**迭代器和生成器**

上篇推文我们介绍了Vscode搭建Python的开发环境，帮助了一些小伙伴解决了没有Pycharm专业版的窘境，目前市面上出了一本《Visual Studio Code 权威指南》有兴趣的小伙伴可以阅读下。

大家还记得上篇推文《Python实用教程系列——VSCode Python开发环境搭建》中文末留下的问题么？看到这里大家应该就能想到那个问题是使用迭代器和生成器的相关知识点来解决问题啦。

1. **迭代器**

百度百科中对迭代器的一个解释为:迭代器是一种对象，它能够用来遍历标准模板库容器中的部分或全部元素，每一个迭代器对象代表容器中的确定的地址。

注意在Python中所有的集合都是可以迭代的，也就是说可以使用遍历的方式去获取其中的元素，那么在Python的中迭代器的支持哪些实现呢，比如：

（1）for循环

（2）按行遍历文本文件

（3）列表、字典和集合推导式

（4）元组拆包

比如我们常用的for循环和文本读取的操作：

for each in [**'Python'**,**'学习'**]:  
 print(each)  
和：  
testfile = **'1.txt'**with open(testfile) as f:  
 for readline in f:  
 print(readline)

可以看出Python的循环要比C语言的循环来的更加的抽象。

那么为什么迭代器支持这些实现呢？拿文本文件的遍历来说，是因为文本对象支持(或者说实现了)迭代器的协议。

这里简单的说明两个协议:

◆ 迭代器协议: 对象内部实现了\_\_next\_\_和\_\_iter\_\_方法，其就是迭代器。

◆ 可迭代协议: 对象内部实现了\_\_iter\_\_方法，其就是可迭代对象。

**可迭代对象**

有些小伙伴在进行学习的时候，经常会看到一个叫iterable的词，看到这个的时候就说明我们遇到了“可迭代对象”，那么什么是可迭代对象，它有什么特点呢？

在有一些教材上将可直接使用for循环进行操作的对象为可迭代对象，其实这样理解也没有什么太大的问题。如果从可迭代对象的内部的实现角度出发，如果一个对象实现了\_\_iter\_\_方法，我们就可以认为这个对象是可迭代的，这也正好与可迭代协议一致。

比如在list类中就实现了\_\_iter\_\_函数：



补充一点：如果一个对象内部实现了\_\_getitem\_\_方法，而且参数索引从0开始，那么这种对象也是可迭代的。

在实际的编码开发过程中，有什么方法可以直接判断一个对象是不是可迭代的对象呢？有的：就是使用isinstance()方法来进行判断，比如：

from collections import Iterable  
obj\_1 = [1,2,3,4]  
print(isinstance(obj\_1,Iterable))

输出True,这说明对象obj\_1是可迭代的，实时上列表确实是可以迭代的。

在之前的推文中提到过，在Python中一切皆可为对象，接下来我们看一个示例：

from collections import Iterable  
class Iterable\_test:  
 print(**'我爱python学堂'**)  
 *# 定义\_\_iter\_\_方法* def \_\_iter\_\_(self):  
 pass  
  
f = Iterable\_test()  
print(isinstance(f,Iterable))

我们定义了一个类，并在类中实现了\_\_iter\_\_方法，接着我们实例化对象，最后我们判断这个对象是否可迭代。结果为True，这正好验证了上述的点：如果一个对象实现了\_\_iter\_\_方法，我们就可以认为这个对象是可迭代的。

实际上我们还有一个另外一个方法可以进行判断：使用iter()函数:

num = 100  
print(iter(num))  
*# TypeError: 'int' object is not iterable*

输出报错这说明我们想判断的对象不是可迭代对象，一般出现这样的异常我们try/except处理一下就可以了。再例如:

obj\_1 = [1,2,3,4]  
print(iter(obj\_1))  
*# <list\_iterator object at 0x000001EC8EBC3C18>*

