## 高级特性

## 迭代

如果给定一个list或tuple，我们可以通过for循环来遍历这个list或tuple，这种遍历我们称为迭代（Iteration）。

在Python中，迭代是通过for ... in来完成的，而很多语言比如C语言，迭代list是通过下标完成的，比如Java代码：

for (i=0; i<list.length; i++) {

n = list[i];

}

可以看出，Python的for循环抽象程度要高于C的for循环，因为Python的for循环不仅可以用在list或tuple上，还可以作用在其他可迭代对象上。

list这种数据类型虽然有下标，但很多其他数据类型是没有下标的，但是，只要是可迭代对象，无论有无下标，都可以迭代，比如dict就可以迭代：

>>> d = {'a': 1, 'b': 2, 'c': 3}

>>> **for** key **in** d:

... print(key)

...

a

c

b

因为dict的存储不是按照list的方式顺序排列，所以，迭代出的结果顺序很可能不一样。

默认情况下，dict迭代的是key。如果要迭代value，可以用for value in d.values()，如果要同时迭代key和value，可以用for k, v in d.items()。

由于字符串也是可迭代对象，因此，也可以作用于for循环：

>>> **for** ch **in** 'ABC':

... print(ch)

...

A

B

C

所以，当我们使用for循环时，只要作用于一个可迭代对象，for循环就可以正常运行，而我们不太关心该对象究竟是list还是其他数据类型。

那么，如何判断一个对象是可迭代对象呢？方法是通过collections模块的Iterable类型判断：

>>> **from** collections **import** Iterable

>>> isinstance('abc', Iterable) *# str是否可迭代*

True

>>> isinstance([1,2,3], Iterable) *# list是否可迭代*

True

>>> isinstance(123, Iterable) *# 整数是否可迭代*

False

最后一个小问题，如果要对list实现类似Java那样的下标循环怎么办？Python内置的enumerate函数可以把一个list变成索引-元素对，这样就可以在for循环中同时迭代索引和元素本身：

>>> **for** i, value **in** enumerate(['A', 'B', 'C']):

... print(i, value)

...

0 A

1 B

2 C

上面的for循环里，同时引用了两个变量，在Python里是很常见的，比如下面的代码：

>>> **for** x, y **in** [(1, 1), (2, 4), (3, 9)]:

... print(x, y)

...

1 1

2 4

3 9

### 练习

请使用迭代查找一个list中最小和最大值，并返回一个tuple：

窗体顶端

# -\*- coding: utf-8 -\*-

def findMinAndMax(L):



# 测试

if findMinAndMax([]) != (None, None):

print('测试失败!')

elif findMinAndMax([7]) != (7, 7):

print('测试失败!')

elif findMinAndMax([7, 1]) != (1, 7):

print('测试失败!')

elif findMinAndMax([7, 1, 3, 9, 5]) != (1, 9):

print('测试失败!')

else:

print('测试成功!')

## 列表生成器

列表生成式即List Comprehensions，是Python内置的非常简单却强大的可以用来创建list的生成式。

举个例子，要生成list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]可以用list(range(1, 11))：

>>> list(range(1, 11))

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

但如果要生成[1x1, 2x2, 3x3, ..., 10x10]怎么做？方法一是循环：

>>> L = []

>>> **for** x **in** range(1, 11):

... L.append(x \* x)

...

>>> L

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

但是循环太繁琐，而列表生成式则可以用一行语句代替循环生成上面的list：

>>> [x \* x **for** x **in** range(1, 11)]

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

写列表生成式时，把要生成的元素x \* x放到前面，后面跟for循环，就可以把list创建出来，十分有用，多写几次，很快就可以熟悉这种语法。

for循环后面还可以加上if判断，这样我们就可以筛选出仅偶数的平方：

>>> [x \* x **for** x **in** range(1, 11) **if** x % 2 == 0]

[4, 16, 36, 64, 100]

还可以使用两层循环，可以生成全排列：

>>> [m + n **for** m **in** 'ABC' **for** n **in** 'XYZ']

['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']

三层和三层以上的循环就很少用到了。

运用列表生成式，可以写出非常简洁的代码。例如，列出当前目录下的所有文件和目录名，可以通过一行代码实现：

>>> **import** os *# 导入os模块，模块的概念后面讲到*

>>> [d **for** d **in** os.listdir('.')] *# os.listdir可以列出文件和目录*

['.emacs.d', '.ssh', '.Trash', 'Adlm', 'Applications', 'Desktop', 'Documents', 'Downloads', 'Library', 'Movies', 'Music', 'Pictures', 'Public', 'VirtualBox VMs', 'Workspace', 'XCode']

for循环其实可以同时使用两个甚至多个变量，比如dict的items()可以同时迭代key和value：

>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }

>>> **for** k, v **in** d.items():

... print(k, '=', v)

...

y = B

x = A

z = C

因此，列表生成式也可以使用两个变量来生成list：

>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }

>>> [k + '=' + v **for** k, v **in** d.items()]

['y=B', 'x=A', 'z=C']

最后把一个list中所有的字符串变成小写：

>>> L = ['Hello', 'World', 'IBM', 'Apple']

