## python基本语法：

python2.0默认不支持中文，需要声明编码 #！-\*-coding:utf-8-\*- 来设置编码

python3.0默认支持中文

单行注释：#

多行注释 ’’’ ‘’’

用户输入：input eg:input(“your name?”)，但是input输入的都是字符串

ifelse语法：

if \*\*\* :

\*\*\*#必须有缩进，缩进的都是在if条件后执行的，缩进必须一致，推荐4个

else:

\*\*\*#必须有缩进，这儿缩进来的都是else需要执行的，相当于java的大括弧

if a=2:

print(“a=2”)

elif a<2:

print(“a<2”)

else:

pring(“a>2”)

if elif else只进一个，只要有一个匹配，进入一个执行完后直接返回

//整除

/除

\*\*指数

print(“aaa”,end=””)表示不换行输出，因为结尾是一个空字符，平常中用的还有end=”\t”,\t是制表符

多行的字符串只能用’’’ ‘’’（三个单引号）将其引住

单行的变量里面的时候，可以用单引号和双引号，他们两个的意义是一样的。

单行里面包含双引号的时候，可以用单引号将其包住，’hello ,It”s me’

单行里面包含单引号的时候，可以用双引号将其包住 ，“hello,It’s me”

单引号里面既包含单引号，又包含双引号的时候，用三引号将其包住，’’’ hello ,”It’s me”?‘’’

# 集成开发环境

IDE（Intergrated Development Environment）集成开发环境

VIM #经典的Linux下的文本编辑器

Emacs #Linux文本编辑器，比vim更容易使用

Eclips # java IDE,支持Python、Java、c、c++

Visual Studio # 微软开发的IDE，Python、c、java、c++

Notepad++

Sublime # Python开发的

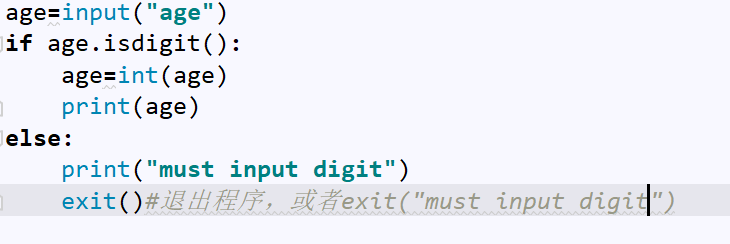
Pycharm，主要用于Python开发的IDE

# 格式化的输出

占位符：%s #s是string的缩写

在字符串的最后边用%()中间写上占位符需要的变量

Eg:msg=’’’Myname:%s,Myage:%s’’’ %(name,age)

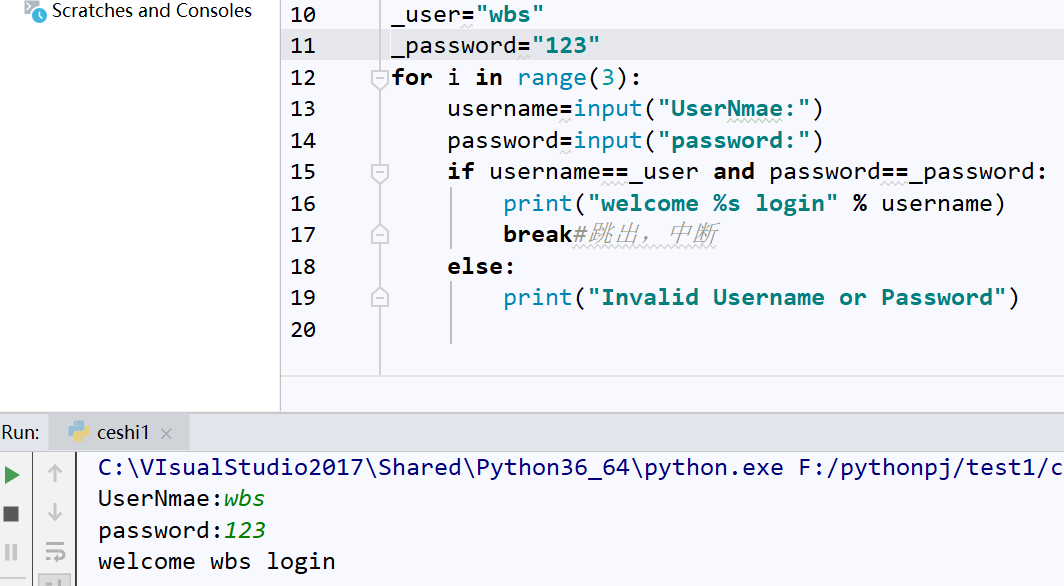


判断是不是一个数字的时候用isdigit();

%d #只能输入数字，d是digit

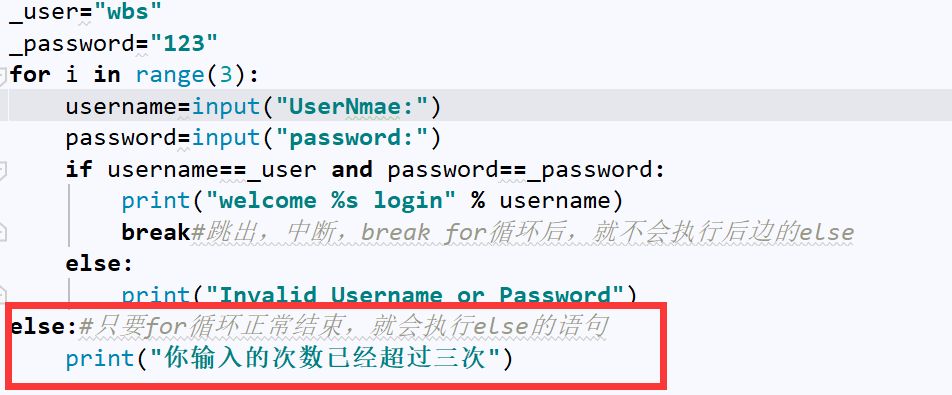
%f #f代表浮点数，约等于小数

简易的登录验证：



用的是for循环，判断三次，如果三次登录出错，就跳出；

for i in range(10):#表示的是0~9，而不是1~10

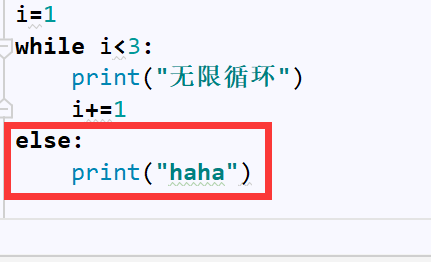


上边的程序，相当于是下边的程序的精简版：



python的for循环之后可以加else，但是不能加elif

# while循环



程序顺利执行完了之后，会执行else后边的代码。

continue #跳出本次的循环，继续进行下次的循环

break #跳出整个当前的循环

# 列表、切片（增删改查）

列表的创建可以用list()

a**=**[**'jinxing'**,**'sanpang'**,**'xiaohong'**,**'xiaomi'**,**'daming'**,**'ligang'**]  
print(a[1**:**2]) **#只取到了下标为1的**print(a[1**:**4])**#去到了下表为1，2，3的**print(a[1**:**])**#取到最后一个**print(a[1**:-**1])**#除了最后一个都能取到**print(a[1**:-**1**:**2])**#最后一个1代表步长，从左到右一个一个去取，取到倒数第二**print(a[1**::**2])**#最后一个1代表步长，从左到右一个一个去取，取到最后**print(a[6**::-**2])**#从右往左取，步长为2，**print(a[**-**3**::**])**#从左往右，从到书第三个开始取，一直取到最后**print(a[**-**3**::-**1])**#从右往左，从到书第三个开始取，一直取到最后**a.append(**'zhangqiang'**)**#只能插入到最后的位置**print(a)  
a.insert(1,**'aaa'**)**#在第一个后边插入aaa,可以将数据插入到任意位置insert(index,'内容')**print(a)  
a[1]**='bbb'#修改a[1]为bbb**print(a)  
a[1**:**3]**=**[**'a'**,**'b'**]**#同时修改1，2位置为a,b**print(a)  
a.remove(**'a'**)**#或者a.remove(a[1])也可以,remove删除的时候是作为一个整体，比如a.remove(['a','b'])是错误的删除方法**print(a)  
a.pop(1)**#pop可以根据索引删除元素，而且可以得到删除元素的值，不指定元素的时候删除最后一个元素**aa**=**a.pop(1)**#将删除的值赋给aa变量**print(a)  
**del** a[0]**#直接删除**print(a)  
**del** a**#也可以删除整个对象**t**=**[**'l'**,**'p'**,**'o'**,**'i'**,**'u'**,**'y'**,**'t'**,**'r'**,**'e'**,**'w'**,**'q'**,**'q'**].count(**'q'**)**#计算所包含元素的个数**print(t)  
a**=**[1,2,3]  
b**=**[4,5,6]  
a.extend(b)  
print(a)**#将b的元素添加到a**print(b)  
print(a.index(5))**#可以取出来某一个元素的索引值**a.reverse()**#将a元素倒转,无返回值**print(a)  
a.sort()**#将排序,无返回值**a.sort(reverse**=True**)**#排序完成后将其倒转**

b**=**sorted(a)**#sorted也是排序**print(a)  
print(5 **in** a)**#判断5是否在a里面**print(type(a) **is** list)**#判断a是否是一个列表**

a**=**[1,2,3]  
b**=**[4,5,6]  
c**=**a**+**b**#可以对连个列表进行相加，就是元素相加，而原来的a和b的元素不发生变化**print(c)

## 字符格式化输出

占位符 %s s = string

%d d = digit 整数

%f f = float 浮点数，约等于小数

## 数据运算

数据类型出初识

数字

整数 int(integer)

整型

长整型

in py3 已经不区分整型与长整型，统一都叫整型

in C int age 22 , long age

布尔 只有2种状态，分别是

真 True

假 False

字符串

salary.isdigit()

计算机中， 一切皆为对象

世界万物，皆为对象，一切对象皆可分类

## 循环loop

有限循环 ，次数限制

无限循环=死循环

continue 结束本次循环，继续下一次循环

break 跳出整个当前的循环

for

while

break , continue

## 数据类型

整数

字符串

列表，元组

查

索引(下标) ，都是从0开始

切片

.count 查某个元素的出现次数

.index 根据内容找其对应的位置

"haidilao ge" in a

增加

a.append() 追加

a.insert(index, "内容")

a.extend 扩展

修改

a[index] = "新的值"

a[start:end] = [a,b,c]

删除

remove("内容")

pop(index)

del a, del a[index]，默认不写时候删除最后一个元素

a.clear() 清空

排序

sort ()

reverse()

身份判断

>>> type(a) is list

True

>>>

## 购物车程序

salary = 5000

1. iphone6s 5800

2. mac book 9000

3. coffee 32

4. python book 80

5. bicyle 1500

>>>:1

余额不足，-3000

>>>:5

已加入bicyle 到你的购物车， 当前余额:3500

>>>:quit

您已购买一下商品

bicyle 1500

coffee 30

您的余额为:2970

欢迎下次光临

**列表里面可以存放各种类型的元素**

# 元组

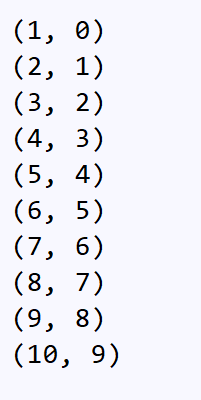
元组和列表差不多，值是将[]换成了()

