Conservative Python 3 Porting Guide Documentation

Release 1.0

portingguide contributors

Sep 09, 2021

Contents

This document will guide you through porting your software to Python 3. It is geared towards projects that are being ported because support for Python 2 is ending in a few years, and less for those that are porting because Python 3 as a language allows writing expressive, maintainable and correct code more easily. It mainly targets projects with large, conservative codebases.

We assume the *maintainers* of the codebase will only grudgingly accept porting-related changes, not necessarily that *you* specifically have an aversion to Python 3. If *you* are not convinced that Python 3 is a good choice, please read the foreword of Lennart Regebro's book, and skim Nick Coghlan's Python 3 Q & A, which discusses the issues (both with Python 2 and 3) in depth.

This guide does *not* cover Python3-only features. If you're interested in updating your code to take advantage of current best practices, rather than doing the minimum amount of work necessary to keep your software working on modern versions of Python, a better resource for you would be Lennart Regebro's book, Supporting Python 3 (known as "Porting to Python 3" in earlier editions).

This is an *opinionated* guide. It explains one tried way to do the porting, rather than listing all alternatives and leaving you to research them and choose.

Still with us? Let's dive in!

CHAPTER 1

The Porting Process

This chapter documents the entire process of porting a conservative project to Python 3. We recommend that you read it before you embark on your first porting project.

1.1 Make Sure your Dependencies are Ported

Before you start porting, any libraries that your code imports need to run with Python 3. Check if this is the case.

Porting status information of a library is usually available in `setup.py` or `setup.cfg` files, where compatible Python versions can be mentioned in `classifiers` argument to `setup()`.

Some projects took advantage of the backwards incompatible upgrade to clean up their interfaces. For any libraries that are ported already, look in their documentation for any notes specific to Python 3, and if you find any, note them for later.

If you depend on a library that is not ported, inquire of its authors about the porting status. If the library is open-source, consider helping to port it – the experience will likely help in your own project. If authors are unwilling to port to Python 3, or if the library is unmaintained, start looking for a replacement. For projects in Fedora, the portingdb project lists known alternatives for dropped packages.

1.2 Run the Tests

It's impractical to make any changes to untested code, let alone porting the entire codebase to a new version of the programming language.

If the project has automatic tests, run them under Python 2 to make sure they pass. If not, write them – or you'll need to resort to testing manually.

1.3 Drop Python 2.5 and Lower

Python 2.6 and 2.7 were released in lockstep with the early 3.x versions, and contain several features that make supporting both 2 and 3 possible in the same codebase.

Python 2.5 has been unmaintained for several years now, so any *new* code written for it does not have much of a future. Bring this up with the software's maintainers.

If compatibility with Python 2.5 is *really* necessary, we recommend that you fork the codebase, i.e. work on a copy and regularly merge in any new development.

1.4 Port the Code

Actual porting can be conceptually split into two phases:

Modernization Migrate away from deprecated features that have a Python3-compatible equivalent available in Python 2.

Porting Add support for Python 3 while keeping compatibility with Python 2 by introducing specific workarounds and helpers.

We don't recommend separating these phases. For larger projects, it is much better to separate the work by modules – port low-level code first, then move on to things that depend on what's already ported.

We provide some general porting tips below:

1.4.1 Use The Tools

The *Tools* chapter describes a selection of tools that can automate or ease the porting process, and warn about potential problems or common regressions. We recommend that you get familiar with these tools before porting any substantial project.

In particular, this guide includes "fixers" where appropriate. These can automate a lot, if not most, of the porting work. But please read the *notes for the python-modernize tool* before running them to avoid any surprises.

1.4.2 Port Small Pieces First

If the codebase contains a small, self-contained module, port it first before moving on to larger pieces or the entire code.

If you want to learn porting in a more practical way before you port your own software, you can help developers with porting some open source software or your favorite library or application.

1.4.3 Use Separate Commits for Automated Changes

For changes that are mechanical, and easily automated, we recommend that you do only one type of change per commit/patch. For example, one patch to change the *except syntax*, then another for the *raise syntax*.

Even more importantly, do not combine large automated changes with manual fixups. It is much easier to review two patches: one done by a tool (which the reviewer can potentially re-run to verify the commit), and another that fixes up places where human care is needed.

The descriptions of individual items in this guide are written so that you can use them in commit messages to explain why each change is necessary and to link to more information.

1.4.4 Follow the Rest of this Guide

The next chapter, *Tools*, explains how to automate porting and checking.

Each of the subsequent chapters explains one area where Python 3 differs from Python 2, and how to adapt the code. The chapters are arranged roughly according to the order in which they are tackled in a typical project.

We recommend that you skim the introduction of each of the chapters, so that you know what you're up against before you start.

Note that while the guide is fairly comprehensive, there are changes it does not cover. Be prepared to find a few issues specific to your code base that you'll need to figure out independently.

Also note that the guide was written for Python 3.6. It includes several updates for newer versions, but we recommend skimming [What's New lists](https://docs.python.org/3/whatsnew/index.html) in the Python documentation to familiarize yourself with the changes in newer versions of Python.

1.5 Drop Python 2

The final step of the porting is dropping support for Python 2, which can happen after a long time – even several years from releasing a Python 3-compatible version. For less conservative projects, dropping Python 2 support will include removing compatibility workarounds.

Targeting Python 3 only will enable you to start using all the new features in the new major version – but those are for another guide.

CHAPTER 2

Tools

Several tools exist to automate as much of the porting as possible, and to check for common errors. Here is a survey of tools we recommend.

2.1 Compatibility library: `six`

When porting a large piece of software, it is desirable to support both Python 2 and Python 3 in the same codebase. Many projects will need this dual support for a long time, but even those that can drop Python 2 support as soon as the port is done, will typically go through a period of adding Python 3 support, in which the software should continue to work on Python 2.

Benjamin Peterson's `six` module makes it practical to write such version-straddling code by offering compatibility wrappers over the differences.

For example, the Python 3 syntax for specifying metaclasses is not valid Python 2, and the Python 2 way does nothing in Python 3, so `six` provides an `add_metaclass` decorator for this purpose. It also provides stable names for standard library modules that were moved or reorganized in Python 3.

Six is a run-time dependency, albeit a very small one. If your project is unfriendly to third-party dependencies, push for this one as hard as possible. If you do not use `six`, you will most likely end up reimplementing it or outright copying relevant pieces of it into your code.

2.2 Automated fixer: `python-modernize`

Some steps of the porting process are quite mechanical, and can be automated. These are best handled by the `python-modernize` tool – a code-to-code translator that takes a Python 2 codebase and updates it to be compatible with both Python 2 and 3.

Note: `python-modernize` was built on top of `2to3` from of Python's standard library. `2to3` was once intended as the main porting tool. It turned out inadequate for that task, but `python-modernize` (among others) successfully reuses its general infrastructure. Because `2to3` itself is built into Python and thus missing improvements newer than the the Python that runs it, `python-modernize` now uses a fork of `2to3` called `fissix`.

Assuming code is in version control, you'll generally want to run `python-modernize` with the `-wn` flags: `-w` flag causes the tool to actually change the affected files, and `-n` suppresses creating backups.

The tool operates by applying individual *fixers* – one for each type of change needed. You can select individual fixers to run using the `-f` option. We've found that running a single fixer at a time results in changes that are easier to review and more likely to be accepted, so that is what this guide will recommend. The order of fixers matters sometimes. This guide will present them in order, but if you skip around, you will need to pay a bit more attention.

The tool always needs a directory (or individual files) to operate on; usually you'll use the current directory (`.`).

Combining all that, the recommended invocation is:

```
python-modernize -wnf <fixer-name> .
```

While `python-modernize` is useful, it is not perfect. Some changes it makes might not make sense at all times, and in many cases. It is necessary to know *what* and *why* is changed, and to review the result as closely as if a human wrote it. This guide will provide the necessary background for each fixer as we go along.

2.3 Compatibility headers and guide for C extensions: `py3c`

Some projects involve extension modules written in C/C++, or embed Python in a C/C++-based application. An easy way to find these is to search your codebase for `PyObject`. For these, we have two pieces of advice:

- Even though this is a conservative guide, we encourage you to try porting C extensions away from the Python C API. For wrappers to external libraries we recommend CFFI; for code that needs to be fast there's Cython.

  While this is relatively disruptive, the result will very likely be more maintainable and less buggy, as well as more portable to alternative Python implementations.

- If you decide to keep your C extension, follow a dedicated porting guide similar to this one, which also comes with a `six`-like library for C extensions: py3c.

2.4 Automated checker: `pylint --py3k`

Pylint is a static code analyzer that can catch mistakes such as initialized variables, unused imports, and duplicated code. It also has a mode that flags code incompatible with Python 3.

If you are already using Pylint, you can run the tool with the `--py3k` option on any code that is already ported. This will prevent most regressions.

You can also run `pylint --py3k` on unported code to get an idea of what will need to change, though `python-modernize` is usually a better choice here.

CHAPTER 3

Syntax Changes

Python 3 cleaned up some warts of the language's syntax.

The changes needed to accommodate this are mostly mechanical, with little chance of breaking code, so they work well as the first patches to send to a project when intending to port it.

3.1 Tabs and Spaces

- *Fixer*: see below
- Prevalence: Very common (unless the code uses a style linter)

In Python 2, a tab character in indentation was considered equal to 8 spaces or less. In Python 3, a tab is only equal to another tab, so the following code is rejected (whitespace highlighted):

```
def f(cond):
····if cond:
→       do_something()
····else:
→       do_something_else()
```

If your code mixes tabs and spaces, the easiest way to fix this is converting all tabs to spaces. You can use the following Bash command for this:

```
find . -name '*.py' -type f -exec bash -c 'T=$(mktemp); expand -i -t 8 "$0" > "$T" &&␣
↪mv "$T" "$0"' {} \;
```

3.2 Tuple Unpacking in Parameter Lists

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_tuple_params` (fixup needed)
- Prevalence: Common

Python 3 requires that each argument of a `def` function has a name. This simplifies code that uses introspection (such as help systems, documentation generation, and automatic dispatchers), but it does have a drawback: tuples are no longer allowed in formal parameter lists.

