讲课内容:

闭包

装饰器

迭代器

生成器

列表推导式,生成器表达式

匿名函数

内置函数(sorted, map, filter, reduce)

一. 闭包

闭包. 其实很简单. 就是内层函数使用了外层函数中的变量, 就是闭包

```
def outer():
    a = 10
    def inner():
        print(a)  # 这个就是闭包
    return inner  # 闭包通常都是返回内层函数
```

闭包有什么用呢. 注意看了. 当我们外部访问了这个outer()函数. 得到的结果就是inner函数

```
a = outer()
```

此时. 我们拿到了一个变量a. 而这个变量a是outer()的返回值. 也就是inner函数. 所以. 我们可以

```
a()  # 此时执行的是inner这个函数
```

由于函数inner的执行时间是在outer()外部. 这就决定了inner执行的时间我们是不确定的. 而变量a是一个局部变量. 正常情况下执行玩儿outer(), 它就没有意义了. 但是, 此时由于inner函数执行时间的不确定. 又必须保证inner能正常执行. python就规定. 闭包函数中使用的变量会常驻于内存. 而且. 在outer()外部. 是无法改变这个值的. 故称: 闭包. 目的有两个: 其一是不许外面改变这个变量. 其二是让这个变量常驻于内存.

闭包的应用: 装饰器

二. 装饰器

装饰器是干嘛的呢? 它是一种固定的语法. 可以让我们在不修改原有函数内部代码的基础上, 给函数增加新的功能.

```
def add():
    pass
def delete():
    pass
def update():
    pass
def search():
    pass
```

此时, 我想给每个函数添加一个新功能. 记录日志. 记录一下. 在xxx时间执行的xxx函数.

```
def add():
    # 记录日志的代码
```

```
    pass
def delete():
    # 记录日志的代码
    pass
def update():
    # 记录日志的代码
    pass
def search():
    # 记录日志的代码
    pass
```

设想一下, 如果我现在想更换日志格式: xxx函数在xxx时间执行了. 你怎么办.

传统办法: 改代码. 修改每个函数. --> 太蠢了. 如果这一段代码有1000次重复. 你要修改1000次.

高级办法: 把记录日志的代码提取成函数. 然后每个函数分别调用.

```

def log():
    # 记录日志的代码
    pass

def add():
    log()
    pass
def delete():
    log()
    pass
def update():
    log()
    pass
def search():
    log()
    pass
```

这样是不是好多了. 我们只要修改log()函数就可以完成我们想要的结果了. 但是, 随着需求的进一步增加. 你会发现你这几个函数没有消停的时候了. 例如, 不论执行增删改查任何操作之前都要进行登录验证.

```
def log():
    # 记录日志的代码
    pass

def add():
    while 1:
        uname = input(">>>")
        pwd = input(">>>")
        if uname == 'alex' and pwd == "123":
            log()
            pass

```

```
            break
        else:
            print("密码错误")

def delete():
    log()
    pass
def update():
    log()
    pass
def search():
    log()
    pass
```

发现没有, 如果继续下去. 你会发现你这个add方法永无宁日. 不停的再改. 而新增的代码可能早就被复杂的而又不属于新增的业务逻辑所污染.

那怎么办呢? 在程序设计上我们的程序要遵循开闭原则

开: 对添加新功能开放

闭: 对修改函数中的源代码封闭.

普通话: 在不修改源代码内部的基础上给函数添加新功能. 这正好契合我们的装饰器.

2.1 装饰器雏形

```
def wrapper(fn):
    def inner():
        """ 在执行fn之前 """
        fn()  # 这里是一个闭包的效果. 外面函数中有一个fn: 局部变量被内层函数使用.
        """ 在执行fn之后 """
    return inner

def add():
    pass
def delete():
    pass
def update():
    pass
def search():
    pass

```

上面的wrapper就是一个装饰器. 其实看起来没什么特别的, 就是一个闭包而已. 那这东西怎么用呢?

