迭代器与列表解析

一.概念：

1.容器：是一种把多个元素组织在一起的数据结构，容器中的元素可以逐个地迭代获取，可以用in 或not in关键字判断元素是否包含在容器中. 通常这类数据结构把所有的元素存储在内存中，常见的容器对象有 list tuple set dict

2.可迭代对象：很多容器都是可迭代对象，此外还有更多的对象同样也是可迭代对象；比如处于打开状态的files，可迭代对象实现了__iter__方法，该方法返回一个迭代器对象

3.迭代器：是一个带状态的对象，他能在你调用next()方法的时候返回容器中的下一个值，任何实现了__iter__和__next__()方法的对象都是迭代器，__iter__返回迭代器自身，**__next__返回容器中的下一个值，如果容器中没有更多元素了，则抛出StopIteration异常.**

在前面说的文件这个内置类型中通过open()函数打开的文件中有一个方法readline，可以一次从文件中读取一行文本，每次调用readline方法时，会前进到下一列，达到文件末尾时返回空字符串，我们可通它来检测文件末尾，从而退出循环，但是单纯因为遇到空行就退出也不太现实，因为可能文档的内容中就本身就存在空行，所以我们应该使用next()方法，区别在于每次调用的时候，都会返回文件的下一行，当到达文件末尾时，next()将会引发stopiteration异常，这个next接口就是Python中所谓的迭代协议，所有迭代工具内部工作起来都是每次调用next()并且捕捉StopIteration异常来决定何时退出.

In [3]:
```
f = open('123.log')
print f.readline()
print f.readline()
print f.readline()
print f.readline()
print f.readline()
f.close()
```

```
dfs

sdf



sdfsdfsdf

sdfsa11
```

In [6]:
```
f = open('123.log')
print f.next()
print f.next()
print f.next()
print f.next()
print f.next()
print f.next()
print f.next()
print f.next()
print f.next()
f.close()
```

```
dfs

sdf



sdfsdfsdf

sdfsa11



---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-6-11d8815f97b6> in <module>()
      7 print f.next()
      8 print f.next()
----> 9 print f.next()
     10 print f.next()
     11 f.close()

StopIteration:
```

二.迭代器(有两个基本方法)

iter: 返回迭代器对象本身

next(): 返回迭代器的下一个元素

In [8]:
```
l = [1, 2, 3]
i = iter(l)  #将列表变成迭代器
print i
print l
```

```
<listiterator object at 0x7ff5e4494d90>
[1, 2, 3]
```

In [9]:
```
dir(l)
```

Out[9]:
```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__delslice__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getslice__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__setslice__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

In [10]:
```
dir(i)
```

Out[10]:
```
['__class__',
 '__delattr__',
 '__doc__',
 '__format__',
 '__getattribute__',
 '__hash__',
 '__init__',
 '__iter__',
 '__length_hint__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'next']
```

当for循环开始时, 会把可迭代对象传递给iter内置函数, 以便从可迭代对象中获取一个迭代器, 返回的对象中需要有next方法.

例:

for循环内部如何处理列表这类内置序列类型?

In [12]:
```
l = [1, 2, 3]
i = iter(l)
print i.next()
print i.next()
print i.next()
print i.next()
```

```
1
2
3

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-12-56980a974edb> in <module>()
      4 print i.next()
      5 print i.next()
----> 6 print i.next()

StopIteration:
```

对于文件对象来说iter不是必须的, 因为文件对象就是自己的迭代器. 有**next**方法, 所以每次调用时候就会返回文件的下一行.

In [13]:
```
f = open('123.log')

print dir(f)
```

```
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__', '__format__', '__getattribute__', '__hash__', '__init__', '__iter__', '__new__', '__reduce__', '__re
duce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'close', 'closed', 'encoding', 'errors', 'fileno', 'flush', 'isatty', 'mode', 'na
me', 'newlines', 'next', 'read', 'readinto', 'readline', 'readlines', 'seek', 'softspace', 'tell', 'truncate', 'write', 'writelines', 'xreadlines']
```

三. 列表解析

列表解析是最常应用迭代协议的环境之一.

In [14]:
```
#代码示例1: 对列表中的每个数字元素加10, 得到的结果组成一个新的列表
l = [1,2,3]
res = []
for j in l:
    res.append(j + 10)
else:
    print res
```

```
[11, 12, 13]
```

In [15]:
```
#以上是传统的方法, 它可能不是Python中的最佳实现方法, 我们可以通过列表解析, 通过更少的代码更快的速度实现上面的需求
l = [1,2,3]
print [j + 10 for j in l]

#仅仅只需要1行代码就可以解决上面的问题, 列表解析编写起来代码更加精简, 速度比手工去写for循环速度更快, 因为在解释器内部列表解析是以
#近乎于C语言的速度来执行的, 使用列表解析具有性能优势
```

```
[11, 12, 13]
```

列表解析写在一个[]中, 因为它最终构建了一个新的列表, 它以一个任意的表达式开始, 如 i + 10 后面是一个类似for循环头部的部分, 在python解释器内部执行一个遍历range(10)的迭代, 按照顺序把j赋给每一个元素, 并且收集对各个元素运行左边的表达式结果, 得到的结果列表就是列表解析要表达的内容.

列表解析有更高级的应用, 表达式中嵌套的for循环可以有一个相关的if语句.

In [16]:
```
#示例代码2: 对列表中所有大于3的数字加10, 得到结果组成一个新的列表

l = [1,2,3,4,5]
res = []
for j in l:
    if j > 3:
        res.append(j + 3)
else:
    print res
```

