Incompatibilities moving from Python 2 to Python 3 All Versions

Unlike most languages, Python supports two major versions. Since 2008 when Python 3 was released, many have made the transition, while many have not. In order to understand both, this section covers the important differences between Python 2 and Python 3.

Examples

Integer Division

The standard **division symbol** (`/`) operates differently in Python 3 and Python 2 when applied to integers.

When dividing an integer by another integer in Python 3, the division operation `x / y` represents a **true division** and produces a floating point result. In Python 3 the `/` maps to the `__truediv__` function. Meanwhile, the same operation in Python 2 represents a **classic division** that rounds the result down toward negative infinity (also known as taking the *floor*).

For example:

Code	Python 2 output	Python 3 output
`3 / 2`	1	1.5
`2 / 3`	0	0.6666666666666666
`-3 / 2`	-2	-1.5

The rounding-towards-zero behavior was deprecated in Python 2.2 , but remains in Python 2.7 for the sake of backward compatibility and was removed in Python 3.

Note: To get a *float* result in Python 2 (without floor rounding) we can specify one of the operands with the decimal point. The above example of `2/3` which gives `0` in Python 2 shall be used as `2 / 3.0` or `2.0 / 3` or `2.0/3.0` to get `0.6666666666666666`

Code	Python 2 output	Python 3 output
`3.0 / 2.0`	1.5	1.5
`2 / 3.0`	0.6666666666666666	0.6666666666666666
`-3.0 / 2`	-1.5	-1.5

There is also the **floor division operator** (`//`), which works the same way in both versions: it rounds down to the nearest integer (i.e., works the same as the `/` operator in Python 2). In Python 3 the `//` operator maps to `__floordiv__` .

Code	Python 2 output	Python 3 output
`3 // 2`	1	1
`2 // 3`	0	0
`-3 // 2`	-2	-2
`3.0 // 2.0`	1	1
`2.0 // 3`	0	0
`-3 // 2.0`	-2	-2

One can explicitly enforce true division or floor division using native functions in the `operator` module:

```
from operator import truediv, floordiv
assert truediv(10, 8) == 1.25           # equivalent to `/` in Python 3
assert floordiv(10, 8) == 1             # equivalent to `/` in Python 2  or `//`
```

While clear and explicit, using operator functions for every division can be tedious. Changing the behavior of the `/` operator will often be preferred. A common practice is to eliminate typical division behavior by adding `from __future__ import division` as the first statement in each module:

```
# needs to be the first statement in a module
from __future__ import division
```

Code	Python 2 output	Python 3 output
`3 / 2`	1.5	1.5

Code	Python 2 output	Python 3 output
2 / 3	0.6666666666666666	0.6666666666666666
-3 / 2	-1.5	-1.5

`from __future__ import division` guarantees that the `/` operator represents true division and only within the modules that contain the `__future__` import, so there are no compelling reasons for not enabling it in all new modules.

Note : Some other programming languages use *rounding toward zero* (truncation) rather than *rounding down toward negative infinity* as Python does (i.e. in those languages `-3 / 2 == -1`). This behavior may create confusion when porting or comparing code.

Note on float operands : As an alternative to `from __future__ import division` , one could use the usual division symbol `/` and ensure that at least one of the operands is a float: `3 / 2.0 == 1.5` . However, this can be considered bad practice. It is just too easy to write `average = sum(items) / len(items)` and forget to cast one of the arguments to float. Moreover, such cases may frequently evade notice during testing, e.g., if you test on an array containing `float` s but receive an array of `int` s in production. Additionally, if the same code is used in Python 3, programs that expect `3 / 2 == 1` to be True will not work correctly.

See PEP 238 for more detailed rationale why the division operator was changed in Python 3 and why old-style division should be avoided.

See the *Simple Math* topic for more about division.

Unpacking Iterables

Python 3.x ≥3.0

In Python 3, you can unpack an iterable without knowing the exact number of items in it, and even have a variable hold the end of the iterable. For that, you provide a variable that may collect a list of values. This is done by placing an asterisk before the name. For example, unpacking a `list` :

```
first, second, *tail, last = [1, 2, 3, 4, 5]
print(first)
# Out: 1
print(second)
# Out: 2
print(tail)
# Out: [3, 4]
print(last)
# Out: 5
```

Note : When using the `*variable` syntax, the `variable` will always be a list, even if the original type wasn't a list. It may contain zero or more elements depending on the number of elements in the original list.

```
first, second, *tail, last = [1, 2, 3, 4]
print(tail)
# Out: [3]

first, second, *tail, last = [1, 2, 3]
print(tail)
# Out: []
print(last)
# Out: 3
```

Similarly, unpacking a `str` :

```
begin, *tail = "Hello"
print(begin)
# Out: 'H'
print(tail)
# Out: ['e', 'l', 'l', 'o']
```

Example of unpacking a `date` ; `_` is used in this example as a throwaway variable (we are interested only in `year` value):

```
person = ('John', 'Doe', (10, 16, 2016))
*_, (*_, year_of_birth) = person
print(year_of_birth)
# Out: 2016
```

It is worth mentioning that, since `*` eats up a variable number of items in the given sequence, assignments do not allow two `*` s for the same expression — it wouldn't know how many elements went into the first unpacking, and how many in the second:

```
*head, *tail = [1, 2]
# Out: SyntaxError: two starred expressions in assignment
```