元组只可以将其中的值拿出来，但是不能修改，是只读的列表

元组用的时候在最后一个加一个逗号。

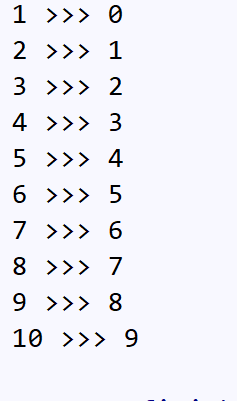
a**=**range(0,10)  
**for** i **in** enumerate(a,1)**:#enumerate可以自动加一个序号，默认从0开始，可以加一个参数enumerate(a,1)使得其编号从1开始** print(i)

结果如下：



**for** i,j **in** enumerate(a, 1)**: #前边加两个参数的时候，将第一个变量赋给i,将第二个变量赋给j,这样就可以不输出一个元组** print(i,**'>>>'**,j)

结果如下：



a,b**=**[3,4]**#将第一个元素给a，将第二个元素给b**print(a,**'----'**,b)

结果：

len(a)#可以计算出列表a的长度，

# Dictionary（字典）

字典是用{}包住

id(a)#可以算出a的地址。

字典是无序的，是可修改的类型。字典中的键必须是不可修改的类型，值可以是修改的类型

**#python的字典**dic1**=**{1**:'a'**,2**:'b'**}**#常用的创建字典的方式**print(dic1)  
dic2**=**dict(((**'name'**,**'wbs'**),))**#另外的一种创建字典的方式**print(dic2)  
dic3**=**dict([[**'name'**,**'wbs'**],])**#中括弧和小括弧的效果是一样的**print(dic3)

## #python字典的增加

dic1**=**{**'name':'wbs'**}  
dic1[**'age'**]**=**20**#加入一个元素**dic1[**'name'**]**='bo'#已经有了键的时候是修改**print(dic1)  
dic1.setdefault(**'age'**,18)**#如果键已经存在，不做任何修改，返回键对应的值，如果没有，加入字典，并返回相应的值**print(dic1)

## 字典的查找

a**=**list([**'a'**,**'d'**,4])**#列表的创建**print(a)  
  
dic4**=**{**'age':**18,**'name':'wbs'**,**'hobby':'girl'**}  
print(list(dic4.keys()))**#dic4.keys()返回的是一个dict\_keys类型的，可以将其转换成list**print(list(dic4.values()))  
print(list(dic4.items()))

## 字典的修改

**#对list的修改**li**=**[1,2,3,4]  
li[2]**=**10**#对列表的修改直接赋值就行**print(li[2])  
**#字典的修改**dic5**=**{**'age':**18,**'name':'wbs'**,**'hobby':'girl'**}  
dic5[**'age'**]**=**20**#将键取出来，直接修改**print(dic5)

## 字典的增加

**#字典的增加**dic6**=**{**'1':**1,**'2':**2}  
dic7**=**{**'3':**3,**'4':**4}  
dic6.update(dic7)**#将dic7加入到dic6中，dic7不做任何改变，如果有相同的，则更新，没有则加入dic6**print(dic6)

## 字典的删除

**#字典的删除  
del** dic6[**'1'**]**#将键值对一起删除**print(dic6)  
dic6.pop(**'2'**)**#根据索引删除，但是pop返回值是键对应的值**print(dic6)  
dic6.popitem()**#随机删除键值对，并且以元组的方式返回，用处不大**print(dic6)  
dic6.clear()**#直接清空dic6**print(dic6)

**del** dic6**#删除整个字典**

dic7**=**dict.fromkeys([**'host1'**,**'host2'**,**'hsot3'**],**'test'**)**#将三个key的值都赋值成test,一般做初始化**

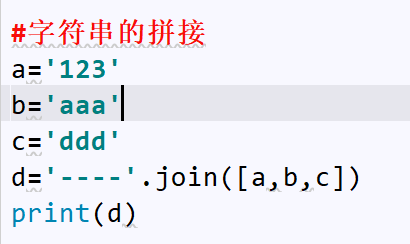
## 字典的排序

**#字典的排序**dic8**=**{3**:'666'**,7**:'555'**,2**:'444'**}  
print(sorted(dic8))**#默认按照键排序，结果是键的集合**print(sorted(dic8.values()))**#按照值排序，结果是值的集合**print(sorted(dic8.items()))**#按照键排序，结果是整个集合**

字典的遍历dic9**=**{**'age':**18,**'name':'wbs'**,**'hobby':'girl'**}  
**for** i **in** dic9**:** print(i)**#默认打印键  
for** i **in** dic9**:** print(i,dic9[i])**#打印键值对  
for** i **in** dic9.items()**:** print(i)**#打印键值**

**for** i,j **in** dic9.items()**:** print(i,j)**#打印键值**

# 字符串



结果如右：

#string的内置对象str**='hello wbs'**print(str.count(**'l'**))  
print(str.capitalize())**#将字符串首字母转换成大写**print(str.center(50,**'-'**))**#居中：--------------------hello wbs---------------------**print(str.endswith(**'bs'**))**#判断是否是以某个字符结尾**print(str.startswith(**'h'**))**#判断是否以某字符开头**

str1**='hellowo\trld'**print(str1.expandtabs(tabsize**=**10))**#在某地方加入10个空格**str.find(**'l'**)**#产找到第一个元素，并将索引值返回**

## 格式化字符串

str2**='hello {name} id {age}'**print(str2.format(name**='wbs'**,age**='20'**))**#格式化字符串方法1**print(str2.format\_map({**'name':'bo'**,**'age':**20}))**#格式化字符串方法2**

print(str2.index(**'ll'**))**#index和find一样的，但是找不到的时候会报错**print(**'abc456'**.isalnum())**#判断是否是一个字符或数字**

print(**'3'**.isidentifier())**#判断是否是一个整数**print(**'qq'**.isalpha())**#判断是否是一个字母**print(**'34abc'**.isidentifier())**#检验是否是一个非法字符**print(**'AAb'**.islower())**#判断是否全部是小写，isupper()正好相反**print(**' '**.isspace())**#判断是否是一个空格**

print(**'My Title'**.istitle())**#判断是否是一个标题，标题必须是首字母大写**print(**'My title'**.lower())**#所有的字符变成小写**print(**'My Title'**.upper())**#所有的字符编程大学**print(**'My Title'**.swapcase())**#所有的字符反转大小写**print(**'aaa'**.ljust(50,**'\*'**))**#在aaa右边跟50个\***print(**'aaa'**.rjust(50,**'\*'**))**#在aaa左边跟50个\***print(**' MMM MMM '**.strip())**#将文本前后的空格都去掉，lstrip,rstrip效果是去掉一左一右的空格**print(**'My Title'**.replace(**'Title'**,**'wbs'**))**#替换，可以加第三个参数，来控制替换的次数**print(**'My Title Title'**.replace(**'Title'**,**'bo'**,1))**#只替换一次**print(**'My title title'**.split(**' '**))**#将一个字符串分割成一个列表**print(**'My title'**.title())**#将字符串写成title的类型**

## 重点的方法：

切片、format、split、strip、replace、find、center、lower、upper、startwith、count、

# Python2中解码与编码的过程

**Unicode**

encode

decode

encode

decode

**GBK**

**utf-8**

GBK需要转换成utf-8格式流程如上：

1. 首先通过编码【decode】转换成Unicode编码
2. 然后通过解码【encode】转换成utf-8编码

# Python3转码过程

python3中默认编码是unicode，支持中文，不用转换成别的格式

注意：python3在encode时候，会转成bytes类型，解码的时候会把bytes转换成字符串

s**='王博盛'**unicode\_to\_gbk**=**s.encode(**'gbk'**)**#python中encode的时候会自动转换成一个bytes类型**print(s)**#python3是以Unicode编码，支持中文**print(unicode\_to\_gbk)**#转换成字节数组的时候是看不懂的 b'\xcd\xf5\xb2\xa9\xca\xa2'**print(unicode\_to\_gbk.decode(**'gbk'**))**#只有继续将其转换成Unicode时候，才可以是看到原来的样子，**

# 文件操作

**#读操作**f**=**open(**'wenben.txt'**,**'r'**,encoding**='utf-8'**)**#打开文件**data**=**f.read()**#读文件,read()可以加数字，比如read(5)表示读5个字符**print(data)  
f.close()  
  
  
**#写操作**f**=**open(**'wenben1.txt'**,**'w'**,encoding**='utf-8'**)**#打开文件，并清空文件，当文件不存在的时候，创建文件并清空文件的内容**f.write(**'hello world\n'**)**#写入多少个字符，指针跟着移动多少位，下次写入的时候接着从上边的地方写入**f.write(**'hello 王博盛'**)**#上边的写与这一行的写入文件都是从一行的末尾直接写入，不会自动换行**f.close()**#打开一个，一定要关闭一个文件  
  
#追加模式 a-->append**f**=**open(**'wenben1.txt'**,**'a'**,encoding**='utf-8'**)  
f.write(**'aaa'**)**#不会清空之前文本的内容**time.sleep(10)**#10s后执行下边的语句**f.close()**#缓冲区的内容写入文件中**

f**=**open(**'wenben1.txt'**,**'r'**,encoding**='utf-8'**)  
  
**for** i **in** f**:** print(i.strip())**#用for循环取出来其中的值的时候，不会全部加载到内存，是因为for循环做成了迭代器，开始打印第二行的时候，第一行的内容已经不存在了，以后读取文件就用这样的  
 #而readlines一下子会把所有的内容加载到内存中，尽量不要用Readlines**f.close()  
f**=**open(**'wenben1.txt'**,**'r'**,encoding**='utf-8'**)  
print(f.tell())**#打印当前光标的位置**print(f.read(4))  
print(f.tell())**#打印当前光标的位置，utf-8格式一个中文占三个字符，read与tell在中文的时候是对应的，但是时中文的时候是不对应的**f.seek(0)**#将光标移动到头**f.flush()**#将内存中的内容写到磁盘上  
  
import** sys,time  
**for** i **in** range(30)**:** sys.stdout.write(**'\*'**)  
 sys.stdout.flush()**#刷新将缓存区的写到磁盘或者屏幕,如果没有这句话，直接是等到所有的字符都写到缓冲区了，再直接打印到屏幕，而不是一个一个打印  
 #上边两行代码可以写成 print('\*',end='',flush=True)** time.sleep(0.2)  
  
f**=**open(**'wenben1.txt'**,**'a'**,encoding**='utf-8'**)  
f.truncate(5)**#从第五个字符后边的字符全部删掉，不加参数表示全部删掉，此方法必须要以a的方式打开文件**

# r+ w+ a+模式

**#\_\_author:bo  
#date:2018/9/1  
  
#w+ r+ a+**f**=**open(**'wenben1.txt'**,**'r+'**,encoding**='utf-8'**)**#r+模式读和正常的一样，但是写的时候是在最后写的**f**=**open(**'wenben1.txt'**,**'w+'**,encoding**='utf-8'**)**#w+模式是先清空，再去读**print(f.readline())  
f.write(**'wbs'**)**#光标跑到文本最后，再写入wbs**print(f.readline())**#此时是打印不到任何东西的，因为写的时候是将光标跑到最后，再读文件的时候是读不到东西的**f**=**open(**'wenben1.txt'**,**'a+'**,encoding**='utf-8'**)**#a+模式打开的时候其实的位置是在文本的最后边，  
#注意：r+ w+ 模式写入文本的时候永远是在文本的末尾写入的，不管光标在哪个位置，写的时候永远是在末尾写**

