Python2orPython3 - Python Wiki

•

Should I use Python 2 or Python 3 for my development activity?

Contents

What are the differences?

Short version: Python 2.x is legacy, Python 3.x is the present and future of the language

Python 3.0 was released in 2008. The final 2.x version 2.7 release came out in mid-2010, with a statement of extended support for this end-of-life release. The 2.x branch will see no new major releases after that. 3.x is under active development and has already seen over five years of stable releases, including version 3.3 in 2012, 3.4 in 2014, and 3.5 in 2015. This means that all recent standard library improvements, for example, are only available by default in Python 3.x.

Guido van Rossum (the original creator of the Python language) decided to clean up Python 2.x properly, with less regard for backwards compatibility than is the case for new releases in the 2.x range. The most drastic improvement is the better Unicode support (with all text strings being Unicode by default) as well as saner bytes/Unicode separation.

Besides, several aspects of the core language (such as print and exec being statements, integers using floor division) have been adjusted to be easier for newcomers to learn and to be more consistent with the rest of the language, and old cruft has been removed (for example, all classes are now new-style, "range()" returns a memory efficient iterable, not a list as in 2.x).

The What's New in Python 3.0 document provides a good overview of the major language changes and likely sources of incompatibility with existing Python 2.x code. Nick Coghlan (one of the CPython core developers) has also created a relatively extensive FAQ regarding the transition.

However, the broader Python ecosystem has amassed a significant amount of quality software over the years. The downside of breaking backwards compatibility in 3.x is that some of that software (especially in-house software in companies) still doesn't work on 3.x yet.

Which version should I use?

Which version you ought to use is mostly dependent on what you want to get done.

If you can do exactly what you want with Python 3.x, great! There are a few minor downsides, such as very slightly worse library support[1] and the fact that some current Linux distributions and Macs are still using 2.x as

default (although Python 3 ships with many of them), but as a language Python 3.x is definitely ready. As long as Python 3.x is installed on your user's computers (which ought to be easy, since many people reading this may only be developing something for themselves or an environment they control) and you're writing things where you know none of the Python 2.x modules are needed, it is an excellent choice. Also, most Linux distributions have Python 3.x already installed, and all have it available for end-users. Some are phasing out Python 2 as preinstalled default.[2]

In particular, instructors introducing Python to new programmers should consider teaching Python 3 first and then introducing the differences in Python 2 afterwards (if necessary), since Python 3 eliminates many quirks that can unnecessarily trip up beginning programmers trying to learn Python 2.

However, there are some key issues that may require you to use Python 2 rather than Python 3.

- Firstly, if you're deploying to an environment you don't control, that may impose a specific version, rather than allowing you a free selection from the available versions.
- Secondly, if you want to use a specific third party package or utility that doesn't yet have a released version that is compatible with Python 3, and porting that package is a non-trivial task, you may choose to use Python 2 in order to retain access to that package.

Python 3 already broadly supports creating GUI applications, with Tkinter in the standard library. Python 3 has been supported by PyQt almost from the day Python 3 was released; PySide added Python 3 support in 2011. GTK+ GUIs can be created with PyGObject which supports Python 3 and is the successor to PyGtk.

Many other major packages have been ported to Python 3 including:

- NumPy and SciPy (for number crunching and scientific computing)
- Django, Flask, CherryPy and Pyramid (for Web sites)
- PIL (an image processing module) was superseded by its fork Pillow, which supports Python 3.
- cx_Freeze (for packaging applications with their dependencies)
- py2exe (for packaging your application for Windows users)
- OpenCV 3, (an open source computer vision and machine learning library) now supports Python 3 in versions 3.0 and later.
- Requests, (an HTTP library for humans)
- lxml, (a powerful and Pythonic XML processing library combining libxml2/libxslt with the ElementTree API)
- BeautifulSoup4, (a screen-scraping library for parsing HTML and XML)
- The IPython/Jupyter project for interactive computing fully supports Python 3.
- And many, many more!

If you want to use Python 3.x, but you're afraid to because of a dependency, it's probably worthwhile doing some research first. This is a work in progress and this wiki page might be outdated. Furthermore, with the large common subset supported by both Python 2.6+ and Python 3.3+, much modern Python code should run largely unmodified on Python 3, especially code written to interoperate with web and GUI frameworks that force applications to correctly distinguish binary data and text (some assistance from the six compatibility module may be needed to handle name changes.

Even though the official python documentation and the tutorial have been completely updated for Python 3, there is still a lot of documentation (including examples) on the Web and in reference books that use Python 2, although more are being updated all the time. This can require some adjustment to make things work with Python 3 instead.

