Porting Python 2 Code to Python 3

Release 3.7.7

Guido van Rossum
and the Python development team

March 10, 2020

Python Software Foundation
Email: docs@python.org

Contents

1 The Short Explanation **1**

2 Details **2**
2.1 Drop support for Python 2.6 and older . . . 2
2.2 Make sure you specify the proper version support in your `setup.py` file . . . 2
2.3 Have good test coverage . . . 3
2.4 Learn the differences between Python 2 & 3 . . . 3
2.5 Update your code . . . 3
2.6 Prevent compatibility regressions . . . 5
2.7 Check which dependencies block your transition . . . 6
2.8 Update your `setup.py` file to denote Python 3 compatibility . . . 6
2.9 Use continuous integration to stay compatible . . . 6
2.10 Consider using optional static type checking . . . 6

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 or Brett Cannon's Why Python 3 exists.

For help with porting, you can email the python-porting mailing list with questions.

1 The Short Explanation

To make your project be single-source Python 2/3 compatible, the basic steps are:

1. Only worry about supporting Python 2.7
2. Make sure you have good test coverage (coverage.py can help; `pip install coverage`)
3. Learn the differences between Python 2 & 3
4. Use Futurize (or Modernize) to update your code (e.g. `pip install future`)
5. Use Pylint to help make sure you don't regress on your Python 3 support (`pip install pylint`)
6. Use caniusepython3 to find out which of your dependencies are blocking your use of Python 3 (`pip install caniusepython3`)
7. Once your dependencies are no longer blocking you, use continuous integration to make sure you stay compatible with Python 2 & 3 (tox can help test against multiple versions of Python; `pip install tox`)
8. Consider using optional static type checking to make sure your type usage works in both Python 2 & 3 (e.g. use mypy to check your typing under both Python 2 & Python 3).

2 Details

A key point about supporting Python 2 & 3 simultaneously is that you can start **today**! Even if your dependencies are not supporting Python 3 yet that does not mean you can't modernize your code **now** to support Python 3. Most changes required to support Python 3 lead to cleaner code using newer practices even in Python 2 code.

Another key point is that modernizing your Python 2 code to also support Python 3 is largely automated for you. While you might have to make some API decisions thanks to Python 3 clarifying text data versus binary data, the lower-level work is now mostly done for you and thus can at least benefit from the automated changes immediately.

Keep those key points in mind while you read on about the details of porting your code to support Python 2 & 3 simultaneously.

2.1 Drop support for Python 2.6 and older

While you can make Python 2.5 work with Python 3, it is **much** easier if you only have to work with Python 2.7. If dropping Python 2.5 is not an option then the six project can help you support Python 2.5 & 3 simultaneously (`pip install six`). Do realize, though, that nearly all the projects listed in this HOWTO will not be available to you.

If you are able to skip Python 2.5 and older, then the required changes to your code should continue to look and feel like idiomatic Python code. At worst you will have to use a function instead of a method in some instances or have to import a function instead of using a built-in one, but otherwise the overall transformation should not feel foreign to you.

But you should aim for only supporting Python 2.7. Python 2.6 is no longer freely supported and thus is not receiving bugfixes. This means **you** will have to work around any issues you come across with Python 2.6. There are also some tools mentioned in this HOWTO which do not support Python 2.6 (e.g., Pylint), and this will become more commonplace as time goes on. It will simply be easier for you if you only support the versions of Python that you have to support.

2.2 Make sure you specify the proper version support in your `setup.py` file

In your `setup.py` file you should have the proper trove classifier specifying what versions of Python you support. As your project does not support Python 3 yet you should at least have `Programming Language :: Python :: 2 :: Only` specified. Ideally you should also specify each major/minor version of Python that you do support, e.g. `Programming Language :: Python :: 2.7`.