For example, functions like these are no longer allowed in Python 3:

```
def line((x1, y1), (x2, y2)):
    connect_points(Point(x1, y1), Point(x2, y2))

lambda (key, item): (item, key)
```

The recommended fixer does a good job in finding places that need fixing, but it does need some manual cleanup. The above example would be rewritten to:

```
def line(xxx_todo_changeme, xxx_todo_changeme1):
    (x1, y1) = xxx_todo_changeme
    (x2, y2) = xxx_todo_changeme1
    connect_points(Point(x1, y1), Point(x2, y2))

lambda key_item: (key_item[1], key_item[0])
```

For `def`, each of the newly introduced variables should be renamed to something more appropriate.

As for `lambda`, this transformation can leave the code less readable than before. For each such `lambda`, you should consider if replacing it with a regular named function would be an improvement.

3.3 Backticks

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_repr` (with caveat)
- Prevalence: Common

The backtick (`) operator was removed in Python 3. It is confusingly similar to a single quote, and hard to type on some keyboards. Instead of the backtick, use the equivalent built-in function `repr()`.

The recommended fixer does a good job, though it doesn't catch the case where the name `repr` is redefined, as in:

```
repr = None
print(`1+2`)
```

which becomes:

```
repr = None
print(repr(1+2))
```

Re-defining built-in functions is usually considered bad style, but it never hurts to check if the code does it.

3.4 The Inequality Operator

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_ne`
- Prevalence: Rare

In the spirit of "There's only one way to do it", Python 3 removes the little-known alternate spelling for inequality: the <> operator.

The recommended fixer will replace all occurrences with `!=`.

3.5 New Reserved Words

- *Fixer*: None
- Prevalence: Rare

3.5.1 Constants

In Python 3, `None`, `True` and `False` are syntactically keywords, not variable names, and cannot be assigned to. This was partially the case with `None` even in Python 2.6.

Hopefully, production code does not assign to `True` or `False`. If yours does, figure a way to do it differently.

3.5.2 `async` and `await`

Since Python 3.7, `async` and `await` are also keywords.

If your code uses these names, rename it. If other code depends on the names, keep the old name available for old Python versions. The way to do this will be different in each case, but generally you'll need to take advantage of the fact that in Python's various namespaces the strings `'async'` and `'await'` are still valid keys, even if they are not accesible usual with the syntax.

For module-level functions, classes and constants, also assign the original name using `globals()`. For example, a function previously named `async` could look like this:

```
def asynchronous():
    """...

    This function used to be called `async`.
    It is still available under old name.
    """

globals()['async'] = asynchronous
```

For methods, and class-level constants, assign the original name using `setattr`:

```
class MyClass:
    def asynchronous(self):
        """...

        This method used to be called `async`.
        It is still available under old name.
        """

setattr(MyClass, 'async', MyClass.asynchronous)
```

For function parameters, more work is required. The result will depend on whether the argument is optional and whether `None` is a valid value for it. Here is a general starting point:

```
def process_something(asynchronous=None, **kwargs):
    if asynchronous is None:
        asynchronous = kwargs.get('async', None)
    else:
        if 'async' in kwargs:
            raise TypeError('Both `asynchronous` and `async` specified')
```

(continues on next page)

(continued from previous page)

```
    if asynchronous is None:
        raise TypeError('The argument `asynchronous` is required')
```

For function arguments, if the parameter cannot be renamed as above, use “double star” syntax that allows you to pass arbitrary argument names:

```
process_something(**{'async': True})
```

3.6 Other Syntax Changes

For convenience and completeness, this section lists syntax changes covered in other chapters:

- *The print() function*
- *The new except syntax*
- *The new raise syntax*
- *import * in Functions*
- *The L suffix not allowed in numeric literals*
- *Octal Literals*
- *The exec() function*

CHAPTER 4

Exceptions

Very early Python versions used simple strings to signalize errors. Later, Python allowed raising arbitrary classes, and added specialized exception classes to the standard library. For backwards compatibility reasons, some deprecated practices were still allowed in Python 2. This presents confusion to learners of the language, and prevents some performance optimizations.

Python 3 removes the deprecated practices. It also further consolidates the exception model. Exceptions are now instances of dedicated classes, and contain all information about the error: the type, value and traceback.

This chapter mentions all exception-related changes needed to start supporting Python 3.

4.1 The new `except` syntax

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_except`
- Prevalence: Very common

In Python 2, the syntax for catching exceptions was `except ExceptionType:`, or `except ExceptionType, target:` when the exception object is desired. `ExceptionType` can be a tuple, as in, for example, `except (TypeError, ValueError):`.

This could result in hard-to-spot bugs: the command `except TypeError, ValueError:` (note lack of parentheses) will only handle `TypeError`. It will also assign the exception object to the name `ValueError`, shadowing the built-in.

To fix this, Python 2.6 introduced an alternate syntax: `except ExceptionType as target:`. In Python 3, the old syntax is no longer allowed.

You will need to switch to the new syntax. The recommended fixer works quite reliably, and it also fixes the *Iterating Exceptions* problem described below.

4.2 The new `raise` syntax

- *Fixer*: `python-modernize -wnf libmodernize.fixes.fix_raise -f libmodernize.`

  fixes.fix_raise_six

- Prevalence: Common

Python 2's raise statement was designed at a time when exceptions weren't classes, and an exception's *type*, *value*, and *traceback* components were three separate objects:

```
raise ValueError, 'invalid input'
raise ValueError, 'invalid input', some_traceback
```

In Python 3, one single object includes all information about an exception:

```
raise ValueError('invalid input')

e = ValueError('invalid input')
e.__traceback__ = some_traceback
raise e
```

Python 2.6 allows the first variant. For the second, re-raising an exception, the *Compatibility library: six* library includes a convenience wrapper that works in both versions:

```
import six
six.reraise(ValueError, 'invalid input',  some_traceback)
```

The recommended fixers will do these conversions automatically and quite reliably, but do verify the resulting changes.

4.3 Caught Exception "Scope"

- *Fixer*: None
- Prevalence: Rare

As *discussed previously*, in Python 3, all information about an exception, including the traceback, is contained in the exception object. Since the traceback holds references to the values of all local variables, storing an exception in a local variable usually forms a reference cycle, keeping all local variables allocated until the next garbage collection pass.

To prevent this issue, to quote from Python's documentation:

> When an exception has been assigned using as target, it is cleared at the end of the except clause. This is as if
>
> ```
> except E as N:
>     foo
> ```
>
> was translated to
>
> ```
> except E as N:
>     try:
>         foo
>     finally:
>         del N
> ```
>
> This means the exception must be assigned to a different name to be able to refer to it after the except clause.

Unfortunately, *Automated fixer: python-modernize* does not provide a fixer for this change. This issue results in a loud NameError when tests are run. When you see this error, apply the recommended fix – assign a different name to the exception to use it outside the except clause.

4.4 Iterating Exceptions

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_except` (but see caveat below)
- Prevalence: Rare

In Python 2, exceptions were *iterable*, so it was possible to "unpack" the arguments of an exception as part of the `except` statement:

```
except RuntimeError as (num, message):
```

In Python 3, this is no longer true, and the arguments must be accessed through the `args` attribute:

```
except RuntimeError as e:
    num, message = e.args
```

The recommended fixer catches the easy cases of unpacking in `except` statements. If your code iterates through exceptions elsewhere, you need to manually change it to iterate over `args` instead.

Additionally, the fixer does not do a good job on single-line suites such as:

```
except RuntimeError as (num, message): pass
```

Inspect the output and break these into multiple lines manually.

4.5 Raising Non-Exceptions

- *Fixer*: None
- Prevalence: Rare

In Python 3, an object used with `raise` must be an instance of `BaseException`, while Python 2 also allowed old-style classes. Similarly, Python 3 bans catching non-exception classes in the `except` statement.

Raising non-Exception classes was obsolete as early as in Python 2.0, but code that does this can still be found.

Each case needs to be handled manually. If there is a dedicated class for the exception, make it inherit from `Exception`. Otherwise, switch to using a dedicated Exception class.

4.6 The Removed `StandardError`

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_standarderror` (but see caveat below)
- Prevalence: Rare

The `StandardError` class is removed in Python 3. It was the base class for built-in exceptions, and it proved to be an unnecessary link in almost any exception's inheritance chain.

The recommended fixer will replace all uses of `StandardError` with `Exception`. Review the result to check if this is correct.

Some code might rely on the name of an exception class, or on exceptions not derived from `StandardError`, or otherwise handle `StandardError` specially. You'll need to handle these casses manually.

4.7 Removed `sys.exc_type`, `sys.exc_value`, `sys.exc_traceback`

- *Fixer*: None
- Prevalence: Rare

These exception-related attributes of the `sys` module are not thread-safe, and were deprecated since Python 1.5. They have been dropped for Python 3.

The information can be retrieved with a call to `exc_info()`:

```
exc_type, exc_value, exc_traceback = sys.exc_info()
```

CHAPTER 5

Importing

Python 3 brings a complete overhaul of the way `import` works – the import machinery was ported from C to Python.

Developer-visible changes are summarised below.

5.1 Absolute imports

- *Fixer*: `python-modernize -wnf libmodernize.fixes.fix_import` (See caveat below)
- Prevalence: Common
- Future import: `from __future__ import absolute_import`
- Specification: PEP 328

Under Python 2, when importing from inside a package, the package's own modules were considered before global ones. For example, given a package like this:

```
mypkg/
    __init__.py
    collections.py
    core.py
    ...
```

If `core.py` contains:

```
from collections import deque
```

it would import the `deque` from `mypkg/collections.py`. The standard library's `collections` module would be unavailable.

In Python 2.5, the situation began changing with the introduction of explicit relative imports, using a dot (`.`) before the submodule name. Given the structure above, these statements would be equivalent (in `core.py`):

```
from .collections import deque
from mypkg.collections import deque
```

Additionally, a *future import* was added to make all imports absolute (unless explicitly relative):

```
from __future__ import absolute_import
```

Using this feature, `from collections import deque` will import from the standard library's `collections` module.

In Python 3, the feature becomes the default.

To prepare for this, make sure all imports are either absolute, or *explicitly* relative. Both the `mypkg.collections` style and the `.collections` style are adequate; we recommend the former for increased readability[1].

The recommended fixer simply adds the future import to all files that do a potentially ambiguous import. This may be too much churn for your project; in most cases it is enough to verify that your imports are not ambiguous.

