The key differences between Python 2.7.x and Python 3.x with examples

Jun 1, 2014
by Sebastian Raschka

Many beginning Python users are wondering with which version of Python they should start. My answer to this question is usually something along the lines “just go with the version your favorite tutorial was written in, and check out the differences later on.”

But what if you are starting a new project and have the choice to pick? I would say there is currently no “right” or “wrong” as long as both Python 2.7.x and Python 3.x support the libraries that you are planning to use. However, it is worthwhile to have a look at the major differences between those two most popular versions of Python to avoid common pitfalls when writing the code for either one of them, or if you are planning to port your project.

Sections

The `__future__` module

Python 3.x introduced some Python 2-incompatible keywords and features that can be imported via the in-built `__future__` module in Python 2. It is recommended to use `__future__` imports it if you are planning Python 3.x support for your code. For example, if we want Python 3.x's integer division behavior in Python 2, we can import it via

```
from __future__ import division
```

More features that can be imported from the `__future__` module are listed in the table below:

feature	optional in	mandatory in	effect

nested_scopes	2.1.0b1	2.2	**PEP 227**: *Statically Nested Scopes*
generators	2.2.0a1	2.3	**PEP 255**: *Simple Generators*
division	2.2.0a2	3.0	**PEP 238**: *Changing the Division Operator*
absolute_import	2.5.0a1	3.0	**PEP 328**: *Imports: Multi-Line and Absolute/Relative*
with_statement	2.5.0a1	2.6	**PEP 343**: *The "with" Statement*
print_function	2.6.0a2	3.0	**PEP 3105**: *Make print a function*
unicode_literals	2.6.0a2	3.0	**PEP 3112**: *Bytes literals in Python 3000*

(Source: [https://docs.python.org/2/library/__future__.html] (https://docs.python.org/2/library/__future__.html#module-__future__))

```
from platform import python_version
```

The print function

Very trivial, and the change in the print-syntax is probably the most widely known change, but still it is worth mentioning: Python 2's print statement has been replaced by the `print()` function, meaning that we have to wrap the object that we want to print in parantheses.

Python 2 doesn't have a problem with additional parantheses, but in contrast, Python 3 would raise a `SyntaxError` if we called the print function the Python 2-way without the parentheses.

Python 2

```
print 'Python', python_version()
print 'Hello, World!'
print('Hello, World!')
print "text", ; print 'print more text on the same line'
```

```
Python 2.7.6
Hello, World!
Hello, World!
text print more text on the same line
```

Python 3

```
print('Python', python_version())
print('Hello, World!')
```

```
print("some text,", end="")
print(' print more text on the same line')
```

```
Python 3.4.1
Hello, World!
some text, print more text on the same line
```

```
print 'Hello, World!'
```

```
  File "<ipython-input-3-139a7c5835bd>", line 1
    print 'Hello, World!'
                         ^
SyntaxError: invalid syntax
```

Note:

Printing “Hello, World” above via Python 2 looked quite “normal”. However, if we have multiple objects inside the parantheses, we will create a tuple, since `print` is a “statement” in Python 2, not a function call.

```
print 'Python', python_version()
print('a', 'b')
print 'a', 'b'
```

```
Python 2.7.7
('a', 'b')
a b
```

Integer division

This change is particularly dangerous if you are porting code, or if you are executing Python 3 code in Python 2, since the change in integer-division behavior can often go unnoticed (it doesn’t raise a `SyntaxError`).
So, I still tend to use a `float(3)/2` or `3/2.0` instead of a `3/2` in my Python 3 scripts to save the Python 2 guys some trouble (and vice versa, I recommend a `from __future__ import division` in your Python 2 scripts).

Python 2

```
print 'Python', python_version()
print '3 / 2 =', 3 / 2
print '3 // 2 =', 3 // 2
print '3 / 2.0 =', 3 / 2.0
print '3 // 2.0 =', 3 // 2.0
```

```
Python 2.7.6
3 / 2 = 1
3 // 2 = 1
3 / 2.0 = 1.5
3 // 2.0 = 1.0
```

Python 3

```
print('Python', python_version())
print('3 / 2 =', 3 / 2)
print('3 // 2 =', 3 // 2)
print('3 / 2.0 =', 3 / 2.0)
print('3 // 2.0 =', 3 // 2.0)
```

```
Python 3.4.1
3 / 2 = 1.5
3 // 2 = 1
3 / 2.0 = 1.5
3 // 2.0 = 1.0
```

Unicode

Python 2 has ASCII `str()` types, separate `unicode()`, but no `byte` type.

Now, in Python 3, we finally have Unicode (utf-8) `str`ings, and 2 byte classes: `byte` and `bytearray`s.

