1.迭代器与可迭代对象的区别

迭代是访问集合元素的一种方式。迭代器是一个可以记住遍历的位置的对象。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

1.1. 可迭代对象

可以直接作用于 for 循环的数据类型有以下几种：

一类是集合数据类型，如 list 、 tuple 、 dict 、 set 、 str 等；

一类是 generator ，包括生成器和带yield的generator function。这

些可以直接作用于 for 循环的对象统称为可迭代对象： Iterable 。

1.2. 判断是否可迭代

可以使用 isinstance() 来判断一个对象是否是 Iterable 对象：

```
In [50]: from collections import Iterable

In [51]: isinstance([], Iterable)
Out[51]: True

In [52]: isinstance({}, Iterable)
Out[52]: True

In [53]: isinstance('abc', Iterable)
Out[53]: True

In [54]: isinstance((x for x in range(10)), Iterable)
Out[54]: True

In [55]: isinstance(100, Iterable)
Out[55]: False
```

1.3. 迭代器

但是生成器不但可以作用于 for 循环，还可以被 next() 函数不断调用并返回下一个值，直到最后抛出 StopIteration 错误表示无法继续返回下一个值。

可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator。

可以使用 isinstance() 判断一个对象是否是 Iterator 对象：

```
In [56]: from collections import Iterator

In [57]: isinstance((x for x in range(10)), Iterator)
```

```
Out[57]: True

In [58]: isinstance([], Iterator)
Out[58]: False

In [59]: isinstance({}, Iterator)
Out[59]: False

In [60]: isinstance('abc', Iterator)
Out[60]: False

In [61]: isinstance(100, Iterator)
Out[61]: False
```

1.4. iter()函数

生成器都是 Iterator 对象，但 list 、 dict 、 str 虽然是 Iterable ， 却不是

Iterator 。想把 list 、 dict 、 str 等 Iterable 变成 Iterator 可以使用 iter() 函数：

```
In [62]: isinstance(iter([]), Iterator)
Out[62]: True

In [63]: isinstance(iter('abc'), Iterator)
Out[63]: True
```

总结

- 凡是可作用于 for 循环的对象都是 Iterable 类型；
- 凡是可作用于 next() 函数的对象都是 Iterator 类型
- 集合数据类型如 list 、 dict 、 str 等是 Iterable 但不是 Iterator ， 不过可以通过 iter() 函数获得一个 Iterator 对象。

其实Python的for循环本质上就是通过不断调用next()函数实现的

例如：

```
for x in [1, 2, 3, 4, 5]:
    pass
```

就等价于

```
# 首先获得Iterator对象:
it = iter([1, 2, 3, 4, 5])
# 循环:
while True:
    try:
        # 获得下一个值:
        x = next(it)
    except StopIteration:
        # 遇到StopIteration就退出循环
        break
```

2.生成器

2.1. 什么是生成器

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，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，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器：generator。

2.2. 创建生成器方法:生成器表达式

要创建一个生成器，有很多种方法。第一种方法很简单，只要把一个列表生成式的 [ ] 改成 ( )就行了。

```
In [1]: M = [ x*3 for x in range(5)]

In [2]: M
Out[3]: [0, 3, 6, 9, 12]

In [4]: N = ( x*2 for x in range(5))

In [5]: N
Out[6]: <generator object <genexpr> at 0x7f626c132db0>

In [7]:
```

创建 M 和 N 的区别仅在于最外层的 [ ] 和 ( )， M 是一个列表，而 N 是一个生成器。我们可以直接打印出L的每一个元素，但我们怎么打印出G的每一个元素呢？如果要一个一个打印出来，可以通过 next() 函数获得生成器的下一个返回值：

```
In [7]: next(N)
Out[8]: 0
```

```
In [9]: next(N)
Out[9]: 3

In [10]: next(N)
Out[10]: 6

In [11]: next(N)
Out[11]: 9

In [12]: next(N)
Out[12]: 12

In [13]: next(N)
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-24-380e167d6934> in <module>()
----> 1 next(N)

StopIteration:

In [14]:
In [15]: N = ( x*3 for x in range(5))

In [16]: for x in N:
   ....:     print(x)
   ....:
0
3
6
9
12

In [17]:
```

