迭代器和解析

一、 迭代器

1、 可迭代对象：

for 循环可以用于 Python 中任何序列类型，包括列表、元组、字典、字符串等。实际上 for 循环比这个更为通用，它可以应用于任何可迭代对象：如果对象是实际保存的序列，或者可以在迭代工具中一次产生一个结果的对象，就可以看作是可迭代的。常用的迭代工具有：for 循环、列表解析、in 成员关系测试以及 map 内置函数等。

2、 文件迭代器：

已经打开的文件有一个方法名叫 readline()，可以一次性从一个文件中读取一行文本，每次调用时就会前进到下一列，到达文件末尾时就会返回空字符，我们可以通过这个来检测跳出循环

```
>>> f=open(r"C:\Users\liuyi\Desktop\script1.py")
>>> f.readline()
'import sys\n'
>>> f.readline()
'print(sys.path)\n'
>>> f.readline()
'x=2\n'
>>> f.readline()
'print(2**33)\n'
>>> f.readline()
''
>>> f.readline()
''
```

如今，文件也有一个办法，名叫__next__()方法，差不多有同样的方法，每次调用时就会返回文件的下一行。唯一不同的是到达文件末尾时，_next_会引发内置的 StopIteration 异常。

```
>>> f=open(r"C:\Users\liuyi\Desktop\script1.py")
>>> f.__next__()
'import sys\n'
>>> f.__next__()
'print(sys.path)\n'
>>> f.__next__()
'x=2\n'
>>> f.__next__()
'print(2**33)\n'
>>> f.__next__()
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    f.__next__()
StopIteration
```

这个接口就是 Python 中的迭代协议，任何这类对象都认为是可迭代的。任何这类对象都能够以 for 循环或者其他迭代工具遍历，因为所有迭代工具内部工作起来都是在每次迭代过程中调用__next__，并且捕捉 StopIteration 来确定何时离开。

因此，对于可迭代的文件，逐行读取文本的最佳方法就是根本不要读取，用 for 循环在每轮自动调用 next 前进到下一行。例如，下面是逐行读取文件的方法：

```
>>> for line in open(r"C:\Users\liuyi\Desktop\script1.py"):
...     print(line,end=' ')
...
...
...

import sys
 print(sys.path)
 x=2
 print(2**33)
```

这是读取文件最佳方法，首先它足够简洁，而且运行最快，且内存使用情况足够好。相同效果的原始方式是用 for 循环调用文件的 readlines 方法(for line in f.readlines():)然而这种方法将文件内容全部加载到内存，做成字符串的列表，这样是很差的内存使用的方法。另外一个版本是用 while 循环和 readline 来读取，但是这种方法会比上述方法更慢。这是因为迭代器在 Python 语言中是以 C 语言的速度运行的，而 while 循环版本则是 Python 的虚拟机运行 Python 字节代码的。

3、 手动迭代：iter 和 next

python 标准库中的函数，iter 和 next。iter 函数用来创建一个迭代器， next 函数用来从迭代器中一个个获取数据。这两个函数一般都是在一起使用。

```
>>> a = [1,2,3,4,5]
>>> ga = iter(a)
...
>>>
>>> next(ga)
1
>>> next(ga)
2
>>> next(ga)
3
>>> next(ga)
4
>>> next(ga)
5
>>> next(ga)
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    next(ga)
StopIteration
```

所有数据都取出后，如果再取，就会得到 StopIteration 异常。

在 Python 中，好些内置的数据对象都可以用来作为 iter 函数的参数，获取一个迭代器。上面代码示例用的是 list，tuple，string，set，以及 dict 对象都可以用来创建迭代器：

```
>>> d = {'a':1,'b':2,'c':3}
>>> gd = iter(d)
>>> next(gd)
'a'
>>> next(gd)
'b'
>>> next(gd)
'c'
>>> next(gd)
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    next(gd)
StopIteration
```

用字典对象创建迭代器，获取的元素都是字典对象的 key。

二、 解析

1、 列表解析：通过对序列中的每一项运行一个表达式来创建一个新列表的方法。

（1）基础知识

Eg:L=[x+10 for x in L]

列表解析写在一个方括号里，该表达式使用了循环变量 x+10，后边跟着可迭代对象(for x in L)。

列表解析的结果可以用一个 for 循环来进行模拟：

```
>>> res=[]
>>> L=[1,2,3,4,5]
>>> for x in L:
...     res.append(x+10)
...
...
>>> res
[11, 12, 13, 14, 15]
```

然而列表解析不仅写起来更加精简，而且往往比手写 for 循环运行速度更快（往往快一倍），因为这种迭代在解释器内部是以 C 语言的速度执行的。

（2）在文件上使用列表解析

文件对象有一个 readlines 的方法，可以将文件一次性载入行字符串的列表中。结果的行往往带有一个’ \n’，我们可以通过列表解析一次性去除它。

```
>>> f=open(r"C:\Users\liuyi\Desktop\script1.py")
>>> lines=f.readlines()
>>> lines
['import sys\n', 'print(sys.path)\n', 'x=2\n', 'print(2**33)\n']
>>> lines=[line.rstrip() for line in lines]
>>> lines
['import sys', 'print(sys.path)', 'x=2', 'print(2**33)']
```