这样类似的输出就说明我们想要判断的对象是可迭代对象，使用iter()函数也是非常方便的。

**迭代器**

之前提到，迭代器是一种可以记住遍历位置的对象，其内部实现了\_\_iter\_\_方法和\_\_next\_\_方法。既然这样，迭代器当然也属于可迭代对象了，只不过在Python中是通过可迭代对象来获取迭代器的。

比如我们在使用for循环来遍历string\_1 = 'Python'字符串的时候，对象string\_1的背后就是存在迭代器的，只是隐式的存在。

下面我们来介绍一下上面涉及到的两个方法：\_\_next\_\_方法和\_\_iter\_\_方法，这也是标准的迭代器接口存在的两个方法。

**迭代器\_\_iter\_\_()方法**

从迭代器和可迭代对象的角度来看，\_\_iter\_\_方法是迭代器与可迭代对象都要实现的方法，该方法返回的是当前对象迭代器类的实例。直接返回self（即自己本身），以便在应该使用可迭代对象的地方使用迭代器， 例如在 for 循环中。

**迭代器next()方法**

返回迭代过程中的每一步，如下一个可用的元素， 如果没有元素了， 抛出 StopIteration异常。

**这里抛出一个问题：python的迭代器为什么一定要实现\_\_iter\_\_方法？**

简单的解释：如果迭代器不实现\_iter\_方法的话，那么其就不能迭代了，也就不是可迭代的对象。从另外一个角度上来看，在迭代器返回元素的过程中是先调用\_\_iter\_\_来获取迭代对象，然后再调用可迭代对象的\_\_next\_\_来实现的。

最后我们来看看实际中我们怎么使用迭代器：

class Fib(object):  
 def \_\_init\_\_(self):  
 self.num\_1, self.num\_2 = 0, 1  
 def \_\_iter\_\_(self):  
 return self  
 def \_\_next\_\_(self):  
 self.num\_1, self.num\_2 = self.num\_2, self.num\_1 + self.num\_2  
 if self.num\_1 > 100:  
 raise StopIteration()  
 return self.num\_1  
  
for each in Fib():  
 print (each)

稍微解释一下，每一次for循环的时候会调用Fib()中\_\_iter\_\_的方法，这个方法返回一个可迭代对象，然后调用该迭代对象的next()方法拿到返回值，直到遇到StopIteration错误时退出循环，以上就是使用迭代器实现的Fib()函数。

实际上，我们可以使用迭代器来实现Fib()函数的，如下：

def Fib(n):  
 num\_1, num\_2 = 0, 1  
 while num\_1 < 100:  
 print(num\_1)  
 num\_1, num\_2 = num\_2, num\_1 + num\_2  
Fib(2)

当然了，如果我们想将next()得到的值反向输出，我们就可以使用reversed()创建

反序访问的迭代器。

我们分步骤看看next()会的实际效果：

S = **'python'**S\_1 = iter(target)  
print(next(S\_1)) *#p*print(next(S\_1)) *#y*print(next(S\_1)) *#t*print(next(S\_1)) *#h*print(next(S\_1)) *#o*print(next(S\_1)) *#h*print(next(S\_1)) *# Traceback (most recent call last)...StopIteration*

1. **生成器**

百度百科中对生成器(generator)的一个解释为:生成器是一次生成一个值的特殊类型的函数，可以将其视为可恢复函数，调用该函数将返回一个可用于生成连续目标值的生成器，在函数的执行过程中，yield语句会把你需要的值返回给调用生成器的地方，然后退出函数，下次调用生成器函数的时候又从上次中断的地方开始执行，而生成器内的所有变量参数都会被保存下来供下次调用的时候使用。

**关键词yield**

作用: yield 的作用就是把一个函数变成一个生成器，实现了yield 的函数不再是一个普通函数，Python的解释器会将其视为一个生成器。或者说yield是一个具有return功能的关键字，只是含有yield的函数返回的是个生成器。