```
[7, 8]
```

以上是传统的方法, 它可能不是Python中的最佳实现方法, 我们可以通过列表解析, 通过更少的代码更快的速度实现上面的需求

In [17]:
```
print [j + 3 for j in range(1,6) if j > 3]
```

```
[7, 8]
```

In [18]:
```
#示例代码3: 列表解析结合三元表达式, 对列表中所有大于3的元素加10, 对其他元素加20

l = [1,2,3,4,5]
res = []
for j in l:
    if j > 3:
        res.append(j + 10)
    else:
        res.append(j + 20)
else:
    print res

#以上是传统的方法, 它可能不是Python中的最佳实现方法, 我们可以通过列表解析, 通过更少的代码更快的速度实现上面的需求
```

```
[21, 22, 23, 14, 15]
```

In [20]:
```
res = [j + 10 if j > 3 else j + 20 for j in range(1, 6)]
print res
```

```
[21, 22, 23, 14, 15]
```

In [24]:
```
#示例代码4: 如果我们需要的话, 列表解析可以更加的复杂, 例如可以包含嵌套的循环
res = []
l = [1,2,3]

for j in l:
    for k in l:
        res.append('%s * %s = %s' %(j, k, j*k))
else:
    print res
#以上是传统的方法, 它可能不是Python中的最佳实现方法, 我们可以通过列表解析, 通过更少的代码更快的速度实现上面的需求
```

```
['1 * 1 = 1', '1 * 2 = 2', '1 * 3 = 3', '2 * 1 = 2', '2 * 2 = 4', '2 * 3 = 6', '3 * 1 = 3', '3 * 2 = 6', '3 * 3 = 9']
```

In [23]:
```
l = [1,2,3]
res = ['%s * %s = %s'%(j, k, j * k) for j in l for k in l]
print res
```

```
['1 * 1 = 1', '1 * 2 = 2', '1 * 3 = 3', '2 * 1 = 2', '2 * 2 = 4', '2 * 3 = 6', '3 * 1 = 3', '3 * 2 = 6', '3 * 3 = 9']
```

四.函数式编程

map: 它是一个内置函数, 把一个函数的调用应用于传入的可迭代对象的每一项.

filter: 选择一个函数为真的项

reduce: 针对可迭代对象中成对的项运行一个函数

zip: 并行遍历

In [25]:
```
#map()函数会根据提供的函数对指定序列做映射, 结果收集在列表中.

help(map)
```

```
Help on built-in function map in module __builtin__:

map(...)
    map(function, sequence[, sequence, ...]) -> list

    Return a list of the results of applying the function to the items of
    the argument sequence(s).  If more than one sequence is given, the
    function is called with an argument list consisting of the corresponding
    item of each sequence, substituting None for missing values when not all
    sequences have the same length.  If the function is None, return a list of
    the items of the sequence (or a list of tuples if more than one sequence).
```

In [28]:
```
def f1(x):
    return x + 10

print map(f1, [1,2,3,4,5])
```

```
[12, 14, 16, 18, 20]
```

In [30]:
```
def f1(x, y):
    return x + y + 10

print map(f1, [1,2,3,4,5], [1,2,3,4,5])
```

```
[12, 14, 16, 18, 20]
```

In [34]:
```
#当函数为None时, 操作和zip相似
print map(None, [1,2,3], [1,2,3])
```

```
[(1, 1), (2, 2), (3, 3)]
```

In [35]:
```
#zip()并行遍历
help(zip)
```

```
Help on built-in function zip in module __builtin__:

zip(...)
    zip(seq1 [, seq2 [...]]) -> [(seq1[0], seq2[0] ...), (...)]

    Return a list of tuples, where each tuple contains the i-th element
    from each of the argument sequences.  The returned list is truncated
    in length to the length of the shortest argument sequence.
```

In [36]:
```
print zip([1,2,3,4,5], ['a', 'b', 'c', 'd', 'e'])
```

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

In [37]:
```
#filter()筛选出可迭代对象中符合条件的元素
help(filter)
```

```
Help on built-in function filter in module __builtin__:

filter(...)
    filter(function or None, sequence) -> list, tuple, or string

    Return those items of sequence for which function(item) is true.  If
    function is None, return the items that are true.  If sequence is a tuple
    or string, return the same type, else return a list.
```

In [38]:
```
def f1(x):
    if x > 3:
        return x

print filter(f1, [1,2,3,4,5])
```

```
[4, 5]
```

In [39]:
```
#当参数function为None时, 返回可迭代对象中为true的元素
print filter(None, [0,1,2,3])
```

```
[1, 2, 3]
```

In [40]:
```
#reduce() 针对可迭代对象中成对的项运行一个函数
help(reduce)
```

```
Help on built-in function reduce in module __builtin__:

reduce(...)
    reduce(function, sequence[, initial]) -> value

    Apply a function of two arguments cumulatively to the items of a sequence,
    from left to right, so as to reduce the sequence to a single value.
    For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates
    ((((1+2)+3)+4)+5).  If initial is present, it is placed before the items
    of the sequence in the calculation, and serves as a default when the
    sequence is empty.
```

In [41]:
```
def f1(x, y):
    return x + y

print reduce(f1, [1,2,3,4,5])
```

```
15
```

In [3]:
```
#问题?  MAP与filter的区别的例子
l = [1,2,3,4,5]


def f1(x):
    if x >= 3:
        return x

r = list(map(f1, l))
print(r)


r = list(filter(f1, l))
print(r)
```

```
[None, None, 3, 4, 5]
[3, 4, 5]
```

In [ ]:
```
#综上 MAP和filter返回的结果是有区别的, filter只会把结果为True的值收集起来!
```