生成器保存的是算法，每次调用 next(N) ，就计算出 N 的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration 的异常。当然，这种不断调用 next() 实在是太不合理了，正确的方法是使用 for 循
环，因为生成器也是可迭代对象。所以，我们创建了一个生成器后，基本上永远不会调用 next() ，而是通过 for 循环来迭代它，并且不需要关心 StopIteration 异常。

2.3. 创建生成器方法:生成器函数

generator非常强大。如果推算的算法比较复杂，用类似列表生成式的 for 循环无法实现，这个时候我们可以用函数来实现。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

```
In [28]: def fn(times):
```

```
   ....:     n = 0
   ....:     a,b = 0,1
   ....:     while n<times:
   ....:         print(b)
   ....:         a, b = b, a+b
   ....:         n+=1
   ....:     return 'done'
   ....:

In [29]: fn(5)
1
1
2
3
5
Out[29]: 'done'
```

仔细观察，可以看出，fn函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

也就是说，上面的函数和generator仅一步之遥。要把fn函数变成generator，只需要把print(b)改为yield b就可以了：

```
In [30]: def fn(times):
   ....:     n = 0
   ....:     a,b = 0,1
   ....:     while n<times:
   ....:         yield  b
   ....:         a,b = b,a+b
   ....:         n+=1
   ....:     return 'done'
   ....:

In [31]: F = fn(5)

In [32]: next(F)
Out[32]: 1

In [33]: next(F)
Out[33]: 1

In [34]: next(F)
Out[34]: 2

In [35]: next(F)
Out[35]: 3

In [36]: next(F)
Out[36]: 5

In [37]: next(F)
---------------------------------------------------------------------------
```

```
StopIteration                             Traceback (most recent call last)
<ipython-input-37-8c2b02b4361a> in <module>()
----> 1 next(F)

StopIteration: done
```

在上面fn 的例子，我们在循环过程中不断调用 yield ， 就会不断的中断。所以要给循环设置一个条件来退出循环，不然的话就会产生一个无限数列出来。

同样的，把函数改成generator后，我们基本上从来不会用 next() 来获取下一个返回值，也是直接使用 for 循环来迭代：

```
In [38]: for n in fn(5):
   ....:     print(n)
   ....:
1
1
2
3
5

In [39]:
```

但是还有一个问题：用for循环调用generator时，你会发现你拿不到generator的return语句的返回值。如果想要拿到返回值，必须要捕获StopIteration错误，返回值包含在StopIteration的value中：

```
In [39]: g = fib(5)

In [40]: while True:
   ....:     try:
   ....:         x = next(g)
   ....:         print("value:%d"%x)
   ....:     except StopIteration as e:
   ....:         print("生成器返回值:%s"%e.value)
   ....:         break
   ....:
value:1
value:1
value:2
value:3
value:5
生成器返回值:done

In [41]:
```

总结：

生成器是这样一个函数，它记住上一次返回时在函数体中的位置。对生成器函数的第二次（或第 n 次）调用跳转至该函数中间，而上次调用的所有局部变量都保持不变。

生成器不仅“记住”了它数据状态；生成器还“记住”了它在流控制构造（在命令式编程中，这种构造不只是数据值）中的位置。

生成器的特点：

1. 节约内存
2. 迭代到下一次的调用时，所使用的参数都是第一次所保留下的，即是说，在整个所有函数调用的参数都是第一次所调用时保留的，而不是新创建的

3.闭包

3.1. 函数引用

```
def test1():
    print("--- in test1 func----")

#调用函数
test1()

#引用函数
ret = test1

print(id(ret))
print(id(test1))

#通过引用调用函数
ret()
```