# 字符串与字典类型的转换

**#字典类型转换成字符串类型**s**="{'name':{'age':18}}"#此时是一个文本类型**s**=**str(s)**#此时是一个字符串类型**print(type(s))**#此时将字典类型完全转换成了一个字符串类型的对象**print(s)  
s**=**eval(s)**#此时又转换成了一个字典类型**print(s[**'name'**])

# #with语句 with open('wenben1.txt','r') as f: f.readline() f.read() print('hello')#退出with代码块的时候，with会自动关闭文件，而不用我们手动去关闭，推荐用with #可以同时打开多个文件 with open('wenben1.txt','r') as f,open('wenben1.txt','r') as f2: print('文件操作')

# 运算符

身份运算符：is 、 is not 、

成员运算符：in 、not in

a**=**1  
id(a)**#id用来查看内存地址**b**=**5  
print(b**//**2)**#/代表是正常的除，//是整除**c**=**2  
print(c**\*\***8)**#\*\*是次方  
  
  
#！/usr/bin/env python放在有些代码的第一行，起到解释器的作用，  
# 如果执行的时候使用python a.py的方式执行代码，name这个第一行的#！/usr/bin/env python会没用的**

**input不管输入的是什么，都会当做字符串来解释**

aa**=**3333333  
print(id(aa))

字符串的表示形式：三种

**'aa'  
"aaa"**s**=""""aaaaaa"""**print(s)

# 深浅拷贝

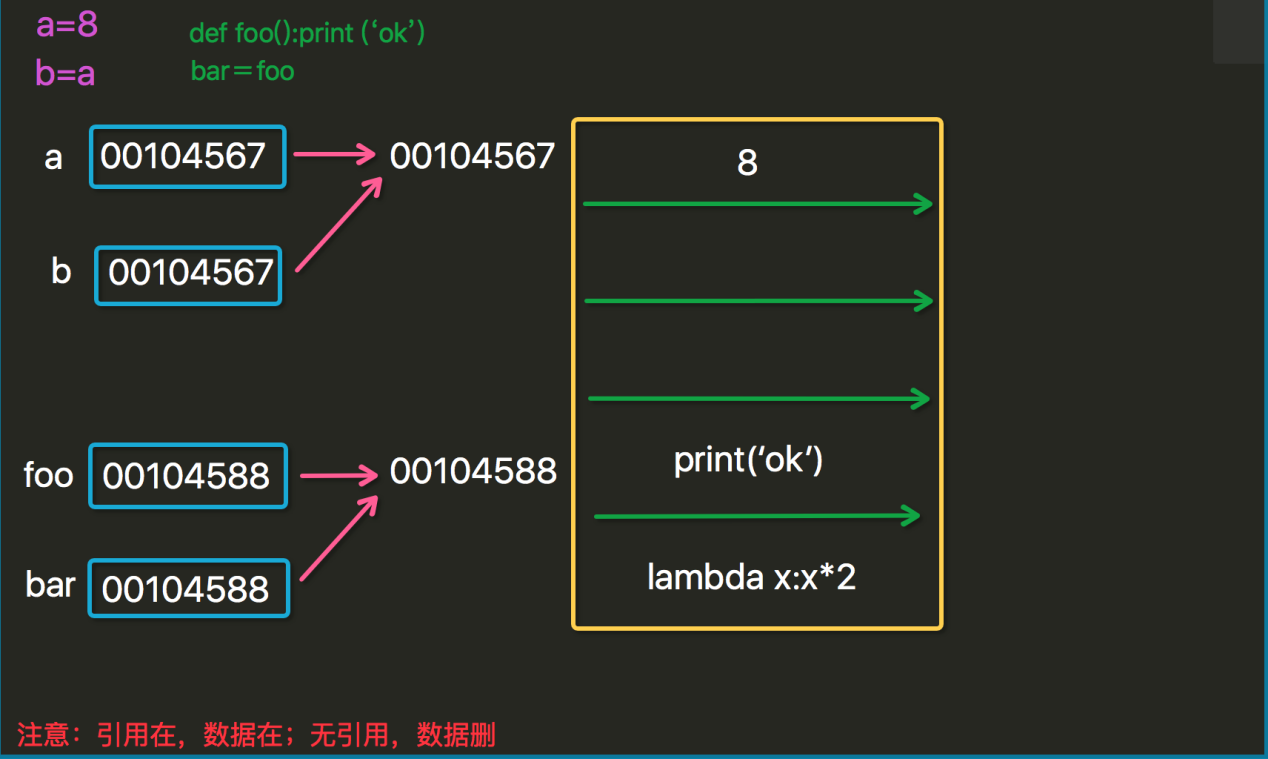
**import** copy  
s**=**[1,**'wbs'**,**'wbs2'**]  
s1**=**s.copy()**#将s列表完全拷贝一份**s1[0]**=**3**#修改拷贝后的，不会影响之前的元素**print(s)  
print(s1)**#修改字符串的时候，s不会跟着变化的**s2**=**[[1,2],**'wbs'**,**'wbs1'**]  
s3**=**s2.copy()  
s3[0][1]**=**5**#修改s3**print(s3)  
print(s2)**#s2也跟着变化，因为在修改的时候直接修改了第一个元组总第二个元素的指针**print(**'============================'**)  
a**=**[[1,2],3,4]  
b**=**a**#这种拷贝，改变一个元素，拷贝前后的元素都是保持一致的**print(a)  
print(b)  
c**=**a.copy()**#这种拷贝，普通类型的修改不会改变原来的值，叫做浅拷贝，浅拷贝意味着有些没有真正的拷贝，而是共享**c[1]**=**55555  
print(a)  
print(c)  
**#浅拷贝是拷贝第一层，深拷贝是拷贝全部**print(**"============="**)  
a**=**[[1,2],3,4]  
b**=**copy.deepcopy(a)**#这个是深拷贝**b[0][1]**=**5555**#只修改了深拷贝后的内容**print(a)  
print(b)

# 集合

**#列表**a**=**[1,2,3]  
b**=**list([4,5,6])**#创建列表的两种方式**c**=**(1,2,3)  
d**=**tuple([4,5,6])**#创建元组的两种方式  
#集合的创建方式set()**s**=**set(**'wbs'**)  
print(s)**#运行结果 {'b', 'w', 's'}，无序**s**=**set(**'wbswbs'**)  
print(s)**#运行结果 {'b', 'w', 's'}，无序,会自动去掉重复的内容  
#使用set可以去重功能**a**=**[**'wbs'**,**'xiaoPOooo'**,**'wbs'**]  
b**=**set(a)**#会自动去掉重复的**print(b)**#结果：{'xiaoPOooo', 'wbs'}  
#注意：对某一个元素进行set的时候，每一个元素必须是课Hash的，即每一个元素都是不可变的（可以是元组，不可以是字典和列表）  
#集合的访问，只能通过遍历或者迭代器**a**=**[**'wbs'**,**'xiaoPOooo'**,**'wbs'**]  
b**=**set(a)  
b.add(**'tianjia'**)**#集合的添加，将添加的内容作为一个整体添加到集合中**print(b)  
b.update(**"很好"**)**#update也是一个更新，但是这个元素会被拆分成一个个字符来对待**print(b)  
b.update([12,**"添加2"**])**#此时，12和‘添加2’被当做两个整体来对待**print(b)  
b.remove(12)**#删除元素**print(b)  
b.pop()**#随机删除一个**print(b)  
b.clear()**#清空，结果是：set()，代表一个空集合**print(b)  
**del** b**#直接删除b  
# print(b)#打印会报错**print(set(**'wbswbs'**)**==**set(**'wbs'**))**#结果为true**print(set(**'wbswbs'**)**<**set(**'wbsaaa'**))**#结果为true,判断为是否包含**print(set(**'wbswbs'**)**<**set(**'wbs'**))**#结果为false,相等的关系  
#联合 or**print(set(**'wbs'**) **and** set(**"wbsabc"**))**#and 取出两个的**print(set(**'wbs'**) **or** set(**'1243'**))**#只取一个  
#交叉并**print(**"================"**)  
a**=**set([1,2,3,4,5])  
b**=**set([3,4,5,6,7])  
print(a.intersection(b))**#取交集**print(a **&** b)**#求并集**print(a.union(b))  
print(a**|**b)**#管道符合union是一样的**print(a.difference(b))**#求交集，a中有b中没有的**print(a**-**b)**#减号是求差集**print(a.symmetric\_difference(b))**#对称差集，取两个集合中不是共有的**print(a**^**b)**#是求对称差集**

# 函数

## 作用域：



**#函数  
import** time  
**def F**()**:** print(**"ok"**)  
  
  
**for** i **in** range(10)**:** F()**#打印了10此ok  
#注意：  
#函数名是区分大小写的  
#函数名不能是保留字  
#函数名不能使用标点符号**print(**"======================"**)  
**def sum**(*a*,*b*)**:** print(*a***+***b*)  
 print(*a*)  
 print(*b*)**#实参和形参的顺序是按照参数的名称来分配**sum(5,4)  
print(**"======================"**)  
**def f**(*name*,*age*)**:** print(**'我是 %s,今年%s' %**(*name*,*age*) )  
f(**'wbs'**,**'20'**)  
f(age**=**30,name**='wbs'**)**#可以写上对应的关键字参数，效果和上边的一样  
def f1**(*name*,*sex***='man'**)**:** print(**"%s is %s"%**(*name*,*sex*) )**#可以设置一个默认的性别**f1(**'wbs'**)  
f1(**"alix"**,**'girl'**)**#也可以设定参数**time\_format**='%Y-%m-%d %X'**current\_time**=**time.strftime(time\_format)  
print(current\_time)  
  
**def f2**(*\*args*)**:** sum**=**0  
 **for** i **in** *args***:** sum**+=**i  
 print(sum)  
f2(1)  
f2(1,2,3)**#不定长的参数，输出的是一个元组  
def f3**(*\*args*,*\*\*kwargs*)**:** print(*args*)**#结果：('wbs', 12)，被处理称为一个元组** print(*kwargs*)**#结果：{'name': 'wbs', 'age': 12}，被处理称为一个字典  
 for** i **in** *kwargs***:** print(**'%s:%s'%**(i,*kwargs*[i]))**#打印键值对  
 #注意：无命名参数放在前边，有键值对的参数放在右边**f3(**'wbs'**,12,name**='wbs'**,age**=**12)  
  
**def f4**(sex**='male'**,*\*args*,**\*\***kwargs)**:#有名的参数必须在无名的前边** print(*args*)  
f4()  
f4(**'man'**,1,2,3,4)  
f4(1,2,3,4,**'wbs'**)

**#\_\_author:bo  
#date:2018/9/14  
  
#函数的返回值  
def f**()**:** print(**'function'**)  
 **return 'ok'#结束函数，返回某个对象，后边的程序不会执行，不加return的时候自动添加return None  
#注意：如果return多个对象，那么python会将多个对象封装成一个元组返回**a**=**f()  
print(a)  
  
