Cheat Sheet: Writing Python 2-3 compatible code

- **Author:** Ed Schofield.

A PDF version is here: http://python-future.org/compatible_idioms.pdf (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 (http://www.youtube.com/watch?v=KOqk8j11aAI&t=10m14s).)

Minimum versions:

- Python 2: 2.6+
- Python 3: 3.3+

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

Essential syntax differences

print

In [ ]:

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

In [ ]:

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

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

In [ ]:

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

In [ ]:

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

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

In [ ]:

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

In [ ]:

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

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

In [ ]:

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

In [ ]:

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

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

Raising exceptions

In [ ]:

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

In [ ]:

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

Raising exceptions with a traceback:

In [ ]:

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

In [ ]:

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

In [ ]:

```
# 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)
```

In [ ]:

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

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

Exception chaining (PEP 3134):

In [3]:

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

In [ ]:

```
# 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
```

In [16]:

```
# 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)
```

In [17]:

```
# Testing the above:
try:
    fd = FileDatabase('non_existent_file.txt')
except Exception as e:
    assert isinstance(e.__cause__, IOError)    # FileNotFoundError on Py
3.3+ inherits from IOError
```

Catching exceptions

In [ ]:

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

In [ ]:

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

Division

Integer division (rounding down):

In [ ]:

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

In [ ]:

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

"True division" (float division):

In [ ]:

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

In [ ]:

```
# 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):

In [ ]:

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

In [ ]:

```
# 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`).

In [ ]:

```
# Python 2 only
k = 9223372036854775808L

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

In [ ]:

```
# 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):

In [ ]:

```
# 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

In [ ]:

```
0644     # Python 2 only
```

In [ ]:

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

Backtick repr

In [ ]:

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

In [ ]:

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

Metaclasses

In [ ]:

```
class BaseForm(object):
    pass

class FormType(type):
    pass
```

In [ ]:

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

In [ ]:

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

In [ ]:

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

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

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:

In [ ]:

```
# Python 2 only
s1 = 'The Zen of Python'
s2 = u'きたないのよりきれいな方がいい\n'

# Python 2 and 3
s1 = u'The Zen of Python'

s2 = u'きたないのよりきれいな方がいい\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:

In [ ]:

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

s1 = 'The Zen of Python'
s2 = 'きたないのよりきれいな方がいい\n'
```

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

Byte-string literals

In [ ]:

```
# 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:

In [ ]:

```
# 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:
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:

In [ ]:

```
# 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

In [ ]:

```
# 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))
```

In [ ]:

```
# 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

In [ ]:

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

In [ ]:

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

In [ ]:

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

StringIO

In [ ]:

```
# 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
```

Imports relative to a package

Suppose the package is:

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

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

In [ ]:

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

In [ ]:

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

In [ ]:

```
# 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
```

Dictionaries

In [ ]:

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

Iterating through `dict` keys/values/items

Iterable dict keys:

In [ ]:

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

In [ ]:

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

Iterable dict values:

In [ ]:

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

In [ ]:

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

In [8]:

```
# 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
    ...
```

In [ ]:

```
# Python 2 and 3: option 2
from builtins import itervalues
# or
from six import itervalues

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

Iterable dict items:

In [ ]:

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

In [ ]:

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

In [ ]:

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

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

In [ ]:

```
# 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:

In [ ]:

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

In [ ]:

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

dict values as a list:

In [ ]:

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

In [ ]:

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

In [ ]:

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

heights = dict(Fred=175, Anne=166, Joe=192)

valuelist = list(heights.values())
```

In [ ]:

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

valuelist = listvalues(heights)
```

In [ ]:

```
# 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:

In [ ]:

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

In [ ]:

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

itemlist = listitems(heights)
```

In [ ]:

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

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

Custom class behaviour

Custom iterators

In [ ]:

```
# 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')
```

In [ ]:

```
# 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')
```

In [ ]:

```
# 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

In [ ]:

```
# 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
```

In [11]:

```
# Python 2 and 3:
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()
print(a)    # prints string encoded as utf-8 on Py2
```

```
Unicode string: 孔子
```

Custom __nonzero__ vs __bool__ method:

In [ ]:

```
# 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)
```

In [ ]:

```
# 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)
```

Lists versus iterators

xrange

In [ ]:

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

In [ ]:

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

In [ ]:

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

range

In [ ]:

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

In [ ]:

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

In [ ]:

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

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

In [ ]:

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

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

In [ ]:

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

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

map

In [ ]:

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

In [ ]:

```
# 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]
```

In [ ]:

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

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

In [ ]:

```
# 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]
```

In [ ]:

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

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

In [ ]:

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

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

imap

In [ ]:

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

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

In [ ]:

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

In [ ]:

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

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

In [ ]:

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

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.

Other builtins

File IO with open()

In [ ]:

```
# 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
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()

In [ ]:

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

In [ ]:

```
# 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()

In [ ]:

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

In [ ]:

```
# 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()

In [ ]:

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

In [ ]:

```
# 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()

In [ ]:

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

In [ ]:

```
# 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
```

execfile()

In [ ]:

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

In [ ]:

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

execfile('myfile.py')
```

In [ ]:

```
# 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()

In [ ]:

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

In [ ]:

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

In [ ]:

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

intern()

In [ ]:

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

In [ ]:

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

In [ ]:

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

In [ ]:

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

In [ ]:

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

In [ ]:

```
# Python 2 and 3: alternative 2
try:
    from sys import intern
except ImportError:
    pass
intern('mystring')
```

apply()

In [ ]:

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

In [ ]:

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

In [ ]:

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

In [ ]:

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

chr()

In [ ]:

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

In [ ]:

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

In [ ]:

```
# 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'
```

In [ ]:

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

In [ ]:

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

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

cmp()

In [ ]:

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

In [ ]:

```
# 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
```

In [ ]:

```
# 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()

In [ ]:

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

In [ ]:

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

Standard library

dbm modules

In [ ]:

```
# 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()

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

In [ ]:

```
# 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
```

subprocess.check_output()

In [ ]:

```
# Python 2.7 and above
from subprocess import check_output

# Python 2.6 and above: alternative 1
from future.moves.subprocess import check_output

# Python 2.6 and above: alternative 2
from future import standard_library
standard_library.install_aliases()


from subprocess import check_output
```

collections: Counter, OrderedDict, ChainMap

In [6]:

```
# Python 2.7 and above
from collections import Counter, OrderedDict, ChainMap

# Python 2.6 and above: alternative 1
from future.backports import Counter, OrderedDict, ChainMap

# Python 2.6 and above: alternative 2
from future import standard_library
standard_library.install_aliases()

from collections import Counter, OrderedDict, ChainMap
```

StringIO module

In [ ]:

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

In [ ]:

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

http module

In [ ]:

```
# 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

In [ ]:

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

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

In [ ]:

```
# Python 2 only:
import xmlrpclib

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

html escaping and entities

In [ ]:

```
# 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

In [ ]:

```
# 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 may like to use Requests (http://python-requests.org (http://python-requests.org)) instead.

In [ ]:

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

In [2]:

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

In [ ]:

```
# 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
```

In [ ]:

```
# 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
```

In [ ]:

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

In [ ]:

```
# 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

In [ ]:

```
# 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

# Python 2 and 3 (after ``pip install future``):
import tkinter
import tkinter.dialog
import tkinter.filedialog
import tkinter.scolledtext
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
import tkinter.simpledialog
import tkinter.ttk
```

socketserver

In [ ]:

```
# Python 2 only:
import SocketServer

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

copy_reg, copyreg

In [ ]:

```
# Python 2 only:
import copy_reg

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

configparser

In [ ]:

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

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

queue

In [ ]:

```
# 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

In [ ]:

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

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

UserDict, UserList, UserString

In [ ]:

```
# 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

In [ ]:

```
# 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
```