Porting Python 2 Code to Python 3

Release 3.4.2

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

October 08, 2014

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 figure out how best to support both Python 2 & 3 simultaneously.
If you are looking to port an extension module instead of pure Python code, please see *cporting-howto*.
If you would like to read one core Python developer's take on why Python 3 came into existence, you can read Nick Coghlan's Python 3 Q & A.
If you prefer to read a (free) book on porting a project to Python 3, consider reading Porting to Python 3 by Lennart Regebro which should cover much of what is discussed in this HOWTO.
For help with porting, you can email the python-porting mailing list with questions.

1 The Short Version

- Decide what's the oldest version of Python 2 you want to support (if at all)
- Make sure you have a thorough test suite and use continuous integration testing to make sure you stay compatible with the versions of Python you care about
- If you have dependencies, check their Python 3 status using caniusepython3 (command-line tool, web app)

With that done, your options are:

- If you are dropping Python 2 support, use 2to3 to port to Python 3
- If you are keeping Python 2 support, then start writing Python 2/3-compatible code starting **TODAY**
  - If you have dependencies that have not been ported, reach out to them to port their project while working to make your code compatible with Python 3 so you're ready when your dependencies are all ported
  - If all your dependencies have been ported (or you have none), go ahead and port to Python 3
- If you are creating a new project that wants to have 2/3 compatibility, code in Python 3 and then backport to Python 2

2 Before You Begin

If your project is on the Cheeseshop/PyPI, make sure it has the proper trove classifiers to signify what versions of Python it **currently** supports. At minimum you should specify the major version(s), e.g. `Programming Language :: Python :: 2` if your project currently only supports Python 2. It is preferrable that you be as specific as possible by listing every major/minor version of Python that you support, e.g. if your project supports Python 2.6 and 2.7, then you want the classifiers of:

```
Programming Language :: Python :: 2
Programming Language :: Python :: 2.6
Programming Language :: Python :: 2.7
```

Once your project supports Python 3 you will want to go back and add the appropriate classifiers for Python 3 as well. This is important as setting the `Programming Language :: Python :: 3` classifier will lead to your project being listed under the Python 3 Packages section of PyPI.

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/feature release of Python. This means making sure your test suite is thorough and is ported properly between Python 2 & 3 (consider using coverage to measure that you have effective test coverage). You will also most likely want to use something like tox to automate testing between all of your supported versions of Python.

You will also want to **port your tests first** so that you can make sure that you detect breakage during the transition. Tests also tend to be simpler than the code they are testing so it gives you an idea of how easy it can be to port code.

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. Choose the newest version of Python which you believe can be your minimum support version and work from there.

Target the newest version of Python 3 that you can. Beyond just the usual bugfixes, compatibility has continued to improve between Python 2 and 3 as time has passed. E.g. Python 3.3 added back the `u` prefix for strings, making source-compatible Python code easier to write.

3 Writing Source-Compatible Python 2/3 Code

Over the years the Python community has discovered that the easiest way to support both Python 2 and 3 in parallel is to write Python code that works in either version. While this might sound counter-intuitive at first, it actually is not difficult and typically only requires following some select (non-idiomatic) practices and using some key projects to help make bridging between Python 2 and 3 easier.

3.1 Projects to Consider

The lowest level library for supporting Python 2 & 3 simultaneously is six. Reading through its documentation will give you an idea of where exactly the Python language changed between versions 2 & 3 and thus what you will want the library to help you continue to support.

To help automate porting your code over to using six, you can use modernize. This project will attempt to rewrite your code to be as modern as possible while using six to smooth out any differences between Python 2 & 3.

If you want to write your compatible code to feel more like Python 3 there is the future project. It tries to provide backports of objects from Python 3 so that you can use them from Python 2-compatible code, e.g. replacing the `bytes` type from Python 2 with the one from Python 3. It also provides a translation script like modernize (its translation code is actually partially based on it) to help start working with a pre-existing code base. It is also unique in that its translation script will also port Python 3 code backwards as well as Python 2 code forwards.

3.2 Tips & Tricks

To help with writing source-compatible code using one of the projects mentioned in Projects to Consider, consider following the below suggestions. Some of them are handled by the suggested projects, so if you do use one of them then read their documentation first to see which suggestions below will taken care of for you.

