powered by Google™ Search

Appendix A. Porting Code to Python 3 with 2to3

Difficulty level: ♦♦♦♦♦

❝ *Life is pleasant. Death is peaceful. It's the transition that's troublesome.* ❞
— *Isaac Asimov (attributed)*

‣ show table of contents

A.1. Diving In

So much has changed between Python 2 and Python 3, there are vanishingly few programs that will run unmodified under both. But don't despair! To help with this transition, Python 3 comes with a utility script called `2to3`, which takes your actual Python 2 source code as input and auto-converts as much as it can to Python 3. Case study: porting `chardet` to Python 3 describes how to run the `2to3` script, then shows some things it can't fix automatically. This appendix documents what it *can* fix automatically.

A.2. print statement

In Python 2, `print` was a statement. Whatever you wanted to print simply followed the `print` keyword. In Python 3, `print()` is a function. Whatever you want to print, pass it to `print()` like any other function.

Notes	Python 2	Python 3
①	`print`	`print()`
②	`print 1`	`print(1)`
③	`print 1, 2`	`print(1, 2)`
④	`print 1, 2,`	`print(1, 2, end=' ')`
⑤	`print >>sys.stderr, 1, 2, 3`	`print(1, 2, 3, file=sys.stderr)`

1. To print a blank line, call `print()` without any arguments.
2. To print a single value, call `print()` with one argument.
3. To print two values separated by a space, call `print()` with two arguments.
4. This one is a little tricky. In Python 2, if you ended a `print` statement with a comma, it would print the values separated by spaces, then print a trailing space, then stop without printing a carriage return. (Technically, it's a little more complicated than that. The `print` statement in Python 2 used a now-deprecated attribute called `softspace`. Instead of printing a space, Python 2 would set `sys.stdout.softspace` to 1. The space character wasn't really printed until something else got printed on the same line. If the next `print` statement printed a carriage return, `sys.stdout.softspace` would be set to 0 and the space would never be printed. You probably never noticed the difference unless your application was sensitive to the presence or absence of trailing whitespace in `print`-generated output.) In Python 3, the way to do this is to pass `end=' '` as a keyword argument to the `print()` function. The `end` argument defaults to `'\n'` (a carriage return), so overriding it will suppress the carriage return after printing the other arguments.
5. In Python 2, you could redirect the output to a pipe — like `sys.stderr` — by using the `>>pipe_name` syntax. In Python 3, the way to do this is to pass the pipe in the `file` keyword argument. The `file` argument defaults to `sys.stdout` (standard out), so overriding it will output to a different pipe instead.

A.3. Unicode string literals

Python 2 had two string types: Unicode strings and non-Unicode strings. Python 3 has one string type: Unicode strings.

Notes	Python 2	Python 3
①	`u'PapayaWhip'`	`'PapayaWhip'`
②	`ur'PapayaWhip\foo'`	`r'PapayaWhip\foo'`

1. Unicode string literals are simply converted into string literals, which, in Python 3, are always Unicode.
2. Unicode raw strings (in which Python does not auto-escape backslashes) are converted to raw strings. In Python 3, raw strings are always Unicode.

A 4. unicode() [illegible]

Python 2 had two global functions to coerce objects into strings: `unicode()` to coerce them into Unicode strings, and `str()` to coerce them into non-Unicode strings. Python 3 has only one string type, Unicode strings, so the `str()` function is all you need. (The `unicode()` function no longer exists.)

Notes	Python 2	Python 3
	`unicode(anything)`	`str(anything)`

A 5. long [illegible]

Python 2 had separate `int` and `long` types for non-floating-point numbers. An `int` could not be any larger than `sys.maxint`, which varied by platform. Longs were defined by appending an `L` to the end of the number, and they could be, well, longer than ints. In Python 3, there is only one integer type, called `int`, which mostly behaves like the `long` type in Python 2. Since there are no longer two types, there is no need for special syntax to distinguish them.

Further reading: PEP 237: Unifying Long Integers and Integers.

Notes	Python 2	Python 3
①	`x = 1000000000000L`	`x = 1000000000000`
②	`x = 0xFFFFFFFFFFFFL`	`x = 0xFFFFFFFFFFFF`
③	`long(x)`	`int(x)`
④	`type(x) is long`	`type(x) is int`
⑤	`isinstance(x, long)`	`isinstance(x, int)`

1. Base 10 long integer literals become base 10 integer literals.
2. Base 16 long integer literals become base 16 integer literals.
3. In Python 3, the old `long()` function no longer exists, since longs don't exist. To coerce a variable to an integer, use the `int()` function.
4. To check whether a variable is an integer, get its type and compare it to `int`, not `long`.
5. You can also use the `isinstance()` function to check data types; again, use `int`, not `long`, to check for integers.

A 6. [illegible]

Python 2 supported `<>` as a synonym for `!=`, the not-equals comparison operator. Python 3 supports the `!=` operator, but not `<>`.

Notes	Python 2	Python 3
①	`if x <> y:`	`if x != y:`
②	`if x <> y <> z:`	`if x != y != z:`

1. A simple comparison.
2. A more complex comparison between three values.

A 7. has_key() [illegible]

In Python 2, dictionaries had a `has_key()` method to test whether the dictionary had a certain key. In Python 3, this method no longer exists. Instead, you need to use the `in` operator.

Notes	Python 2	Python 3
①	`a_dictionary.has_key('PapayaWhip')`	`'PapayaWhip' in a_dictionary`
②	`a_dictionary.has_key(x) or a_dictionary.has_key(y)`	`x in a_dictionary or y in a_dictionary`
③	`a_dictionary.has_key(x or y)`	`(x or y) in a_dictionary`
④	`a_dictionary.has_key(x + y)`	`(x + y) in a_dictionary`
⑤	`x + a_dictionary.has_key(y)`	`x + (y in a_dictionary)`

1. The simplest form.
2. The `in` operator takes precedence over the `or` operator, so there is no need for parentheses around `x in a_dictionary` or around `y in`

a_dictionary.

