Python生成器的使用技巧详解

0.本集概览

1.生成器可以避免一次性生成整个列表

2.生成器函数的运行过程解析及状态保存

3.生成器表达式的使用方法

4.生成器表达式的可迭代特性

之前我们介绍了**列表解析式**，他的优点很多，比如运行速度快、编写简单，但是有一点我们不要忘了，**他是一次性生成整个列表**。如果整个列表非常大，这对内存也同样会造成很大压力，**想要实现内存的节约，可以将列表解析式转换为生成器表达式**。

1.避免一次性生成整个列表

避免一次性生成整个结果列表的本质是**在需要的时候才逐次产生结果，而不是立即产生全部的结果**，Python中有两种语言结构可以实现这种思路。

一个是生成器函数。外表看上去像是一个函数，但是没有用return语句一次性的返回整个结果对象列表，取而代之的是使用yield语句一次返回一个结果。

另一个是生成器表达式。类似于列表解析，但是方括号换成了圆括号，他们返回按需产生的一个结果对象，而不是构建一个结果列表。

这个“按需”指的是在迭代的环境中，每次迭代按需产生一个对象，因此，上述二者都不会一次性构建整个列表，从而节约了内存空间。

2.生成器函数

下面具体结合例子说说生成器函数。

2.1.运行过程分析

首先，我们还没有详细介绍过函数，先简单说一下，常规函数接受输入的参数然后立即送回单个结果，之后这个函数调用就结束了。

但生成器函数却不同，他通过**yield关键字**返回一个值后，还能从其退出的地方继续运行，因此可以随时间产生一系列的值。**他们自动实现了迭代协议，并且可以出现在迭代环境中。**

运行的过程是这样的：生成器函数返回一个迭代器，for循环等迭代环境对这个迭代器不断调用next函数，不断的运行到下一个yield语句，逐一取得每一个返回值，直到没有yield语句可以运行，最终引发StopIteration异常。看，这个过程是不是很熟悉。

首先，下面这个例子证实了生成器函数返回的是一个迭代器代码片段：

```
def gen_squares(num):
    for x in range(num):
        yield x ** 2

G = gen_squares(5)
print(G)
print(iter(G))
```

运行结果：

```
<generator object gen_squares at 0x0000000002402558>
<generator object gen_squares at 0x0000000002402558>
```

然后**再用手动模拟循环的方式来看看生成器函数的运行过程**，你会发现和前面介绍过的熟悉场景并无二致。 代码片段：

```
def gen_squares(num):
    for x in range(num):
        yield x ** 2

G = gen_squares(3)
print(G)
print(iter(G))
print(next(G))
print(next(G))
print(next(G))
print(next(G))
```

运行结果：

```
<generator object gen_squares at 0x00000000021C2558>
<generator object gen_squares at 0x00000000021C2558>
0
1
4
Traceback (most recent call last):
File "E:/12homework/12homework.py", line 10, in <module>
print(next(G))
StopIteration
```

那这么看，在for循环等真正的使用场景中使用也不难了代码片段：

```
def gen_squares(num):
    for x in range(num):
        yield x ** 2

for i in gen_squares(5):
    print(i, end=' ')
```

运行结果：

```
0 1 4 9 16
```

2.2.状态保存

我们进一步来说说生成器函数里状态保存的话题。在每次循环的时候，生成器函数都会在yield处产生一个值，并将其返回给调用者，即for循环。然后在yield处保存内部状态，并挂起中断退出。在下一轮迭代调用时，从yield的地方继续执行，并且沿用上一轮的函数内部变量的状态，直到内部循环过程结束。

关于这个问题，具体可以看看这个例子：

代码片段：

```
def gen_squares(num):
    for x in range(num):
        yield x ** 2
        print('x={}'.format(x))

for i in gen_squares(4):
    print('x ** 2={}'.format(i))
    print('-------------')
```

运行结果：

```
x ** 2=0
-------------
x=0
x ** 2=1
-------------
x=1
x ** 2=4
-------------
x=2
x ** 2=9
-------------
x=3
```

我们不难发现，生成器函数计算出x的平方后就挂起退出了，但他仍然保存了此时x的值，而yield后的print语句会在for循环的下一轮迭代中首先调用，此时x的值即是上一轮退出时保存的值。

3.生成器表达式

再说说生成器表达式吧。

3.1.使用方法

列表解析式已经是一个不错的选择，从内存使用的角度而言，生成器更优，**因为他不用一次性生成整个对象列表**，这二者之间如何转化呢？

生成器表达式写法上很像列表解析式，但是外面的方括号换成了圆括号，结果大不同，简单的看看： 代码片段：

```
print([x ** 2 for x in range(5)])
print((x ** 2 for x in range(5)))
```

运行结果：

```
[0, 1, 4, 9, 16]
<generator object <genexpr> at 0x0000000002212558>
```

方括号是熟悉的列表解析式，一次性返回整个列表，圆括号是生成器表达式，返回一个生成器对象，而不是一次性生成整个列表。

3.2.适用于迭代环境

同时他支持迭代协议，适用于所有的迭代环境，略举几个例子： 代码片段：

```
for x in (x ** 2 for x in range(5)):
    print(x, end=',')
```

运行结果：

```
0,1,4,9,16,
```

代码片段：

```
print(sum(x ** 2 for x in range(5)))
```

运行结果：

```
30
```

代码片段：

```
print(sorted((x ** 2 for x in range(5)), reverse=True))
```

运行结果：

```
[16, 9, 4, 1, 0]
```

代码片段：

```
print(list(x ** 2 for x in range(5)))
```

运行结果：

```
[0, 1, 4, 9, 16]
```

3.3.集合解析式与生成器对象

集合解析式等效于将生成器对象传入到list、 set、 dict等函数中作为构造参数代码片段：

```
set(f(x) for x in S if P(x))
{f(x) for x in S if P(x)}

{key:val for (key, val) in zip(keys, vals)}
dict(zip(keys, vals))

{x:f(x) for x in items}
dict((x, f(x)) for x in items)
```

作者：酱油哥，清华大学计算机硕士，泰康资管/军工央企职场经历。

[廖雪峰博客]
(https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014317799226173f45ce40636141b6abc8424e12b5fb27000)