26.4. 2to3 - Automated Python 2 to 3 code translation¶

2to3 is a Python program that reads Python 2.x source code and applies a series of *fixers* to transform it into valid Python 3.x code. The standard library contains a rich set of fixers that will handle almost all code. 2to3 supporting library `lib2to3` is, however, a flexible and generic library, so it is possible to write your own fixers for 2to3. `lib2to3` could also be adapted to custom applications in which Python code needs to be edited automatically.

26.4.1. Using 2to3¶

2to3 will usually be installed with the Python interpreter as a script. It is also located in the `Tools/scripts` directory of the Python root.

2to3's basic arguments are a list of files or directories to transform. The directories are to recursively traversed for Python sources.

Here is a sample Python 2.x source file, `example.py`:

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

It can be converted to Python 3.x code via 2to3 on the command line:

```
$ 2to3 example.py
```

A diff against the original source file is printed. 2to3 can also write the needed modifications right back to the source file. (A backup of the original file is made unless *-n* is also given.) Writing the changes back is enabled with the *-w* flag:

```
$ 2to3 -w example.py
```

After transformation, `example.py` looks like this:

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

Comments and exact indentation are preserved throughout the translation process.

By default, 2to3 runs a set of *predefined fixers*. The *-l* flag lists all available fixers. An explicit set of fixers to run can be given with *-f*. Likewise the *-x* explicitly disables a fixer. The following example runs only the `imports` and `has_key` fixers:

```
$ 2to3 -f imports -f has_key example.py
```

This command runs every fixer except the `apply` fixer:

```
$ 2to3 -x apply example.py
```

Some fixers are *explicit*, meaning they aren't run by default and must be listed on the command line to be run. Here, in addition to the default fixers, the `idioms` fixer is run:

```
$ 2to3 -f all -f idioms example.py
```

Notice how passing `all` enables all default fixers.

Sometimes 2to3 will find a place in your source code that needs to be changed, but 2to3 cannot fix automatically. In this case, 2to3 will print a warning beneath the diff for a file. You should address the warning in order to have compliant 3.x code.

2to3 can also refactor doctests. To enable this mode, use the *-d* flag. Note that *only* doctests will be refactored. This also doesn't require the module to be valid Python. For example, doctest like examples in a reST document could also be refactored with this option.

The -v option enables output of more information on the translation process.

26.4.2. Fixers¶

Each step of tranforming code is encapsulated in a fixer. The command `2to3 -l` lists them. As *documented above*, each can be turned on and off individually. They are described here in more detail.

`apply`¶

Removes usage of `apply()`. For example `apply(function, *args, **kwargs)` is converted to `function(*args, **kwargs)`.

`basestring`¶

Converts `basestring` to `str`.

`buffer`¶

Converts `buffer` to `memoryview`. This fixer is optional because the `memoryview` API is similar but not exactly the same as that of `buffer`.

`callable`¶

Converts `callable(x)` to `hasattr(x, "__call__")`.

`dict`¶

Fixes dictionary iteration methods. `dict.iteritems()` is converted to `dict.items()`, `dict.iterkeys()` to `dict.keys()`, and `dict.itervalues()` to `dict.values()`. It also wraps existing usages of `dict.items()`, `dict.keys()`, and `dict.values()` in a call to `list`.

`except`¶

Converts `except X, T` to `except X as T`.

`exec`¶

Converts the `exec` statement to the `exec()` function.

`execfile`¶

Removes usage of `execfile()`. The argument to `execfile()` is wrapped in calls to `open()`, `compile()`, and `exec()`.

`filter`¶

Wraps `filter()` usage in a `list` call.

`funcattrs`¶

Fixes function attributes that have been renamed. For example, `my_function.func_closure` is converted to `my_function.__closure__`.

`future`¶

Removes `from __future__ import new_feature` statements.

`getcwdu`¶

Renames `os.getcwdu()` to `os.getcwd()`.