3. On the other hand, you *do* need parentheses around `x or y` here, for the same reason — `in` takes precedence over `or`. (Note: this code is completely different from the previous line. Python interprets `x or y` first, which results in either `x` (if `x` is true in a boolean context) or `y`. Then it takes that singular value and checks whether it is a key in `a_dictionary`.)
4. The `+` operator takes precedence over the `in` operator, so this form technically doesn't need parentheses around `x + y`, but `2to3` includes them anyway.
5. This form definitely needs parentheses around `y in a_dictionary`, since the `+` operator takes precedence over the `in` operator.

A.8. D[illegible]

In Python 2, many dictionary methods returned lists. The most frequently used methods were `keys()`, `items()`, and `values()`. In Python 3, all of these methods return dynamic views. In some contexts, this is not a problem. If the method's return value is immediately passed to another function that iterates through the entire sequence, it makes no difference whether the actual type is a list or a view. In other contexts, it matters a great deal. If you were expecting a complete list with individually addressable elements, your code will choke, because views do not support indexing.

Notes	Python 2	Python 3
①	`a_dictionary.keys()`	`list(a_dictionary.keys())`
②	`a_dictionary.items()`	`list(a_dictionary.items())`
③	`a_dictionary.iterkeys()`	`iter(a_dictionary.keys())`
④	`[i for i in a_dictionary.iterkeys()]`	`[i for i in a_dictionary.keys()]`
⑤	`min(a_dictionary.keys())`	*no change*

1. `2to3` errs on the side of safety, converting the return value from `keys()` to a static list with the `list()` function. This will always work, but it will be less efficient than using a view. You should examine the converted code to see if a list is absolutely necessary, or if a view would do.
2. Another view-to-list conversion, with the `items()` method. `2to3` will do the same thing with the `values()` method.
3. Python 3 does not support the `iterkeys()` method anymore. Use `keys()`, and if necessary, convert the view to an iterator with the `iter()` function.
4. `2to3` recognizes when the `iterkeys()` method is used inside a list comprehension, and converts it to the `keys()` method (without wrapping it in an extra call to `iter()`). This works because views are iterable.
5. `2to3` recognizes that the `keys()` method is immediately passed to a function which iterates through an entire sequence, so there is no need to convert the return value to a list first. The `min()` function will happily iterate through the view instead. This applies to `min()`, `max()`, `sum()`, `list()`, `tuple()`, `set()`, `sorted()`, `any()`, and `all()`.

A.9. M[illegible]

Several modules in the Python Standard Library have been renamed. Several other modules which are related to each other have been combined or reorganized to make their association more logical.

A.9.1. `http`

In Python 3, several related HTTP modules have been combined into a single package, `http`.

Notes	Python 2	Python 3
①	`import httplib`	`import http.client`
②	`import Cookie`	`import http.cookies`
③	`import cookielib`	`import http.cookiejar`

④	`import BaseHTTPServer`
	`import SimpleHTTPServer`
	`import CGIHttpServer`
		`import http.server`

1. The `http.client` module implements a low-level library that can request HTTP resources and interpret HTTP responses.
2. The `http.cookies` module provides a Pythonic interface to browser cookies that are sent in a `Set-Cookie:` HTTP header.
3. The `http.cookiejar` module manipulates the actual files on disk that popular web browsers use to store cookies.
4. The `http.server` module provides a basic HTTP server.

A.9.2. `urllib`

Python 2 had a rat's nest of overlapping modules to parse, encode, and fetch URLs. In Python 3, these have all been refactored and combined in a single package, `urllib`.

Notes	Python 2	Python 3
①	`import urllib`	`import urllib.request, urllib.parse, urllib.error`
②	`import urllib2`	`import urllib.request, urllib.error`
③	`import urlparse`	`import urllib.parse`
④	`import robotparser`	`import urllib.robotparser`

⑤	`from urllib import FancyURLopener`
	`from urllib import urlencode`
		`from urllib.request import FancyURLopener`
		`from urllib.parse import urlencode`

⑥	`from urllib2 import Request`
	`from urllib2 import HTTPError`
		`from urllib.request import Request`
		`from urllib.error import HTTPError`

1. The old `urllib` module in Python 2 had a variety of functions, including `urlopen()` for fetching data and `splittype()`, `splithost()`, and `splituser()` for splitting a URL into its constituent parts. These functions have been reorganized more logically within the new `urllib` package. `2to3` will also change all calls to these functions so they use the new naming scheme.
2. The old `urllib2` module in Python 2 has been folded into the `urllib` package in Python 3. All your `urllib2` favorites — the `build_opener()` method, `Request` objects, and `HTTPBasicAuthHandler` and friends — are still available.
3. The `urllib.parse` module in Python 3 contains all the parsing functions from the old `urlparse` module in Python 2.
4. The `urllib.robotparser` module parses `robots.txt` files.
5. The `FancyURLopener` class, which handles HTTP redirects and other status codes, is still available in the new `urllib.request` module. The `urlencode()` function has moved to `urllib.parse`.
6. The `Request` object is still available in `urllib.request`, but constants like `HTTPError` have been moved to `urllib.error`.

Did I mention that `2to3` will rewrite your function calls too? For example, if your Python 2 code imports the `urllib` module and calls `urllib.urlopen()` to fetch data, `2to3` will fix both the import statement and the function call.

Notes	Python 2	Python 3

	`import urllib`
	`print urllib.urlopen('http://diveintopython3.org/').read()`
		`import urllib.request, urllib.parse, urllib.error`
		`print(urllib.request.urlopen('http://diveintopython3.org/').read())`

A.9.3. `dbm`

All the various DBM clones are now in a single package, `dbm`. If you need a specific variant like GNU DBM, you can import the appropriate module within the `dbm` package.

