

12. 前方高能-生成器和生成器表达式

本节主要内容:

1. 生成器和生成器函数
2. 列表推导式

一. 生成器

什么是生成器. 生成器实质就是迭代器.
在python中有三种方式来获取生成器:

1. 通过生成器函数
2. 通过生成器表达式来实现生成器

首先, 我们先看一个很简单的函数:

```
def func():  
    print("111")  
    return 222  
  
ret = func()  
print(ret)
```

结果:
111
222

将函数中的return换成yield就是生成器

```
def func():  
    print("111")  
    yield 222  
  
ret = func()  
print(ret)
```

结果:
<generator object func at 0x10567ff68>

运行的结果和上面不一样. 为什么呢. 由于函数中存在着yield. 那么这个函数就是一个生成器函数. 这个时候. 我们再执行这个函数的时候. 就不再是函数的执行了. 而是获取这个生成器. 如何使用呢? 想想迭代器. 生成器的本质是迭代器. 所以. 我们可以直接执行__next__()来执行

以下生成器.

```
def func():  
    print("111")  
    yield 222  
  
gener = func() # 这个时候函数不会执行. 而是获取到生成器  
ret = gener.__next__() # 这个时候函数才会执行. yield的作用和return一样. 也是返回数据  
print(ret)  
结果:  
111  
222
```

那么我们可以看到, yield和return的效果是一样的. 有什么区别呢? yield是分段来执行一个函数. return呢? 直接停止执行函数.

```
def func():  
    print("111")  
    yield 222  
    print("333")  
    yield 444  
  
gener = func()  
ret = gener.__next__()   
print(ret)  
ret2 = gener.__next__()   
print(ret2)  
ret3 = gener.__next__() # 最后一个yield执行完毕. 再次__next__()程序报错, 也就是说. 和return无关了.  
print(ret3)  
  
结果:  
111  
Traceback (most recent call last):  
222  
333  
  File "/Users/sylar/PycharmProjects/oldboy/iterator.py", line 55, in  
<module>  
444  
    ret3 = gener.__next__() # 最后一个yield执行完毕. 再次__next__()程序报错, 也就是说. 和return无关了.  
StopIteration
```

当程序运行完最后一个yield. 那么后面继续进行__next__()程序会报错.

好了生成器说完了. 生成器有什么作用呢? 我们来看这样一个需求. 老男孩向JACK JONES订购10000套学生服. JACK JONES就比较实在. 直接造出来10000套衣服.

```
def cloth():
```

```

lst = []
for i in range(0, 10000):
    lst.append("衣服"+str(i))
return lst
cl = cloth()

```

但是呢, 问题来了. 老男孩现在没有这么多学生啊. 一次性给我这么多. 我往哪里放啊. 很尴尬啊. 最好的效果是什么样呢? 我要1套. 你给我1套. 一共10000套. 是不是最完美的.

```

def cloth():
    for i in range(0, 10000):
        yield "衣服"+str(i)
cl = cloth()
print(cl.__next__())
print(cl.__next__())
print(cl.__next__())
print(cl.__next__())

```

区别: 第一种是直接一次性全部拿出来. 会很占用内存. 第二种使用生成器. 一次就一个. 用多少生成多少. 生成器是一个一个的指向下一个. 不会回去, `__next__()`到哪, 指针就指到哪儿. 下一次继续获取指针指向的值.

接下来我们来看send方法, send和`__next__()`一样都可以让生成器执行到下一个yield.

```

def eat():
    print("我吃什么啊")
    a = yield "馒头"
    print("a=", a)
    b = yield "大饼"
    print("b=", b)
    c = yield "韭菜盒子"
    print("c=", c)
    yield "GAME OVER"

gen = eat()      # 获取生成器
ret1 = gen.__next__()
print(ret1)
ret2 = gen.send("胡辣汤")
print(ret2)
ret3 = gen.send("狗粮")
print(ret3)
ret4 = gen.send("猫粮")
print(ret4)

```

send和`__next__()`区别:

1. send和next()都是让生成器向下走一次
2. send可以给上一个yield的位置传递值, 不能给最后一个yield发送值. 在第一次执行生

成器代码的时候不能使用send()

生成器可以使用for循环来循环获取内部的元素:

```
def func():  
    print(111)  
    yield 222  
    print(333)  
    yield 444  
    print(555)  
    yield 666
```

```
gen = func()  
for i in gen:  
    print(i)
```

结果:

```
111  
222  
333  
444  
555  
666
```

二. 列表推导式, 生成器表达式以及其他推导式

首先我们先看一下这样的代码, 给出一个列表, 通过循环, 向列表中添加1-13 :

```
lst = []  
for i in range(1, 15):  
    lst.append(i)  
print(lst)
```

替换成列表推导式:

```
lst = [i for i in range(1, 15)]  
print(lst)
```

列表推导式是通过一行来构建你要的列表, 列表推导式看起来代码简单. 但是出现错误之后很难排查.