5.2 `import *` in Functions

- *Fixer*: None
- Prevalence: Rare

In Python 3, "star imports" are only allowed on the module level, not in classes or functions. For example, this won't work:

```
def coords(angle, distance):
    from math import *
    return distance * cos(angle), distance * sin(angle)
```

The reason for this limitation is that a function's local variables are optimized at compile time, but the names imported via `*` are not known in advance. Python 2 reverted to using unoptimized bytecode for such functions; Python 3 includes only the optimized case.

This code raised a `SyntaxWarning` already in Python 2.6. In Python 3, this becomes a `SyntaxError`, so module-level test coverage is enough to spot the error.

5.3 Import Cycles

- *Fixer*: None
- Prevalence: Rare

Python 3 introduced a reworked implementation of `import` in the form of the `importlib` module. The new machinery is backwards-compatible in practice, except that some import cycles, especially those involving submodules, now raise `ImportError`.

If you encounter such errors, check for import cycles (these should be visible from the traceback as one module imported multiple times). In most cases you can break circular imports by refactoring common code into a separate module.

[1] The downside of spelling out the package name is that it becomes harder to rename or reorganize the package. In practice, if you do rename a project, the work added by absolute imports tends to be insignificant compared to updating all external modules that import your package.

CHAPTER 6

Standard Library Reorganization

The standard library has been reorganized for Python 3.

6.1 Renamed Modules

- *Fixer*: `python-modernize -wnf libmodernize.fixes.fix_imports_six`
- Prevalence: Common

Many modules were simply renamed, usually to unify file naming conventions (e.g. `ConfigParser` to `configparser`) or to consolidate related modules in a namespace (e.g. `tkFont` to `tkinter.font`).

The *Compatibility library: six* library includes `six.moves`, a pseudo-package that exposes moved modules under names that work in both Python 2 and 3. For example, instead of:

```
from ConfigParser import ConfigParser
```

you should import from `six.moves`:

```
from six.moves.configparser import ConfigParser
```

A list of all renamed modules is included in six documentation.

The recommended fixer will automatically change imports to use `six.moves`.

6.2 Removed modules

- *Fixer*: None
- Prevalence: Uncommon

Some modules have been removed entirely. Usually, these modules were supplanted by better alternatives (e.g. `mimetools` by `email`), specific to now-unsupported operating systems (e.g. `fl`), or known to be broken (e.g. `Bastion`).

Lennart Regebro compiled a list of these modules in the book "Supporting Python 3", which is available online.

If your code uses any of the removed modules, check the *Python 2* documentation of the specific module for recommended replacements.

6.3 The `urllib` modules

- *Fixer*: None
- Prevalence: Common

The `urllib`, `urllib2` and `urlparse` modules were reorganized more heavily, with individual functions and classes redistributed to submodules of Python 3's `urllib`: `urllib.parse`, `urllib.error`, `urllib.request`, and `urllib.response`.

These functions are included in `six.moves`, and the six documentation has details on what moved where. Use this information to adjust your code.

The `fix_imports_six` fixer recommended above does not handle all urllib moves, so manual changes may be necessary.

6.4 The `string` module

- *Fixer*: None
- Prevalence: Rare

In Python 2, the `string` module included functions that mirrored `str` methods, such as `string.lower()` and `string.join()` that mirror `str.lower()` and `str.join()`. These have been deprecated since Python 2.4, and they are removed in Python 3.

Convert all uses of these functions to string methods.

For example, this code:

```
import string
products = ['widget', 'thingy', 'whatchamacallit']
print string.join(products, sep=', ')
```

should be replaced with:

```
products = ['widget', 'thingy', 'whatchamacallit']
print(', '.join(products))
```

The *Automated fixer: python-modernize* tool doesn't provide an automated fixer for these changes.

CHAPTER 7

Numbers

There have been two major changes in how Python 3 handles numbers: true division replaces truncating division, and the `long` type was merged into `int`.

This section describes these changes in detail, along with other, minor ones.

7.1 Division

- *Fixer*: None
- Future import: `from __future__ import division`
- Prevalence: Common

In Python 2, dividing two integers resulted in an integer:

```
>>> print 2 / 5
0
```

This *truncating division* was inherited from C-based languages, but confused people who don't know those languages, such as those coming from JavaScript or pure math.

In Python 3, dividing two integers results in a float:

```
>>> print(2 / 5)
0.4
```

The `//` operator, which was added all the way back in Python 2.2, always performs truncating division:

```
whole_minutes = seconds // 60
```

The `from __future__ import division` directive causes the `/` operator to behave the same in Python 2 as it does in Python 3. We recommend adding it to all modules that use the division operator, so that differences between Python versions are minimal.

When adding the future import, check all divisions in the file and decide if the operator should be changed to `//`.

7.1.1 Special Methods

To overload the `/` operator for a class in Python 2, one defined the `__div__` special method. With the division change, there are two methods to define:

- `__floordiv__`

  Defines the behavior of the `//` operator.

- `__truediv__`

  Defines the behavior of the `/` operator in Python 3, and in Python 2 when the `division` future import is in effect.

- `__div__`

  Defines the behavior of the `/` operator in Python 2, when the `division` future import is *not* in effect.

  Not used at all in Python 3.

Check all classes that define `__div__`, and add `__floordiv__` and/or `__truediv__` as needed. This can be done with a simple alias:

```
class CustomClass(object):
    def __div__(self, other):
        return _divide(self, other)

    __truediv__ = __div__
```

7.2 Unification of `int` and `long`

Python 3 does not have a `long` type. Instead, `int` itself allows large values (limited only by available memory); in effect, Python 2's `long` was renamed to `int`.

This change has several consequences.

7.2.1 Removal of the `long` type

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_long`
- Prevalence: Common

The `long` builtin no longer exists.

In Python 2, calling `int` on a number that doesn't fit in the machine `int` range would automatically create a `long` with the appropriate value.

The same automatic conversion to `long` happened on all operations on `int` that overflow: for example, `10**50` resulted in a `long` on most systems.

The range of Python 2's `int` was system-dependent. Together with the automatic conversion, this means that code that depends on the `long`/`int` distinction is fragile – Python 2 didn't provide very strong guarantees regarding the distinction.

If your code relies on the distinction, you will need to modify it.

Once your code does not rely on the `long`/`int` distinction, you can replace all calls to `long` with `int`. The recommended fixer will do this.

7.2.2 The `L` suffix not allowed in numeric literals

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_numliterals` (but see below)
- Prevalence: Very common

In Python 2, `12345L` designated a `long` literal. For numbers that exceed the range of `int`, the `L` suffix was optional: `1234567890123456789012345678901234567890` always named a `long` on current architectures.

In Python 3, the `L` suffix is not allowed.

In code that does not depend on the `int`/`long` distinction, you can simply drop the `L` suffix. The recommended fixer will do this, along with *octal literal fixes* described below.

If the specific type is important, you will need to refactor the code so that it does not rely on the distinction, as discussed above.

7.2.3 The `L` suffix dropped from the representation

- *Fixer*: None
- Prevalence: Rare

In Python 2, canonical representations of long integers included the `L` suffix. For example, `repr(2**64)` was `18446744073709551616L` on most systems. In Python 3, the suffix does not appear. Note that this only affected `repr`, the string representation (given by `str()` or `print()`) had no suffix.

The canonical representations are rarely used, except in doctests.

As discussed previously, relying on the `int`/`long` distinction is fragile. By extension, relying on the output of `repr` of long numbers is also fragile. Call `str()` instead of `repr()` when the result might be a (long) integer.

7.3 Octal Literals

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_numliterals` (but see below)
- Prevalence: Uncommon

Python 2's other holdover from C-based languages is the syntax of octal literals: zero-prefixed numbers are interpreted in base 8. For example, the value of `0123` was `83`, and `0987` caused a rather unhelpful SyntaxError. This is surprising to those not familiar with C, and it can lead to hard-to-spot errors.

Python 2.6 introduced the `0o` prefix as an alternative to plain `0`. Python 3 drops the `0` prefix: integer literals that start with `0` are illegal (except zero itself, and `0x`/`0o`/`0b` prefixes).

You will need to change the leading zero in all `0`-prefixed literals to `0o`. The recommended fixer will do this automatically, along with *long literal fixes* described above.

CHAPTER 8

Strings

From a developer's point of view, the largest change in Python 3 is the handling of strings. In Python 2, the `str` type was used for two different kinds of values – *text* and *bytes*, whereas in Python 3, these are separate and incompatible types.

- **Text** contains human-readable messages, represented as a sequence of Unicode codepoints. Usually, it does not contain unprintable control characters such as `\0`.

  This type is available as `str` in Python 3, and `unicode` in Python 2.

  In code, we will refer to this type as `unicode` – a short, unambiguous name, although one that is not built-in in Python 3. Some projects refer to it as `six.text_type` (from the *six library*).

- **Bytes** or *bytestring* is a binary serialization format suitable for storing data on disk or sending it over the wire. It is a sequence of integers between 0 and 255. Most data – images, sound, configuration info, or *text* – can be serialized (encoded) to bytes and deserialized (decoded) from bytes, using an appropriate protocol such as PNG, VAW, JSON or UTF-8.

  In both Python 2.6+ and 3, this type is available as `bytes`.

Ideally, every "stringy" value will explicitly and unambiguously be one of these types (or the native string, below). This means that you need to go through the entire codebase, and decide which value is what type. Unfortunately, this process generally cannot be automated.

We recommend replacing the word "string" in developer documentation (including docstrings and comments) with either "text"/"text string" or "bytes"/"byte string", as appropriate.

8.1 The Native String

Additionally, code that supports both Python 2 and 3 in the same codebase can use what is conceptually a third type:

- The **native string** (`str`) – text in Python 3, bytes in Python 2

Custom `__str__` and `__repr__` methods and code that deals with Python language objects (such as attribute/function names) will always need to use the native string, because that is what each version of Python uses for internal text-like data. Developer-oriented texts, such as exception messages, could also be native strings.

For other data, you should only use the native string if all of the following hold:

- you are working with textual data,
- Under Python 2, each "native string" value has a single well-defined encoding (e.g. `UTF-8` or `locale.getpreferredencoding()`), and
- you do not mix native strings with either bytes or text – always encode/decode diligently when converting to these types.

Native strings affect the semantics under Python 2 as little as possible, while not requiring the resulting Python 3 API to feel bad. But, having a third incompatible type makes the porting process harder. Native strings are suitable mostly for conservative projects, where ensuring stability under Python 2 justifies extra porting effort.