Python 2

```
print 'Python', python_version()
```

```
Python 2.7.6
```

```
print type(unicode('this is like a python3 str type'))
```

```
<type 'unicode'>
```

```
print type(b'byte type does not exist')
```

```
<type 'str'>
```

```
print 'they are really' + b' the same'
```

```
they are really the same
```

```
print type(bytearray(b'bytearray oddly does exist though'))
```

```
<type 'bytearray'>
```

Python 3

```
print('Python', python_version())
print('strings are now utf-8 \u03BCnico\u0394é!')
```

```
Python 3.4.1
strings are now utf-8 μnicoΔé!
```

```
print('Python', python_version(), end="")
print(' has', type(b' bytes for storing data'))
```

```
Python 3.4.1 has <class 'bytes'>
```

```
print('and Python', python_version(), end="")
print(' also has', type(bytearray(b'bytearrays')))
```

```
and Python 3.4.1 also has <class 'bytearray'>
```

```
'note that we cannot add a string' + b'bytes for data'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-13-d3e8942ccf81> in <module>()
----> 1 'note that we cannot add a string' + b'bytes for data'


TypeError: Can't convert 'bytes' object to str implicitly
```

xrange

The usage of `xrange()` is very popular in Python 2.x for creating an iterable object, e.g., in a for-loop or list/set-dictionary-comprehension.
The behavior was quite similar to a generator (i.e., “lazy evaluation”), but here the xrange-iterable is not exhaustible - meaning, you could iterate over it infinitely.

Thanks to its “lazy-evaluation”, the advantage of the regular `range()` is that `xrange()` is generally faster if you have to iterate over it only once (e.g., in a for-loop). However, in contrast to 1-time iterations, it is not recommended if you repeat the iteration multiple times, since the generation happens every time from scratch!

In Python 3, the `range()` was implemented like the `xrange()` function so that a dedicated `xrange()` function does not exist anymore (`xrange()` raises a `NameError` in Python 3).

```
import timeit

n = 10000
def test_range(n):
    return for i in range(n):
        pass

def test_xrange(n):
```

```
    for i in xrange(n):
        pass
```

Python 2

```
print 'Python', python_version()

print '\ntiming range()'
%timeit test_range(n)

print '\n\ntiming xrange()'
%timeit test_xrange(n)
```

```
Python 2.7.6

timing range()
1000 loops, best of 3: 433 µs per loop


timing xrange()
1000 loops, best of 3: 350 µs per loop
```

Python 3

```
print('Python', python_version())

print('\ntiming range()')
%timeit test_range(n)
```

```
Python 3.4.1

timing range()
1000 loops, best of 3: 520 µs per loop
```

```
print(xrange(10))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-5-5d8f9b79ea70> in <module>()
```

```
----> 1 print(xrange(10))


NameError: name 'xrange' is not defined
```

The `__contains__` method for `range` objects in Python 3

Another thing worth mentioning is that `range` got a "new" `__contains__` method in Python 3.x (thanks to Yuchen Ying, who pointed this out). The `__contains__` method can speedup "look-ups" in Python 3.x `range` significantly for integer and Boolean types.

```
x = 10000000
```

```
def val_in_range(x, val):
    return val in range(x)
```

```
def val_in_xrange(x, val):
    return val in xrange(x)
```

```
print('Python', python_version())
assert(val_in_range(x, x/2) == True)
assert(val_in_range(x, x//2) == True)
%timeit val_in_range(x, x/2)
%timeit val_in_range(x, x//2)
```

```
Python 3.4.1
1 loops, best of 3: 742 ms per loop
1000000 loops, best of 3: 1.19 µs per loop
```

Based on the `timeit` results above, you see that the execution for the "look up" was about 60,000 faster when it was of an integer type rather than a float. However, since Python 2.x's `range` or `xrange` doesn't have a `__contains__` method, the "look-up speed" wouldn't be that much different for integers or floats:

```
print 'Python', python_version()
assert(val_in_xrange(x, x/2.0) == True)
assert(val_in_xrange(x, x/2) == True)
assert(val_in_range(x, x/2) == True)
assert(val_in_range(x, x//2) == True)
```

```
%timeit val_in_xrange(x, x/2.0)
%timeit val_in_xrange(x, x/2)
%timeit val_in_range(x, x/2.0)
%timeit val_in_range(x, x/2)
```

```
Python 2.7.7
1 loops, best of 3: 285 ms per loop
1 loops, best of 3: 179 ms per loop
1 loops, best of 3: 658 ms per loop
1 loops, best of 3: 556 ms per loop
```

