

Search



- » [PortingToPy3k](#)
- » [BilingualQuickRef](#)
- » [BilingualQuickRef](#)
- » [FrontPage](#)
- » [RecentChanges](#)
- » [FindPage](#)
- » [HelpContents](#)
- » [BilingualQuickRef](#)

Page

- » Immutable Page
- » [Info](#)
- » [Attachments](#)
- »

User

- » [Login](#)

Writing code that runs under both Python2 and 3

Contents



1. [Writing code that runs under both Python2 and 3](#)
 1. [Before you start](#)
 2. [Pure Python Source](#)
 1. [Basic compatibility](#)
 2. [built-ins](#)
 3. [New Style Classes](#)
 4. [codecs](#)
 5. [dictionaries](#)
 6. [doctests](#)
 7. [gettext](#)
 8. [iterators](#)
 9. [metaclasses](#)
 10. [operators](#)
 11. [raise](#)
 12. [strings/bytes/unicodes](#)
 13. [subprocess](#)
 14. [zope.interfaces](#)
 3. [Python extension modules](#)
 1. [Compatibility macros](#)
 2. [C types](#)
 3. [PyArg_Parse\(\)](#)
 4. [PyCObject](#)
 5. [reprs](#)
 4. [Third party packages](#)
 5. [Resources](#)

The intent of *this* page is to provide specific guidelines in a quick reference format for writing code that is compatible with both Python2 and Python3. The idea is that you can check this single page once you're familiar with the basic concepts and approaches but need a refresher on a specific coding techniques.

Most entries here will link to a more in-depth explanation of the basic recipe given here in case you need more than a simple refresher on the subject. At the bottom of this page, you will find various resources for diving more into aspects of supporting Python 3, from the pure-Python, C extension module, packaging, and other perspectives.

Before you start

Here are recommendations for you to follow before you start porting.

- » Identify the minimum python versions you can target: Python 3.3 and 2.7 provide significant forward and backwards compatibility compared to their respective predecessors.. If you have to, targeting 3.2 and 2.6 is still possible. Ignore anything older than that if at all possible (it gets much harder to be bilingual the farther back you go).
- » Try to avoid the production use of  [2to3](#). 2to3 is a pretty slow tool so it can impede the speed with which you develop your code. However you may find it helpful to use 2to3 in *report-only* (aka non-overwrite) mode to get a good sense of what changes will be necessary for your package. Generally, a single-source approach is the easiest for bilingual (i.e. Py2 and Py3) Python apps.
- » Use the third party  [six](#) module sparingly (i.e. only if necessary). One good use case is the `reraise()` method.

- » Even if you can't complete the port to Python 3 because of your dependencies, start by modernizing your Python 2, getting it working cleanly with `python2.7 -3` first.
- » **Clarify your data model:** what are bytes (binary data) and what are strings (text)? If you can't answer this, you are in for a world of hurt (but you probably are already in Python 2).

I cannot overemphasize the last point. Without a clear separation in your mind and data model between bytes and strings, your port will likely be much more painful than it needs to be. This is the biggest distinction between Python 2 and Python 3. Where Python 2 let you be sloppy, with its 8-bit strings that served as both data and ASCII strings, with automatic (but error prone) conversions between 8-bit strings and unicondes, in Python 3 there are only bytes and strings (i.e. unicondes), with no automatic conversion between the two. This is A Good Thing.





Pure Python Source

Basic compatibility



Put the following at the top of all your Python files:

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

This turns on some important compatibility flags.


- » Absolute imports are the default in Python 3  [\[more info\]](#)
- » Enables non-truncating division (dividing two integers results in a float.) Use `//` for the old floor division behaviour.  [\[more info\]](#)
- » `print()` is a function in Python 3  [\[more info\]](#)
- » Unadorned string literals are unicode type in Python 3  [\[more info\]](#)

In your code, make these changes:

- » Change all your `print` statements to use `print()` functions, and remove all the `u''` prefixes from your strings.
- » If you have string literals in your code that represent data, prefix them all with the `b''` prefix  [\[more info\]](#)
- » Remove all `L` suffixes from your long integers.  [\[more info\]](#)


Commentary: Some folks don't like to import `unicode_literals` because it has the potential to change your API (e.g. possibly returning unicondes where before it returned 8-bit strings in Python 2). One developer recommends making this change in your tests only, or only doing this if you have exceptionally good test coverage. YMMV.

built-ins

- » Change usage of `xrange()` to `range()`.  [\[more info\]](#)
- » When mocking builtins, remember that in Python 3, you need to use the `builtins` module instead of the `__builtin__` module, e.g.

```
def mock_it(builtin_name):
    name = ('builtins.%s' if sys.version_info >= (3,) else '__builtin__.%s')
    % builtin_name
    return mock.patch(name)
```

 [\[more info\]](#)

- » Be explicit in the mode you `open()` files in; don't assume they will be opened in binary or text mode, pass in the `t` or `b` flag as appropriate.  [\[more info\]](#) (link also contains other good advice for opening files compatibly).