8.2 Conversion between text and bytes

It is possible to `encode()` text to binary data, or `decode()` bytes into a text string, using a particular encoding. By itself, a bytes object has no inherent encoding, so it is not possible to encode/decode without knowing the encoding.

It's similar to images: an open image file might be encoded in PNG, JPG, or another image format, so it's not possible to "just read" the file without either relying on external data (such as the filename), or effectively trying all alternatives. Unlike images, one bytestring can often be successfully decoded using more than one encoding.

8.2.1 Encodings

Never assume a text encoding without consulting relevant documentation and/or researching a string's use cases. If an encoding for a particular use case is determined, document it. For example, a function docstring can specify that some argument is "a bytestring holding UTF-8-encoded text data", or module documentation may clarify that „as per RFC 4514, LDAP attribute names are encoded to UTF-8 for transmission".

Some common text encodings are:

- `UTF-8`: A widely used encoding that can encode any Unicode text, using one to four bytes per character.
- `UTF-16`: Used in some APIs, most notably Windows and Java ones. Can also encode the entire Unicode character set, but uses two to four bytes per character.
- `ascii`: A 7-bit (128-character) encoding, useful for some machine-readable identifiers such as hostnames (`'python.org'`), or textual representations of numbers (`'1234'`, `'127.0.0.1'`). Always check the relevant standard/protocol/documentation before assuming a string can only ever be pure ASCII.
- `locale.getpreferredencoding()`: The "preferred encoding" for command-line arguments, environment variables, and terminal input/output.

If *you* are choosing an encoding to use – for example, your application *defines* a file format rather than using a format standardized externally – consider `UTF-8`. And whatever your choice is, explicitly document it.

8.2.2 Conversion to text

There is no built-in function that converts to text in both Python versions. The *six library* provides `six.text_type`, which is fine if it appears once or twice in uncomplicated code. For better readability, we recommend using `unicode`, which is unambiguous and clear, but it needs to be introduced with the following code at the beginning of a file:

```
try:
    # Python 2: "unicode" is built-in
    unicode
except NameError:
    unicode = str
```

8.2.3 Conversion to bytes

There is no good function that converts an arbitrary object to bytes, as this operation does not make sense on arbitrary objects. Depending on what you need, explicitly use a serialization function (e.g. `pickle.dumps()`), or convert to text and encode the text.

8.3 String Literals

- *Fixer*: None
- Prevalence: Very common

Quoted string literals can be prefixed with `b` or `u` to get bytes or text, respectively. These prefixes work both in Python 2 (2.6+) and 3 (3.3+). Literals without these prefixes result in native strings.

In Python 3, the `u` prefix does nothing; it is only allowed for backwards compatibility. Likewise, the `b` prefix does nothing in Python 2.

Add a `b` or `u` prefix to all strings, unless a native string is desired. Unfortunately, the choice between text and bytes cannot generally be automated.

8.3.1 Raw Unicode strings

- *Fixer*: None
- Prevalence: Rare

In Python 2, the `r` prefix could be combined with `u` to avoid processing backslash escapes. However, this *did not* turn off processing Unicode escapes (`\u....` or `\U........`), as the `u` prefix took precedence over `r`:

```
>>> print u"\x23\u2744"    # Python 2 with Encoding: UTF-8
#
>>> print ur"\x23\u2744"   # Python 2 with Encoding: UTF-8
\x23
```

This may be confusing at first. Keeping this would be even more confusing in Python 3, where the `u` prefix is a no-op with backwards-compatible behavior. Python 3 avoids the choice between confusing or backwards-incompatible semantics by forbidding `ur` altogether.

Avoid the `ur` prefix in string literals.

The most straightforward way to do this is to use plain `u` literals with `\\` for a literal backslash:

```
>>> print(u"\\x23\u2744")
\x23
```

8.4 String operations

In Python 3, text and bytes cannot be mixed. For example, these are all illegal:

```
b'one' + 'two'

b', '.join(['one', 'two'])

import re
pattern = re.compile(b'a+')
pattern.match('aaaaaa')
```

Encode or decode the data to make the types match.

8.5 Type checking

- *Fixer*: `python-modernize -wnf libmodernize.fixes.fix_basestring`
- Prevalence: Rare

Because the `str` and `unicode` types in Python 2 could be used interchangeably, it sometimes didn't matter which of the types a particular value had. For these cases, Python 2 provided the class `basestring`, from which both `str` and `unicode` derived:

```
if isinstance(value, basestring):
    print("It's stringy!")
```

In Python 3, the concept of `basestring` makes no sense: text is only represented by `str`.

For type-checking text strings in code compatible with both versions, the *six library* offers `string_types`, which is `(basestring,)` in Python 2 and `(str,)` in Python 3. The above code can be replaced by:

```
import six

if isinstance(value, six.string_types):
    print("It's stringy!")
```

The recommended fixer will import `six` and replace any uses of `basestring` by `string_types`.

8.6 File I/O

- *Fixer*: `python-modernize -wnf libmodernize.fixes.fix_open`
- Prevalence: Common

In Python 2, reading from a file opened by `open()` yielded the generic `str`. In Python 3, the type of file contents depends on the mode the file was opened with. By default, this is text strings; `b` in mode selects bytes:

```
with open('/etc/passwd') as f:
    f.read()   # text

with open('/bin/sh', 'rb') as f:
    f.read()   # bytes
```

On disk, all files are stored as bytes. For text-mode files, their content is decoded automatically. The default encoding is `locale.getpreferredencoding(False)`, but this might not always be appropriate, and may cause different behavior across systems. If the encoding of a file is known, we recommend always specifying it:

```
with open('data.txt', encoding='utf-8') as f:
    f.read()
```

Similar considerations apply when writing to files.

The behavior of `open` is quite different between Python 2 and 3. However, from Python 2.6 on, the Python 3 version is available in the `io` module. We recommend replacing the built-in `open` function with `io.open`, and using the new semantics – that is, text files contain `unicode`:

```
from io import open

with open('data.txt', encoding='utf-8') as f:
    f.read()
```

Note that under Python 2, the object returned by `io.open` has a different type than that returned by `open`. If your code does strict type checking, consult the notes on the *file() built-in*.

The recommended fixer will add the `from io import open` import, but it will not add `encoding` arguments. We recommend adding them manually if the encoding is known.

8.7 Testing Strings

When everything is ported and tests are passing, it is a good idea to make sure your code handles strings correctly – even in unusual situations.

Many of the tests recommended below exercise behavior that "works" in Python 2 (does not raise an exception – but may produce subtly wrong results), while a Python 3 version will involve more thought and code.

You might discover mistakes in how the Python 2 version processes strings. In these cases, it might be a good idea to enable new tests for Python 3 only: if some bugs in edge cases survived so far, they can probably live until Python 2 is retired. Apply your own judgement.

Things to test follow.

8.7.1 Non-ASCII data

Ensure that your software works (or, if appropriate, fails cleanly) with non-ASCII input, especially input from end-users. Example characters to check are:

- `é ñ ü Đ ř ů Å ß ç ı İ` (from European personal names)
- ½ (alternate forms and ligatures)
- € ¥ (currency symbols)
- η (various scripts)
- (symbols and emoji)

8.7.2 Encodings and locales

If your software handles multiple text encodings, or handles user-specified encodings, make sure this capability is well-tested.

Under Linux, run your software with the `LC_ALL` environment variable set to `C` and to `tr_TR.utf8`. Check handling of any command-line arguments and environment variables that may contain non-ASCII characters.

8.7.3 Invalid input

Test how the code handles invalid text input. If your software deals with files, try it a on non-UTF8 filename.

Using Python 3, such a file can be created by:

```
with open(b'bad-\xFF-filename', 'wb') as file:
    file.write(b'binary-\xFF-data')
```

CHAPTER 9

Dictionaries

There are three most significant changes related to dictionaries in Python 3.

9.1 Removed `dict.has_key()`

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_has_key` (See caveat below)
- Prevalence: Common

The `dict.has_key()` method, long deprecated in favor of the `in` operator, is no longer available in Python 3.

Instead of:

```
dictionary.has_key('keyname')
```

you should use:

```
'keyname' in dictionary
```

Note that the recommended fixer replaces all calls to *any* `has_key` method; it does not check that its object is actually a dictionary.

If you use a third-party dict-like class, it should implement `in` already. If not, notify its author: it should have been added as part of adding Python 3 support.

If your own codebase contains a custom dict-like class, add a `__contains__()` method to it to implement the `in` operator. If possible, mark the `has_key` method as deprecated. Then run the fixer, and review the output. Typically, the fixer's changes will need to be reverted in tests for the `has_key` method itself.

If you are using objects with unrelated semantics for the attribute `has_key`, you'll need to review the fixer's output and revert its changes for such objects.

9.2 Changed Key Order

- *Fixer*: None
- Prevalence: Uncommon

The Python language specification has never guaranteed order of keys in a dictionary, and mentioned that applications shouldn't rely on it. In practice, however, the order of elements in a dict was usually remained consistent between successive executions of Python 2.

Suppose we have a simple script with the following content:

```
$ cat order.py
dictionary = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
print(list(dictionary.items()))
```

With `python2`, the result contained elements of dict in the same order for every execution:

```
$ python2 order.py
[('a', 1), ('c', 3), ('b', 2), ('e', 5), ('d', 4)]

$ python2 order.py
[('a', 1), ('c', 3), ('b', 2), ('e', 5), ('d', 4)]

$ python2 order.py
[('a', 1), ('c', 3), ('b', 2), ('e', 5), ('d', 4)]
```

The predictable ordering is a side effect of predictable `hashing`. Unfortunately, in some cases malicious users could take advantage of the predictability to cause denial of service attacks. (See CVE-2012-1150 for more details.) To counter this vulnerability, Python 2.6.8+ and 2.7.3+ allowed randomizing the hash function, and thus dictionary order, on each invocation of the interpreter. This is done by setting the environment variable `$PYTHONHASHSEED` to `random`:

```
$ PYTHONHASHSEED=random python2 order.py
[('b', 2), ('c', 3), ('a', 1), ('d', 4), ('e', 5)]

$ PYTHONHASHSEED=random python2 order.py
[('e', 5), ('d', 4), ('a', 1), ('c', 3), ('b', 2)]
```

In Python 3.3+, this setting is the default:

```
$ python3 order.py
[('a', 1), ('d', 4), ('e', 5), ('c', 3), ('b', 2)]

$ python3 order.py
[('c', 3), ('e', 5), ('d', 4), ('a', 1), ('b', 2)]
```

Additionally, CPython 3.6+ uses a new implementation of dictionaries, which makes them appear sorted by insertion order (though you can only rely on this behavior in Python 3.7+):

```
$ python3 order.py
[('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)]
```

Unfortunately, an automated fixer for removing dependencies on dict order is not available. However, the issue can be detected by running the code under Python 2 with `PYTHONHASHSEED=random`. Do that, and investigate and fix any failures.