**#函数的作用域  
#built\_in>global>enclosing>local**a**=**5**#global**b**=**int(2.9)**#built\_in  
def outer**()**:** a**=**4**#enclosing  
 def inner**()**:** b**='aa'#local** print(b)**#优先级是local>enclosing>global>built\_in** inner()  
outer()  
**#注意：局部作用域不能修改全局作用域的变量**count**=**10  
**def f**()**:** print(count)  
 **#count+=1#报错，内部引用的时候已经是全局的count，不能对他修改  
 #count=5#这句话会报错，因为是在局部中已经引用全局的变量了，但是局部内不能修改全局的变量  
def f1**()**:** count**=**5  
 print(count)**#这句是可以执行的，因为在局部中重新创建了一个count，与外边的不是一个对象  
#如果在局部的函数中间需要修改全局的变量，只需要定义一下global  
def f3**()**:  
 global** count**#表示函数内部用的count是全局的count** print(count)**#10** count**+=**1  
 print(count)**#11**f3()  
  
  
**def outer1**()**:** a**=**4**#enclosing  
 def inner1**()**:  
 nonlocal** a**#如果在局部需要声明一个enclosing的变量  
 # ，需要用nonlocal来声明，作用和用global声明全局的变量是一样的** print(a)**#4** a**+=**1  
 print(a)**#5** inner1()  
outer1()  
**#global nonlocal 在局部函数中的作用差不多是一样的，都是讲外部的变量声明为可以在函数内部可以修改的**

**def f5**(*\*args*)**:** print(*args*)  
a**=**[1,2,3]  
f5(**\***a)**#将一个元组传入到args,传入的时候带一个\*就可以了  
def f6**(*\*\*kwargs*)**:** print(*kwargs*)  
f6(name**='www'**,age**=**20)**#{'name': 'www', 'age': 20}**f6(**\*\***{**'name':'wbs'**,**'age':'12'**})**#{'name': 'wbs', 'age': '12'}  
#两个\* 号直接可以传入一个字典类型**print(**'================'**)  
**def fun1**(*n*)**:  
 return** *n***\****n***def fun**(*a*,*b*,*f*)**:** print(*f*(*a*)**+***f*(*b*))  
  
fun(3,4,fun1)**#25,可以传入一个函数  
# 函数名可以赋值，  
# 函数名字既可以作为函数的参数，又可以做为函数的返回值  
def foo**()**:  
 def foo1**()**:  
 return** 8  
 **return** foo1  
result**=**foo()  
print(result)**#得到函数foo()的地址<function foo.<locals>.foo1 at 0x000001DE682CE620>**print(result())**#得到函数的结果：8**

**#递归函数  
def fun**(*n*)**:  
 if** *n***==**1**:  
 return** 1  
 **return** *n***\***fun(*n***-**1)**# 可以用递归函数解决的问题都可以用循环解决**print(fun(5))  
**# 递归的效率很低,递归必须要有结束条件  
  
# 斐波那契数列  
# 0 1 1 2 3 5 8 13 21 34 55 。。。。  
def fibo**(*n*)**:  
 if** *n***<=**2**:  
 return** *n***-**1  
  
 **return** fibo(*n***-**1)**+**fibo(*n***-**2)  
print(fibo(8))

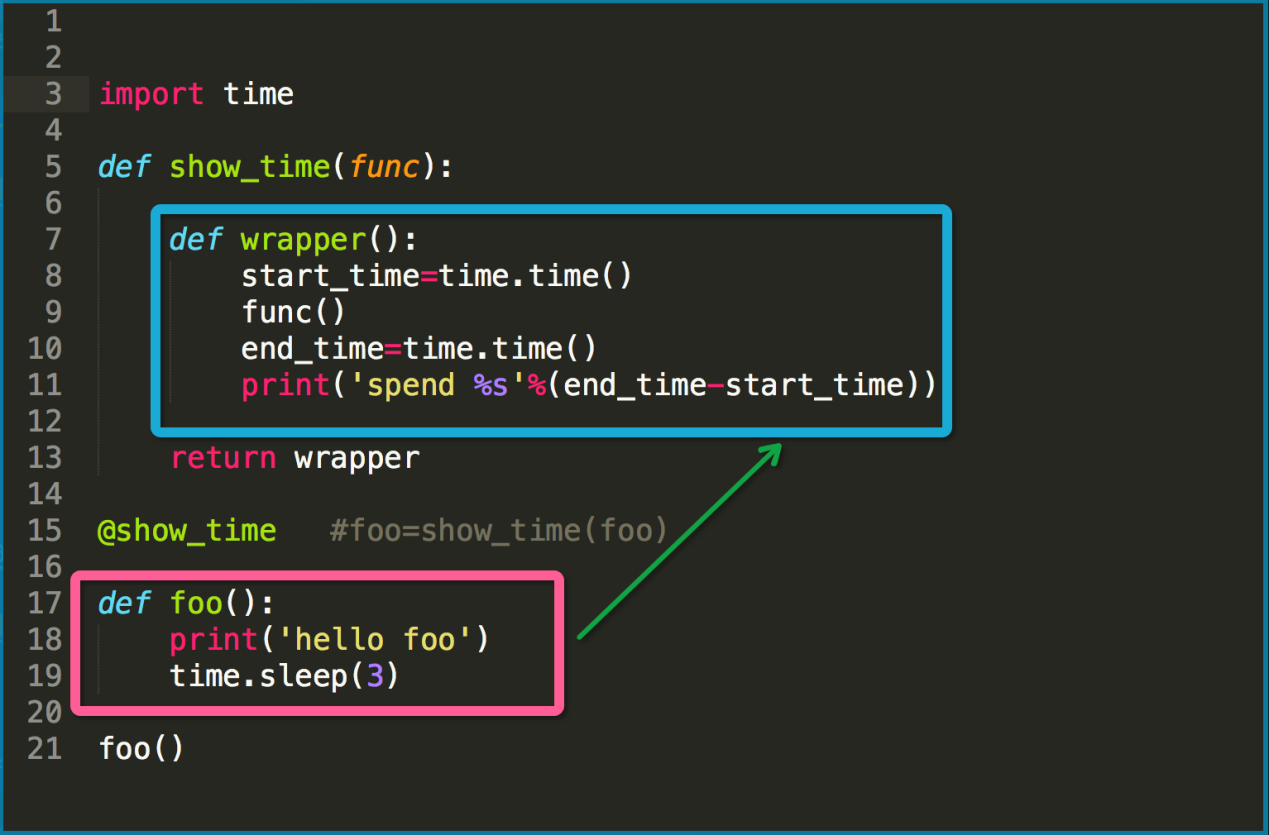
闭包(closure)是函数式编程的重要的语法结构。

定义：如果在一个内部函数里，对在外部作用域（但不是在全局作用域）的变量进行引用，那么内部函数就被认为是闭包(closure).

**@show\_time # foo=show\_time(foo)  
def foo**()**:** print(**'hello foo'**)  
 time.sleep(3)  
  
  
**@show\_time # bar=show\_time(bar)  
def bar**()**:** print(**'in the bar'**)  
 time.sleep(2)  
  
  
foo()  
print(**'\*\*\*\*\*\*\*\*\*\*\*'**)  
bar()

如上所示，这样我们就可以省去bar = show\_time(bar)这一句了，直接调用bar()即可得到想要的结果。如果我们有其他的类似函数，我们可以继续调用装饰器来修饰函数，而不用重复修改函数或者增加新的封装。这样，我们就提高了程序的可重复利用性，并增加了程序的可读性。

      这里需要注意的问题：  foo=show\_time(foo)其实是把wrapper引用的对象引用给了foo，而wrapper里的变量func之所以可以用，就是因为wrapper是一个闭包函数。



@show\_time帮我们做的事情就是当我们执行业务逻辑foo()时，执行的代码由粉框部分转到蓝框部分，仅此而已！

### 带参数的被装饰函数

**import** time  
  
  
**def show\_time**(*func*)**:  
 def wrapper**(*a*, *b*)**:** start\_time **=** time.time()  
 func(*a*, *b*)  
 end\_time **=** time.time()  
 print(**'spend %s' %** (end\_time **-** start\_time))  
  
 **return** wrapper  
  
  
**@show\_time # add=show\_time(add)  
def add**(*a*, *b*)**:** time.sleep(1)  
 print(*a* **+** *b*)  
  
  
add(2, 4)

**import** time  
  
**def show\_time**(*func*)**:  
  
 def wrapper**(*a*,*b*)**:** start\_time**=**time.time()  
 ret**=**func(*a*,*b*)  
 end\_time**=**time.time()  
 print(**'spend %s'%**(end\_time**-**start\_time))  
 **return** ret  
  
 **return** wrapper  
  
**@show\_time #add=show\_time(add)  
def add**(*a*,*b*)**:** time.sleep(1)  
 **return** *a***+***b*print(add(2,5))

**#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*不定长参数  
import** time  
  
**def show\_time**(*func*)**:  
  
 def wrapper**(*\*args*,*\*\*kwargs*)**:** start\_time**=**time.time()  
 func(**\****args*,**\*\****kwargs*)  
 end\_time**=**time.time()  
 print(**'spend %s'%**(end\_time**-**start\_time))  
  
 **return** wrapper  
  
**@show\_time #add=show\_time(add)  
def add**(*\*args*,**\*\***kwargs)**:** time.sleep(1)  
 sum**=**0  
 **for** i **in** *args***:** sum**+=**i  
 print(sum)  
  
add(2,4,8,9)

**不定长参数**

**#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*不定长参数  
import** time  
  
**def show\_time**(*func*)**:  
  
 def wrapper**(*\*args*,*\*\*kwargs*)**:** start\_time**=**time.time()  
 func(**\****args*,**\*\****kwargs*)  
 end\_time**=**time.time()  
 print(**'spend %s'%**(end\_time**-**start\_time))  
  
 **return** wrapper  
  
**@show\_time #add=show\_time(add)  
def add**(*\*args*,**\*\***kwargs)**:** time.sleep(1)  
 sum**=**0  
 **for** i **in** *args***:** sum**+=**i  
 print(sum)  
  
add(2,4,8,9)

### 带参数的装饰器

装饰器还有更大的灵活性，例如带参数的装饰器：在上面的装饰器调用中，比如@show\_time，该装饰器唯一的参数就是执行业务的函数。装饰器的语法允许我们在调用时，提供其它参数，比如@decorator(a)。这样，就为装饰器的编写和使用提供了更大的灵活性。

**import** time  
  
  
**def time\_logger**(*flag***=**0)**:  
 def show\_time**(*func*)**:  
 def wrapper**(*\*args*, *\*\*kwargs*)**:** start\_time **=** time.time()  
 func(**\****args*, **\*\****kwargs*)  
 end\_time **=** time.time()  
 print(**'spend %s' %** (end\_time **-** start\_time))  
 **if** flag**:** print(**'将这个操作的时间记录到日志中'**)  
 **return** wrapper  
 **return** show\_time  
  
  
**@time\_logger**(3)  
**def add**(*\*args*, **\*\***kwargs)**:** time.sleep(1)  
 sum **=** 0  
 **for** i **in** *args***:** sum **+=** i  
 print(sum)  
  
  
add(2, 7, 5)

@time\_logger(3) 做了两件事：

    （1）time\_logger(3)：得到闭包函数show\_time，里面保存环境变量flag

    （2）@show\_time   ：add＝show\_time(add)

上面的time\_logger是允许带参数的装饰器。它实际上是对原有装饰器的一个函数封装，并返回一个装饰器(一个含有参数的闭包函数)。当我 们使用@time\_logger(3)调用的时候，Python能够发现这一层的封装，并把参数传递到装饰器的环境中。