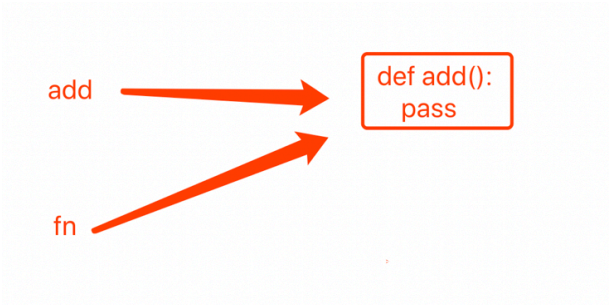
```
add = wrapper(add)
add()
```

其他的都不看. 就看第一行代码. add = wrapper(add) 把add函数作为参数传递给wrapper. 那么wrapper()中的fn就是外面的add函数. 然后wrapper返回inner. 此时注意了. add这个变量被修改. 重新指向inner这个函数.

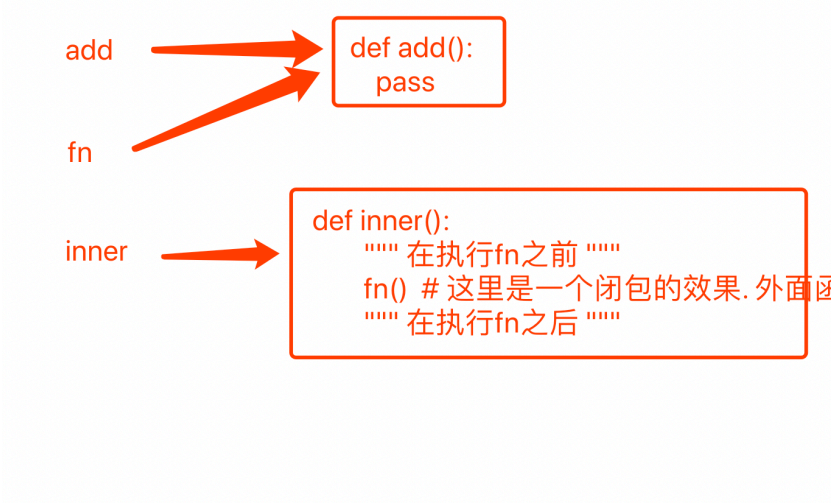
那么. 注意了. 此时我们用add()执行的时候. 实际上执行的是inner()这个函数. 而inner中执行的是fn(). fn是原来的add() 饶了一大圈. 执行的还是原来的add() .

如果不理解, 看图(画图人的美术功底极差, 凑合看吧)

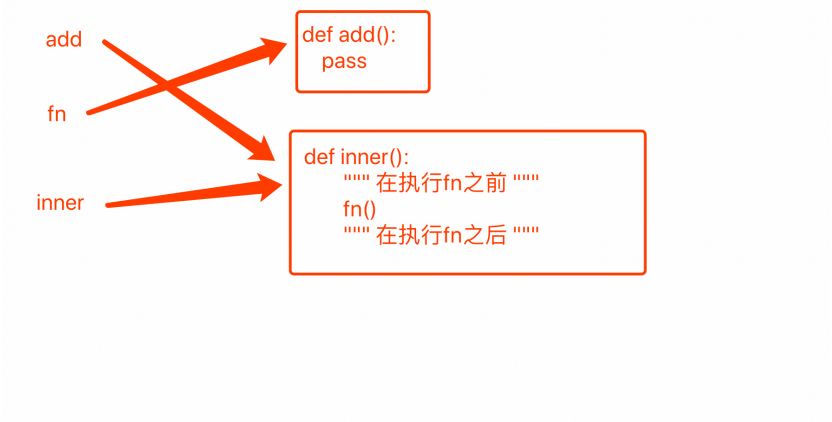
最开始, 在访问wrapper()的一瞬间是这样的:



然后. 内存中产生inner() 并返回inner



再然后. wrapper返回的inner重新赋值给add



此时如果执行 add() 我们能看到, 实际上再执行inner() , 而inner中访问的是fn(), fn函数恰恰是原来的add(). 正好饶了一圈.

那么, 绕这么大一圈的好处和作用是什么呢. 注意. 我们可以在inner函数中 访问fn()之前和之后加入你想加入的任何代码. 而没有改变原来的add()

```
def wrapper(fn):
    def inner():
        """ 在执行fn之前 """
```

```
        while 1:
        uname = input(">>>")
        pwd = input(">>>")
        if uname == 'alex' and pwd == "123":
            log()
            fn()
            break
        else:
            print("密码错误")
        """ 在执行fn之后 """
    return inner

def add():
    pass
def delete():
    pass
def update():
    pass
def search():
    pass
add = wrapper(add)
add()
```

OK. 然后. 还有一个问题. add = wrapper(add)看着是真的难受. 不光你们看着不爽. python的作者也不爽. 怎么办呢. python提供了一种语法糖. 可以帮助我们简化这句话

```
@wrapper   # 相当于add = wrapper(add)
def add():
    pass

```

@后面加上装饰器的名字(装饰器函数名) 就是语法糖. 相当于写了一个add = warpper(add)

这就是装饰器的雏形. 接下来. 我们要介绍一些概念了.