Python 3.x ≥3.5

So far we have discussed unpacking in assignments. * and ** were extended in Python 3.5 . It's now possible to have several unpacking operations in one expression:

```
{*range(4), 4, *(5, 6, 7)}
# Out: {0, 1, 2, 3, 4, 5, 6, 7}
```

Python 2.x ≥2.0

It is also possible to unpack an iterable into function arguments:

```
iterable = [1, 2, 3, 4, 5]
print(iterable)
# Out: [1, 2, 3, 4, 5]
print(*iterable)
# Out: 1 2 3 4 5
```

Python 3.x ≥3.5

Unpacking a dictionary uses two adjacent stars ** (PEP 448):

```
tail = {'y': 2, 'z': 3}
{'x': 1, **tail}
 # Out: {'x': 1, 'y': 2, 'z': 3}
```

This allows for both overriding old values and merging dictionaries.

```
dict1 = {'x': 1, 'y': 1}
dict2 = {'y': 2, 'z': 3}
{**dict1, **dict2}
# Out: {'x': 1, 'y': 2, 'z': 3}
```

Python 3.x ≥3.0

Python 3 removed tuple unpacking in functions. Hence the following doesn't work in Python 3

```
# Works in Python 2, but syntax error in Python 3:
map(lambda (x, y): x + y, zip(range(5), range(5)))
# Same is true for non-lambdas:
def example((x, y)):
    pass

# Works in both Python 2 and Python 3:
map(lambda x: x[0] + x[1], zip(range(5), range(5)))
# And non-lambdas, too:
def working_example(x_y):
    x, y = x_y
    pass
```

See PEP 3113 for detailed rationale.

Strings: Bytes versus Unicode

Python 2.x ≤2.7

In Python 2 there are two variants of string: those made of bytes with type (str) and those made of text with type (unicode).

In Python 2, an object of type str is always a byte sequence, but is commonly used for both text and binary data.

A string literal is interpreted as a byte string.

```
s = 'Cafe'    # type(s) == str
```

There are two exceptions: You can define a *Unicode (text) literal* explicitly by prefixing the literal with u :

```
s = u'Café'   # type(s) == unicode
b = 'Lorem ipsum'  # type(b) == str
```

Alternatively, you can specify that a whole module's string literals should create Unicode (text) literals:

```
from __future__ import unicode_literals

s = 'Café'   # type(s) == unicode
```

```
b = 'Lorem ipsum'  # type(b) == unicode
```

In order to check whether your variable is a string (either Unicode or a byte string), you can use:

```
isinstance(s, basestring)
```

Python 3.x ≥3.0

In Python 3, the `str` type is a Unicode text type.

```
s = 'Cafe'           # type(s) == str
s = 'Café'           # type(s) == str (note the accented trailing e)
```

Additionally, Python 3 added a `bytes` object , suitable for binary "blobs" or writing to encoding-independent files. To create a bytes object, you can prefix `b` to a string literal or call the string's `encode` method:

```
# Or, if you really need a byte string:
s = b'Cafe'          # type(s) == bytes
s = 'Café'.encode()  # type(s) == bytes
```

To test whether a value is a string, use:

```
isinstance(s, str)
```

Python 3.x ≥3.3

It is also possible to prefix string literals with a `u` prefix to ease compatibility between Python 2 and Python 3 code bases. Since, in Python 3, all strings are Unicode by default, prepending a string literal with `u` has no effect:

```
u'Cafe' == 'Cafe'
```

Python 2's raw Unicode string prefix `ur` is not supported, however:

```
>>> ur'Café'
  File "<stdin>", line 1
    ur'Café'
           ^
SyntaxError: invalid syntax
```

Note that you must `encode` a Python 3 text (`str`) object to convert it into a `bytes` representation of that text. The default encoding of this method is `UTF-8` .

You can use `decode` to ask a `bytes` object for what Unicode text it represents:

```
>>> b.decode()
'Café'
```

Python 2.x ≥2.6

While the `bytes` type exists in both Python 2 and 3, the `unicode` type only exists in Python 2. To use Python 3's implicit Unicode strings in Python 2, add the following to the top of your code file:

```
from __future__ import unicode_literals
print(repr("hi"))
# u'hi'
```

Python 3.x ≥3.0

Another important difference is that indexing bytes in Python 3 results in an `int` output like so:

```
b"abc"[0] == 97
```

Whilst slicing in a size of one results in a length 1 bytes object:

```
b"abc"[0:1] == b"a"
```

In addition, Python 3 fixes some unusual behavior with unicode, i.e. reversing byte strings in Python 2. For example, the following issue is resolved:

```
# -*- coding: utf8 -*-
print("Hi, my name is łukasz Langa.")
print(u"Hi, my name is łukasz Langa."[::-1])
print("Hi, my name is łukasz Langa."[::-1])

# Output in Python 2
# Hi, my name is łukasz Langa.
```

```
# .agnaL zsakuŁ si eman ym ,iH
# .agnaL zsaku�� si eman ym ,iH

# Output in Python 3
# Hi, my name is łukasz Langa.
# .agnaL zsakuŁ si eman ym ,iH
# .agnaL zsakuŁ si eman ym ,iH
```

Print statement vs. Print function

In Python 2, `print` is a statement:

Python 2.x ≤2.7

```
print "Hello World"
print                          # print a newline
print "No newline",            # add trailing comma to remove newline
print >>sys.stderr, "Error"    # print to stderr
print("hello")                 # print "hello", since ("hello") == "hello"
print()                        # print an empty tuple "()"
print 1, 2, 3                  # print space-separated arguments: "1 2 3"
print(1, 2, 3)                 # print tuple "(1, 2, 3)"
```

In Python 3, `print()` is a function, with keyword arguments for common uses:

Python 3.x ≥3.0

```
print "Hello World"              # SyntaxError
print("Hello World")
print()                          # print a newline (must use parentheses)
print("No newline", end="")      # end specifies what to append (defaults to newline)
print("Error", file=sys.stderr)  # file specifies the output buffer
print("Comma", "separated", "output", sep=",")  # sep specifies the separator
print("A", "B", "C", sep="")     # null string for sep: prints as ABC
print("Flush this", flush=True)  # flush the output buffer, added in Python 3.3
print(1, 2, 3)                   # print space-separated arguments: "1 2 3"
print((1, 2, 3))                 # print tuple "(1, 2, 3)"
```

The print function has the following parameters:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

`sep` is what separates the objects you pass to print. For example:

```
print('foo', 'bar', sep='~') # out: foo~bar
print('foo', 'bar', sep='.') # out: foo.bar
```

`end` is what the end of the print statement is followed by. For example:

```
print('foo', 'bar', end='!') # out: foo bar!
```

Printing again following a non-newline ending print statement *will* print to the same line:

```
print('foo', end='~')
print('bar')
# out: foo~bar
```

Note : For future compatibility, `print` *function* is also available in Python 2.6 onwards; however it cannot be used unless parsing of the `print` *statement* is disabled with

```
from __future__ import print_function
```

This function has exactly same format as Python 3's, except that it lacks the `flush` parameter.

See PEP 3105 for rationale.

Differences between range and xrange functions

In Python 2, `range` function returns a list while `xrange` creates a special `xrange` object, which is an immutable sequence, which unlike other built-in sequence types, doesn't support slicing and has neither `index` nor `count` methods:

Python 2.x ≥2.3

```
print(range(1, 10))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]

print(xrange(1, 10))
# Out: xrange(1, 10)

print(isinstance(xrange(1, 10), xrange))
# Out: True
```

In Python 3, `xrange` was expanded to `range`, which thus now creates a `range` object. There is no `xrange` type:

Python 3.x ≥3.0

```
print(range(1, 10))
# Out: range(1, 10)

print(isinstance(range(1, 10), range))
# Out: True

# print(xrange(1, 10))
# The output will be:
#Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
#NameError: name 'xrange' is not defined
```

Additionally, since Python 3.2, `range` also supports slicing, `index` and `count`:

```
print(range(1, 10)[3:7])
# Out: range(3, 7)
print(range(1, 10).count(5))
# Out: 1
print(range(1, 10).index(7))
# Out: 6
```

The advantage of using a special sequence type instead of a list is that the interpreter does not have to allocate memory for a list and populate it:

Python 2.x ≥2.3

```
# range(10000000000000000)
# The output would be:
# Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
# MemoryError

print(xrange(100000000000000000))
# Out: xrange(100000000000000000)
```

Since the latter behaviour is generally desired, the former one was removed in Python 3. If you still want to have a list in Python 3, you can simply use the `list()` constructor on a `range` object:

Python 3.x ≥3.0

```
print(list(range(1, 10)))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Compatibility

In order to maintain compatibility between both Python 2.x and Python 3.x versions, you can use the `builtins` module from the external package `future` to achieve both *forward-compatiblity* and *backward-compatiblity*:

Python 2.x ≥2.0

```
#forward-compatible
from builtins import range

for i in range(10**8):
    pass
```

Python 3.x ≥3.0

```
#backward-compatible
from past.builtins import xrange

for i in xrange(10**8):
    pass
```

The `range` in `future` library supports slicing, `index` and `count` in all Python versions, just like the built-in method on Python 3.2+.

Raising and handling Exceptions

This is the Python 2 syntax, note the commas `,` on the `raise` and `except` lines:

Python 2.x ≥2.3

```
try:
    raise IOError, "input/output error"
except IOError, exc:
    print exc
```

In Python 3, the `,` syntax is dropped and replaced by parenthesis and the `as` keyword:

```
try:
    raise IOError("input/output error")
except IOError as exc:
    print(exc)
```

For backwards compatibility, the Python 3 syntax is also available in Python 2.6 onwards, so it should be used for all new code that does not need to be compatible with previous versions.

Python 3.x ≥3.0

Python 3 also adds exception chaining , wherein you can signal that some other exception was the *cause* for this exception. For example

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from e
```

The exception raised in the `except` statement is of type `DatabaseError` , but the original exception is marked as the `__cause__` attribute of that exception. When the traceback is displayed, the original exception will also be displayed in the traceback:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

If you throw in an `except` block *without* explicit chaining:

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}')
```

The traceback is

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

Python 2.x ≥2.0

Neither one is supported in Python 2.x; the original exception and its traceback will be lost if another exception is raised in the except block. The following code can be used for compatibility:

```
import sys
import traceback

try:
    funcWithError()
except:
    sys_vers = getattr(sys, 'version_info', (0,))
```

```
    if sys_vers < (3, 0):
        traceback.print_exc()
    raise Exception("new exception")
```

Python 3.x ≥3.0

To "forget" the previously thrown exception, use `raise from None`

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from None
```