Notes	Python 2	Python 3
	`import dbm`	`import dbm.ndbm`
	`import gdbm`	`import dbm.gnu`
	`import dbhash`	`import dbm.bsd`
	`import dumbdbm`	`import dbm.dumb`

	`import anydbm`
	`import whichdb`
		`import dbm`

A.9.4. `xmlrpc`

XML-RPC is a lightweight method of performing remote RPC calls over HTTP. The XML-RPC client library and several XML-RPC server implementations are now combined in a single package, `xmlrpc`.

Notes	Python 2	Python 3
	`import xmlrpclib`	`import xmlrpc.client`

	`import DocXMLRPCServer`
	`import SimpleXMLRPCServer`
		`import xmlrpc.server`

A.9.5. Other modules

Notes	Python 2	Python 3

①	`try:`
	`    import cStringIO as StringIO`
	`except ImportError:`
	`    import StringIO`
		`import io`

②	`try:`
	`    import cPickle as pickle`
	`except ImportError:`
	`    import pickle`
		`import pickle`

③	`import __builtin__`	`import builtins`
④	`import copy_reg`	`import copyreg`
⑤	`import Queue`	`import queue`
⑥	`import SocketServer`	`import socketserver`
⑦	`import ConfigParser`	`import configparser`
⑧	`import repr`	`import reprlib`
⑨	`import commands`	`import subprocess`

1. A common idiom in Python 2 was to try to import`cStringIO as StringIO`, and if that failed, to import`StringIO` instead. Do not do this in Python 3; the `io` module does it for you. It will find the fastest implementation available and use it automatically.
2. A similar idiom was used to import the fastest pickle implementation. Do not do this in Python 3; the`pickle` module does it for you.
3. The `builtins` module contains the global functions, classes, and constants used throughout the Python language. Redefining a function in the `builtins` module will redefine the global function everywhere. That is exactly as powerful and scary as it sounds.
4. The `copyreg` module adds pickle support for custom types defined in C.
5. The `queue` module implements a multi-producer, multi-consumer queue.
6. The `socketserver` module provides generic base classes for implementing different kinds of socket servers.
7. The `configparser` module parses I Nstyle configuration files.
8. The `reprlib` module reimplements the built-in`repr()` function, with additional controls on how long the representations can be before they are truncated.
9. The `subprocess` module allows you to spawn processes, connect to their pipes, and obtain their return codes.

[illegible]

A package is a group of related modules that function as a single entity. In Python 2, when modules within a package need to reference each other, you use `import foo` or `from foo import Bar`. The Python 2 interpreter first searches within the current package to find`foo.py`, and then moves on to the other directories in the Python search path (`sys.path`). Python 3 works a bit differently. Instead of searching the current package, it goes directly to the Python search path. If you want one module within a package to import another module in the same package, you need to explicitly provide the relative path between the two modules.

Suppose you had this package, with multiple files in the same directory:

```
chardet/
|
+--__init__.py
|
+--constants.py
|
+--mbcharsetprober.py
|
+--universaldetector.py
```

Now suppose that `universaldetector.py` needs to import the entire`constants.py` file and one class from`mbcharsetprober.py`. How do you do it?

Notes	Python 2	Python 3
①	`import constants`	`from . import constants`
②	`from mbcharsetprober import MultiByteCharSetProber`	`from .mbcharsetprober import MultiByteCharsetProber`

1. When you need to import an entire module from elsewhere in your package, use the new `from . import` syntax. The period is actually a relative path from this file (`universaldetector.py`) to the file you want to import (`constants.py`). In this case, they are in the same directory, thus the single period. You can also import from the parent directory (`from .. import anothermodule`) or a subdirectory.
2. To import a specific class or function from another module directly into your module's namespace, prefix the target module with a relative path, minus the trailing slash. In this case, `mbcharsetprober.py` is in the same directory as `universaldetector.py`, so the path is a single period. You can also import form the parent directory (`from ..anothermodule import AnotherClass`) or a subdirectory.

A 1 . next() [illegible]

In Python 2, iterators had a `next()` method which returned the next item in the sequence. That's still true in Python 3, but there is now also a global `next()` function that takes an iterator as an argument.

Notes	Python 2	Python 3
①	`anIterator.next()`	`next(anIterator)`
②	`a_function_that_returns_an_iterator().next()`	`next(a_function_that_returns_an_iterator())`

③	`class A:`
	`    def next(self):`
	`        pass`
		`class A:`
		`    def __next__(self):`
		`        pass`

④	`class A:`
	`    def next(self, x, y):`
	`        pass`
		no change

⑤	`next = 42`
	`for an_iterator in a_sequence_of_iterators:`
	`    an_iterator.next()`
		`next = 42`
		`for an_iterator in a_sequence_of_iterators:`
		`    an_iterator.__next__()`

1. In the simplest case, instead of calling an iterator's `next()` method, you now pass the iterator itself to the global `next()` function.
2. If you have a function that returns an iterator, call the function and pass the result to the `next()` function. (The `2to3` script is smart enough to convert this properly.)
3. If you define your own class and mean to use it as an iterator, define the `__next__()` special method.
4. If you define your own class and just happen to have a method named `next()` that takes one or more arguments, `2to3` will not touch it. This class can not be used as an iterator, because its `next()` method takes arguments.
5. This one is a bit tricky. If you have a local variable named `next`, then it takes precedence over the new global `next()` function. In this case, you need to call the iterator's special `__next__()` method to get the next item in the sequence. (Alternatively, you could also refactor the code so the local variable wasn't named `next`, but `2to3` will not do that for you automatically.)