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 where compatibility might be an issue (the `-3` flag is in Python 2.6 but Python 2.7 adds more warnings).

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 supporting Python 3.

But if you project must keep support for Python 2.5 then it is still possible to simultaneously support 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 and handle them appropriately (which is where library help from e.g. six comes in handy).

```
from __future__ import print_function
```

It will not only get you used to typing `print()` as a function instead of a statement, but it will also give you the various benefits the function has over the Python 2 statement (six provides a function if you support Python 2.5 or older).

```
from __future__ import unicode_literals
```

If you choose to use this future statement then all string literals in Python 2 will be assumed to be Unicode (as is already the case in Python 3). If you choose not to use this future statement then you should mark all of your text strings with a `u` prefix and only support Python 3.3 or newer. But you are **strongly** advised to do one or the other (six provides a function in case you don't want to use the future statement **and** you want to support Python 3.2 or older).

Bytes/string literals

This is a **very** important one. Prefix Python 2 strings that are meant to contain bytes with a `b` prefix to very clearly delineate what is and is not a Python 3 text string (six provides a function to use for Python 2.5 compatibility).

This point cannot be stressed enough: make sure you know what all of your string literals in Python 2 are meant to be in Python 3. Any string literal that should be treated as bytes should have the `b` prefix. Any string literal that should be Unicode/text in Python 2 should either have the `u` literal (supported, but ignored, in Python 3.3 and later) or you should have `from __future__ import unicode_literals` at the top of the file. But the key point is you should know how Python 3 will treat every one one of your string literals and you should mark them as appropriate.

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. See the Handle Common "Gotchas" section for what to watch out for.

```
from __future__ import absolute_import
```

Discussed in more detail below, but you should use this future statement to prevent yourself from accidentally using implicit relative imports.

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`) do 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 this __future__ statement.

Mark all Unicode strings with a `u` prefix

While Python 2.6 has a `__future__` statement to automatically cause Python 2 to treat all string literals as Unicode, Python 2.5 does not have that shortcut. This means you should go through and mark all string literals with a `u` prefix to turn them explicitly into text strings where appropriate and only support Python 3.3 or newer. Otherwise use a project like six which provides a function to pass all text string literals through.

Capturing 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; in Python 2.6/2.7 it will "leak".
  pass
```

Because of this syntax change you must change how you capture the current exception in Python 2.5 and earlier 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.

Handle Common "Gotchas"

These are things to watch out for no matter what version of Python 2 you are supporting which are not syntactic considerations.

```
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 to handling this issue is to make sure that **every** string literal in your Python 2 code is either syntactically or 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).

You can use a mixin class to work around this. 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

Don't forget to make them Python 2/3 compatible as well. 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, when `map` was given more than one input sequence it would pad the shorter sequences 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, wrap the sequence arguments in `itertools.zip_longest()`, e.g. `map(func, *sequences)` becomes `list(map(func, itertools.zip_longest(*sequences)))`.

3.3 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 are semantic changes between Python 2 and 3. Try to eliminate those warnings to make your code even more portable to Python 3.

4 Alternative Approaches

While supporting Python 2 & 3 simultaneously is typically the preferred choice by people so that they can continue to improve code and have it work for the most number of users, your life may be easier if you only have to support one major version of Python going forward.

4.1 Supporting Only Python 3 Going Forward From Python 2 Code

If you have Python 2 code but going forward only want to improve it as Python 3 code, then you can use 2to3 to translate your Python 2 code to Python 3 code. This is only recommended, though, if your current version of your project is going into maintenance mode and you want all new features to be exclusive to Python 3.

4.2 Backporting Python 3 code to Python 2

If you have Python 3 code and have little interest in supporting Python 2 you can use 3to2 to translate from Python 3 code to Python 2 code. This is only recommended if you don't plan to heavily support Python 2 users. Otherwise write your code for Python 3 and then backport as far back as you want. This is typically easier than going from Python 2 to 3 as you will have worked out any difficulties with e.g. bytes/strings, etc.

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 and its various revisions over the years):

- http://wiki.python.org/moin/PortingPythonToPy3k
- 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/
- https://wiki.ubuntu.com/Python/3

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