New Style Classes

In Python 2, you have classic classes and new style classes, whereas in Python 3 you have only the latter. One way of declaring a class to be new style in Python 2 is to inherit from built-in `object`. Such classes will also be new style in Python 3, but with the added cruft of an unnecessary base class. A better way is to add the following to the top of the modules containing your new style classes:

```
__metaclass__ = type
```

As above, this will have no effect in Python 3, but it's much easier to remove one line of cruft than a ton of useless subclasses. Of course, in Python 2, this will make all the classes in that module new style, but since you'll have to use new style classes in Python 3 anyway, and because new style classes are better, just go ahead and covert all your classes now.

See below for how to define classes with different metaclasses than the default.

codecs

- » Python 2 codecs which do str-to-str conversions (e.g. *rot-13*) do not work in Python 3. Use this instead:

```
from codecs import getencoder
encoder = getencoder('rot-13')
rot13string = encoder(mystring)[0]
```

 [\[more info\]](#)

- » Writing unicode safely against `LC_ALL=C` to stdout:

```
sys.stdout = io.TextIOWrapper(sys.stdout.detach(), encoding='UTF-8',
line_buffering=True)
```

dictionaries

- » Change all your uses of the dictionary methods `iteritems()`, `iterkeys()`, and `itervalues()` to use the non-iter variety, e.g. `items()`, `keys()`, and `values()` respectively. These return dictionary views in

Python 3, not concrete lists, so if you need a concrete list, wrap these calls in `list()` or `sorted()`.

 [\[more info\]](#)

doctests

- » In your doctest's `setUp()`, add these globals to your test object's `globals` so they'll have the same `__future__` environment that your code has:

```
from __future__ import absolute_import, print_function, unicode_literals
def setUp(testobj):
    testobj.globals['absolute_import'] = absolute_import
    testobj.globals['print_function'] = print_function
    testobj.globals['unicode_literals'] = unicode_literals
```

 [\[more info\]](#)

- » Bytes have different reprs in Python 2 and Python 3. This convenience function is used to print bytes objects in cross-compatible ways:


```
def print_bytes(obj)
    if bytes is not str:
        obj = repr(obj)[2:-1]
    print(obj)
```

gettext


- » `gettext.install()` only takes `unicode=True` in Python 2; this keyword is absent and unnecessary in Python 3 (Python 3 always returns unicodes). A compatible call would be:

```
kwargs = {}
if sys.version_info >= (3,):
    kwargs['unicode'] = True
gettext.install(domain, LOCALEDIR, **kwargs)
```

iterators

- » Change your iterator classes from providing a `next()` method to providing a `__next__()` method. For cross-compatibility, in your class, set `next = __next__`.  [\[more info\]](#)
- » Use `itertools.zip_longest()` in Python 3, with a conditional import for `itertools.izip_longest()` in Python 2.

metaclasses

- » Syntax for creating instances with different metaclasses is very different between Python 2 and 3. Use the ability to call `type` instances as a way to portably create such instances. For example (from the  [fluf.enum](#) package):

```
# Define the Enum class using metaclass syntax compatible with both Python
2
# and Python 3.
Enum = EnumMetaClass(str('Enum'), (), {
    '__doc__': 'The public API Enum class.',
})
```

Here `EnumMetaClass` is the metaclass (duh!) and `Enum` is the class you're creating which has the custom metaclass. You pass in the base classes (of which there are none, hence the empty tuple) and the dictionary of attributes for the class you're creating. The use of `str()` here is a bit odd, but it's because the `enum` module uses `from __future__ import unicode_literals` for Python 2, but in Python 2, class names must be str/bytes. We can't use the `b''` prefix though because in Python 3, class names must be unicodes.

operators

» Python 3 has no `operator.isSequenceType()`. Use the following code for cross-compatibility.

```
from collections import Sequence
return isinstance(obj, Sequence)
```

raise

» Three argument `raise` is simply not source compatible between Python 2 and 3 (the Python 2 form is a `SyntaxError` in Python 3). You'll need to use `exec()` to work around this, or use `six.reraise()` (which essentially hides the nasty `exec()` from you!). [🌐 \[more info\]](#)

strings/bytes/unicodes

» bytes objects in Python 3 have no `.format()` method. Use concatenation instead.
 » For raw-bytes objects, use the `br''` string prefix (`rb''` was added to Python 3.3)

subprocess

» Consider passing `universal_newlines=True` to `subprocess.Popen()` and friends to get text output directly.

zope.interfaces

» The `implements()` method does not work in Python 3. Use the `@implementer` class decorator instead. [🌐 \[more info\]](#)

Python extension modules

» Define a `PY3` macro which you can later `#ifdef` on for C code which cannot be written portably for both Python 2 and Python 3. [🌐 \[example\]](#)

