# Python3.x和Python2.x的区别

1.性能

Py3.0运行 pystone benchmark的速度比Py2.5慢30%。Guido认为Py3.0有极大的优化空间，在字符串和整形操作上可

以取得很好的优化结果。

Py3.1性能比Py2.5慢15%，还有很大的提升空间。

2.编码

Py3.X源码文件默认使用utf-8编码，这就使得以下代码是合法的：

 >>> 中国 = 'china'

 >>>print(中国)

 china

3. 语法

1）去除了<>，全部改用!=

2）去除``，全部改用repr()

3）关键词加入as 和with，还有True,False,None

4）整型除法返回浮点数，要得到整型结果，请使用//

5）加入nonlocal语句。使用noclocal x可以直接指派外围（非全局）变量

6）去除print语句，加入print()函数实现相同的功能。同样的还有 exec语句，已经改为exec()函数

 例如：

 2.X: print "The answer is", 2\*2

 3.X: print("The answer is", 2\*2)

 2.X: print x, # 使用逗号结尾禁止换行

 3.X: print(x, end=" ") # 使用空格代替换行

 2.X: print # 输出新行

 3.X: print() # 输出新行

 2.X: print >>sys.stderr, "fatal error"

 3.X: print("fatal error", file=sys.stderr)

 2.X: print (x, y) # 输出repr((x, y))

 3.X: print((x, y)) # 不同于print(x, y)!

7）改变了顺序操作符的行为，例如x<y，当x和y类型不匹配时抛出TypeError而不是返回随即的 bool值

8）输入函数改变了，删除了raw\_input，用input代替：

 2.X:guess = int(raw\_input('Enter an integer : ')) # 读取键盘输入的方法

 3.X:guess = int(input('Enter an integer : '))

9）去除元组参数解包。不能def(a, (b, c)):pass这样定义函数了

10）新式的8进制字变量，相应地修改了oct()函数。

 2.X的方式如下：

 >>> 0666

 438

 >>> oct(438)

 '0666'

 3.X这样：

 >>> 0666

 SyntaxError: invalid token (<pyshell#63>, line 1)

 >>> 0o666

 438

 >>> oct(438)

 '0o666'

11）增加了 2进制字面量和bin()函数

 >>> bin(438)

 '0b110110110'

 >>> \_438 = '0b110110110'

 >>> \_438

 '0b110110110'

12）扩展的可迭代解包。在Py3.X 里，a, b, \*rest = seq和 \*rest, a = seq都是合法的，只要求两点：rest是list

对象和seq是可迭代的。

13）新的super()，可以不再给super()传参数，

 >>> class C(object):

 def \_\_init\_\_(self, a):

 print('C', a)

 >>> class D(C):

 def \_\_init(self, a):

 super().\_\_init\_\_(a) # 无参数调用super()

 >>> D(8)

 C 8

 <\_\_main\_\_.D object at 0x00D7ED90>

14）新的metaclass语法：

 class Foo(\*bases, \*\*kwds):

 pass

15）支持class decorator。用法与函数decorator一样：

 >>> def foo(cls\_a):

 def print\_func(self):

 print('Hello, world!')

 cls\_a.print = print\_func

 return cls\_a

 >>> @foo

 class C(object):

 pass

 >>> C().print()

 Hello, world!

class decorator可以用来玩玩狸猫换太子的大把戏。更多请参阅PEP 3129

4. 字符串和字节串

1）现在字符串只有str一种类型，但它跟2.x版本的unicode几乎一样。

2）关于字节串，请参阅“数据类型”的第2条目

5.数据类型

1）Py3.X去除了long类型，现在只有一种整型——int，但它的行为就像2.X版本的long

2）新增了bytes类型，对应于2.X版本的八位串，定义一个bytes字面量的方法如下：

 >>> b = b'china'

 >>> type(b)

 <type 'bytes'>

str对象和bytes对象可以使用.encode() (str -> bytes) or .decode() (bytes -> str)方法相互转化。

 >>> s = b.decode()

 >>> s

 'china'

 >>> b1 = s.encode()

 >>> b1

 b'china'

3）dict的.keys()、.items 和.values()方法返回迭代器，而之前的iterkeys()等函数都被废弃。同时去掉的还有

dict.has\_key()，用 in替代它吧

6.面向对象

1）引入抽象基类（Abstraact Base Classes，ABCs）。

2）容器类和迭代器类被ABCs化，所以cellections模块里的类型比Py2.5多了很多。

 >>> import collections

 >>> print('\n'.join(dir(collections)))

 Callable

 Container

 Hashable

 ItemsView

 Iterable

 Iterator

 KeysView

 Mapping

 MappingView

 MutableMapping

 MutableSequence

 MutableSet

 NamedTuple

 Sequence

 Set

 Sized

 ValuesView

 \_\_all\_\_

 \_\_builtins\_\_

 \_\_doc\_\_

 \_\_file\_\_

 \_\_name\_\_

 \_abcoll

 \_itemgetter

 \_sys

 defaultdict

 deque

另外，数值类型也被ABCs化。关于这两点，请参阅 PEP 3119和PEP 3141。

3）迭代器的next()方法改名为\_\_next\_\_()，并增加内置函数next()，用以调用迭代器的\_\_next\_\_()方法

4）增加了@abstractmethod和 @abstractproperty两个 decorator，编写抽象方法（属性）更加方便。

7.异常

1）所以异常都从 BaseException继承，并删除了StardardError

2）去除了异常类的序列行为和.message属性

3）用 raise Exception(args)代替 raise Exception, args语法

4）捕获异常的语法改变，引入了as关键字来标识异常实例，在Py2.5中：

 >>> try:

 ... raise NotImplementedError('Error')

 ... except NotImplementedError, error:

 ... print error.message

 ...