**#装饰器加参数  
# 需求：传入的参数为true的时候打印日志，出入false时候不打印日志  
import** time  
**def logger**(*flag***=''**)**:  
 def show\_time**(*f*)**:  
 def inner**(*\*x*, *\*\*y*)**:** start **=** time.time()  
 f(**\****x*, **\*\****y*)  
 time.sleep(1)  
 end **=** time.time()  
 **if** flag**:** print(**'打印日志'**)  
 print(**'spend time is%s' %** (end **-** start))  
 **return** inner  
 **return** show\_time  
**@logger**(**'false'**)**#这个不仅可以在需要的时候打印日志，而且可以计算时间  
def add**(*\*a*,**\*\***b)**:** sums**=**0  
 **for** i **in** *a***:** sums**+=**i  
 print(**'sums is '**,sums)  
add(1,2,3)

# 列表生成器

**#\_\_author:bo  
#date:2018/9/16  
#列表生成器**a**=**[x **for** x **in** range(10)]  
b**=**[x**\***2 **for** x **in** range(10)]**#先取出列表中的元素，再对列表中的元素乘以2，最后一个一个放入列表中**print(a)  
print(b)  
print(**'================================================================'**)  
**def f**(*n*)**:  
 return** *n***\****n*a**=**[f(x) **for** x **in** range(10)]**#这种方式直接将每个元素放入了内存中**print(a)**#也可以在一个列表中传入一个函数  
#简单的赋值方法**t**=**[**'wbs'**,21]  
a,b**=**t**#简单的取出列表中的元素的方法，接受元素的参数个数不能多出列表中的元素的个数  
#上边的语句相当于是**a**=**t[0]  
b**=**t[1]  
print(a,**' '**,b)  
**#a,b,c=t#报错not enough values to unpack (expected 3, got 2)  
#t=['wbs',21,22]# 报错 too many values to unpack (expected 2)，值太多**a,b**=**t  
print(a,**' '**,b)  
  
  
a**=**(x **for** x **in** range(10))**#创建生成器**print(a)**#<generator object <genexpr> at 0x000001F1C3E21C50>  
#print(a.\_\_next\_\_())#\_\_next\_\_()是内部方法，不建议使用**print(next(a))**#等价于上边的方法，建议使用,python2中是a.next()**print(next(a))**# 这次打印的时候是从上边的值的下一个  
  
#生成器就是一个可迭代对象  
# 用for循环来遍历一个生成器的时候，不会将生成器的所有元素加载到  
# 内存，它只是用一个加载一个，然后再用垃圾回收机制回收一个  
for** i **in** a**:** print(i)  
  
**#生成器的两种创建方式  
# 1 a=(x for x in range(10))  
# 2 yield方式创建  
def f**()**:#yield方式创建生成器** print(**'ok'**)**#生成器中的代码不会执行  
 yield** 1**#yield相当于是返回一个1，下边的代码不在执行** print(**'ok2'**)  
 **yield** 2**# 返回的是1 ，** print(**'ok3'**)  
 **yield** 3**#相当于返回的是2**g**=**f()**#是一个生成器对象，而不是执行这个函数  
# next(g)  
# next(g)#第二个next是从上一个next接着开始的，而不是从头开始的  
# next(g)#得到的是f()返回的结果  
for** i **in** g**:#每次进入到g=f()生成器的时候，会执行next(f())这段代码，执行完了将其yield 1返回给i，在for循环中打印了出来** print(i)  
**#上边的循环的结果是：  
# ok  
# 1 #这个1是f（）函数返回的那个yield 1，在for循环中将其打印了处理啊  
# ok2 #这个ok2是在for循环中的g=f()生成器自动执行next(f())的时候打印的  
# 2 #这个2是f（）函数返回的那个 yield 2，在for循环中将其打印了处理啊  
# ok3 #这个ok3是在for循环中的g=f()生成器自动执行next(f())的时候打印的  
# 3**a**=**[1,1,2,3]  
a.\_\_iter\_\_()  
b**=**(1,2,3)  
b.\_\_iter\_\_()  
c**=**{**'name':'wbs'**}  
c.\_\_iter\_\_()  
**#for循环后边加的是可迭代对象  
#什么是可迭代对象？  
# 看是否是可迭代对象，只需要看它是否有————next（）————方法或者\_\_iter\_\_()方法  
# 字典、集合、元组都是可迭代对象**print(**'=================='**)  
**#斐波那契数列  
def fib**(*max*)**:** n,before,after**=**0,0,1  
 **while** n**<***max***:** print(before)  
 before,after**=**after,before**+**after  
 n**+=**1  
fib(10)  
  
print(**'==================='**)  
**def fib**(*max*)**:** n,before,after**=**0,0,1  
 **while** n**<***max***:  
 yield** before**#用yield返回一个值，在下边需要的时候再执行，这样就不会直接加载到内存中** before,after**=**after,before**+**after  
 n**+=**1  
g**=**fib(10)  
print(next(g))  
print(next(g))  
print(next(g))  
print(next(g))  
print(next(g))  
print(next(g))  
print(next(g))  
print(next(g))  
  
print(**'==================='**)  
**def f**()**:** print(**'ok'**)  
 count**=yield** 1**#可以给yield加一个接受值的参数** print(count)  
 print(**'ok2'**)  
 **yield** 2  
 print(**'ok3'**)  
 **yield** 3  
b**=**f()**# 将生成器赋给b**s**=**b.send(**None**)**# 此句话相当于是next（b）,第一次send前如果没有next，只能给它传入一个None  
#用send执行的时候，第一次必须传入None这个参数，不然会 报错**print(s)  
s**=**b.send(**'wbs'**)**# 在send中传入值的时候，可以在yield中接收到这个值  
# 注意：第一次没有执行next并且执行send的时候，必须要传入None，因为第一次的时候没有  
# 执行到yield，因此没有变量去接收传入的参数，会报错**print(s)

生成器都是迭代器，但是迭代器不一定是生成器

a**=**[1,2,3,4]  
b**=**iter(a)**# iter()生成一个迭代对象并且返回  
# 迭代器对象的两个条件  
# 有iter()方法 有next()方法  
  
# for循环做的三件事  
 # 1 调用可迭代对象的iter()方法返回一个可迭代对象  
 # 2 不断调用迭代器对象的next方法  
 # 3.处理StopIteration  
  
from** collections **import** Iterator,Iterable  
print(isinstance([1,2,3],list))**# 判断前边的[1,2,3]是不是list**print(isinstance([1,2,3],Iterable))**# 判断是否是一个可迭代对象  
# Iterable是可迭代对象，Iterator是迭代器**

时间  
**#\_\_author:bo  
#date:2018/9/17  
  
import** time  
print(help(time))  
print(time.time())**# 打印时间戳**print(time.clock())**# 计算CPU花费的时间**print(time.gmtime())**# UTC标准时间，结构化时间  
# 结果： time.struct\_time(tm\_year=2018, tm\_mon=9, tm\_mday=17, tm\_hour=14, tm\_min=30, tm\_sec=4, tm\_wday=0, tm\_yday=260, tm\_isdst=0)**print(time.localtime())**# 东八区的时间**print(time.strftime)**# 字符串时间**print(help(time.strftime))**# 字符串时间  
  
# %Y Year with century as a decimal number.  
# %m Month as a decimal number [01,12].  
# %d Day of the month as a decimal number [01,31].  
# %H Hour (24-hour clock) as a decimal number [00,23].  
# %M Minute as a decimal number [00,59].  
# %S Second as a decimal number [00,61].  
# %z Time zone offset from UTC.  
# %a Locale's abbreviated weekday name.  
# %A Locale's full weekday name.  
# %b Locale's abbreviated month name.  
# %B Locale's full month name.  
# %c Locale's appropriate date and time representation.  
# %I Hour (12-hour clock) as a decimal number [01,12].  
# %p Locale's equivalent of either AM or PM.**struct\_time**=**time.localtime()  
print(time.strftime(**'%Y-%m--%d %H:%M:%S'**,struct\_time))**# 可以自己拼接时间**print(time.strftime(**'%Y-%m--%d %H:%M:%S'**))**# 可以自己拼接时间**print(time.strptime(**'2018-09--17 22:47:56'**,**'%Y-%m--%d %H:%M:%S'**))**# 转换成结构化时间  
# 结果：time.struct\_time(tm\_year=2018, tm\_mon=9, tm\_mday=17, tm\_hour=22, tm\_min=47, tm\_sec=56, tm\_wday=0, tm\_yday=260, tm\_isdst=-1)**a**=**time.strptime(**'2018-09--17 22:47:56'**,**'%Y-%m--%d %H:%M:%S'**)  
print(a.tm\_mday)**# 可以将一个时间转换成结构化时间后再取出其中的具体参数的值**print(time.ctime())**# Mon Sep 17 22:52:41 2018，也可以加一个时间戳，来转换成一个可以显示的时间**print(time.ctime(12344))**# 转换成标准时间后为：Thu Jan 1 11:25:44 1970**print(time.mktime(time.localtime()))**# 1537196076.0,结构化方法转换成一个时间戳**print(time.localtime())**# 打印结构化时间  
  
import** datetime  
print(datetime.datetime.now())**# 2018-09-17 22:56:28.235684**

# random

**import** random  
print(random.random())**# 打印0~1的随机数**print(random.randint(1,8))**# [1,8]区间的数**print(random.choice(**'hello'**))  
  
print(random.choice([**'123'**,1,4,8,4]))**# 随机选择一个元素**print(random.sample([**'12'**,5,3,**'a'**,[3,4]],2))**# 随机从列表中取两个元素**print(random.randrange(1,8))**# 返回一个[1,8)之间的数  
  
def v\_code**()**:** code**=''  
 for** i **in** range(5)**:  
 if** i**==**random.randrange(0,9)**:** add**=**random.randrange(10)  
 **else:** add**=**chr(random.randrange(65,91))  
 code**+=**str(add)  
 print(code)  
v\_code()**# 随机生成5位验证码的过程  
  
# print(chr(88))# 将一个数转换成ascii码  
def v\_code2**()**:** code**=''  
 for** i **in** range(5)**:# 数字字母验证码的生成过程** add**=**random.choice([random.randrange(10),chr(random.randrange(65,91))])**# 从数字和字母中随机=取一个，添加到验证码中** code**+=**str(add)  
 print(code)  
v\_code2()

