# Python-Future Documentation

Python Charmers

Nov 23, 2020

# CONTENTS

`python-future` is the missing compatibility layer between Python 2 and Python 3. It allows you to use a single, clean Python 3.x-compatible codebase to support both Python 2 and Python 3 with minimal overhead.

CHAPTER

ONE

# CONTENTS:

## 1.1 What's New

### 1.1.1 What's new in version 0.18.2 (2019-10-30)

This is a minor bug-fix release containing a number of fixes:

- Fix min/max functions with generators, and 'None' default (PR #514)
- Use BaseException in raise_() (PR #515)
- Fix builtins.round() for Decimals (Issue #501)
- Fix raise_from() to prevent failures with immutable classes (PR #518)
- Make FixInput idempotent (Issue #427)
- Fix type in newround (PR #521)
- Support mimetype guessing in urllib2 for Py3.8+ (Issue #508)

Python 3.8 is not yet officially supported.

### 1.1.2 What's new in version 0.18.1 (2019-10-09)

This is a minor bug-fix release containing a fix for raise_() when passed an exception that's not an Exception (e.g. BaseException subclasses)

### 1.1.3 What's new in version 0.18.0 (2019-10-09)

This is a major bug-fix and feature release, including:

- Fix collections.abc import for py38+
- Remove import for isnewbytes() function, reducing CPU cost significantly
- Fix bug with importing past.translation when importing past which breaks zipped python installations
- Fix an issue with copyreg import under Py3 that results in unexposed stdlib functionality

- Export and document types in future.utils
- Update behavior of newstr.__eq__() to match str.__eq__() as per reference docs
- Fix raising and the raising fixer to handle cases where the syntax is ambigious
- Allow "default" parameter in min() and max() (Issue #334)
- Implement __hash__() in newstr (Issue #454)
- Future proof some version checks to handle the fact that Py4 won't be a major breaking release
- Fix urllib.request imports for Python 3.8 compatibility (Issue #447)
- Fix future import ordering (Issue #445)
- Fixed bug in fix_division_safe fixture (Issue #434)
- Do not globally destroy re.ASCII in PY3
- Fix a bug in email.Message.set_boundary() (Issue #429)
- Implement format_map() in str
- Implement readinto() for socket.fp

As well as a number of corrections to a variety of documentation, and updates to test infrastructure.

### 1.1.4 What's new in version 0.17.1 (2018-10-30)

This release address a packaging error because of an erroneous declaration that any built wheels are universal.

### 1.1.5 What's new in version 0.17.0 (2018-10-19)

This is a major bug-fix release, including:

- Fix `from collections import ChainMap` after install_aliases() (issue #226)
- Fix multiple import from `__future__` bug in futurize (issue #113)
- Add support for proper %s formatting of newbytes
- Properly implement iterator protocol for newrange object
- Fix `past.translation` on read-only file systems
- Fix Tkinter import bug introduced in Python 2.7.4 (issue #262)
- Correct TypeError to ValueError in a specific edge case for newrange
- Support inequality tests betwen newstrs and newbytes
- Add type check to __get__ in newsuper

- Fix fix_divsion_safe to support better conversion of complex expressions, and skip obvious float division.

As well as a number of corrections to a variety of documentation, and updates to test infrastructure.

### 1.1.6 What's new in version 0.16.0 (2016-10-27)

This release removes the `configparser` package as an alias for `ConfigParser` on Py2 to improve compatibility with the backported *configparser package <https://pypi.org/project/configparser/>*. Previously `python-future` and the PyPI `configparser` backport clashed, causing various compatibility issues. (Issues #118, #181)

If your code previously relied on `configparser` being supplied by `python-future`, the recommended upgrade path is to run `pip install configparser` or add `configparser` to your `requirements.txt` file.

Note that, if you are upgrading `future` with `pip`, you may need to uninstall the old version of future or manually remove the `site-packages/future-0.15.2-py2.7.egg` folder for this change to take effect on your system.

This releases also fixes these bugs:

- Fix `newbytes` constructor bug. (Issue #171)
- Fix semantics of `bool()` with `newobject`. (Issue #211)
- Fix `standard_library.install_aliases()` on PyPy. (Issue #205)
- Fix assertRaises for `pow` and `compile`` on Python 3.5. (Issue #183)
- Fix return argument of `future.utils.ensure_new_type` if conversion to new type does not exist. (Issue #185)
- Add missing `cmp_to_key` for Py2.6. (Issue #189)
- Allow the `old_div` fixer to be disabled. (Issue #190)
- Improve compatibility with Google App Engine. (Issue #231)
- Add some missing imports to the `tkinter` and `tkinter.filedialog` package namespaces. (Issues #212 and #233)
- More complete implementation of `raise_from` on PY3. (Issues #141, #213 and #235, fix provided by Varriount)

### 1.1.7 What's new in version 0.15.2 (2015-09-11)

This is a minor bug-fix release:

- Fix `socket.create_connection()` backport on Py2.6 (issue #162)
- Add more tests of `urllib.request` etc.
- Fix `newsuper()` calls from the `__init__` method of PyQt subclassses (issue #160, thanks to Christopher Arndt)

### 1.1.8 What's new in version 0.15.1 (2015-09-09)

This is a minor bug-fix release:

- Use 3-argument `socket.create_connection()` backport to restore Py2.6 compatibility in `urllib.request.urlopen()` (issue #162)
- Remove breakpoint in `future.backports.http.client` triggered on certain data (issue #164)
- Move `exec` fixer to stage 1 of `futurize` because the forward-compatible `exec(a, b)` idiom is supported in Python 2.6 and 2.7. See https://docs.python.org/2/reference/simple_stmts.html#exec.

### 1.1.9 What's new in version 0.15.0 (2015-07-25)

This release fixes compatibility bugs with CherryPy's Py2/3 compat layer and the latest version of the `urllib3` package. It also adds some additional backports for Py2.6 and Py2.7 from Py3.4's standard library.

New features:

- `install_aliases()` now exposes full backports of the Py3 urllib submodules (`parse`, `request` etc.) from `future.backports.urllib` as submodules of `urllib` on Py2. This implies, for example, that `urllib.parse.unquote` now takes an optional encoding argument as it does on Py3. This improves compatibility with CherryPy's Py2/3 compat layer (issue #158).
- `tkinter.ttk` support (issue #151)
- Backport of `collections.ChainMap` (issue #150)
- Backport of `itertools.count` for Py2.6 (issue #152)
- Enable and document support for the `surrogateescape` error handler for `newstr` and `newbytes` objects on Py2.x (issue #116). This feature is currently in alpha.
- Add constants to `http.client` such as `HTTP_PORT` and `BAD_REQUEST` (issue #137)
- Backport of `reprlib.recursive_repr` to Py2

Bug fixes:

- Add `HTTPMessage` to `http.client`, which is missing from `httplib.__all__` on Python <= 2.7.10. This restores compatibility with the latest `urllib3` package (issue #159, thanks to Waldemar Kornewald)
- Expand newint.__divmod__ and newint.__rdivmod__ to fall back to <type 'long'> implementations where appropriate (issue #146 - thanks to Matt Bogosian)
- Fix newrange slicing for some slice/range combos (issue #132, thanks to Brad Walker)
- Small doc fixes (thanks to Michael Joseph and Tim Tröndle)
- Improve robustness of test suite against opening .pyc files as text on Py2
- Update backports of `Counter` and `OrderedDict` to use the newer implementations from Py3.4. This fixes `.copy()` preserving subclasses etc.
- `futurize` no longer breaks working Py2 code by changing `basestring` to `str`. Instead it imports the `basestring` forward-port from `past.builtins` (issues #127 and #156)
- `future.utils`: add `string_types` etc. and update docs (issue #126)

### 1.1.10 Previous versions

See *Changes in previous versions* for versions prior to v0.15.

## 1.2 Overview: Easy, clean, reliable Python 2/3 compatibility

`python-future` is the missing compatibility layer between Python 2 and Python 3. It allows you to use a single, clean Python 3.x-compatible codebase to support both Python 2 and Python 3 with minimal overhead.

It provides `future` and `past` packages with backports and forward ports of features from Python 3 and 2. It also comes with `futurize` and `pasteurize`, customized 2to3-based scripts that helps you to convert either Py2 or Py3 code easily to support both Python 2 and 3 in a single clean Py3-style codebase, module by module.

Notable projects that use `python-future` for Python 2/3 compatibility are Mezzanine and ObsPy.

### 1.2.1 Features

- `future.builtins` package (also available as `builtins` on Py2) provides backports and remappings for 20 builtins with different semantics on Py3 versus Py2
- support for directly importing 30 standard library modules under their Python 3 names on Py2
- support for importing the other 14 refactored standard library modules under their Py3 names relatively cleanly via `future.standard_library` and `future.moves`
- `past.builtins` package provides forward-ports of 19 Python 2 types and builtin functions. These can aid with per-module code migrations.
- `past.translation` package supports transparent translation of Python 2 modules to Python 3 upon import. [This feature is currently in alpha.]
- 1000+ unit tests, including many from the Py3.3 source tree.
- `futurize` and `pasteurize` scripts based on `2to3` and parts of `3to2` and `python-modernize`, for automatic conversion from either Py2 or Py3 to a clean single-source codebase compatible with Python 2.6+ and Python 3.3+.
- a curated set of utility functions and decorators in `future.utils` and `past.utils` selected from Py2/3 compatibility interfaces from projects like `six`, `IPython`, `Jinja2`, `Django`, and `Pandas`.
- support for the `surrogateescape` error handler when encoding and decoding the backported `str` and `bytes` objects. [This feature is currently in alpha.]
- support for pre-commit hooks

### 1.2.2 Code examples

Replacements for Py2's built-in functions and types are designed to be imported at the top of each Python module together with Python's built-in `__future__` statements. For example, this code behaves identically on Python 2.6/2.7 after these imports as it does on Python 3.3+:

```
from __future__ import absolute_import, division, print_function
from builtins import (bytes, str, open, super, range,
                      zip, round, input, int, pow, object)

# Backported Py3 bytes object
b = bytes(b'ABCD')
assert list(b) == [65, 66, 67, 68]
assert repr(b) == "b'ABCD'"
# These raise TypeErrors:
# b + u'EFGH'
# bytes(b',').join([u'Fred', u'Bill'])

# Backported Py3 str object
s = str(u'ABCD')
```

(continues on next page)

(continued from previous page)

```
assert s != bytes(b'ABCD')
assert isinstance(s.encode('utf-8'), bytes)
assert isinstance(b.decode('utf-8'), str)
assert repr(s) == "'ABCD'"      # consistent repr with Py3 (no u␣
↪prefix)
# These raise TypeErrors:
# bytes(b'B') in s
# s.find(bytes(b'A'))

# Extra arguments for the open() function
f = open('japanese.txt', encoding='utf-8', errors='replace')

# New zero-argument super() function:
class VerboseList(list):
    def append(self, item):
        print('Adding an item')
        super().append(item)

# New iterable range object with slicing support
for i in range(10**15)[:10]:
    pass

# Other iterators: map, zip, filter
my_iter = zip(range(3), ['a', 'b', 'c'])
assert my_iter != list(my_iter)

# The round() function behaves as it does in Python 3, using
# "Banker's Rounding" to the nearest even last digit:
assert round(0.1250, 2) == 0.12

# input() replaces Py2's raw_input() (with no eval()):
name = input('What is your name? ')
print('Hello ' + name)

# pow() supports fractional exponents of negative numbers like in␣
↪Py3:
z = pow(-1, 0.5)

# Compatible output from isinstance() across Py2/3:
assert isinstance(2**64, int)        # long integers
assert isinstance(u'blah', str)
assert isinstance('blah', str)       # only if unicode_literals is␣
↪in effect

# Py3-style iterators written as new-style classes (subclasses of
# future.types.newobject) are automatically backward compatible␣
↪with Py2:
class Upper(object):
    def __init__(self, iterable):
```

(continues on next page)

(continued from previous page)

```
        self._iter = iter(iterable)
    def __next__(self):                 # note the Py3 interface
        return next(self._iter).upper()
    def __iter__(self):
        return self
assert list(Upper('hello')) == list('HELLO')
```

There is also support for renamed standard library modules. The recommended interface works like this:

```
# Many Py3 module names are supported directly on both Py2.x and 3.
↪x:
from http.client import HttpConnection
import html.parser
import queue
import xmlrpc.client

# Refactored modules with clashing names on Py2 and Py3 are␣
↪supported
# as follows:
from future import standard_library
standard_library.install_aliases()

# Then, for example:
from itertools import filterfalse, zip_longest
from urllib.request import urlopen
from collections import ChainMap
from collections import UserDict, UserList, UserString
from subprocess import getoutput, getstatusoutput
from collections import Counter, OrderedDict   # backported to Py2.6
```

### 1.2.3 Automatic conversion to Py2/3-compatible code

`python-future` comes with two scripts called `futurize` and `pasteurize` to aid in making Python 2 code or Python 3 code compatible with both platforms (Py2/3). It is based on 2to3 and uses fixers from `lib2to3`, `lib3to2`, and `python-modernize`, as well as custom fixers.

`futurize` passes Python 2 code through all the appropriate fixers to turn it into valid Python 3 code, and then adds `__future__` and `future` package imports so that it also runs under Python 2.

For conversions from Python 3 code to Py2/3, use the `pasteurize` script instead. This converts Py3-only constructs (e.g. new metaclass syntax) to Py2/3 compatible constructs and adds `__future__` and `future` imports to the top of each module.

In both cases, the result should be relatively clean Py3-style code that runs mostly unchanged on both Python 2 and Python 3.

#### Futurize: 2 to both

For example, running `futurize -w mymodule.py` turns this Python 2 code:

```
import Queue
from urllib2 import urlopen

def greet(name):
    print 'Hello',
    print name

print "What's your name?",
name = raw_input()
greet(name)
```

into this code which runs on both Py2 and Py3:

```
from __future__ import print_function
from future import standard_library
standard_library.install_aliases()
from builtins import input
import queue
from urllib.request import urlopen

def greet(name):
    print('Hello', end=' ')
    print(name)

print("What's your name?", end=' ')
name = input()
greet(name)
```

See forwards-conversion and backwards-conversion for more details.

### 1.2.4 Automatic translation

The `past` package can automatically translate some simple Python 2 modules to Python 3 upon import. The goal is to support the "long tail" of real-world Python 2 modules (e.g. on PyPI) that have not been ported yet. For example, here is how to use a Python 2-only package called `plotrique` on Python 3. First install it:

```
$ pip3 install plotrique==0.2.5-7 --no-compile   # to ignore␣
↪SyntaxErrors
```

(or use `pip` if this points to your Py3 environment.)

Then pass a whitelist of module name prefixes to the `autotranslate()` function. Example:

```
$ python3
```

(continues on next page)

(continued from previous page)

```
>>> from past.translation import autotranslate
>>> autotranslate(['plotrique'])
>>> import plotrique
```

This transparently translates and runs the `plotrique` module and any submodules in the `plotrique` package that `plotrique` imports.

This is intended to help you migrate to Python 3 without the need for all your code's dependencies to support Python 3 yet. It should be used as a last resort; ideally Python 2-only dependencies should be ported properly to a Python 2/3 compatible codebase using a tool like `futurize` and the changes should be pushed to the upstream project.

Note: the auto-translation feature is still in alpha; it needs more testing and development, and will likely never be perfect.

For more info, see translation.

### 1.2.5 Pre-commit hooks

Pre-commit is a framework for managing and maintaining multi-language pre-commit hooks.

In case you need to port your project from Python 2 to Python 3, you might consider using such hook during the transition period.

First:

```
$ pip install pre-commit
```

and then in your project's directory:

```
$ pre-commit install
```

Next, you need to add this entry to your `.pre-commit-config.yaml`

```
-   repo: https://github.com/PythonCharmers/python-future
    rev: master
    hooks:
        - id: futurize
          args: [--both-stages]
```

The `args` part is optional, by default only stage1 is applied.

### 1.2.6 Licensing

**Author** Ed Schofield, Jordan M. Adler, et al



**Sponsors** Python Charmers Pty Ltd, Australia, and Python Charmers Pte Ltd, Singapore. http://pythoncharmers.com

Pinterest https://opensource.pinterest.com/

**Licence** MIT. See `LICENSE.txt` or here.

**Other credits** See here.

### 1.2.7 Next steps

If you are new to Python-Future, check out the Quickstart Guide.

For an update on changes in the latest version, see the What's New page.

## 1.3 Quick-start guide

You can use `future` to help to port your code from Python 2 to Python 3 today – and still have it run on Python 2.

If you already have Python 3 code, you can instead use `future` to offer Python 2 compatibility with almost no extra work.

### 1.3.1 Installation

To install the latest stable version, type:

```
pip install future
```

If you would prefer the latest development version, it is available here.

### 1.3.2 If you are writing code from scratch

The easiest way is to start each new module with these lines:

```
from __future__ import (absolute_import, division,
                        print_function, unicode_literals)
from builtins import *
```

Then write standard Python 3 code. The `future` package will provide support for running your code on Python 2.7, and 3.4+ mostly unchanged.

- For explicit import forms, see *Explicit imports*.
- For more details, see *What else you need to know*.
- For a cheat sheet, see *Cheat Sheet: Writing Python 2-3 compatible code*.

### 1.3.3 To convert existing Python 3 code

To offer backward compatibility with Python 2 from your Python 3 code, you can use the `pasteurize` script. This adds these lines at the top of each module:

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

from builtins import open
from builtins import str
# etc., as needed

from future import standard_library
standard_library.install_aliases()
```

and converts several Python 3-only constructs (like keyword-only arguments) to a form compatible with both Py3 and Py2. Most remaining Python 3 code should simply work on Python 2.

See backwards-conversion for more details.

### 1.3.4 To convert existing Python 2 code

The `futurize` script passes Python 2 code through all the appropriate fixers to turn it into valid Python 3 code, and then adds `__future__` and `future` package imports to re-enable compatibility with Python 2.

For example, running `futurize` turns this Python 2 code:

```
import ConfigParser                    # Py2 module name

class Upper(object):
    def __init__(self, iterable):
        self._iter = iter(iterable)
    def next(self):                    # Py2-style iterator interface
        return next(self._iter).upper()
    def __iter__(self):
        return self

itr = Upper('hello')
print next(itr),
```

(continues on next page)

(continued from previous page)

```
for letter in itr:
    print letter,                    # Py2-style print statement
```

into this code which runs on both Py2 and Py3:

```
from __future__ import print_function
from future import standard_library
standard_library.install_aliases()
from future.builtins import next
from future.builtins import object
import configparser                  # Py3-style import

class Upper(object):
    def __init__(self, iterable):
        self._iter = iter(iterable)
    def __next__(self):              # Py3-style iterator interface
        return next(self._iter).upper()
    def __iter__(self):
        return self

itr = Upper('hello')
print(next(itr), end=' ')            # Py3-style print function
for letter in itr:
    print(letter, end=' ')
```

To write out all the changes to your Python files that `futurize` suggests, use the `-w` flag.

For complex projects, it is probably best to divide the porting into two stages. Stage 1 is for "safe" changes that modernize the code but do not break Python 2.7 compatibility or introduce a dependency on the `future` package. Stage 2 is to complete the process.

See forwards-conversion-stage1 and forwards-conversion-stage2 for more details.

### 1.3.5 Standard library reorganization

`future` supports the standard library reorganization (PEP 3108) via one of several mechanisms, allowing most moved standard library modules to be accessed under their Python 3 names and locations in Python 2:

```
from future import standard_library
standard_library.install_aliases()

# Then these Py3-style imports work on both Python 2 and Python 3:
import socketserver
import queue
from collections import UserDict, UserList, UserString
from collections import ChainMap  # even on Py2.7
from itertools import filterfalse, zip_longest
```

(continues on next page)

(continued from previous page)

```
import html
import html.entities
import html.parser

import http
import http.client
import http.server
import http.cookies
import http.cookiejar

import urllib.request
import urllib.parse
import urllib.response
import urllib.error
import urllib.robotparser

import xmlrpc.client
import xmlrpc.server
```

and others. For a complete list, see direct-imports.

### 1.3.6 Python 2-only dependencies

If you have dependencies that support only Python 2, you may be able to use the `past` module to automatically translate these Python 2 modules to Python 3 upon import. First, install the Python 2-only package into your Python 3 environment:

```
$ pip3 install mypackagename --no-compile   # to ignore SyntaxErrors
```

(or use `pip` if this points to your Py3 environment.)

Then add the following code at the top of your (Py3 or Py2/3-compatible) code:

```
from past.translation import autotranslate
autotranslate(['mypackagename'])
import mypackagename
```

This feature is experimental, and we would appreciate your feedback on how well this works or doesn't work for you. Please file an issue here or post to the python-porting mailing list.

For more information on the automatic translation feature, see translation.

### 1.3.7 Next steps

For more information about writing Py2/3-compatible code, see:

- *Cheat Sheet: Writing Python 2-3 compatible code*
- *What else you need to know*.

## 1.4 Cheat Sheet: Writing Python 2-3 compatible code

- **Copyright (c):** 2013-2019 Python Charmers Pty Ltd, Australia.
- **Author:** Ed Schofield.
- **Licence:** Creative Commons Attribution.

A PDF version is here: http://python-future.org/compatible_idioms.pdf

This notebook shows you idioms for writing future-proof code that is compatible with both versions of Python: 2 and 3. It accompanies Ed Schofield's talk at PyCon AU 2014, "Writing 2/3 compatible code". (The video is here: http://www.youtube.com/watch?v=KOqk8j11aAI&t=10m14s.)

Minimum versions:

- Python 2: 2.7+
- Python 3: 3.4+

### 1.4.1 Setup

The imports below refer to these `pip`-installable packages on PyPI:

```
import future           # pip install future
import builtins         # pip install future
import past             # pip install future
import six              # pip install six
```

The following scripts are also `pip`-installable:

```
futurize                # pip install future
pasteurize              # pip install future
```

See http://python-future.org and https://pythonhosted.org/six/ for more information.

### 1.4.2 Essential syntax differences

#### print

```
# Python 2 only:
print 'Hello'
```

```
# Python 2 and 3:
print('Hello')
```

To print multiple strings, import `print_function` to prevent Py2 from interpreting it as a tuple:

```
# Python 2 only:
print 'Hello', 'Guido'
```

```
# Python 2 and 3:
from __future__ import print_function    # (at top of module)

print('Hello', 'Guido')
```

```
# Python 2 only:
print >> sys.stderr, 'Hello'
```

```
# Python 2 and 3:
from __future__ import print_function

print('Hello', file=sys.stderr)
```

```
# Python 2 only:
print 'Hello',
```

```
# Python 2 and 3:
from __future__ import print_function

print('Hello', end='')
```

#### Raising exceptions

```
# Python 2 only:
raise ValueError, "dodgy value"
```

```
# Python 2 and 3:
raise ValueError("dodgy value")
```

Raising exceptions with a traceback:

```
# Python 2 only:
traceback = sys.exc_info()[2]
raise ValueError, "dodgy value", traceback
```

```
# Python 3 only:
raise ValueError("dodgy value").with_traceback()
```

```
# Python 2 and 3: option 1
from six import reraise as raise_
# or
from future.utils import raise_

traceback = sys.exc_info()[2]
raise_(ValueError, "dodgy value", traceback)
```

```
# Python 2 and 3: option 2
from future.utils import raise_with_traceback

raise_with_traceback(ValueError("dodgy value"))
```

Exception chaining (PEP 3134):

```
# Setup:
class DatabaseError(Exception):
    pass
```

```
# Python 3 only
class FileDatabase:
    def __init__(self, filename):
        try:
            self.file = open(filename)
        except IOError as exc:
            raise DatabaseError('failed to open') from exc
```

```
# Python 2 and 3:
from future.utils import raise_from

class FileDatabase:
    def __init__(self, filename):
        try:
            self.file = open(filename)
        except IOError as exc:
            raise_from(DatabaseError('failed to open'), exc)
```

```
# Testing the above:
try:
    fd = FileDatabase('non_existent_file.txt')
except Exception as e:
```

(continues on next page)

(continued from previous page)

```
    assert isinstance(e.__cause__, IOError)    # FileNotFoundError␣
↪on Py3.3+ inherits from IOError
```

### Catching exceptions

```
# Python 2 only:
try:
    ...
except ValueError, e:
    ...
```

```
# Python 2 and 3:
try:
    ...
except ValueError as e:
    ...
```

### Division

Integer division (rounding down):

```
# Python 2 only:
assert 2 / 3 == 0
```

```
# Python 2 and 3:
assert 2 // 3 == 0
```

"True division" (float division):

```
# Python 3 only:
assert 3 / 2 == 1.5
```

```
# Python 2 and 3:
from __future__ import division    # (at top of module)

assert 3 / 2 == 1.5
```

"Old division" (i.e. compatible with Py2 behaviour):

```
# Python 2 only:
a = b / c              # with any types
```

```
# Python 2 and 3:
from past.utils import old_div

a = old_div(b, c)    # always same as / on Py2
```

### Long integers

Short integers are gone in Python 3 and `long` has become `int` (without the trailing `L` in the `repr`).

```
# Python 2 only
k = 9223372036854775808L

# Python 2 and 3:
k = 9223372036854775808
```

```
# Python 2 only
bigint = 1L

# Python 2 and 3
from builtins import int
bigint = int(1)
```

To test whether a value is an integer (of any kind):

```
# Python 2 only:
if isinstance(x, (int, long)):
    ...

# Python 3 only:
if isinstance(x, int):
    ...

# Python 2 and 3: option 1
from builtins import int    # subclass of long on Py2

if isinstance(x, int):              # matches both int and long on␣
↪Py2
    ...

# Python 2 and 3: option 2
from past.builtins import long

if isinstance(x, (int, long)):
    ...
```

### Octal constants

```
0644     # Python 2 only
```

```
0o644    # Python 2 and 3
```

### Backtick repr

```
`x`      # Python 2 only
```

```
repr(x)  # Python 2 and 3
```

### Metaclasses

```
class BaseForm(object):
    pass

class FormType(type):
    pass
```

```
# Python 2 only:
class Form(BaseForm):
    __metaclass__ = FormType
    pass
```

```
# Python 3 only:
class Form(BaseForm, metaclass=FormType):
    pass
```

```
# Python 2 and 3:
from six import with_metaclass
# or
from future.utils import with_metaclass

class Form(with_metaclass(FormType, BaseForm)):
    pass
```

### 1.4.3 Strings and bytes

#### Unicode (text) string literals

If you are upgrading an existing Python 2 codebase, it may be preferable to mark up all string literals as unicode explicitly with `u` prefixes:

```
# Python 2 only
s1 = 'The Zen of Python'
s2 = u'\n'

# Python 2 and 3
s1 = u'The Zen of Python'
s2 = u'\n'
```

The `futurize` and `python-modernize` tools do not currently offer an option to do this automatically.

If you are writing code for a new project or new codebase, you can use this idiom to make all string literals in a module unicode strings:

```
# Python 2 and 3
from __future__ import unicode_literals    # at top of module

s1 = 'The Zen of Python'
s2 = '\n'
```

See http://python-future.org/unicode_literals.html for more discussion on which style to use.

#### Byte-string literals

```
# Python 2 only
s = 'This must be a byte-string'

# Python 2 and 3
s = b'This must be a byte-string'
```

To loop over a byte-string with possible high-bit characters, obtaining each character as a byte-string of length 1:

```
# Python 2 only:
for bytechar in 'byte-string with high-bit chars like \xf9':
    ...

# Python 3 only:
for myint in b'byte-string with high-bit chars like \xf9':
    bytechar = bytes([myint])

# Python 2 and 3:
```

(continues on next page)

(continued from previous page)

```
from builtins import bytes
for myint in bytes(b'byte-string with high-bit chars like \xf9'):
    bytechar = bytes([myint])
```

As an alternative, `chr()` and `.encode('latin-1')` can be used to convert an int into a 1-char byte string:

```
# Python 3 only:
for myint in b'byte-string with high-bit chars like \xf9':
    char = chr(myint)    # returns a unicode string
    bytechar = char.encode('latin-1')

# Python 2 and 3:
from builtins import bytes, chr
for myint in bytes(b'byte-string with high-bit chars like \xf9'):
    char = chr(myint)    # returns a unicode string
    bytechar = char.encode('latin-1')    # forces returning a byte␣
↪str
```

### basestring

```
# Python 2 only:
a = u'abc'
b = 'def'
assert (isinstance(a, basestring) and isinstance(b, basestring))

# Python 2 and 3: alternative 1
from past.builtins import basestring    # pip install future

a = u'abc'
b = b'def'
assert (isinstance(a, basestring) and isinstance(b, basestring))
```

```
# Python 2 and 3: alternative 2: refactor the code to avoid␣
↪considering
# byte-strings as strings.

from builtins import str
a = u'abc'
b = b'def'
c = b.decode()
assert isinstance(a, str) and isinstance(c, str)
# ...
```

#### unicode

```
# Python 2 only:
templates = [u"blog/blog_post_detail_%s.html" % unicode(slug)]
```

```
# Python 2 and 3: alternative 1
from builtins import str
templates = [u"blog/blog_post_detail_%s.html" % str(slug)]
```

```
# Python 2 and 3: alternative 2
from builtins import str as text
templates = [u"blog/blog_post_detail_%s.html" % text(slug)]
```

#### StringIO

```
# Python 2 only:
from StringIO import StringIO
# or:
from cStringIO import StringIO

# Python 2 and 3:
from io import BytesIO      # for handling byte strings
from io import StringIO     # for handling unicode strings
```

### 1.4.4 Imports relative to a package

Suppose the package is:

```
mypackage/
    __init__.py
    submodule1.py
    submodule2.py
```

and the code below is in `submodule1.py`:

```
# Python 2 only:
import submodule2
```

```
# Python 2 and 3:
from . import submodule2
```

```
# Python 2 and 3:
# To make Py2 code safer (more like Py3) by preventing
# implicit relative imports, you can also add this to the top:
from __future__ import absolute_import
```

### 1.4.5 Dictionaries

```
heights = {'Fred': 175, 'Anne': 166, 'Joe': 192}
```

#### Iterating through `dict` keys/values/items

Iterable dict keys:

```
# Python 2 only:
for key in heights.iterkeys():
    ...
```

```
# Python 2 and 3:
for key in heights:
    ...
```

Iterable dict values:

```
# Python 2 only:
for value in heights.itervalues():
    ...
```

```
# Idiomatic Python 3
for value in heights.values():    # extra memory overhead on Py2
    ...
```

```
# Python 2 and 3: option 1
from builtins import dict

heights = dict(Fred=175, Anne=166, Joe=192)
for key in heights.values():    # efficient on Py2 and Py3
    ...
```

```
# Python 2 and 3: option 2
from future.utils import itervalues
# or
from six import itervalues

for key in itervalues(heights):
    ...
```

Iterable dict items:

```
# Python 2 only:
for (key, value) in heights.iteritems():
    ...
```

```
# Python 2 and 3: option 1
for (key, value) in heights.items():     # inefficient on Py2
    ...
```

```
# Python 2 and 3: option 2
from future.utils import viewitems

for (key, value) in viewitems(heights):    # also behaves like a set
    ...
```

```
# Python 2 and 3: option 3
from future.utils import iteritems
# or
from six import iteritems

for (key, value) in iteritems(heights):
    ...
```

### dict keys/values/items as a list

dict keys as a list:

```
# Python 2 only:
keylist = heights.keys()
assert isinstance(keylist, list)
```

```
# Python 2 and 3:
keylist = list(heights)
assert isinstance(keylist, list)
```

dict values as a list:

```
# Python 2 only:
heights = {'Fred': 175, 'Anne': 166, 'Joe': 192}
valuelist = heights.values()
assert isinstance(valuelist, list)
```

```
# Python 2 and 3: option 1
valuelist = list(heights.values())     # inefficient on Py2
```

```
# Python 2 and 3: option 2
from builtins import dict

heights = dict(Fred=175, Anne=166, Joe=192)
valuelist = list(heights.values())
```

```
# Python 2 and 3: option 3
from future.utils import listvalues

valuelist = listvalues(heights)
```

```
# Python 2 and 3: option 4
from future.utils import itervalues
# or
from six import itervalues

valuelist = list(itervalues(heights))
```

dict items as a list:

```
# Python 2 and 3: option 1
itemlist = list(heights.items())     # inefficient on Py2
```

```
# Python 2 and 3: option 2
from future.utils import listitems

itemlist = listitems(heights)
```

```
# Python 2 and 3: option 3
from future.utils import iteritems
# or
from six import iteritems

itemlist = list(iteritems(heights))
```

### 1.4.6 Custom class behaviour

#### Custom iterators

```
# Python 2 only
class Upper(object):
    def __init__(self, iterable):
        self._iter = iter(iterable)
    def next(self):                 # Py2-style
        return self._iter.next().upper()
    def __iter__(self):
        return self

itr = Upper('hello')
assert itr.next() == 'H'      # Py2-style
assert list(itr) == list('ELLO')
```

```
# Python 2 and 3: option 1
from builtins import object

class Upper(object):
    def __init__(self, iterable):
        self._iter = iter(iterable)
    def __next__(self):       # Py3-style iterator interface
        return next(self._iter).upper()  # builtin next() function␣
↪calls
    def __iter__(self):
        return self

itr = Upper('hello')
assert next(itr) == 'H'       # compatible style
assert list(itr) == list('ELLO')
```

```
# Python 2 and 3: option 2
from future.utils import implements_iterator

@implements_iterator
class Upper(object):
    def __init__(self, iterable):
        self._iter = iter(iterable)
    def __next__(self):                    # Py3-style iterator␣
↪interface
        return next(self._iter).upper()  # builtin next() function␣
↪calls
    def __iter__(self):
        return self

itr = Upper('hello')
assert next(itr) == 'H'
assert list(itr) == list('ELLO')
```

### Custom `__str__` methods

```
# Python 2 only:
class MyClass(object):
    def __unicode__(self):
        return 'Unicode string: \u5b54\u5b50'
    def __str__(self):
        return unicode(self).encode('utf-8')

a = MyClass()
print(a)    # prints encoded string
```

```
# Python 2 and 3:
from future.utils import python_2_unicode_compatible
```

(continues on next page)

(continued from previous page)

```
@python_2_unicode_compatible
class MyClass(object):
    def __str__(self):
        return u'Unicode string: \u5b54\u5b50'

a = MyClass()
print(a)    # prints string encoded as utf-8 on Py2
```

```
Unicode string:
```

#### Custom `__nonzero__` vs `__bool__` method:

```
# Python 2 only:
class AllOrNothing(object):
    def __init__(self, l):
        self.l = l
    def __nonzero__(self):
        return all(self.l)

container = AllOrNothing([0, 100, 200])
assert not bool(container)
```

```
# Python 2 and 3:
from builtins import object

class AllOrNothing(object):
    def __init__(self, l):
        self.l = l
    def __bool__(self):
        return all(self.l)

container = AllOrNothing([0, 100, 200])
assert not bool(container)
```

### 1.4.7 Lists versus iterators

#### xrange

```
# Python 2 only:
for i in xrange(10**8):
    ...
```

```
# Python 2 and 3: forward-compatible
from builtins import range
for i in range(10**8):
    ...
```

```
# Python 2 and 3: backward-compatible
from past.builtins import xrange
for i in xrange(10**8):
    ...
```

### range

```
# Python 2 only
mylist = range(5)
assert mylist == [0, 1, 2, 3, 4]
```

```
# Python 2 and 3: forward-compatible: option 1
mylist = list(range(5))            # copies memory on Py2
assert mylist == [0, 1, 2, 3, 4]
```

```
# Python 2 and 3: forward-compatible: option 2
from builtins import range

mylist = list(range(5))
assert mylist == [0, 1, 2, 3, 4]
```

```
# Python 2 and 3: option 3
from future.utils import lrange

mylist = lrange(5)
assert mylist == [0, 1, 2, 3, 4]
```

```
# Python 2 and 3: backward compatible
from past.builtins import range

mylist = range(5)
assert mylist == [0, 1, 2, 3, 4]
```

### map

```
# Python 2 only:
mynewlist = map(f, myoldlist)
assert mynewlist == [f(x) for x in myoldlist]
```

```
# Python 2 and 3: option 1
# Idiomatic Py3, but inefficient on Py2
mynewlist = list(map(f, myoldlist))
assert mynewlist == [f(x) for x in myoldlist]
```

```
# Python 2 and 3: option 2
from builtins import map

mynewlist = list(map(f, myoldlist))
assert mynewlist == [f(x) for x in myoldlist]
```

```
# Python 2 and 3: option 3
try:
    import itertools.imap as map
except ImportError:
    pass

mynewlist = list(map(f, myoldlist))    # inefficient on Py2
assert mynewlist == [f(x) for x in myoldlist]
```

```
# Python 2 and 3: option 4
from future.utils import lmap

mynewlist = lmap(f, myoldlist)
assert mynewlist == [f(x) for x in myoldlist]
```

```
# Python 2 and 3: option 5
from past.builtins import map

mynewlist = map(f, myoldlist)
assert mynewlist == [f(x) for x in myoldlist]
```

### imap

```
# Python 2 only:
from itertools import imap

myiter = imap(func, myoldlist)
assert isinstance(myiter, iter)
```

```
# Python 3 only:
myiter = map(func, myoldlist)
assert isinstance(myiter, iter)
```

```
# Python 2 and 3: option 1
from builtins import map

myiter = map(func, myoldlist)
assert isinstance(myiter, iter)
```

```
# Python 2 and 3: option 2
try:
    import itertools.imap as map
except ImportError:
    pass

myiter = map(func, myoldlist)
assert isinstance(myiter, iter)
```

```
# Python 2 and 3: option 3
from six.moves import map

myiter = map(func, myoldlist)
assert isinstance(myiter, iter)
```

#### zip, izip

As above with `zip` and `itertools.izip`.

#### filter, ifilter

As above with `filter` and `itertools.ifilter` too.

### 1.4.8 Other builtins

#### File IO with open()

```
# Python 2 only
f = open('myfile.txt')
data = f.read()                # as a byte string
text = data.decode('utf-8')

# Python 2 and 3: alternative 1
from io import open
```

(continues on next page)

(continued from previous page)

```
f = open('myfile.txt', 'rb')
data = f.read()                 # as bytes
text = data.decode('utf-8')   # unicode, not bytes

# Python 2 and 3: alternative 2
from io import open
f = open('myfile.txt', encoding='utf-8')
text = f.read()     # unicode, not bytes
```

### reduce()

```
# Python 2 only:
assert reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) == 1+2+3+4+5
```

```
# Python 2 and 3:
from functools import reduce

assert reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) == 1+2+3+4+5
```

### raw_input()

```
# Python 2 only:
name = raw_input('What is your name? ')
assert isinstance(name, str)     # native str
```

```
# Python 2 and 3:
from builtins import input

name = input('What is your name? ')
assert isinstance(name, str)     # native str on Py2 and Py3
```

### input()

```
# Python 2 only:
input("Type something safe please: ")
```

```
# Python 2 and 3
from builtins import input
eval(input("Type something safe please: "))
```

Warning: using either of these is **unsafe** with untrusted input.

### file()

```
# Python 2 only:
f = file(pathname)
```

```
# Python 2 and 3:
f = open(pathname)

# But preferably, use this:
from io import open
f = open(pathname, 'rb')    # if f.read() should return bytes
# or
f = open(pathname, 'rt')    # if f.read() should return unicode text
```

### exec

```
# Python 2 only:
exec 'x = 10'

# Python 2 and 3:
exec('x = 10')
```

```
# Python 2 only:
g = globals()
exec 'x = 10' in g

# Python 2 and 3:
g = globals()
exec('x = 10', g)
```

```
# Python 2 only:
l = locals()
exec 'x = 10' in g, l

# Python 2 and 3:
exec('x = 10', g, l)
```

### execfile()

```
# Python 2 only:
execfile('myfile.py')
```

```
# Python 2 and 3: alternative 1
from past.builtins import execfile

execfile('myfile.py')
```

```
# Python 2 and 3: alternative 2
exec(compile(open('myfile.py').read()))

# This can sometimes cause this:
#     SyntaxError: function ... uses import * and bare exec ...
# See https://github.com/PythonCharmers/python-future/issues/37
```

### unichr()

```
# Python 2 only:
assert unichr(8364) == '€'
```

```
# Python 3 only:
assert chr(8364) == '€'
```

```
# Python 2 and 3:
from builtins import chr
assert chr(8364) == '€'
```

### intern()

```
# Python 2 only:
intern('mystring')
```

```
# Python 3 only:
from sys import intern
intern('mystring')
```

```
# Python 2 and 3: alternative 1
from past.builtins import intern
intern('mystring')
```

```
# Python 2 and 3: alternative 2
from six.moves import intern
intern('mystring')
```

```
# Python 2 and 3: alternative 3
from future.standard_library import install_aliases
install_aliases()
from sys import intern
intern('mystring')
```

```
# Python 2 and 3: alternative 2
try:
```

(continues on next page)

(continued from previous page)

```
    from sys import intern
except ImportError:
    pass
intern('mystring')
```

### apply()

```
args = ('a', 'b')
kwargs = {'kwarg1': True}
```

```
# Python 2 only:
apply(f, args, kwargs)
```

```
# Python 2 and 3: alternative 1
f(*args, **kwargs)
```

```
# Python 2 and 3: alternative 2
from past.builtins import apply
apply(f, args, kwargs)
```

### chr()

```
# Python 2 only:
assert chr(64) == b'@'
assert chr(200) == b'\xc8'
```

```
# Python 3 only: option 1
assert chr(64).encode('latin-1') == b'@'
assert chr(0xc8).encode('latin-1') == b'\xc8'
```

```
# Python 2 and 3: option 1
from builtins import chr

assert chr(64).encode('latin-1') == b'@'
assert chr(0xc8).encode('latin-1') == b'\xc8'
```

```
# Python 3 only: option 2
assert bytes([64]) == b'@'
assert bytes([0xc8]) == b'\xc8'
```

```
# Python 2 and 3: option 2
from builtins import bytes

```

(continues on next page)

(continued from previous page)

```
assert bytes([64]) == b'@'
assert bytes([0xc8]) == b'\xc8'
```

### cmp()

```
# Python 2 only:
assert cmp('a', 'b') < 0 and cmp('b', 'a') > 0 and cmp('c', 'c') ==␣
↪0
```

```
# Python 2 and 3: alternative 1
from past.builtins import cmp
assert cmp('a', 'b') < 0 and cmp('b', 'a') > 0 and cmp('c', 'c') ==␣
↪0
```

```
# Python 2 and 3: alternative 2
cmp = lambda(x, y): (x > y) - (x < y)
assert cmp('a', 'b') < 0 and cmp('b', 'a') > 0 and cmp('c', 'c') ==␣
↪0
```

### reload()

```
# Python 2 only:
reload(mymodule)
```

```
# Python 2 and 3
from imp import reload
reload(mymodule)
```

## 1.4.9 Standard library

### dbm modules

```
# Python 2 only
import anydbm
import whichdb
import dbm
import dumbdbm
import gdbm

# Python 2 and 3: alternative 1
from future import standard_library
standard_library.install_aliases()
```

(continues on next page)

(continued from previous page)

```
import dbm
import dbm.ndbm
import dbm.dumb
import dbm.gnu

# Python 2 and 3: alternative 2
from future.moves import dbm
from future.moves.dbm import dumb
from future.moves.dbm import ndbm
from future.moves.dbm import gnu

# Python 2 and 3: alternative 3
from six.moves import dbm_gnu
# (others not supported)
```

### commands / subprocess modules

```
# Python 2 only
from commands import getoutput, getstatusoutput

# Python 2 and 3
from future import standard_library
standard_library.install_aliases()

from subprocess import getoutput, getstatusoutput
```

### StringIO module

```
# Python 2 only
from StringIO import StringIO
from cStringIO import StringIO
```

```
# Python 2 and 3
from io import BytesIO
# and refactor StringIO() calls to BytesIO() if passing byte-strings
```

### http module

```
# Python 2 only:
import httplib
import Cookie
import cookielib
import BaseHTTPServer
import SimpleHTTPServer
import CGIHttpServer

# Python 2 and 3 (after ``pip install future``):
import http.client
import http.cookies
import http.cookiejar
import http.server
```

### xmlrpc module

```
# Python 2 only:
import DocXMLRPCServer
import SimpleXMLRPCServer

# Python 2 and 3 (after ``pip install future``):
import xmlrpc.server
```

```
# Python 2 only:
import xmlrpclib

# Python 2 and 3 (after ``pip install future``):
import xmlrpc.client
```

### html escaping and entities

```
# Python 2 and 3:
from cgi import escape

# Safer (Python 2 and 3, after ``pip install future``):
from html import escape

# Python 2 only:
from htmlentitydefs import codepoint2name, entitydefs,␣
↪name2codepoint

# Python 2 and 3 (after ``pip install future``):
from html.entities import codepoint2name, entitydefs, name2codepoint
```

### html parsing

```
# Python 2 only:
from HTMLParser import HTMLParser

# Python 2 and 3 (after ``pip install future``)
from html.parser import HTMLParser

# Python 2 and 3 (alternative 2):
from future.moves.html.parser import HTMLParser
```

### urllib module

`urllib` is the hardest module to use from Python 2/3 compatible code. You might want to switch to Requests (http://python-requests.org) instead.