Now the traceback would simply be

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

Or in order to make it compatible with both Python 2 and 3 you may use the six package like so:

```
import six
try:
    file = open('database.db')
except FileNotFoundError as e:
    six.raise_from(DatabaseError('Cannot open {}'), None)
```

Leaked variables in list comprehension

Python 2.x ≥2.3

```
x = 'hello world!'
vowels = [x for x in 'AEIOU']

print (vowels)
# Out: ['A', 'E', 'I', 'O', 'U']
print(x)
# Out: 'U'
```

Python 3.x ≥3.0

```
x = 'hello world!'
vowels = [x for x in 'AEIOU']

print (vowels)
# Out: ['A', 'E', 'I', 'O', 'U']
print(x)
# Out: 'hello world!'
```

As can be seen from the example, for `Python2` the value of `x` was leaked: it masked `hello world!` and printed out `U`, since this was the last value of `x` when the loop ended.

However, in `Python3` `x` prints the orignally defined `hello world!`, since the local variable from the list comprehension does not mask variables from the surrounding scope.

Additionally, neither generator expressions (available in Python since 2.5) nor dictionary or set comprehensions (which were backported to Python 2.7 from Python 3) leak variables in Python 2.

Note that in both Python 2 and Python 3, variables will leak into the surrounding scope when using a for loop:

```
x = 'hello world!'
vowels = []
for x in 'AEIOU':
    vowels.append(x)
print(x)
# Out: 'U'
```

True, False and None

In Python 2, `True`, `False` and `None` are built-in constants. Which means it's possible to reassign them.

Python 2.x ≥2.0

```
True, False = False, True
True   # False
False  # True
```

You can't do this with `None` since Python 2.4.

Python 2.x ≥2.4

```
None = None  # SyntaxError: cannot assign to None
```

In Python 3, `True`, `False`, and `None` are now keywords.

Python 3.x ≥3.0

```
True, False = False, True  # SyntaxError: can't assign to keyword

None = None  # SyntaxError: can't assign to keyword
```

Comparison of different types

Python 2.x ≥2.3

Objects of different types can be compared. The results are arbitrary, but consistent. They are ordered such that `None` is less than anything else, numeric types are smaller than non-numeric types, and everything else is ordered lexicographically by type. Thus, an `int` is less than a `str` and a `tuple` is greater than a `list`:

```
[1, 2] > 'foo'
# Out: False
(1, 2) > 'foo'
# Out: True
[1, 2] > (1, 2)
# Out: False
100 < [1, 'x'] < 'xyz' < (1, 'x')
# Out: True
```

This was originally done so a list of mixed types could be sorted and objects would be grouped together by type:

```
l = [7, 'x', (1, 2), [5, 6], 5, 8.0, 'y', 1.2, [7, 8], 'z']
sorted(l)
# Out: [1.2, 5, 7, 8.0, [5, 6], [7, 8], 'x', 'y', 'z', (1, 2)]
```

Python 3.x ≥3.0

An exception is raised when comparing different (non-numeric) types:

```
1 < 1.5
# Out: True

[1, 2] > 'foo'
# TypeError: unorderable types: list() > str()
(1, 2) > 'foo'
# TypeError: unorderable types: tuple() > str()
[1, 2] > (1, 2)
# TypeError: unorderable types: list() > tuple()
```

To sort mixed lists in Python 3 by types and to achieve compatibility between versions, you have to provide a key to the sorted function:

```
>>> list = [1, 'hello', [3, 4], {'python': 2}, 'stackoverflow', 8, {'python': 3}, [5, 6]]
>>> sorted(list, key=lambda x: str(x))
# Out: [1, 8, [3, 4], [5, 6], 'hello', 'stackoverflow', {'python': 2}, {'python': 3}]
```

`lambda x: str(x)` is converting each item to string only for the function. It then sees the string representation starting with either `[`, `'`, `{` or `0-9` and it's able to sort those (and all the following characters).

User Input

In Python 2, user input is accepted using the `raw_input` function,

Python 2.x ≥2.3

```
user_input = raw_input()
```

While in Python 3 user input is accepted using the `input` function.

Python 3.x ≥3.0

```
user_input = input()
```

In Python 2, the `input` function will accept input and *interpret* it. While this can be useful, it has several security considerations and was removed in Python 3. To access the same functionality, `eval(input())` can be used.

To keep a script portable across the two versions, you can put the code below near the top of your Python script:

```
try:
    input = raw_input
except NameError:
    pass
```

Removed operators <> and ``, synonymous with != and repr()

In Python 2, `<>` is a synonym for `!=` ; likewise, `` `foo` `` is a synonym for `repr(foo)` .

Python 2.x ≤2.7

```
>>> 1 <> 2
True
>>> 1 <> 1
False
>>> foo = 'hello world'
>>> repr(foo)
"'hello world'"
>>> `foo`
"'hello world'"
```

Python 3.x ≥3.0

```
>>> 1 <> 2
  File "<stdin>", line 1
    1 <> 2
       ^
SyntaxError: invalid syntax
>>> `foo`
  File "<stdin>", line 1
    `foo`
    ^
SyntaxError: invalid syntax
```

.next() method on iterators renamed

In Python 2, an iterator can be traversed by using a method called `next` on the iterator itself:

Python 2.x ≥2.3

```
g = (i for i in range(0, 3))
g.next()  # Yields 0
g.next()  # Yields 1
g.next()  # Yields 2
```

In Python 3 the `.next` method has been renamed to `.__next__` , acknowledging its “magic” role, so calling `.next` will raise an `AttributeError` . The correct way to access this functionality in both Python 2 and Python 3 is to call the `next` *function* with the iterator as an argument.