在上个例子中. wrapper被称为装饰器. add和fn被称为被装饰的函数(目标函数).

2.2 被装饰的函数如果带参数怎么办?

上代码

```
def wrapper(fn):
    def inner():
        """ 在执行fn之前 """
        fn()
        """ 在执行fn之后 """
    return inner

@wrapper
def play_game(username, password):
```

```
10      print(username, password)
11
12  play_game("alex", "dsb")
```

此时, 我们发现程序报错了. 为什么呢?

注意. 我们在12行代码 执行的其实不是原来的play_game() 而是inner, 而inner又没有参数. 此时不符合传参的定义. 那怎么办呢? 我们需要给inner添加参数. 并传递给fn()

```
1  def wrapper(fn):
2      def inner(username, password):
3          """ 在执行fn之前 """
4          fn(username, password)
5          """ 在执行fn之后 """
6      return inner
```

那如果被装饰的函数有很多参数呢. 怎么办. 我们之前学过一个*args, **kwargs. 无敌传参

```
1  def wrapper(fn):
2      def inner(*args, **kwargs):
3          """ 在执行fn之前 """
4          fn(*args, **kwargs) # 把接收到的参数打散再发出去
5          """ 在执行fn之后 """
6      return inner
```

一定看清楚了.在inner()的小括号里写的是形参. 表示可以接受各种参数

在fn()位置的*args和**kwargs是实参. 这里表示的是把args, 和kwargs打散再发送出去

2.3 被装饰的函数如果带返回值怎么办

如果我们原来的函数带有返回值怎么办, 上代码

```
1  def wrapper(fn):
2      def inner(*args, **kwargs):
3          """ 在执行fn之前 """
4          fn(*args, **kwargs)
5          """ 在执行fn之后 """
6      return inner
7
8  @wrapper
9  def play_game(username, password):
10      print(username, password)
11      return "倚天屠龙剑"
12
13 ret = play_game("alex", "dsb")
14 print(ret) # None
```

为什么是None呢. 注意. 由于play_game函数被装饰器装饰过了. 那么此时. 我们执行play_game实际上执行的是inner函数. 而真正执行play_game的位置是在fn()这里. 而fn()这里并没有接收返回值. inner也没有返回值.

那么对于外界访问inner的时候自然没有返回值了.

怎么办?

```
1  def wrapper(fn):
```

```
    def inner(*args, **kwargs):
        """ 在执行fn之前 """
        ret = fn(*args, **kwargs)
        """ 在执行fn之后 """
        return ret  # 把fn执行之后的结果返回
    return inner

@wrapper
def play_game(username, password):
    print(username, password)
    return "倚天屠龙剑"

ret = play_game("alex", "dsb")
print(ret)  # 倚天屠龙剑
```

解决问题.

2.4 通用装饰器写法(背下来. 记住)

OK.到此位置. 我们得到了这样一种装饰器的写法

```
def wrapper(fn):
    def inner(*args, **kwargs):
        """ 在执行fn之前 """
        ret = fn(*args, **kwargs)
        """ 在执行fn之后 """
        return ret  # 把fn执行之后的结果返回
    return inner
```

这就是通用装饰器的写法. 这个要求各位老铁记住. 以后只要是想写装饰器了. 先把这段代码罗出来. 然后再去写自己的业务

此处应该去做做练习1,2,3

2.5 带参数的装饰器(了解)

有时候我们可能需要通过参数来控制装饰器装饰的效果. 比如, 我现在想玩游戏开挂. 但是呢. 好多外挂在我面前, 我想自由切换怎么办? 难道要写很多个装饰器么. NO, 我们可以通过参数来控制装饰器内部的执行流程

```
def gua(fn):
    def inner(*args, **kwargs):
        ret = fn(*args, **kwargs)
        return ret
    return inner

@gua
def play():
    pass
play()
```