```
# Python 2 only:
from urlparse import urlparse
from urllib import urlencode
from urllib2 import urlopen, Request, HTTPError
```

```
# Python 3 only:
from urllib.parse import urlparse, urlencode
from urllib.request import urlopen, Request
from urllib.error import HTTPError
```

```
# Python 2 and 3: easiest option
from future.standard_library import install_aliases
install_aliases()

from urllib.parse import urlparse, urlencode
from urllib.request import urlopen, Request
from urllib.error import HTTPError
```

```
# Python 2 and 3: alternative 2
from future.standard_library import hooks

with hooks():
    from urllib.parse import urlparse, urlencode
    from urllib.request import urlopen, Request
    from urllib.error import HTTPError
```

```
# Python 2 and 3: alternative 3
from future.moves.urllib.parse import urlparse, urlencode
from future.moves.urllib.request import urlopen, Request
from future.moves.urllib.error import HTTPError
# or
```

(continues on next page)

(continued from previous page)

```
from six.moves.urllib.parse import urlparse, urlencode
from six.moves.urllib.request import urlopen
from six.moves.urllib.error import HTTPError
```

```
# Python 2 and 3: alternative 4
try:
    from urllib.parse import urlparse, urlencode
    from urllib.request import urlopen, Request
    from urllib.error import HTTPError
except ImportError:
    from urlparse import urlparse
    from urllib import urlencode
    from urllib2 import urlopen, Request, HTTPError
```

### Tkinter

```
# Python 2 only:
import Tkinter
import Dialog
import FileDialog
import ScrolledText
import SimpleDialog
import Tix
import Tkconstants
import Tkdnd
import tkColorChooser
import tkCommonDialog
import tkFileDialog
import tkFont
import tkMessageBox
import tkSimpleDialog
import ttk

# Python 2 and 3 (after ``pip install future``):
import tkinter
import tkinter.dialog
import tkinter.filedialog
import tkinter.scrolledtext
import tkinter.simpledialog
import tkinter.tix
import tkinter.constants
import tkinter.dnd
import tkinter.colorchooser
import tkinter.commondialog
import tkinter.filedialog
import tkinter.font
import tkinter.messagebox
```

(continues on next page)

(continued from previous page)

```
import tkinter.simpledialog
import tkinter.ttk
```

### socketserver

```
# Python 2 only:
import SocketServer

# Python 2 and 3 (after ``pip install future``):
import socketserver
```

### copy_reg, copyreg

```
# Python 2 only:
import copy_reg

# Python 2 and 3 (after ``pip install future``):
import copyreg
```

### configparser

```
# Python 2 only:
from ConfigParser import ConfigParser

# Python 2 and 3 (after ``pip install configparser``):
from configparser import ConfigParser
```

### queue

```
# Python 2 only:
from Queue import Queue, heapq, deque

# Python 2 and 3 (after ``pip install future``):
from queue import Queue, heapq, deque
```

### repr, reprlib

```
# Python 2 only:
from repr import aRepr, repr

# Python 2 and 3 (after ``pip install future``):
from reprlib import aRepr, repr
```

### UserDict, UserList, UserString

```
# Python 2 only:
from UserDict import UserDict
from UserList import UserList
from UserString import UserString

# Python 3 only:
from collections import UserDict, UserList, UserString

# Python 2 and 3: alternative 1
from future.moves.collections import UserDict, UserList, UserString

# Python 2 and 3: alternative 2
from six.moves import UserDict, UserList, UserString

# Python 2 and 3: alternative 3
from future.standard_library import install_aliases
install_aliases()
from collections import UserDict, UserList, UserString
```

### itertools: filterfalse, zip_longest

```
# Python 2 only:
from itertools import ifilterfalse, izip_longest

# Python 3 only:
from itertools import filterfalse, zip_longest

# Python 2 and 3: alternative 1
from future.moves.itertools import filterfalse, zip_longest

# Python 2 and 3: alternative 2
from six.moves import filterfalse, zip_longest

# Python 2 and 3: alternative 3
from future.standard_library import install_aliases
install_aliases()
from itertools import filterfalse, zip_longest
```

## 1.5 Imports

### 1.5.1 __future__ imports

To write a Python 2/3 compatible codebase, the first step is to add this line to the top of each module:

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

For guidelines about whether to import `unicode_literals` too, see below (unicode-literals).

For more information about the `__future__` imports, which are a standard feature of Python, see the following docs:

- absolute_import: PEP 328: Imports: Multi-Line and Absolute/Relative
- division: PEP 238: Changing the Division Operator
- print_function: PEP 3105: Make print a function
- unicode_literals: PEP 3112: Bytes literals in Python 3000

These are all available in Python 2.7 and up, and enabled by default in Python 3.x.

### 1.5.2 Imports of builtins

#### Implicit imports

If you don't mind namespace pollution, the easiest way to provide Py2/3 compatibility for new code using `future` is to include the following imports at the top of every module:

```
from builtins import *
```

On Python 3, this has no effect. (It shadows builtins with globals of the same names.)

On Python 2, this import line shadows 18 builtins (listed below) to provide their Python 3 semantics.

#### Explicit imports

Explicit forms of the imports are often preferred and are necessary for using certain automated code-analysis tools.

The complete set of imports of builtins from `future` is:

```
from builtins import (ascii, bytes, chr, dict, filter, hex, input,
                      int, map, next, oct, open, pow, range, round,
                      str, super, zip)
```

These are also available under the `future.builtins` namespace for backward compatibility.

Importing only some of the builtins is cleaner but increases the risk of introducing Py2/3 portability bugs as your code evolves over time. For example, be aware of forgetting to import `input`, which could expose a security vulnerability on Python 2 if Python 3's semantics are expected.

The internal API is currently as follows:

```
from future.types import bytes, dict, int, range, str
from future.builtins.misc import (ascii, chr, hex, input, next,
                                  oct, open, pow, round, super)
from future.builtins.iterators import filter, map, zip
```

Please note that this internal API is evolving and may not be stable between different versions of `future`. To understand the details of the backported builtins on Python 2, see the docs for these modules.

For more information on what the backported types provide, see *What else you need to know*.

#### Obsolete Python 2 builtins

Twelve Python 2 builtins have been removed from Python 3. To aid with porting code to Python 3 module by module, you can use the following import to cause a `NameError` exception to be raised on Python 2 when any of the obsolete builtins is used, just as would occur on Python 3:

```
from future.builtins.disabled import *
```

This is equivalent to:

```
from future.builtins.disabled import (apply, cmp, coerce, execfile,
                                      file, long, raw_input, reduce, reload,
                                      unicode, xrange, StandardError)
```

Running `futurize` over code that uses these Python 2 builtins does not import the disabled versions; instead, it replaces them with their equivalent Python 3 forms and then adds `future` imports to resurrect Python 2 support, as described in forwards-conversion-stage2.

### 1.5.3 Standard library imports

`future` supports the standard library reorganization (PEP 3108) through several mechanisms.

### Direct imports

As of version 0.14, the `future` package comes with top-level packages for Python 2.x that provide access to the reorganized standard library modules under their Python 3.x names.

Direct imports are the preferred mechanism for accesing the renamed standard library modules in Python 2/3 compatible code. For example, the following clean Python 3 code runs unchanged on Python 2 after installing `future`:

```
>>> # Alias for future.builtins on Py2:
>>> from builtins import str, open, range, dict

>>> # Top-level packages with Py3 names provided on Py2:
>>> import queue
>>> import tkinter.dialog
>>> etc.
```

Notice that this code actually runs on Python 3 without the presence of the `future` package.

Of the 44 modules that were refactored with PEP 3108 (standard library reorganization), 29 are supported with direct imports in the above manner. The complete list is here:

```
### Renamed modules:

import builtins

import copyreg

import html
import html.entities
import html.parser

import http.client
import http.cookies
import http.cookiejar
import http.server

import queue

import reprlib

import socketserver

from tkinter import colorchooser
from tkinter import commondialog
from tkinter import constants
from tkinter import dialog
from tkinter import dnd
from tkinter import filedialog
from tkinter import font
from tkinter import messagebox
```

(continues on next page)

(continued from previous page)

```
from tkinter import scrolledtext
from tkinter import simpledialog
from tkinter import tix
from tkinter import ttk

import winreg                       # Windows only

import xmlrpc.client
import xmlrpc.server

import _dummy_thread
import _markupbase
import _thread
```

Note that, as of v0.16.0, `python-future` no longer includes an alias for the `configparser` module because a full backport exists (see https://pypi.org/project/configparser/).

### Aliased imports

The following 14 modules were refactored or extended from Python 2.7 to 3.x but were neither renamed in Py3.x nor were the new APIs backported to Py2.x. This precludes compatibility interfaces that work out-of-the-box. Instead, the `future` package makes the Python 3.x APIs available on Python 2.x as follows:

```
from future.standard_library import install_aliases
install_aliases()

from collections import UserDict, UserList, UserString

import urllib.parse
import urllib.request
import urllib.response
import urllib.robotparser
import urllib.error

import dbm
import dbm.dumb
import dbm.gnu                  # requires Python dbm support
import dbm.ndbm                 # requires Python dbm support

from itertools import filterfalse, zip_longest

from subprocess import getoutput, getstatusoutput

from sys import intern

import test.support
```

The newly exposed `urllib` submodules are backports of those from Py3.x. This means, for example, that `urllib.parse.unquote()` now exists and takes an optional `encoding` argument on Py2.x as it does on Py3.x.

**Limitation:** Note that the `http`-based backports do not currently support HTTPS (as of 2015-09-11) because the SSL support changed considerably in Python 3.x. If you need HTTPS support, please use this idiom for now:

```
from future.moves.urllib.request import urlopen
```

Backports also exist of the following features from Python 3.4:

- `math.ceil` returns an int on Py3
- `collections.ChainMap` (for 2.7)
- `reprlib.recursive_repr` (for 2.7)

These can then be imported on Python 2.7+ as follows:

```
from future.standard_library import install_aliases
install_aliases()

from math import ceil      # now returns an int
from collections import ChainMap
from reprlib import recursive_repr
```

### 1.5.4 External standard-library backports

Backports of the following modules from the Python 3.x standard library are available independently of the python-future project:

```
import enum                     # pip install enum34
import singledispatch           # pip install singledispatch
import pathlib                  # pip install pathlib
```

A few modules from Python 3.4 are also available in the `backports` package namespace after `pip install backports.lzma` etc.:

```
from backports import lzma
from backports import functools_lru_cache as lru_cache
```

### 1.5.5 Included full backports

Alpha-quality full backports of the following modules from Python 3.3's standard library to Python 2.x are also available in `future.backports`:

```
http.client
http.server
html.entities
html.parser
urllib
xmlrpc.client
xmlrpc.server
```

The goal for these modules, unlike the modules in the `future.moves` package or top-level namespace, is to backport new functionality introduced in Python 3.3.

If you need the full backport of one of these packages, please open an issue here.

### 1.5.6 Using Python 2-only dependencies on Python 3

The `past` module provides an experimental `translation` package to help with importing and using old Python 2 modules in a Python 3 environment.

This is implemented using PEP 414 import hooks together with fixers from `lib2to3` and `libfuturize` (included with `python-future`) that attempt to automatically translate Python 2 code to Python 3 code with equivalent semantics upon import.

*Note* This feature is still in alpha and needs further development to support a full range of real-world Python 2 modules. Also be aware that the API for this package might change considerably in later versions.

Here is how to use it:

```
$ pip3 install plotrique==0.2.5-7 --no-compile   # to ignore␣
↪SyntaxErrors
$ python3
```

Then pass in a whitelist of module name prefixes to the `past.translation.autotranslate()` function. Example:

```
>>> from past.translation import autotranslate
>>> autotranslate(['plotrique'])
>>> import plotrique
```

Here is another example:

```
>>> from past.translation import install_hooks, remove_hooks
>>> install_hooks(['mypy2module'])
>>> import mypy2module
>>> remove_hooks()
```

This will translate, import and run Python 2 code such as the following:

```
### File: mypy2module.py

# Print statements are translated transparently to functions:
print 'Hello from a print statement'

# xrange() is translated to Py3's range():
total = 0
for i in xrange(10):
    total += i
print 'Total is: %d' % total

# Dictionary methods like .keys() and .items() are supported and
# return lists as on Python 2:
d = {'a': 1, 'b': 2}
assert d.keys() == ['a', 'b']
assert isinstance(d.items(), list)

# Functions like range, reduce, map, filter also return lists:
assert isinstance(range(10), list)

# The exec statement is supported:
exec 'total += 1'
print 'Total is now: %d' % total

# Long integers are supported:
k = 1234983424324L
print 'k + 1 = %d' % k

# Most renamed standard library modules are supported:
import ConfigParser
import HTMLParser
import urllib
```

The attributes of the module are then accessible normally from Python 3. For example:

```
# This Python 3 code works
>>> type(mypy2module.d)
builtins.dict
```

This is a standard Python 3 data type, so, when called from Python 3 code, `keys()` returns a view, not a list:

```
>>> type(mypy2module.d.keys())
builtins.dict_keys
```

- It currently requires a newline at the end of the module or it throws a `ParseError`.
- This only works with pure-Python modules. C extension modules and Cython code are not supported.

- The biggest hurdle to automatic translation is likely to be ambiguity about byte-strings and text (unicode strings) in the Python 2 code. If the `past.autotranslate` feature fails because of this, you could try running `futurize` over the code and adding a `b''` or `u''` prefix to the relevant string literals. To convert between byte-strings and text (unicode strings), add an `.encode` or `.decode` method call. If this succeeds, please push your patches upstream to the package maintainers.
- Otherwise, the source translation feature offered by the `past.translation` package has similar limitations to the `futurize` script (see futurize-limitations). Help developing and testing this feature further would be particularly welcome.

Please report any bugs you find on the `python-future` bug tracker.

### 1.5.7 Should I import unicode_literals?

The `future` package can be used with or without `unicode_literals` imports.

In general, it is more compelling to use `unicode_literals` when back-porting new or existing Python 3 code to Python 2/3 than when porting existing Python 2 code to 2/3. In the latter case, explicitly marking up all unicode string literals with `u''` prefixes would help to avoid unintentionally changing the existing Python 2 API. However, if changing the existing Python 2 API is not a concern, using `unicode_literals` may speed up the porting process.

This section summarizes the benefits and drawbacks of using `unicode_literals`. To avoid confusion, we recommend using `unicode_literals` everywhere across a code-base or not at all, instead of turning on for only some modules.

#### Benefits

1. String literals are unicode on Python 3. Making them unicode on Python 2 leads to more consistency of your string types across the two runtimes. This can make it easier to understand and debug your code.
2. Code without `u''` prefixes is cleaner, one of the claimed advantages of Python 3. Even though some unicode strings would require a function call to invert them to native strings for some Python 2 APIs (see *Standard library incompatibilities*), the incidence of these function calls would usually be much lower than the incidence of `u''` prefixes for text strings in the absence of `unicode_literals`.
3. The diff when porting to a Python 2/3-compatible codebase may be smaller, less noisy, and easier to review with `unicode_literals` than if an explicit `u''` prefix is added to every unadorned string literal.
4. If support for Python 3.2 is required (e.g. for Ubuntu 12.04 LTS or Debian wheezy), `u''` prefixes are a `SyntaxError`, making `unicode_literals` the only option for a Python 2/3 compatible codebase. [However, note that `future` doesn't support Python 3.0-3.2.]

### Drawbacks

1. Adding `unicode_literals` to a module amounts to a "global flag day" for that module, changing the data types of all strings in the module at once. Cautious developers may prefer an incremental approach. (See here for an excellent article describing the superiority of an incremental patch-set in the the case of the Linux kernel.)

2. Changing to `unicode_literals` will likely introduce regressions on Python 2 that require an initial investment of time to find and fix. The APIs may be changed in subtle ways that are not immediately obvious.

   An example on Python 2:

   ```
   ### Module: mypaths.py

   ...
   def unix_style_path(path):
       return path.replace('\\', '/')
   ...