`has_key`¶

Changes `dict.has_key(key)` to `key in dict`.

`idioms`¶

This optional fixer preforms several transformations that make Python code more idiomatic. Type comparisions like `type(x) is SomeClass` and `type(x) == SomeClass` are converted to `isinstance(x, SomeClass)`. `while 1` becomes `while True`. This fixer also tries to make use of `sorted()` in appropiate places. For example, this block

```
L = list(some_iterable)
L.sort()
```

is changed to

```
L = sorted(some_iterable)
```

`import`¶

Detects sibling imports and converts them to relative imports.

`imports`¶

Handles module renames in the standard library.

`imports2`¶

Handles other modules renames in the standard library. It is separate from the `imports` fixer only because of technical limitations.

`input`¶

Converts `input(prompt)` to `eval(input(prompt))`

`intern`¶

Converts `intern()` to `sys.intern()`.

`isinstance`¶

Fixes duplicate types in the second argument of `isinstance()`. For example, `isinstance(x, (int, int))` is converted to `isinstance(x, (int))`.

`itertools_imports`¶

Removes imports of `itertools.ifilter()`, `itertools.izip()`, and `itertools.imap()`. Imports of `itertools.ifilterfalse()` are also changed to `itertools.filterfalse()`.

`itertools`¶

Changes usage of itertools.ifilter(), itertools.izip(), and itertools.imap() to their built-in equivalents. itertools.ifilterfalse() is changed to itertools.filterfalse().

long¶

Strips the L prefix on long literals and renames long to int.

map¶

Wraps map() in a list call. It also changes map(None, x) to list(x). Using from future_builtins import map disables this fixer.

metaclass¶

Converts the old metaclass syntax (__metaclass__ = Meta in the class body) to the new (class X(metaclass=Meta)).

methodattrs¶

Fixes old method attribute names. For example, meth.im_func is converted to meth.__func__.

ne¶

Converts the old not-equal syntax, <>, to !=.

next¶

Converts the use of iterator's next() methods to the next() function. It also renames next() methods to __next__().

nonzero¶

Renames __nonzero__() to __bool__().

numliterals¶

Converts octal literals into the new syntax.

paren¶

Add extra parenthesis where they are required in list comprehensions. For example, [x for x in 1, 2] becomes [x for x in (1, 2)].

print¶

Converts the print statement to the print() function.

raises¶

Converts raise E, V to raise E(V), and raise E, V, T to raise E(V).with_traceback(T). If E is a tuple, the translation will be incorrect because substituting tuples for exceptions has been removed in 3.0.

raw_input¶

Converts raw_input() to input().

reduce¶

Handles the move of reduce() to functools.reduce().

renames¶

Changes sys.maxint to sys.maxsize.

repr¶

Replaces backtick repr with the repr() function.

set_literal¶

Replaces use of the set constructor with set literals. This fixer is optional.

standard_error¶

Renames StandardError to Exception.

sys_exc¶

Changes the deprecated sys.exc_value, sys.exc_type, sys.exc_traceback to use sys.exc_info().

throw¶

Fixes the API change in generator's throw() method.

tuple_params¶

Removes implicit tuple parameter unpacking. This fixer inserts temporary variables.

types¶

Fixes code broken from the removal of some members in the types module.

unicode¶

Renames unicode to str.

urllib¶

Handles the rename of urllib and urllib2 to the urllib package.

ws_comma¶

Removes excess whitespace from comma separated items. This fixer is optional.

xrange¶

Renames xrange() to range() and wraps existing range() calls with list.

xreadlines¶

Changes for x in file.xreadlines() to for x in file.

zip¶

Wraps zip() usage in a list call. This is disabled when from future_builtins import zip appears.

26.4.3. lib2to3 - 2to3's library¶

Note

The `lib2to3` API should be considered unstable and may change drastically in the future.

Previous topic

26.3. `unittest` — Unit testing framework

Next topic

26.5. `test` — Regression tests package for Python

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 26. Development Tools »