 Error

在Py3.0中：

 >>> try:

 raise NotImplementedError('Error')

 except NotImplementedError as error: #注意这个 as

 print(str(error))

 Error

5）异常链，因为\_\_context\_\_在3.0a1版本中没有实现

8.模块变动

1）移除了cPickle模块，可以使用pickle模块代替。最终我们将会有一个透明高效的模块。

2）移除了imageop模块

3）移除了 audiodev, Bastion, bsddb185, exceptions, linuxaudiodev, md5, MimeWriter, mimify, popen2,

rexec, sets, sha, stringold, strop, sunaudiodev, timing和xmllib模块

4）移除了bsddb模块(单独发布，可以从http://www.jcea.es/programacion/pybsddb.htm获取)

5）移除了new模块

6）os.tmpnam()和os.tmpfile()函数被移动到tmpfile模块下

7）tokenize模块现在使用bytes工作。主要的入口点不再是generate\_tokens，而是 tokenize.tokenize()

9.其它

1）xrange() 改名为range()，要想使用range()获得一个list，必须显式调用：

 >>> list(range(10))

 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

2）bytes对象不能hash，也不支持 b.lower()、b.strip()和b.split()方法，但对于后两者可以使用 b.strip(b’

\n\t\r \f’)和b.split(b’ ‘)来达到相同目的

3）zip()、map()和filter()都返回迭代器。而apply()、 callable()、coerce()、 execfile()、reduce()和reload

()函数都被去除了

现在可以使用hasattr()来替换 callable(). hasattr()的语法如：hasattr(string, '\_\_name\_\_')

4）string.letters和相关的.lowercase和.uppercase被去除，请改用string.ascii\_letters 等

5）如果x < y的不能比较，抛出TypeError异常。2.x版本是返回伪随机布尔值的

6）\_\_getslice\_\_系列成员被废弃。a[i:j]根据上下文转换为a.\_\_getitem\_\_(slice(I, j))或 \_\_setitem\_\_和

\_\_delitem\_\_调用

7）file类被废弃，在Py2.5中：

 >>> file

 <type 'file'>

在Py3.X中：

 >>> file

 Traceback (most recent call last):

 File "<pyshell#120>", line 1, in <module>

 file

 NameError: name 'file' is not defined

**使用**2to3**将代码移植到Python 3**

**概述**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#divingin)

几乎所有的Python 2程序都需要一些修改才能正常地运行在Python 3的环境下。为了简化这个转换过程，Python 3自带了一个叫做2to3的实用脚本(Utility Script)，这个脚本会将你的Python 2程序源文件作为输入，然后自动将其转换到Python 3的形式。[案例研究:将chardet移植到Python 3(porting chardet to Python 3)](http://woodpecker.org.cn/diveintopython3/case-study-porting-chardet-to-python-3.html#running2to3)描述了如何运行这个脚本，然后展示了一些它不能自动修复的情况。这篇附录描述了它*能够*自动修复的内容。

print**语句**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#print)

在Python 2里，print是一个语句。无论你想输出什么，只要将它们放在[print](http://woodpecker.org.cn/diveintopython3/your-first-python-program.html#divingin)关键字后边就可以。在Python 3里，print()是一个函数。就像其他的函数一样，print()需要你将想要输出的东西作为参数传给它。

|  |  |  |
| --- | --- | --- |
| **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. 为输出一个空白行，需要调用不带参数的print()。
2. 为输出一个单独的值，需要将这这个值作为print()的一个参数就可以了。
3. 为输出使用一个空格分隔的两个值，用两个参数调用print()即可。
4. 这个例子有一些技巧。在Python 2里，如果你使用一个逗号(,)作为print语句的结尾，它将会用空格分隔输出的结果，然后在输出一个尾随的空格(trailing space)，而不输出回车(carriage return)。在Python 3里，通过把end=' '作为一个关键字参数传给print()可以实现同样的效果。参数end的默认值为'\n'，所以通过重新指定end参数的值，可以取消在末尾输出回车符。
5. 在Python 2里，你可以通过使用>>pipe\_name语法，把输出重定向到一个管道，比如sys.stderr。在Python 3里，你可以通过将管道作为关键字参数file的值传递给print()来完成同样的功能。参数file的默认值为std.stdout，所以重新指定它的值将会使print()输出到一个另外一个管道。

**Unicode字符串**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#unicodeliteral)

Python 2有两种字符串类型：*Unicode*字符串和非Unicode字符串。Python 3只有一种类型：[Unicode字符串(Unicode strings)](http://woodpecker.org.cn/diveintopython3/strings.html#divingin)。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
| ① | u'PapayaWhip' | 'PapayaWhip' |
| ② | ur'PapayaWhip\foo' | r'PapayaWhip\foo' |

1. Python 2里的Unicode字符串在Python 3里即普通字符串，因为在Python 3里字符串总是Unicode形式的。
2. Unicode原始字符串(raw string)(使用这种字符串，Python不会自动转义反斜线"\")也被替换为普通的字符串，因为在Python 3里，所有原始字符串都是以Unicode编码的。

**全局函数**unicode()[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#unicode)

Python 2有两个全局函数可以把对象强制转换成字符串：unicode()把对象转换成Unicode字符串，还有str()把对象转换为非Unicode字符串。Python 3只有一种字符串类型，[Unicode字符串](http://woodpecker.org.cn/diveintopython3/strings.html#divingin)，所以str()函数即可完成所有的功能。(unicode()函数在Python 3里不再存在了。)

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|   | unicode(anything) | str(anything) |

long**长整型**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#long)

Python 2有为非浮点数准备的int和long类型。int类型的最大值不能超过[sys.maxint](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#renames)，而且这个最大值是平台相关的。可以通过在数字的末尾附上一个L来定义长整型，显然，它比int类型表示的数字范围更大。在Python 3里，[只有一种整数类型int](http://woodpecker.org.cn/diveintopython3/native-datatypes.html#numbers)，大多数情况下，它很像Python 2里的长整型。由于已经不存在两种类型的整数，所以就没有必要使用特殊的语法去区别他们。

[进一步阅读：pep 237：统一长整型和整型](http://www.python.org/dev/peps/pep-0237/)。

|  |  |  |
| --- | --- | --- |
| **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. 在Python 2里的十进制长整型在Python 3里被替换为十进制的普通整数。
2. 在Python 2里的十六进制长整型在Python 3里被替换为十六进制的普通整数。
3. 在Python 3里，由于长整型已经不存在了，自然原来的long()函数也没有了。为了强制转换一个变量到整型，可以使用int()函数。
4. 检查一个变量是否是整型，获得它的数据类型，并与一个int类型(不是long)的作比较。
5. 你也可以使用isinstance()函数来检查数据类型；再强调一次，使用int，而不是long，来检查整数类型。