运行结果:

```
--- in test1 func----
140212571149040
140212571149040
--- in test1 func----
```

3.2. 什么是闭包

如果在一个函数的内部定义了另一个函数，外部的我们叫他外函数，内部的我们叫他内函数。

在一个外函数中定义了一个内函数，内函数里运用了外函数的临时变量，并且外函数的返回值是内函数的引用。这样就构成了一个闭包。说来说去闭包(closure)其实就是一种代码结构，是函数式编程的重要的语法结构，提高了代码的可重复使用性。

```
# outer是外部函数 a和b都是外函数的临时变量
def outer( a ):
      b = 10
      # inner是内函数
      def inner():
      #在内函数中 用到了外函数的临时变量
          print(a+b)
          # 外函数的返回值是内函数的引用
```

```
    return inner


# 在这里我们调用外函数传入参数5
#此时外函数两个临时变量 a是5，b是10，并创建了内函数，然后把内函数的引用返回给了demo
#    外函数结束的时候发现内部函数将会用到自己的临时变量，这两个临时变量就不会释放，会绑定给这个内部函数
demo = outer(5)
#   我们调用内部函数，看一看内部函数是不是能使用外部函数的临时变量
# demo存了外函数的返回值，也就是inner函数的引用，这里相当于执行inner函数
demo()

demo2 = outer(7)
demo2()
```

运行结果：

```
15
17
```

3.3. 闭包再理解

在闭包内函数中，我们可以随意使用外函数绑定来的临时变量，但是如果我们想修改外函数临时变量数值的时候就会发现有问题！咋回事呢？？？

在基本的python语法当中，一个函数可以随意读取全局数据，但是要修改全局数据的时候有两种方法：

1. global 声明全局变量
2. 全局变量是可变类型数据的时候可以修改

在闭包内函数也是类似的情况。在内函数中想修改闭包变量（外函数绑定给内函数的局部变量）的时候：

1. 在python3中，可以用nonlocal 关键字声明 一个变量， 表示这个变量不是局部变量空间的变量，需要向上一层变量空间找这个变量。
2. 在python2中，没有nonlocal这个关键字，我们可以把闭包变量改成可变类型数据进行修改，比如列表。

```
# outer是外部函数 a和b都是外函数的临时变量
def outer( a ):
    b = 10  # a和b都是闭包变量
    c = [a] # 这里对应修改闭包变量的方法2
    # inner是内函数
    def inner():
        #内函数中想修改闭包变量
        # 方法1 nonlocal关键字声明
        nonlocal  b
        b+=1
        # 方法二，把闭包变量修改成可变数据类型 比如列表
        c[0] += 1
        print(c[0])
        print(b)
    # 外函数的返回值是内函数的引用
    return inner
```

```
demo = outer(5)
demo()
```

运行结果：

```
6
11
```

从上面代码中我们能看出来，在内函数中，分别对闭包变量进行了修改，打印出来的结果也确实是修改之后的结果。以上两种方法就是内函数修改闭包变量的方法。

还有一点需要注意的是：使用闭包的过程中，一旦外函数被调用一次返回了内函数的引用，虽然每次调用内函数，是开启一个函数执行过后消亡，但是闭包变量实际上只有一份，每次开启内函数都在使用同一份闭包变量

```
#coding:utf8
def outer(x):
    def inner(y):
        nonlocal x
        x+=y
        return x
    return inner

a = outer(10)
print(a(1))
print(a(3))
```

运行结果：

```
11
14
```

两次分别打印出11和14，由此可见，每次调用inner的时候，使用的闭包变量x实际上是同一个。

总结：

```
1.闭包似优化了变量，有效的减少了函数所需定义的参数数目，原来需要类对象完成的工作，闭包也可以完成
2.由于闭包引用了外部函数的局部变量，则外部函数的局部变量没有及时释放，消耗内存
```