列表推导式的常用写法:

[结果 for 变量 in 可迭代对象]

例. 从python1期到python14期写入列表lst:

```
lst = ['python%s' % i for i in range(1,15)]  
print(lst)
```

我们还可以对列表中的数据进行筛选

筛选模式:

[结果 for 变量 in 可迭代对象 if 条件]

```
# 获取1-100内所有的偶数
lst = [i for i in range(1, 100) if i % 2 == 0]
print(lst)
```

生成器表达式和列表推导式的语法基本上是一样的. 只是把[]替换成()

```
gen = (i for i in range(10))
print(gen)
```

结果:

```
<generator object <genexpr> at 0x106768f10>
```

打印的结果就是一个生成器. 我们可以使用for循环来循环这个生成器:

```
gen = ("麻花藤我第%s次爱你" % i for i in range(10))
for i in gen:
    print(i)
```

生成器表达式也可以进行筛选:

```
# 获取1-100内能被3整除的数
gen = (i for i in range(1,100) if i % 3 == 0)
for num in gen:
    print(num)

# 100以内能被3整除的数的平方
gen = (i * i for i in range(100) if i % 3 == 0)
for num in gen:
    print(num)

# 寻找名字中带有两个e的人的名字
names = [['Tom', 'Billy', 'Jefferson', 'Andrew', 'Wesley', 'Steven',
          'Joe'],
          ['Alice', 'Jill', 'Ana', 'Wendy', 'Jennifer', 'Sherry', 'Eva']]

# 不用推导式和表达式
result = []
for first in names:
    for name in first:
        if name.count("e") >= 2:
            result.append(name)

print(result)

# 推导式
gen = (name for first in names for name in first if name.count("e") >= 2)
for name in gen:
    print(name)
```

生成器表达式和列表推导式的区别:

1. 列表推导式比较耗内存. 一次性加载. 生成器表达式几乎不占用内存. 使用的时候才分配和使用内存
2. 得到的值不一样. 列表推导式得到的是一个列表. 生成器表达式获取的是一个生成器.

举个栗子.

同样一篮子鸡蛋. 列表推导式: 直接拿到一篮子鸡蛋. 生成器表达式: 拿到一个老母鸡. 需要鸡蛋就给你下鸡蛋.

生成器的惰性机制: 生成器只有在访问的时候才取值. 说白了. 你找他要他才给你值. 不找他要. 他是不会执行的.

```
def func():
    print(111)
    yield 222

g = func() # 生成器g
g1 = (i for i in g) # 生成器g1. 但是g1的数据来源于g
g2 = (i for i in g1) # 生成器g2. 来源g1

print(list(g)) # 获取g中的数据. 这时func()才会被执行. 打印111. 获取到222. g完毕.
print(list(g1)) # 获取g1中的数据. g1的数据来源是g. 但是g已经取完了. g1 也就没有数据了
print(list(g2)) # 和g1同理
```

深坑==> 生成器. 要值得时候才拿值.

字典推导式:

根据名字应该也能猜到. 推到出来的是字典

```
# 把字典中的key和value互换
dic = {'a': 1, 'b': '2'}
new_dic = {dic[key]: key for key in dic}
print(new_dic)

# 在以下list中. 从lst1中获取的数据和lst2中相对应的位置的数据组成一个新字典
lst1 = ['jay', 'jj', 'sylar']
lst2 = ['周杰伦', '林俊杰', '邱彦涛']
dic = {lst1[i]: lst2[i] for i in range(len(lst1))}
print(dic)
```

集合推导式:

集合推导式可以帮我们直接生成一个集合. 集合的特点: 无序, 不重复. 所以集合推导式自带去重功能

```
lst = [1, -1, 8, -8, 12]
# 绝对值去重
s = {abs(i) for i in lst}
print(s)
```

总结: 推导式有, 列表推导式, 字典推导式, 集合推导式, 没有元组推导式

生成器表达式: (结果 for 变量 in 可迭代对象 if 条件筛选)

生成器表达式可以直接获取到生成器对象. 生成器对象可以直接进行for循环. 生成器具有惰性机制.

一个面试题. 难度系数5000000000颗星:

```
def add(a, b):
    return a + b

def test():
    for r_i in range(4):
        yield r_i

g = test()

for n in [2, 10]:
    g = (add(n, i) for i in g)

print(list(g))
```

友情提示: 惰性机制, 不到最后不会拿值

这个题要先读一下. 然后自己分析出结果. 最后用机器跑一下.

最后一个知识点: yield from

在python3中提供了一种可以直接把可迭代对象中的每一个数据作为生成器的结果进行返回

```
def gen():
    lst = ["麻花藤", "胡辣汤", "微星牌饼铛", "Mac牌锅铲"]
    yield from lst

g = gen()
for el in g:
    print(el)
```

小坑: yield from是将列表中的每一个元素返回. 所以. 如果写两个yield from 并不会产生交替的效果.

```
def gen():  
    lst = ["麻花藤", "胡辣汤", "微星牌饼铛", "Mac牌锅铲"]  
    lst2 = ["饼铛还是微星的好", "联想不能煮鸡蛋", "微星就可以", "还可以烙饼"]  
    yield from lst  
    yield from lst2  
  
g = gen()  
for el in g:  
    print(el)
```

效果：

麻花藤
胡辣汤
微星牌饼铛
Mac牌锅铲
饼铛还是微星的好
联想不能煮鸡蛋
微星就可以
还可以烙饼