一般情况下，包含yield 的函数我们成为生成器函数，一般包含return 的函数我们就认为是一般函数。

我们来看一个例子：

def myGenerator() :  
 mylist = range(1,3)  
 for i in mylist :  
 yield i\*i  
  
test\_generator = myGenerator()  
print(**'test\_generator:'**,test\_generator)  
for i in test\_generator:  
 print(i)

上述输出：

test\_generator: <generator object myGenerator at 0x000001FACC866DB0>

1

4

可以看出，对象test\_generator是一个generator。唯一要注意的是：当我们的代码运行到test\_generator = myGenerator()的时候，其实函数内部由于存在yield代码并不会马上就执行 ，这个函数只是返回一个生成器对象。然后我们使用for循环来读取生成器对象中的值。

那么使用生成器有什么好处呢？我觉得有两个好处：

1. 精简代码量
2. 提高代码性能

**精简代码量**

比如我们之前在上述使用迭代器实现的Fib的前N数，我们使用的代码量还是很多的，如果我们使用yield的话就非常舒服了：

def Fib(n):  
 num\_1, num\_2 = 0, 1  
 while num\_1 < n:  
 yield num\_1  
 num\_1, num\_2 = num\_2, num\_1 + num\_2  
  
for n in Fib(4):  
 print(n)

注意的是我们仍然使用yiel构造出了一个生成器函数,通过for循环来进行数据的提取。

**提高代码性能**

第二个优点就是提高性能，这也是解决我们上次推文中提及到的问题的关键。我们试想一下在N（假设N>50）个数据中，这样取数：任意选2个、任意选3个

…任意选N个,把所有这样选择的数都加起来，我们知道这个数量是非常大的，所有把所有的组合存入列表丢到内存中，这很有可能将内存撑爆的，因此我们有必要来优化性能。

那么使用yield的生成器实际上是怎么优化性能的呢？我们来看看怎么实现的：

import itertools  
def onegenerate(string\_index):  
 *#for number\_1 in range(2, len(string\_index) + 1):* for number\_1 in range(len(string\_index), 1, -1):  
 iter = itertools.combinations(string\_index, number\_1)  
 iter\_list = list(iter)  
 for number\_2 in range(len(iter\_list)):  
 each = iter\_list[number\_2]  
 each\_list = list(each)  
 yield each\_list  
  
list\_test = [i for i in range(1,21)]  
  
mygene = onegenerate(list\_test)  
print(mygene)  
total = 0  
for n in mygene: *#迭代器里面的内容  
 # print(n)* total = total + 1  
print(total)

大家可以试着运行一下输出是什么，这里就不告诉大家输出是什么了。

输出的结果是反向的，我们可以使用reversed()进行反转的。

写到这里，还有一个实际项目中用到的技巧，假设我们电脑内存是8G，我们遇到了16G大小的数据，这怎么办呢？没错，我们可以使用分块读取的技巧。也就是使用答案是使用 yield 构造 generator，可以这样：

def chunk\_readtest(file, CHUNK\_SIZE):  
 while True:  
 chunk\_data = file.read(CHUNK\_SIZE)  
 if chunk:  
 yield chunk\_data  
 else:  
 return   
  
f = open(**'yourdata'**)  
*# chunk\_size为每次读入数据的块大小*CHUNK\_SIZE = 2048  
for chunk in chunk\_readtest(f,CHUNK\_SIZE):  
 *# 使用数据的函数* use\_chunk(chunk)

这个技术在我们处理大数据的时候非常有用的哦。

1. **总结**

以上就是迭代器和生成器的全部内容，其实它们也不是很难，个人认为在实际的项目中生成器用的还是多一点，大家可以将重点放在生成器的使用上。对于迭代器，大家也要掌握好，因为它能让我们明白为什么可以使用for循环来进行迭代取值。