Python 3.x ≥3.0

```
g = (i for i in range(0, 3))
next(g)  # Yields 0
next(g)  # Yields 1
next(g)  # Yields 2
```

This code is portable across versions from 2.6 through to current releases.

filter(), map() and zip() return iterators instead of sequences

Python 2.x ≤2.7

In Python 2 filter , map and zip built-in functions return a sequence. map and zip always return a list while with filter the return type depends on the type of given parameter:

```
>>> s = filter(lambda x: x.isalpha(), 'a1b2c3')
>>> s
'abc'
>>> s = map(lambda x: x * x, [0, 1, 2])
>>> s
[0, 1, 4]
>>> s = zip([0, 1, 2], [3, 4, 5])
>>> s
[(0, 3), (1, 4), (2, 5)]
```

Python 3.x ≥3.0

In Python 3 filter , map and zip return iterator instead:

```
>>> it = filter(lambda x: x.isalpha(), 'a1b2c3')
>>> it
<filter object at 0x00000098A55C2518>
>>> ''.join(it)
'abc'
>>> it = map(lambda x: x * x, [0, 1, 2])
>>> it
<map object at 0x000000E0763C2D30>
>>> list(it)
[0, 1, 4]
>>> it = zip([0, 1, 2], [3, 4, 5])
>>> it
<zip object at 0x000000E0763C52C8>
>>> list(it)
[(0, 3), (1, 4), (2, 5)]
```

Since Python 2 itertools.izip is equivalent of Python 3 zip izip has been removed on Python 3.

Renamed modules

A few modules in the standard library have been renamed:

Old name	New name
_winreg	winreg
ConfigParser	configparser
copy_reg	copyreg
Queue	queue
SocketServer	socketserver
_markupbase	markupbase
repr	reprlib
test.test_support	test.support
Tkinter	tkinter
tkFileDialog	tkinter.filedialog
urllib / urllib2	urllib, urllib.parse, urllib.error, urllib.response, urllib.request, urllib.robotparser

Some modules have even been converted from files to libraries. Take tkinter and urllib from above as an example.

Compatibility

When maintaining compatibility between both Python 2.x and 3.x versions, you can use the future external package to enable importing top-level standard library packages with Python 3.x names on Python 2.x versions.

long vs. int

In Python 2, any integer larger than a C ssize_t would be converted into the long data type, indicated by an L suffix on the literal. For example, on a 32 bit build of Python:

Python 2.x ≤2.7

```
>>> 2**31
2147483648L
>>> type(2**31)
<type 'long'>
>>> 2**30
1073741824
>>> type(2**30)
<type 'int'>
>>> 2**31 - 1  # 2**31 is long and long - int is long
2147483647L
```

However, in Python 3, the long data type was removed; no matter how big the integer is, it will be an int .

Python 3.x ≥3.0

```
2**1024
# Output: 17976931348623159077293051907890247336179769789423065727343008115773267580550096313
print(-(2**1024))
# Output: -179769313486231590772930519078902473361797697894230657273430081157732675805500963
type(2**1024)
# Output: <class 'int'>
```

All classes are "new-style classes" in Python 3.

In Python 3.x all classes are *new-style classes* ; when defining a new class python implicitly makes it inherit from object . As such, specifying object in a class definition is a completely optional:

Python 3.x ≥3.0

```
class X: pass
class Y(object): pass
```

Both of these classes now contain object in their mro (method resolution order):

Python 3.x ≥3.0

```
>>> X.__mro__
(__main__.X, object)

>>> Y.__mro__
(__main__.Y, object)
```

In Python 2.x classes are, by default, old-style classes; they do not implicitly inherit from object . This causes the semantics of classes to differ depending on if we explicitly add object as a base class :

Python 2.x ≥2.3

```
class X: pass
class Y(object): pass
```

In this case, if we try to print the __mro__ of Y , similar output as that in the Python 3.x case will appear:

Python 2.x ≥2.3

```
>>> Y.__mro__
(<class '__main__.Y'>, <type 'object'>)
```

This happens because we explicitly made Y inherit from object when defining it: class Y(object): pass . For class X which does *not* inherit from object the __mro__ attribute does not exist, trying to access it results in an AttributeError .

In order to **ensure compatibility** between both versions of Python, classes can be defined with object as a base class:

```
class mycls(object):
    """I am fully compatible with Python 2/3"""
```

Alternatively, if the `__metaclass__` variable is set to `type` at global scope, all subsequently defined classes in a given module are implicitly new-style without needing to explicitly inherit from `object` :

```
__metaclass__ = type

class mycls:
    """I am also fully compatible with Python 2/3"""
```

---

map()

`map()` is a builtin that is useful for applying a function to elements of an iterable. In Python 2, `map` returns a list. In Python 3, `map` returns a *map object* , which is a generator.

```
# Python 2.X
>>> map(str, [1, 2, 3, 4, 5])
['1', '2', '3', '4', '5']
>>> type(_)
>>> <class 'list'>

# Python 3.X
>>> map(str, [1, 2, 3, 4, 5])
<map object at 0x*>
>>> type(_)
<class 'map'>

# We need to apply map again because we "consumed" the previous map....
>>> map(str, [1, 2, 3, 4, 5])
>>> list(_)
['1', '2', '3', '4', '5']
```

In Python 2, you can pass `None` to serve as an identity function. This no longer works in Python 3.

