things to help you write portable Python code. You should make sure to read its documentation from beginning to end and use any and all features it provides. That way you will minimize any mistakes you might make in writing cross-version code.

4.3 Capturing the Currently Raised Exception

One change between Python 2 and 3 that will require changing how you code (if you support Python 2.5 and earlier) is accessing the currently raised exception. In Python 2.5 and earlier the syntax to access the current exception is:

```
try:
  raise Exception()
except Exception, exc:
  # Current exception is 'exc'
  pass
```

This syntax changed in Python 3 (and backported to Python 2.6 and later) to:

```
try:
  raise Exception()
except Exception as exc:
  # Current exception is 'exc'
  # In Python 3, 'exc' is restricted to the block; Python 2.6 will "leak"
  pass
```

Because of this syntax change you must change to capturing the current exception to:

```
try:
  raise Exception()
except Exception:
  import sys
  exc = sys.exc_info()[1]
  # Current exception is 'exc'
  pass
```

You can get more information about the raised exception from `sys.exc_info()` than simply the current exception instance, but you most likely don't need it.

Note: In Python 3, the traceback is attached to the exception instance through the `__traceback__` attribute. If the instance is saved in a local variable that persists outside of the `except` block, the traceback will create a reference cycle with the current frame and its dictionary of local variables. This will delay reclaiming dead resources until the next cyclic *garbage collection* pass.

In Python 2, this problem only occurs if you save the traceback itself (e.g. the third element of the tuple returned by `sys.exc_info()`) in a variable.

---

5 Other Resources

The authors of the following blog posts, wiki pages, and books deserve special thanks for making public their tips for porting Python 2 code to Python 3 (and thus helping provide information for this document):

- http://python3porting.com/
- http://docs.pythonsprints.com/python3_porting/py-porting.html
- http://techspot.zzzeek.org/2011/01/24/zzzeek-s-guide-to-python-3-porting/
- http://dabeaz.blogspot.com/2011/01/porting-py65-and-my-superboard-to.html

- http://lucumr.pocoo.org/2011/1/22/forwards-compatible-python/
- http://lucumr.pocoo.org/2010/2/11/porting-to-python-3-a-guide/
- http://wiki.python.org/moin/PortingPythonToPy3k

If you feel there is something missing from this document that should be added, please email the python-porting mailing list.