**<> 比较运算符**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#ne)

Python 2支持<>作为!=的同义词。Python 3只支持!=，不再支持<>了。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
| ① | if x <> y: | if x != y: |
| ② | if x <> y <> z: | if x != y != z: |

1. 简单地比较。
2. 相对复杂的三个值之间的比较。

**字典类方法**has\_key()[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#has_key)

在Python 2里，字典对象的*has\_key*()方法用来测试字典是否包含特定的键(key)。Python 3不再支持这个方法了。你需要使用[in运算符](http://woodpecker.org.cn/diveintopython3/native-datatypes.html#mixed-value-dictionaries)。

|  |  |  |
| --- | --- | --- |
| **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. 最简单的形式。
2. 运算符or的优先级高于运算符in，所以这里不需要添加括号。
3. 另一方面，出于同样的原因 — or的优先级大于in，这里需要添加括号。(注意：这里的代码与前面那行完全不同。Python会先解释x or y，得到结果x(如果x[在布尔上下文里的值是真](http://woodpecker.org.cn/diveintopython3/native-datatypes.html#booleans))或者y。然后Python检查这个结果是不是a\_dictionary的一个键。)
4. 运算符in的优先级大于运算符+，所以代码里的这种形式从技术上说不需要括号，但是2to3还是添加了。
5. 这种形式一定需要括号，因为in的优先级大于+。

**返回列表的字典类方法**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#dict)

在Python 2里，许多字典类方法的返回值是列表。其中最常用方法的有*keys*，*items*和*values*。在Python 3里，所有以上方法的返回值改为动态视图(dynamic view)。在一些上下文环境里，这种改变并不会产生影响。如果这些方法的返回值被立即传递给另外一个函数，并且那个函数会遍历整个序列，那么以上方法的返回值是列表或者视图并不会产生什么不同。在另外一些情况下，Python 3的这些改变干系重大。如果你期待一个能被独立寻址元素的列表，那么Python 3的这些改变将会使你的代码卡住(choke)，因为视图(view)不支持索引(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. 使用list()函数将keys()的返回值转换为一个静态列表，出于安全方面的考量，2to3可能会报错。这样的代码是有效的，但是对于使用视图来说，它的效率低一些。你应该检查转换后的代码，看看是否一定需要列表，也许视图也能完成同样的工作。
2. 这是另外一种视图(关于items()方法的)到列表的转换。2to3对values()方法返回值的转换也是一样的。
3. Python 3里不再支持iterkeys()了。如果必要，使用iter()将keys()的返回值转换成为一个迭代器。
4. 2to3能够识别出iterkeys()方法在列表解析里被使用，然后将它转换为Python 3里的keys()方法(不需要使用额外的iter()去包装其返回值)。这样是可行的，因为视图是可迭代的。
5. 2to3也能识别出keys()方法的返回值被立即传给另外一个会遍历整个序列的函数，所以也就没有必要先把keys()的返回值转换到一个列表。相反的，min()函数会很乐意遍历视图。这个过程对min()，max()，sum()，list()，tuple()，set()，sorted()，any()和all()同样有效。

**被重命名或者重新组织的模块**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#imports)

从Python 2到Python 3，标准库里的一些模块已经被重命名了。还有一些相互关联的模块也被组合或者重新组织，以使得这种关联更有逻辑性。

http[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#http)

在Python 3里，几个相关的http模块被组合成一个单独的包，即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. http.client模块实现了一个底层的库，可以用来请求http资源，解析http响应。
2. http.cookies模块提供一个蟒样的(Pythonic)接口来获取通过http头部(http header)Set-Cookie发送的cookies
3. 常用的流行的浏览器会把cookies以文件形式存放在磁盘上，http.cookiejar模块可以操作这些文件。
4. http.server模块实现了一个基本的http服务器

urllib[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#urllib)

Python 2有一些用来分析，编码和获取URL的模块，但是这些模块就像老鼠窝一样相互重叠。在Python 3里，这些模块被重构、组合成了一个单独的包，即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 FancyURLopenerfrom urllib.parse import urlencode |
| ⑥ | from urllib2 import *Request*from urllib2 import *HTTPError* | from urllib.request import Requestfrom urllib.error import HTTPError |

1. 以前，Python 2里的urllib模块有各种各样的函数，包括用来获取数据的urlopen()，还有用来将url分割成其组成部分的splittype()，splithost()和splituser()函数。在新的urllib包里，这些函数被组织得更有逻辑性。2to3将会修改这些函数的调用以适应新的命名方案。
2. 在Python 3里，以前的urllib2模块被并入了urllib包。同时，以urllib2里各种你最喜爱的东西将会一个不缺地出现在Python 3的urllib模块里，比如build\_opener()方法，Request对象，HTTPBasicAuthHandler和friends。
3. Python 3里的urllib.parse模块包含了原来Python 2里urlparse模块所有的解析函数。
4. urllib.robotparse模块解析[robots.txt文件](http://www.robotstxt.org/)。
5. 处理http重定向和其他状态码的FancyURLopener类在Python 3里的urllib.request模块里依然有效。urlencode()函数已经被转移到了urllib.parse里。
6. Request对象在urllib.request里依然有效，但是像HTTPError这样的常量已经被转移到了urllib.error里。

我是否有提到2to3也会重写你的函数调用？比如，如果你的Python 2代码里导入了urllib模块，调用了urllib.urlopen()函数获取数据，2to3会同时修改import语句和函数调用。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|   | import urllibprint urllib.urlopen('http://diveintopython3.org/').read() | import urllib.request, urllib.parse, urllib.errorprint(urllib.request.urlopen('http://diveintopython3.org/').read()) |

dbm[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#dbm)