# os模块

**#\_\_author:bo  
#date:2018/9/18  
import** os  
print(os.getcwd())**# 获取当前工作目录**os.makedirs(**'abc\\wbs\\aa'**)**# 在当前工作目录下边生成多个文件夹**os.removedirs(**'abc\\wbs\\aa'**)**# 删除文件夹，只能删除空的文件夹**os.mkdir(**'cc'**)**# 生成一个文件夹**os.mkdir(**'cc\\bb'**)**# 在已经存在的文件夹下生成一个文件夹**os.rmdir(**'cc\\bb'**)**# 删除一个文件夹**os.rmdir(**'cc'**)  
  
  
print(os.chdir(**r'D:\BaiduNetdiskDownload\下载器2.1.3\我的资源\全套第2套\最新全栈python第2期视频教程 全套完整版\day18-python 全栈开发-基础篇'**))  
print(os.listdir(**r'D:\BaiduNetdiskDownload\下载器2.1.3\我的资源\全套第2套\最新全栈python第2期视频教程 全套完整版\day18-python 全栈开发-基础篇'**))**# 将目录中的文件放在列表中国展示出来  
# 上边可以改变工作目录的路径**print(os.getcwd())  
  
print(os.curdir)**# 返回当前文件夹**print(os.pardir)**# 获取当前目录的父目录字符串名  
  
# os.remove('a.py') # reomve只能删除文件，不能删除文件夹  
# os.rename('oldname','newname')# 修改文件的名称  
  
#info=os.stat('os模块.py')# 获取文件/目录信息  
# print(info)**s**=**os.sep**# 输出操作系统特定的路径分隔符，win下为"\\",Linux下为"/",可以用os.sep连接一个文件的路径，这样就可以在Linux和win下任意使用**print(s) **# 'c:%sUser%swbs' %s**s**=**os.linesep **# 输出当前平台使用的行终止符，win下为"\r\n",Linux下为"\n"**print(**'s=='**,s,**'=='**)  
  
s**=**os.pathsep **# 输出用于分割文件路径的字符串**print(s)  
  
print(os.environ)**# 获取系统环境变量**print(os.path.abspath(**'./abc'**))**# 返回path规范化的绝对路径**print(os.path.split(**r'./abc'**)) **# 将path分割成目录和文件名二元组返回,根据最后一个分隔符找到**print(os.path.split(**r'D:\BaiduNetdiskDownload\下载器2.1.3\我的资源\全套第2套\最新全栈python第2期视频教程 全套完整版\day18-python 全栈开发-基础篇'**)) **# 将path分割成目录和文件名二元组返回)**print(os.path.dirname(**r'./abc'**)) **#返回path的目录。其实就是os.path.split(path)的第一个元素,即拿到参数的上一个文件夹的目录**print(os.path.basename(**r'./abc'**)) **#返回path最后的文件名。如何path以／或\结尾，那么就会返回空值。即os.path.split(path)的第二个元素**print(os.path.join(**'abc'**,**'bcd'**)) **# 结果：abc\bcd 将多个路径组合后返回，第一个绝对路径之前的参数将被忽略  
  
# os.path.getatime(r'bcd') #返回path所指向的文件或者目录的最后存取时间  
# os.path.getmtime(r'bcd') #返回path所指向的文件或者目录的最后修改时间  
  
  
# os.getcwd() 获取当前工作目录，即当前python脚本工作的目录路径  
# os.chdir("dirname") 改变当前脚本工作目录；相当于shell下cd  
# os.curdir 返回当前目录: ('.')  
# os.pardir 获取当前目录的父目录字符串名：('..')  
# os.makedirs('dirname1/dirname2') 可生成多层递归目录  
# os.removedirs('dirname1') 若目录为空，则删除，并递归到上一级目录，如若也为空，则删除，依此类推  
# os.mkdir('dirname') 生成单级目录；相当于shell中mkdir dirname  
# os.rmdir('dirname') 删除单级空目录，若目录不为空则无法删除，报错；相当于shell中rmdir dirname  
# os.listdir('dirname') 列出指定目录下的所有文件和子目录，包括隐藏文件，并以列表方式打印  
# os.remove() 删除一个文件  
# os.rename("oldname","newname") 重命名文件/目录  
# os.stat('path/filename') 获取文件/目录信息  
# os.linesep 输出当前平台使用的行终止符，win下为"\t\n",Linux下为"\n"  
# os.pathsep 输出用于分割文件路径的字符串  
# os.name 输出字符串指示当前使用平台。win->'nt'; Linux->'posix'  
# os.system("bash command") 运行shell命令，直接显示  
# os.environ 获取系统环境变量  
# os.path.abspath(path) 返回path规范化的绝对路径  
# os.path.split(path) 将path分割成目录和文件名二元组返回  
# os.path.dirname(path) 返回path的目录。其实就是os.path.split(path)的第一个元素  
# os.path.basename(path) 返回path最后的文件名。如何path以／或\结尾，那么就会返回空值。即os.path.split(path)的第二个元素  
# os.path.exists(path) 如果path存在，返回True；如果path不存在，返回False  
# os.path.isabs(path) 如果path是绝对路径，返回True  
# os.path.isfile(path) 如果path是一个存在的文件，返回True。否则返回False  
# os.path.isdir(path) 如果path是一个存在的目录，则返回True。否则返回False  
# os.path.join(path1[, path2[, ...]]) 将多个路径组合后返回，第一个绝对路径之前的参数将被忽略  
# os.path.getatime(path) 返回path所指向的文件或者目录的最后存取时间  
# os.path.getmtime(path) 返回path所指向的文件或者目录的最后修改时间**

# sys模块

**#\_\_author:bo  
#date:2018/9/18  
import** sys  
print(sys.argv)**# 可以向argv中传入很多参数，在执行的时候可以判断做什么事  
# if sys.argv[1]=='post':  
# # 上传文件函数  
# elif sys.argv[1]=='download':  
  
 # download()# 执行下载函数  
  
 # sys.argv  
 # 命令行参数List，第一个元素是程序本身路径  
 # sys.exit(n)  
 # 退出程序，正常退出时exit(0)  
 # sys.version  
 # 获取Python解释程序的版本信息  
 # sys.maxint  
 # 最大的Int值  
 # sys.path**print(sys.path)  
 **# 返回模块的搜索路径，初始化时使用PYTHONPATH环境变量的值  
 # sys.platform  
import** os  
print(sys.platform)  
**if** sys.platform**=='win32':** os.system(**'dir'**)  
**elif** sys.platform**=='linux':** os.system(**'ls'**)  
 **# 返回操作系统平台名称  
 # sys.stdout.write('please:')  
 # val = sys.stdin.readline()[:-1]**

# hashlib

**import** hashlib  
m**=**hashlib.md5()  
m.update(**'aaa'**.encode(**'utf-8'**))  
print(m.hexdigest())**# 47bce5c74f589f4867dbd57e9ca9f808**m.update(**'bbb'**.encode(**'utf-8'**))  
print(m.hexdigest())**# 6547436690a26a399603a7096e876a2d，相当于是将上边的aaa和bbb加起来进行md5**m2**=**hashlib.md5()  
m2.update(**'aaabbb'**.encode(**'utf-8'**))  
print(m2.hexdigest())**# 6547436690a26a399603a7096e876a2d,这个和上边的是一样的**sha**=**hashlib.sha256()  
sha.update(**'aaa'**.encode(**'utf-8'**))  
print(sha.hexdigest())**# 9834876dcfb05cb167a5c24953eba58c4ac89b1adf57f28f2f9d09af107ee8f0**

# logging

默认情况下[Python](http://lib.csdn.net/base/11)的logging模块将日志打印到了标准输出中，且只显示了大于等于WARNING级别的日志，这说明默认的日志级别设置为WARNING（日志级别等级CRITICAL > ERROR > WARNING > INFO > DEBUG > NOTSET），默认的日志格式为日志级别：Logger名称：用户输出消息。

import logging

logging.basicConfig(level=logging.DEBUG,

format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s',

datefmt='%a, %d %b %Y %H:%M:%S',

filename='/tmp/test.log',

filemode='w')

logging.debug('debug message')

logging.info('info message')

logging.warning('warning message')

logging.error('error message')

logging.critical('critical message')

查看输出：  
cat /tmp/test.log   
Mon, 05 May 2014 16:29:53 test\_logging.py[line:9] DEBUG debug message  
Mon, 05 May 2014 16:29:53 test\_logging.py[line:10] INFO info message  
Mon, 05 May 2014 16:29:53 test\_logging.py[line:11] WARNING warning message  
Mon, 05 May 2014 16:29:53 test\_logging.py[line:12] ERROR error message  
Mon, 05 May 2014 16:29:53 test\_logging.py[line:13] CRITICAL critical message  
  
可见在logging.basicConfig()函数中可通过具体参数来更改logging模块默认行为，可用参数有  
filename：用指定的文件名创建FiledHandler（后边会具体讲解handler的概念），这样日志会被存储在指定的文件中。  
filemode：文件打开方式，在指定了filename时使用这个参数，默认值为“a”还可指定为“w”。  
format：指定handler使用的日志显示格式。   
datefmt：指定日期时间格式。   
level：设置rootlogger（后边会讲解具体概念）的日志级别   
stream：用指定的stream创建StreamHandler。可以指定输出到sys.stderr,sys.stdout或者文件(f=open('test.log','w'))，默认为sys.stderr。若同时列出了filename和stream两个参数，则stream参数会被忽略。  
  
format参数中可能用到的格式化串：  
%(name)s Logger的名字  
%(levelno)s 数字形式的日志级别  
%(levelname)s 文本形式的日志级别  
%(pathname)s 调用日志输出函数的模块的完整路径名，可能没有  
%(filename)s 调用日志输出函数的模块的文件名  
%(module)s 调用日志输出函数的模块名  
%(funcName)s 调用日志输出函数的函数名  
%(lineno)d 调用日志输出函数的语句所在的代码行  
%(created)f 当前时间，用UNIX标准的表示时间的浮 点数表示  
%(relativeCreated)d 输出日志信息时的，自Logger创建以 来的毫秒数  
%(asctime)s 字符串形式的当前时间。默认格式是 “2003-07-08 16:49:45,896”。逗号后面的是毫秒  
%(thread)d 线程ID。可能没有  
%(threadName)s 线程名。可能没有  
%(process)d 进程ID。可能没有  
%(message)s用户输出的消息

**#\_\_author:bo  
#date:2018/9/19  
import** logging  
  
  
**# 日志是有级别的**logging.basicConfig(level**=**logging.DEBUG,  
 format**='%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s'**,**# 日志打印的内容  
# Fri, 21 Sep 2018 13:37:39 loggin1g.py[line:13] DEBUG debug message** datefmt**='%a, %d %b %Y %H:%M:%S'**,**# 时间的配置** filename**='test.log'**,**# 将日志放到文件中如果没有文件名，则会默认屏幕输出**

filemode**='a'**)**# w每次清除以前的，a是每次追加**logging.debug(**'debug message'**)  
logging.info(**'info message'**)  
logging.warning(**'warning message'**)  
logging.error(**'error message'**)  
logging.critical(**'critical message'**)

## 既输出到文件，又打印到屏幕

**#\_\_author:bo  
#date:2018/9/21  
  
import** logging  
  
logger**=**logging.getLogger()  
**# 创建Handler，用于写入日志文件**fh**=**logging.FileHandler(**'test1.log'**)  
**# 在创建一个Handler，用于输出到控制台**ch**=**logging.StreamHandler()**# 用于输出到屏幕**formatter**=**logging.Formatter(**'%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s'**)  
ch.setFormatter(formatter)  
fh.setFormatter(formatter)**# 给它们两个设置自己的输出格式**logger.addHandler(fh)  
logger.addHandler(ch)**# 将fh ch加入到logger对象**logger.setLevel(logging.DEBUG)**# 修改日志级别**logger.debug(**'debug message22222'**)  
logger.info(**'info message2222'**)  
logger.warning(**'warning message2222'**)  
logger.error(**'error message2222'**)  
logger.critical(**'critical message2222'**)**# 调用logger对象的方法，会在屏幕和文件中都输出日志  
  
# 上述几个例子中我们了解到了logging.debug()、logging.info()、logging.warning()、  
# logging.error()、logging.critical()（分别用以记录不同级别的日志信息），  
# logging.basicConfig()（用默认日志格式（Formatter）为日志系统建立一个默认的流处理器  
# （StreamHandler），设置基础配置（如日志级别等）并加到root logger（根Logger）中）  
# 这几个logging模块级别的函数，另外还有一个模块级别的函数是logging.getLogger([name])  
# （返回一个logger对象，如果没有指定名字将返回root logger）**