Python 2.x ≥2.3

```
>>> map(None, [0, 1, 2, 3, 0, 4])
[0, 1, 2, 3, 0, 4]
```

Python 3.x ≥3.0

```
>>> list(map(None, [0, 1, 2, 3, 0, 5]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not callable
```

Moreover, when passing more than one iterable as argument in Python 2, `map` pads the shorter iterables with `None` (similar to `itertools.izip_longest`). In Python 3, iteration stops after the shortest iterable.

In Python 2:

Python 2.x ≥2.3

```
>>> map(None, [1, 2, 3], [1, 2], [1, 2, 3, 4, 5])
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

In Python 3:

Python 3.x ≥3.0

```
>>> list(map(lambda x, y, z: (x, y, z), [1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2)]

# to obtain the same padding as in Python 2 use zip_longest from itertools
>>> import itertools
>>> list(itertools.zip_longest([1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

Note : instead of `map` consider using list comprehensions, which are Python 2/3 compatible. Replacing `map(str, [1, 2, 3, 4, 5])` :

```
>>> [str(i) for i in [1, 2, 3, 4, 5]]
['1', '2', '3', '4', '5']
```

Reduce is no longer a built-in

In Python 2, `reduce` is available either as a built-in function or from the `functools` package (version 2.6 onwards), whereas in Python 3 `reduce` is available only from `functools` . However the syntax for `reduce` in both Python2 and Python3 is the same and is `reduce(function_to_reduce, list_to_reduce)` .

As an example, let us consider reducing a list to a single value by dividing each of the adjacent numbers. Here we use `truediv` function from the `operator` library.

In Python 2.x it is as simple as:

Python 2.x ≥2.3

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator
>>> reduce(operator.truediv, my_list)
0.008333333333333333
```

In Python 3.x the example becomes a bit more complicated:

Python 3.x ≥3.0

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator, functools
>>> functools.reduce(operator.truediv, my_list)
0.008333333333333333
```

We can also use `from functools import reduce` to avoid calling `reduce` with the namespace name.

Absolute/Relative Imports

In Python 3, PEP 404 changes the way imports work from Python 2. *Implicit relative* imports are no longer allowed in packages and `from ... import *` imports are only allowed in module level code.

To achieve Python 3 behavior in Python 2:

- the absolute imports feature can be enabled with `from __future__ import absolute_import`
- *explicit relative* imports are encouraged in place of *implicit relative* imports

---

For clarification, in Python 2, a module can import the contents of another module located in the same directory as follows:

```
import foo
```

Notice the location of `foo` is ambiguous from the import statement alone. This type of implicit relative import is thus discouraged in favor of explicit relative imports , which look like the following:

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
from ...package import bar
from ...sys import path
```

The dot `.` allows an explicit declaration of the module location within the directory tree.

Improvements requested:

Branching based on Python version

If you need to execute different code for different Python versions, then do consider that in time be a Python 4 with which your code could be compatible. Thus, for version tests, test whether the current version is Python 2, or greater than Python 2. It is common to define two variables `PY2` and `PY3` where `PY2` is true for Pythons 2 (and 1 and 0.x), and `PY3` for versions 3.0 and up:

```
import sys
PY2 = sys.version_info < (3,)
PY3 = not PY2
```

Some guides advocate the way ~~`PY3 = sys.version_info[0] == 3`~~, but it will break whenever Python 4 is released.

Then you can use for example

```
if PY2:
    text_type = unicode
else:
    text_type = str
```

You shouldn't branch everywhere but instead refactor your code so that a whole function definition is changed based on whether 2 or 3 is in use. Thus do not write

```
def my_func():
    if PY2:
        # Python 2 way
    else:
        # Python 3 way
```

but rather

```
if PY2:
    def my_func():
        # Python 2 way
else:
    def my_func():
        # Python 3 way
```

Meanwhile, defining an exact function according to the version instead of judging the version in the function itself will increase the modularity and the readability of the code.

The round() function tie-breaking and return type

round() tie breaking

In Python 2, using `round()` on a number equally close to two integers will return the one furthest from 0. For example:

Python 2.x ≤2.7

```
round(1.5)  # Out: 2.0
round(0.5)  # Out: 1.0
round(-0.5)  # Out: -1.0
round(-1.5)  # Out: -2.0
```

In Python 3 however, `round()` will return the even integer (aka *bankers' rounding*). For example:

Python 3.x ≥3.0

```
round(1.5)  # Out: 2
round(0.5)  # Out: 0
round(-0.5)  # Out: 0
round(-1.5)  # Out: -2
```

The round() function follows the *half to even rounding* strategy that will round half-way numbers to the nearest even integer (for example, `round(2.5)` now returns 2 rather than 3.0).

As per reference in Wikipedia , this is also known as *unbiased rounding* , *convergent rounding* , *statistician's rounding* , *Dutch rounding* , *Gaussian rounding* , or *odd-even rounding* .

Half to even rounding is part of the IEEE 754 standard and it's also the default rounding mode in Microsoft's .NET.

This rounding strategy tends to reduce the total rounding error. Since on average the amount of numbers that are rounded up is the same as the amount of numbers that are rounded down, rounding errors cancel out. Other rounding methods instead tend to have an upwards or downwards bias in the average error.

---

round() return type

The `round()` function returns a `float` type in Python 2.7

Python 2.x ≤2.7

