生成器详解

通过列表解析，我们可以创建一个列表．但是由于受到内存的限制，列表容量肯定是有限的．而且如果创建一个包含100万个元素的列表，不仅占用很大的存储空间，然而如果我们仅仅需要访问前面几个元素的话，那后面绝大多数元素占用的空间都白白浪费了．

一．生成器表达式

1．生成器表达式：通列表解析语法，只不过把列表解析的[]换成()．

2．生成器表达式能做的事情列表解析基本都能处理，只不过在需要处理的序列比较大时，列表解析比较费内存．

In [1]:
```
#示例代码1.

import sys
l = [j for j in range(1, 100000)]
i = (j for j in range(1, 100000))

sys.getsizeof(l)  #查看占用内存的数量
```

Out[1]:
```
824464
```

In [2]:
```
sys.getsizeof(i) #查看占用内存的数量
```

Out[2]:
```
88
```

二. 生成器函数

在函数中如果出现了yield关键字, 那么该函数就不再是普通函数, 而是生成器函数.

例子:

生成器函数可以产生一个无限的序列, 而这种无限的序列用列表是没有办法处理的.

In [4]:
```
#示例代码2. 函数返回所有的奇数

def odd():
    n=1
    while True:
        yield n
        n+=2

odd_num = odd()
count = 0
for o in odd_num:
    if count >=5:
        break
    print(o)
    count +=1
```

```
1
3
5
7
9
```

生成器函数中包含有 **iter** 与 **next** 方法, 所以可以直接使用for循环来进行迭代.

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

```
['__class__', '__del__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclas
s__', '__iter__', '__le__', '__lt__', '__name__', '__ne__', '__new__', '__next__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeo
f__', '__str__', '__subclasshook__', 'close', 'gi_code', 'gi_frame', 'gi_running', 'gi_yieldfrom', 'send', 'throw']
```

三. yield与return

例1. 在一个生成器中如果没有return则默认执行到函数完毕时返回StopIteration

In [9]:
```
def f1():
    yield 1
x  = f1()

print(dir(x))
```

```
['__class__', '__del__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclas
s__', '__iter__', '__le__', '__lt__', '__name__', '__ne__', '__new__', '__next__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeo
f__', '__str__', '__subclasshook__', 'close', 'gi_code', 'gi_frame', 'gi_running', 'gi_yieldfrom', 'send', 'throw']
```

In [12]:
```
print(type(x))
```

```
<class 'generator'>
```

In [13]:
```
print(next(x))
```

```
1
```

In [14]:
```
print(next(x))
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-14-ff675a2ee99e> in <module>
----> 1 print(next(x))

StopIteration:
```

例2. 如果在执行过程中遇到return则直接抛出StopIteration异常并终止迭代.

In [16]:
```
def f1():
    yield 'a'
    return
    yield 'b'

x = f1()
print(next(x))
```

```
a
```

In [17]:
```
print(next(x))
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-17-ff675a2ee99e> in <module>
----> 1 print(next(x))

StopIteration:
```

三. 生成器支持的方法

In [18]:
```
l = (j for j in range(1, 10))
```

In [19]:
```
print(dir(l))
```

```
['__class__', '__del__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lt__', '__name__', '__ne__', '__new__', '__next__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'close', 'gi_code', 'gi_frame', 'gi_running', 'gi_yieldfrom', 'send', 'throw']
```

close() 手动关闭生成器函数, 后面的调用会直接返回StopIteration异常

In [20]:
```
#示例代码3.

def f1():
    yield 1
    yield 2
    yield 3

x = f1()
print(next(x))
x.close()  #因为遇到了close所以手动关闭了生成器函数
print(next(x))
```

```
1

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-20-acd68aa7452a> in <module>
      9 print(next(x))
     10 x.close()
---> 11 print(next(x))

StopIteration:
```

send() 接受外部传值根据变量内容返回结果, 实现与生成器的交互; 这是生成器最难理解也是最重要的地方可以用它来实现协程.