A 2 . filter() [illegible]

In Python 2, the `filter()` function returned a list, the result of filtering a sequence through a function that returned `True` or `False` for each item in the sequence. In Python 3, the `filter()` function returns an iterator, not a list.

Notes	Python 2	Python 3
①	`filter(a_function, a_sequence)`	`list(filter(a_function, a_sequence))`
②	`list(filter(a_function, a_sequence))`	*no change*
③	`filter(None, a_sequence)`	`[i for i in a_sequence if i]`
④	`for i in filter(None, a_sequence):`	*no change*
⑤	`[i for i in filter(a_function, a_sequence)]`	*no change*

1. In the most basic case, `2to3` will wrap a call to `filter()` with a call to `list()`, which simply iterates through its argument and returns a real list.
2. However, if the call to `filter()` is *already* wrapped in `list()`, `2to3` will do nothing, since the fact that `filter()` is returning an iterator is irrelevant.
3. For the special syntax of `filter(None, ...)`, `2to3` will transform the call into a semantically equivalent list comprehension.
4. In contexts like `for` loops, which iterate through the entire sequence anyway, no changes are necessary.
5. Again, no changes are necessary, because the list comprehension will iterate through the entire sequence, and it can do that just as well if `filter()` returns an iterator as if it returns a list.

A.13. map() [illegible]

In much the same way as `filter()`, the `map()` function now returns an iterator. (In Python 2, it returned a list.)

Notes	Python 2	Python 3
①	`map(a_function, 'PapayaWhip')`	`list(map(a_function, 'PapayaWhip'))`
②	`map(None, 'PapayaWhip')`	`list('PapayaWhip')`
③	`map(lambda x: x+1, range(42))`	`[x+1 for x in range(42)]`
④	`for i in map(a_function, a_sequence):`	*no change*
⑤	`[i for i in map(a_function, a_sequence)]`	*no change*

1. As with `filter()`, in the most basic case, `2to3` will wrap a call to `map()` with a call to `list()`.
2. For the special syntax of `map(None, ...)`, the identity function, `2to3` will convert it to an equivalent call to `list()`.
3. If the first argument to `map()` is a lambda function, `2to3` will convert it to an equivalent list comprehension.
4. In contexts like `for` loops, which iterate through the entire sequence anyway, no changes are necessary.
5. Again, no changes are necessary, because the list comprehension will iterate through the entire sequence, and it can do that just as well if `map()` returns an iterator as if it returns a list.

A.14. reduce() [illegible]

In Python 3, the `reduce()` function has been removed from the global namespace and placed in the `functools` module.

Notes	Python 2	Python 3

	`reduce(a, b, c)`	`from functools import reduce`
		`reduce(a, b, c)`

A.15. apply() [illegible]

Python 2 had a global function called `apply()`, which took a function `f` and a list `[a, b, c]` and returned `f(a, b, c)`. You can accomplish the same thing by calling the function directly and passing it the list of arguments preceded by an asterisk. In Python 3, the `apply()` function no longer exists; you must use the asterisk notation.

Notes	Python 2	Python 3
①	`apply(a_function, a_list_of_args)`	`a_function(*a_list_of_args)`
②	`apply(a_function, a_list_of_args, a_dictionary_of_named_args)`	`a_function(*a_list_of_args, **a_dictionary_of_named_args)`
③	`apply(a_function, a_list_of_args + z)`	`a_function(*a_list_of_args + z)`
④	`apply(aModule.a_function, a_list_of_args)`	`aModule.a_function(*a_list_of_args)`

1. In the simplest form, you can call a function with a list of arguments (an actual list like `[a, b, c]`) by prepending the list with an asterisk (`*`). This is exactly equivalent to the old `apply()` function in Python 2.
2. In Python 2, the `apply()` function could actually take three parameters: a function, a list of arguments, and a dictionary of named arguments. In Python 3, you can accomplish the same thing by prepending the list of arguments with an asterisk (`*`) and the dictionary of named arguments with two asterisks (`**`).
3. The `+` operator, used here for list concatenation, takes precedence over the `*` operator, so there is no need for extra parentheses around `a_list_of_args + z`.
4. The `2to3` script is smart enough to convert complex `apply()` calls, including calling functions within imported modules.

A.16. intern() [illegible]

In Python 2, you could call the `intern()` function on a string to intern it as a performance optimization. In Python 3, the `intern()` function has been moved to the `sys` module.

Notes	Python 2	Python 3
	`intern(aString)`	`sys.intern(aString)`

A.17. exec statement

Just as the `print` statement became a function in Python 3, so too has the `exec` statement. The `exec()` function takes a string which contains arbitrary Python code and executes it as if it were just another statement or expression. `exec()` is like `eval()`, but even more powerful and evil. The `eval()` function can only evaluate a single expression, but `exec()` can execute multiple statements, imports, function declarations — essentially an entire Python program in a string.

Notes	Python 2	Python 3
①	`exec codeString`	`exec(codeString)`
②	`exec codeString in a_global_namespace`	`exec(codeString, a_global_namespace)`
③	`exec codeString in a_global_namespace, a_local_namespace`	`exec(codeString, a_global_namespace, a_local_namespace)`

1. In the simplest form, the `2to3` script simply encloses the code-as-a-string in parentheses, since `exec()` is now a function instead of a statement.
2. The old `exec` statement could take a namespace, a private environment of globals in which the code-as-a-string would be executed. Python 3 can also do this; just pass the namespace as the second argument to the `exec()` function.
3. Even fancier, the old `exec` statement could also take a local namespace (like the variables defined within a function). In Python 3, the `exec()` function can do that too.

