列表生成式

有列表[0，1，2，3，4，5，6，7，8，9]，要求把列表里面的每个值加1

方法1:循环

```
info = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
b = []
# for index,i in enumerate(info):
# print(i+1)
# b.append(i+1)
# print(b)
#enumerate() 函数用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据和数据下标，一般用在 for 循环当中。

for index in info:
  info[index] +=1
 print(info)
```

方法2:map+匿名函数

```
info = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
a = map(lambda x:x+1, info)
print(a)
for i in a:
 print(i)
```

方法3:列表生成式

```
info = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
a = [i+1 for i in range(10)]
print(a)
```

生成器

什么是生成器？

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

生成器是一个特殊的程序，可以被用作控制循环的迭代行为，python中生成器是迭代器的一种，使用yield返回值函数，每次调用yield会暂停，而可以使用next()函数和send()函数恢复生成器。

生成器**类似于返回值为数组的一个函数**，这个函数可以接受参数，可以被调用，但是，不同于一般的函数会一次性返回包括了所有数值的数组，生成器一次只能产生一个值，这样消耗的内存数量将大大减小，而且允许调用函数可以很快的处理前几个返回值，因此生成器看起来像是一个函数，但是表现得却像是迭代器

python中的生成器

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[]中括号改为（ ）小括号，就创建一个generator**

```
#列表生成式
lis = [x*x for x in range(10)]
print(lis)
#生成器
generator_ex = (x*x for x in range(10))
print(generator_ex)

结果:
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
<generator object <genexpr> at 0x000002A4CBF9EBA0>
```

那么创建lis和generator_ex，的区别是什么呢？从表面看就是[ ]和（ ）,但是结果却不一样，一个打印出来是列表（因为是列表生成式），而第二个打印出来却是<generator object <genexpr> at 0x000002A4CBF9EBA0>，那么如何打印出来generator_ex的每一个元素呢？

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

```
#生成器
generator_ex = (x*x for x in range(10))
print(next(generator_ex))
print(next(generator_ex))
print(next(generator_ex))
print(next(generator_ex))
print(next(generator_ex))
print(next(generator_ex))
```

```
print(next(generator_ex))
 print(next(generator_ex))
 print(next(generator_ex))
 print(next(generator_ex))
 print(next(generator_ex))
结果:
0
1
4
9
16
25
36
49
64
81
Traceback (most recent call last):

 File "列表生成式.py", line 42, in <module>

 print(next(generator_ex))

StopIteration
```

大家可以看到，generator保存的是算法，每次调用next(generaotr_ex)就计算出他的下一个元素的值，直到计算出最后一个元素，没有更多的元素时，抛出StopIteration的错误，而且上面这样不断调用是一个不好的习惯，正确的方法是使用for循环，因为**generator也是可迭代对象**：

```
#生成器
generator_ex = (x*x for x in range(10))
for i in generator_ex:
 print(i)

结果:
0
1
4
9
16
```

```
25
36
49
64
81
```

所以我们创建一个generator后，基本上永远不会调用next()，而是通过for循环来迭代，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。

比如著名的斐波那契数列，除第一个和第二个数外，任何一个数都可以由前两个相加得到：

1，1，2，3，5，8，12，21，34.....

斐波那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：

a,b = b ,a+b 其实相当于 t =a+b ,a =b ,b =t ，所以不必写显示写出临时变量t，就可以输出斐波那契数列的前N个数字。上面输出的结果如下：

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也可以用generator来实现，上面我们发现，print(b)每次函数运行都要打印，占内存，所以为了不占内存，我们也可以使用生成器，这里叫yield。如下：

```python
def fib(max):
    n,a,b =0,0,1
    while n < max:
        yield b
        a,b =b,a+b
        n = n+1
    return 'done'

a = fib(10)
```

```
print(fib(10))
但是返回的不再是一个值，而是一个生成器，和上面的例子一样，大家可以看一下结果：

<generator object fib at 0x000001C03AC34FC0>
```

那么这样就不占内存了，这里说一下generator和函数的执行流程，函数是顺序执行的，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，再次被next（ ）调用时候从上次的返回yield语句处急需执行，也就是用多少，取多少，不占内存。

```
def fib(max):
    n,a,b =0,0,1
    while n < max:
        yield b
        a,b =b,a+b
        n = n+1
    return 'done'

a = fib(10)
print(fib(10))
print(a.__next__())
print(a.__next__())
print(a.__next__())
print("可以顺便干其他事情")
print(a.__next__())
print(a.__next__())

结果：
<generator object fib at 0x0000023A21A34FC0>
1
1
2
可以顺便干其他事情
3
5
```