此时我们只是开挂了. 开什么挂还不知道. 这里就可以选择使用带参数的装饰器了

```
def gua_bi(kind_of_gua):
```

```
    def gua(fn):
        def inner(*args, **kwargs):
            print("我要使用%s" % kind_of_gua)
            ret = fn(*args, **kwargs)
            return ret
        return inner
    return gua  # 注意这句话. 返回的是一个装饰器

@gua_bi("科比")
def play_1():
    pass
@gua_bi("独行侠")
def play_2():
    pass
play_1()
play_2()
```

此时注意了. @gua_bi("科比") 这句话要拆开来看. 先执行的是后半段. gua_bi("科比") 这就是一个普通函数的调用. 执行完这个函数. 返回的是gua. 再和前面的@组合. 正好是@gua. 还是原来装饰器的样子

2.6 同一个函数被多个装饰器装饰

```
def wrapper1(fn):
    def inner(*args, **kwargs):
        print("before wrapper1")
        ret = fn(*args, **kwargs)
        print("after wrapper1")
        return ret
    return inner


def wrapper2(fn):
    def inner(*args, **kwargs):
        print("before wrapper2")
        ret = fn(*args, **kwargs)
        print("after wrapper2")
        return ret
    return inner


def wrapper3(fn):
    def inner(*args, **kwargs):
        print("before wrapper3")
        ret = fn(*args, **kwargs)
        print("after wrapper3")
        return ret
    return inner

```

```
27
28 @wrapper1
29 @wrapper2
30 @wrapper3
31 def func():
32     print("target function")
33
34 func()
35
36 # 结果
37 before wrapper1
38 before wrapper2
39 before wrapper3
40 target function
41 after wrapper3
42 after wrapper2
43 after wrapper1
```

看结果应该就能明白. wrapper1装饰的是warpper2装饰的结果.wrapper2装饰的是wrapper3装饰的结果..... 以此类推. 最接近目标函数的是wrapper3. 所以目标函数之前和之后执行的是wrapper3. 然后外面套上一层wrapper2. 最后wrapper1

3. 迭代器

我们之前一直在用可迭代对象进行迭代操作. 那么到底什么是可迭代对象. 本小节主要讨论可迭代对象. 首先我们先回顾一下目前我们所熟知的可迭代对象有哪些:

str, list, tuple, dict, set. 那为什么我们可以称他们为可迭代对象呢? 因为他们都遵循了可迭代协议. 什么是可迭代协议. 首先我们先看一段错误代码:

```
1 # 对的
2 s = "abc"
3 for c in s:
4     print(c)
5
6 # 错的
7 for i in 123:
8     print(i)
9
10 结果:
11 Traceback (most recent call last):
12   File "/Users/sylar/PycharmProjects/oldboy/iterator.py", line 8, in <module>
13     for i in 123:
14 TypeError: 'int' object is not iterable
15
```

注意看报错信息中有这样一句话. 'int' object is not iterable . 翻译过来就是整数类型对象是不可迭代的. iterable表示可迭代的. 表示可迭代协议. 那么如何进行验证你的数据类型是否符合可迭代协议. 我们可以通过dir函数来查看类中定义好的所有方法.

```
1 s = "我的哈哈哈"
```

```
print(dir(s))        # 可以打印对象中的方法和函数
print(dir(str))      # 也可以打印类中声明的方法和函数
```

在打印结果中. 寻找__iter__ 如果能找到. 那么这个类的对象就是一个可迭代对象.

```
print("__iter__" in dir(s)) # True

```

我们发现在字符串中可以找到__iter__. 继续看一下list, tuple, dict, set

```
print("__iter__" in dir(tuple)) # True
print("__iter__" in dir(list)) # True
print("__iter__" in dir(open("护士少妇嫩模.txt")))  # 文件对象 # True
print("__iter__" in dir(set)) # True
print("__iter__" in dir(dict)) # True
print("__iter__" in dir(int)) # False
```

我们发现这几个可以进行for循环的东西都有__iter__函数, 包括range也有. 可以自己试一下.