三  logger对象

    上述几个例子中我们了解到了logging.debug()、logging.info()、logging.warning()、logging.error()、logging.critical()（分别用以记录不同级别的日志信息），logging.basicConfig()（用默认日志格式（Formatter）为日志系统建立一个默认的流处理器（StreamHandler），设置基础配置（如日志级别等）并加到root logger（根Logger）中）这几个logging模块级别的函数，另外还有一个模块级别的函数是logging.getLogger([name])（返回一个logger对象，如果没有指定名字将返回root logger）

     先看一个最简单的过程：

[复制代码](javascript:void(0);)

import logging

logger = logging.getLogger()

# 创建一个handler，用于写入日志文件

fh = logging.FileHandler('test.log')

# 再创建一个handler，用于输出到控制台

ch = logging.StreamHandler()

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

fh.setFormatter(formatter)

ch.setFormatter(formatter)

logger.addHandler(fh) #logger对象可以添加多个fh和ch对象

logger.addHandler(ch)

logger.debug('logger debug message')

logger.info('logger info message')

logger.warning('logger warning message')

logger.error('logger error message')

logger.critical('logger critical message')

[复制代码](javascript:void(0);)

      先简单介绍一下，logging库提供了多个组件：Logger、Handler、Filter、Formatter。Logger对象提供应用程序可直接使用的接口，Handler发送日志到适当的目的地，Filter提供了过滤日志信息的方法，Formatter指定日志显示格式。

     (1)

      Logger是一个树形层级结构，输出信息之前都要获得一个Logger（如果没有显示的获取则自动创建并使用root Logger，如第一个例子所示）。  
      logger = logging.getLogger()返回一个默认的Logger也即root Logger，并应用默认的日志级别、Handler和Formatter设置。  
当然也可以通过Logger.setLevel(lel)指定最低的日志级别，可用的日志级别有logging.DEBUG、logging.INFO、logging.WARNING、logging.ERROR、logging.CRITICAL。  
      Logger.debug()、Logger.info()、Logger.warning()、Logger.error()、Logger.critical()输出不同级别的日志，只有日志等级大于或等于设置的日志级别的日志才会被输出。

logger.debug('logger debug message')

logger.info('logger info message')

logger.warning('logger warning message')

logger.error('logger error message')

logger.critical('logger critical message')

只输出了  
2014-05-06 12:54:43,222 - root - WARNING - logger warning message  
2014-05-06 12:54:43,223 - root - ERROR - logger error message  
2014-05-06 12:54:43,224 - root - CRITICAL - logger critical message  
     从这个输出可以看出logger = logging.getLogger()返回的Logger名为root。这里没有用logger.setLevel(logging.Debug)显示的为logger设置日志级别，所以使用默认的日志级别WARNIING，故结果只输出了大于等于WARNIING级别的信息。

     (2) 如果我们再创建两个logger对象：

[复制代码](javascript:void(0);)

##################################################

logger1 = logging.getLogger('mylogger')

logger1.setLevel(logging.DEBUG)

logger2 = logging.getLogger('mylogger')

logger2.setLevel(logging.INFO)

logger1.addHandler(fh)

logger1.addHandler(ch)

logger2.addHandler(fh)

logger2.addHandler(ch)

logger1.debug('logger1 debug message')

logger1.info('logger1 info message')

logger1.warning('logger1 warning message')

logger1.error('logger1 error message')

logger1.critical('logger1 critical message')

logger2.debug('logger2 debug message')

logger2.info('logger2 info message')

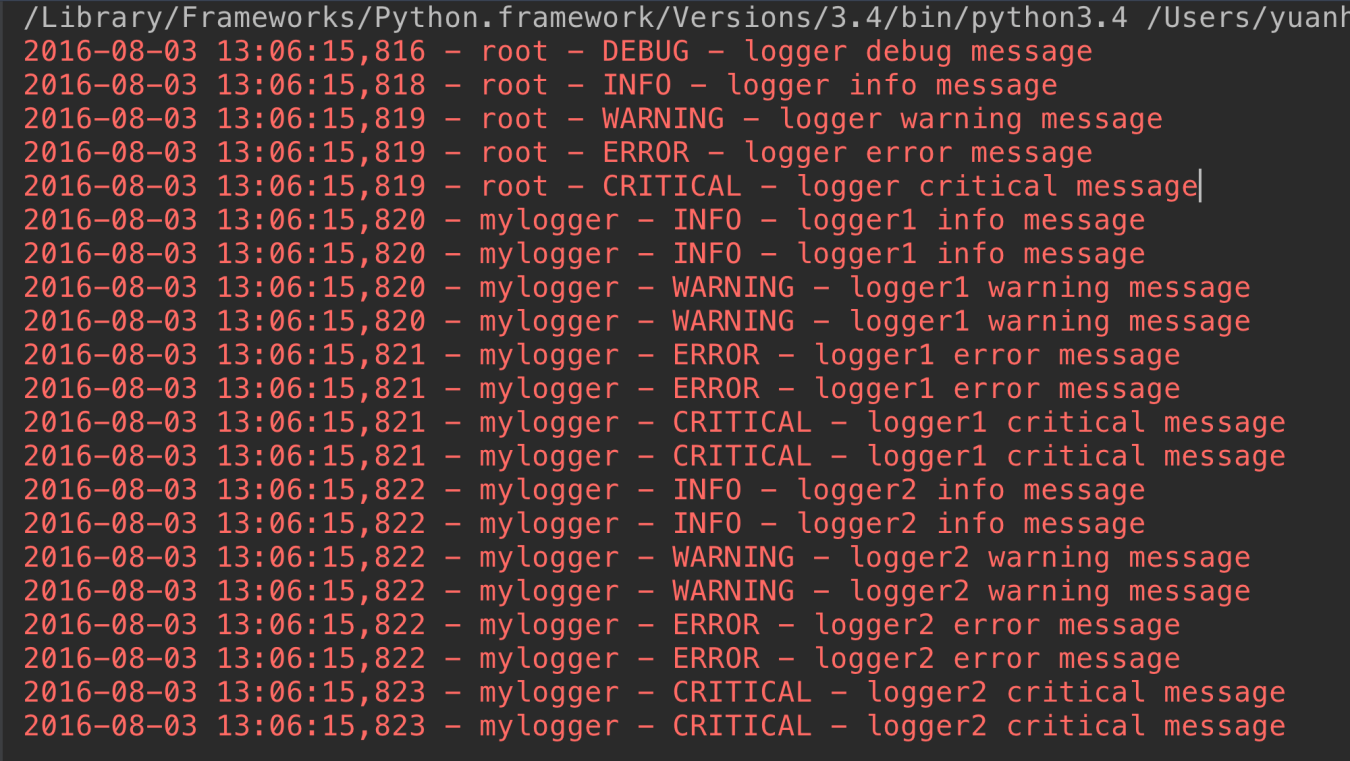
logger2.warning('logger2 warning message')

logger2.error('logger2 error message')

logger2.critical('logger2 critical message')

[复制代码](javascript:void(0);)

结果：



这里有两个个问题：

      <1>我们明明通过logger1.setLevel(logging.DEBUG)将logger1的日志级别设置为了DEBUG，为何显示的时候没有显示出DEBUG级别的日志信息，而是从INFO级别的日志开始显示呢？

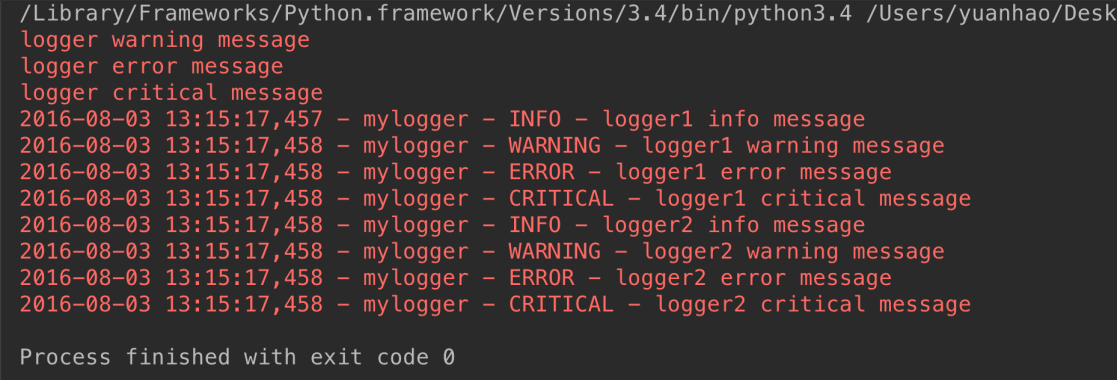
       原来logger1和logger2对应的是同一个Logger实例，只要logging.getLogger（name）中名称参数name相同则返回的Logger实例就是同一个，且仅有一个，也即name与Logger实例一一对应。在logger2实例中通过logger2.setLevel(logging.INFO)设置mylogger的日志级别为logging.INFO，所以最后logger1的输出遵从了后来设置的日志级别。

      <2>为什么logger1、logger2对应的每个输出分别显示两次?  
       这是因为我们通过logger = logging.getLogger()显示的创建了root Logger，而logger1 = logging.getLogger('mylogger')创建了root Logger的孩子(root.)mylogger,logger2同样。而孩子,孙子，重孙……既会将消息分发给他的handler进行处理也会传递给所有的祖先Logger处理。

        ok,那么现在我们把

# logger.addHandler(fh)

# logger.addHandler(ch)  注释掉，我们再来看效果：

**

*因为我们注释了logger对象显示的位置，所以才用了默认方式，即标准输出方式。*因为它的父级没有设置文件显示方式，所以在这里只打印了一次。

孩子,孙子，重孙……可逐层继承来自祖先的日志级别、Handler、Filter设置，也可以通过Logger.setLevel(lel)、Logger.addHandler(hdlr)、Logger.removeHandler(hdlr)、Logger.addFilter(filt)、Logger.removeFilter(filt)。设置自己特别的日志级别、Handler、Filter。若不设置则使用继承来的值。

<3>**Filter**  
     限制只有满足过滤规则的日志才会输出。  
     比如我们定义了filter = logging.Filter('a.b.c'),并将这个Filter添加到了一个Handler上，则使用该Handler的Logger中只有名字带          a.b.c前缀的Logger才能输出其日志。

     filter = logging.Filter('mylogger')

     logger.addFilter(filter)

     这是只对logger这个对象进行筛选

     如果想对所有的对象进行筛选，则：

      filter = logging.Filter('mylogger')

      fh.addFilter(filter)

      ch.addFilter(filter)

*这样，所有添加fh或者ch的logger对象都会进行筛选。*

## 完整代码1

**import** logging  
  
logger **=** logging.getLogger()  
**# 创建一个handler，用于写入日志文件**fh **=** logging.FileHandler(**'test.log'**)  
  
**# 再创建一个handler，用于输出到控制台**ch **=** logging.StreamHandler()  
  
formatter **=** logging.Formatter(**'%(asctime)s - %(name)s - %(levelname)s - %(message)s'**)  
  
fh.setFormatter(formatter)  
ch.setFormatter(formatter)  
  