A.18. execfile statement

Like the old `exec` statement, the old `execfile` statement will execute strings as if they were Python code. Where `exec` took a string, `execfile` took a filename. In Python 3, the `execfile` statement has been eliminated. If you really need to take a file of Python code and execute it (but you're not willing to simply import it), you can accomplish the same thing by opening the file, reading its contents, calling the global `compile()` function to force the Python interpreter to compile the code, and then call the new `exec()` function.

Notes	Python 2	Python 3
	`execfile('a_filename')`	`exec(compile(open('a_filename').read(), 'a_filename', 'exec'))`

A.19. repr literals (backticks)

In Python 2, there was a special syntax of wrapping any object in backticks (like `` `x` ``) to get a representation of the object. In Python 3, this capability still exists, but you can no longer use backticks to get it. Instead, use the global `repr()` function.

Notes	Python 2	Python 3
①	`` `x` ``	`repr(x)`
②	``` `'PapayaWhip' + `2`` ```	`repr('PapayaWhip' + repr(2))`

1. Remember, `x` can be anything — a class, a function, a module, a primitive data type, &c. The `repr()` function works on everything.
2. In Python 2, backticks could be nested, leading to this sort of confusing (but valid) expression. The `2to3` tool is smart enough to convert this into nested calls to `repr()`.

A.20. try...except statement

The syntax for catching exceptions has changed slightly between Python 2 and Python 3.

Notes	Python 2	Python 3

①	`try:`
	`    import mymodule`
	`except ImportError, e`
	`    pass`
		`try:`
		`    import mymodule`
		`except ImportError as e:`
		`    pass`

②	`try:`
	`    import mymodule`
	`except (RuntimeError, ImportError), e`
	`    pass`
		`try:`
		`    import mymodule`
		`except (RuntimeError, ImportError) as e:`
		`    pass`

③	`try:`
	`    import mymodule`
	`except ImportError:`
	`    pass`
		no change

④	`try:`
	`    import mymodule`
	`except:`
	`    pass`
		no change

1. Instead of a comma after the exception type, Python 3 uses a new keyword,`as`.
2. The `as` keyword also works for catching multiple types of exceptions at once.
3. If you catch an exception but don’t actually care about accessing theexception object itself, the syntax is identical between Python 2 and Python 3.
4. Similarly, if you use a fallback to catch*all* exceptions, the syntax is identical.

☞ You should never use a fallback to catch*all* exceptions when importing modules (or most other times). Doing so will catch things like `KeyboardInterrupt` (if the user pressed`Ctrl-C` to interrupt the program) and can make it more difficult to debug errors.

A 21 . [illegible]

The syntax for raising your own exceptions has changed slightly between Python 2 and Python 3.

Notes	Python 2	Python 3
①	`raise MyException`	*unchanged*
②	`raise MyException, 'error message'`	`raise MyException('error message')`
③	`raise MyException, 'error message', a_traceback`	`raise MyException('error message').with_traceback(a_traceback)`
④	`raise 'error message'`	*unsupported*

1. In the simplest form, raising an exception without a custom error message, the syntax is unchanged.
2. The change becomes noticeable when you want to raise an exception with a custom error message. Python 2 separated the exception class and the message with a comma; Python 3 passes the error message as a parameter.
3. Python 2 supported a more complex syntax to raise an exception with a custom traceback (stack trace). You can do this in Python 3 as well, but the syntax is quite different.
4. In Python 2, you could raise an exception with no exception class, just an error message. In Python 3, this is no longer possible`2to3` will warn you that it was unable to fix this automatically.

A 2 . [illegible]

In Python 2, generators have a`throw()` method. Calling `a_generator.throw()` raises an exception at the point where the generator was paused, then returns the next value yielded by the generator function. In Python 3, this functionality is still available, but the syntax is slightly different.

Notes	Python 2	Python 3
①	`a_generator.throw(MyException)`	*no change*
②	`a_generator.throw(MyException, 'error message')`	`a_generator.throw(MyException('error message'))`
③	`a_generator.throw('error message')`	*unsupported*

1. In the simplest form, a generator throws an exception without a custom error message. In this case, the syntax has not changed between Python 2 and Python 3.
2. If the generator throws an exception*with* a custom error message, you need to pass the error string to the exception when you create it.
3. Python 2 also supported throwing an exception with*only* a custom error message. Python 3 does not support this, and the`2to3` script will display a warning telling you that you will need to fix this code manually.

A.3. xrange() [illegible]

In Python 2, there were two ways to get a range of numbers: `range()`, which returned a list, and `xrange()`, which returned an iterator. In Python 3, `range()` returns an iterator, and `xrange()` doesn’t exist.

Notes	Python 2	Python 3
①	`xrange(10)`	`range(10)`
②	`a_list = range(10)`	`a_list = list(range(10))`
③	`[i for i in xrange(10)]`	`[i for i in range(10)]`
④	`for i in range(10):`	*no change*
⑤	`sum(range(10))`	*no change*

1. In the simplest case, the `2to3` script will simply convert `xrange()` to `range()`.
2. If your Python 2 code used `range()`, the `2to3` script does not know whether you needed a list, or whether an iterator would do. It errs on the side of caution and coerces the return value into a list by calling the `list()` function.
3. If the `xrange()` function was inside a list comprehension, the `2to3` script is clever enough *not* to wrap the `range()` function with a call to `list()`. The list comprehension will work just fine with the iterator that the `range()` function returns.
4. Similarly, a `for` loop will work just fine with an iterator, so there is no need to change anything here.
5. The `sum()` function will also work with an iterator, so `2to3` makes no changes here either. Like dictionary methods that return views instead of lists, this applies to `min()`, `max()`, `sum()`, `list()`, `tuple()`, `set()`, `sorted()`, `any()`, and `all()`.