   ### User code:

   >>> path1 = '\\Users\\Ed'
   >>> unix_style_path(path1)
   '/Users/ed'
   ```

   On Python 2, adding a `unicode_literals` import to `mypaths.py` would change the return type of the `unix_style_path` function from `str` to `unicode` in the user code, which is difficult to anticipate and probably unintended.

   The counter-argument is that this code is broken, in a portability sense; we see this from Python 3 raising a `TypeError` upon passing the function a byte-string. The code needs to be changed to make explicit whether the `path` argument is to be a byte string or a unicode string.

3. With `unicode_literals` in effect, there is no way to specify a native string literal (`str` type on both platforms). This can be worked around as follows:

   ```
   >>> from __future__ import unicode_literals
   >>> ...
   >>> from future.utils import bytes_to_native_str as n

   >>> s = n(b'ABCD')
   >>> s
   'ABCD'  # on both Py2 and Py3
   ```

   although this incurs a performance penalty (a function call and, on Py3, a `decode` method call.)

   This is a little awkward because various Python library APIs (standard and non-standard) require a native string to be passed on both Py2 and Py3. (See *Standard library incompatibilities* for some examples. WSGI dictionaries are another.)

3. If a codebase already explicitly marks up all text with `u''` prefixes, and if support for Python versions 3.0-3.2 can be dropped, then removing the existing `u''` prefixes and replacing these with `unicode_literals` imports (the porting approach Django used) would introduce more noise into the patch and make it more difficult to review. However, note that the `futurize` script takes advantage of PEP 414 and does not remove explicit `u''` prefixes that already exist.

4. Turning on `unicode_literals` converts even docstrings to unicode, but Pydoc breaks with unicode docstrings containing non-ASCII characters for Python versions < 2.7.7. (Fix committed in Jan 2014.):

```
>>> def f():
...     u"Author: Martin von Löwis"

>>> help(f)

/Users/schofield/Install/anaconda/python.app/Contents/lib/
↪python2.7/pydoc.pyc in pipepager(text, cmd)
   1376     pipe = os.popen(cmd, 'w')
   1377     try:
-> 1378         pipe.write(text)
   1379         pipe.close()
   1380     except IOError:

UnicodeEncodeError: 'ascii' codec can't encode character u'\xf6
↪' in position 71: ordinal not in range(128)
```

See this Stack Overflow thread for other gotchas.

### Others' perspectives

Django recommends importing `unicode_literals` as its top porting tip for migrating Django extension modules to Python 3. The following quote is from Aymeric Augustin on 23 August 2012 regarding why he chose `unicode_literals` for the port of Django to a Python 2/3-compatible codebase.:

> "... I'd like to explain why this PEP [PEP 414, which allows explicit `u''` prefixes for unicode literals on Python 3.3+] is at odds with the porting philosophy I've applied to Django, and why I would have vetoed taking advantage of it.
>
> "I believe that aiming for a Python 2 codebase with Python 3 compatibility hacks is a counter-productive way to port a project. You end up with all the drawbacks of Python 2 (including the legacy *u* prefixes) and none of the advantages Python 3 (especially the sane string handling).
>
> "Working to write Python 3 code, with legacy compatibility for Python 2, is much more rewarding. Of course it takes more effort, but the results are much cleaner and much more maintainable. It's really about looking towards the future or towards the past.
>
> "I understand the reasons why PEP 414 was proposed and why it was accepted. It

makes sense for legacy software that is minimally maintained. I hope nobody puts Django in this category!"

"There are so many subtle problems that `unicode_literals` causes. For instance lots of people accidentally introduce unicode into filenames and that seems to work, until they are using it on a system where there are unicode characters in the filesystem path."

—Armin Ronacher

"+1 from me for avoiding the unicode_literals future, as it can have very strange side effects in Python 2.... This is one of the key reasons I backed Armin's PEP 414."

—Nick Coghlan

"Yeah, one of the nuisances of the WSGI spec is that the header values IIRC are the str or StringType on both py2 and py3. With unicode_literals this causes hard-to-spot bugs, as some WSGI servers might be more tolerant than others, but usually using unicode in python 2 for WSGI headers will cause the response to fail."

—Antti Haapala

### 1.5.8 Next steps

See *What else you need to know*.

## 1.6 What else you need to know

The following points are important to know about when writing Python 2/3 compatible code.

### 1.6.1 bytes

Handling `bytes` consistently and correctly has traditionally been one of the most difficult tasks in writing a Py2/3 compatible codebase. This is because the Python 2 `bytes` object is simply an alias for Python 2's `str`, rather than a true implementation of the Python 3 `bytes` object, which is substantially different.

`future` contains a backport of the `bytes` object from Python 3 which passes most of the Python 3 tests for `bytes`. (See `tests/test_future/test_bytes.py` in the source tree.) You can use it as follows:

```
>>> from builtins import bytes
>>> b = bytes(b'ABCD')
```

On Py3, this is simply the builtin `bytes` object. On Py2, this object is a subclass of Python 2's `str` that enforces the same strict separation of unicode strings and byte strings as Python 3's `bytes` object:

```
>>> b + u'EFGH'       # TypeError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: argument can't be unicode string

>>> bytes(b',').join([u'Fred', u'Bill'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected bytes, found unicode string

>>> b == u'ABCD'
False

>>> b < u'abc'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: bytes() and <type 'unicode'>
```

In most other ways, these `bytes` objects have identical behaviours to Python 3's `bytes`:

```
b = bytes(b'ABCD')
assert list(b) == [65, 66, 67, 68]
assert repr(b) == "b'ABCD'"
assert b.split(b'B') == [b'A', b'CD']
```

Currently the easiest way to ensure identical behaviour of byte-strings in a Py2/3 codebase is to wrap all byte-string literals `b'...'` in a `bytes()` call as follows:

```
from builtins import bytes

# ...

b = bytes(b'This is my bytestring')

# ...
```

This is not perfect, but it is superior to manually debugging and fixing code incompatibilities caused by the many differences between Py3 bytes and Py2 strings.

The `bytes` type from `builtins` also provides support for the `surrogateescape` error handler on Python 2.x. Here is an example that works identically on Python 2.x and 3.x:

```
>>> from builtins import bytes
>>> b = bytes(b'\xff')
>>> b.decode('utf-8', 'surrogateescape')
'\udcc3'
```

This feature is in alpha. Please leave feedback here about whether this works for you.

### 1.6.2 str

The `str` object in Python 3 is quite similar but not identical to the Python 2 `unicode` object.

The major difference is the stricter type-checking of Py3's `str` that enforces a distinction between unicode strings and byte-strings, such as when comparing, concatenating, joining, or replacing parts of strings.

There are also other differences, such as the `repr` of unicode strings in Py2 having a `u'...'` prefix, versus simply `'...'`, and the removal of the `str.decode()` method in Py3.

`future` contains a `newstr` type that is a backport of the `str` object from Python 3. This inherits from the Python 2 `unicode` class but has customizations to improve compatibility with Python 3's `str` object. You can use it as follows:

```
>>> from __future__ import unicode_literals
>>> from builtins import str
```

On Py2, this gives us:

```
>>> str
future.types.newstr.newstr
```

(On Py3, it is simply the usual builtin `str` object.)

Then, for example, the following code has the same effect on Py2 as on Py3:

```
>>> s = str(u'ABCD')
>>> assert s != b'ABCD'
>>> assert isinstance(s.encode('utf-8'), bytes)
>>> assert isinstance(b.decode('utf-8'), str)

These raise TypeErrors:

>>> bytes(b'B') in s
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'in <string>' requires string as left operand, not <type
 ↪'str'>

>>> s.find(bytes(b'A'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: argument can't be <type 'str'>
```

Various other operations that mix strings and bytes or other types are permitted on Py2 with the `newstr` class even though they are illegal with Python 3. For example:

```
>>> s2 = b'/' + str('ABCD')
>>> s2
'/ABCD'
```

(continues on next page)

(continued from previous page)

```
>>> type(s2)
future.types.newstr.newstr
```

This is allowed for compatibility with parts of the Python 2 standard library and various third-party libraries that mix byte-strings and unicode strings loosely. One example is `os.path.join` on Python 2, which attempts to add the byte-string `b'/'` to its arguments, whether or not they are unicode. (See `posixpath.py`.) Another example is the `escape()` function in Django 1.4's `django.utils.html`.

In most other ways, these `builtins.str` objects on Py2 have the same behaviours as Python 3's `str`:

```
>>> s = str('ABCD')
>>> assert repr(s) == 'ABCD'       # consistent repr with Py3 (no u␣
 ↪prefix)
>>> assert list(s) == ['A', 'B', 'C', 'D']
>>> assert s.split('B') == ['A', 'CD']
```

The `str` type from `builtins` also provides support for the `surrogateescape` error handler on Python 2.x. Here is an example that works identically on Python 2.x and 3.x:

```
>>> from builtins import str
>>> s = str(u'\udcff')
>>> s.encode('utf-8', 'surrogateescape')
b'\xff'
```

This feature is in alpha. Please leave feedback here about whether this works for you.

### 1.6.3 dict

Python 3 dictionaries have `.keys()`, `.values()`, and `.items()` methods which return memory-efficient set-like iterator objects, not lists. (See PEP 3106.)

If your dictionaries are small, performance is not critical, and you don't need the set-like behaviour of iterator objects from Python 3, you can of course stick with standard Python 3 code in your Py2/3 compatible codebase:

```
# Assuming d is a native dict ...

for key in d:
    # code here

for item in d.items():
    # code here

for value in d.values():
    # code here
```

In this case there will be memory overhead of list creation on Py2 for each call to `items`, `values` or `keys`.

For improved efficiency, `future.builtins` (aliased to `builtins`) provides a Python 2 `dict` subclass whose `keys()`, `values()`, and `items()` methods return iterators on all versions of Python >= 2.7. On Python 2.7, these iterators also have the same set-like view behaviour as dictionaries in Python 3. This can streamline code that iterates over large dictionaries. For example:

```
from __future__ import print_function
from builtins import dict, range

# Memory-efficient construction:
d = dict((i, i**2) for i in range(10**7))

assert not isinstance(d.items(), list)

# Because items() is memory-efficient, so is this:
d2 = dict((v, k) for (k, v) in d.items())
```

As usual, on Python 3 `dict` imported from either `builtins` or `future.builtins` is just the built-in `dict` class.

### Memory-efficiency and alternatives

If you already have large native dictionaries, the downside to wrapping them in a `dict` call is that memory is copied (on both Py3 and on Py2). For example:

```
# This allocates and then frees a large amount of temporary memory:
d = dict({i: i**2 for i in range(10**7)})
```

If dictionary methods like `values` and `items` are called only once, this obviously negates the memory benefits offered by the overridden methods through not creating temporary lists.

The memory-efficient (and CPU-efficient) alternatives are:

- to construct a dictionary from an iterator. The above line could use a generator like this:

  ```
  d = dict((i, i**2) for i in range(10**7))
  ```

- to construct an empty dictionary with a `dict()` call using `builtins.dict` (rather than `{}`) and then update it;
- to use the `viewitems` etc. functions from *`future.utils`*, passing in regular dictionaries:

  ```
  from future.utils import viewkeys, viewvalues, viewitems

  for (key, value) in viewitems(hugedictionary):
      # some code here
  ```

(continues on next page)

(continued from previous page)

```
# Set intersection:
d = {i**2: i for i in range(1000)}
both = viewkeys(d) & set(range(0, 1000, 7))

# Set union:
both = viewvalues(d1) | viewvalues(d2)
```

For compatibility, the functions `iteritems` etc. are also available in *`future.utils`*. These are equivalent to the functions of the same names in `six`, which is equivalent to calling the `iteritems` etc. methods on Python 2, or to calling `items` etc. on Python 3.

### 1.6.4 int

Python 3's `int` type is very similar to Python 2's `long`, except for the representation (which omits the `L` suffix in Python 2). Python 2's usual (short) integers have been removed from Python 3, as has the `long` builtin name.

Python 3:

```
>>> 2**64
18446744073709551616
```

Python 2:

```
>>> 2**64
18446744073709551616L
```

`future` includes a backport of Python 3's `int` that is a subclass of Python 2's `long` with the same representation behaviour as Python 3's `int`. To ensure an integer is long compatibly with both Py3 and Py2, cast it like this:

```
>>> from builtins import int
>>> must_be_a_long_integer = int(1234)
```

The backported `int` object helps with writing doctests and simplifies code that deals with `long` and `int` as special cases on Py2. An example is the following code from `xlwt-future` (called by the `xlwt.antlr.BitSet` class) for writing out Excel `.xls` spreadsheets. With `future`, the code is:

```
from builtins import int

def longify(data):
    """
    Turns data (an int or long, or a list of ints or longs) into a
    list of longs.
    """
    if not data:
        return [int(0)]
```

(continues on next page)

(continued from previous page)

```
    if not isinstance(data, list):
        return [int(data)]
    return list(map(int, data))
```

Without `future` (or with `future` < 0.7), this might be:

```
def longify(data):
    """
    Turns data (an int or long, or a list of ints or longs) into a
    list of longs.
    """
    if not data:
        if PY3:
            return [0]
        else:
            return [long(0)]
    if not isinstance(data,list):
        if PY3:
            return [int(data)]
        else:
            return [long(data)]
    if PY3:
        return list(map(int, data))    # same as returning data, but␣
↪with up-front typechecking
    else:
        return list(map(long, data))
```

### 1.6.5 isinstance

The following tests all pass on Python 3:

```
>>> assert isinstance(2**62, int)
>>> assert isinstance(2**63, int)
>>> assert isinstance(b'my byte-string', bytes)
>>> assert isinstance(u'unicode string 1', str)
>>> assert isinstance('unicode string 2', str)
```

However, two of these normally fail on Python 2:

```
>>> assert isinstance(2**63, int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError

>>> assert isinstance(u'my unicode string', str)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

And if this import is in effect on Python 2:

```
>>> from __future__ import unicode_literals
```

then the fifth test fails too:

```
>>> assert isinstance('unicode string 2', str)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

After importing the builtins from `future`, all these tests pass on Python 2 as on Python 3:

```
>>> from builtins import bytes, int, str

>>> assert isinstance(10, int)
>>> assert isinstance(10**100, int)
>>> assert isinstance(b'my byte-string', bytes)
>>> assert isinstance(u'unicode string 1', str)
```

However, note that the last test requires that `unicode_literals` be imported to succeed.:

```
>>> from __future__ import unicode_literals
>>> assert isinstance('unicode string 2', str)
```

This works because the backported types `int`, `bytes` and `str` (and others) have metaclasses that override `__instancecheck__`. See PEP 3119 for details.

### 1.6.6 Passing data to/from Python 2 libraries

If you are passing any of the backported types (`bytes`, `int`, `dict`, ``str`) into brittle library code that performs type-checks using `type()`, rather than `isinstance()`, or requires that you pass Python 2's native types (rather than subclasses) for some other reason, it may be necessary to upcast the types from `future` to their native superclasses on Py2.

The `native` function in `future.utils` is provided for this. Here is how to use it. (The output showing is from Py2):

```
>>> from builtins import int, bytes, str
>>> from future.utils import native

>>> a = int(10**20)      # Py3-like long int
>>> a
100000000000000000000
>>> type(a)
future.types.newint.newint
>>> native(a)
100000000000000000000L
>>> type(native(a))
long
```

(continues on next page)

(continued from previous page)

```
>>> b = bytes(b'ABC')
>>> type(b)
future.types.newbytes.newbytes
>>> native(b)
'ABC'
>>> type(native(b))
str

>>> s = str(u'ABC')
>>> type(s)
future.types.newstr.newstr
>>> native(s)
u'ABC'
>>> type(native(s))
unicode
```

On Py3, the `native()` function is a no-op.

### 1.6.7 Native string type

Some library code, include standard library code like the `array.array()` constructor, require native strings on Python 2 and Python 3. This means that there is no simple way to pass the appropriate string type when the `unicode_literals` import from `__future__` is in effect.

The objects `native_str` and `native_bytes` are available in `future.utils` for this case. These are equivalent to the `str` and `bytes` objects in `__builtin__` on Python 2 or in `builtins` on Python 3.

The functions `native_str_to_bytes` and `bytes_to_native_str` are also available for more explicit conversions.

### 1.6.8 open()

The Python 3 builtin `open()` function for opening files returns file contents as (unicode) strings unless the binary (`b`) flag is passed, as in:

```
open(filename, 'rb')
```

in which case its methods like `read()` return Py3 `bytes` objects.

On Py2 with `future` installed, the `builtins` module provides an `open` function that is mostly compatible with that on Python 3 (e.g. it offers keyword arguments like `encoding`). This maps to the `open` backport available in the standard library `io` module on Py2.7.

One difference to be aware of between the Python 3 `open` and `future.builtins.open` on Python 2 is that the return types of methods such as `read()` from the file object that

`open` returns are not automatically cast from native bytes or unicode strings on Python 2 to the corresponding `future.builtins.bytes` or `future.builtins.str` types. If you need the returned data to behave the exactly same way on Py2 as on Py3, you can cast it explicitly as follows:

```
from __future__ import unicode_literals
from builtins import open, bytes

data = open('image.png', 'rb').read()
# On Py2, data is a standard 8-bit str with loose Unicode coercion.
# data + u'' would likely raise a UnicodeDecodeError

data = bytes(data)
# Now it behaves like a Py3 bytes object...

assert data[:4] == b'\x89PNG'
assert data[4] == 13     # integer
# Raises TypeError:
# data + u''
```

### 1.6.9 Custom __str__ methods

If you define a custom `__str__` method for any of your classes, functions like `print()` expect `__str__` on Py2 to return a byte string, whereas on Py3 they expect a (unicode) string.

Use the following decorator to map the `__str__` to `__unicode__` on Py2 and define `__str__` to encode it as utf-8:

```
from future.utils import python_2_unicode_compatible

@python_2_unicode_compatible
class MyClass(object):
    def __str__(self):
        return u'Unicode string: \u5b54\u5b50'
a = MyClass()

# This then prints the name of a Chinese philosopher:
print(a)
```

This decorator is identical to the decorator of the same name in `django.utils.encoding`.

This decorator is a no-op on Python 3.

### 1.6.10 Custom iterators

If you define your own iterators, there is an incompatibility in the method name to retrieve the next item across Py3 and Py2. On Python 3 it is `__next__`, whereas on Python 2 it is `next`.

The most elegant solution to this is to derive your custom iterator class from `builtins.object` and define a `__next__` method as you normally would on Python 3. On Python 2, `object` then refers to the `future.types.newobject` base class, which provides a fallback `next` method that calls your `__next__`. Use it as follows:

```
from builtins import object

class Upper(object):
    def __init__(self, iterable):
        self._iter = iter(iterable)
    def __next__(self):                 # Py3-style iterator interface
        return next(self._iter).upper()
    def __iter__(self):
        return self

itr = Upper('hello')
assert next(itr) == 'H'
assert next(itr) == 'E'
assert list(itr) == list('LLO')
```

You can use this approach unless you are defining a custom iterator as a subclass of a base class defined elsewhere that does not derive from `newobject`. In that case, you can provide compatibility across Python 2 and Python 3 using the `next` function from `future.builtins`:

```
from builtins import next

from some_module import some_base_class

class Upper2(some_base_class):
    def __init__(self, iterable):
        self._iter = iter(iterable)
    def __next__(self):                 # Py3-style iterator interface
        return next(self._iter).upper()
    def __iter__(self):
        return self

itr2 = Upper2('hello')
assert next(itr2) == 'H'
assert next(itr2) == 'E'
```

`next()` also works with regular Python 2 iterators with a `.next` method:

```
itr3 = iter(['one', 'three', 'five'])
```

(continues on next page)

(continued from previous page)

```
assert 'next' in dir(itr3)
assert next(itr3) == 'one'
```

This approach is feasible whenever your code calls the `next()` function explicitly. If you consume the iterator implicitly in a `for` loop or `list()` call or by some other means, the `future.builtins.next` function will not help; the third assertion below would fail on Python 2:

```
itr2 = Upper2('hello')

assert next(itr2) == 'H'
assert next(itr2) == 'E'
assert list(itr2) == list('LLO')       # fails because Py2␣
↪implicitly looks
                                       # for a ``next`` method.
```

Instead, you can use a decorator called `implements_iterator` from `future.utils` to allow Py3-style iterators to work identically on Py2, even if they don't inherit from `future.builtins.object`. Use it as follows:

```
from future.utils import implements_iterator

Upper2 = implements_iterator(Upper2)

print(list(Upper2('hello')))
# prints ['H', 'E', 'L', 'L', 'O']
```

This can of course also be used with the `@` decorator syntax when defining the iterator as follows:

```
@implements_iterator
class Upper2(some_base_class):
    def __init__(self, iterable):
        self._iter = iter(iterable)
    def __next__(self):                    # note the Py3 interface
        return next(self._iter).upper()
    def __iter__(self):
        return self
```

On Python 3, as usual, this decorator does nothing.

### 1.6.11 Binding a method to a class

Python 2 draws a distinction between bound and unbound methods, whereas in Python 3 this distinction is gone: unbound methods have been removed from the language. To bind a method to a class compatibly across Python 3 and Python 2, you can use the `bind_method()` helper function:

```
from future.utils import bind_method

class Greeter(object):
    pass

def greet(self, message):
    print(message)

bind_method(Greeter, 'greet', greet)

g = Greeter()
g.greet('Hi!')
```

On Python 3, calling `bind_method(cls, name, func)` is equivalent to calling `setattr(cls, name, func)`. On Python 2 it is equivalent to:

```
import types
setattr(cls, name, types.MethodType(func, None, cls))
```

### 1.6.12 Metaclasses

Python 3 and Python 2 syntax for metaclasses are incompatible. `future` provides a function (from `jinja2/_compat.py`) called `with_metaclass()` that can assist with specifying metaclasses portably across Py3 and Py2. Use it like this:

```
from future.utils import with_metaclass

class BaseForm(object):
    pass

class FormType(type):
    pass

class Form(with_metaclass(FormType, BaseForm)):
    pass
```

## 1.7 Automatic conversion to Py2/3

The `future` source tree includes scripts called `futurize` and `pasteurize` to aid in making Python 2 code or Python 3 code compatible with both platforms (Py2/3) using the `future` module. These are based on `lib2to3` and use fixers from `2to3`, `3to2`, and `python-modernize`.

`futurize` passes Python 2 code through all the appropriate fixers to turn it into valid Python 3 code, and then adds `__future__` and `future` package imports.

For conversions from Python 3 code to Py2/3, use the `pasteurize` script instead. This converts Py3-only constructs (e.g. new metaclass syntax) and adds `__future__` and `future` imports to the top of each module.

In both cases, the result should be relatively clean Py3-style code that runs mostly unchanged on both Python 2 and Python 3.

### 1.7.1 `futurize`: Py2 to Py2/3

The `futurize` script passes Python 2 code through all the appropriate fixers to turn it into valid Python 3 code, and then adds `__future__` and `future` package imports to re-enable compatibility with Python 2.

For example, running `futurize` turns this Python 2 code:

```
import ConfigParser                    # Py2 module name

class Upper(object):
    def __init__(self, iterable):
        self._iter = iter(iterable)
    def next(self):                    # Py2-style iterator interface
        return next(self._iter).upper()
    def __iter__(self):
        return self

itr = Upper('hello')
print next(itr),
for letter in itr:
    print letter,                      # Py2-style print statement
```

into this code which runs on both Py2 and Py3:

```
from __future__ import print_function
from future import standard_library
standard_library.install_aliases()
from future.builtins import next
from future.builtins import object
import configparser                    # Py3-style import

class Upper(object):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, iterable):
        self._iter = iter(iterable)
    def __next__(self):                 # Py3-style iterator interface
        return next(self._iter).upper()
    def __iter__(self):
        return self

itr = Upper('hello')
print(next(itr), end=' ')            # Py3-style print function
for letter in itr:
    print(letter, end=' ')
```

To write out all the changes to your Python files that `futurize` suggests, use the `-w` flag.

For complex projects, it is probably best to divide the porting into two stages. Stage 1 is for "safe" changes that modernize the code but do not break Python 2.7 compatibility or introduce a dependency on the `future` package. Stage 2 is to complete the process.

### Stage 1: "safe" fixes

Run the first stage of the conversion process with:

```
futurize --stage1 mypackage/*.py
```

or, if you are using zsh, recursively:

```
futurize --stage1 mypackage/**/*.py
```

This applies fixes that modernize Python 2 code without changing the effect of the code. With luck, this will not introduce any bugs into the code, or will at least be trivial to fix. The changes are those that bring the Python code up-to-date without breaking Py2 compatibility. The resulting code will be modern Python 2.7-compatible code plus `__future__` imports from the following set:

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

Only those `__future__` imports deemed necessary will be added unless the `--all-imports` command-line option is passed to `futurize`, in which case they are all added.

The `from __future__ import unicode_literals` declaration is not added unless the `--unicode-literals` flag is passed to `futurize`.

The changes include:

```
- except MyException, e:
+ except MyException as e:
```

(continues on next page)

(continued from previous page)

```
- print >>stderr, "Blah"
+ from __future__ import print_function
+ print("Blah", stderr)

- class MyClass:
+ class MyClass(object):

- def next(self):
+ def __next__(self):

- if d.has_key(key):
+ if key in d:
```

Implicit relative imports fixed, e.g.:

```
- import mymodule
+ from __future__ import absolute_import
+ from . import mymodule
```

Stage 1 does not add any imports from the `future` package. The output of stage 1 will probably not (yet) run on Python 3.

The goal for this stage is to create most of the `diff` for the entire porting process, but without introducing any bugs. It should be uncontroversial and safe to apply to every Python 2 package. The subsequent patches introducing Python 3 compatibility should then be shorter and easier to review.

The complete set of fixers applied by `futurize --stage1` is:

```
lib2to3.fixes.fix_apply
lib2to3.fixes.fix_except
lib2to3.fixes.fix_exec
lib2to3.fixes.fix_exitfunc
lib2to3.fixes.fix_funcattrs
lib2to3.fixes.fix_has_key
lib2to3.fixes.fix_idioms
lib2to3.fixes.fix_intern
lib2to3.fixes.fix_isinstance
lib2to3.fixes.fix_methodattrs
lib2to3.fixes.fix_ne
lib2to3.fixes.fix_numliterals
lib2to3.fixes.fix_paren
lib2to3.fixes.fix_reduce
lib2to3.fixes.fix_renames
lib2to3.fixes.fix_repr
lib2to3.fixes.fix_standarderror
lib2to3.fixes.fix_sys_exc
lib2to3.fixes.fix_throw
lib2to3.fixes.fix_tuple_params
```

(continues on next page)

(continued from previous page)

```
lib2to3.fixes.fix_types
lib2to3.fixes.fix_ws_comma
lib2to3.fixes.fix_xreadlines
libfuturize.fixes.fix_absolute_import
libfuturize.fixes.fix_next_call
libfuturize.fixes.fix_print_with_import
libfuturize.fixes.fix_raise
```

The following fixers from `lib2to3` are not applied:

```
lib2to3.fixes.fix_import
```

The `fix_absolute_import` fixer in `libfuturize.fixes` is applied instead of `lib2to3.fixes.fix_import`. The new fixer both makes implicit relative imports explicit and adds the declaration `from __future__ import absolute_import` at the top of each relevant module.

```
lib2to3.fixes.fix_next
```

The `fix_next_call` fixer in `libfuturize.fixes` is applied instead of `fix_next` in stage 1. The new fixer changes any `obj.next()` calls to `next(obj)`, which is Py2/3 compatible, but doesn't change any `next` method names to `__next__`, which would break Py2 compatibility.

`fix_next` is applied in stage 2.

```
lib2to3.fixes.fix_print
```

The `fix_print_with_import` fixer in `libfuturize.fixes` changes the code to use print as a function and also adds `from __future__ import print_function` to the top of modules using `print()`.

In addition, it avoids adding an extra set of parentheses if these already exist. So `print(x)` does not become `print((x))`.

```
lib2to3.fixes.fix_raise
```

This fixer translates code to use the Python 3-only `with_traceback()` method on exceptions.

```
lib2to3.fixes.fix_set_literal
```

This converts `set([1, 2, 3])` to `{1, 2, 3}`.

```
lib2to3.fixes.fix_ws_comma
```

This performs cosmetic changes. This is not applied by default because it does not serve to improve Python 2/3 compatibility. (In some cases it may also reduce readability: see issue #58.)

### Stage 2: Py3-style code with wrappers for Py2

Run stage 2 of the conversion process with:

```
futurize --stage2 myfolder/*.py
```

This stage adds a dependency on the `future` package. The goal for stage 2 is to make further mostly safe changes to the Python 2 code to use Python 3-style code that then still runs on Python 2 with the help of the appropriate builtins and utilities in `future`.

For example:

```
name = raw_input('What is your name?\n')

for k, v in d.iteritems():
    assert isinstance(v, basestring)

class MyClass(object):
    def __unicode__(self):
        return u'My object'
    def __str__(self):
        return unicode(self).encode('utf-8')
```

would be converted by Stage 2 to this code:

```
from builtins import input
from builtins import str
from future.utils import iteritems, python_2_unicode_compatible

name = input('What is your name?\n')

for k, v in iteritems(d):
    assert isinstance(v, (str, bytes))

@python_2_unicode_compatible
class MyClass(object):
    def __str__(self):
        return u'My object'
```

Stage 2 also renames standard-library imports to their Py3 names and adds these two lines:

```
from future import standard_library
standard_library.install_aliases()
```

For example:

```
import ConfigParser
```

becomes:

```
from future import standard_library
standard_library.install_aliases()
import configparser
```

The complete list of fixers applied in Stage 2 is:

```
lib2to3.fixes.fix_dict
lib2to3.fixes.fix_filter
lib2to3.fixes.fix_getcwdu
lib2to3.fixes.fix_input
lib2to3.fixes.fix_itertools
lib2to3.fixes.fix_itertools_imports
lib2to3.fixes.fix_long
lib2to3.fixes.fix_map
lib2to3.fixes.fix_next
lib2to3.fixes.fix_nonzero
lib2to3.fixes.fix_operator
lib2to3.fixes.fix_raw_input
lib2to3.fixes.fix_zip

libfuturize.fixes.fix_basestring
libfuturize.fixes.fix_cmp
libfuturize.fixes.fix_division_safe
libfuturize.fixes.fix_execfile
libfuturize.fixes.fix_future_builtins
libfuturize.fixes.fix_future_standard_library
libfuturize.fixes.fix_future_standard_library_urllib
libfuturize.fixes.fix_metaclass
libpasteurize.fixes.fix_newstyle
libfuturize.fixes.fix_object
libfuturize.fixes.fix_unicode_keep_u
libfuturize.fixes.fix_xrange_with_import
```

Not applied:

```
lib2to3.fixes.fix_buffer    # Perhaps not safe. Test this.
lib2to3.fixes.fix_callable  # Not needed in Py3.2+
lib2to3.fixes.fix_execfile  # Some problems: see issue #37.
                            # We use the custom libfuturize.fixes.
↪fix_execfile instead.
lib2to3.fixes.fix_future    # Removing __future__ imports is bad␣
↪for Py2 compatibility!
lib2to3.fixes.fix_imports   # Called by libfuturize.fixes.fix_
↪future_standard_library
lib2to3.fixes.fix_imports2  # We don't handle this yet (dbm)
lib2to3.fixes.fix_metaclass # Causes SyntaxError in Py2! Use the␣
↪one from ``six`` instead
lib2to3.fixes.fix_unicode   # Strips off the u'' prefix, which␣
↪removes a potentially
                            # helpful source of information for␣
↪disambiguating
```

(continues on next page)

(continued from previous page)

```
                                # unicode/byte strings.
lib2to3.fixes.fix_urllib        # Included in libfuturize.fix_future_
↪standard_library_urllib
lib2to3.fixes.fix_xrange        # Custom one because of a bug with Py3.3
↪'s lib2to3
```

### Separating text from bytes

After applying stage 2, the recommended step is to decide which of your Python 2 strings represent text and which represent binary data and to prefix all string literals with either `b` or `u` accordingly. Furthermore, to ensure that these types behave similarly on Python 2 as on Python 3, also wrap byte-strings or text in the `bytes` and `str` types from `future`. For example:

```
from builtins import bytes, str
b = bytes(b'\x00ABCD')
s = str(u'This is normal text')
```

Any unadorned string literals will then represent native platform strings (byte-strings on Py2, unicode strings on Py3).

An alternative is to pass the `--unicode-literals` flag:

```
$ futurize --unicode-literals mypython2script.py
```

After running this, all string literals that were not explicitly marked up as `b''` will mean text (Python 3 `str` or Python 2 `unicode`).

### Post-conversion

After running `futurize`, we recommend first running your tests on Python 3 and making further code changes until they pass on Python 3.

The next step would be manually tweaking the code to re-enable Python 2 compatibility with the help of the `future` package. For example, you can add the `@python_2_unicode_compatible` decorator to any classes that define custom `__str__` methods. See *What else you need to know* for more info.

## 1.7.2 `futurize` quick-start guide

How to convert Py2 code to Py2/3 code using `futurize`:

### Step 0: setup

Step 0 goal: set up and see the tests passing on Python 2 and failing on Python 3.

a. Clone the package from github/bitbucket. Optionally rename your repo to `package-future`. Examples: `reportlab-future`, `paramiko-future`, `mezzanine-future`.

b. Create and activate a Python 2 conda environment or virtualenv. Install the package with `python setup.py install` and run its test suite on Py2.7 (e.g. `python setup.py test` or `py.test`)

c. Optionally: if there is a `.travis.yml` file, add Python version 3.6 and remove any versions < 2.6.

d. Install Python 3 with e.g. `sudo apt-get install python3`. On other platforms, an easy way is to use Miniconda. Then e.g.:

```
conda create -n py36 python=3.6 pip
```

### Step 1: modern Py2 code

The goal for this step is to modernize the Python 2 code without introducing any dependencies (on `future` or e.g. `six`) at this stage.

**1a**. Install `future` into the virtualenv using:

```
pip install future
```

**1b**. Run `futurize --stage1 -w *.py subdir1/*.py subdir2/*.py`. Note that with recursive globbing in `bash` or `zsh`, you can apply stage 1 to all source files recursively with:

```
futurize --stage1 -w .
```

**1c**. Commit all changes

**1d**. Re-run the test suite on Py2 and fix any errors.

See forwards-conversion-stage1 for more info.

### Example error

One relatively common error after conversion is:

```
Traceback (most recent call last):
  ...
  File "/home/user/Install/BleedingEdge/reportlab/tests/test_
↪encrypt.py", line 19, in <module>
    from .test_pdfencryption import parsedoc
ValueError: Attempted relative import in non-package
```

If you get this error, try adding an empty `__init__.py` file in the package directory. (In this example, in the tests/ directory.) If this doesn't help, and if this message appears for all tests, they must be invoked differently (from the cmd line or e.g. `setup.py`). The way to run a module inside a package on Python 3, or on Python 2 with `absolute_import` in effect, is:

```
python -m tests.test_platypus_xref
```

(For more info, see PEP 328 and the PEP 8 section on absolute imports.)

### Step 2: working Py3 code that still supports Py2

The goal for this step is to get the tests passing first on Py3 and then on Py2 again with the help of the `future` package.

**2a**. Run:

```
futurize --stage2 myfolder1/*.py myfolder2/*.py
```

You can view the stage 2 changes to all Python source files recursively with:

```
futurize --stage2 .
```

To apply the changes, add the `-w` argument.

This stage makes further conversions needed to support both Python 2 and 3. These will likely require imports from `future` on Py2 (and sometimes on Py3), such as:

```
from future import standard_library
standard_library.install_aliases()
# ...
from builtins import bytes
from builtins import open
from future.utils import with_metaclass
```

Optionally, you can use the `--unicode-literals` flag to add this import to the top of each module:

```
from __future__ import unicode_literals
```

All strings in the module would then be unicode on Py2 (as on Py3) unless explicitly marked with a `b''` prefix.

If you would like `futurize` to import all the changed builtins to have their Python 3 semantics on Python 2, invoke it like this:

```
futurize --stage2 --all-imports myfolder/*.py
```

**2b**. Re-run your tests on Py3 now. Make changes until your tests pass on Python 3.

**2c**. Commit your changes! :)

**2d**. Now run your tests on Python 2 and notice the errors. Add wrappers from `future` to re-enable Python 2 compatibility. See the *Cheat Sheet: Writing Python 2-3 compatible code* cheat sheet and *What else you need to know* for more info.

After each change, re-run the tests on Py3 and Py2 to ensure they pass on both.

**2e**. You're done! Celebrate! Push your code and announce to the world! Hashtags #python3 #python-future.

### 1.7.3 `pasteurize`: Py3 to Py2/3

Running `pasteurize -w mypy3module.py` turns this Python 3 code:

```
import configparser
import copyreg

class Blah:
    pass
print('Hello', end=None)
```

into this code which runs on both Py2 and Py3:

```
from __future__ import print_function
from future import standard_library
standard_library.install_hooks()

import configparser
import copyreg

class Blah(object):
    pass
print('Hello', end=None)
```

Notice that both `futurize` and `pasteurize` create explicit new-style classes that inherit from `object` on both Python versions, and both refer to stdlib modules (as well as builtins) under their Py3 names.

Note also that the `configparser` module is a special case; there is a full backport available on PyPI (https://pypi.org/project/configparser/), so, as of v0.16.0, `python-future` no longer provides a `configparser` package alias. To use the resulting code on Py2, install the `configparser` backport with `pip install configparser` or by adding it to your `requirements.txt` file.

`pasteurize` also handles the following Python 3 features:

- keyword-only arguments
- metaclasses (using *`with_metaclass()`*)
- extended tuple unpacking (PEP 3132)

To handle function annotations (PEP 3107), see func_annotations.

### 1.7.4 Known limitations

`futurize` and `pasteurize` are useful to automate much of the work of porting, particularly the boring repetitive text substitutions. They also help to flag which parts of the code require attention.

Nevertheless, `futurize` and `pasteurize` are still incomplete and make some mistakes, like 2to3, on which they are based. Please report bugs on GitHub. Contributions to the `lib2to3`-based fixers for `futurize` and `pasteurize` are particularly welcome! Please see *Contributing*.

`futurize` doesn't currently make the following change automatically:

1. Strings containing `\U` produce a `SyntaxError` on Python 3. An example is:

```
s = 'C:\Users'.
```

   Python 2 expands this to `s = 'C:\\Users'`, but Python 3 requires a raw prefix (`r'...'`). This also applies to multi-line strings (including multi-line docstrings).

Also see the tests in `future/tests/test_futurize.py` marked `@expectedFailure` or `@skip` for known limitations.

## 1.8 Frequently Asked Questions (FAQ)

### 1.8.1 Who is this for?

1. People with existing or new Python 3 codebases who wish to provide ongoing Python 2.7 support easily and with little maintenance burden.

2. People who wish to ease and accelerate migration of their Python 2 codebases to Python 3.4+, module by module, without giving up Python 2 compatibility.

### 1.8.2 Why upgrade to Python 3?

> "Python 2 is the next COBOL."
>
> —Alex Gaynor, at PyCon AU 2013

Python 2.7 is the end of the Python 2 line. (See PEP 404.) The language and standard libraries are improving only in Python 3.x.

Python 3.x is a better language and better set of standard libraries than Python 2.x in many ways. Python 3.x is cleaner, less warty, and easier to learn than Python 2. It has better memory efficiency, easier Unicode handling, and powerful new features like the asyncio module.

### 1.8.3 Porting philosophy

#### Why write Python 3-style code?

Here are some quotes:

- "Django's developers have found that attempting to write Python 3 code that's compatible with Python 2 is much more rewarding than the opposite." from the Django docs.
- "Thanks to Python 3 being more strict about things than Python 2 (e.g., bytes vs. strings), the source translation [from Python 3 to 2] 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." from the official guide "Porting Python 2 Code to Python 3" by Brett Cannon.
- "Developer energy should be reserved for addressing real technical difficulties associated with the Python 3 transition (like distinguishing their 8-bit text strings from their binary data). They shouldn't be punished with additional code changes ..." from PEP 414 by Armin Ronacher and Nick Coghlan.

#### Can't I just roll my own Py2/3 compatibility layer?

Yes, but using `python-future` will probably be easier and lead to cleaner code with fewer bugs.

Consider this quote:

> "Duplication of effort is wasteful, and replacing the various home-grown approaches with a standard feature usually ends up making things more readable, and interoperable as well."
>
> —Guido van Rossum (blog post)

`future` also includes various Py2/3 compatibility tools in *`future.utils`* picked from large projects (including IPython, Django, Jinja2, Pandas), which should reduce the burden on every project to roll its own py3k compatibility wrapper module.

#### What inspired this project?

In our Python training courses, we at Python Charmers faced a dilemma: teach people Python 3, which was future-proof but not as useful to them today because of weaker 3rd-party package support, or teach people Python 2, which was more useful today but would require them to change their code and unlearn various habits soon. We searched for ways to avoid polluting the world with more deprecated code, but didn't find a good way.

Also, in attempting to help with porting packages such as scikit-learn to Python 3, I (Ed) was dissatisfied with how much code cruft was necessary to introduce to support Python 2 and 3 from a single codebase (the preferred porting option). Since backward-compatibility with Python 2 may be necessary for at least the next 5 years, one of the promised benefits of Python

3 – cleaner code with fewer of Python 2's warts – was difficult to realize before in practice in a single codebase that supported both platforms.

The goal is to accelerate the uptake of Python 3 and help the strong Python community to remain united around a single version of the language.

### 1.8.4 Maturity

#### How well has it been tested?

`future` is used by several major projects, including mezzanine and ObsPy. It is also currently being used to help with porting 800,000 lines of Python 2 code in Sage to Python 2/3.

Currently `python-future` has over 1000 unit tests. Many of these are straight from the Python 3.3 and 3.4 test suites.

In general, the `future` package itself is in good shape, whereas the `futurize` script for automatic porting is imperfect; chances are it will require some manual cleanup afterwards. The `past` package also needs to be expanded.

#### Is the API stable?

Not yet; `future` is still in beta. Where possible, we will try not to break anything which was documented and used to work. After version 1.0 is released, the API will not change in backward-incompatible ways until a hypothetical version 2.0.

### 1.8.5 Relationship between python-future and other compatibility tools

#### How does this relate to `2to3`?

`2to3` is a powerful and flexible tool that can produce different styles of Python 3 code. It is, however, primarily designed for one-way porting efforts, for projects that can leave behind Python 2 support.

The example at the top of the 2to3 docs demonstrates this. After transformation by `2to3`, `example.py` looks like this:

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

This is Python 3 code that, although syntactically valid on Python 2, is semantically incorrect. On Python 2, it raises an exception for most inputs; worse, it allows arbitrary code execution by the user for specially crafted inputs because of the `eval()` executed by Python 2's `input()` function.

This is not an isolated example; almost every output of `2to3` will need modification to provide backward compatibility with Python 2. As an alternative, the `python-future` project provides a script called `futurize` that is based on `lib2to3` but will produce code that is compatible with both platforms (Py2 and Py3).

### Can I maintain a Python 2 codebase and use 2to3 to automatically convert to Python 3 in the setup script?

This was originally the approach recommended by Python's core developers, but it has some large drawbacks:

1. First, your actual working codebase will be stuck with Python 2's warts and smaller feature set for as long as you need to retain Python 2 compatibility. This may be at least 5 years for many projects, possibly much longer.

2. Second, this approach carries the significant disadvantage that you cannot apply patches submitted by Python 3 users against the auto-generated Python 3 code. (See this talk by Jacob Kaplan-Moss.)

### What is the relationship between `future` and `six`?

`python-future` is a higher-level compatibility layer than `six` that includes more backported functionality from Python 3, more forward-ported functionality from Python 2, and supports cleaner code, but requires more modern Python versions to run.

`python-future` and `six` share the same goal of making it possible to write a single-source codebase that works on both Python 2 and Python 3. `python-future` has the further goal of allowing standard Py3 code to run with almost no modification on both Py3 and Py2. `future` provides a more complete set of support for Python 3's features, including backports of Python 3 builtins such as the `bytes` object (which is very different to Python 2's `str` object) and several standard library modules.

`python-future` supports only Python 2.7+ and Python 3.4+, whereas `six` supports all versions of Python from 2.4 onwards. (See *Which versions of Python does python-future support?*.) If you must support older Python versions, `six` will be essential for you. However, beware that maintaining single-source compatibility with older Python versions is ugly and not fun.

If you can drop support for older Python versions, `python-future` leverages some important features introduced into Python 2.7, such as import hooks, and a comprehensive and well-tested set of backported functionality, to allow you to write more idiomatic, maintainable code with fewer compatibility hacks.

#### What is the relationship between `python-future` and `python-modernize`?

`python-future` contains, in addition to the `future` compatibility package, a `futurize` script that is similar to `python-modernize.py` in intent and design. Both are based heavily on `2to3`.

Whereas `python-modernize` converts Py2 code into a common subset of Python 2 and 3, with `six` as a run-time dependency, `futurize` converts either Py2 or Py3 code into (almost) standard Python 3 code, with `future` as a run-time dependency.

Because `future` provides more backported Py3 behaviours from `six`, the code resulting from `futurize` is more likely to work identically on both Py3 and Py2 with less additional manual porting effort.

### 1.8.6 Platform and version support

#### Which versions of Python does `python-future` support?

Python 2.7, and 3.4+ only.

Python 2.7 introduced many important forward-compatibility features (such as import hooks, `b'...'` literals and `__future__` definitions) that greatly reduce the maintenance burden for single-source Py2/3 compatible code. `future` leverages these features and aims to close the remaining gap between Python 3 and 2.7.

#### Do you support Pypy?

Yes, except for the standard library import hooks (currently). Feedback and pull requests are welcome!

#### Do you support IronPython and/or Jython?

Not sure. This would be nice...

### 1.8.7 Support

#### Is there a mailing list?

Yes, please ask any questions on the python-porting mailing list.

### 1.8.8 Contributing

#### Can I help?

Yes please :) We welcome bug reports, additional tests, pull requests, and stories of either success or failure with using it. Help with the fixers for the `futurize` script is particularly welcome.

#### Where is the repo?

https://github.com/PythonCharmers/python-future.

## 1.9 Standard library incompatibilities

Some standard library interfaces have changed in ways that require different code than normal Py3 code in order to achieve Py2/3 compatibility.

Here we will attempt to document these, together with known workarounds:

Table 1: Standard library incompatibilities

| module | object / feature | section |
|---|---|---|
| `array` | `array` constructor | *array.array()* |
| `array` | `array.read()` method | *array.array.read()* |
| `base64` | `decodebytes()` function | *base64.decodebytes()* and *base64.encodebytes()* |
| `re` | `ASCII` mode | *re.ASCII* |

To contribute to this, please email the python-porting list or send a pull request. See *Contributing*.

### 1.9.1 array.array()

The first argument to `array.array(typecode[, initializer])` must be a native platform string: unicode string on Python 3, byte string on Python 2.

**Python 2::**