>>> [s.lower() **for** s **in** L]

['hello', 'world', 'ibm', 'apple']

### if ... else

使用列表生成式的时候，有些童鞋经常搞不清楚if...else的用法。

例如，以下代码正常输出偶数：

>>> [x **for** x **in** range(1, 11) **if** x % 2 == 0]

[2, 4, 6, 8, 10]

但是，我们不能在最后的if加上else：

>>> [x **for** x **in** range(1, 11) **if** x % 2 == 0 **else** 0]

File "<stdin>", line 1

[x **for** x **in** range(1, 11) **if** x % 2 == 0 **else** 0]

^

SyntaxError: invalid syntax

这是因为跟在for后面的if是一个筛选条件，不能带else，否则如何筛选？

另一些童鞋发现把if写在for前面必须加else，否则报错：

>>> [x **if** x % 2 == 0 **for** x **in** range(1, 11)]

File "<stdin>", line 1

[x **if** x % 2 == 0 **for** x **in** range(1, 11)]

^

SyntaxError: invalid syntax

这是因为for前面的部分是一个表达式，它必须根据x计算出一个结果。因此，考察表达式：x if x % 2 == 0，它无法根据x计算出结果，因为缺少else，必须加上else：

>>> [x **if** x % 2 == 0 **else** -x **for** x **in** range(1, 11)]

[-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]

上述for前面的表达式x if x % 2 == 0 else -x才能根据x计算出确定的结果。

可见，在一个列表生成式中，for前面的if ... else是表达式，而for后面的if是过滤条件，不能带else。

### 练习

如果list中既包含字符串，又包含整数，由于非字符串类型没有lower()方法，所以列表生成式会报错：

>>> L = ['Hello', 'World', 18, 'Apple', None]

>>> [s.lower() for s in L]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 1, in <listcomp>

AttributeError: 'int' object has no attribute 'lower'

使用内建的isinstance函数可以判断一个变量是不是字符串：

>>> x = 'abc'

>>> y = 123

>>> isinstance(x, str)

True

>>> isinstance(y, str)

False

请修改列表生成式，通过添加if语句保证列表生成式能正确地执行：

窗体顶端

# -\*- coding: utf-8 -\*-

L1 = ['Hello', 'World', 18, 'Apple', None]



# 测试:

print(L2)

if L2 == ['hello', 'world', 'apple']:

print('测试通过!')

else:

print('测试失败!')

 Run

窗体底端

### 小结

运用列表生成式，可以快速生成list，可以通过一个list推导出另一个list，而代码却十分简洁。

## 生成器

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器：generator。

要创建一个generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的[]改成()，就创建了一个generator：

>>> L = [x \* x **for** x **in** range(10)]

>>> L

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> g = (x \* x **for** x **in** range(10))

>>> g

<generator object <genexpr> at 0x1022ef630>

创建L和g的区别仅在于最外层的[]和()，L是一个list，而g是一个generator。

我们可以直接打印出list的每一个元素，但我们怎么打印出generator的每一个元素呢？

如果要一个一个打印出来，可以通过next()函数获得generator的下一个返回值：

>>> **next**(g)

0

>>> **next**(g)

1

>>> **next**(g)

4

>>> **next**(g)

9

>>> **next**(g)

16

>>> **next**(g)

25

>>> **next**(g)

36

>>> **next**(g)

49

>>> **next**(g)

64

>>> **next**(g)

81

>>> **next**(g)

Traceback (most recent call last):

File "<stdin>", line 1, **in** <**module**>

StopIteration

我们讲过，generator保存的是算法，每次调用next(g)，就计算出g的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration的错误。

当然，上面这种不断调用next(g)实在是太变态了，正确的方法是使用for循环，因为generator也是可迭代对象：

>>> g = (x \* x **for** x **in** range(10))

>>> **for** n **in** g:

... print(n)

...

0

1

4

9

16

25

36

49

64

81

所以，我们创建了一个generator后，基本上永远不会调用next()，而是通过for循环来迭代它，并且不需要关心StopIteration的错误。

generator非常强大。如果推算的算法比较复杂，用类似列表生成式的for循环无法实现的时候，还可以用函数来实现。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

**def** **fib**(max):

n, a, b = 0, 0, 1

**while** n < max:

print(b)

a, b = b, a + b

n = n + 1

**return** 'done'

注意，赋值语句：

a, b = b, a + b

相当于：

t = (b, a + b) # t是一个tuple

a = t[0]

b = t[1]

但不必显式写出临时变量t就可以赋值。

上面的函数可以输出斐波那契数列的前N个数：

>>> fib(6)

1

1

2

3

5

8

'done'

仔细观察，可以看出，fib函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

也就是说，上面的函数和generator仅一步之遥。要把fib函数变成generator，只需要把print(b)改为yield b就可以了：

**def** **fib**(max):

n, a, b = 0, 0, 1

**while** n < max:

**yield** b

a, b = b, a + b

n = n + 1

**return** 'done'

这就是定义generator的另一种方法。如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator：

>>> f = fib(6)