A.4. raw_input() [illegible] input() [illegible]

Python 2 had two global functions for asking the user for input on the command line. The first, called `input()`, expected the user to enter a Python expression (and returned the result). The second, called `raw_input()`, just returned whatever the user typed. This was wildly confusing for beginners and widely regarded as a “wart” in the language. Python 3 excises this wart by renaming `raw_input()` to `input()`, so it works the way everyone naively expects it to work.

Notes	Python 2	Python 3
①	`raw_input()`	`input()`
②	`raw_input('prompt')`	`input('prompt')`
③	`input()`	`eval(input())`

1. In the simplest form, `raw_input()` becomes `input()`.
2. In Python 2, the `raw_input()` function could take a prompt as a parameter. This has been retained in Python 3.
3. If you actually need to ask the user for a Python expression to evaluate, use the `input()` function and pass the result to `eval()`.

A.5. func_* [illegible]

In Python 2, code within functions can access special attributes about the function itself. In Python 3, these special function attributes have been renamed for consistency with other attributes.

Notes	Python 2	Python 3
①	`a_function.func_name`	`a_function.__name__`
②	`a_function.func_doc`	`a_function.__doc__`
③	`a_function.func_defaults`	`a_function.__defaults__`
④	`a_function.func_dict`	`a_function.__dict__`
⑤	`a_function.func_closure`	`a_function.__closure__`
⑥	`a_function.func_globals`	`a_function.__globals__`
⑦	`a_function.func_code`	`a_function.__code__`

1. The `__name__` attribute (previously `func_name`) contains the function’s name.
2. The `__doc__` attribute (previously `func_doc`) contains the *docstring* that you defined in the function’s source code.
3. The `__defaults__` attribute (previously `func_defaults`) is a tuple containing default argument values for those arguments that have default values.
4. The `__dict__` attribute (previously `func_dict`) is the namespace supporting arbitrary function attributes.

5. The `__closure__` attribute (previously `func_closure`) is a tuple of cells that contain bindings for the function's free variables.
6. The `__globals__` attribute (previously `func_globals`) is a reference to the global namespace of the module in which the function was defined.
7. The `__code__` attribute (previously `func_code`) is a code object representing the compiled function body.

A [illegible] . x[illegible]

In Python 2, file objects had an `xreadlines()` method which returned an iterator that would read the file one line at a time. This was useful in `for` loops, among other places. In fact, it was so useful, later versions of Python 2 added the capability to file objects themselves.

In Python 3, the `xreadlines()` method no longer exists. `2to3` can fix the simple cases, but some edge cases will require manual intervention.

Notes	Python 2	Python 3
①	`for line in a_file.xreadlines():`	`for line in a_file:`
②	`for line in a_file.xreadlines(5):`	*no change (broken)*

1. If you used to call `xreadlines()` with no arguments, `2to3` will convert it to just the file object. In Python 3, this will accomplish the same thing: read the file one line at a time and execute the body of the `for` loop.
2. If you used to call `xreadlines()` with an argument (the number of lines to read at a time), `2to3` will not fix it, and your code will fail with an `AttributeError: '_io.TextIOWrapper' object has no attribute 'xreadlines'`. You can manually change `xreadlines()` to `readlines()` to get it to work in Python 3. (The `readlines()` method now returns an iterator, so it is just as efficient as `xreadlines()` was in Python 2.)

A 2[illegible] . [illegible]

In Python 2, you could define anonymous `lambda` functions which took multiple parameters by defining the function as taking a tuple with a specific number of items. In effect, Python 2 would "unpack" the tuple into named arguments, which you could then reference (by name) within the `lambda` function. In Python 3, you can still pass a tuple to a `lambda` function, but the Python interpreter will not unpack the tuple into named arguments. Instead, you will need to reference each argument by its positional index.

Notes	Python 2	Python 3
①	`lambda (x,): x + f(x)`	`lambda x1: x1[0] + f(x1[0])`
②	`lambda (x, y): x + f(y)`	`lambda x_y: x_y[0] + f(x_y[1])`
③	`lambda (x, (y, z)): x + y + z`	`lambda x_y_z: x_y_z[0] + x_y_z[1][0] + x_y_z[1][1]`
④	`lambda x, y, z: x + y + z`	*unchanged*

1. If you had defined a `lambda` function that took a tuple of one item, in Python 3 that would become a `lambda` with references to `x1[0]`. The name `x1` is autogenerated by the `2to3` script, based on the named arguments in the original tuple.
2. A `lambda` function with a two-item tuple `(x, y)` gets converted to `x_y` with positional arguments `x_y[0]` and `x_y[1]`.
3. The `2to3` script can even handle `lambda` functions with nested tuples of named arguments. The resulting Python 3 code is a bit unreadable, but it works the same as the old code did in Python 2.
4. You can define `lambda` functions that take multiple arguments. Without parentheses around the arguments, Python 2 just treats it as a `lambda` function with multiple arguments; within the `lambda` function, you simply reference the arguments by name, just like any other function. This syntax still works in Python 3.

A [illegible]

In Python 2, class methods can reference the class object in which they are defined, as well as the method object itself. `im_self` is the class instance object; `im_func` is the function object; `im_class` is the class of `im_self`. In Python 3, these special method attributes have been renamed to follow the naming conventions of other attributes.

Notes	Python 2	Python 3
	`aClassInstance.aClassMethod.im_func`	`aClassInstance.aClassMethod.__func__`
	`aClassInstance.aClassMethod.im_self`	`aClassInstance.aClassMethod.__self__`
	`aClassInstance.aClassMethod.im_class`	`aClassInstance.aClassMethod.__self__.__class__`

A.29. [illegible]

In Python 2, you could build your own classes that could be used in a boolean context. For example, you could instantiate the class and then use the instance in an `if` statement. To do this, you defined a special `__nonzero__()` method which returned `True` or `False`, and it was called whenever the instance was used in a boolean context. In Python 3, you can still do this, but the name of the method has changed to `__bool__()`.