4.装饰器

装饰器是程序开发中经常会用到的一个功能，用好了装饰器，开发效率如虎添翼，所以这也是Python面试中必问的问题。装饰器是程序开发的基础知识，这个都不会的话就别跟人家说你会Python了。

4.1.什么是装饰器

先来个比喻，内裤可以用来遮羞，但是到了冬天它没法为我们防风御寒，于是聪明的人们发明了长裤，有了长裤后之后就再也不冷了，装饰器就像我们这里说的长裤，在不影响内裤作用的前提下，给我们的身子提供了保暖的功效。

再回到我们的主题，

装饰器本质上是一个Python函数，它可以让其他函数在不需要做任何代码变动的前提下增加额外功能，装饰器的返回值也是一个函数对象,就是运用了闭包的基本原理。装饰器的作用就是为已经存在的对象添加额外的功能

4.2.装饰器(decorator)功能

它经常用于有切面需求的场景，比如：

1. 插入日志
2. 性能测试
3. 事务处理
4. 缓存
5. 权限校验

装饰器是解决这类问题的绝佳设计，有了装饰器，我们就可以抽离出大量与函数功能本身无关的雷同代码并继续重用。

4.3.装饰器详解

再来举个例子：在工作中，你写了一个用户信息功能，我写了一个购买商品功能，一般我们在淘宝买东西都是要先登录之后才能购买商品和查看自己信息，所以现在老板要给我们两个人的功能都加一个登录验证，只有验证登录了才能使用我们的功能，那该怎么办呢，

```
#个人信息功能
def f1():
    print('f1')

#购买商品功能
def f2():
    print('f2')
```

然后你听了之后心想：这不是很简单吗，直接每个功能加一个验证功能就是了，然后你美滋滋的跟老板说交给你解决，于是你这样做了：

```
def f1():
    # 验证登录
    print('f1')

def f2():
    # 验证登录
    print('f2')
```

结果是你被老板开除了,这个时候只剩我一个人了，我想了想于是这样写：

```
def check_login():
    # 验证是否登录
    pass

def f1():
    check_login()
    print('f1')

def f2():
    check_login()
    print('f2')
```

老板看了之后嘴角漏出了一丝的欣慰的笑，语重心长的跟我聊了很多，我到现在都还记得老板是怎么跟我说的：

老板说： 写代码要遵循开放封闭原则，虽然在这个原则是用的面向对象开发，但是也适用于函数式编程，简单来说，它规定已经实现的功能代码不允许被修改，但可以被扩展，即：

- 封闭：已实现的功能代码块
- 开放：对扩展开发

如果将开放封闭原则应用在上述需求中，那么就不允许在函数 f1 、f2的内部进行修改代码，老板就给了我一个实现方案：

```
def login(func):
    def inner():
        # 验证登录
        func()
    return inner

@login
def f1():
    print('f1')

@login
def f2():
    print('f2')
```

对于上述代码，也是仅仅对基础功能代码进行修改，就可以实现在其他人调用函数 f1 f2 之前都进行验证操作，并且我们都无需做任何操作。

4.4. 详解装饰器

单独以f1为例：

```
def login(func):
    def inner():
        # 验证登录
        func()
    return inner

@login
def f1():
    print('f1')
```

执行login函数

@函数名 是python的一种语法糖，执行login函数 ， 就会将 @login 下面的函数作为login函数的参数，即：@login 等价于 login(f1) 所以，内部就会去执行：

```
def inner():
    #验证登录
    f1()     # func是参数，此时 func 等于 f1
return inner# 返回的 inner，inner代表的是函数，非执行函数 ,其实就是将原来的 f1 函数塞进另外一个函数中
```

login的返回值

```
新f1 = def inner():
            #验证登录
             原来f1()
        return inner
```

所以，以后业务部门想要执行 f1 函数时，就会执行 新f1 函数，在新f1 函数内部先执行验证，再执行原来的f1 函数，然后将原来f1 函数的返回值返回给了业务调用者。