列表解析具有很强的表现能力，下面是其他几个事例。

```
>>> lines=[line+'\n' for line in lines]
>>> lines
['import sys\n', 'print(sys.path)\n', 'x=2\n', 'print(2**33)\n']
>>> lines=[line.upper() for line in lines]
>>> lines
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X=2\n', 'PRINT(2**33)\n']
>>> lines=[line.lower().split() for line in lines]
>>> lines
[['import', 'sys'], ['print(sys.path)'], ['x=2'], ['print(2**33)']]
>>> lines=[line.replace(' ','!')for line in open(r"C:\Users\liuyi\Desktop\script1.py")]
>>> lines
['import!sys\n', 'print(sys.path)\n', 'x=2\n', 'print(2**33)\n']
```

（3）列表解析与函数式编程工具

i.列表解析与 map

当我们想将[1,2,3,4,5,6]中每一项的平方数组成新列表时，可以采用列表解析和 map 函数两种方式：

```
>>> s=[1,2,3,4,5,6]
>>> newlist=[x**2 for x in s]
>>> newlist
[1, 4, 9, 16, 25, 36]
>>> newlist=list(map(lambda x:x**2,s))
>>> newlist
[1, 4, 9, 16, 25, 36]
```

当我们想将字符串中每一个字符的 ASC 码组成列表时也可以采用这两种方式：

```
>>> s='spam'
>>> List=[ord(x)for x in s]
>>> List
[115, 112, 97, 109]
>>> List=list(map(ord,s))
>>> List
[115, 112, 97, 109]
```

ii.列表解析与 filter

由于列表解析可以在 for 循环后边编写一个 if 分支，用来增加逻辑，所以使用了 if 分支的列表解析可以实现 filter 的功能。

例如当我们收集 0-9 偶数平方时，可以使用列表解析或者 map 和 filter 组合的方式实现。

```
>>> List1=[x**2 for x in range(10) if x%2==0] #列表解析方法
>>> List1
[0, 4, 16, 36, 64]
>>> List2=list(map(lambda x:x**2,filter(lambda x:x%2==0,range(10))))#map和filter
>>> List2
[0, 4, 16, 36, 64]
```

三、 生成器

1、生成器基础知识

前面章节中，已经详细介绍了什么是迭代器。生成器本质上也是迭代器，不过它比较特殊。以 list 容器为例，在使用该容器迭代一组数据时，必须事先将所有数据存储到容器中，才能开始迭代；而生成器却不同，它可以实现在迭代的同时生成元素。

不仅如此，生成器的创建方式也比迭代器简单很多，大体分为以下 2 步：

1、 定义一个以 yield 关键字标识返回值的函数；

2、 调用刚刚创建的函数，即可创建一个生成器。

```
>>> def generator(N):
...     for i in range(5):
...         yield i**2
...
...
>>> for i in generator(5):
...     print(i,end='\t')
...
...
0       1       4       9       16
```

定义的这个生成器对象支持迭代器协议，也就是说生成器对象有一个 next 方法，它可以开始这个函数，或者从上一个 yield 值地方恢复，并且在得到一系列值的最后产生异常。

```
>>> x=generator(5)
>>> x
<generator object generator at 0x00000229A591F4C0>
>>> next(x)
0
>>> next(x)
1
>>> next(x)
4
>>> next(x)
9
>>> next(x)
16
>>> next(x)
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    next(x)
StopIteration
```

同样的，我们也可以不用生成器，用其他方式来完成，比如直接定义一个返回列表的函数：

```
>>> def res(n):
...     x=[]
...     for i in range(n):x.append(i**2)
...     return x
...
>>> for i in res(5):
...     print(i)
...
...
0
1
4
9
16
```

也可以用 map 函数或者列表解析式：

Version1：

```
>>> #列表解析式
>>> for i in [x**2 for x in range(5)]:
...     print(i,end=' ')
...
...
0 1 4 9 16
```

Version2：

```
>>> #map函数
>>> for i in list(map(lambda x:x**2,range(5))):
...     print(i,end=' ')
...
...
0 1 4 9 16
```

尽管如此，生成器在内存使用和性能方面都更好。他们允许函数避免临时再做所有的工作，生成器将在 loop 迭代中处理一系列值的时间分布开来。它提供了一个简单的替代方案来手动将类对象保存在迭代的状态中。有了生成器，函数变量就能够进行自动的保存和恢复。

2、生成器表达式

从语法上来说，生成器表达式就像一般的列表解析一样，但是他们是括在圆括号而非方括号中的。

```
>>> [x**2 for x in range(5)]
[0, 1, 4, 9, 16]
>>> (x**2 for x in range(5))
<generator object <genexpr> at 0x00000229A34E5E00>
```

实际上，编写一个列表解析基本上等同于在一个内置的调用中包含一个生成器表达式以迫使其一次生成列表中所有结果。但从执行过程来看，生成器表达式很不一样：它不是在内存中构建结果，而是返回一个生成器对象，这个对象会支持迭代协议并在任意的迭代语境操作中。

我们可以用 next 方法去操作这个生成器表达式，但是一般不会这样做，因为 for 循环会自动触发：

```
>>> for i in (i**2 for i in range(5)):
...     print(i,end=' ')
...
...
0 1 4 9 16
```

事实上，每一个迭代语境都是这样，比如 sum、map、sorted 函数等。注意，如果生成器表达式是在其他括号内，生成器自身的括号就不是必须的了。

```
>>> sum(x**2 for x in range(4))
14
>>> sorted(x**2 for x in range(5) if x%2==0)
[0, 4, 16]
```

生成器表达式大体上可以看作是对内存空间的优化，他们不需要像方括号的列表解析一样一次构造出整个结果列表。它们在实际运行起来可能会稍微慢一些，所以它们可能只对于非常大的结果集合的运算是最优选择。