Notes	Python 2	Python 3

①	`class A:`
	`    def __nonzero__(self):`
	`        pass`
		`class A:`
		`    def __bool__(self):`
		`        pass`

②	`class A:`
	`    def __nonzero__(self, x, y):`
	`        pass`
		no change

1. Instead of `__nonzero__()`, Python 3 calls the `__bool__()` method when evaluating an instance in a boolean context.
2. However, if you have a `__nonzero__()` method that takes arguments, the `2to3` tool will assume that you were using it for some other purpose, and it will not make any changes.

A.30 [illegible]

The syntax for defining base 8 (octal) numbers has changed slightly between Python 2 and Python 3.

Notes	Python 2	Python 3
	`x = 0755`	`x = 0o755`

A.31. [illegible]

Due to the integration of the `long` and `int` types, the `sys.maxint` constant is no longer accurate. Because the value may still be useful in determining platform-specific capabilities, it has been retained but renamed as `sys.maxsize`.

Notes	Python 2	Python 3
①	`from sys import maxint`	`from sys import maxsize`
②	`a_function(sys.maxint)`	`a_function(sys.maxsize)`

1. `maxint` becomes `maxsize`.
2. Any usage of `sys.maxint` becomes `sys.maxsize`.

A.32. [illegible]

In Python 2, you could check whether an object was callable (like a function) with the global `callable()` function. In Python 3, this global function has been eliminated. To check whether an object is callable, check for the existence of the `__call__()` special method.

Notes	Python 2	Python 3
	`callable(anything)`	`hasattr(anything, '__call__')`

A.33. [illegible]

In Python 2, the global `zip()` function took any number of sequences and returned a list of tuples. The first tuple contained the first item from each sequence; the second tuple contained the second item from each sequence; and so on. In Python 3, `zip()` returns an iterator instead of a list.

Notes	Python 2	Python 3
①	`zip(a, b, c)`	`list(zip(a, b, c))`
②	`d.join(zip(a, b, c))`	*no change*

1. In the simplest form, you can get the old behavior of the `zip()` function by wrapping the return value in a call to `list()`, which will run through the iterator that `zip()` returns and return a real list of the results.
2. In contexts that already iterate through all the items of a sequence (such as this call to the `join()` method), the iterator that `zip()` returns will work just fine. The `2to3` script is smart enough to detect these cases and make no change to your code.

A.24. StandardError exception

In Python 2, `StandardError` was the base class for all built-in exceptions other than `StopIteration`, `GeneratorExit`, `KeyboardInterrupt`, and `SystemExit`. In Python 3, `StandardError` has been eliminated; use `Exception` instead.

Notes	Python 2	Python 3
	`x = StandardError()`	`x = Exception()`
	`x = StandardError(a, b, c)`	`x = Exception(a, b, c)`

A.25. types module constants

The `types` module contains a variety of constants to help you determine the type of an object. In Python 2, it contained constants for all primitive types like `dict` and `int`. In Python 3, these constants have been eliminated; just use the primitive type name instead.

Notes	Python 2	Python 3
	`types.UnicodeType`	`str`
	`types.StringType`	`bytes`
	`types.DictType`	`dict`
	`types.IntType`	`int`
	`types.LongType`	`int`
	`types.ListType`	`list`
	`types.NoneType`	`type(None)`
	`types.BooleanType`	`bool`
	`types.BufferType`	`memoryview`
	`types.ClassType`	`type`
	`types.ComplexType`	`complex`
	`types.EllipsisType`	`type(Ellipsis)`
	`types.FloatType`	`float`
	`types.ObjectType`	`object`
	`types.NotImplementedType`	`type(NotImplemented)`
	`types.SliceType`	`slice`
	`types.TupleType`	`tuple`
	`types.TypeType`	`type`
	`types.XRangeType`	`range`

☞ `types.StringType` gets mapped to `bytes` instead of `str` because a Python 2 “string” (not a Unicode string, just a regular string) is really just a sequence of bytes in a particular character encoding.

A.26. isinstance() global function

The `isinstance()` function checks whether an object is an instance of a particular class or type. In Python 2, you could pass a tuple of types, and `isinstance()` would return `True` if the object was any of those types. In Python 3, you can still do this, but passing the same type twice is deprecated.

Notes	Python 2	Python 3
	`isinstance(x, (int, float, int))`	`isinstance(x, (int, float))`

A.27. basestring datatype

Python 2 had two string types: Unicode and non-Unicode. But there was also another type, `basestring`. It was an abstract type, a superclass for both the `str` and `unicode` types. It couldn't be called or instantiated directly, but you could pass it to the global `isinstance()` function to check whether an object was either a Unicode or non-Unicode string. In Python 3, there is only one string type, so `basestring` has no reason to exist.

Notes	Python 2	Python 3
	`isinstance(x, basestring)`	`isinstance(x, str)`

A.8. itertools [illegible]

Python 2.3 introduced the `itertools` module, which defined variants of the global `zip()`, `map()`, and `filter()` functions that returned iterators instead of lists. In Python 3, those global functions return iterators, so those functions in the `itertools` module have been eliminated. (There are still lots of useful functions in the `itertools` module, just not these.)

Notes	Python 2	Python 3
①	`itertools.izip(a, b)`	`zip(a, b)`
②	`itertools.imap(a, b)`	`map(a, b)`
③	`itertools.ifilter(a, b)`	`filter(a, b)`
④	`from itertools import imap, izip, foo`	`from itertools import foo`

1. Instead of `itertools.izip()`, just use the global `zip()` function.
2. Instead of `itertools.imap()`, just use `map()`.
3. `itertools.ifilter()` becomes `filter()`.
4. The `itertools` module still exists in Python 3, it just doesn't have the functions that have migrated to the global namespace. The `2to3` script is smart enough to remove the specific imports that no longer exist, while leaving other imports intact.