>>> f

<generator object fib at 0x104feaaa0>

这里，最难理解的就是generator和函数的执行流程不一样。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。

举个简单的例子，定义一个generator，依次返回数字1，3，5：

**def** **odd**():

print('step 1')

**yield** 1

print('step 2')

**yield**(3)

print('step 3')

**yield**(5)

调用该generator时，首先要生成一个generator对象，然后用next()函数不断获得下一个返回值：

>>> o = odd()

>>> **next**(o)

step 1

1

>>> **next**(o)

step 2

3

>>> **next**(o)

step 3

5

>>> **next**(o)

Traceback (most recent call last):

File "<stdin>", line 1, **in** <**module**>

StopIteration

可以看到，odd不是普通函数，而是generator，在执行过程中，遇到yield就中断，下次又继续执行。执行3次yield后，已经没有yield可以执行了，所以，第4次调用next(o)就报错。

回到fib的例子，我们在循环过程中不断调用yield，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator后，我们基本上从来不会用next()来获取下一个返回值，而是直接使用for循环来迭代：

>>> **for** n **in** fib(6):

... print(n)

...

1

1

2

3

5

8

但是用for循环调用generator时，发现拿不到generator的return语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration的value中：

>>> g = fib(6)

>>> **while** True:

... **try**:

... x = next(g)

... print('g:', x)

... **except** StopIteration **as** e:

... print('Generator return value:', e.value)

... **break**

...

g: 1

g: 1

g: 2

g: 3

g: 5

g: 8

Generator **return** value: done

关于如何捕获错误，后面的错误处理还会详细讲解。

### 练习

[杨辉三角](http://baike.baidu.com/view/7804.htm)定义如下：

1

/ \

1 1

/ \ / \

1 2 1

/ \ / \ / \

1 3 3 1

/ \ / \ / \ / \

1 4 6 4 1

/ \ / \ / \ / \ / \

1 5 10 10 5 1

把每一行看做一个list，试写一个generator，不断输出下一行的list：

窗体顶端

# -\*- coding: utf-8 -\*-

def triangles():



# 期待输出:

# [1]

# [1, 1]

# [1, 2, 1]

# [1, 3, 3, 1]

# [1, 4, 6, 4, 1]

# [1, 5, 10, 10, 5, 1]

# [1, 6, 15, 20, 15, 6, 1]

# [1, 7, 21, 35, 35, 21, 7, 1]

# [1, 8, 28, 56, 70, 56, 28, 8, 1]

# [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]

n = 0

results = []

for t in triangles():

results.append(t)

n = n + 1

if n == 10:

break

for t in results:

print(t)

if results == [

[1],

[1, 1],

[1, 2, 1],

[1, 3, 3, 1],

[1, 4, 6, 4, 1],

[1, 5, 10, 10, 5, 1],

[1, 6, 15, 20, 15, 6, 1],

[1, 7, 21, 35, 35, 21, 7, 1],

[1, 8, 28, 56, 70, 56, 28, 8, 1],

[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]

]:

print('测试通过!')

else:

print('测试失败!')

 Run

窗体底端

### 小结

generator是非常强大的工具，在Python中，可以简单地把列表生成式改成generator，也可以通过函数实现复杂逻辑的generator。

要理解generator的工作原理，它是在for循环的过程中不断计算出下一个元素，并在适当的条件结束for循环。对于函数改成的generator来说，遇到return语句或者执行到函数体最后一行语句，就是结束generator的指令，for循环随之结束。

请注意区分普通函数和generator函数，普通函数调用直接返回结果：

>>> r = abs(6)

>>> r

6

generator函数的“调用”实际返回一个generator对象：

>>> g = fib(6)

>>> g

<generator object fib at 0x1022ef948>

## 高阶函数

### 变量可以指向函数

X = abs(10) 函数abs计算的结果，赋值给var x

* 10

Y = abs 变量y指向函数abs

### 函数名也是变量

那么函数名是什么呢？函数名其实就是指向函数的变量！对于abs()这个函数，完全可以把函数名abs 看成变量，它指向一个可以计算绝对值的函数

### 传入函数

既然变量可以指向函数，函数的参数能接收变量，那么一个函数可以接收另一个函数作为参数，这种函数就称之为高阶函数

### map/reduce

map()函数,接收两个参数，一个是函数对象，一个是Iterable对象，map将传入的函数一次作用到序列的每一个元素，并把结果作为新的Iterable返回。

所以，map()作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的f(x) = x^2,还可以计算任意复杂的函数，比如，把这个list所有数字转为字符串

L= [1,2,3,4]

L2 = list(map(str,L))

Reduce()函数，接收两个参数，一个是函数对象，一个是Iterable对象，reduce把结果继续和下一个序列元素做累积计算，其效果就是

Reduce( f , [ x1,x2, x3, x4 ] ) =  **f( f(f(x1,x2) ,x3 ), x4 )**

### Filter

filter()函数,接收两个参数，一个是函数对象，一个是Iterable对象，fileter将传入的函数一次作用到序列的每一个元素，根据返回值True False 来决定这个元素师去还是留，并把结果作为新的Iterable返回。

### sorted