Some people just don't want to use Python 3.x, which is their prerogative. However, they are in the minority.

It is worth noting that if you wish to use an alternative implementation of Python such as IronPython, Jython or Pyston (or one of the longer list of Python platform or compiler implementations), Python 3 support is still relatively rare. This may affect you if you are interested in choosing such an implementation for reasons of integration with other systems or for performance.

But wouldn't I want to avoid 2.x? It's an old language with many mistakes, and it took a major version to get them out.

Well, not entirely. Some of the less disruptive improvements in 3.0 and 3.1 have been backported to 2.6 and 2.7, respectively. For more details on the backported features, see What's New in Python 2.6 and What's New in Python 2.7.

A non-exhaustive list of features which are only available in 3.x releases and won't be backported to the 2.x series:

- strings are Unicode by default
- clean Unicode/bytes separation
- exception chaining
- function annotations
- syntax for keyword-only arguments
- extended tuple unpacking
- non-local variable declarations

Also, language evolution is not limited to core syntactic or semantic changes. It also regards the standard library, where many improvements are done in 3.x that will not be backported directly to Python 2. See What's New in Python 3, for example. However, a number of the standard library improvements are also available through PyPI.

That said, well-written 2.x code can be a lot like 3.x code. That can mean many things, including using new-style classes, not using ancient deprecated arcane incantations of print, using lazy iterators where available,

etc. A practical example: good 2.x code will typically use xrange instead of range; xrange was the starting point for the Python 3.x range implementation (although range is even better in Python 3, since it can handle values larger than sys.maxint). It should be noted that xrange() is not included in Python 3.

Above all, it is recommended that you focus on writing *good* code so that 2.x vs 3.x becomes less of an issue. That includes writing full unit test suites, and getting Unicode right. (Python 3.x is significantly less forgiving than 2.x about Unicode versus bytes issues: This is considered to be a good thing, though it makes porting some software packages fairly annoying.)

I want to use Python 3, but there's this tiny library I want to use that's Python 2.x only. Do I really have to revert to using Python 2 or give up on using that library?

Assuming you can't find an alternative package that already supports Python 3, you still have a few options to consider:

- Port the library to 3.x. ("Porting" means that you make the library work on 3.x.)
- If that turns out to be really hard, and all your other dependencies do exist in 2.x, consider starting off in 2.x. As has already been explained in other places, good 2.x code will typically make switching painless as soon as every dependency has been successfully ported.
- Decide if the feature is really that important. Maybe you could drop it?

The ideal situation is that you try to port the library to 3.x. Often you'll find someone is already working on this. Even when that's not the case, existing project members will usually appreciate the help, especially as porting often finds bugs in the original software, improving the quality of both the original and the 3.x port. Porting isn't always easy, but it's usually easier than writing your own thing from scratch.

How you're supposed to do porting is explained in this Python 2 porting guide. The basic idea is to take the 2.x version of the library and check that all the unit tests still pass without warning when using the -3 command line switch in Python 2. If tests fail or emit warnings, modify the sources and try again (this may require dropping compatibility with older Python versions). Once the code runs without warnings when using the -3 switch, then try running it with Python 3. The best possible case is when this "just works" - code written using modern Python 2 idioms is source compatible with Python 3, so it's possible that the "port" may be complete at this point.

If the tests still fail under Python 3, then the standard library's 2to3 utility can often automatically create a version that will run under Python 3. Alternatively, Armin Ronacher's python-modernize utility instead targets the common subset of Python 2.6+ and either 3.2+ or 3.3+ (depending on the command line options used). (If using the latter, it's important to check the tests still pass under Python 2 as well!)

Either approach makes it feasible to support 2.x and 3.x in parallel from a single 2.x code base. This is much easier than trying to maintain separate 2.x and 3.x branches in parallel (just ask the core Python developers about that one - they've been stuck with doing that for quite a few years now!).

If the tests still fail after automated conversion or modernization, the code may be affected by a semantic

change between Python 2 and 3 that the converters can't handle automatically and that isn't detected by the -3 switch. Such issues should be rare, but may still exist - if one is encountered, it's worth filing a bug against CPython requesting a new -3 warning.

The porting situation is potentially more complicated if there are C extension modules involved and the project isn't using a wrapper generator like Cython, cffi or SWIG that automatically handles the differences between Python 2 and 3, but even then it is still likely to be easier than inventing your own equivalent package. The extension porting guide covers some of the key differences.

There are also some more in depth guides right here on the wiki: PortingPythonToPy3k, PortingExtensionModulesToPy3k

I decided to write something in 3.x but now someone wants to use it who only has 2.x. What do I do?