A.9. sys.exc_type, sys.exc_value, sys.exc_traceback

Python 2 had three variables in the `sys` module that you could access while an exception was being handled: `sys.exc_type`, `sys.exc_value`, `sys.exc_traceback`. (Actually, these date all the way back to Python 1.) Ever since Python 1.5, these variables have been deprecated in favor of `sys.exc_info()`, which is a function that returns a tuple containing those three values. In Python 3, these individual variables have finally gone away; you must use the `sys.exc_info()` function.

Notes	Python 2	Python 3
	`sys.exc_type`	`sys.exc_info()[0]`
	`sys.exc_value`	`sys.exc_info()[1]`
	`sys.exc_traceback`	`sys.exc_info()[2]`

A.40. [illegible]

In Python 2, if you wanted to code a list comprehension that iterated over a tuple, you did not need to put parentheses around the tuple values. In Python 3, explicit parentheses are required.

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

A.41. os.getcwdu() [illegible]

Python 2 had a function named `os.getcwd()`, which returned the current working directory as a (non-Unicode) string. Because modern file systems can handle directory names in any character encoding, Python 2.3 introduced `os.getcwdu()`. The `os.getcwdu()` function returned the current working directory as a Unicode string. In Python 3, there is only one string type (Unicode), so `os.getcwd()` is all you need.

Notes	Python 2	Python 3
	`os.getcwdu()`	`os.getcwd()`

A.42. [illegible]

In Python 2, you could create metaclasses either by defining the`metaclass` argument in the class declaration, or by defining a special class-level `__metaclass__` attribute. In Python 3, the class-level attribute has been eliminated.

Notes	Python 2	Python 3

①	`class C(metaclass=PapayaMeta):`
	`    pass`
		unchanged

②	`class Whip:`
	`    __metaclass__ = PapayaMeta`
		`class Whip(metaclass=PapayaMeta):`
		`    pass`

③	`class C(Whipper, Beater):`
	`    __metaclass__ = PapayaMeta`
		`class C(Whipper, Beater, metaclass=PapayaMeta):`
		`    pass`

1. Declaring the metaclass in the class declaration worked in Python 2, and it still works the same in Python 3.
2. Declaring the metaclass in a class attribute worked in Python 2, but doesn't work in Python 3.
3. The `2to3` script is smart enough to construct a valid class declaration, even if the class is inherited from one or more base classes.

A.43. [illegible]

The rest of the "fixes" listed here aren't really fixes per se. That is, the things they change are matters of style, not substance. They work just as well in Python 3 as they do in Python 2, but the developers of Python have a vested interest in making Python code as uniform as possible. To that end, there is an official Python style guide which outlines — in excruciating detail — all sorts of nitpicky details that you almost certainly don't care about. And given that `2to3` provides such a great infrastructure for converting Python code from one thing to another, the authors took it upon themselves to add a few optional features to improve the readability of your Python programs.

A.43.1. `set()` LITERALS (EXPLICIT)

In Python 2, the only way to define a literal set in your code was to call`set(a_sequence)`. This still works in Python 3, but a clearer way of doing it is to use the new set literal notation: curly braces. This works for everything except empty sets, because dictionaries also use curly braces, so `{}` is an empty dictionary, not an empty set.

☞ The `2to3` script will not fix`set()` literals by default. To enable this fix, specify`-f set_literal` on the command line when you call `2to3`.

Notes	Before	After
	`set([1, 2, 3])`	`{1, 2, 3}`
	`set((1, 2, 3))`	`{1, 2, 3}`
	`set([i for i in a_sequence])`	`{i for i in a_sequence}`

A.43.2. `buffer()` GLOBAL FUNCTION (EXPLICIT)

Python objects implemented in C can export a "buffer interface," which allows other Python code to directly read and write a block of memory. (That is exactly as powerful and scary as it sounds.) In Python 3, `buffer()` has been renamed to`memoryview()`. (It's a little more complicated than that, but you can almost certainly ignore the differences.)

☞ The `2to3` script will not fix the`buffer()` function by default. To enable this fix, specify`-f buffer` on the command line when you call `2to3`.

Notes	Before	After
	`x = buffer(y)`	`x = memoryview(y)`

A.43.3. WHITESPACE AROUND COMMAS (EXPLICIT)

Despite being draconian about whitespace for indenting and outdenting, Python is actually quite liberal about whitespace in other areas. Within lists, tuples, sets, and dictionaries, whitespace can appear before and after commas with no ill effects. However, the Python style guide states that commas should be preceded by zero spaces and followed by one. Although this is purely an aesthetic issue (the code works either way, in both Python 2 and Python 3), the `2to3` script can optionally fix this for you.

☞ The `2to3` script will not fix whitespace around commas by default. To enable this fix, specify `-f wscomma` on the command line when you call `2to3`.

Notes	Before	After
	`a ,b`	`a, b`
	`{a :b}`	`{a: b}`

A.43.4. Common idioms (explicit)

There were a number of common idioms built up in the Python community. Some, like the `while 1:` loop, date back to Python 1. (Python didn't have a true boolean type until version 2.3, so developers used `1` and 0 instead.) Modern Python programmers should train their brains to use modern versions of these idioms instead.

☞ The `2to3` script will not fix common idioms by default. To enable this fix, specify `-f idioms` on the command line when you call `2to3`.

Notes	Before	After

	`while 1:`
	`    do_stuff()`
		`while True:`
		`    do_stuff()`

	`type(x) == T`	`isinstance(x, T)`
	`type(x) is T`	`isinstance(x, T)`

	`a_list = list(a_sequence)`
	`a_list.sort()`
	`do_stuff(a_list)`
		`a_list = sorted(a_sequence)`
		`do_stuff(a_list)`