所有的dbm克隆(dbm clone)现在在单独的一个包里，即dbm。如果你需要其中某个特定的变体，比如gnu dbm，你可以导入dbm包中合适的模块。

|  |  |  |
| --- | --- | --- |
| **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 |

xmlrpc[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#xmlrpc)

xml-rpc是一个通过http协议执行远程rpc调用的轻重级方法。一些xml-rpc客户端和xml-rpc服务端的实现库现在被组合到了独立的包，即xmlrpc。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|   | import *xmlrpclib* | import xmlrpc.client |
|   | import *DocXMLRPCServer*import *SimpleXMLRPCServer* | import xmlrpc.server |

**其他模块**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#othermodules)

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
| ① | try:    import *cStringIO* as *StringIO*except ImportError:    import StringIO | import io |
| ② | try:    import cPickle as pickleexcept 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. 在Python 2里，你通常会这样做，首先尝试把cStringIO导入作为StringIO的替代，如果失败了，再导入StringIO。不要在Python 3里这样做；io模块会帮你处理好这件事情。它会找出可用的最快实现方法，然后自动使用它。
2. 在Python 2里，导入最快的pickle实现也是一个与上边相似的能用方法。在Python 3里，pickle模块会自动为你处理，所以不要再这样做。
3. builtins模块包含了在整个Python语言里都会使用的全局函数，类和常量。重新定义builtins模块里的某个函数意味着在每处都重定义了这个全局函数。这听起来很强大，但是同时也是很可怕的。
4. copyreg模块为用C语言定义的用户自定义类型添加了pickle模块的支持。
5. queue模块实现一个生产者消费者队列(multi-producer, multi-consumer queue)。
6. socketserver模块为实现各种socket server提供了通用基础类。
7. configparser模块用来解析ini-style配置文件。
8. reprlib模块重新实现了内置函数repr()，并添加了对字符串表示被截断前长度的控制。
9. subprocess模块允许你创建子进程，连接到他们的管道，然后获取他们的返回值。

**包内的相对导入**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#import)

包是由一组相关联的模块共同组成的单个实体。在Python 2的时候，为了实现同一个包内模块的相互引用，你会使用import foo或者from foo import Bar。Python 2解释器会先在当前目录里搜索foo.py，然后再去Python搜索路径(sys.path)里搜索。在Python 3里这个过程有一点不同。Python 3不会首先在当前路径搜索，它会直接在Python的搜索路径里寻找。如果你想要包里的一个模块导入包里的另外一个模块，你需要显式地提供两个模块的相对路径。

假设你有如下包，多个文件在同一个目录下：

chardet/

|

+--\_\_init\_\_.py

|

+--constants.py

|

+--mbcharsetprober.py

|

+--universaldetector.py

现在假设universaldetector.py需要整个导入constants.py，另外还需要导入mbcharsetprober.py的一个类。你会怎样做?

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
| ① | import constants | from . import constants |
| ② | from mbcharsetprober import MultiByteCharSetProber | from .mbcharsetprober import MultiByteCharsetProber |

1. 当你需要从包的其他地方导入整个模块，使用新的from . import语法。这里的句号(.)即表示当前文件(universaldetector.py)和你想要导入文件(constants.py)之间的相对路径。在这个样例中，这两个文件在同一个目录里，所以使用了单个句号。你也可以从父目录(from .. import anothermodule)或者子目录里导入。
2. 为了将一个特定的类或者函数从其他模块里直接导入到你的模块的名字空间里，在需要导入的模块名前加上相对路径，并且去掉最后一个斜线(slash)。在这个例子中，mbcharsetprober.py与universaldetector.py在同一个目录里，所以相对路径名就是一个句号。你也可以从父目录(from .. import anothermodule)或者子目录里导入。

**迭代器方法**next()[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#next)

在Python 2里，迭代器有一个*next*()方法，用来返回序列里的下一项。在Python 3里这同样成立，但是现在有了一个新的全局的函数[next()](http://woodpecker.org.cn/diveintopython3/generators.html#generators)，它使用一个迭代器作为参数。

|  |  |  |
| --- | --- | --- |
| **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 = 42for an\_iterator in a\_sequence\_of\_iterators:    an\_iterator.next() | next = 42for an\_iterator in a\_sequence\_of\_iterators:    an\_iterator.\_\_next\_\_() |

1. 最简单的例子，你不再调用一个迭代器的next()方法，现在你将迭代器自身作为参数传递给全局函数next()。
2. 假如你有一个返回值是迭代器的函数，调用这个函数然后把结果作为参数传递给next()函数。(2to3脚本足够智能以正确执行这种转换。)
3. 假如你想定义你自己的类，然后把它用作一个迭代器，在Python 3里，你可以通过定义特殊方法\_\_next\_\_()来实现。
4. 如果你定义的类里刚好有一个next()，它使用一个或者多个参数，2to3执行的时候不会动它。这个类不能被当作迭代器使用，因为它的next()方法带有参数。
5. 这一个有些复杂。如果你恰好有一个叫做next的本地变量，在Python 3里它的优先级会高于全局函数next()。在这种情况下，你需要调用迭代器的特别方法\_\_next\_\_()来获取序列里的下一个元素。(或者，你也可以重构代码以使这个本地变量的名字不叫next，但是2to3不会为你做这件事。)

**全局函数**filter()[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#filter)

在Python 2里，*filter*()方法返回一个列表，这个列表是通过一个返回值为True或者False的函数来检测序列里的每一项得到的。在Python 3里，filter()函数返回一个迭代器，不再是列表。

|  |  |  |
| --- | --- | --- |
| **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. 最简单的情况下，2to3会用一个list()函数来包装filter()，list()函数会遍历它的参数然后返回一个列表。
2. 然而，如果filter()调用已经被list()包裹，2to3不会再做处理，因为这种情况下filter()的返回值是否是一个迭代器是无关紧要的。
3. 为了处理filter(None, ...)这种特殊的语法，2to3会将这种调用从语法上等价地转换为列表解析。
4. 由于for循环会遍历整个序列，所以没有必要再做修改。
5. 与上面相同，不需要做修改，因为列表解析会遍历整个序列，即使filter()返回一个迭代器，它仍能像以前的filter()返回列表那样正常工作。