9.3 Dict Views and Iterators

- *Fixer*: `python-modernize -wnf libmodernize.fixes.fix_dict_six` (See caveat below)
- Prevalence: Common

The methods `dict.keys()`, `dict.items()` and `dict.values()` now return views instead of lists.

The following are the most important differences:

- Unlike lists, a view does not hold copy the data. Updates to the underlying dict are reflected in the view.
- The items in a view are not accessible by indexing. If you need that you'll need to convert the view to a list (e.g. `list(d.values())`).
- Key and value views support set operations, such as intersection and union.

The following common operations work the same between views and lists, as long as the underlying dict is not modified:

- Iteration (e.g. `for x in d.values()`)
- Member testing (e.g. `if x in d.values()`)
- Length testing (e.g. `len(d.values())`)

The methods `dict.iterkeys()`, `dict.iteritems()` and `dict.itervalues()`, and the less-used `dict.viewkeys()`, `dict.viewitems()` and `dict.viewvalues()`, are no longer available.

9.3.1 Cross-Version Iteration and Views

To get iterators in both Python 2 and Python 3, calls to `iterkeys()`, `itervalues()` and `iteritems()` can be replaced by calls to functions from the *Compatibility library: six* library:

```
six.iterkeys(dictionary)
six.iteritems(dictionary)
six.itervalues(dictionary)
```

Similarly, `viewkeys()`, `viewvalues()` and `viewitems()` have compatibility wrappers in *Compatibility library: six*:

```
six.viewkeys(dictionary)
six.viewitems(dictionary)
six.viewvalues(dictionary)
```

In Python 3, both `iter*` and `view*` functions correspond to `keys()`, `items()`, and `values()`.

However, we recommend avoiding the `six` wrappers whenever it's sensible. For example, one often sees `iter*` functions in Python 2 code:

```
for v in dictionary.itervalues():
    print(v)
```

To be compatible with Python 3, this code can be changed to use `six`:

```
for v in six.itervalues(dictionary):
    print(v)
```

. . . or a "native" method:

```
for v in dictionary.values():
    print(v)
```

The latter is more readable. However, it can be argued that the former is more memory-efficient in Python 2, as a new list is not created.

In most real-world use cases, the memory difference is entirely negligible: the extra list is a fraction of the size of a dictionary, and tiny compared to the data itself. Any speed difference is almost always negligible. So, we suggest using the more readable variant unless either:

- not all items are processed (for example, a `break` ends the loop early), or
- special optimizations are needed (for example, if the dictionary could contain millions of items or more).

9.3.2 Fixer caveats

The recommended fixer rewrites the usage of dict methods, but very often its changes are not ideal. We recommend treating its output as "markers" that indicate code that needs to change, but addressing each such place individually by hand.

For example, the fixer will change:

```
key_list = dictionary.keys()
for key in key_list:
    print(key)
```

to:

```
key_list = list(dictionary.keys())
for key in key_list:
    print(key)
```

This change is entirely unnecessary. The new version is less performant (in both Python 2 and Python 3), and less readable. However, the fixer cannot detect that the list is only used for iteration, so it emits overly defensive code.

In this case, both speed and readability can be improved by iterating over the dict itself:

```
for key in dictionary:
    print(key)
```

Also, the fixer will not change instances code that modifies a dictionary while iterating over it. The following is valid in Python 2, where an extra copy of keys is iterated over:

```
for key in dictionary.keys():
    del dictionary[key]
```

In Python 3, this will raise `RuntimeError: dictionary changed size during iteration`.

In this particular case, `dictionary.clear()` can replace the loop. More generally, this kind of issue may be solved by creating a list explicitly:

```
for key in list(dictionary.keys()):
    del dictionary[key]
```

The fixer will not change code like this. However, the `RuntimeError` makes the issue easy to detect.

CHAPTER 10

Iterators

Python 3 changes return values of several basic functions from list to iterator. The main reason for this change is that iterators usually cause better memory consumption than lists.

If you need to keep Python2-compatible behavior, you can wrap the affected functions with a call to `list`. However, in most cases it is better to apply a more specific fix.

10.1 New behavior of `map()` and `filter()`

- *Fixers* (See caveat below):
  - `python-modernize -wnf libmodernize.fixes.fix_map`
  - `python-modernize -wnf libmodernize.fixes.fix_filter`
- Prevalence: Common

In Python 3, the `map()` and `filter()` functions return iterators (`map` or `filter` objects, respectively). In Python 2, they returned lists.

In Python 2, the iterator behavior is available as `itertools.imap()` and `itertools.ifilter()`.

The *Compatibility library: six* library provides the iterator behavior under names common to both Python versions: `from six.moves import map` and `from six.moves import filter`.

10.1.1 Higher-order functions vs. List Comprehensions

The `map` and `filter` functions are often used with `lambda` functions to change or filter iterables. For example:

```
numbers = [1, 2, 3, 4, 5, 6, 7]

powers_of_two = map(lambda x: 2**x, numbers)

```

(continues on next page)

(continued from previous page)

```
for number in filter(lambda x: x < 20, powers_of_two):
    print(number)
```

In these cases, the call can be rewritten using a list comprehension, making the code faster and more readable:

```
numbers = [1, 2, 3, 4, 5, 6, 7]

powers_of_two = [2**x for x in numbers]

for number in [x for x in powers_of_two if x < 20]:
    print(number)
```

If named functions, rather than `lambda`, are used, we also recommend rewriting the code to use a list comprehension. For example, this code:

```
def power_function(x):
    return(2**x)

powered = map(power_function, numbers)
```

should be changed to:

```
def power_function(x):
    return(2**x)

powered = [power_function(num) for num in numbers]
```

Alternatively, you can keep the higher-order function call, and wrap the result in `list`. However, many people will find the resulting code less readable:

```
def power_function(x):
    return(2**x)

powered = list(map(power_function, numbers))
```

10.1.2 Iterators vs. Lists

In cases where the result of `map` or `filter` is only iterated over, and only once, it makes sense to use a *generator expression* rather than a list. For example, this code:

```
numbers = [1, 2, 3, 4, 5, 6, 7]

powers_of_two = map(lambda x: 2**x, numbers)

for number in filter(lambda x: x < 20, powers_of_two):
    print(number)
```

can be rewritten as:

```
numbers = [1, 2, 3, 4, 5, 6, 7]

powers_of_two = (2**x for x in numbers)

for number in (x**2 for x in powers_of_two if x < 20):
    print(number)
```

This keeps memory requirements to a minimum. However, the resulting generator object is much less powerful than a list: it cannot be mutated, indexed or sliced, or iterated more than once.

10.1.3 Fixer Considerations

When the recommended fixers detect calls to `map()` or `filter()`, they add the imports `from six.moves import filter` or `from six.moves import map` to the top of the file.

In many cases, the fixers do a good job discerning the different usages of `map()` and `filter()` and, if necessary, adding a call to `list()`. But they are not perfect. Always review the fixers' result with the above advice in mind.

The fixers do not work properly if the names `map` or `filter` are rebound to something else than the built-in functions. If your code does this, you'll need to do appropriate changes manually.

10.2 New behavior of `zip()`

- *Fixer*: `python-modernize -wnf libmodernize.fixes.fix_zip` (See caveat below)
- Prevalence: Common

Similarly to `map` and `filter` above, in Python 3, the `zip()` function returns an iterator (specifically, a `zip` object). In Python 2, it returned a list.

The *Compatibility library: six* library provides the iterator behavior under a name common to both Python versions, using the `from six.moves import zip` statement.

With this import in place, the call `zip(...)` can be rewritten to `list(zip(...))`. Note, however, that the `list` is unnecessary when the result is only iterated over, and only iterated once, as in `for items in zip(...)`.

The recommended fixer adds the mentioned import, and changes calls to `list(zip(...)` if necessary. If you review the result, you might find additional places where conversion to `list` is not necessary.

The fixer does not work properly if the name `zip` is rebound to something else than the built-in function. If your code does this, you'll need to do appropriate changes manually.

10.3 New behavior of `range()`

- *Fixer*: `python-modernize -wnf libmodernize.fixes.fix_xrange_six` (See caveat below)
- Prevalence: Common

In Python 3, the `range` function returns an iterable `range` object, like the `xrange()` function did in Python 2. The `xrange` function was removed in Python 3.

Note that Python 3's `range` object, like `xrange` in Python 2, supports many list-like operations: for example indexing, slicing, length queries using `len()`, or membership testing using `in`. Also, unlike `map`, `filter` and `zip` objects, the `range` object can be iterated multiple times.

The *Compatibility library: six* library provides the "`xrange`" behavior in both Python versions, using the `from six.moves import range` statement.

Using this import, the calls:

```
a_list = range(9)
a_range_object = xrange(9)
```

can be replaced with:

```
from six.moves import range

a_list = list(range(9))
a_range_object = range(9)
```

The fixer does the change automatically.

Note that in many cases, code will work the same under both versions with just the built-in `range` function. If the result is not mutated, and the number of elements doesn't exceed several thousands, the list and the range behave very similarly. In this case, just change `xrange` to `range`; no import is needed.

If the name `range` is rebound to something else than the built-in function, the fixer will not work properly. In this case you'll need to do appropriate changes manually.

10.4 New iteration protocol: `next()`

- *Fixer*: `python-modernize -wnf libmodernize.fixes.fix_next` (See caveat below)
- Prevalence: Common

In Python 3, the built-in function `next()` is used to get the next result from an iterator. It works by calling the `__next__()` special method, similarly to how `len()` calls `iterator.__len__`. In Python 2, iterators had the `next` method.

The `next()` built-in was backported to Python 2.6+, where it calls the `next` method.