在上面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：

```
def fib(max):
    n,a,b =0,0,1
    while n < max:
        yield b
        a,b =b,a+b
        n = n+1
    return 'done'
for i in fib(6):
    print(i)

 结果:
 1
 1
 2
 3
 5
 8
```

```
def fib(max):
    n,a,b =0,0,1
    while n < max:
        yield b
        a,b =b,a+b
        n = n+1
    return 'done'
g = fib(6)
while True:
    try:
        x = next(g)
        print('generator: ',x)
    except StopIteration as e:
        print("生成器返回值: ",e.value)
        break


 结果:
```

```
generator: 1
generator: 1
generator: 2
generator: 3
generator: 5
generator: 8
生成器返回值： done
```

还可以通过yield实现在单线程的情况下实现并发运算的效果

```python
import time
def consumer(name):
    print("%s 准备学习啦!" %name)
    while True:
        lesson = yield

        print("开始[%s]了,[%s]老师来讲课了!" %(lesson,name))


def producer(name):
    c = consumer('A')
    c2 = consumer('B')
    c.__next__()
    c2.__next__()
    print("同学们开始上课 了!")
    for i in range(10):
        time.sleep(1)
        print("到了两个同学!")
        c.send(i)
        c2.send(i)
```

结果：

```
A 准备学习啦!
B 准备学习啦!
同学们开始上课 了!
到了两个同学!
开始[0]了,[A]老师来讲课了!
开始[0]了,[B]老师来讲课了!
到了两个同学!
开始[1]了,[A]老师来讲课了!
开始[1]了,[B]老师来讲课了!
```

```
到了两个同学!
开始[2]了,[A]老师来讲课了!
开始[2]了,[B]老师来讲课了!
到了两个同学!
开始[3]了,[A]老师来讲课了!
开始[3]了,[B]老师来讲课了!
到了两个同学!
开始[4]了,[A]老师来讲课了!
开始[4]了,[B]老师来讲课了!
到了两个同学!
开始[5]了,[A]老师来讲课了!
开始[5]了,[B]老师来讲课了!
到了两个同学!
开始[6]了,[A]老师来讲课了!
开始[6]了,[B]老师来讲课了!
到了两个同学!
```

def consumer(name):

由上面的例子我么可以发现，python提供了两种基本的方式

生成器函数：也是用def定义的，利用关键字yield一次性返回一个结果，阻塞，重新开始

生成器表达式：返回一个对象，这个对象只有在需要的时候才产生结果——生成器函数

为什么叫生成器函数？因为它随着时间的推移生成了一个数值队列。一般的函数在执行完毕之后会返回一个值然后退出，但是生成器函数会自动挂起，然后重新拾起急需执行，他会利用yield关键字关起函数，给调用者返回一个值，同时保留了当前的足够多的状态，可以使函数继续执行，生成器和迭代协议是密切相关的，可迭代的对象都有一个**next()__**成员方法，这个方法要么返回迭代的下一项，要买引起异常结束迭代。

```python
# 函数有了yield之后，函数名+（）就变成了生成器
# return在生成器中代表生成器的中止，直接报错
# next的作用是唤醒并继续执行
# send的作用是唤醒并继续执行，发送一个信息到生成器内部
'''生成器'''

def create_counter(n):
    print("create_counter")
```

```
    while True:
        yield n
        print("increment n")
        n +=1

gen = create_counter(2)
print(gen)
print(next(gen))
print(next(gen))

结果：
<generator object create_counter at 0x0000023A1694A938>
create_counter
2
increment n
3
Process finished with exit code 0
```

——生成器表达式

把列表生成式加了圆括号

生成器表达式来源于迭代和列表解析的组合，生成器和列表解析类似，但是它使用尖括号而不是方括号

```
>>> # 列表解析生成列表
>>> [ x ** 3 for x in range(5)]
[0, 1, 8, 27, 64]
>>>
>>> # 生成器表达式
>>> (x ** 3 for x in range(5))
<generator object <genexpr> at 0x000000000315F678>
>>> # 两者之间转换
>>> list(x ** 3 for x in range(5))
[0, 1, 8, 27, 64]
```