Below the “proofs” that the `__contain__` method wasn’t added to Python 2.x yet:

```
print('Python', python_version())
range.__contains__
```

```
Python 3.4.1



<slot wrapper '__contains__' of 'range' objects>
```

```
print 'Python', python_version()
range.__contains__
```

```
Python 2.7.7



---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

<ipython-input-7-05327350dafb> in <module>()
      1 print 'Python', python_version()
----> 2 range.__contains__


AttributeError: 'builtin_function_or_method' object has no attribute '__contains__'
```

```
print 'Python', python_version()
xrange.__contains__
```

```
Python 2.7.7



---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

<ipython-input-8-7d1a71bfee8e> in <module>()
      1 print 'Python', python_version()
----> 2 xrange.__contains__


AttributeError: type object 'xrange' has no attribute '__contains__'
```

Note about the speed differences in Python 2 and 3

Some people pointed out the speed difference between Python 3's `range()` and Python2's `xrange()`. Since they are implemented the same way one would expect the same speed. However the difference here just comes from the fact that Python 3 generally tends to run slower than Python 2.

```
def test_while():
    i = 0
    while i < 20000:
        i += 1
    return
```

```
print('Python', python_version())
%timeit test_while()
```

```
Python 3.4.1
100 loops, best of 3: 2.68 ms per loop
```

```
print 'Python', python_version()
%timeit test_while()
```

```
Python 2.7.6
1000 loops, best of 3: 1.72 ms per loop
```

Raising exceptions

Where Python 2 accepts both notations, the ‘old’ and the ‘new’ syntax, Python 3 chokes (and raises a `SyntaxError` in turn) if we don’t enclose the exception argument in parentheses:

Python 2

```
print 'Python', python_version()
```

```
Python 2.7.6
```

```
raise IOError, "file error"
```

```
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)

<ipython-input-8-25f049caebb0> in <module>()
----> 1 raise IOError, "file error"


IOError: file error
```

```
raise IOError("file error")
```

```
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)

<ipython-input-9-6f1c43f525b2> in <module>()
----> 1 raise IOError("file error")


IOError: file error
```

Python 3

```
print('Python', python_version())
```

```
Python 3.4.1
```

```
raise IOError, "file error"
```

```
  File "<ipython-input-10-25f049caebb0>", line 1
    raise IOError, "file error"
                 ^
SyntaxError: invalid syntax
```

The proper way to raise an exception in Python 3:

```
print('Python', python_version())
raise IOError("file error")
```

```
Python 3.4.1



---------------------------------------------------------------------------
OSError                                   Traceback (most recent call last)

<ipython-input-11-c350544d15da> in <module>()
      1 print('Python', python_version())
----> 2 raise IOError("file error")


OSError: file error
```

Handling exceptions

Also the handling of exceptions has slightly changed in Python 3. In Python 3 we have to use the “ `as` ” keyword now

Python 2

```
print 'Python', python_version()
try:
    let_us_cause_a_NameError
except NameError, err:
    print err, '--> our error message'
```

```
Python 2.7.6
name 'let_us_cause_a_NameError' is not defined --> our error message
```

Python 3

```
print('Python', python_version())
try:
    let_us_cause_a_NameError
except NameError as err:
    print(err, '--> our error message')
```

```
Python 3.4.1
name 'let_us_cause_a_NameError' is not defined --> our error message
```

The next() function and .next() method

Since `next()` (`.next()`) is such a commonly used function (method), this is another syntax change (or rather change in implementation) that is worth mentioning: where you can use both the function and method syntax in Python 2.7.5, the `next()` function is all that remains in Python 3 (calling the `.next()` method raises an `AttributeError`).

Python 2

```
print 'Python', python_version()

my_generator = (letter for letter in 'abcdefg')

next(my_generator)
my_generator.next()
```

```
Python 2.7.6




'b'
```

Python 3

```
print('Python', python_version())

my_generator = (letter for letter in 'abcdefg')

next(my_generator)
```

```
Python 3.4.1




'a'
```

```
my_generator.next()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

<ipython-input-14-125f388bb61b> in <module>()
----> 1 my_generator.next()


AttributeError: 'generator' object has no attribute 'next'
```

For-loop variables and the global namespace leak

Good news is: In Python 3.x for-loop variables don't leak into the global namespace anymore!

This goes back to a change that was made in Python 3.x and is described in What's New In Python 3.0 as follows:

"List comprehensions no longer support the syntactic form `[... for var in item1, item2, ...]`. Use `[... for var in (item1, item2, ...)]` instead. Also note that list comprehensions have different semantics: they are closer to syntactic sugar for a generator expression inside a `list()` constructor, and in particular the loop control variables are no longer leaked into the surrounding scope."