In [3]:
```
#示例代码4.

def employee():
    print('我是一名员工!')
    while True:
        print("head")
        task = yield
        print('balala')
        if not task:
            print('今天没活干')
        else:
            if task == 'coding':
                print('我是程序员!')
            elif task == 'devops':
                print('我是运维攻城狮')
            elif task == 'study':
                print('我爱学习')
        print("end")

x = employee()
x.__next__()
x.send('study')
print("##" * 10)
x.send('devops')
print("##" * 10)
x.send('coding')
print("##" * 10)
x.send('devops')
print("##" * 10)
x.send("")
```

```
我是一名员工!
head
balala
我爱学习
end
head
####################
balala
我是运维攻城狮
end
head
####################
balala
我是程序员!
end
head
####################
balala
我是运维攻城狮
end
head
####################
balala
今天没活干
end
head
```

问题：生成器中next和send的运行流程是怎样的?

答：假设有如下代码，对于普通的生成器，第一个next调用，相当于启动生成器，会从生成器函数的第一行代码开始执行，直到第一次执行完yield语句(第4行)后，跳出生成器函数. 然后第二个next调用，进入生成器函数后，从yield语句的下一句语句(第5行)开始执行，然后重新运行到yield语句，执行后，跳出生成器函数. 以此类推.

In [6]:
```
def consumer():
    r = 'here'
    for i in range(3):
        yield r
        r = '200 OK'+ str(i)

c = consumer()
print(c.__next__(), '第一次')
print(c.__next__(), '第二次')
print(c.__next__(), '第三次')
```

```
here 第一次
200 OK0 第二次
200 OK1 第三次
```

了解了next()如何让包含yield的函数执行后，我们再来看另外一个非常重要的函数send(msg)。其实next()和send()在一定意义上作用是相似的，区别是send()可以传递yield表达式的值进去，而next()不能传递特定的值，只能传递None进去。因此，我们可以看做c.next()和c.send(None)的作用是一样的。需要提醒的是，第一次调用时，请使用next()语句或是send(None)。不能使用send发送一个非None的值，否则会出错的，因为没有Python yield语句来接收这个值。下面来着重说明下send执行的顺序。

In [37]:
```
#示例代码5. 一个简单的协程例子

import time

def producer(c):
    c.__next__()
    n = 0
    while n < 5:
        n = n + 1
        print('producer.. %s' %n)
        c.send(n)
        print('producer.. %s over' %n)
    c.close()

def customer():
    import time
    while 1:
        n = yield
        if not n:
            return
        print('customer %s' %n)
        time.sleep(1)
x = customer()
producer(x)
```

```
producer.. 1
customer 1
producer.. 1 over
producer.. 2
customer 2
producer.. 2 over
producer.. 3
customer 3
producer.. 3 over
producer.. 4
customer 4
producer.. 4 over
producer.. 5
customer 5
producer.. 5 over
```

启动生成器，从生成器函数的第一行代码开始执行，直到第一次执行完yield(对应第customer函数第4行)后，跳出生成器函数。这个过程中，n一直没有定义。下面运行到send(1)时，进入生成器函数，注意这里与调用next的不同，这里是从第4行开始执行，把1赋值给n，但是并不执行yield部分，下面继续从yield的下一语句继续执行，然后重新运行到yield语句，执行后，跳出生成器函数。即send和next相比，只是开始多了一次赋值的动作，其他运行流程是相同的。

In [38]:
```
#示例代码6. 向生成器传入异常

def f1():
    while 1:
        try:
            yield 1
            yield 2
            yield 3
        except ValueError:
            print('this is value error')
x = f1()
x.__next__()
```

Out[38]: 1

In [39]:
```
x.__next__()
```

Out[39]: 2

In [40]:
```
x.throw(ValueError) #throw会跳过后续所有的try…直接进入except…; 然后再回到while消耗一个yield所以后面的x.__next__()返回的是2
```

```
this is value error
```

Out[40]: 1

In [41]:
```
x.__next__()
```

Out[41]: 2

In [ ]:
```

```