python的迭代对象和迭代器

```
1.可以直接使用for循环遍历的数据类型就是迭代对象

#判断该数据类型是否为迭代对象
from collections import Iterable

print(isinstance("hello",Iterable))

2.迭代器,可以被next函数调用,并返回下一个值的对象称为迭代器(iterator)
#判断该数据类型是否是迭代器
from collections import Iterator
print(isinstance("hello",Iterator))#False

print(isinstance((x for x in range(6)),Iterator))#True

```

for ... in ... 遍历的原理

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

mystr="abcdefg"
# i=0
# while i<len(mystr):
#     print(mystr[i])
#     i+=1

for i in mystr:
    print(i)

'''
1.内部调用iter函数,将要遍历的对象作为参数传入
2.等价调用了__iter__()函数
3.然后使用next函数去执行迭代器对象
'''
# print(mystr.__iter__())
# print(iter(mystr))
it=iter(mystr)
print(next(it))

```

生成器介绍

```
生成器本质上也是迭代器，不过它比较特殊。
以 list 容器为例，在使用该容器迭代一组数据时，必须事先将所有数据存储到容器中，才能开始迭代；而生成器却不同，它可以实现在迭代的同时生成元素。
也就是说，对于可以用某种算法推算得到的多个数据，生成器并不会一次性生成它们，而是什么时候需要，才什么时候生成。

不仅如此，生成器的创建方式也比迭代器简单很多，大体分为以下 2 步：
定义一个以 yield 关键字标识返回值的函数；
调用刚刚创建的函数，即可创建一个生成器。
```

生成器

```
def mytest(n):
    i=0
    while i<n:
        yield
        i+=1
#创建生成器
it=mytest(5)
#mytest函数返回时使用yield,而不是return,所以这类函数又叫生成器函数

也可以使用for循环遍历生成器函数
for i in it:
    print(i)
```

写成类方式来实现生成器

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====


def myrange(n):
    index=10
    while index<n:
        yield index
        index+=1


class Maker():
    def __init__(self,n):
        self.n=n
```

```
20
21      def __iter__(self):
22          return myrange(self.n)
23
24  m=Maker(20)
25  for i in m:
26      print(i)
```

生成器推导式

```
1
2   # !/usr/bin/env python
3   # -*- coding:utf-8 -*-
4   # ====#====#====#====
5   # Author:
6   # CreatDate:
7   # Version:
8   # ====#====#====#====
9
10  class Maker():
11      def __init__(self, n):
12          self.n = n
13
14      def __iter__(self):
15          return (i for i in range(self.n))
16
17
18  m = Maker(20)
19  for i in m:
20      print(i)
21
22
```

进阶使用

```
1   #!/usr/bin/env python
2   # -*- coding:utf-8 -*-
3   #====#====#====#====
4   #Author:
5   #CreatDate:
6   #Version:
7   #====#====#====#====
8   def myfunc():
9       v1=yield 1
10      v1=yield 0
11      print("v1=",v1)
12      v2=yield 1
13      print("v2=", v2)
14      v3=yield 2
```

```
    print("v3=", v3)

#send方法就是next()的功能,加上传值给yield,发送的值赋值上次中断时yield表达式的执行结果
k=myfunc()
# print(k.__next__())#1
# print("--------------")
# print(k.__next__())#0
# print("--------------")
# print(k.__next__())#v1=None 1
# print("--------------")
# print(k.__next__())#v2=None 2
# print("--------------")
#
print(k.__next__())
print("--------------")
print(k.send(1))#把send(1)中的1赋值给v1=yield 1
print("--------------")
print(k.send(2))#把send(2)中的2赋值给v1=yield 0
print("--------------")
print(k.__next__())

```

装饰器 介绍

```
@装饰器名称
作用： 给被装饰的对象增加额外的属性或者功能

原理：
1. 装饰器本质上是一个函数(可调用对象)
2. 这个函数的参数是一个函数对象(被装饰的函数)
3. 这个函数的返回值是一个函数对象（基于被装饰的函数添加了额外属性或者功能的函数）
原始语法：
@decorator
def function():
    pass

原始语法：    function = decorator(function)
```

装饰器的基本用法

```
目的,把普通人变为有钱人
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

#因为是myfunc=Okmyfunc(myfunc),那么这个函数返回的是一个函数
```

```
def Okmyfunc(myfunc):
    def OKK():
        myfunc()
        #增加额外的功能
        print("银行存款10位数")
        print("飞机,轮船多艘")
        print("知心朋友超多")
    return OKK


@Okmyfunc   #myfunc=Okmyfunc(myfunc)
def myfunc():
    print("月薪1万左右")
    print("无房子")
    print("无车")
    print("无对象")

myfunc()

```

案例： 统计函数的执行时间

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import time

def sum_times(testfunc):
    def sum_t():
        start=time.time()#这是记录代码运行到这里的时间
        testfunc()
        end=time.time()
        print(end-start)
    return sum_t

@sum_times
def testfunc():
    time.sleep(2)

testfunc()
```

被装饰的函数有返回值

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

```

```
#Author:
#CreatDate:
#Version:
#====#====#====#====
import time

def sum_times(testfunc):
    def sum_t():
        start=time.time()#这是记录代码运行到这里的时间
        testfunc()
        end=time.time()
        return end-start
    return sum_t

@sum_times  #testfunc=sum_times(testfunc)
def testfunc():
    time.sleep(2)


print(testfunc())
```

被装饰的函数存在参数，并且参数的个数不同

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import time

def funcMaker(func):
    def start_maker(*args,**kwargs):
        ret=func(*args,**kwargs)
        return ret+100
    return start_maker



@funcMaker
def myfunc1(a,b):
    time.sleep(2)
    return a+b

print(myfunc1(1,2))


@funcMaker
def myfunc2(a,b,c):
    time.sleep(2)
    return a+b+c
```

```
30
31 print(myfunc2(1,2,3))
```

我还是我吗？

```
1  #!/usr/bin/env python
2  # -*- coding:utf-8 -*-
3  #====#====#====#====
4  #Author:
5  #CreatDate:
6  #Version:
7  #====#====#====#====
8
9  def Makerfunc(func):
10     def mytest():
11         func()
12         print("helh")
13     return  mytest
14
15
16 @Makerfunc
17 def myfunc():
18     print("myfunc")
19
20 print(myfunc.__name__)#没有装饰之前是myfunc,装饰之后是mytest
21
22
```

带参数的装饰器

```
1  def myfunc(
2  ,n):
3      def mymaker():
4          func()
5          if n==1:
6              print("11111")
7          elif n==2:
8              print('22222')
9
10 @myfunc(1) #装饰器带参数
11 def mytest():
12     print("hello")
13
14  上面的代码报错,是因为装饰器只能有一个参数,而这个参数就是函数对象
15
16
17  #!/usr/bin/env python
18 # -*- coding:utf-8 -*-
19 #====#====#====#====
20 #Author:
```

```
#CreatDate:
#Version:
#====#====#====#====

def myfunc(n):
    def mymaker(func):
        def mymaker2():
            func()
            if n==1:
                print("11111")
            elif n==2:
                print('22222')
        return mymaker2
    return mymaker

@myfunc(1) #装饰器带参数
def mytest():
    print("hello")

mytest()
```