这是查看一个对象是否是可迭代对象的第一种办法. 我们还可以通过isinstence()函数来查看一个对象是什么类型的

```
l = [1,2,3]
l_iter = l.__iter__()
from collections import Iterable
from collections import Iterator
print(isinstance(l,Iterable))    #True
print(isinstance(l,Iterator))    #False
print(isinstance(l_iter,Iterator))    #True
print(isinstance(l_iter,Iterable))    #True
```

综上. 我们可以确定. 如果对象中有__iter__函数. 那么我们认为这个对象遵守了可迭代协议. 就可以获取到相应的迭代器. 这里的__iter__是帮助我们获取到对象的迭代器. 我们使用迭代器中的__next__()来获取到一个迭代器中的元素. 那么我们之前讲的for的工作原理到底是什么? 继续看代码

```
s = "我爱北京天安门"
c = s.__iter__()    # 获取迭代器
print(c.__next__())     # 使用迭代器进行迭代. 获取一个元素    我
print(c.__next__())     # 爱
print(c.__next__())     # 北
print(c.__next__())     # 京
print(c.__next__())     # 天
print(c.__next__())     # 安
print(c.__next__())     # 门
print(c.__next__())     # StopIteration
```

for循环的机制:

```
for i in [1,2,3]:
    print(i)
```

使用while循环+迭代器来模拟for循环(必须要掌握)

```
lst = [1,2,3]
lst_iter = lst.__iter__()
while True:
```

```
    try:
        i = lst_iter.__next__()
        print(i)
    except StopIteration:
        break

```

list可以一次性把迭代器中的内容全部拿空. 并装载在一个新列表中

```
s = "我要吃饭, 你吃不吃".__iter__()
print(list(s)) # ['我', '要', '吃', '饭', ',', ' ', '你', '吃', '不', '吃']
```

总结:

Iterable: 可迭代对象. 内部包含__iter__()函数

Iterator: 迭代器. 内部包含__iter__() 同时包含__next__().

迭代器的特点:

1. 节省内存.
2. 惰性机制
3. 不能反复, 只能向下执行.

4. 生成器

生成器的本质就是迭代器. 在python中有两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. 由于函数中存在了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
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ret
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
```

5. 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.

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
```

```
4
5 g = gen()
6 for el in g:
7 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.

```
1 def gen():
2     lst = ["麻花藤", "胡辣汤", "微星牌饼铛", "Mac牌锅铲"]
3     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"]
4     yield from lst
5     yield from lst2
6
7
8 g = gen()
9 for el in g:
10     print(el)
11
12 效果:
13
14 麻花藤
15 胡辣汤
16 微星牌饼铛
17 Mac牌锅铲
18 饼铛还是微星的好
19 联想不能煮鸡蛋
20 微星就可以
21 还可以烙饼
```

6. 匿名函数

为了解决一些简单的需求而设计的一句话函数

```
1 # 计算n的n次方
2 def func(n):
3     return n**n
4 print(func(10))
5
6 f = lambda n: n**n
7 print(f(10))
```

lambda表示的是匿名函数. 不需要用def来声明, 一句话就可以声明出一个函数

语法:

函数名 = lambda 参数: 返回值

注意:

1. 函数的参数可以有多个. 多个参数之间用逗号隔开
2. 匿名函数不管多复杂. 只能写一行, 且逻辑结束后直接返回数据
3. 返回值和正常的函数一样, 可以是任意数据类型

匿名函数并不是说一定没有名字. 这里前面的变量就是一个函数名. 说他是匿名原因是我们通过__name__查看的时候是没有名字的. 统一都叫lambda. 在调用的时候没有什么特别之处. 像正常的函数调用即可

7. 内置函数(sorted, map, filter, reduce)

7.1. sorted

排序函数.

语法: sorted(Iterable, key=None, reverse=False)

Iterable: 可迭代对象

key: 排序规则(排序函数), 在sorted内部会将可迭代对象中的每一个元素传递给这个函数的参数. 根据函数运算的结果进行排序

reverse: 是否是倒叙. True: 倒叙, False: 正序

```
lst = [1,5,3,4,6]
lst2 = sorted(lst)
print(lst)  # 原列表不会改变
print(lst2) # 返回的新列表是经过排序的

dic = {1:'A', 3:'C', 2:'B'}
print(sorted(dic))  # 如果是字典. 则返回排序过后的key
```

和函数组合使用

```
# 根据字符串长度进行排序
lst = ["麻花藤", "冈本次郎", "中央情报局", "狐仙"]

# 计算字符串长度
def func(s):
    return len(s)

print(sorted(lst, key=func))
```

和lambda组合使用

```
# 根据字符串长度进行排序
lst = ["麻花藤", "冈本次郎", "中央情报局", "狐仙"]