When getting items from an iterator, the `next` built-in function should be used instead of the `next` method. For example, the code:

```
iterator = iter([1, 2, 3])
one = iterator.next()
two = iterator.next()
three = iterator.next()
```

should be rewritten as:

```
iterator = iter([1, 2, 3])
one = next(iterator)
two = next(iterator)
three = next(iterator)
```

Another change concerns custom iterator classes. These should provide both methods, `next` and `__next__`. An easy way to do this is to define `__next__`, and assign that function to `next` as well:

```
class IteratorOfZeroes(object):
    def __next__(self):
        return 0

    next = __next__  # for Python 2
```

The recommended fixer will only do the first change – rewriting `next` calls. Additionally, it will rewrite calls to *any* method called `next`, whether it is used for iterating or not. If you use a class that uses `next` for an unrelated purpose, check the fixer's output and revert the changes for objects of this class.

The fixer will not add a `__next__` method to your classes. You will need to do this manually.

10.5 Generators cannot raise `StopIteration`

- *Fixer*: None
- Prevalence: Rare

Since Python 3.7, generators cannot raise `StopIteration` directly, but must stop with `return` (or at the end of the function). This change was done to prevent subtle errors when a `StopIteration` exception "leaks" between unrelated generators.

For example, the following generator is considered a programming error, and in Python 3.7+ it raises `RuntimeError`:

```
def count_to(maximum):
    i = 0
    while True:
        yield i
        i += 1
        if i >= maximum:
            raise StopIteration()
```

Convert the `raise StopIteration()` to `return`.

If your code uses a helper function that can raise `StopIteration` to end the generator that calls it, you will need to move the returning logic to the generator itself.

CHAPTER 11

Built-In Function Changes

Python 3 saw some changes to built-in functions. These changes are detailed in this section.

11.1 The `print()` function

- *Fixer*: `python-modernize -wnf libmodernize.fixes.fix_print`
- Prevalence: Very Common

Before Python first introduced keyword arguments, and even functions with variable numbers of arguments, it had the `print` statement. It worked for simple use cases, but grew idiosyncratic syntax for advanced features like (not) ending lines and output to arbitrary files:

```
print 'a + b =',
print a + b
print >> sys.stderr, 'Computed the sum'
```

In Python 3, the statement is gone. Instead, you can use the `print()` *function*, which has clear semantics (but requires an extra pair of parentheses in the common case):

```
print('a + b =', end=' ')
print(a + b)
print('Computed the sum', file=sys.stderr)
```

The function form of `print` is available in Python 2.6+, but to use it, the statement form must be turned off with a future import:

```
from __future__ import print_function
```

The recommended fixer will add the future import and rewrite all uses of `print`.

11.2 Safe `input()`

- *Fixer*: `python-modernize -wnf libmodernize.fixes.fix_input_six`
- Prevalence: Uncommon

In Python 2, the function `input()` read a line from standard input, *evaluated it as Python code*, and returned the result. This is almost never useful – most users aren't expected to know Python syntax. It is also a security risk, as it allows users to run arbitrary code.

Python 2 also had a sane version, `raw_input()`, which read a line and returned it as a string.

In Python 3, `input()` has the sane semantics, and `raw_input` was removed.

The *Compatibility library: six* library includes a helper, `six.moves.input`, that has the Python 3 semantics in both versions.

The recommended fixer will import that helper as `input`, replace `raw_input(...)` with `input(...)`, and replace `input(...)` with `eval(input(...))`. After running it, examine the output to determine if any `eval()` it produces is really necessary.

11.3 Removed `file()`

- *Fixer*: `python-modernize -wnf libmodernize.fixes.fix_file` (but see below)
- Prevalence: Rare

In Python 2, `file()` was the type of an open file. It was used in two ways:

- To open files, i.e. as an alias for `open()`. The documentation mentions that `open` is more appropriate for this case.
- To check if an object is a file, as in `isinstance(f, file)`.

The recommended fixer addresses the first use: it will rewrite all calls to `file()` to `open()`. If your code uses the name `file` for a different function, you will need to revert the fixer's change.

The fixer does not address the second case. There are many kinds of file-like objects in Python; in most circumstances it is better to check for a `read` or `write` method instead of querying the type. This guide's *section on strings* even recommends using the `io` library, whose `open` function produces file-like objects that aren't of the `file` type.

If type-checking for files is necessary, we recommend using a tuple of types that includes `io.IOBase` and, under Python 2, `file`:

```
import io

try:
    # Python 2: "file" is built-in
    file_types = file, io.IOBase
except NameError:
    # Python 3: "file" fully replaced with IOBase
    file_types = (io.IOBase,)

...
isinstance(f, file_types)
```

11.4 Removed `apply()`

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_apply` (but see below)
- Prevalence: Common

In Python 2, the function `apply()` was built in. It was useful before Python added support for passing an argument list to a function via the `*` syntax.

The code:

```
arguments = [7, 3]
apply(complex, arguments)
```

can be replaced with:

```
arguments = [7, 3]
complex(*arguments)
```

The recommended fixer replaces all calls to `apply` with the new syntax. If the variable `apply` names a different function in some of your modules, revert the fixer's changes in that module.

11.5 Moved `reduce()`

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_reduce`
- Prevalence: Uncommon

In Python 2, the function `reduce()` was built in. In Python 3, in an effort to reduce the number of builtins, it was moved to the `functools` module.

The new location is also available in Python 2.6+, so this removal can be fixed by importing it for all versions of Python:

```
from functools import reduce
```

The recommended fixer will add this import automatically.

11.6 The `exec()` function

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_exec`
- Prevalence: Rare

In Python 2, `exec()` was a statement. In Python 3, it is a function.

There were three cases for the statement form of `exec`:

```
exec some_code
exec some_code in globals
exec some_code in globals, locals
```

Similarly, the function `exec` takes one to three arguments:

```
exec(some_code)
exec(some_code, globals)
exec(some_code, globals, locals)
```

In Python 2, the syntax was extended so the first expression may be a 2- or 3-tuple. This means the function-like syntax works even in Python 2.

The recommended fixer will convert all uses of `exec` to the function-like syntax.

11.7 Removed `execfile()`

- *Fixer*: None recommended
- Prevalence: Very rare

Python 2 included the function `execfile()`, which executed a Python file by name. The call:

```
execfile(filename)
```

was roughly equivalent to:

```
from io import open

def compile_file(filename):
    with open(filename, encoding='utf-8') as f:
        return compile(f.read(), filename, 'exec')

exec(compile_file(filename))
```

If your code uses `execfile`, add the above `compile_file` function to an appropriate place, then change all calls to `execfile` to `exec` as above.

Although *Automated fixer: python-modernize* has an `execfile` fixer, we don't recommend using it, as it doesn't close the file correctly.

Note that the above hard-codes the `utf-8` encoding (which also works if your code uses ASCII). If your code uses a different encoding, substitute that. If you don't know the encoding in advance, you will need to honor PEP 263 special comments: on Python 3 use the above with `tokenize.open()` instead of `open()`, and on Python 2 fall back to the old `execfile()`.

The *io.open()* function is discussed in this guide's *section on strings*.

11.8 Moved `reload()`

- *Fixer*: None
- Prevalence: Very rare

The `reload()` function was built-in in Python 2. In Python 3, it is moved to the `importlib` module.

Python 2.7 included an `importlib` module, but without a `reload` function. Python 2.6 and below didn't have an `importlib` module.

If your code uses `reload()`, import it conditionally if it doesn't exist (using feature detection):

```
try:
    # Python 2: "reload" is built-in
    reload
except NameError:
    from importlib import reload
```

11.9 Moved `intern()`

- *Fixer*: None
- Prevalence: Very rare

The `intern()` function was built-in in Python 2. In Python 3, it is moved to the `sys` module.

If your code uses `intern()`, import it conditionally if it doesn't exist (using feature detection):

```
try:
    # Python 2: "intern" is built-in
    intern
except NameError:
    from sys import intern
```

11.10 Removed `coerce()`

- *Fixer*: None
- Prevalence: Rare

Python 3 removes the deprecated function `coerce()`, which was only useful in early versions of Python.

If your code uses it, modify the code to not require it.

If any of your classes defines the special method `__coerce__`, remove that as well, and test that the removal did not break semantics.

CHAPTER 12

Comparing and Sorting

Python 3 is strict when comparing objects of disparate types. It also drops *cmp*-based comparison and sorting in favor of rich comparisons and key-based sorting, modern alternatives that have been available at least since Python 2.4. Details and porting strategies follow.

12.1 Unorderable Types

The strict approach to comparing in Python 3 makes it generally impossible to compare different types of objects.

For example, in Python 2, comparing `int` and `str` works (with results that are unpredictable across Python implementations):

```
>>> 2 < '2'
True
```

but in Python 3, it fails with a well described error message:

```
>>> 2 < '2'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() < str()
```

The change usually manifests itself in sorting lists: in Python 3, lists with items of different types are generally not sortable.

If you need to sort heterogeneous lists, or compare different types of objects, implement a key function to fully describe how disparate types should be ordered.

12.2 Rich Comparisons

- *Fixer*: None

- Prevalence: Common

The `__cmp__()` special method is no longer honored in Python 3.

In Python 2, `__cmp__(self, other)` implemented comparison between two objects, returning a negative value if `self < other`, positive if `self > other`, and zero if they were equal.

This approach of representing comparison results is common in C-style languages. But, early in Python 2 development, it became apparent that only allowing three cases for the relative order of objects is too limiting.

This led to the introduction of *rich comparison* methods, which assign a special method to each operator:

Operator	Method
==	`__eq__`
!=	`__ne__`
<	`__lt__`
<=	`__le__`
>	`__gt__`
>=	`__ge__`

Each takes the same two arguments as *cmp*, and must return either a result value (typically Boolean), raise an exception, or return `NotImplemented` to signal the operation is not defined.

In Python 3, the *cmp* style of comparisons was dropped. All objects that implemented `__cmp__` must be updated to implement *all* of the rich methods instead. (There is one exception: on Python 3, `__ne__` will, by default, delegate to `__eq__` and return the inverted result . However, this is *not* the case in Python 2.)

To avoid the hassle of providing all six functions, you can implement `__eq__`, `__ne__`, and only one of the ordering operators, and use the `functools.total_ordering()` decorator to fill in the rest. Note that the decorator is not available in Python 2.6. If you need to support that version, you'll need to supply all six methods.