```
>>> array.array(b'b')
array.array(b'b')
```

```
>>> array.array(u'u')
TypeError: must be char, not unicode
```

**Python 3::**

```
>>> array.array(b'b')
TypeError: must be a unicode character, not bytes
```

```
>>> array.array(u'b')
array('b')
```

This means that the typecode cannot be specified portably across Python 3 and Python 2 with a single string literal when `from __future__ import unicode_literals` is in effect.

You can use the following code on both Python 3 and Python 2:

```
from __future__ import unicode_literals
from future.utils import bytes_to_native_str
import array

# ...

a = array.array(bytes_to_native_str(b'b'))
```

This was fixed in Python 2.7.11. Since then, `array.array()` now also accepts unicode format typecode.

### 1.9.2 array.array.read()

This method has been removed in Py3. This crops up in e.g. porting `http.client`.

### 1.9.3 base64.decodebytes() and base64.encodebytes()

The `base64` module on Py2 has no `decodebytes` or `encodebytes` functions.

### 1.9.4 re.ASCII

Python 3 code using regular expressions sometimes looks like this (from `urllib.request`):

```
re.compile(r":\d+$", re.ASCII)
```

This enables 'ASCII mode' for regular expressions (see the docs here). Python 2's `re` module has no equivalent mode.

### 1.9.5 struct.pack()

Before Python version 2.7.7, the `struct.pack()` function required a native string as its format argument. For example:

```
>>> from __future__ import unicode_literals
>>> from struct import pack
>>> pack('<4H2I', version, rec_type, build, year, file_hist_flags,␣
↪ver_can_read)
```

raised `TypeError: Struct() argument 1 must be string, not unicode`.

This was fixed in Python 2.7.7. Since then, `struct.pack()` now also accepts unicode format strings.

## 1.10 Older interfaces

In addition to the direct and `install_aliases()` interfaces (described in standard-library-imports), `future` supports four other interfaces to the reorganized standard library. This is largely for historical reasons (for versions prior to 0.14).

### 1.10.1 `future.moves` interface

The `future.moves` interface avoids import hooks. It may therefore be more robust, at the cost of less idiomatic code. Use it as follows:

```
from future.moves import queue
from future.moves import socketserver
from future.moves.http.client import HTTPConnection
# etc.
```

If you wish to achieve the effect of a two-level import such as this:

```
import http.client
```

portably on both Python 2 and Python 3, note that Python currently does not support syntax like this:

```
from future.moves import http.client
```

One workaround is to replace the dot with an underscore:

```
import future.moves.http.client as http_client
```

#### Comparing future.moves and six.moves

`future.moves` and `six.moves` provide a similar Python 3-style interface to the native standard library module definitions.

The major difference is that the `future.moves` package is a real Python package (`future/moves/__init__.py`) with real modules provided as `.py` files, whereas `six.moves` constructs fake `_LazyModule` module objects within the Python code and injects them into the `sys.modules` cache.

The advantage of `six.moves` is that the code fits in a single module that can be copied into a project that seeks to eliminate external dependencies.

The advantage of `future.moves` is that it is likely to be more robust in the face of magic like Django's auto-reloader and tools like `py2exe` and `cx_freeze`. See issues #51, #53, #56, and #63 in the `six` project for more detail of bugs related to the `six.moves` approach.

### 1.10.2 `import_` and `from_import` functions

The functional interface is to use the `import_` and `from_import` functions from `future.standard_library` as follows:

```
from future.standard_library import import_, from_import

http = import_('http.client')
urllib = import_('urllib.request')

urlopen, urlsplit = from_import('urllib.request', 'urlopen',
↪'urlsplit')
```

This interface also works with two-level imports.

### 1.10.3 Context-manager for import hooks

The context-manager interface is via a context-manager called `hooks`:

```
from future.standard_library import hooks
with hooks():
    import socketserver
    import queue
    import configparser
    import test.support
    import html.parser
    from collections import UserList
    from itertools import filterfalse, zip_longest
    from http.client import HttpConnection
    import urllib.request
    # and other moved modules and definitions
```

This interface is straightforward and effective, using PEP 302 import hooks. However, there are reports that this sometimes leads to problems (see issue #238). Until this is resolved, it is probably safer to use direct imports or one of the other import mechanisms listed above.

### 1.10.4 install_hooks() call (deprecated)

The last interface to the reorganized standard library is via a call to `install_hooks()`:

```
from future import standard_library
standard_library.install_hooks()

import urllib
f = urllib.request.urlopen('http://www.python.org/')

standard_library.remove_hooks()
```

If you use this interface, it is recommended to disable the import hooks again after use by calling `remove_hooks()`, in order to prevent the futurized modules from being invoked inadvertently by other modules. (Python does not automatically disable import hooks at the end of a module, but keeps them active for the life of a process unless removed.)

## 1.11 Changes in previous versions

Changes in the most recent major version are here: *What's New*.

### 1.11.1 Changes in version 0.14.3 (2014-12-15)

This is a bug-fix release:

- Expose contents of `thread` (not `dummy_thread`) as `_thread` on Py2 (Issue #124)
- Add signed support for `newint.to_bytes()` (Issue #128)
- Fix `OrderedDict.clear()` on Py2.6 (Issue #125)
- Improve `newrange`: equality and slicing, start/stop/step properties, refactoring (Issues #129, #130)
- Minor doc updates

### 1.11.2 Changes in version 0.14.2 (2014-11-21)

This is a bug-fix release:

- Speed up importing of `past.translation` (Issue #117)
- `html.escape()`: replace function with the more robust one from Py3.4
- `futurize`: avoid displacing encoding comments by `__future__` imports (Issues #97, #10, #121)
- `futurize`: don't swallow exit code (Issue #119)
- Packaging: don't forcibly remove the old build dir in `setup.py` (Issue #108)
- Docs: update further docs and tests to refer to `install_aliases()` instead of `install_hooks()`
- Docs: fix `iteritems` import error in cheat sheet (Issue #120)
- Tests: don't rely on presence of `test.test_support` on Py2 or `test.support` on Py3 (Issue #109)
- Tests: don't override existing `PYTHONPATH` for tests (PR #111)

### 1.11.3 Changes in version 0.14.1 (2014-10-02)

This is a minor bug-fix release:

- Docs: add a missing template file for building docs (Issue #108)
- Tests: fix a bug in error handling while reporting failed script runs (Issue #109)
- `install_aliases()`: don't assume that the `test.test_support` module always exists on Py2 (Issue #109)

### 1.11.4 Changes in version 0.14.0 (2014-10-02)

This is a major new release that offers a cleaner interface for most imports in Python 2/3 compatible code.

Instead of this interface:

```
>>> from future.builtins import str, open, range, dict

>>> from future.standard_library import hooks
>>> with hooks():
...     import queue
...     import configparser
...     import tkinter.dialog
...     # etc.
```

You can now use the following interface for much Python 2/3 compatible code:

```
>>> # Alias for future.builtins on Py2:
>>> from builtins import str, open, range, dict

>>> # Alias for future.moves.* on Py2:
>>> import queue
>>> import configparser
>>> import tkinter.dialog
>>> etc.
```

Notice that the above code will run on Python 3 even without the presence of the `future` package. Of the 44 standard library modules that were refactored with PEP 3108, 30 are supported with direct imports in this manner. (These are listed here: direct-imports.)

The other 14 standard library modules that kept the same top-level names in Py3.x are not supported with this direct import interface on Py2. These include the 5 modules in the Py3 `urllib` package. These modules are accessible through the following interface (as well as the interfaces offered in previous versions of `python-future`):

```
from future.standard_library import install_aliases
install_aliases()

from collections import UserDict, UserList, UserString
import dbm.gnu
from itertools import filterfalse, zip_longest
from subprocess import getoutput, getstatusoutput
from sys import intern
import test.support
from urllib.request import urlopen
from urllib.parse import urlparse
# etc.
from collections import Counter, OrderedDict      # backported to␣
↪Py2.6
```

The complete list of packages supported with this interface is here: list-standard-library-refactored.

For more information on these and other interfaces to the standard library, see standard-library-imports.

### Bug fixes

- This release expands the `future.moves` package to include most of the remaining modules that were moved in the standard library reorganization (PEP 3108). (Issue #104)
- This release also removes the broken `--doctests_only` option from the `futurize` and `pasteurize` scripts for now. (Issue #103)

### Internal cleanups

The project folder structure has changed. Top-level packages are now in a `src` folder and the tests have been moved into a project-level `tests` folder.

The following deprecated internal modules have been removed (Issue #80):

- `future.utils.encoding` and `future.utils.six`.

### Deprecations

The following internal functions have been deprecated and will be removed in a future release:

- `future.standard_library.scrub_py2_sys_modules`
- `future.standard_library.scrub_future_sys_modules`

## 1.11.5 Changes in version 0.13.1 (2014-09-23)

This is a bug-fix release:

- Fix (multiple) inheritance of `future.builtins.object` with metaclasses (Issues #91, #96)
- Fix `futurize`'s refactoring of `urllib` imports (Issue #94)
- Fix `futurize --all-imports` (Issue #101)
- Fix `futurize --output-dir` logging (Issue #102)
- Doc formatting fix (Issues #98, #100)

## 1.11.6 Changes in version 0.13.0 (2014-08-13)

This is mostly a clean-up release. It adds some small new compatibility features and fixes several bugs.

### Deprecations

The following unused internal modules are now deprecated. They will be removed in a future release:

- `future.utils.encoding` and `future.utils.six`.

(Issue #80). See here for the rationale for unbundling them.

### New features

- Docs: Add *Cheat Sheet: Writing Python 2-3 compatible code* from Ed Schofield's PyConAU 2014 talk.
- Add `newint.to_bytes()` and `newint.from_bytes()`. (Issue #85)
- Add `future.utils.raise_from` as an equivalent to Py3's `raise ... from ...` syntax. (Issue #86)
- Add `past.builtins.oct()` function.
- Add backports for Python 2.6 of `subprocess.check_output()`, `itertools.combinations_with_replacement()`, and `functools.cmp_to_key()`.

### Bug fixes

- Use a private logger instead of the global logger in `future.standard_library` (Issue #82). This restores compatibility of the standard library hooks with `flask`. (Issue #79)
- Stage 1 of `futurize` no longer renames `next` methods to `__next__` (Issue #81). It still converts `obj.next()` method calls to `next(obj)` correctly.
- Prevent introduction of a second set of parentheses in `print()` calls in some further cases.
- Fix `isinstance` checks for subclasses of future types. (Issue #89)
- Be explicit about encoding file contents as UTF-8 in unit tests. (Issue #63) Useful for building RPMs and in other environments where `LANG=C`.
- Fix for 3-argument `pow(x, y, z)` with `newint` arguments. (Thanks to @str4d.) (Issue #87)

## 1.11.7 Changes in version 0.12.4 (2014-07-18)

- Fix upcasting behaviour of `newint`. (Issue #76)

## 1.11.8 Changes in version 0.12.3 (2014-06-19)

- Add "official Python 3.4 support": Py3.4 is now listed among the PyPI Trove classifiers and the tests now run successfully on Py3.4. (Issue #67)
- Add backports of `collections.OrderedDict` and `collections.Counter` for Python 2.6. (Issue #52)
- Add `--version` option for `futurize` and `pasteurize` scripts. (Issue #57)
- Fix `future.utils.ensure_new_type` with `long` input. (Issue #65)

- Remove some false alarms on checks for ambiguous fixer names with `futurize -f ...`.
- **Testing fixes:**
  - Don't hard-code Python interpreter command in tests. (Issue #62)
  - Fix deprecated `unittest` usage in Py3. (Issue #62)
  - Be explicit about encoding temporary file contents as UTF-8 for when `LANG=C` (e.g., when building an RPM). (Issue #63)
  - All undecorated tests are now passing again on Python 2.6, 2.7, 3.3, and 3.4 (thanks to Elliott Sales de Andrade).
- **Docs:**
  - Add list of fixers used by `futurize`. (Issue #58)
  - Add list of contributors to the Credits page.

### 1.11.9 Changes in version 0.12.2 (2014-05-25)

- Add `bytes.maketrans()` method. (Issue #51)
- Add support for Python versions between 2.7.0 and 2.7.3 (inclusive). (Issue #53)
- Bug fix for `newlist(newlist([1, 2, 3]))`. (Issue #50)

### 1.11.10 Changes in version 0.12.1 (2014-05-14)

- Python 2.6 support: `future.standard_library` now isolates the `importlib` dependency to one function (`import_`) so the `importlib` backport may not be needed.
- Doc updates

### 1.11.11 Changes in version 0.12.0 (2014-05-06)

The major new feature in this version is improvements in the support for the reorganized standard library (PEP 3108) and compatibility of the import mechanism with 3rd-party modules.

### More robust standard-library import hooks

**Note: backwards-incompatible change:** As previously announced (see *Deprecated feature: auto-installation of standard-library import hooks*), the import hooks must now be enabled explicitly, as follows:

```
from future import standard_library
with standard_library.hooks():
    import html.parser
    import http.client
    ...
```

This now causes these modules to be imported from `future.moves`, a new package that provides wrappers over the native Python 2 standard library with the new Python 3 organization. As a consequence, the import hooks provided in `future.standard_library` are now fully compatible with the Requests library.

The functional interface with `install_hooks()` is still supported for backwards compatibility:

```
from future import standard_library
standard_library.install_hooks():

import html.parser
import http.client
...
standard_library.remove_hooks()
```

Explicit installation of import hooks allows finer-grained control over whether they are enabled for other imported modules that provide their own Python 2/3 compatibility layer. This also improves compatibility of `future` with tools like `py2exe`.

### `newobject` base object defines fallback Py2-compatible special methods

There is a new `future.types.newobject` base class (available as `future.builtins.object`) that can streamline Py2/3 compatible code by providing fallback Py2-compatible special methods for its subclasses. It currently provides `next()` and `__nonzero__()` as fallback methods on Py2 when its subclasses define the corresponding Py3-style `__next__()` and `__bool__()` methods.

This obviates the need to add certain compatibility hacks or decorators to the code such as the `@implements_iterator` decorator for classes that define a Py3-style `__next__` method.

In this example, the code defines a Py3-style iterator with a `__next__` method. The `object` class defines a `next` method for Python 2 that maps to `__next__`:

```
from future.builtins import object

class Upper(object):
    def __init__(self, iterable):
```

(continues on next page)

(continued from previous page)

```
        self._iter = iter(iterable)
    def __next__(self):                  # note the Py3 interface
        return next(self._iter).upper()
    def __iter__(self):
        return self

assert list(Upper('hello')) == list('HELLO')
```

`newobject` defines other Py2-compatible special methods similarly: currently these include `__nonzero__` (mapped to `__bool__`) and `__long__` (mapped to `__int__`).

Inheriting from `newobject` on Python 2 is safe even if your class defines its own Python 2-style `__nonzero__` and `next` and `__long__` methods. Your custom methods will simply override those on the base class.

On Python 3, as usual, `future.builtins.object` simply refers to `builtins.object`.

### `past.builtins` module improved

The `past.builtins` module is much more compatible with the corresponding builtins on Python 2; many more of the Py2 unit tests pass on Py3. For example, functions like `map()` and `filter()` now behave as they do on Py2 with with `None` as the first argument.

The `past.builtins` module has also been extended to add Py3 support for additional Py2 constructs that are not adequately handled by `lib2to3` (see Issue #37). This includes new `execfile()` and `cmp()` functions. `futurize` now invokes imports of these functions from `past.builtins`.

### `surrogateescape` error handler

The `newstr` type (`future.builtins.str`) now supports a backport of the Py3.x `'surrogateescape'` error handler for preserving high-bit characters when encoding and decoding strings with unknown encodings.

### `newlist` type

There is a new `list` type in `future.builtins` that offers `.copy()` and `.clear()` methods like the `list` type in Python 3.

### `listvalues` and `listitems`

`future.utils` now contains helper functions `listvalues` and `listitems`, which provide Python 2-style list snapshotting semantics for dictionaries in both Python 2 and Python 3.

These came out of the discussion around Nick Coghlan's now-withdrawn PEP 469.

There is no corresponding `listkeys(d)` function; use `list(d)` instead.

### Tests

The number of unit tests has increased from 600 to over 800. Most of the new tests come from Python 3.3's test suite.

### Refactoring of `future.standard_library.*` -> `future.backports`

The backported standard library modules have been moved to `future.backports` to make the distinction clearer between these and the new `future.moves` package.

### Backported `http.server` and `urllib` modules

Alpha versions of backports of the `http.server` and `urllib` module from Python 3.3's standard library are now provided in `future.backports`.

Use them like this:

```
from future.backports.urllib.request import Request     # etc.
from future.backports.http import server as http_server
```

Or with this new interface:

```
from future.standard_library import import_, from_import

Request = from_import('urllib.request', 'Request', backport=True)
http = import_('http.server', backport=True)
```

### Internal refactoring

The `future.builtins.types` module has been moved to `future.types`. Likewise, `past.builtins.types` has been moved to `past.types`. The only user-visible effect of this is to change `repr(type(obj))` for instances of these types. For example:

```
>>> from future.builtins import bytes
>>> bytes(b'abc')
>>> type(b)
future.types.newbytes.newbytes
```

Instead of:

```
>>> type(b)             # prior to v0.12
future.builtins.types.newbytes.newbytes
```

### Bug fixes

Many small improvements and fixes have been made across the project. Some highlights are:

- Fixes and updates from Python 3.3.5 have been included in the backported standard library modules.
- Scrubbing of the `sys.modules` cache performed by `remove_hooks()` (also called by the `suspend_hooks` and `hooks` context managers) is now more conservative.
- The `fix_next` and `fix_reduce` fixers have been moved to stage 1 of `futurize`.
- `futurize`: Shebang lines such as `#!/usr/bin/env python` and source code file encoding declarations like `# -*- coding=utf-8 -*-` are no longer occasionally displaced by `from __future__ import ...` statements. (Issue #10)
- Improved compatibility with `py2exe` (Issue #31).
- The `future.utils.bytes_to_native_str` function now returns a platform-native string object and `future.utils.native_str_to_bytes` returns a `newbytes` object on Py2. (Issue #47).
- The backported `http.client` module and related modules use other new backported modules such as `email`. As a result they are more compliant with the Python 3.3 equivalents.

## 1.11.12 Changes in version 0.11.4 (2014-05-25)

This release contains various small improvements and fixes:

- This release restores Python 2.6 compatibility. (Issue #42)
- The `fix_absolute_import` fixer now supports Cython `.pyx` modules. (Issue #35)
- Right-division with `newint` objects is fixed. (Issue #38)
- The `fix_dict` fixer has been moved to stage2 of `futurize`.
- Calls to `bytes(string, encoding[, errors])` now work with `encoding` and `errors` passed as positional arguments. Previously this only worked if `encoding` and `errors` were passed as keyword arguments.
- The 0-argument `super()` function now works from inside static methods such as `__new__`. (Issue #36)
- `future.utils.native(d)` calls now work for `future.builtins.dict` objects.

### 1.11.13 Changes in version 0.11.3 (2014-02-27)

This release has improvements in the standard library import hooks mechanism and its compatibility with 3rd-party modules:

#### Improved compatibility with `requests`

The `__exit__` function of the `hooks` context manager and the `remove_hooks` function both now remove submodules of `future.standard_library` from the `sys.modules` cache. Therefore this code is now possible on Python 2 and 3:

```
from future import standard_library
standard_library.install_hooks()
import http.client
standard_library.remove_hooks()
import requests

data = requests.get('http://www.google.com')
```

Previously, this required manually removing `http` and `http.client` from `sys.modules` before importing `requests` on Python 2.x. (Issue #19)

This change should also improve the compatibility of the standard library hooks with any other module that provides its own Python 2/3 compatibility code.

Note that the situation will improve further in version 0.12; import hooks will require an explicit function call or the `hooks` context manager.

#### Conversion scripts explicitly install import hooks

The `futurize` and `pasteurize` scripts now add an explicit call to `install_hooks()` to install the standard library import hooks. These scripts now add these two lines:

```
from future import standard_library
standard_library.install_hooks()
```

instead of just the first one. The next major version of `future` (0.12) will require the explicit call or use of the `hooks` context manager. This will allow finer-grained control over whether import hooks are enabled for other imported modules, such as `requests`, which provide their own Python 2/3 compatibility code.

#### `futurize` script no longer adds `unicode_literals` by default

There is a new `--unicode-literals` flag to `futurize` that adds the import:

```
from __future__ import unicode_literals
```

to the top of each converted module. Without this flag, `futurize` now no longer adds this import. (Issue #22)

The `pasteurize` script for converting from Py3 to Py2/3 still adds `unicode_literals`. (See the comments in Issue #22 for an explanation.)

### 1.11.14 Changes in version 0.11 (2014-01-28)

There are several major new features in version 0.11.

#### `past` package

The python-future project now provides a `past` package in addition to the `future` package. Whereas `future` provides improved compatibility with Python 3 code to Python 2, `past` provides support for using and interacting with Python 2 code from Python 3. The structure reflects that of `future`, with `past.builtins` and `past.utils`. There is also a new `past.translation` package that provides transparent translation of Python 2 code to Python 3. (See below.)

One purpose of `past` is to ease module-by-module upgrades to codebases from Python 2. Another is to help with enabling Python 2 libraries to support Python 3 without breaking the API they currently provide. (For example, user code may expect these libraries to pass them Python 2's 8-bit strings, rather than Python 3's `bytes` object.) A third purpose is to help migrate projects to Python 3 even if one or more dependencies are still on Python 2.

Currently `past.builtins` provides forward-ports of Python 2's `str` and `dict` objects, `basestring`, and list-producing iterator functions. In later releases, `past.builtins` will be used internally by the `past.translation` package to help with importing and using old Python 2 modules in a Python 3 environment.

#### Auto-translation of Python 2 modules upon import

`past` provides an experimental `translation` package to help with importing and using old Python 2 modules in a Python 3 environment.

This is implemented using import hooks that attempt to automatically translate Python 2 modules to Python 3 syntax and semantics upon import. Use it like this:

```
$ pip3 install plotrique==0.2.5-7 --no-compile   # to ignore␣
↪SyntaxErrors
$ python3
```

Then pass in a whitelist of module name prefixes to the `past.translation.autotranslate()` function. Example:

```
>>> from past.translation import autotranslate
>>> autotranslate(['plotrique'])
>>> import plotrique
```

This is intended to help you migrate to Python 3 without the need for all your code's dependencies to support Python 3 yet. It should be used as a last resort; ideally Python 2-only dependencies should be ported properly to a Python 2/3 compatible codebase using a tool like `futurize` and the changes should be pushed to the upstream project.

For more information, see translation.

### Separate `pasteurize` script

The functionality from `futurize --from3` is now in a separate script called `pasteurize`. Use `pasteurize` when converting from Python 3 code to Python 2/3 compatible source. For more information, see backwards-conversion.

#### `pow()`

There is now a `pow()` function in `future.builtins.misc` that behaves like the Python 3 `pow()` function when raising a negative number to a fractional power (returning a complex number).

### `input()` no longer disabled globally on Py2

Previous versions of `future` deleted the `input()` function from `__builtin__` on Python 2 as a security measure. This was because Python 2's `input()` function allows arbitrary code execution and could present a security vulnerability on Python 2 if someone expects Python 3 semantics but forgets to import `input` from `future.builtins`. This behaviour has been reverted, in the interests of broadening the compatibility of `future` with other Python 2 modules.

Please remember to import `input` from `future.builtins` if you use `input()` in a Python 2/3 compatible codebase.

### Deprecated feature: auto-installation of standard-library import hooks

Previous versions of `python-future` installed import hooks automatically upon importing the `standard_library` module from `future`. This has been deprecated in order to improve robustness and compatibility with modules like `requests` that already perform their own single-source Python 2/3 compatibility.

As of v0.12, importing `future.standard_library` will no longer install import hooks by default. Instead, please install the import hooks explicitly as follows:

```
from future import standard_library
standard_library.install_hooks()
```

And uninstall them after your import statements using:

```
standard_library.remove_hooks()
```

*Note*: This is a backward-incompatible change.

#### Internal changes

The internal `future.builtins.backports` module has been renamed to `future.builtins.types`. This will change the `repr` of `future` types but not their use.

### 1.11.15 Changes in version 0.10.2 (2014-01-11)

#### New context-manager interface to `standard_library.hooks`

There is a new context manager `future.standard_library.hooks`. Use it like this:

```
from future import standard_library
with standard_library.hooks():
    import queue
    import configserver
    from http.client import HTTPConnection
    # etc.
```

If not using this context manager, it is now encouraged to add an explicit call to `standard_library.install_hooks()` as follows:

```
from future import standard_library
standard_library.install_hooks()

import queue
import html
import http.client
# etc.
```

And to remove the hooks afterwards with:

```
standard_library.remove_hooks()
```

The functions `install_hooks()` and `remove_hooks()` were previously called `enable_hooks()` and `disable_hooks()`. The old names are deprecated (but are still available as aliases).

As usual, this feature has no effect on Python 3.

## 1.11.16 Changes in version 0.10.0 (2013-12-02)

### Backported `dict` type

`future.builtins` now provides a Python 2 `dict` subclass whose `keys()`, `values()`, and `items()` methods produce memory-efficient iterators. On Python 2.7, these also have the same set-like view behaviour as on Python 3. This can streamline code needing to iterate over large dictionaries. For example:

```
from __future__ import print_function
from future.builtins import dict, range

squares = dict({i: i**2 for i in range(10**7)})

assert not isinstance(d.items(), list)
# Because items() is memory-efficient, so is this:
square_roots = dict((i_squared, i) for (i, i_squared) in squares.
↪items())
```

For more information, see *dict*.

### Utility functions `raise_` and `exec_`

The functions `raise_with_traceback()` and `raise_()` were added to `future.utils` to offer either the Python 3.x or Python 2.x behaviour for raising exceptions. Thanks to Joel Tratner for the contribution of these. `future.utils.reraise()` is now deprecated.

A portable `exec_()` function has been added to `future.utils` from `six`.

### Bugfixes

- Fixed `newint.__divmod__`
- Improved robustness of installing and removing import hooks in *future.standard_library*
- v0.10.1: Fixed broken `pip install future` on Py3

## 1.11.17 Changes in version 0.9 (2013-11-06)

### `isinstance` checks are supported natively with backported types

The `isinstance` function is no longer redefined in `future.builtins` to operate with the backported `int`, `bytes` and `str`. `isinstance` checks with the backported types now work correctly by default; we achieve this through overriding the `__instancecheck__` method of metaclasses of the backported types.

For more information, see *isinstance*.

#### `futurize`: minimal imports by default

By default, the `futurize` script now only adds the minimal set of imports deemed necessary.

There is now an `--all-imports` option to the `futurize` script which gives the previous behaviour, which is to add all `__future__` imports and `from future.builtins import *` imports to every module. (This even applies to an empty `__init__.py` file.)

#### Looser type-checking for the backported `str` object

Now the `future.builtins.str` object behaves more like the Python 2 `unicode` object with regard to type-checking. This is to work around some bugs / sloppiness in the Python 2 standard library involving mixing of byte-strings and unicode strings, such as `os.path.join` in `posixpath.py`.

`future.builtins.str` still raises the expected `TypeError` exceptions from Python 3 when attempting to mix it with `future.builtins.bytes`.

#### `suspend_hooks()` context manager added to `future.standard_library`

Pychecker (as of v0.6.1)'s `checker.py` attempts to import the `builtins` module as a way of determining whether Python 3 is running. Since this succeeds when `from future import standard_library` is in effect, this check does not work and pychecker sets the wrong value for its internal `PY2` flag is set.

To work around this, `future` now provides a context manager called `suspend_hooks` that can be used as follows:

```
from future import standard_library
...
with standard_library.suspend_hooks():
    from pychecker.checker import Checker
```

### 1.11.18 Changes in version 0.8 (2013-10-28)

#### Python 2.6 support

`future` now includes support for Python 2.6.

To run the `future` test suite on Python 2.6, this additional package is needed:

```
pip install unittest2
```

`http.server` also requires the `argparse` package:

```
pip install argparse
```

#### Unused modules removed

The `future.six` module has been removed. `future` doesn't require `six` (and hasn't since version 0.3). If you need support for Python versions before 2.6, `six` is the best option. `future` and `six` can be installed alongside each other easily if needed.

The unused `hacks` module has also been removed from the source tree.

#### `isinstance()` added to `future.builtins` (v0.8.2)

It is now possible to use `isinstance()` calls normally after importing `isinstance` from `future.builtins`. On Python 2, this is specially defined to be compatible with `future`'s backported `int`, `str`, and `bytes` types, as well as handling Python 2's `int`/`long` distinction.

The result is that code that uses `isinstance` to perform type-checking of ints, strings, and bytes should now work identically on Python 2 as on Python 3.

The utility functions `isint`, `istext`, and `isbytes` provided before for compatible type-checking across Python 2 and 3 in *`future.utils`* are now deprecated.

### 1.11.19 Summary of all changes

**v0.15.0:**

- Full backports of `urllib.parse` and other `urllib` submodules are exposed by `install_aliases()`.
- `tkinter.ttk` support
- Initial `surrogateescape` support
- Additional backports: `collections`, `http` constants, etc.
- Bug fixes

**v0.14.3:**

- Bug fixes

**v0.14.2:**

- Bug fixes

**v0.14.1:**

- Bug fixes

**v0.14.0:**

- New top-level `builtins` package on Py2 for cleaner imports. Equivalent to `future.builtins`
- New top-level packages on Py2 with the same names as Py3 standard modules: `configparser`, `copyreg`, `html`, `http`, `xmlrpc`, `winreg`

**v0.13.1:**

- Bug fixes

**v0.13.0:**

- Cheat sheet for writing Python 2/3 compatible code
- `to_int` and `from_int` methods for `newbytes`
- Bug fixes

**v0.12.0:**

- Add `newobject` and `newlist` types
- Improve compatibility of import hooks with `Requests`, `py2exe`
- No more auto-installation of import hooks by `future.standard_library`
- New `future.moves` package
- `past.builtins` improved
- `newstr.encode(..., errors='surrogateescape')` supported
- Refactoring: `future.standard_library` submodules -> `future.backports`
- Refactoring: `future.builtins.types` -> `future.types`
- Refactoring: `past.builtins.types` -> `past.types`
- New `listvalues` and `listitems` functions in `future.utils`
- Many bug fixes to `futurize`, `future.builtins`, etc.

**v0.11.4:**

- Restore Py2.6 compatibility

**v0.11.3:**

- The `futurize` and `pasteurize` scripts add an explicit call to `future.standard_library.install_hooks()` whenever modules affected by PEP 3108 are imported.
- The `future.builtins.bytes` constructor now accepts `frozenset` objects as on Py3.

**v0.11.2:**

- The `past.translation.autotranslate` feature now finds modules to import more robustly and works with Python eggs.

**v0.11.1:**

- Update to `requirements_py26.txt` for Python 2.6. Small updates to docs and tests.

**v0.11:**

- New `past` package with `past.builtins` and `past.translation` modules.

**v0.10.2:**

- Improvements to stdlib hooks. New context manager: `future.standard_library.hooks()`.
- New `raise_` and `raise_with_traceback` functions in `future.utils`.

**v0.10:**

- New backported `dict` object with set-like `keys`, `values`, `items`

**v0.9:**

- `isinstance()` hack removed in favour of `__instancecheck__` on the metaclasses of the backported types
- `futurize` now only adds necessary imports by default
- Looser type-checking by `future.builtins.str` when combining with Py2 native byte-strings.

**v0.8.3:**

- New `--all-imports` option to `futurize`
- Fix bug with `str.encode()` with encoding as a non-keyword arg

**v0.8.2:**

- New `isinstance` function in *`future.builtins`*. This obviates and deprecates the utility functions for type-checking in *`future.utils`*.

**v0.8.1:**

- Backported `socketserver.py`. Fixes sporadic test failures with `http.server` (related to threading and old-style classes used in Py2.7's `SocketServer.py`).
- Move a few more safe `futurize` fixes from stage2 to stage1
- Bug fixes to *`future.utils`*

**v0.8:**

- Added Python 2.6 support
- Removed unused modules: `future.six` and `future.hacks`
- Removed undocumented functions from *`future.utils`*

**v0.7:**

- Added a backported Py3-like `int` object (inherits from `long`).
- Added utility functions for type-checking and docs about `isinstance` uses/alternatives.
- Fixes and stricter type-checking for `bytes` and `str` objects

- Added many more tests for the `futurize` script
- We no longer disable obsolete Py2 builtins by default with `from future.builtins import *`. Use `from future.builtins.disabled import *` instead.

**v0.6:**

- Added a backported Py3-like `str` object (inherits from Py2's `unicode`)
- Removed support for the form `from future import *`; use `from future.builtins import *` instead

**v0.5.3:**

- Doc improvements

**v0.5.2:**

- Add lots of docs and a Sphinx project

**v0.5.1:**

- Upgraded included `six` module (included as `future.utils.six`) to v1.4.1
- `http.server` module backported
- `bytes.split()` and `.rsplit()` bugfixes

**v0.5.0:**

- Added backported Py3-like `bytes` object

**v0.4.2:**

- Various fixes

**v0.4.1:**

- Added `open()` (from `io` module on Py2)
- Improved docs

**v0.4.0:**

- Added various useful compatibility functions to *`future.utils`*
- Reorganized package: moved all builtins to *`future.builtins`*; moved all stdlib things to `future.standard_library`
- Renamed `python-futurize` console script to `futurize`
- Moved `future.six` to `future.utils.six` and pulled the most relevant definitions to *`future.utils`*.
- More improvements to "Py3 to both" conversion (`futurize.py --from3`)

**v0.3.5:**

- Fixed broken package setup ("package directory 'libfuturize/tests' does not exist")

**v0.3.4:**

- Added `itertools.zip_longest`
- Updated `2to3_backcompat` tests to use `futurize.py`
- Improved `libfuturize` fixers: correct order of imports; add imports only when necessary (except `absolute_import` currently)

**v0.3.3:**

- Added `python-futurize` console script
- Added `itertools.filterfalse`
- Removed docs about unfinished backports (`urllib` etc.)
- Removed old Py2 syntax in some files that breaks py3 `setup.py install`

**v0.3.2:**

- Added `test.support` module
- Added `UserList`, `UserString`, `UserDict` classes to `collections` module
- Removed `int` -> `long` mapping
- Added backported `_markupbase.py` etc. with new-style classes to fix travis-ci build problems
- Added working `html` and `http.client` backported modules

**v0.3.0:**

- Generalized import hooks to allow dotted imports
- Added backports of `urllib`, `html`, `http` modules from Py3.3 stdlib using `future`
- Added `futurize` script for automatically turning Py2 or Py3 modules into cross-platform Py3 modules
- Renamed `future.standard_library_renames` to `future.standard_library`. (No longer just renames, but backports too.)

**v0.2.2.1:**

- Small bug fixes to get tests passing on travis-ci.org

**v0.2.1:**

- Small bug fixes

**v0.2.0:**

- `Features` module renamed to `modified_builtins`
- New functions added: `round()`, `input()`
- No more namespace pollution as a policy:

```
from future import *
```

should have no effect on Python 3. On Python 2, it only shadows the builtins; it doesn't introduce any new names.

- End-to-end tests with Python 2 code and `2to3` now work

**v0.1.0:**

- first version with tests!
- removed the inspect-module magic

**v0.0.x:**

- initial releases. Use at your peril.

## 1.12 Licensing and credits

### 1.12.1 Licence

The software is distributed under an MIT licence. The text is as follows (from `LICENSE.txt`):

```
Copyright (c) 2013-2019 Python Charmers Pty Ltd, Australia

Permission is hereby granted, free of charge, to any person␣
↪obtaining a copy
of this software and associated documentation files (the "Software
↪"), to deal
in the Software without restriction, including without limitation␣
↪the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/
↪or sell
copies of the Software, and to permit persons to whom the Software␣
↪is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be␣
↪included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,␣
↪EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF␣
↪MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT␣
↪SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR␣
↪OTHER
```

(continues on next page)

(continued from previous page)

```
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,␣
↪ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER␣
↪DEALINGS IN
THE SOFTWARE.
```

### 1.12.2 Sponsors

Python Charmers Pty Ltd, Australia, and Python Charmers Pte Ltd, Singapore. http://pythoncharmers.com

Pinterest https://opensource.pinterest.com/

### 1.12.3 Maintainer

Python-Future is currently maintained by Jordan M. Adler <jordan.m.adler@gmail.com>.

### 1.12.4 Authors

Python-Future is largely written by Ed Schofield <ed@pythoncharmers.com> with the help of various contributors:

- Jordan Adler
- Jeroen Akkerman
- Kyle Altendorf
- Grant Bakker
- Jacob Beck
- Fumihiro (Ben) Bessho
- Shiva Bhusal
- Nate Bogdanowicz
- Tomer Chachamu
- Christian Clauss
- Denis Cornehl
- Nicolas Delaby
- Chad Dombrova
- Jon Dufresne
- Corey Farwell

- Eric Firing
- Joe Gordon
- Maximilian Hils
- Miro Hrončok
- Mark Huang
- Martijn Jacobs
- Michael Joseph
- Waldemar Kornewald
- Alexey Kotlyarov
- Steve Kowalik
- Lion Krischer
- Marcin Kuzminski
- Joshua Landau
- German Larrain
- Chris Lasher
- ghanshyam lele
- Calum Lind
- Tobias Megies
- Anika Mukherji
- Jon Parise
- Matthew Parnell
- Tom Picton
- Miga Purg
- Éloi Rivard
- Sesh Sadasivam
- Elliott Sales de Andrade
- Aiden Scandella
- Yury Selivanov
- Tim Shaffer
- Sameera Somisetty
- Louis Sautier
- Gregory P. Smith

- Chase Sterling
- Daniel Szoska
- Flaviu Tamas
- Jeff Tratner
- Tim Tröndle
- Brad Walker
- Andrew Wason
- Jeff Widman
- Dan Yeaw
- Hackalog (GitHub user)
- lsm (GiHub user)
- Mystic-Mirage (GitHub user)
- str4d (GitHub user)
- ucodery (GitHub user)
- urain39 (GitHub user)
- 9seconds (GitHub user)
- Varriount (GitHub user)

### Suggestions and Feedback

- Chris Adams
- Martijn Faassen
- Joe Gordon
- Lion Krischer
- Danielle Madeley
- Val Markovic
- wluebbe (GitHub user)

### 1.12.5 Other Credits

- The backported `super()` and `range()` functions are derived from Ryan Kelly's `magicsuper` module and Dan Crosta's `xrange` module.
- The `futurize` and `pasteurize` scripts use `lib2to3`, `lib3to2`, and parts of Armin Ronacher's `python-modernize` code.
- The `python_2_unicode_compatible` decorator is from Django. The `implements_iterator` and `with_metaclass` decorators are from Jinja2.
- The `exec_` function and some others in `future.utils` are from the `six` module by Benjamin Peterson.
- The `raise_` and `raise_with_traceback` functions were contributed by Jeff Tratner.
- A working version of `raise_from` was contributed by Varriount (GitHub).
- Documentation is generated with Sphinx using the `sphinx-bootstrap` theme.
- `past.translation` is inspired by and borrows some code from Sanjay Vinip's `uprefix` module.

## 1.13 API Reference (in progress)

**NOTE: This page is still a work in progress... We need to go through our docstrings and make them sphinx-compliant, and figure out how to improve formatting with the sphinx-bootstrap-theme plugin. Pull requests would be very welcome.**

### 1.13.1 future.builtins Interface

A module that brings in equivalents of the new and modified Python 3 builtins into Py2. Has no effect on Py3.

See the docs here (`docs/what-else.rst`) for more information.

### 1.13.2 Backported types from Python 3

This module contains backports the data types that were significantly changed in the transition from Python 2 to Python 3.

- an implementation of Python 3's bytes object (pure Python subclass of Python 2's builtin 8-bit str type)
- an implementation of Python 3's str object (pure Python subclass of Python 2's builtin unicode type)
- a backport of the range iterator from Py3 with slicing support

It is used as follows:

```
from __future__ import division, absolute_import, print_function
from builtins import bytes, dict, int, range, str
```

to bring in the new semantics for these functions from Python 3. And then, for example:

```
b = bytes(b'ABCD')
assert list(b) == [65, 66, 67, 68]
assert repr(b) == "b'ABCD'"
assert [65, 66] in b

# These raise TypeErrors:
# b + u'EFGH'
# b.split(u'B')
# bytes(b',').join([u'Fred', u'Bill'])


s = str(u'ABCD')

# These raise TypeErrors:
# s.join([b'Fred', b'Bill'])
# s.startswith(b'A')
# b'B' in s
# s.find(b'A')
# s.replace(u'A', b'a')

# This raises an AttributeError:
# s.decode('utf-8')

assert repr(s) == 'ABCD'      # consistent repr with Py3 (no u␣
↪prefix)
```

(continues on next page)

(continued from previous page)

```
for i in range(10**11)[:10]:
    pass
```

and:

```
class VerboseList(list):
    def append(self, item):
        print('Adding an item')
        super().append(item)        # new simpler super() function
```

### For more information:

- future.types.newbytes
- future.types.newdict
- future.types.newint
- future.types.newobject
- future.types.newrange
- future.types.newstr

### Notes

### range()

`range` is a custom class that backports the slicing behaviour from Python 3 (based on the `xrange` module by Dan Crosta). See the `newrange` module docstring for more details.

### super()

`super()` is based on Ryan Kelly's `magicsuper` module. See the `newsuper` module docstring for more details.

#### round()

Python 3 modifies the behaviour of `round()` to use "Banker's Rounding". See http://stackoverflow.com/a/10825998. See the `newround` module docstring for more details.

### 1.13.3 future.standard_library Interface

Python 3 reorganized the standard library (PEP 3108). This module exposes several standard library modules to Python 2 under their new Python 3 names.

It is designed to be used as follows:

```
from future import standard_library
standard_library.install_aliases()
```

And then these normal Py3 imports work on both Py3 and Py2:

```
import builtins
import copyreg
import queue
import reprlib
import socketserver
import winreg    # on Windows only
import test.support
import html, html.parser, html.entites
import http, http.client, http.server
import http.cookies, http.cookiejar
import urllib.parse, urllib.request, urllib.response, urllib.error,␣
↪urllib.robotparser
import xmlrpc.client, xmlrpc.server

import _thread
import _dummy_thread
import _markupbase

from itertools import filterfalse, zip_longest
from sys import intern
from collections import UserDict, UserList, UserString
from collections import OrderedDict, Counter, ChainMap     # even␣
↪on Py2.6
from subprocess import getoutput, getstatusoutput
from subprocess import check_output              # even on Py2.6
```

(The renamed modules and functions are still available under their old names on Python 2.)

This is a cleaner alternative to this idiom (see http://docs.pythonsprints.com/python3_porting/py-porting.html):

```
try:
    import queue
```

(continues on next page)

(continued from previous page)

```
except ImportError:
    import Queue as queue
```

### Limitations

We don't currently support these modules, but would like to:

```
import dbm
import dbm.dumb
import dbm.gnu
import collections.abc  # on Py33
import pickle      # should (optionally) bring in cPickle on Python 2
```

**class** future.standard_library.**RenameImport**(*old_to_new*)
A class for import hooks mapping Py3 module names etc. to the Py2 equivalents.

future.standard_library.**cache_py2_modules**()
Currently this function is unneeded, as we are not attempting to provide import hooks for modules with ambiguous names: email, urllib, pickle.

future.standard_library.**detect_hooks**()
Returns True if the import hooks are installed, False if not.

future.standard_library.**disable_hooks**()
Deprecated. Use remove_hooks() instead. This will be removed by future v1.0.

future.standard_library.**enable_hooks**()
Deprecated. Use install_hooks() instead. This will be removed by future v1.0.

**class** future.standard_library.**exclude_local_folder_imports**(**args*)
A context-manager that prevents standard library modules like configparser from being imported from the local python-future source folder on Py3.

(This was need prior to v0.16.0 because the presence of a configparser folder would otherwise have prevented setuptools from running on Py3. Maybe it's not needed any more?)

future.standard_library.**from_import**(*module_name*, **symbol_names*, ***kwargs*)

**Example use:**

```
>>> HTTPConnection = from_import('http.client',
↪'HTTPConnection')
>>> HTTPServer = from_import('http.server', 'HTTPServer')
>>> urlopen, urlparse = from_import('urllib.request',
↪'urlopen', 'urlparse')
```

Equivalent to this on Py3:

```
>>> from module_name import symbol_names[0], symbol_names[1], ..
 ↪.
```

and this on Py2:

```
>>> from future.moves.module_name import symbol_names[0], ...
```

or:

```
>>> from future.backports.module_name import symbol_names[0], ..
 ↪.
```

except that it also handles dotted module names such as `http.client`.

**class** future.standard_library.**hooks**

Acts as a context manager. Saves the state of sys.modules and restores it after the 'with' block.

Use like this:

```
>>> from future import standard_library
>>> with standard_library.hooks():
...     import http.client
>>> import requests
```

For this to work, http.client will be scrubbed from sys.modules after the 'with' block. That way the modules imported in the 'with' block will continue to be accessible in the current namespace but not from any imported modules (like requests).

future.standard_library.**import_**(*module_name*, *backport=False*)

Pass a (potentially dotted) module name of a Python 3 standard library module. This function imports the module compatibly on Py2 and Py3 and returns the top-level module.

**Example use:**

```
>>> http = import_('http.client')
>>> http = import_('http.server')
>>> urllib = import_('urllib.request')
```

**Then:**

```
>>> conn = http.client.HTTPConnection(...)
>>> response = urllib.request.urlopen('http://mywebsite.com
 ↪')
>>> # etc.
```

**Use as follows:**

```
>>> package_name = import_(module_name)
```

On Py3, equivalent to this:

```
>>> import module_name
```

On Py2, equivalent to this if backport=False:

```
>>> from future.moves import module_name
```

or to this if backport=True:

```
>>> from future.backports import module_name
```

except that it also handles dotted module names such as `http.client` The effect then is like this:

```
>>> from future.backports import module
>>> from future.backports.module import submodule
>>> module.submodule = submodule
```

Note that this would be a SyntaxError in Python:

```
>>> from future.backports import http.client
```

`future.standard_library.`**`install_aliases`**`()`
Monkey-patches the standard library in Py2.6/7 to provide aliases for better Py3 compatibility.

`future.standard_library.`**`install_hooks`**`()`
This function installs the future.standard_library import hook into sys.meta_path.

`future.standard_library.`**`is_py2_stdlib_module`**`(`*m*`)`
Tries to infer whether the module m is from the Python 2 standard library. This may not be reliable on all systems.

`future.standard_library.`**`remove_hooks`**`(`*scrub_sys_modules=False*`)`
This function removes the import hook from sys.meta_path.

`future.standard_library.`**`restore_sys_modules`**`(`*scrubbed*`)`
Add any previously scrubbed modules back to the sys.modules cache, but only if it's safe to do so.

`future.standard_library.`**`scrub_future_sys_modules`**`()`
Deprecated.

`future.standard_library.`**`scrub_py2_sys_modules`**`()`
Removes any Python 2 standard library modules from `sys.modules` that would interfere with Py3-style imports using import hooks. Examples are modules with the same names (like urllib or email).

(Note that currently import hooks are disabled for modules like these with ambiguous names anyway ...)

**`class`** `future.standard_library.`**`suspend_hooks`**
Acts as a context manager. Use like this:

```
>>> from future import standard_library
>>> standard_library.install_hooks()
>>> import http.client
>>> # ...
>>> with standard_library.suspend_hooks():
>>>     import requests     # incompatible with ``future``'s␣
↪standard library hooks
```

If the hooks were disabled before the context, they are not installed when the context is left.

### 1.13.4 future.utils Interface

A selection of cross-compatible functions for Python 2 and 3.

This module exports useful functions for 2/3 compatible code:

- bind_method: binds functions to classes
- `native_str_to_bytes` and `bytes_to_native_str`
- `native_str`: always equal to the native platform string object (because this may be shadowed by imports from future.builtins)
- lists: lrange(), lmap(), lzip(), lfilter()
- **iterable method compatibility:**
  - iteritems, iterkeys, itervalues
  - viewitems, viewkeys, viewvalues

  These use the original method if available, otherwise they use items, keys, values.
- types:
  - text_type: unicode in Python 2, str in Python 3
  - string_types: basestring in Python 2, str in Python 3
  - binary_type: str in Python 2, bytes in Python 3
  - integer_types: (int, long) in Python 2, int in Python 3
  - class_types: (type, types.ClassType) in Python 2, type in Python 3
- **bchr(c):** Take an integer and make a 1-character byte string
- **bord(c)** Take the result of indexing on a byte string and make an integer
- **tobytes(s)** Take a text string, a byte string, or a sequence of characters taken from a byte string, and make a byte string.
- raise_from()
- raise_with_traceback()

This module also defines these decorators:

- `python_2_unicode_compatible`
- `with_metaclass`
- `implements_iterator`

Some of the functions in this module come from the following sources:

- Jinja2 (BSD licensed: see https://github.com/mitsuhiko/jinja2/blob/master/LICENSE)
- Pandas compatibility module pandas.compat
- six.py by Benjamin Peterson
- Django

`future.utils.`**`as_native_str`**(*encoding='utf-8'*)

A decorator to turn a function or method call that returns text, i.e. unicode, into one that returns a native platform str.

Use it as a decorator like this:

```
from __future__ import unicode_literals

class MyClass(object):
    @as_native_str(encoding='ascii')
    def __repr__(self):
        return next(self._iter).upper()
```

`future.utils.`**`binary_type`**

alias of `builtins.bytes`

`future.utils.`**`bind_method`**(*cls*, *name*, *func*)

Bind a method to class, python 2 and python 3 compatible.

**cls** [type] class to receive bound method

**name** [basestring] name of method on class instance

**func** [function] function to be bound as method

None

`future.utils.`**`exec_`**(*source*, *globals=None*, *locals=None*, */*)

Execute the given source in the context of globals and locals.

The source may be a string representing one or more Python statements or a code object as returned by compile(). The globals must be a dictionary and locals can be any mapping, defaulting to the current globals and locals. If only globals is given, locals defaults to it.

`future.utils.`**`implements_iterator`**(*cls*)

From jinja2/_compat.py. License: BSD.

Use as a decorator like this:

```
@implements_iterator
class UppercasingIterator(object):
    def __init__(self, iterable):
        self._iter = iter(iterable)
    def __iter__(self):
        return self
    def __next__(self):
        return next(self._iter).upper()
```

future.utils.**is_new_style**(*cls*)

Python 2.7 has both new-style and old-style classes. Old-style classes can be pesky in some circumstances, such as when using inheritance. Use this function to test for whether a class is new-style. (Python 3 only has new-style classes.)

future.utils.**isbytes**(*obj*)

**Deprecated. Use::**

```
>>> isinstance(obj, bytes)
```

**after this import:**

```
>>> from future.builtins import bytes
```

future.utils.**isidentifier**(*s*, *dotted=False*)

A function equivalent to the str.isidentifier method on Py3

future.utils.**isint**(*obj*)

Deprecated. Tests whether an object is a Py3 `int` or either a Py2 `int` or `long`.

Instead of using this function, you can use:

```
>>> from future.builtins import int
>>> isinstance(obj, int)
```

The following idiom is equivalent:

```
>>> from numbers import Integral
>>> isinstance(obj, Integral)
```

future.utils.**isnewbytes**(*obj*)

Equivalent to the result of `type(obj) == type(newbytes)` in other words, it is REALLY a newbytes instance, not a Py2 native str object?

Note that this does not cover subclasses of newbytes, and it is not equivalent to ininstance(obj, newbytes)

future.utils.**istext**(*obj*)

**Deprecated. Use::**

```
>>> isinstance(obj, str)
```

**after this import:**

```
>>> from future.builtins import str
```

future.utils.**iteritems**(*obj*, ***kwargs*)
Use this only if compatibility with Python versions before 2.7 is required. Otherwise, prefer viewitems().

future.utils.**iterkeys**(*obj*, ***kwargs*)
Use this only if compatibility with Python versions before 2.7 is required. Otherwise, prefer viewkeys().

future.utils.**itervalues**(*obj*, ***kwargs*)
Use this only if compatibility with Python versions before 2.7 is required. Otherwise, prefer viewvalues().

future.utils.**native**(*obj*)
On Py3, this is a no-op: native(obj) -> obj

On Py2, returns the corresponding native Py2 types that are superclasses for backported objects from Py3:

```
>>> from builtins import str, bytes, int
```

```
>>> native(str(u'ABC'))
u'ABC'
>>> type(native(str(u'ABC')))
unicode
```

```
>>> native(bytes(b'ABC'))
b'ABC'
>>> type(native(bytes(b'ABC')))
bytes
```

```
>>> native(int(10**20))
100000000000000000000L
>>> type(native(int(10**20)))
long
```

Existing native types on Py2 will be returned unchanged:

```
>>> type(native(u'ABC'))
unicode
```

future.utils.**native_bytes**
alias of builtins.bytes

future.utils.**native_str**
alias of builtins.str

future.utils.**native_str_to_bytes**(*s*, *encoding='utf-8'*)
On Py3, returns an encoded string. On Py2, returns a newbytes type, ignoring the

encoding argument.

future.utils.**old_div**(*a*, *b*)
DEPRECATED: import old_div from past.utils instead.

Equivalent to a / b on Python 2 without from __future__ import division.

TODO: generalize this to other objects (like arrays etc.)

future.utils.**python_2_unicode_compatible**(*cls*)
A decorator that defines __unicode__ and __str__ methods under Python 2. Under Python 3, this decorator is a no-op.

To support Python 2 and 3 with a single code base, define a __str__ method returning unicode text and apply this decorator to the class, like this:

```
>>> from future.utils import python_2_unicode_compatible
```

```
>>> @python_2_unicode_compatible
... class MyClass(object):
...     def __str__(self):
...         return u'Unicode string: '
```

```
>>> a = MyClass()
```

Then, after this import:

```
>>> from future.builtins import str
```

the following is True on both Python 3 and 2:

```
>>> str(a) == a.encode('utf-8').decode('utf-8')
```

True

and, on a Unicode-enabled terminal with the right fonts, these both print the Chinese characters for Confucius:

```
>>> print(a)
>>> print(str(a))
```

The implementation comes from django.utils.encoding.

future.utils.**raise_**(*tp*, *value=None*, *tb=None*)
A function that matches the Python 2.x raise statement. This allows re-raising exceptions with the cls value and traceback on Python 2 and 3.

future.utils.**raise_with_traceback**(*exc*, *traceback=Ellipsis*)
Raise exception with existing traceback. If traceback is not passed, uses sys.exc_info() to get traceback.

future.utils.**reraise**(*tp*, *value=None*, *tb=None*)
A function that matches the Python 2.x `raise` statement. This allows re-raising exceptions with the cls value and traceback on Python 2 and 3.

future.utils.**text_type**
alias of `builtins.str`

future.utils.**tobytes**(*s*)
Encodes to latin-1 (where the first 256 chars are the same as ASCII.)

future.utils.**viewitems**(*obj*, ***kwargs*)
Function for iterating over dictionary items with the same set-like behaviour on Py2.7 as on Py3.

Passes kwargs to method.

future.utils.**viewkeys**(*obj*, ***kwargs*)
Function for iterating over dictionary keys with the same set-like behaviour on Py2.7 as on Py3.

Passes kwargs to method.

future.utils.**viewvalues**(*obj*, ***kwargs*)
Function for iterating over dictionary values with the same set-like behaviour on Py2.7 as on Py3.

Passes kwargs to method.

future.utils.**with_metaclass**(*meta*, **bases*)
Function from jinja2/_compat.py. License: BSD.

Use it like this:

```
class BaseForm(object):
    pass

class FormType(type):
    pass

class Form(with_metaclass(FormType, BaseForm)):
    pass
```

This requires a bit of explanation: the basic idea is to make a dummy metaclass for one level of class instantiation that replaces itself with the actual metaclass. Because of internal type checks we also need to make sure that we downgrade the custom metaclass for one level to something closer to type (that's why __call__ and __init__ comes back from type etc.).

This has the advantage over six.with_metaclass of not introducing dummy classes into the final MRO.

### 1.13.5 past.builtins Interface

A resurrection of some old functions from Python 2 for use in Python 3. These should be used sparingly, to help with porting efforts, since code using them is no longer standard Python 3 code.

This module provides the following:

1. Implementations of these builtin functions which have no equivalent on Py3:
   - apply
   - chr
   - cmp
   - execfile
2. Aliases:
   - intern <- sys.intern
   - raw_input <- input
   - reduce <- functools.reduce
   - reload <- imp.reload
   - unichr <- chr
   - unicode <- str
   - xrange <- range
3. List-producing versions of the corresponding Python 3 iterator-producing functions:
   - filter
   - map
   - range
   - zip
4. Forward-ported Py2 types:
   - basestring
   - dict
   - str
   - long
   - unicode

**class** past.builtins.**basestring**
A minimal backport of the Python 2 basestring type to Py3

past.builtins.**chr**(*i*)
Return a byte-string of one character with ordinal i; 0 <= i <= 256

past.builtins.**cmp**(*x*, *y*) → integer
Return negative if x<y, zero if x==y, positive if x>y.

past.builtins.**dict**
alias of past.types.olddict.olddict

past.builtins.**execfile**(*filename*, *myglobals=None*, *mylocals=None*)
Read and execute a Python script from a file in the given namespaces. The globals and locals are dictionaries, defaulting to the current globals and locals. If only globals is given, locals defaults to it.

past.builtins.**filter**(*function or None*, *sequence*) → list, tuple, or string
Return those items of sequence for which function(item) is true. If function is None, return the items that are true. If sequence is a tuple or string, return the same type, else return a list.

past.builtins.**intern**(*string*) → string
``Intern'' the given string. This enters the string in the (global) table of interned strings whose purpose is to speed up dictionary lookups. Return the string itself or the previously interned string object with the same value.

past.builtins.**long**
alias of builtins.int

past.builtins.**map**(*function*, *sequence*[, *sequence*, ...]) → list
Return a list of the results of applying the function to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence, substituting None for missing values when not all sequences have the same length. If the function is None, return a list of the items of the sequence (or a list of tuples if more than one sequence).

Test cases: >>> oldmap(None, 'hello world') ['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']

```
>>> oldmap(None, range(4))
[0, 1, 2, 3]
```

More test cases are in test_past.test_builtins.

past.builtins.**raw_input**(*prompt=None*, */*)
Read a string from standard input. The trailing newline is stripped.

The prompt string, if given, is printed to standard output without a trailing newline before reading input.

If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise EOFError. On *nix systems, readline is used if available.

past.builtins.**reduce**(*function*, *sequence*[, *initial*]) → value
Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). If initial is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

past.builtins.**reload**(*module*)
**DEPRECATED**

Reload the module and return it.

The module must have been successfully imported before.

past.builtins.**str**
alias of past.types.oldstr.oldstr

past.builtins.**unichr**(*i*)
Return a byte-string of one character with ordinal i; 0 <= i <= 256

past.builtins.**unicode**
alias of builtins.str

past.builtins.**xrange**
alias of builtins.range

### 1.13.6 Forward-ported types from Python 2

Forward-ports of types from Python 2 for use with Python 3:

- basestring: equivalent to (str, bytes) in isinstance checks
- dict: with list-producing .keys() etc. methods
- str: bytes-like, but iterating over them doesn't product integers
- long: alias of Py3 int with L suffix in the repr
- unicode: alias of Py3 str with u prefix in the repr

**class** past.types.**basestring**
A minimal backport of the Python 2 basestring type to Py3

past.types.**long**
alias of builtins.int

**class** past.types.**olddict**
A backport of the Python 3 dict object to Py2

**has_key**(*k*) → True if D has a key k, else False

**items**() → a set-like object providing a view on D's items

**iteritems**()
D.items() -> a set-like object providing a view on D's items

**iterkeys**()
D.keys() -> a set-like object providing a view on D's keys

**itervalues**()
D.values() -> an object providing a view on D's values

**keys**() → a set-like object providing a view on D's keys

**values**() → an object providing a view on D's values

**viewitems**()
D.items() -> a set-like object providing a view on D's items

**viewkeys**()
D.keys() -> a set-like object providing a view on D's keys

**viewvalues**()
D.values() -> an object providing a view on D's values

**class** past.types.**oldstr**
A forward port of the Python 2 8-bit string object to Py3

past.types.**unicode**
alias of builtins.str

CHAPTER

TWO

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## f



## p

# INDEX