# 计算字符串长度
def func(s):
    return len(s)

print(sorted(lst, key=lambda s: len(s)))

lst = [{"id":1, "name":'alex', "age":18},
       {"id":2, "name":'wusir', "age":16},
       {"id":3, "name":'taibai', "age":17}]
# 按照年龄对学生信息进行排序
print(sorted(lst, key=lambda e: e['age']))
```

7.2. filter()

筛选函数

语法: filter(function. Iterable)

function: 用来筛选的函数. 在filter中会自动的把iterable中的元素传递给function. 然后根据function返回的True或者False来判断是否保留此项数据

Iterable: 可迭代对象

```
lst = [1,2,3,4,5,6,7]
ll = filter(lambda x: x%2==0, lst)    # 筛选所有的偶数
print(ll)
print(list(ll))

lst = [{"id":1, "name":'alex', "age":18},
       {"id":2, "name":'wusir', "age":16},
       {"id":3, "name":'taibai', "age":17}]

fl = filter(lambda e: e['age'] > 16, lst)   # 筛选年龄大于16的数据
print(list(fl))
```

7.3.map()

映射函数

语法: map(function, iterable) 可以对可迭代对象中的每一个元素进行映射. 分别取执行function

计算列表中每个元素的平方 ,返回新列表

```
def func(e):
    return e*e

mp = map(func, [1, 2, 3, 4, 5])
print(mp)
print(list(mp))
    改写成lambda
print(list(map(lambda x: x * x, [1, 2, 3, 4, 5])))
    计算两个列表中相同位置的数据的和
# 计算两个列表相同位置的数据的和
lst1 = [1, 2, 3, 4, 5]
lst2 = [2, 4, 6, 8, 10]
print(list(map(lambda x, y: x+y, lst1, lst2)))
```

7.4. reduce()

reduce和map正好相反. map是把数据分散出去. reduce是把数据收回来.

语法:

reduce(function, iterable, initial)

function: reduce函数会自动把可迭代对象中的每一个拿出来传递给function, 要求funciton接收两个参数.

iterable: 可迭代对象

initial: 初始值. 默认是None.

工作流程: 如果initial是None. 则自动把前两个数据传递给function. 然后通过function计算之后会得到一个结果. 把这个结果和下一个数据继续传递给function. 直到所有数据都走一遍. 而如果initial不为None, 则直接把initial和第一个

数据传递给function. 后面以此类推

```
from functools import reduce

lst = [1, 2, 3, 4, 5]

def func(x, y):
    print(x, y)
    return x + y


r = reduce(func, lst)  # 1+2+3+4+5
print(r)


r = reduce(lambda x, y: x + y, lst)  # 1+2+3+4+5
print(r)

```

8. 三元表达式和递归

简单提一提. 这两个知识点了解一下即可

1. 三元表达式.

    各大语言都有三元表达式. 目的是把简单的if判断写成一行.

    语法:

        结果1 if 条件 else 结果2

    过程:

        判断条件是否为真, 如果真, 返回结果1, 否则返回结果2

```
a = input(">>>")
b = input(">>>")
c = a if a > b else b
print(c)
```

    跑两遍, 思考一下. 你就知道咋回事了

2. 递归

    递归, 用好了就是神, 用不好.....呵呵

    递归: 函数自己调用自己

```
def func():
    print("我爱你")
    func()
func()
```

    讲道理, 不出意外, 这个函数会一直执行下去. 但是. 在python中有一个规定. 函数不可以无限的调用下去. python中规定了默认递归最大深度(最多你能调用多少层)

```
import sys
print(sys.getrecursionlimit())  # 递归最大深度 1000
```

    这个数值是可以改的. 请不要这么做. 存在即合理.

    为什么python不让无限的去调用呢? 因为每次调用一个函数. 都需要开辟一个内存. 如果你无限的这么访问下

去. 内存容易受不鸟. 第二就是如果1000次递归你还不能完成你的操作. 百分之99.999是你程序代码逻辑不正常导致的.

总结:

```
闭包: 内层函数对外层函数的变量的使用叫闭包.
作用: 让一个变量常驻内存. 并保护该变量不被侵害
def outer():
    a = 10
    def inner():
        print(a)
    return inner
```