The `@total_ordering` decorator does come with the cost of somewhat slower execution and more complex stack traces for the derived comparison methods, so defining all six explicitly may be necessary in some cases even if Python 2.6 support is dropped.

As an example, suppose that you have a class to represent a person with `__cmp__()` implemented:

```
class Person(object):
    def __init__(self, firstname, lastname):
         self.first = firstname
         self.last = lastname

    def __cmp__(self, other):
        return cmp((self.last, self.first), (other.last, other.first))

    def __repr__(self):
        return "%s %s" % (self.first, self.last)
```

With `total_ordering`, the class would become:

```
from functools import total_ordering

@total_ordering
class Person(object):

    def __init__(self, firstname, lastname):
        self.first = firstname
        self.last = lastname
```

(continues on next page)

(continued from previous page)

```
    def __eq__(self, other):
        return ((self.last, self.first) == (other.last, other.first))

    def __ne__(self, other):
        return not (self == other)

    def __lt__(self, other):
        return ((self.last, self.first) < (other.last, other.first))

    def __repr__(self):
        return "%s %s" % (self.first, self.last)
```

If `total_ordering` cannot be used, or if efficiency is important, all methods can be given explicitly:

```
class Person(object):

    def __init__(self, firstname, lastname):
        self.first = firstname
        self.last = lastname

    def __eq__(self, other):
        return ((self.last, self.first) == (other.last, other.first))

    def __ne__(self, other):
        return ((self.last, self.first) != (other.last, other.first))

    def __lt__(self, other):
        return ((self.last, self.first) < (other.last, other.first))

    def __le__(self, other):
        return ((self.last, self.first) <= (other.last, other.first))

    def __gt__(self, other):
        return ((self.last, self.first) > (other.last, other.first))

    def __ge__(self, other):
        return ((self.last, self.first) >= (other.last, other.first))

    def __repr__(self):
        return "%s %s" % (self.first, self.last)
```

12.3 The `cmp` Function

- *Fixer*: None
- Prevalence: Common

As part of the move away from *cmp*-style comparisons, the `cmp()` function was removed in Python 3.

If it is necessary (usually to conform to an external API), you can provide it with this code:

```
def cmp(x, y):
    """
    Replacement for built-in function cmp that was removed in Python 3
```

(continues on next page)

(continued from previous page)

```
    Compare the two objects x and y and return an integer according to
    the outcome. The return value is negative if x < y, zero if x == y
    and strictly positive if x > y.
    """

    return (x > y) - (x < y)
```

The expression used is not straightforward, so if you need the functionality, we recommend adding the full, documented function to your project's utility library.

12.4 The `cmp` Argument

- *Fixer*: None
- Prevalence: Uncommon

In Python 2, `.sort()` or `sorted()` functions have a `cmp` parameter, which determines the sort order. The argument for `cmp` is a function that, like all *cmp*-style functions, returns a negative, zero, or positive result depending on the order of its two arguments.

For example, given a list of instances of a Person class (defined above):

```
>>> actors = [Person('Eric', 'Idle'),
...           Person('John', 'Cleese'),
...           Person('Michael', 'Palin'),
...           Person('Terry', 'Gilliam'),
...           Person('Terry', 'Jones')]
...
```

one way to sort it by last name in Python 2 would be:

```
>>> def cmp_last_name(a, b):
...     """ Compare names by last name"""
...     return cmp(a.last, b.last)
...
>>> sorted(actors, cmp=cmp_last_name)
['John Cleese', 'Terry Gilliam', 'Eric Idle', 'Terry Jones', 'Michael Palin']
```

This function is called many times – O(n log n) – during the comparison.

As an alternative to *cmp*, sorting functions can take a keyword-only `key` parameter, a function that returns the key under which to sort:

```
>>> def keyfunction(item):
...     """Key for comparison by last name"""
...     return item.last
...
>>> sorted(actors, key=keyfunction)
['John Cleese', 'Terry Gilliam', 'Eric Idle', 'Terry Jones', 'Michael Palin']
```

The advantage of this approach is that this function is called only once for each item. When simple types such as tuples, strings, and numbers are used for keys, the many comparisons are then handled by optimized C code. Also, in most cases key functions are more readable than *cmp*: usually, people think of sorting by some aspect of an object (such as last name), rather than by comparing individual objects. The main disadvantage is that the old *cmp* style is commonly used in C-language APIs, so external libraries are likely to provide similar functions.

In Python 3, the `cmp` parameter was removed, and only `key` (or no argument at all) can be used.

There is no fixer for this change. However, discovering it is straightforward: the calling `sort` with the `cmp` argument raises TypeError in Python 3. Each *cmp* function must be replaced by a *key* function. There are two ways to do this:

- If the function did a common operation on both arguments, and then compared the results, replace it by just the common operation. In other words, `cmp(f(a), f(b))` should be replaced with `f(item)`
- If the above does not apply, wrap the *cmp*-style function with `functools.cmp_to_key()`. See its documentation for details.

  The `cmp_to_key` function is not available in Python 2.6, so if you need to support that version, you'll need copy it from Python sources

CHAPTER 13

Classes

Python 3 drops support for "old-style" classes, and introduces dedicated syntax for metaclasses. Read on for details.

13.1 New-Style Classes

- *Fixer*: None
- Prevalence: Very common

Python 2 had two styles of classes: "old-style" and "new-style".

Old-style classes were defined without a superclass (or by deriving from other old-style classes):

```
class OldStyle:
    pass

class OldStyleChild(OldStyle):
    pass
```

New-style classes derive from a built-in class – in most cases, `object`:

```
class NewStyle(object):
    pass

class NewInt(int):
    pass
```

In Python 3, all classes are new-style: `object` is the default superclass.

For code compatible across Python versions, all classes should be defined with explicit superclasses: add `(object)` to all class definitions with no superclass list. To find all places to change, you can run the following command over the codebase:

```
grep --perl 'class\s+[a-zA-Z_]+:'
```

However, you will need to test the result thoroughly. Old- and new-style classes have slightly differend semantics, described below.

13.1.1 Method resolution

From a developer's point of view, the main difference between the two is method resolution in multiple inheritance chains. This means that if your code uses multiple inheritance, there can be differences between which method is used for a particular subclass.

The differences are summarized on the Python wiki, and the new semantics are explained in a Howto document from Python 2.3.

13.1.2 Object model details

Another difference is in the behavior of arithmetic operations: in old-style classes, operators like + or `%` generally coerced both operands to the same type. In new-style classes, instead of coercion, several special methods (e.g. `__add__`/`__radd__`) may be tried to arrive at the result.

Other differences are in the object model: only new-style classes have `__mro__` or `mro()`, and writing to special attributes like `__bases__`, `__name__`, `__class__` is restricted or impossible.

13.2 Metaclasses

- *Fixer*: `python-modernize -wnf libmodernize.fixes.fix_metaclass`
- Prevalence: Rare

For metaclasses, Python 2 uses a specially named class attribute:

```
class Foo(Parent):
    __metaclass__ = Meta
```

In Python 3, metaclasses are more powerful, but the metaclass needs to be known before the body of the class statement is executed. For this reason, metaclasses are now specified with a keyword argument:

```
class Foo(Parent, metaclass=Meta):
    ...
```

The new style is not compatible with Python 2 syntax. However, the *Compatibility library: six* library provides a workaround that works in both versions – a base class named `with_metaclass`. This workaround does a bit of magic to ensure that the result is the same as if a metaclass was specified normally:

```
import six

class Foo(six.with_metaclass(Meta, Parent)):
    pass
```

The recommended fixer will import `six` and add `with_metaclass` quite reliably, but do test that the result still works.

CHAPTER 14

Comprehensions

List comprehensions, a shortcut for creating lists, have been in Python since version 2.0. Python 2.4 added a similar feature – generator expressions; then 2.7 (and 3.0) introduced set and dict comprehensions.

All three can be thought as syntactic sugar for defining and calling a generator function, but since list comprehensions came before generators, they behaved slightly differently than the other two. Python 3 removes the differences.

14.1 Leaking of the Iteration Variable

- *Fixer*: None
- Prevalence: Rare

In Python 2, the iteration variable(s) of list comprehensions were considered local to the code containing the expression. For example:

```
>>> powers = [2**i for i in range(10)]
>>> print(i)
9
```

This did *not* apply apply to generators, or to set/dict comprehensions (added in Python 2.7).

In Python 3, list expressions have their own scope: they are *functions*, just defined with a special syntax, and automatically called. Thus, the iteration variable(s) don't "leak" out:

```
>>> powers = [2**i for i in range(10)]
>>> print(i)
Traceback (most recent call last):
  File "...", line 1, in <module>
NameError: name 'i' is not defined
```

In most cases, effects of the change are easy to find, as running the code under Python 3 will result in a NameError. To fix this, either rewrite the code to not use the iteration variable after a list comprehension, or convert the comprehension to a `for` loop:

```
powers = []
for i in range(10):
    powers.append(2**i)
```

In some cases, the change might silently cause different behavior. This is when a variable of the same name is set before the comprehension, or in a surrounding scope. For example:

```
i = 'global'
def foo():
    powers = [2**i for i in range(10)]
    return i

>>> foo()  # Python 2
9
>>> foo()  # Python 3
'global'
```

Unfortunately, you will need to find and fix these cases manually.

14.2 Comprehensions over Tuples

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_paren`
- Prevalence: Rare

Python 2 allowed list comprehensions over bare, non-parenthesized tuples:

```
>>> [i for i in 1, 2, 3]
[1, 2, 3]
```

In Python 3, this is a syntax error. The tuple must be enclosed in parentheses:

```
>>> [i for i in (1, 2, 3)]
[1, 2, 3]
```

The recommended fixer will add the parentheses in the vast majority of cases. It does not deal with nested loops, such as `[x*y for x in 1, 2 for y in 1, 2]`. These cases are easily found, since they raise `SyntaxError` under Python 3. If they appear in your code, add the parentheses manually.

CHAPTER 15

Other Core Object Changes

This page details miscellaneous changes to core objects: functions and classes.

15.1 Function Attributes

- *Fixer*: `python-modernize -wnf fissix.fixes.fix_funcattrs` (but see below)
- Prevalence: Rare

In Python, functions are mutable objects that support custom attributes. In such cases, special attributes (ones provided or used by the Python language itself) are prefixed and postfixed by double underscores.

