Python可迭代对象，迭代器，生成器的区别

2017年05月16日 12:39:57 阅读数：2319

本篇文章简单谈谈可迭代对象，迭代器和生成器之间的关系。

三者简要关系图

可迭代对象与迭代器

刚开始我认为这两者是等同的，但后来发现并不是这样；下面直接抛出结论：

1）可迭代对象包含迭代器。
2）如果一个对象拥有__iter__方法，其是可迭代对象；如果一个对象拥有next方法，其是迭代器。
3）定义可迭代对象，必须实现__iter__方法；定义迭代器，必须实现__iter__和next方法。

你也许会问，结论3与结论2是不是有一点矛盾？既然一个对象拥有了next方法就是迭代器，那为什么迭代器必须同时实现两方法呢？

因为结论1，迭代器也是可迭代对象，因此迭代器必须也实现__iter__方法。

介绍一下上面涉及到的两个方法：

1）__iter__()

该方法返回的是当前对象的迭代器类的实例。因为可迭代对象与迭代器都要实现这个方法，因此有以下两种写法。

写法一：用于可迭代对象类的写法，返回该可迭代对象的迭代器类的实例。

写法二：用于迭代器类的写法，直接返回self（即自己本身），表示自身即是自己的迭代器。

也许有点晕，没关系，下面会给出两写法的例子，我们结合具体例子看。

2）next()
返回迭代的每一步，实现该方法时注意要最后超出边界要抛出StopIteration异常。

下面举个可迭代对象与迭代器的例子：

[python]

```python
#!/usr/bin/env python
# coding=utf-8


class MyList(object):            # 定义可迭代对象类

    def __init__(self, num):
        self.data = num          # 上边界

    def __iter__(self):
        return MyListIterator(self.data)  # 返回该可迭代对象的迭代器类的实例


class MyListIterator(object):    # 定义迭代器类，其是MyList可迭代对象的迭代器类

    def __init__(self, data):
        self.data = data         # 上边界
        self.now = 0             # 当前迭代值，初始为0

    def __iter__(self):
        return self              # 返回该对象的迭代器类的实例；因为自己就是迭代器，所以返回self

    def next(self):              # 迭代器类必须实现的方法
        while self.now < self.data:
```

```
            self.now += 1
            return self.now - 1  # 返回当前迭代值
        raise StopIteration      # 超出上边界，抛出异常


my_list = MyList(5)              # 得到一个可迭代对象
print type(my_list)              # 返回该对象的类型

my_list_iter = iter(my_list)     # 得到该对象的迭代器实例，iter函数在下面会详细解释
print type(my_list_iter)


for i in my_list:                # 迭代
    print i
```

运行结果：

```
<class '__main__.MyList'>
<class '__main__.MyListIterator'>
0
1
2
3
4
```


问题：上面的例子中出现了iter函数，这是什么东西？和__iter__方法有关系吗？
其实该函数与迭代是息息相关的，通过在Python命令行中打印“help(iter)”得知其有以下两种用法。

用法一：iter(callable, sentinel)
不停的调用callable，直至其的返回值等于sentinel。其中的callable可以是函数，方法或实现了__call__方法的实例。

用法二：iter(collection)
1）用于返回collection对象的迭代器实例，这里的collection我认为表示的是可迭代对象，即该对象必须实现__iter__方法；**事实上iter函数与__iter__方法联系非常紧密，iter()是直接调用该对象的__iter__()，并把__iter__()的返回结果作为自己的返回值，故该用法常被称为“创建迭代器”。**
2）iter函数可以显示调用，或当执行“for i in obj:”，Python解释器会在第一次迭代时自动调用iter(obj)，之后的迭代会调用迭代器的next方法，for语句会自动处理最后抛出的StopIteration异常。

通过上面的例子，相信对可迭代对象与迭代器有了更具体的认识，那么生成器与它们有什么关系呢？下面简单谈一谈

生成器

生成器是一种特殊的迭代器，生成器自动实现了“迭代器协议”（即__iter__和next方法），不需要再手动实现两方法。

生成器在迭代的过程中可以改变当前迭代值，而修改普通迭代器的当前迭代值往往会发生异常，影响程序的执行。

看一个生成器的例子：

[python]

```python
#!/usr/bin/env python
# coding=utf-8


def myList(num):      # 定义生成器
    now = 0           # 当前迭代值，初始为0
    while now < num:
        val = (yield now)                    # 返回当前迭代值，并接受可能的send发送值；yield在下面会解释
        now = now + 1 if val is None else val  # val为None，迭代值自增1，否则重新设定当前迭代值为val

my_list = myList(5)   # 得到一个生成器对象

print my_list.next()  # 返回当前迭代值
print my_list.next()

my_list.send(3)       # 重新设定当前的迭代值
print my_list.next()

print dir(my_list)    # 返回该对象所拥有的方法名，可以看到__iter__与next在其中
```

运行结果：

```
0
1
4
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__',
 __iter__', '__name__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__si
zeof__', '__str__', '__subclasshook__', 'close', 'gi_code', 'gi_frame', 'gi_running', next, 'sen
d', 'throw']
```


具有yield关键字的函数都是生成器，yield可以理解为return，返回后面的值给调用者。不同的是return返回后，函数会释放，而生成器则不会。在直接调用next方法或用for语句进行下一次迭代时，生成器会从yield下一句开始执行，直至遇到下一个yield。