**参考自** [**https://www.liaoxuefeng.com/wiki/1016959663602400/1017318207388128**](https://www.liaoxuefeng.com/wiki/1016959663602400/1017318207388128)

**切片**

**取前3个元素,Python提供了切片（Slice）操作符**

**L[0:3]表示，从索引0开始取，直到索引3为止，但不包括索引3{也就是说[0:1]，[:1]取的是0位元素的值，[:-1]同理，取的是-2元素的值}。即索引0，1，2，正好是3个元素**

L = ['Michael', 'Sarah', 'Tracy', 'Bob', 'Jack']

>>> L[0:3]

['Michael', 'Sarah', 'Tracy']

**也可以从索引1开始，取出2个元素出来，**

>>> L[1:3]

['Sarah', 'Tracy']

**如果第一个索引是0，还可以省略：**

>>> L[:3]

['Michael', 'Sarah', 'Tracy']

所有数，每5个取一个：

>>> L[::5]

[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]

**类似的，既然Python支持L[-1]取倒数第一个元素，那么它同样支持倒数切片，试试：**

>>> L[-2:]

['Bob', 'Jack']

>>> L[-2:-1]

['Bob']

**甚至什么都不写，只写[:]就可以原样复制一个list：**

>>> L[:]

[0, 1, 2, 3, ..., 99]

**tuple也是一种list，唯一区别是tuple不可变。因此，tuple也可以用切片操作，只是操作的结果仍是tuple**

字符串'xxx'也可以看成是一种list，每个元素就是一个字符。因此，字符串也可以用切片操作，只是操作结果仍是字符串：

>>> 'ABCDEFG'[:3]

'ABC'

>>> 'ABCDEFG'[::2]

'ACEG'

迭代

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

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

>>> for key in d:

... print(key)

...

a

c

b

字符串也可用作循环

>>> for ch in 'ABC':

... print(ch)

...

A

B

C

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

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

也可引用两个变量

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

... print(x, y)

...

1 1

2 4

3 9

如何用迭代来查找List里的最小值和最大值

def findMinAndMax(L):

if len(L)==0:

return (None,None)

elif len(L)==1:

return (L[0],L[0])

else:

a = L[0]

b = L[0]

for x in L:

if x<a:

a = x

for y in L:

if y>b:

b = y

return a, b

列表生成式

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]

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']

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()]

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

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]

生成器generator

定义：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，创建列表会占用很大的空间。

而生成器并不是立即把结果写入内存， 而是保存的一种计算方式， 通过不断的获取， 可以获取到相应的位置的值,所以占用的内存仅仅是对计算对象的保存。

Yield:不反弹的return，使用next()会有一次延迟，一般使用for…in…格式。

yield条件内的代码会优先运行，而条件外的会在条件内全部循环完毕后才允许运行单次

使用for…in…，可以打印 Yield x中x的值，并且x的值只在条件内与yield循环不会被之前的赋值重新赋值

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

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>

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：

>>> 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

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

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：把print(b)改为yield b

def fib(max):

n, a, b = 0, 0, 1

while n < max:

print(b)

a, b = b, a + b

n = n + 1

return 'done'

>>> for n in fib(6):

... print(n)

...

1 1 2 3 5 8

函数成为generator后用法也改变了

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：

def odd():

print('step 1')

yield 1

print('step 2')

yield(3)

print('step 3')

yield(5)

>>> 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

迭代器

如list、tuple、dict、set、str等, generator，包括生成器和带yield的generator function。这些可以直接作用于for循环的对象统称为可迭代对象：Iterable,可以使用isinstance()判断一个对象是否是Iterable对象：

>>> from collections import Iterable

>>> isinstance([], Iterable)

True

>>> isinstance({}, Iterable)

True

>>> isinstance('abc', Iterable)

True

>>> isinstance((x for x in range(10)), Iterable)

True

>>> isinstance(100, Iterable)

False

可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator

可以使用isinstance()判断一个对象是否是Iterator对象：

>>> **from** collections **import** Iterator

>>> isinstance((x **for** x **in** range(10)), Iterator)

True

>>> isinstance([], Iterator)

False

>>> isinstance({}, Iterator)

False

>>> isinstance('abc', Iterator)

False

生成器都是Iterator对象，但list、dict、str虽然是Iterable，却不是Iterator。

把list、dict、str等Iterable变成Iterator可以使用iter()函数：

>>> isinstance(iter([]), Iterator)

True

>>> isinstance(iter('abc'), Iterator)

True

Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的

凡是可作用于for循环的对象都是Iterable类型；

凡是可作用于next()函数的对象都是Iterator类型，它们表示一个惰性计算的序列；

集合数据类型如list、dict、str等是Iterable但不是Iterator，不过可以通过iter()函数获得一个Iterator对象。

for循环本质上就是通过不断调用next()函数实现的，例如：

**for** x **in** [1, 2, 3, 4, 5]:

**pass**

实际上完全等价于：

*# 首先获得Iterator对象:*

it = iter([1, 2, 3, 4, 5])

*# 循环:*

**while** True:

**try**:

*# 获得下一个值:*

x = next(it)

**except** StopIteration:

*# 遇到StopIteration就退出循环*

**break**