迭代器与列表解析

概念:

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, 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

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

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__()（python2 中实现 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， 从而退出循环 ， 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

In [1]: f= open('1235.log')

In [2]: f.readline()

Out[2]: 'abcdapple\n'

In [3]: f.readline()

Out[3]: 'banana\n'

In [4]: f.readline()

Out[4]: 'sdfsdf\n'

In [5]: f.readline()

Out[5]: '\n'

In [6]: f.readline()

Out[6]: ''

文件对象也有一个方法 ，名字为 next()， 每次调用的时候， 都会返回文件的下一行， 与上面的区别在于， 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 [9]: f.next()

Out[9]: 'abcdapple\n'

In [10]: f.next()

Out[10]: 'banana\n'

In [11]: f.next()

Out[11]: 'sdfsdf\n'

In [12]: f.next()

Out[12]: '\n'

In [13]: f.next()

---

StopIteration Traceback (most recent call last)

<ipython-input-13-c3e65e5362fb> in <module>()

----> 1 f.next()

StopIteration:

迭代器有两个基本的方法

__iter__:返回迭代器对象本身

next()：返回迭代器的下一个元素

```
>>> l = [1, 2, 3]
>>> i = iter(l)  ##将列表变成一个迭代器
>>> i
<listiterator object at 0x7e97d0>
>>> l
[1, 2, 3]

>>> dir(l)
```

['__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']

>>> dir(i)

['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__iter__',

'__length_hint__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',

'__str__', '__subclasshook__', 'next']

>>> i

<listiterator object at 0x7e97d0>

当 for 循环开始时 会通过它传给 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

```
>>> l = [1, 2, 3]
>>> i =iter(l)
>>> i.next()
1
>>> i.next()
2
```

```
>>> i.next()
3
>>> i.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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 每次调用时候就会返回文件的下一行

```
>>> f = open('123', 'r')
>>> f.next()
'sdfsdf\n'
>>> f.next()
'sdf\n'
>>> f.next()
'sdf1\n'
```

```
>>> l = [1, 2, 3]
>>> help(l)

>>> i = iter(l)
>>> help(i)

class listiterator(object)  迭代器中有一个 next 方法
 |  Methods defined here:
 |
 |  __getattribute__(...)
 |      x.__getattribute__('name') <==> x.name
 |
 |  __iter__(...)
 |      x.__iter__() <==> iter(x)
 |
 |  __length_hint__(...)
 |      Private method returning an estimate of len(list(it)).
 |
 |  next(...)
 |      x.next() -> the next value, or raise StopIteration
```

列表解析

列表解析

列表解析是最常应用迭代协议的环境之一

代码示例

1. 1. 对列表中的数字元素加上 10 组成新列表

In [1]: l = [1, 2, 3, 4, 5]

In [2]: for i in range(len(l)):

...: l[i] = l[i] + 10

...:

In [3]: l

Out[3]: [11, 12, 13, 14, 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

In [5]: [j + 10 for j in l]

Out[5]: [21, 22, 23, 24, 25]

## 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

列表解析写在一个[]中 因为它最终构建了一个新的列表 如

l = [i + 10 for i in range(10)]

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 x 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 [1]: l = [1, 2, 3, 4, 5]

In [2]: [j + 3 for j in l if j > 3]

Out[2]: [7, 8]

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

In [3]: l = ['a', 'b', 'c', 'd']

In [4]: m = ['x', 'y', 'z', 'u']

In [5]: [ j + k for j in l for k in m] ##通过列表解析构建了一个 x + y 连接的列表

列表解析结合条件控制三元表达式

```
l = [1, 2, 3, 4, 5]
print l
x = [ j + 10 if j < 3 else j + 20  for j in l]
print x
```

其他迭代环境

map 函数：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

1.map 函数

map 函数会根据提供的函数对指定序列做映射。

map 函数的定义：

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

通过定义可以看到，这个函数的第一个参数是一个函数，剩下的参数是一个或多个序列，

返回值是一个集合。

function 可以理解为是一个一对一或多对一函数，map 的作用是以参数序列中的每一个元

素调用 function 函数，返回包含每次 function 函数返回值的 list。

比如要对一个序列中的每个元素进行平方运算：

map(lambda x: x ** 2, [1, 2, 3, 4, 5])

返回结果为：

[1, 4, 9, 16, 25]

在参数存在多个序列时，会依次以每个序列中相同位置的元素做参数调用 function 函数。

比如要对两个序列中的元素依次求和。

如

1.map(lambda x, y: x + y, [1, 3, 5, 7, 9], [2, 4, 6, 8, 10])

2.map((lambda x, y: x + y),['a', 'b'], ['e', 'f'])

map 返回的 list 中第一个元素为，参数序列 1 的第一个元素加参数序列 2 中的第一个元素(1 + 2)，

list 中的第二个元素为，参数序列 1 中的第二个元素加参数序列 2 中的第二个元素(3 + 4)，

依次类推，最后的返回结果为：

[3, 7, 11, 15, 19]

要注意 function 函数的参数数量，要和 map 中提供的集合数量相匹配。

如果集合长度不相等，会以最小长度对所有集合进行截取。

当函数为 None 时，操作和 zip 相似：

map(None, [1, 3, 5, 7, 9], [2, 4, 6, 8, 10])

返回结果为：

[(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]

## 找出 1 到 10 之间的奇数

## filter(lambda x:x%2!=0,range(1,11))

## 返回值

## [1,3,5,7,9]

In [18]: v
Out[18]: [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
In [21]: reduce(lambda x, y: [x[i]+y[i] for i in range(len(x))], v)

Out[21]: [6, 9, 12]

reduce 函数

python 中的 reduce 内建函数是一个二元操作函数，他用来将一个数据集合（链表，元组等）中的所有数据进行如下操作：

用传给 reduce 中的函数 func()（必须是一个二元操作函数）先对集合中的第 1，2 个数据进行操作，得到的结果再与第三个数据用 func()函数运算，最后得到一个结果。

如：

reduce(lambda x, y: x+y, [1, 2, 3])

```
def myadd(x,y):
    return x+y
sum=reduce(myadd,(1,2,3,4,5,6,7))
print sum
```

Out[18]: [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
In [21]: reduce(lambda x, y: [x[i]+y[i] for i in range(len(x))], v)
Out[21]: [6, 9, 12]

Zip()

In [16]: x = [1,2,3]
In [17]: y = [2,3,4]
In [18]: zip(x, y)
Out[18]: [(1, 2), (2, 3), (3, 4)]