In addition to the 2to3 tool that allows 3.x code to be generated from 2.x source code, there's also the 3to2 tool, which aims to convert 3.x code back to 2.x code. In theory, this should work even better than going the other direction, since 3.x doesn't have as many nasty corner cases for the converter to handle (getting rid of as many of those as possible was one of the main reasons for breaking backward compatibility after all!). However, code which makes heavy use of 3.x only features (such as function annotations or extended tuple unpacking) is unlikely to be converted successfully.

It's probably also fair to say that 3to2 is the road less traveled compared to 2to3 at this stage, so you might come across a few rough edges here and there. However, if you want to write 3.x code, it's definitely an idea worth exploring.

Supporting Python 2 and Python 3 in a common code base

The common subset of Python 2.6+ and Python 3.3+ is quite large - the restoration of u prefix support for unicode literals in Python 3.3 means that semantically correct Python 2.6+ code can be made source compatible with Python 3.3+ while still remaining largely idiomatic Python. The main difference is that some things will need to be imported from different places in order to handle the fact they have different names in Python 2 and Python 3.

Accordingly, the six compatibility package is a key utility for supporting Python 2 and Python 3 in a single code base.

The future compatibility package is still in beta and doesn't support as many versions of Python as six (it only goes back as far as Python 2.6, while six supports Python 2.4), but allows Python 2 compatible code to be written in a style that is closer to idiomatic Python 3 (for example, it includes an actual Python 2 compatible implementation of the Python 3 bytes type, rather than relying on the Python 2.x 8-bit string type that exposes a slightly different API).

Another key thing to identify for standard library modules is if there is a more up to date backport on PyPI that can be used in preference to the 2.x standard library version. The following modules are either PyPI backports, or else the original modules that served as the source of (or inspiration for) standard library additions in Python 2.7 or 3.x:

- unittest2 (Michael Foord, stdlib unittest maintainer, needed mostly for 2.6 support)
- mock (Michael Foord, stdlib unittest.mock maintainer)
- contextlib2 (Nick Coghlan, stdlib contextlib maintainer)
- configparser (Łukasz Langa, stdlib configparser maintainer)
- futures (Alex Grönholm and Brian Quinlan, stdlib concurrent.futures maintainer)
- argparse (Steven Bethard, stdlib argparse maintainer, needed mostly for 2.6 support)
- faulthandler (Victor Stinner, stdlib faulthandler maintainer)
- cdecimal (Stefan Krah, stdlib decimal maintainer)
- ipaddr (Peter Moody, stdlib ipaddress maintainer, Google's IP address manipulation module that inspired the design of the stdlib ipaddress module)
- stats (Steven D'Aprano, stdlib statistics maintainer)
- enum34 (Ethan Furman, stdlib enum maintainer)
- funcsigs (Aaron Iles, backport of function signature objects)
- backports.inspect (Tripp Lilley, backport of additional inspect module changes, based on funcsigs)
- backports.datetime_timestamp (Jason R. Coombs, backport of datetime.timestamp method as a module level function accepting a datetime object)
- backports.pbkdf2 (Christian Heimes, stdlib hashlib maintainer, backport of hashlib.pbkdf2_hmac)
- backports.ssl_match_hostname (Brandom Craig Rhodes and Toshio Kuratomi, backport of ssl.match_hostname)
- backports.lzma (Peter Cock, backport of the lzma wrapper module)
- lzmaffi (Tomer Chachamu, alternate lzma backport that uses cffi for better PyPy JIT compatibility)
- tracemalloc (Victor Stinner, stdlib tracemalloc maintainer)
- pathlib (Antoine Pitrou, stdlib pathlib maintainer)

The advantage of using the backports namespace module is that it clearly indicates when something is a cross-version backport of a standard library feature, and also allows the original module name to be used when appropriate without conflicting with the standard library name.

Some smaller Python 3 additions are available as recipes in the ActiveState Python Cookbook.

* functools.lru_cache (Raymond Hettinger)

The following modules aren't backports, but are cross-version compatible alternatives to key standard library APIs:

- requests (higher level HTTP and HTTPS APIs. requests itself is unlikely to be added to the stdlib for assorted technical reasons, but an equivalent client API based on asyncio is a plausible future addition)
- regex (an alternative regular expression engine with in principle approval for eventual stdlib inclusion, but requires a PEP to work out the details of the incorporation)
- lxml.etree (alternative implementation of the ElementTree XML API)

In addition to the above modules that also support Python 2, the asyncio module added to the standard library in Python 3.4 was originally developed as a PyPI module for Python 3.3:

- asyncio (Guido van Rossum, BDFL and stdlib asyncio maintainer, requires "yield from" syntax added in Python 3.3)