2.3 Have good test coverage

Once you have your code supporting the oldest version of Python 2 you want it to, you will want to make sure your test suite has good coverage. A good rule of thumb is that if you want to be confident enough in your test suite that any failures that appear after having tools rewrite your code are actual bugs in the tools and not in your code. If you want a number to aim for, try to get over 80% coverage (and don't feel bad if you find it hard to get better than 90% coverage). If you don't already have a tool to measure test coverage then coverage.py is recommended.

2.4 Learn the differences between Python 2 & 3

Once you have your code well-tested you are ready to begin porting your code to Python 3! But to fully understand how your code is going to change and what you want to look out for while you code, you will want to learn what changes Python 3 makes in terms of Python 2. Typically the two best ways of doing that is reading the "What's New" doc for each release of Python 3 and the Porting to Python 3 book (which is free online). There is also a handy cheat sheet from the Python-Future project.

2.5 Update your code

Once you feel like you know what is different in Python 3 compared to Python 2, it's time to update your code! You have a choice between two tools in porting your code automatically: Futurize and Modernize. Which tool you choose will depend on how much like Python 3 you want your code to be. Futurize does its best to make Python 3 idioms and practices exist in Python 2, e.g. backporting the `bytes` type from Python 3 so that you have semantic parity between the major versions of Python. Modernize, on the other hand, is more conservative and targets a Python 2/3 subset of Python, directly relying on six to help provide compatibility. As Python 3 is the future, it might be best to consider Futurize to begin adjusting to any new practices that Python 3 introduces which you are not accustomed to yet.

Regardless of which tool you choose, they will update your code to run under Python 3 while staying compatible with the version of Python 2 you started with. Depending on how conservative you want to be, you may want to run the tool over your test suite first and visually inspect the diff to make sure the transformation is accurate. After you have transformed your test suite and verified that all the tests still pass as expected, then you can transform your application code knowing that any tests which fail is a translation failure.

Unfortunately the tools can't automate everything to make your code work under Python 3 and so there are a handful of things you will need to update manually to get full Python 3 support (which of these steps are necessary vary between the tools). Read the documentation for the tool you choose to use to see what it fixes by default and what it can do optionally to know what will (not) be fixed for you and what you may have to fix on your own (e.g. using `io.open()` over the built-in `open()` function is off by default in Modernize). Luckily, though, there are only a couple of things to watch out for which can be considered large issues that may be hard to debug if not watched for.

Division

In Python 3, `5 / 2 == 2.5` and not `2`; all division between `int` values result in a `float`. This change has actually been planned since Python 2.2 which was released in 2002. Since then users have been encouraged to add `from __future__ import division` to any and all files which use the `/` and `//` operators or to be running the interpreter with the `-Q` flag. If you have not been doing this then you will need to go through your code and do two things:

1. Add `from __future__ import division` to your files
2. Update any division operator as necessary to either use `//` to use floor division or continue using `/` and expect a float

The reason that `/` isn't simply translated to `//` automatically is that if an object defines a `__truediv__` method but not `__floordiv__` then your code would begin to fail (e.g. a user-defined class that uses `/` to signify some operation but not `//` for the same thing or at all).

Text versus binary data

In Python 2 you could use the `str` type for both text and binary data. Unfortunately this confluence of two different concepts could lead to brittle code which sometimes worked for either kind of data, sometimes not. It also could lead to confusing APIs if people didn't explicitly state that something that accepted `str` accepted either text or binary data instead of one specific type. This complicated the situation especially for anyone supporting multiple languages as APIs wouldn't bother explicitly supporting `unicode` when they claimed text data support.

To make the distinction between text and binary data clearer and more pronounced, Python 3 did what most languages created in the age of the internet have done and made text and binary data distinct types that cannot blindly be mixed together (Python predates widespread access to the internet). For any code that deals only with text or only binary data, this separation doesn't pose an issue. But for code that has to deal with both, it does mean you might have to now care about when you are using text compared to binary data, which is why this cannot be entirely automated.