```
装饰器: 在不改变源函数的代码的基础上给函数添加新功能
通用装饰器写法:
def wrapper(fn):
    def inner(*args, **kwargs):
        '''在执行目标函数之前'''
        ret = fn(*args, **kwargs)
        '''在执行目标函数之后 '''
        return ret
    return inner

@wrapper  # target = wrapper(target)
def target():
    pass

# 此时必须要清楚, 该位置执行的是inner函数.
# inner函数中去执行你原来的target函数.
target()

带参数的装饰器
def wrapper_out(形参):
    def wrapper(fn):
        def inner(*args, **kwargs):
            '''在执行目标函数之前'''
            ret = fn(*args, **kwargs)
            '''在执行目标函数之后 '''
            return ret
        return inner
    return wrapper


# wrapper = wrapper_out(参数)
# @wrapper
@wrapper_out(实参)
def target():
```

```
    pass

target()


同一个函数被多个装饰器装饰
@wrapper_1
@wrapper_2
@wrapper_3
def func():
    pass
执行顺序:
wrapper_1
wrapper_2
wrapper_3
func()
wrapper_3
wrapper_2
wrapper_1

```

```
迭代器: 可以一个一个的拿到数据
可迭代对象: 具有__iter__和__next__的数据
通过__iter__ 可以拿到一个对象的迭代器
迭代器特点:
    1. 省内存(生成器)
    2. 惰性机制(不执行__next__不拿值)
    3. 只能向前, 不能反复(拿光了就没了)

for循环内部使用的就是迭代器.
当一个迭代器内的元素被拿空之后. 继续执行__next__ 会报错: StopIteration
```

```
生成器: 本质就是迭代器
生成器有两种创建方式:
1. 生成器函数
2. 生成器表达式

生成器函数: 函数内部有yield语句就是生成器函数
def gen():
    print(111)
    yield "你好"
g = gen() # 注意, 这句话并没有执行gen()这个函数, 而是通过gen函数创建了一个生成器
ret = g.__next__() # 此时才开始执行生成器. 并返回yield后面的数据
print(ret)   # 你好

生成器表达式: (结果 for循环 if条件)
```

```python
g = (i * 2 for i in range(10) if i % 2== 0)
print(g) # 此时打印的就是一个生成器对象. 并不是数据
print(list(g)) # list()函数可以把一个生成器拿空.并拿到列表 [0, 4, 8, 12, 16]

记住, 生成器具有惰性机制.

```

```
各种推导式
    列表推导式: [结果 for if]
    字典推导式: [k:v for if]
    集合推导式: [k for if]
    没有元组推导式: 那玩意是生成器表达式

```

```
匿名函数
lambda 参数: 返回值
通常配合内置函数一起使用(sorted, map, filter...)
```

```
内置函数(sorted, map, filter, reduce)
    sorted(Iterable, key=None, reverse=False)
        key是关键. 排序规则. sorted会自动把可迭代对象中的每一个传递给key.
        并根据key返回的内容进行排序
    map(function, iterable)
        可以对可迭代对象中的每一个元素进行映射.分别执行function
    filter(function. Iterable)
        function: 用来筛选的函数. 在filter中会自动的把iterable中的元素传递给function.
        然后根据function返回的True或者False来判断是否保留此项数据
     reduce(function, iterable, initial)
         reduce函数会自动把可迭代对象中的每一个拿出来传递给function
         第一次传递两个值进去. 然后把函数的返回值和下一个元素继续作为参数传递给function
         以此类推.最终得到function计算的结果.

```

练习题:

1. 写装饰器. 控制函数被调用的频率. 要求: 5秒钟执行一次. 少于5秒钟直接打印警告信息.

2. 写装饰器. 通过一次调用使函数执行5次

3. (重点, 难点)写装饰器, 要求. 被装饰的函数在执行的时候首先要判断登录状态. 如果是已经登录状态. 则直接执行. 否则提示用户去执行登录操作. 直到用户登录成功为止.

4. 看代码写出结果

```python
def say_hi(func):
    def wrapper(*args, **kwargs):
        print("HI")
        ret = func(*args, **kwargs)
```

```
        print("BYE")
        return ret
    return wrapper


def say_yo(func):
    def wrapper(*args, **kwargs):
        print("YO")
        return func(*args, **kwargs)
    return wrapper

@say_hi
@say_yo
def func():
    print("ROCK & ROLL")

func()
```

5. 一道面试题(坑爹)