一个迭代既可以被写成生成器函数，也可以被写成生成器表达式，均支持自动和手动迭代。而且这些生成器只支持一个active迭代，也就是说生成器的迭代器就是生成器本身。

迭代器（迭代就是循环）

我们已经知道，可以直接作用于for循环的数据类型有以下几种：

一类是集合数据类型，如list,tuple,dict,set,str等

一类是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()判断一个对象是否为可**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
```

而生成器不但可以作用于for循环，还可以被next()函数不断调用并返回下一个值，直到最后抛出StopIteration错误表示无法继续返回下一个值了。

所以这里将一下迭代器

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（可迭代对象）`，却不是`Iterator（迭代器）`。

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
```

你可能会问，为什么list、dict、str等数据类型不是Iterator？

这是因为Python的Iterator对象表示的是一个**数据流**，Iterator对象可以被next()函数调用并不断返回下一个数据，直到没有数据时抛出StopIteration错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过next()函数实现按需计算下一个数据，所以Iterator的计算是惰性的，只有在需要返回下一个数据时它才会计算。

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。

判断下列数据类型是可迭代对象or迭代器

```
s='hello'
l=[1,2,3,4]
t=(1,2,3)
d={'a':1}
set={1,2,3}
f=open('a.txt')

s='hello'      #字符串是可迭代对象，但不是迭代器
l=[1,2,3,4]      #列表是可迭代对象，但不是迭代器
 t=(1,2,3)        #元组是可迭代对象，但不是迭代器
 d={'a':1}        #字典是可迭代对象，但不是迭代器
 set={1,2,3}      #集合是可迭代对象，但不是迭代器
 # ************************************
 f=open('test.txt') #文件是可迭代对象，是迭代器

 #如何判断是可迭代对象，只有__iter__方法，执行该方法得到的迭代器对象。
 # 及可迭代对象通过__iter__转成迭代器对象
 from collections import Iterator #迭代器
 from collections import Iterable #可迭代对象

 print(isinstance(s,Iterator))      #判断是不是迭代器
 print(isinstance(s,Iterable))        #判断是不是可迭代对象

 #把可迭代对象转换为迭代器
```

```
print(isinstance(iter(s),Iterator))
注意：文件的判断

f = open('housing.csv')
from collections import Iterator
from collections import Iterable

print(isinstance(f,Iterator))
print(isinstance(f,Iterable))

True
True
```

小结：

- 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。

Python3的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

对yield的总结

（1）通常的for..in...循环中，in后面是一个数组，这个数组就是一个可迭代对象，类似的还有链表，字符串，文件。他可以是a = [1,2,3]，也可以是a = [x*x for x in range(3)]。

它的缺点也很明显，就是所有数据都在内存里面，如果有海量的数据，将会非常耗内存。

（2）生成器是可以迭代的，但是只可以读取它一次。因为用的时候才生成，比如a = (x*x for x in range(3))。!!!!注意这里是小括号而不是方括号。

（3）生成器（generator）能够迭代的关键是他有next()方法，工作原理就是通过重复调用next()方法，直到捕获一个异常。

（4）**带有yield的函数不再是一个普通的函数，而是一个生成器generator，可用于迭代**

（5）yield是一个类似return 的关键字，迭代一次遇到yield的时候就返回yield后面或者右面的值。而且下一次迭代的时候，从上一次迭代遇到的yield后面的代码开始执行

（6）yield就是return返回的一个值，并且记住这个返回的位置。下一次迭代就从这个位置开始。

（7）带有yield的函数不仅仅是只用于for循环，而且可用于某个函数的参数，只要这个函数的参数也允许迭代参数。

（8）send()和next()的区别就在于send可传递参数给yield表达式，这时候传递的参数就会作为yield表达式的值，而yield的参数是返回给调用者的值，也就是说send可以强行修改上一个yield表达式值。

（9）send()和next()都有返回值，他们的返回值是当前迭代遇到的yield的时候，yield后面表达式的值，其实就是当前迭代yield后面的参数。

（10）第一次调用时候必须先next（）或send（）,否则会报错，send后之所以为None是因为这时候没有上一个yield，所以也可以认为next（）等同于send(None