To start, you will need to decide which APIs take text and which take binary (it is **highly** recommended you don't design APIs that can take both due to the difficulty of keeping the code working; as stated earlier it is difficult to do well). In Python 2 this means making sure the APIs that take text can work with `unicode` and those that work with binary data work with the `bytes` type from Python 3 (which is a subset of `str` in Python 2 and acts as an alias for `bytes` type in Python 2). Usually the biggest issue is realizing which methods exist on which types in Python 2 & 3 simultaneously (for text that's `unicode` in Python 2 and `str` in Python 3, for binary that's `str`/`bytes` in Python 2 and `bytes` in Python 3). The following table lists the **unique** methods of each data type across Python 2 & 3 (e.g., the `decode()` method is usable on the equivalent binary data type in either Python 2 or 3, but it can't be used by the textual data type consistently between Python 2 and 3 because `str` in Python 3 doesn't have the method). Do note that as of Python 3.5 the `__mod__` method was added to the bytes type.

Text data	Binary data
	decode
encode	
format	
isdecimal	
isnumeric	

Making the distinction easier to handle can be accomplished by encoding and decoding between binary data and text at the edge of your code. This means that when you receive text in binary data, you should immediately decode it. And if your code needs to send text as binary data then encode it as late as possible. This allows your code to work with only text internally and thus eliminates having to keep track of what type of data you are working with.

The next issue is making sure you know whether the string literals in your code represent text or binary data. You should add a `b` prefix to any literal that presents binary data. For text you should add a `u` prefix to the text literal. (there is a `__future__` import to force all unspecified literals to be Unicode, but usage has shown it isn't as effective as adding a `b` or `u` prefix to all literals explicitly)

As part of this dichotomy you also need to be careful about opening files. 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 binary data to be read and/or written) or textual access (allowing text data to be read and/or written). You should also use `io.open()` for opening files instead of the built-in `open()` function as the `io` module is consistent from Python 2 to 3 while the built-in `open()` function is not (in Python 3 it's actually `io.open()`). Do not bother with the outdated practice of using `codecs.open()` as that's only necessary for keeping compatibility with Python 2.5.

The constructors of both `str` and `bytes` have different semantics for the same arguments between Python 2 & 3. Passing an integer to `bytes` in Python 2 will give you the string representation of the integer: `bytes(3) == '3'`. But in Python 3, an integer argument to `bytes` will give you a bytes object as long as the integer specified, filled with null bytes: `bytes(3) == b'\x00\x00\x00'`. A similar worry is necessary when passing a bytes object to `str`. In Python 2 you just get the bytes object back: `str(b'3') == b'3'`. But in Python 3 you get the string representation of the bytes object: `str(b'3') == "b'3'"`.

Finally, the indexing of binary data requires careful handling (slicing does **not** require any special handling). In Python 2, `b'123'[1] == b'2'` while in Python 3 `b'123'[1] == 50`. Because binary data is simply a collection

of binary numbers, Python 3 returns the integer value for the byte you index on. But in Python 2 because `bytes == str`, indexing returns a one-item slice of bytes. The six project has a function named `six.indexbytes()` which will return an integer like in Python 3: `six.indexbytes(b'123', 1)`.

To summarize:

1. Decide which of your APIs take text and which take binary data
2. Make sure that your code that works with text also works with `unicode` and code for binary data works with `bytes` in Python 2 (see the table above for what methods you cannot use for each type)
3. Mark all binary literals with a `b` prefix, textual literals with a `u` prefix
4. Decode binary data to text as soon as possible, encode text as binary data as late as possible
5. Open files using `io.open()` and make sure to specify the `b` mode when appropriate
6. Be careful when indexing into binary data

Use feature detection instead of version detection

Inevitably you will have code that has to choose what to do based on what version of Python is running. The best way to do this is with feature detection of whether the version of Python you're running under supports what you need. If for some reason that doesn't work then you should make the version check be against Python 2 and not Python 3. To help explain this, let's look at an example.