```
def nums():
    return [lambda x : x* i for i in range(4)]
print([m(2) for m in nums()])
```

作业题:

简易博客园系统

```
1)，启动程序，首页面应该显示成如下格式：
                欢迎来到博客园首页
                1:请登录
                2:请注册
                3:文章页面
                4:日记页面
                5:注销
                6:退出程序
 2)，用户输入选项，3，4选项必须在用户登录成功之后，才能访问成功。
3)，用户选择登录，用户名密码从register文件中读取验证，三次机会，没成功则结束整个程序，
    登录成功之后，可以选择访问3，4项，访问页面之前，必须要在log文件中打印日志，
    日志格式为 --> 用户:xx在xx年xx月xx日 执行了xxx函数，访问页面时，
    页面内容为：欢迎xx用户访问评论（文章，日记）页面
4)，如果用户没有注册，则可以选择注册，注册成功之后，可以自动完成登录，然后进入首页选择。
5)，注销用户是指注销用户的登录状态，使其在访问任何页面时，必须重新登录。
6)，退出程序为结束整个程序运行。


坑：由于所有的操作都要带着用户名的. 比如，查看alex的文章. 打开的文件就是alex_文章.txt.
```

```
所以，我们需要在登录的时候把用户名记录在全局变量中．方便其他函数访问．

做题顺序：先把框架搭起来．让用户选择不同的菜单．
根据不同的菜单．执行不同的函数．
先做：
    注册
    登录
    日志(装饰器)
    登录状态检查(装饰器)
    注销
    退出程序
后做：
    文章
    日记
其中进入文章和日记之后：
进入文章页面：
    1．查看文章
    2．添加文章
    3．删除文章
    4．返回上一单元

进入日记页面：
    1．查看日记
    2．添加日记
    3．删除日记
    4．返回单一单元

思路：在注册用户的时候，给该用户创建两个文件，一个是存储文章的，一个是存储日记的．
alex_文章.txt:
    {"title":"昨夜大保健被抓实录", "content":"人在江湖飘，哪能不去piao"}
    {"title":"前天大保健被抓实录", "content":"这家不好，以后不来了"}
    {"title":标题，"content"：文章内容}
alex_日记.txt:
    2019-01-02$$今天很无聊．在家看电影
    2019-01-02$$今天很无聊．在家看电影
    时间$$内容


至理名言(我说的)：简单的问题复杂化，复杂的问题简单化．至繁归于至简!
最后：以上思路是我个人的．你可以有自己的想法和发挥．加油干吧．
```

超纲算法题

写函数:(考点: 递归)

有一个数据结构如下所示, 请编写一个函数从该结构中返回由指定的字段和对应的值组成的字典. 如果指定字段不

存在, 则跳过该字段

```
data:{
    "time":"2016-08-05",
    "some_id":"ID123456",
    "grp1":{"fld1":1, "fld2":2},
    "grp2":{"fld3":0, "fld4":0.4},
    "fld6":11,
    "fld7":7,
    "fld46":8
}

# 用户输入的fields: 由"|"链接的以fld开头的字符串, 如: fld1|fld7|fld29

def select(data, fields):
    pass  # 这里是你的代码
    return result
```

上期超缸体答案(day02):

```
"""
      *
    * * *
  * * * * *
* * * * * * *
  * * * * *
    * * *
      *
"""
# 让用户输入行数
max_len = int(input("请输入行数(奇数):"))
# 观察到行数其实就是拥有最多*的那一行的*个数
# 比如输入5, 最多中间那行就是5个*
# 再比如  7, 最多中间那行就是7个*
# 先奇数的数下去. 如果max_len = 5, 就是
# row
# 1
# 3
# 5
# 7
# 9
# 再思考. 行号是1的, 打印4个空格, 1个*
#             3       2个空格, 3个*
#             5       0个空格, 5个*
#             7       2个空格, 3个*
#             9       4个空格, 1个*
```

```
# 对称的效果就出来了
# 先计算空格打印的个数
# max_len - row:  4, 2, 0, -2, -4
# abs即可得到space:  4, 2, 0, 2, 4 : 空格的数量就有了
# *的个数怎么计算?
# max_len - space : 1, 3, 5, 3, 1 : 刚好是*的个数

# 最后完整代码:
for row in range(1, max_len * 2, 2):
    space = abs(max_len - row)
    star = max_len - space
    print(" "*space + "* "*star)

```