如此一来， 即执行了验证的功能，又执行了原来f1函数的内容，并将原f1函数返回值 返回给业务调用着

4.5. 装饰器示例

1): 无参数的函数

```
def test(func):
    def deco():
        print("Call the function %s()"%func.func_name)
        func()
    return deco

@test
def foo():
    print("hello world")

foo()
```

2): 被装饰的函数有参数

```
def test(func):
    def deco(a, b):
        print("Call the function %s()"%func.func_name)
        print(a, b)
        func(a, b)
    return deco

@test
def foo(a, b):
    print(a+b)


foo(3,5)
```

3): 被装饰的函数有不定长参数

```
def test(func):
    def deco(*args, **kwargs):
        print("Call the function %s()"%func.func_name)
        func(*args, **kwargs)
    return deco

@test
def foo(a,b,c):
    print(a+b+c)


foo(3,5,1)
```

4): 装饰器中的*return*

```
def test(func):
    def deco():
        print("Call the function %s()"%func.func_name)
        func()
    return deco


@test
def foo():
    print("hello world")


@test
def fo():



foo()
print(fo())
```

执行结果:

```
Call the function foo()
hello world

Call the function fo()
None
```

如果修改装饰器为return func()：

```
def test(func):
    def deco():
        print("Call the function %s()"%func.func_name)
        return func()
    return deco
```

则运行结果：

```
Call the function foo()
hello world

Call the function fo()
----hello world----
```

- 所以一般情况下为了让装饰器更通用，可以用return

5)：装饰器带参数,在原有装饰器的基础上，设置外部变量

a.被装饰对象无参数：

```
def test(printResult=False):
    def _test(func):
        def   test():
            print('Call the function %s()'%func.func_name)
            if printResult:
                print(func())
            else:
                return func()
        return   test
    return _test

@test(True)
def foo():
    return 'hello world'

@test(False)
def fo():
    return 'hello world'

foo()
print('----------')
fo()
```

运行结果：

```
Call the function foo()
hello world
----------
Call the function fo()
```

b.被装饰对象有参数：

```
def test(printResult=False):
    def _test(func):
        def   test(*args,**kw):
            print('Call the function %s().'%func.func_name)
            if printResult:
                print(func(*args,**kw))
            else:
                return func(*args,**kw)
        return   test
    return _test

@test()
def left(Str,Len):
    return Str[:Len]

@test(True)
def right(Str,Len):
    return Str[:Len]

left('hello world',5)
print('----------')
right('hello world',5)
```

运行结果：

```
Call the function left().
----------
Call the function right().
hello
```

6)：类装饰器

装饰器函数其实是这样一个接口约束，它必须接受一个callable对象作为参数，然后返回一个callable对象。在Python中一般callable对象都是函数，但也有例外。只要某个对象重写了 *call*() 方法，那么这个对象就是callable的。

```
class test():
    def call (self):
        print('call me!')

t = test()
t()
```

```
class tracer:
    def   init  (self,func):
        print("func name is %s"%func. name )
        self.calls = 0
        self.func = func

    def call (self,*args):
        self.calls += 1
        print('call %s to %s' %(self.calls, self.func. name ))
        self.func(*args)
#说明:
#1.   当用tracer来装作装饰器对spam函数进行装饰的时候，首先会创建tracer的实例对象
#    并且会把spam这个函数名当做参数传递到  init  方法中
#    即在  init  方法中的func变量指向了spam函数体
#
#2.  spam函数相当于指向了用tracer创建出来的实例对象
#
#3. 当在使用spam()进行调用时，就相当于让这个对象()，因此会调用这个对象的  call 方法
#
#4. 为了能够在 call 方法中调用原来test指向的函数体，所以在 init 方法中就需要一个实例属性来保存这个函数体的引用
#    所以才有了self.  func = func这句代码，从而在调用  call 方法中能够调用到spam之前的函数体
@tracer
def spam(a, b, c):
    print(a + b + c)
spam(1,2,3)
```

运行结果:

```
func name is spam
call 1 to spam
6
```