**全局函数**map()[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#map)

跟[filter()](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#filter)作的改变一样，*map*()函数现在返回一个迭代器。(在Python 2里，它返回一个列表。)

|  |  |  |
| --- | --- | --- |
| **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. 类似对filter()的处理，在最简单的情况下，2to3会用一个list()函数来包装map()调用。
2. 对于特殊的map(None, ...)语法，跟filter(None, ...)类似，2to3会将其转换成一个使用list()的等价调用
3. 如果map()的第一个参数是一个lambda函数，2to3会将其等价地转换成列表解析。
4. 对于会遍历整个序列的for循环，不需要做改变。
5. 再一次地，这里不需要做修改，因为列表解析会遍历整个序列，即使map()的返回值是迭代器而不是列表它也能正常工作。

**全局函数**reduce()[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#reduce)

在Python 3里，*reduce*()函数已经被从全局名字空间里移除了，它现在被放置在fucntools模块里。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|   | reduce(a, b, c) | from functools import reducereduce(a, b, c) |

**全局函数**apply()[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#apply)

Python 2有一个叫做*apply*()的全局函数，它使用一个函数f和一个列表[a, b, c]作为参数，返回值是f(a, b, c)。你也可以通过直接调用这个函数，在列表前添加一个星号(\*)作为参数传递给它来完成同样的事情。在Python 3里，*apply()*函数不再存在了；必须使用星号标记法。

|  |  |  |
| --- | --- | --- |
| **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. 最简单的形式，可以通过在参数列表(就像[a, b, c]一样)前添加一个星号来调用函数。这跟Python 2里的apply()函数是等价的。
2. 在Python 2里，apply()函数实际上可以带3个参数：一个函数，一个参数列表，一个字典命名参数(dictionary of named arguments)。在Python 3里，你可以通过在参数列表前添加一个星号(\*)，在字典命名参数前添加两个星号(\*\*)来达到同样的效果。
3. 运算符+在这里用作连接列表的功能，它的优先级高于运算符\*，所以没有必要在a\_list\_of\_args + z周围添加额外的括号。
4. 2to3脚本足够智能来转换复杂的apply()调用，包括调用导入模块里的函数。

**全局函数**intern()[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#intern)

在Python 2里，你可以用*intern*()函数作用在一个字符串上来限定(intern)它以达到性能优化。在Python 3里，intern()函数被转移到sys模块里了。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|   | intern(aString) | sys.intern(aString) |

exec**语句**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#exec)

就像[print语句](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#print)在Python 3里变成了一个函数一样，*exec*语句也是这样的。exec()函数使用一个包含任意Python代码的字符串作为参数，然后就像执行语句或者表达式一样执行它。exec()跟[eval()](http://woodpecker.org.cn/diveintopython3/advanced-iterators.html#eval)是相似的，但是exec()更加强大并更具有技巧性。eval()函数只能执行单独一条表达式，但是*exec*()能够执行多条语句，导入(import)，函数声明 — 实际上整个Python程序的字符串表示也可以。

|  |  |  |
| --- | --- | --- |
| **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. 在最简单的形式下，因为exec()现在是一个函数，而不是语句，2to3会把这个字符串形式的代码用括号围起来。
2. Python 2里的exec语句可以指定名字空间，代码将在这个由全局对象组成的私有空间里执行。Python 3也有这样的功能；你只需要把这个名字空间作为第二个参数传递给exec()函数。
3. 更加神奇的是，Python 2里的exec语句还可以指定一个本地名字空间(比如一个函数里声明的变量)。在Python 3里，exec()函数也有这样的功能。

execfile**语句**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#execfile)

就像以前的[exec语句](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#exec)，Python 2里的execfile语句也可以像执行Python代码那样使用字符串。不同的是exec使用字符串，而execfile则使用文件。在Python 3里，execfile语句已经被去掉了。如果你真的想要执行一个文件里的Python代码(但是你不想导入它)，你可以通过打开这个文件，读取它的内容，然后调用compile()全局函数强制Python解释器编译代码，然后调用新的exec()函数。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|   | *execfile*('a\_filename') | exec(compile(open('a\_filename').read(), 'a\_filename', 'exec')) |

repr**(反引号)**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#repr)

在Python 2里，为了得到一个任意对象的字符串表示，有一种把对象包装在反引号里(比如`x`)的特殊语法。在Python 3里，这种能力仍然存在，但是你不能再使用反引号获得这种字符串表示了。你需要使用全局函数repr()。

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

1. 记住，x可以是任何东西 — 一个类，函数，模块，基本数据类型，等等。repr()函数可以使用任何类型的参数。
2. 在Python 2里，反引号可以嵌套，导致了这种令人费解的(但是有效的)表达式。2to3足够智能以将这种嵌套调用转换到repr()函数。

try...except**语句**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#except)

从Python 2到Python 3，[捕获异常](http://woodpecker.org.cn/diveintopython3/your-first-python-program.html#exceptions)的语法有些许变化。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
| ① | try:    import mymodule*except* ImportError, e    pass | try:    import mymoduleexcept ImportError as e:    pass |
| ② | try:    import mymoduleexcept (RuntimeError, ImportError), e    pass | try:    import mymoduleexcept (RuntimeError, ImportError) as e:    pass |
| ③ | try:    import mymoduleexcept ImportError:    pass | *no change* |
| ④ | try:    import mymoduleexcept:    pass | *no change* |

1. 相对于Python 2里在异常类型后添加逗号，Python 3使用了一个新的关键字，as。
2. 关键字as也可以用在一次捕获多种类型异常的情况下。
3. 如果你捕获到一个异常，但是并不在意访问异常对象本身，Python 2和Python 3的语法是一样的。
4. 类似地，如果你使用一个保险方法(fallback)来捕获*所有*异常，Python 2和Python 3的语法是一样的。

☞在导入模块(或者其他大多数情况)的时候，你绝对不应该使用这种方法(指以上的fallback)。不然的话，程序可能会捕获到像KeyboardInterrupt(如果用户按Ctrl-C来中断程序)这样的异常，从而使调试变得更加困难。

raise**语句**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#raise)