Python 2

```
print 'Python', python_version()

i = 1
print 'before: i =', i

print 'comprehension: ', [i for i in range(5)]

print 'after: i =', i
```

```
Python 2.7.6
before: i = 1
comprehension:  [0, 1, 2, 3, 4]
after: i = 4
```

Python 3

```
print('Python', python_version())

i = 1
print('before: i =', i)

print('comprehension:', [i for i in range(5)])

print('after: i =', i)
```

```
Python 3.4.1
before: i = 1
comprehension: [0, 1, 2, 3, 4]
after: i = 1
```

Comparing unorderable types

Another nice change in Python 3 is that a `TypeError` is raised as warning if we try to compare unorderable types.

Python 2

```
print 'Python', python_version()
print "[1, 2] > 'foo' = ", [1, 2] > 'foo'
print "(1, 2) > 'foo' = ", (1, 2) > 'foo'
print "[1, 2] > (1, 2) = ", [1, 2] > (1, 2)
```

```
Python 2.7.6
[1, 2] > 'foo' =  False
(1, 2) > 'foo' =  True
[1, 2] > (1, 2) =  False
```

Python 3

```
print('Python', python_version())
print("[1, 2] > 'foo' = ", [1, 2] > 'foo')
print("(1, 2) > 'foo' = ", (1, 2) > 'foo')
print("[1, 2] > (1, 2) = ", [1, 2] > (1, 2))
```

```
Python 3.4.1



---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-16-a9031729f4a0> in <module>()
      1 print('Python', python_version())
----> 2 print("[1, 2] > 'foo' = ", [1, 2] > 'foo')
      3 print("(1, 2) > 'foo' = ", (1, 2) > 'foo')
      4 print("[1, 2] > (1, 2) = ", [1, 2] > (1, 2))


TypeError: unorderable types: list() > str()
```

Parsing user inputs via input()

Fortunately, the `input()` function was fixed in Python 3 so that it always stores the user inputs as `str` objects. In order to avoid the dangerous behavior in Python 2 to read in other types than `strings`, we have to use `raw_input()` instead.

Python 2

```
Python 2.7.6
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> my_input = input('enter a number: ')

enter a number: 123

>>> type(my_input)
<type 'int'>

>>> my_input = raw_input('enter a number: ')

enter a number: 123

>>> type(my_input)
<type 'str'>
```

Python 3

```
Python 3.4.1
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> my_input = input('enter a number: ')

enter a number: 123

>>> type(my_input)
<class 'str'>
```

Returning iterable objects instead of lists

As we have already seen in the `xrange` section, some functions and methods return iterable objects in Python 3 now - instead of lists in Python 2.

Since we usually iterate over those only once anyway, I think this change makes a lot of sense to save memory. However, it is also possible - in contrast to generators - to iterate over those multiple times if needed, it is only not so efficient.

And for those cases where we really need the `list` -objects, we can simply convert the iterable object into a `list` via the `list()` function.

Python 2

```
print 'Python', python_version()

print range(3)
print type(range(3))
```

```
Python 2.7.6
[0, 1, 2]
<type 'list'>
```

Python 3

```
print('Python', python_version())

print(range(3))
print(type(range(3)))
print(list(range(3)))
```

```
Python 3.4.1
range(0, 3)
<class 'range'>
[0, 1, 2]
```

Some more commonly used functions and methods that don't return lists anymore in Python 3:

- `zip()`
- `map()`
- `filter()`
- dictionary's `.keys()` method
- dictionary's `.values()` method

- dictionary's `.items()` method

Banker's Rounding

Python 3 adopted the now standard way of rounding decimals when it results in a tie (.5) at the last significant digits. Now, in Python 3, decimals are rounded to the nearest even number. Although it's an inconvenience for code portability, it's supposedly a better way of rounding compared to rounding up as it avoids the bias towards large numbers. For more information, see the excellent Wikipedia articles and paragraphs:

- https://en.wikipedia.org/wiki/Rounding#Round_half_to_even
- https://en.wikipedia.org/wiki/IEEE_floating_point#Roundings_to_nearest

Python 2

```
print 'Python', python_version()
```

```
Python 2.7.12
```

```
round(15.5)
```

```
16.0
```

```
round(16.5)
```

```
17.0
```

Python 3

```
print('Python', python_version())
```

```
Python 3.5.1
```

```
round(15.5)
```

```
```

```
16
```

```
round(16.5)
```

```
16
```

More articles about Python 2 and Python 3

Here is a list of some good articles concerning Python 2 and 3 that I would recommend as a follow-up.

// Porting to Python 3

- Should I use Python 2 or Python 3 for my development activity?
- What's New In Python 3.0
- Porting to Python 3
- Porting Python 2 Code to Python 3
- How keep Python 3 moving forward

// Pro and anti Python 3

- 10 awesome features of Python that you can't use because you refuse to upgrade to Python 3
- Everything you did not want to know about Unicode in Python 3
- Python 3 is killing Python
- Python 3 can revive Python
- Python 3 is fine