Let's pretend that you need access to a feature of `importlib` that is available in Python's standard library since Python 3.3 and available for Python 2 through importlib2 on PyPI. You might be tempted to write code to access e.g. the `importlib.abc` module by doing the following:

```
import sys

if sys.version_info[0] == 3:
    from importlib import abc
else:
    from importlib2 import abc
```

The problem with this code is what happens when Python 4 comes out? It would be better to treat Python 2 as the exceptional case instead of Python 3 and assume that future Python versions will be more compatible with Python 3 than Python 2:

```
import sys

if sys.version_info[0] > 2:
    from importlib import abc
else:
    from importlib2 import abc
```

The best solution, though, is to do no version detection at all and instead rely on feature detection. That avoids any potential issues of getting the version detection wrong and helps keep you future-compatible:

```
try:
    from importlib import abc
except ImportError:
    from importlib2 import abc
```

2.6 Prevent compatibility regressions

Once you have fully translated your code to be compatible with Python 3, you will want to make sure your code doesn't regress and stop working under Python 3. This is especially true if you have a dependency which is blocking you from actually running under Python 3 at the moment.

To help with staying compatible, any new modules you create should have at least the following block of code at the top of it:

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

You can also run Python 2 with the `-3` flag to be warned about various compatibility issues your code triggers during execution. If you turn warnings into errors with `-Werror` then you can make sure that you don't accidentally miss a warning.

You can also use the Pylint project and its `--py3k` flag to lint your code to receive warnings when your code begins to deviate from Python 3 compatibility. This also prevents you from having to run Modernize or Futurize over your code regularly to catch compatibility regressions. This does require you only support Python 2.7 and Python 3.4 or newer as that is Pylint's minimum Python version support.

2.7 Check which dependencies block your transition

After you have made your code compatible with Python 3 you should begin to care about whether your dependencies have also been ported. The caniusepython3 project was created to help you determine which projects – directly or indirectly – are blocking you from supporting Python 3. There is both a command-line tool as well as a web interface at https://caniusepython3.com.

The project also provides code which you can integrate into your test suite so that you will have a failing test when you no longer have dependencies blocking you from using Python 3. This allows you to avoid having to manually check your dependencies and to be notified quickly when you can start running on Python 3.

2.8 Update your `setup.py` file to denote Python 3 compatibility

Once your code works under Python 3, you should update the classifiers in your `setup.py` to contain `Programming Language :: Python :: 3` and to not specify sole Python 2 support. This will tell anyone using your code that you support Python 2 **and** 3. Ideally you will also want to add classifiers for each major/minor version of Python you now support.

2.9 Use continuous integration to stay compatible

Once you are able to fully run under Python 3 you will want to make sure your code always works under both Python 2 & 3. Probably the best tool for running your tests under multiple Python interpreters is tox. You can then integrate tox with your continuous integration system so that you never accidentally break Python 2 or 3 support.

You may also want to use the `-bb` flag with the Python 3 interpreter to trigger an exception when you are comparing bytes to strings or bytes to an int (the latter is available starting in Python 3.5). By default type-differing comparisons simply return `False`, but if you made a mistake in your separation of text/binary data handling or indexing on bytes you wouldn't easily find the mistake. This flag will raise an exception when these kinds of comparisons occur, making the mistake much easier to track down.

And that's mostly it! At this point your code base is compatible with both Python 2 and 3 simultaneously. Your testing will also be set up so that you don't accidentally break Python 2 or 3 compatibility regardless of which version you typically run your tests under while developing.

2.10 Consider using optional static type checking

Another way to help port your code is to use a static type checker like mypy or pytype on your code. These tools can be used to analyze your code as if it's being run under Python 2, then you can run the tool a second time as if your code is running under Python 3. By running a static type checker twice like this you can discover if you're e.g. misusing binary data type in one version of Python compared to another. If you add optional type hints to your code

you can also explicitly state whether your APIs use textual or binary data, helping to make sure everything functions as expected in both versions of Python.