Python 3里，[抛出自定义异常](http://woodpecker.org.cn/diveintopython3/your-first-python-program.html#exceptions)的语法有细微的变化。

|  |  |  |
| --- | --- | --- |
| **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. 抛出不带用户自定义错误信息的异常，这种最简单的形式下，语法没有改变。
2. 当你想要抛出一个带用户自定义错误信息的异常时，改变就显而易见了。Python 2用一个逗号来分隔异常类和错误信息；Python 3把错误信息作为参数传递给异常类。
3. Python 2支持一种更加复杂的语法来抛出一个带用户自定义回溯(stack trace，堆栈追踪)的异常。在Python 3里你也可以这样做，但是语法完全不同。
4. 在Python 2里，你可以抛出一个不带异常类的异常，仅仅只有一个异常信息。在Python 3里，这种形式不再被支持。2to3将会警告你它不能自动修复这种语法。

**生成器的**throw**方法**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#throw)

在Python 2里，生成器有一个*throw*()方法。调用a\_generator.throw()会在生成器被暂停的时候抛出一个异常，然后返回由生成器函数获取的下一个值。在Python 3里，这种功能仍然可用，但是语法上有一点不同。

|  |  |  |
| --- | --- | --- |
| **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. 最简单的形式下，生成器抛出不带用户自定义错误信息的异常。这种情况下，从Python 2到Python 3语法上没有变化 。
2. 如果生成器抛出一个带用户自定义错误信息的异常，你需要将这个错误信息字符串(error string)传递给异常类来以实例化它。
3. Python 2还支持抛出只有异常信息的异常。Python 3不支持这种语法，并且2to3会显示一个警告信息，告诉你需要手动地来修复这处代码。

**全局函数**xrange()[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#xrange)

在Python 2里，有两种方法来获得一定范围内的数字：*range*()，它返回一个列表，还有*range*()，它返回一个迭代器。在Python 3里，range()返回迭代器，xrange()不再存在了。

|  |  |  |
| --- | --- | --- |
| **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. 在最简单的情况下，2to3会简单地把xrange()转换为range()。
2. 如果你的Python 2代码使用range()，2to3不知道你是否需要一个列表，或者是否一个迭代器也行。出于谨慎，2to3可能会报错，然后使用list()把range()的返回值强制转换为列表类型。
3. 如果在列表解析里有xrange()函数，就没有必要将其返回值转换为一个列表，因为列表解析对迭代器同样有效。
4. 类似的，for循环也能作用于迭代器，所以这里也没有改变任何东西。
5. 函数sum()能作用于迭代器，所以2to3也没有在这里做出修改。就像[返回值为视图(view)而不再是列表的字典类方法](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#dict)一样，这同样适用于min()，max()，sum()，list()，tuple()，set()，sorted()，any()，all()。

**全局函数**raw\_input()**和**input()[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#raw_input)

Python 2有两个全局函数，用来在命令行请求用户输入。第一个叫做input()，它等待用户输入一个Python表达式(然后返回结果)。第二个叫做*raw\_input*()，用户输入什么它就返回什么。这让初学者非常困惑，并且这被广泛地看作是Python语言的一个“肉赘”(wart)。Python 3通过重命名raw\_input()为input()，从而切掉了这个肉赘，所以现在的input()就像每个人最初期待的那样工作。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
| ① | raw\_input() | input() |
| ② | raw\_input('prompt') | input('prompt') |
| ③ | input() | eval(input()) |

1. 最简单的形式，raw\_input()被替换成input()。
2. 在Python 2里，raw\_input()函数可以指定一个提示符作为参数。Python 3里保留了这个功能。
3. 如果你真的想要请求用户输入一个Python表达式，计算结果，可以通过调用input()函数然后把返回值传递给eval()。

**函数属性**func\_\*[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#funcattrs)

在Python 2里，函数的里的代码可以访问到函数本身的特殊属性。在Python 3里，为了一致性，这些特殊属性被重新命名了。

|  |  |  |
| --- | --- | --- |
| **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. \_\_name\_\_属性(原func\_name)包含了函数的名字。
2. \_\_doc\_\_属性(原funcdoc)包含了你在函数源代码里定义的文档字符串(*docstring*)
3. \_\_defaults\_\_属性(原func\_defaults)是一个保存参数默认值的元组。
4. \_\_dict\_\_属性(原func\_dict)是一个支持任意函数属性的名字空间。
5. \_\_closure\_\_属性(原func\_closure)是一个由cell对象组成的元组，它包含了函数对自由变量(free variable)的绑定。
6. \_\_globals\_\_属性(原func\_globals)是一个对模块全局名字空间的引用，函数本身在这个名字空间里被定义。
7. \_\_code\_\_属性(原func\_code)是一个代码对象，表示编译后的函数体。

**I/O方法**xreadlines()[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#xreadlines)

在Python 2里，文件对象有一个*xreadlines*()方法，它返回一个迭代器，一次读取文件的一行。这在for循环中尤其有用。事实上，后来的Python 2版本给文件对象本身添加了这样的功能。

在Python 3里，xreadlines()方法不再可用了。2to3可以解决简单的情况，但是一些边缘案例则需要人工介入。

|  |  |  |
| --- | --- | --- |
| **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. 如果你以前调用没有参数的xreadlines()，2to3会把它转换成文件对象本身。在Python 3里，这种转换后的代码可以完成前同样的工作：一次读取文件的一行，然后执行for循环的循环体。
2. 如果你以前使用一个参数(每次读取的行数)调用xreadlines()，2to3不能为你完成从Python 2到Python 3的转换，你的代码会以这样的方式失败：AttributeError: '\_io.TextIOWrapper' object has no attribute 'xreadlines'。你可以手工的把xreadlines()改成readlines()以使代码能在Python 3下工作。(readline()方法在Python 3里返回迭代器，所以它跟Python 2里的xreadlines()效率是不相上下的。)