```
round(4.8)
# 5.0
```

Starting from Python 3.0, if the second argument (number of digits) is omitted, it returns an `int` .

Python 3.x ≥3.0

```
round(4.8)
# 5
```

File I/O

`file` is no longer a builtin name in 3.x (`open` still works).

Internal details of file I/O have been moved to the standard library `io` module, which is also the new home of `StringIO` :

```
import io
assert io.open is open # the builtin is an alias
buffer = io.StringIO()
buffer.write('hello, ') # returns number of characters written
buffer.write('world!\n')
buffer.getvalue() # 'hello, world!\n'
```

The file mode (text vs binary) now determines the type of data produced by reading a file (and type required for writing):

```
with open('data.txt') as f:
    first_line = next(f)
    assert type(first_line) is str
with open('data.bin', 'rb') as f:
    first_kb = f.read(1024)
    assert type(first_kb) is bytes
```

The encoding for text files defaults to whatever is returned by `locale.getpreferredencoding(False)` . To specify an encoding explicitly, use the `encoding` keyword parameter:

```
with open('old_japanese_poetry.txt', 'shift_jis') as text:
    haiku = text.read()
```

Octal Constants

In Python 2, an octal literal could be defined as

```
>>> 0755  # only Python 2
```

To ensure cross-compatibility, use

```
0o755  # both Python 2 and Python 3
```

Return value when writing to a file object

In Python 2, writing directly to a file handle returns `None` :

Python 2.x ≥2.3

```
hi = sys.stdout.write('hello world\n')
# Out: hello world
type(hi)
# Out: <type 'NoneType'>
```

In Python 3, writing to a handle will return the number of characters written when writing text, and the number of bytes written when writing bytes:

Python 3.x ≥3.0

```
import sys

char_count = sys.stdout.write('hello world 🐍\n')
# Out: hello world 🐍
char_count
# Out: 14

byte_count = sys.stdout.buffer.write(b'hello world \xf0\x9f\x90\x8d\n')
# Out: hello world 🐍
byte_count
# Out: 17
```

cmp function removed in Python 3

In Python 3 the `cmp` built-in function was removed, together with the `__cmp__` special method.

From the documentation:

> The `cmp()` function should be treated as gone, and the `__cmp__()` special method is no longer supported. Use `__lt__()` for sorting, `__eq__()` with `__hash__()`, and other rich comparisons as needed. (If you really need the `cmp()` functionality, you could use the expression `(a > b) - (a < b)` as the equivalent for `cmp(a, b)`.)

Moreover all built-in functions that accepted the `cmp` parameter now only accept the `key` keyword only parameter.

In the `functools` module there is also useful function `cmp_to_key(func)` that allows you to convert from a `cmp`-style function to a `key`-style function:

> Transform an old-style comparison function to a key function. Used with tools that accept key functions (such as `sorted()`, `min()`, `max()`, `heapq.nlargest()`, `heapq.nsmallest()`, `itertools.groupby()`). This function is primarily used as a transition tool for programs being converted from Python 2 which supported the use of comparison functions.

exec statement is a function in Python 3

In Python 2, `exec` is a statement, with special syntax: `exec code [in globals[, locals]]`. In Python 3 `exec` is now a function: `exec(code, [, globals[, locals]])`, and the Python 2 syntax will raise a `SyntaxError`.

As `print` was changed from statement into a function, a `__future__` import was also added. However, there is no `from __future__ import exec_function`, as it is not needed: the exec statement in Python 2 can be also used with syntax that looks exactly like the `exec` function invocation in Python 3. Thus you can change the statements

Python 2.x ≥2.3

```
exec 'code'
exec 'code' in global_vars
exec 'code' in global_vars, local_vars
```

to forms

Python 3.x ≥3.0

```
exec('code')
exec('code', global_vars)
exec('code', global_vars, local_vars)
```

and the latter forms are guaranteed to work identically in both Python 2 and Python 3.

Class Boolean Value

Python 2.x ≤2.7

In Python 2, if you want to define a class boolean value by yourself, you need to implement the `__nonzero__` method on your class. The value is True by default.

```
class MyClass:
    def __nonzero__(self):
        return False

my_instance = MyClass()
print bool(MyClass)       # True
print bool(my_instance)   # False
```

Python 3.x ≥3.0

In Python 3, `__bool__` is used instead of `__nonzero__`

```
class MyClass:
```

```
    def __bool__(self):
        return False

my_instance = MyClass()
print(bool(MyClass))       # True
print(bool(my_instance))   # False
```

Dictionary method changes

In Python 3, many of the dictionary methods are quite different in behaviour from Python 2, and many were removed as well: `has_key` , `iter*` and `view*` are gone. Instead of `d.has_key(key)` , which had been long deprecated, one must now use `key in d` .

In Python 2, dictionary methods `keys` , `values` and `items` return lists. In Python 3 they return *view* objects instead; the view objects are not iterators, and they differ from them in two ways, namely:

- they have size (one can use the `len` function on them)
- they can be iterated over many times

Additionally, like with iterators, the changes in the dictionary are reflected in the view objects.

Python 2.7 has backported these methods from Python 3; they're available as `viewkeys` , `viewvalues` and `viewitems` . To transform Python 2 code to Python 3 code, the corresponding forms are:

- `d.keys()` , `d.values()` and `d.items()` of Python 2 should be changed to `list(d.keys())` , `list(d.values())` and `list(d.items())`
- `d.iterkeys()` , `d.itervalues()` and `d.iteritems()` should be changed to `iter(d.keys())` , or even better, `iter(d)` ; `iter(d.values())` and `iter(d.items())` respectively
- and finally Python 2.7 method calls `d.viewkeys()` , `d.viewvalues()` and `d.viewitems()` can be replaced with `d.keys()` , `d.values()` and `d.items()` .

Porting Python 2 code that *iterates* over dictionary keys, values or items while mutating it is sometimes tricky. Consider:

```
d = {'a': 0, 'b': 1, 'c': 2, '!': 3}
for key in d.keys():
    if key.isalpha():
        del d[key]
```

The code looks as if it would work similarly in Python 3, but there the `keys` method returns a view object, not a list, and if the dictionary changes size while being iterated over, the Python 3 code will crash with `RuntimeError: dictionary changed size during iteration` . The solution is of course to properly write `for key in list(d)` .

Similarly, view objects behave differently from iterators: one cannot use `next()` on them, and one cannot *resume* iteration; it would instead restart; if Python 2 code passes the return value of `d.iterkeys()` , `d.itervalues()` or `d.iteritems()` to a method that expects an iterator instead of an *iterable* , then that should be `iter(d)` , `iter(d.values())` or `iter(d.items())` in Python 3.

encode/decode to hex no longer available

Python 2.x ≤2.7

```
"1deadbeef3".decode('hex')
# Out: '\x1d\xea\xdb\xee\xf3'
'\x1d\xea\xdb\xee\xf3'.encode('hex')
# Out: 1deadbeef3
```

Python 3.x ≥3.0

```
"1deadbeef3".decode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# AttributeError: 'str' object has no attribute 'decode'

b"1deadbeef3".decode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# LookupError: 'hex' is not a text encoding; use codecs.decode() to handle arbitrary codecs

'\x1d\xea\xdb\xee\xf3'.encode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# LookupError: 'hex' is not a text encoding; use codecs.encode() to handle arbitrary codecs

b'\x1d\xea\xdb\xee\xf3'.encode('hex')
```

```
# Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
# AttributeError: 'bytes' object has no attribute 'encode'
```

However, as suggested by the error message, you can use the codecs module to achieve the same result:

```
import codecs
codecs.decode('1deadbeef4', 'hex')
# Out: b'\x1d\xea\xdb\xee\xf4'
codecs.encode(b'\x1d\xea\xdb\xee\xf4', 'hex')
# Out: b'1deadbeef4'
```

Note that codecs.encode returns a bytes object. To obtain a str object just decode to ASCII:

```
codecs.encode(b'\x1d\xea\xdb\xee\xff', 'hex').decode('ascii')
# Out: '1deadbeeff'
```

hasattr function bug in Python 2

In Python 2, when a property raise a error, hasattr will ignore this property, returning False .

```
class A(object):
    @property
    def get(self):
        raise IOError


class B(object):
    @property
    def get(self):
        return 'get in b'

a = A()
b = B()

print 'a hasattr get: ', hasattr(a, 'get')
# output False in Python 2 (fixed, True in Python 3)
print 'b hasattr get', hasattr(b, 'get')
# output True in Python 2 and Python 3
```

This bug is fixed in Python3. So if you use Python 2, use

```
try:
    a.get
except AttributeError:
    print("no get property!")
```

or use getattr instead

```
p = getattr(a, "get", None)
if p is not None:
    print(p)
else:
    print("no get property!")
```

Syntax

Parameters

Remarks

There are currently two supported versions of Python: 2.7 (Python 2) and 3.6 (Python 3). Additionally versions 3.3 and 3.4 receive security updates in source format.

Python 2.7 is backwards-compatible with most earlier versions of Python, and can run Python code from most 1.x and 2.x versions of Python unchanged. It is broadly available, with an extensive collection of packages. It is also considered deprecated by the CPython developers, and receives only security and bug-fix development. The

CPython developers intend to abandon this version of the language in 2020 .

According to Python Enhancement Proposal 373 there are no planned future releases of Python 2 after 25 June 2016, but bug fixes and security updates will be supported until 2020. (It doesn't specify what exact date in 2020 will be the sunset date of Python 2.)

Python 3 intentionally broke backwards-compatibility, to address concerns the language developers had with the core of the language. Python 3 receives new development and new features. It is the version of the language that the language developers intend to move forward with.

Over the time between the initial release of Python 3.0 and the current version, some features of Python 3 were back-ported into Python 2.6, and other parts of Python 3 were extended to have syntax compatible with Python 2. Therefore it is possible to write Python that will work on both Python 2 and Python 3, by using future imports and special modules (like **six**).

Future imports have to be at the beginning of your module:

```
from __future__ import print_function
# other imports and instructions go after __future__
print('Hello world')
```

For further information on the `__future__` module, see the relevant page in the Python documentation .

The 2to3 tool is a Python program that converts Python 2.x code to Python 3.x code, see also the Python documentation .

The package six provides utilities for Python 2/3 compatibility:

- unified access to renamed libraries
- variables for string/unicode types
- functions for method that got removed or has been renamed

A reference for differences between Python 2 and Python 3 can be found here .