Compatibility macros

- » The `PyInt_*` functions are gone in Python 3. In your extension module, change all of these to `PyLong_*` functions, which will work in both versions. [🌐 \[more info\]](#)
- » `#include <bytesobject.h>` and change all `PyString_*` functions with their `PyBytes_*` equivalents, changing those that really operate on unicodes to use `PyUnicode_*` functions. [🌐 \[more info\]](#)
- » Instead of explicitly dereferencing `ob_type`, use the `Py_TYPE()` macro instead. [🌐 \[more info\]](#)

C types

There are lots of differences you need to be aware of when defining types in C extensions. A few important ones:

- » Use `PyVarObject_HEAD_INIT()` and don't define the `tp_size` slot [🌐 \[more info\]](#)
- » Remove references to `Py_TPFLAGS_HAVE_WEAKREFS` and `Py_TPFLAGS_HAVE_ITER` since these are unnecessary (and undefined) in Python 3. If you need to support both Python 2 and 3, you'll need an `#ifdef`.

PyArg_Parse()

- » `PyArg_Parse()` and friends lack a `y` code (for bytes objects) in Python 2, so you will have to `#ifdef` around these.
- » In Python 3, there's no equivalent of the `z` code for bytes objects (accepting `None` as well). Write an `o&` converter.

PyCObject

- » Rewrite these to use `PyCapsule` instead. If you can drop Python 2.6, there's no need to `#ifdef` these, since `PyCapsule` is available in Python 2.7. [🌐 \[example\]](#)

reprs

- » If you derive new types from builtin C types, e.g. `PyBytes`, and you want to override the `repr` in the subclass, you'll find you have a problem with cross-compatibility. In Python 2, the super class's `repr` will return bytes (a.k.a. 8-bit strings) while in Python 3, they will return unicodes. Python's C API has a little known format code `%v` which can be used to bridge this gap. Add this macro:

```
#define REPRV(obj) \
    (PyUnicode_Check(obj) ? (obj) : NULL), \
    (PyUnicode_Check(obj) ? NULL : PyBytes_AS_STRING(obj))
```

and use it like this:

```
return PyUnicode_FromFormat("...%V...", REPRV(parent_repr));
```

[🌐 \[more info\]](#)

Third party packages

- » If you're using python-apt, you *must* port entirely to the 0.8 API. python-apt tolerated this under Python 2, but the old API is compiled out under Python 3. `/usr/share/python-apt/migrate-0.8.py` may be of some partial help.
- » pyflakes doesn't like it when you define methods twice depending on `sys.version`. You may want to add some excludes for these.
- » python-libxml2 is currently unavailable for Python 3. Consider using something available in both Python 2 and 3 from the stdlib instead, e.g. `xml.etree.cElementTree`. If necessary, port to `python{,3}-lxml`.
- » for gstreamer advice seems to be to use v1.0 which works well with pygi. See the migration guide at [Novacut/GStreamer1.0](#)
- » If your application is using OAuth, [port your code](#) from the old, unmaintained, OAuth1-only, Python 2-only `oauth` library (i.e. `python-oauth`) to the new, maintained, OAuth1 and OAuth2 compatible, Python 2 and Python 3 compatible `oauthlib` (i.e. `python-oauthlib`) package. Porting is not so hard, although some of the terminology updates in the OAuth spec since the original `oauth` was released make it a bit trickier.

Resources

- » [#python3 IRC channel on Freenode](#)
- » [python-porting mailing list](#)
- » [Debian Python packaging style guide \(covers Python 2 and Python 3\)](#)
- » [Python community portal to Python 3](#)
- » Lennart Regebro's **excellent** [in-depth Python 3 porting guide](#) - which you can [contribute to](#)
- » [Python 3 "Wall of Shame"](#)
- » [Common stumbling blocks when porting to Python 3](#)
- » [Cheeseshop packages explicitly claiming Python 3 support](#)
- » [Python wiki porting guide \(pure-Python\)](#)
- » [Python wiki porting guide \(extension modules\)](#)
- » Barry Warsaw's blog
 - » [Debian packaging for Python 2 and 3](#)
 - » [Python 3 porting \(part 1\)](#)
 - » [Python 3 porting \(part 2\)](#)
 - » [Python 3 plans for Ubuntu 12.10 Quantal Quetzal](#)
 - » [Python 3 plans for Ubuntu 12.04 Precise Pangolin](#)
- » Ned Batchelder's Pycon 2012 talk [Pragmatic Unicode, or How Do I Stop the Pain?](#) **Watch this NOW**
- » Armin Ronacher's [Guide to Unicode](#)

- » [MoinMoin Powered](#)
- » [Python Powered](#)
- » [GPL licensed](#)
- » [Valid HTML 4.01](#)

[Unable to edit the page? See the FrontPage for instructions.](#)