☃

**使用元组而非多个参数的**lambda**函数**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#tuple_params)

在Python 2里，你可以定义匿名*lambda*函数(anonymous lambda function)，通过指定作为参数的元组的元素个数，使这个函数实际上能够接收多个参数。事实上，Python 2的解释器把这个元组“解开”(unpack)成命名参数(named arguments)，然后你可以在lambda函数里引用它们(通过名字)。在Python 3里，你仍然可以传递一个元组作为lambda函数的参数，但是Python解释器不会把它解析成命名参数。你需要通过位置索引(positional index)来引用每个参数。

|  |  |  |
| --- | --- | --- |
| **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. 如果你已经定义了一个lambda函数，它使用包含一个元素的元组作为参数，在Python 3里，它会被转换成一个包含到x1[0]的引用的lambda函数。x1是2to3脚本基于原来元组里的命名参数自动生成的。
2. 使用含有两个元素的元组(x, y)作为参数的lambda函数被转换为x\_y，它有两个位置参数，即x\_y[0]和x\_y[1]。
3. 2to3脚本甚至可以处理使用嵌套命名参数的元组作为参数的lambda函数。产生的结果代码有点难以阅读，但是它在Python 3下跟原来的代码在Python 2下的效果是一样的。
4. 你可以定义使用多个参数的lambda函数。如果没有括号包围在参数周围，Python 2会把它当作一个包含多个参数的lambda函数；在这个lambda函数体里，你通过名字引用这些参数，就像在其他类型的函数里所做的一样。这种语法在Python 3里仍然有效。

**特殊的方法属性**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#methodattrs)

在Python 2里，类方法可以访问到定义他们的类对象(class object)，也能访问方法对象(method object)本身。im\_self是类的实例对象；im\_func是函数对象，im\_class是类本身。在Python 3里，这些属性被重新命名，以遵循其他属性的命名约定。

|  |  |  |
| --- | --- | --- |
| **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\_\_ |

\_\_nonzero\_\_**特殊方法**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#nonzero)

在Python 2里，你可以创建自己的类，并使他们能够在布尔上下文(boolean context)中使用。举例来说，你可以实例化这个类，并把这个实例对象用在一个if语句中。为了实现这个目的，你定义一个特别的\_\_nonzero\_\_()方法，它的返回值为True或者False，当实例对象处在布尔上下文中的时候这个方法就会被调用 。在Python 3里，你仍然可以完成同样的功能，但是这个特殊方法的名字变成了\_\_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. 当在布尔上下文使用一个类对象时，Python 3会调用\_\_bool\_\_()，而非\_\_nonzero\_\_()。
2. 然而，如果你有定义了一个使用两个参数的\_\_nonzero\_\_()方法，2to3脚本会假设你定义的这个方法有其他用处，因此不会对代码做修改。

**八进制类型**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#numliterals)

在Python 2和Python 3之间，定义八进制(*octal*)数的语法有轻微的改变。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|   | x = 0755 | x = 0o755 |

sys.maxint[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#renames)

由于[长整型和整型被整合在一起](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#long)了，sys.maxint常量不再精确。但是因为这个值对于检测特定平台的能力还是有用处的，所以它被Python 3保留，并且重命名为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变成了maxsize。
2. 所有的sys.maxint都变成了sys.maxsize。

**全局函数**callable()[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#callable)

在Python 2里，你可以使用全局函数*callable*()来检查一个对象是否可调用(callable，比如函数)。在Python 3里，这个全局函数被取消了。为了检查一个对象是否可调用，可以检查特殊方法\_\_call\_\_()的存在性。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|   | callable(anything) | hasattr(anything, '\_\_call\_\_') |

**全局函数**zip()[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#zip)

在Python 2里，全局函数*zip*()可以使用任意多个序列作为参数，它返回一个由元组构成的列表。第一个元组包含了每个序列的第一个元素；第二个元组包含了每个序列的第二个元素；依次递推下去。在Python 3里，zip()返回一个迭代器，而非列表。

|  |  |  |
| --- | --- | --- |
| **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. 最简单的形式，你可以通过调用list()函数包装zip()的返回值来恢复zip()函数以前的功能，list()函数会遍历这个zip()函数返回的迭代器，然后返回结果的列表表示。
2. 在已经会遍历序列所有元素的上下文环境里(比如这里对join()方法的调用)，zip()返回的迭代器能够正常工作。2to3脚本会检测到这些情况，不会对你的代码作出改变。

StandardError**异常**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#standarderror)

在Python 2里，*StandardError*是除了StopIteration，GeneratorExit，KeyboardInterrupt，SystemExit之外所有其他内置异常的基类。在Python 3里，StandardError已经被取消了；使用Exception替代。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|   | x = StandardError() | x = Exception() |
|   | x = StandardError(a, b, c) | x = Exception(a, b, c) |

types**模块中的常量**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#types)

types模块里各种各样的常量能帮助你决定一个对象的类型。在Python 2里，它包含了代表所有基本数据类型的常量，如dict和int。在Python 3里，这些常量被已经取消了。只需要使用基础类型的名字来替代。

|  |  |  |
| --- | --- | --- |
| **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被映射为bytes，而非str，因为Python 2里的“string”(非Unicode编码的字符串，即普通字符串)事实上只是一些使用某种字符编码的字节序列(a sequence of bytes)。

**全局函数**isinstance()[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#isinstance)

isinstance()函数检查一个对象是否是一个特定类(class)或者类型(type)的实例。在Python 2里，你可以传递一个由类型(types)构成的元组给isinstance()，如果该对象是元组里的任意一种类型，函数返回True。在Python 3里，你依然可以这样做，但是不推荐使用把一种类型作为参数传递两次。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|   | isinstance(x, (int, float, int)) | isinstance(x, (int, float)) |

basestring**数据类型**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#basestring)

Python 2有两种字符串类型：Unicode编码的字符串和非Unicode编码的字符串。但是其实还有另外 一种类型，即*basestring*。它是一个抽象数据类型，是str和unicode类型的超类(superclass)。它不能被直接调用或者实例化，但是你可以把它作为isinstance()的参数来检测一个对象是否是一个Unicode字符串或者非Unicode字符串。在Python 3里，只有一种字符串类型，所以basestring就没有必要再存在了。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|   | isinstance(x, basestring) | isinstance(x, str) |

itertools**模块**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#itertools)