Function objects predate this convention: their built-in attributes were named with the `func_` prefix instead. However, the new "dunder" names were available, as aliases, even in Python 2.

Python 3 removes the old names for these attributes:

Legacy name	New name
`func_closure`	`__closure__`
`func_code`	`__code__`
`func_defaults`	`__defaults__`
`func_dict`	`__dict__`
`func_doc`	`__doc__`
`func_globals`	`__globals__`
`func_name`	`__name__`

The recommended fixer will replace all of the old attribute names with the new ones. However, it does not check that the attribute is retreived from a function object. If your code uses the `func_*` names for other purposes, you'll need to revert the fixer's changes.

15.2 `__oct__`, `__hex__`

- *Fixer*: None
- Prevalence: Rare

The `__oct__` and `__hex__` special methods customized conversion of custom classes to octal or hexadecimal srting representation, i.e. the behavior of the `oct()` and `hex()` built-in functions.

Python 3 adds the `bin()` function, which converts to binary. Instead of introducing a third name like `__bin__`, all three now just use the integer representation of an object, as returned by the `__index__` method. The `__oct__` and `__hex__` methods are no longer used.

To support both Python 2 and 3, all three must be specified:

```
def IntLike:
    def __init__(self, number):
        self._number = int(number)

    def __index__(self):
        return self._number

    def __hex__(self):
        return hex(self._number)

    def __oct__(self):
        return oct(self._number)
```

If your code defines `__oct__` or `__hex__`, add an `__index__` method that returns an appropriate integer. If your `__oct__` or `__hex__` did not return an octal/hexadecimal representation of an integer before, you'll need to change any code that relied on them.

15.3 Old-style slicing: `__getslice__`, `__setslice__`, `__delslice__`

- *Fixer*: None
- Prevalence: Rare

The special methods `__getslice__`, `__setslice__` and `__delslice__`, which had been deprecated since Python 2.0, are no longer used in Python 3. Item access was unified under `__getitem__`, `__setitem__` and `__delitem__`.

If your code uses them, convert them into equivalent `__getitem__`, `__setitem__` and `__delitem__`, possibly adding the functionality to existing methods.

Keep in mind that `slice` objects have a `step` attribute in addition to `start` and `stop`. If your class does not support all steps, remember to raise an error for the ones you don't support.

For example, the equivalent of:

```
class Slicable(object):
    def __init__(self):
        self.contents = list(range(10))

    def __getslice__(self, start, stop):
        return self.contents[start:stop]
```

(continues on next page)

(continued from previous page)

```
    def __setslice__(self, start, stop, value):
        self.contents[start:stop] = value

    def __delslice__(self, start, stop):
        del self.contents[start:stop]
```

would be:

```
class Slicable(object):
    def __init__(self):
        self.contents = list(range(10))

    def __getitem__(self, item):
        if isinstance(item, slice):
            if item.step not in (1, None):
                raise ValueError('only step=1 supported')
            return self.contents[item.start:item.stop]
        else:
            raise TypeError('non-slice indexing not supported')

    def __setitem__(self, item, value):
        if isinstance(item, slice):
            if item.step not in (1, None):
                raise ValueError('only step=1 supported')
            self.contents[item.start:item.stop] = value
        else:
            raise TypeError('non-slice indexing not supported')

    def __delitem__(self, item):
        if isinstance(item, slice):
            if item.step not in (1, None):
                raise ValueError('only step=1 supported')
            del self.contents[item.start:item.stop]
        else:
            raise TypeError('non-slice indexing not supported')
```

15.4 Customizing truthiness: `__bool__`

- *Fixer*: None
- Prevalence: Common

Python 2 used the `__nonzero__` method to convert an object to boolean, i.e. to provide an implementation for `bool()`.

Other special methods that implement behavior for built-in functions are named after their respective functions. Keeping with this theme, Python 3 uses the name `__bool__` instead of `__nonzero__`.

To make your code compatible, you can provide one implementation, and use an alias for the other name:

```
class Falsy(object):
    def __bool__(self):
        return False

    __nonzero__ = __bool__
```

Do this change in all classes that implement `__nonzero__`.

15.5 Unbound Methods

Python 2 had two kinds of methods: *unbound* methods, which you could retreive from a class object, and *bound* methods, which were retreived from an instance:

```
>>> class Hello(object):
...     def say(self):
...         print('hello world')
...
>>> hello_instance = Hello()
>>> print(Hello.say)
<unbound method Hello.say>
>>> print(hello_instance.say)
<bound method Hello.say of <__main__.Hello object at 0x7f6f40afa790>>
```

Bound methods inject `self` in each call to the method:

```
>>> hello_instance.say()
hello world
```

Unbound methods *checked* if their first argument is an instance of the appropriate class:

```
>>> Hello.say(hello_instance)
hello world
>>> Hello.say(1)
TypeError: unbound method say() must be called with Hello instance as first argument␣
↪(got int instance instead)
```

In Python 3, the concept of unbound methods is gone. Instead, regular functions are used:

```
>>> class Hello(object):
...     def say(self):
...         print('hello world')
...
>>> print(Hello.say)
<function Hello.say at 0x7fdc2803cd90>
```

If your code relies on unbound methods type-checking the `self` argument, or on the fact that unbound methods had a different type than functions, you will need to modify your code. Unfortunately, there is no automated way to tell if that's the case.

CHAPTER 16

Invoking Python

While this is not a change in Python 3, the transition increased the number of systems that have more than one Python interpreter installed: it is not uncommon for `python`, `python2`, `python3`, `python3.6` and `python3.9` to all be valid system commands; other interpreters may be installed in non-standard locations.

This makes it important to use the correct command for each situation.

16.1 Current interpreter

The current Python interpreter should be invoked via `sys.executable`.

Python provides the path of the currently running interpreter as `sys.executable`. This variable should be preferred over `python` or other hard-coded commands.

For example, rather than:

```
subprocess.Popen('python', 'somescript.py')
```

use:

```
subprocess.Popen(sys.executable, 'somescript.py')
```

The assumption that `'python'` is correct is only valid in tightly controlled environments; however, even in those environments `sys.executable` is likely to be correct.

The documentation does include a warning:

> If Python is unable to retrieve the real path to its executable, `sys.executable` will be an empty string or `None`.

In practice, this does not apply to mainstream platforms. If `sys.executable` is unusable, then either your platform's concept of launching a process via filename is somehow unusual (and in this case you should know what to do), or there's an issue in Python itself.

16.2 Unix shebangs

On Unix, executables written in Python must have a shebang line identifying the interpreter. The correct shebang to use will depend on the environment you are targeting and on the version compatibility of the project.

General recommendations for Python shebangs are listed in the For Python script publishers section of PEP 394.

CHAPTER 17

Other Changes

This page documents a few miscellaneous topics at the edges of this guide's scope: low-level buffers, doctests, and bytecode cache files.

17.1 Raw buffer protocol: `buffer` and `memoryview`

- *Fixer*: None
- Prevalence: Very rare

Python 2 used a buffer interface for sharing blocks of memory between Python objects and other libraries.

The buffer object and the corresponding C API proved inaequate, and over Python 2.6 and 2.7, a new mechanism was implemented: the Py_buffer structure and the `memoryview` object.

In Python 3, the buffer object and the related C API is removed.

Unfortunately, the specifics of low-level interfaces between Python and non-Python libraries are too different across projects for us to offer universal advice on porting to the new API. If your code uses `buffer` (or the `PyBuffer` C API), you will need to refer to the Python documentation for details, and combine that with knowledge about your particular interface.

17.2 Doctests

- *Fixer*: None
- Prevalence: Common

Doctests are a common practice for quick tests and testing documentation. They work by extracting snippets example code and its output in documentation, running the code, and verifying that its result matches, textually, the example output.

This relies on minute details of textual representation of Python objects, which is generally not under any backwards-compatibility guarantee and may change at any time – even across minor versions of Python (e.g. 2.6 to 2.7 or 3.5 to 3.6).

Note: Some examples of what changed between Python 2 and 3 are:

- String have different `u` and `b` prefixes depending on if they're bytes or text.
- Large integers lost the `L` suffix.
- The order of items in dictionaries may be different (and unpredictable).

Doctests are a good way to ensure that the *documentation* is correct (i.e. it doesn't contain broken examples), but they are *not* a good way to actually test the code.

If your code uses doctests as the main means of testing, rewrite them as tests that do not rely on exact textual output. You can use the built-in `unittest`, or the third-party pytest library, among others.

Once your doctests are only testing documentation, we recommend the following strategy:

- Keep running doctests under Python 2
- Port all code to be compatible with (and tested on) both Python 2 and 3
- At one moment, update examples in the docs, and start only using Python 3 to run the doctests.

Since the code is tested elsewhere, it generally does not matter that code examples are tested under only one of the supported Python versions.

17.3 Reorganization of `.pyc` files

Since compiling Python code is a relatively expensive operation, and many modules do not change often, Python caches compiled bytecode in `.pyc` files.

In Python 2, `.pyc` files were written in the same directory as the corresponding `.py` source files, with only a `c` added to the filename. The exact mechanism had two major drawbacks:

- Bytecode is not compatible across Python versions. If the same module was being imported by different versions of Python, each would overwrite the `.pyc` file with its own flavor of bytecode on import. This would invalidate the cache for all other versions.
- The `.pyc` cache could be used even without a corresponding `.py` file, which allowed some space saving (by distributing only the compiled file). However, if one deleted a `.py` file but forgot to also remove the `.pyc`, Python would act as if the module was still present. This was quite confusing, especially for beginners.

Python 3 puts `.pyc` files in a separate directory called `__pycache__`, and adds version information to the filename. The new mechanism avoids the above two problems: per-version caches are separated, and if the `.py` source is missing, the `.pyc` file is not considered.

If your code relies on the location of `.pyc` files (for example, if your build/packaging system doesn't handle Python 3), you will need to update to the new location.

If you rely on importing `.pyc` files without corresponding source, you will need to move the `.pyc` to the old location, and remove the version tag. For example, move:

```
__pycache__/foo.cpython-36.pyc
```

to:

```
foo.pyc
```

Under this name, the `.pyc` will be recognized by Python 3's import machinery.

CHAPTER 18

Indices and tables

- genindex
- search

Index