Python 2.3引入了itertools模块，它定义了全局函数zip()，map()，filter()的变体(variant)，这些变体的返回类型为迭代器，而非列表。在Python 3里，由于这些全局函数的返回类型本来就是迭代器，所以这些itertools里的这些变体函数就被取消了。([在itertools模块里仍然还有许多其他的有用的函数](http://woodpecker.org.cn/diveintopython3/advanced-iterators.html#more-itertools)，而不仅仅是以上列出的这些。)

|  |  |  |
| --- | --- | --- |
| **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. 使用全局的zip()函数，而非itertools.izip()。
2. 使用map()而非itertools.imap()。
3. itertools.ifilter()变成了filter()。
4. itertools模块在Python 3里仍然存在，它只是不再包含那些已经转移到全局名字空间的函数。2to3脚本能够足够智能地去移除那些不再有用的导入语句，同时保持其他的导入语句的完整性。

sys.exc\_type**,**sys.exc\_value**,**sys.exc\_traceback[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#sys_exc)

处理异常的时候，在sys模块里有三个你可以访问的变量：sys.exc\_type，sys.exc\_value，sys.exc\_traceback。(实际上这些在Python 1的时代就有。)从Python 1.5开始，由于新出的sys.exc\_info，不再推荐使用这三个变量了，这是一个包含所有以上三个元素的元组。在Python 3里，这三个变量终于不再存在了；这意味着，你必须使用sys.exc\_info。

|  |  |  |
| --- | --- | --- |
| **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] |

**对元组的列表解析**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#paren)

在Python 2里，如果你需要编写一个遍历元组的列表解析，你不需要在元组值的周围加上括号。在Python 3里，这些括号是必需的。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|   | [i for i in 1, 2] | [i for i in (1, 2)] |

os.getcwdu()**函数**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#getcwdu)

Python 2有一个叫做os.getcwd()的函数，它将当前的工作目录作为一个(非Unicode编码的)字符串返回。由于现代的文件系统能够处理能何字符编码的目录名，Python 2.3引入了os.getcwdu()函数。os.getcwdu()函数把当前工作目录用Unicode编码的字符串返回。在Python 3里，由于[只有一种字符串类型(Unicode类型的)](http://woodpecker.org.cn/diveintopython3/strings.html#divingin)，所以你只需要os.getcwd()就可以了。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Python 2** | **Python 3** |
|   | *os.getcwdu*() | os.getcwd() |

**元类(metaclass)**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#metaclass)

在Python 2里，你可以通过在类的声明中定义metaclass参数，或者定义一个特殊的类级别的(class-level)*\_\_metaclass\_\_*属性，来创建元类。在Python 3里，\_\_metaclass\_\_属性已经被取消了。

|  |  |  |
| --- | --- | --- |
| **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. 在声明类的时候声明metaclass参数，这在Python 2和Python 3里都有效，它们是一样的。
2. 在类的定义里声明\_\_metaclass\_\_属性在Python 2里有效，但是在Python 3里不再有效。
3. 2to3能够构建一个有效的类声明，即使这个类继承自多个父类。

**关于代码风格**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#nitpick)

以下所列的“修补”(fixes)实质上并不算真正的修补。意思就是，他们只是代码的风格上的事情，而不涉及到代码的本质。但是Python的开发者们在使得代码风格尽可能一致方面非常有兴趣(have a vested interest)。为此，有一个专门[o描述Python代码风格的官方指导手册](http://www.python.org/dev/peps/pep-0008/) — 细致到能使人痛苦 — 都是一些你不太可能关心的在各种各样的细节上的挑剔。鉴于2to3为转换代码提供了一个这么好的条件，脚本的作者们添加了一些可选的特性以使你的代码更具可读性。

set()**字面值(literal)(显式的)**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#set_literal)

在Python 2城，定义一个字面值集合(literal set)的唯一方法就是调用set(a\_sequence)。在Python 3里这仍然有效，但是使用新的标注记号(literal notation)：大括号({})是一种更清晰的方法。这种方法除了空集以外都有效，因为字典也用大括号标记，所以[{}表示一个空的字典，而不是一个空集](http://woodpecker.org.cn/diveintopython3/native-datatypes.html#emptyset)。

☞2to3脚本默认不会修复set()字面值。为了开启这个功能，在命令行调用2to3的时候指定-f set\_literal参数。

|  |  |  |
| --- | --- | --- |
| **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} |

**全局函数**buffer()**(显式的)**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#buffer)

用C实现的Python对象可以导出一个“缓冲区接口”(buffer interface)，它允许其他的Python代码直接读写一块内存。(这听起来很强大，它也同样可怕。)在Python 3里，buffer()被重新命名为memoryview()。(实际的修改更加复杂，但是你几乎可以忽略掉这些不同之处。)

☞2to3脚本默认不会修复buffer()函数。为了开启这个功能，在命令行调用2to3的时候指定-f buffer参数。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Before** | **After** |
|   | x = *buffer*(y) | x = *memoryview*(y) |

**逗号周围的空格(显式的)**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#wscomma)

尽管Python对用于缩进和凸出(indenting and outdenting)的空格要求很严格，但是对于空格在其他方面的使用Python还是很自由的。在列表，元组，集合和字典里，空格可以出现在逗号的前面或者后面，这不会有什么坏影响。但是，Python代码风格指导手册上指出，逗号前不能有空格，逗号后应该包含一个空格。尽管这纯粹只是一个美观上的考量(代码仍然可以正常工作，在Python 2和Python 3里都可以)，但是2to3脚本可以依据手册上的标准为你完成这个修复。

☞2to3脚本默认不会修复逗号周围的空格。为了开启这个功能，在命令行调用2to3的时候指定-f wscomma参数。

|  |  |  |
| --- | --- | --- |
| **Notes** | **Before** | **After** |
|   | a ,b | a, b |
|   | {a :b} | {a: b} |

**惯例(Common idioms)(显式的)**[**#**](http://woodpecker.org.cn/diveintopython3/porting-code-to-python-3-with-2to3.html#idioms)

在Python社区里建立起来了许多惯例。有一些比如while 1: loop，它可以追溯到Python 1。(Python直到Python 2.3才有真正意义上的布尔类型，所以开发者以前使用1和0替代。)当代的Python程序员应该锻炼他们的大脑以使用这些惯例的现代版。

☞2to3脚本默认不会为这些惯例做修复。为了开启这个功能，在命令行调用2to3的时候指定-f idioms参数。

|  |  |  |
| --- | --- | --- |
| **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) |