**# 定义一个filter**filter **=** logging.Filter(**'mylogger'**)  
fh.addFilter(filter)  
ch.addFilter(filter)  
  
**# logger.addFilter(filter)**logger.addHandler(fh)  
logger.addHandler(ch)  
  
  
  
  
logger.setLevel(logging.DEBUG)  
  
logger.debug(**'logger debug message'**)  
logger.info(**'logger info message'**)  
logger.warning(**'logger warning message'**)  
logger.error(**'logger error message'**)  
logger.critical(**'logger critical message'**)  
  
**##################################################**logger1 **=** logging.getLogger(**'mylogger'**)  
logger1.setLevel(logging.DEBUG)  
  
logger2 **=** logging.getLogger(**'mylogger'**)  
logger2.setLevel(logging.INFO)  
  
logger1.addHandler(fh)  
logger1.addHandler(ch)  
  
logger2.addHandler(fh)  
logger2.addHandler(ch)  
  
logger1.debug(**'logger1 debug message'**)  
logger1.info(**'logger1 info message'**)  
logger1.warning(**'logger1 warning message'**)  
logger1.error(**'logger1 error message'**)  
logger1.critical(**'logger1 critical message'**)  
  
logger2.debug(**'logger2 debug message'**)  
logger2.info(**'logger2 info message'**)  
logger2.warning(**'logger2 warning message'**)  
logger2.error(**'logger2 error message'**)  
logger2.critical(**'logger2 critical message'**)

## 完整代码2

**# coding:utf-8  
import** logging  
  
**# 创建一个logger**logger **=** logging.getLogger()  
  
logger1 **=** logging.getLogger(**'mylogger'**)  
logger1.setLevel(logging.DEBUG)  
  
logger2 **=** logging.getLogger(**'mylogger'**)  
logger2.setLevel(logging.INFO)  
  
logger3 **=** logging.getLogger(**'mylogger.child1'**)  
logger3.setLevel(logging.WARNING)  
  
logger4 **=** logging.getLogger(**'mylogger.child1.child2'**)  
logger4.setLevel(logging.DEBUG)  
  
logger5 **=** logging.getLogger(**'mylogger.child1.child2.child3'**)  
logger5.setLevel(logging.DEBUG)  
  
**# 创建一个handler，用于写入日志文件**fh **=** logging.FileHandler(**'/tmp/test.log'**)  
  
**# 再创建一个handler，用于输出到控制台**ch **=** logging.StreamHandler()  
  
**# 定义handler的输出格式formatter**formatter **=** logging.Formatter(**'%(asctime)s - %(name)s - %(levelname)s - %(message)s'**)  
fh.setFormatter(formatter)  
ch.setFormatter(formatter)  
  
**# 定义一个filter  
# filter = logging.Filter('mylogger.child1.child2')  
# fh.addFilter(filter)  
  
# 给logger添加handler  
# logger.addFilter(filter)**logger.addHandler(fh)  
logger.addHandler(ch)  
  
**# logger1.addFilter(filter)**logger1.addHandler(fh)  
logger1.addHandler(ch)  
  
logger2.addHandler(fh)  
logger2.addHandler(ch)  
  
**# logger3.addFilter(filter)**logger3.addHandler(fh)  
logger3.addHandler(ch)  
  
**# logger4.addFilter(filter)**logger4.addHandler(fh)  
logger4.addHandler(ch)  
  
logger5.addHandler(fh)  
logger5.addHandler(ch)  
  
**# 记录一条日志**logger.debug(**'logger debug message'**)  
logger.info(**'logger info message'**)  
logger.warning(**'logger warning message'**)  
logger.error(**'logger error message'**)  
logger.critical(**'logger critical message'**)  
  
logger1.debug(**'logger1 debug message'**)  
logger1.info(**'logger1 info message'**)  
logger1.warning(**'logger1 warning message'**)  
logger1.error(**'logger1 error message'**)  
logger1.critical(**'logger1 critical message'**)  
  
logger2.debug(**'logger2 debug message'**)  
logger2.info(**'logger2 info message'**)  
logger2.warning(**'logger2 warning message'**)  
logger2.error(**'logger2 error message'**)  
logger2.critical(**'logger2 critical message'**)  
  
logger3.debug(**'logger3 debug message'**)  
logger3.info(**'logger3 info message'**)  
logger3.warning(**'logger3 warning message'**)  
logger3.error(**'logger3 error message'**)  
logger3.critical(**'logger3 critical message'**)  
  
logger4.debug(**'logger4 debug message'**)  
logger4.info(**'logger4 info message'**)  
logger4.warning(**'logger4 warning message'**)  
logger4.error(**'logger4 error message'**)  
logger4.critical(**'logger4 critical message'**)  
  
logger5.debug(**'logger5 debug message'**)  
logger5.info(**'logger5 info message'**)  
logger5.warning(**'logger5 warning message'**)  
logger5.error(**'logger5 error message'**)  
logger5.critical(**'logger5 critical message'**)

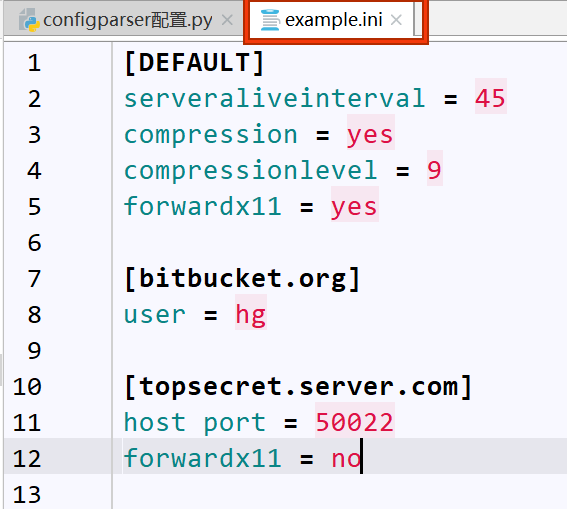
## ATM应用

**import** os  
**import** time  
**import** logging  
**from** config **import** settings  
  
  
**def get\_logger**(*card\_num*, *struct\_time*)**:  
  
 if** *struct\_time*.tm\_mday **<** 23**:** file\_name **= "%s\_%s\_%d" %**(*struct\_time*.tm\_year, *struct\_time*.tm\_mon, 22)  
 **else:** file\_name **= "%s\_%s\_%d" %**(*struct\_time*.tm\_year, *struct\_time*.tm\_mon**+**1, 22)  
 file\_handler **=** logging.FileHandler(  
 os.path.join(settings.USER\_DIR\_FOLDER, *card\_num*, **'record'**, file\_name),  
 encoding**='utf-8'** )  
 fmt **=** logging.Formatter(fmt**="%(asctime)s : %(message)s"**)  
 file\_handler.setFormatter(fmt)  
 logger1 **=** logging.Logger(**'user\_logger'**, level**=**logging.INFO)  
 logger1.addHandler(file\_handler)  
 **return** logger1

# ConfigParser模块

**import** configparser  
config **=** configparser.ConfigParser()  
**# config["DEFAULT"] = {'ServerAliveInterval': '45',  
# 'Compression': 'yes',  
# 'CompressionLevel': '9'}  
# config['bitbucket.org'] = {'User':'hg'}  
# config['topsecret.server.com'] = {}  
# topsecret = config['topsecret.server.com']  
# topsecret['Host Port'] = '50022' # mutates the parser  
# topsecret['ForwardX11'] = 'no' # same here  
# config['DEFAULT']['ForwardX11'] = 'yes'  
# with open('example.ini', 'w') as configfile:# 创建一个文件  
# config.write(configfile)# 必须要有这句，才能生成配置文件，所有的生成的文件是configparser内部实现的**config.read(**'example.ini'**)**# 先用config读出来一个文件，再进行拿出其中的内容**print(config.sections())**# DEFAULT是不会被拿出来的**print(config.defaults())**# 拿到DEFAULT项**print(config[**'bitbucket.org'**][**'user'**])**# 打印出hg  
for** key **in** config**:** print(key)**# 打印出config对象中的key**print(**'=================='**)  
**for** key **in** config[**'topsecret.server.com'**]**:** print(key)**# 打印出config['topsecret.server.com']中的key,default的内容也会自动打印**

根据字典自动生成配置文件，可以很方便的进行配置：结果如下



## 增删改

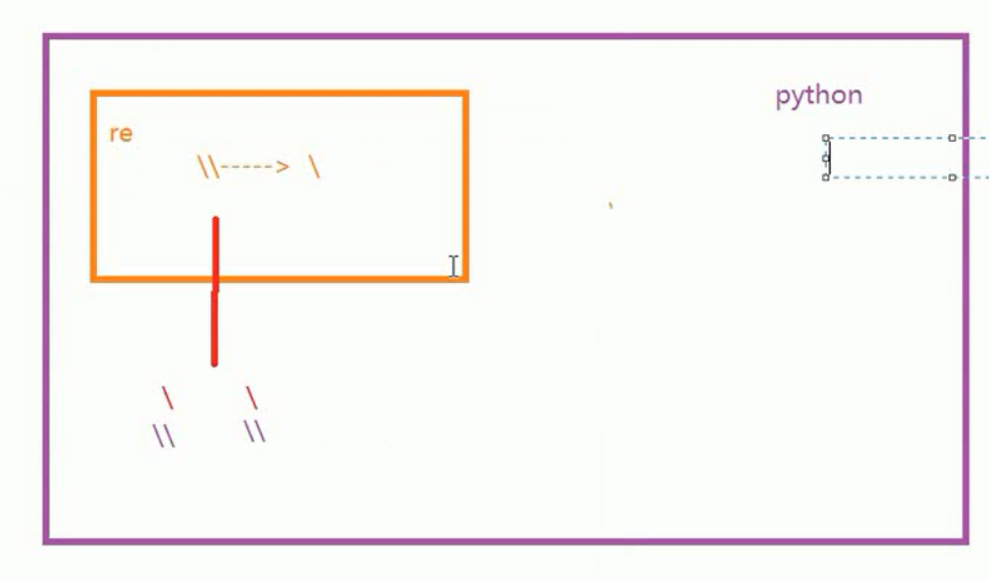
config.read(**'example.ini'**)  
config.remove\_section(**'bitbucket.org'**)**# 删除某个块**config.remove\_option(**'topsecret.server.com'**,**'host port'**)**# 删除某个块下边的键**config.write(open(**'shanchu.ini'**,**'w'**))**# 只能再生成一个文件，将删除后的内容写入到另外一个文件中达到删除的效果，不过也可以写成与原来的文件名一样来覆盖掉原来的文件**print(config.has\_section(**'topsecret.server.com'**))**# 已经将config对象的topsecret.server.com删除，所以会返回false**config.set(**'topsecret.server.com'**,**'forwardx11'**,**'sssssssss'**) **# 修改**

config.write(open(**'set.ini'**,**'w'**))**# 修改后保存文件**

# re模块

**import** re  
  
ret**=**re.findall(**'wbs'**,**'adasfaswbsasdasfd'**)  
print(ret)**# 可以完全匹配，将匹配的全部结果以列表的形式返回  
  
# . 除了换行符之外的都可以代指，只能代指一个东西  
# ^ 尖角只对开始的字符匹配，而不对后边的字符匹配，即使后边的可以匹配上  
# $ 只对结尾的进行匹配  
# \* 匹配重复\*前边出现的0~无穷个内容，\*可以匹配0次，贪婪匹配  
# + 匹配1~无穷个  
# ? 匹配0个或者1个  
# {} 匹配括弧内的个数  
# {1,3} 匹配1~3次,贪婪匹配  
# {1,} 匹配1~无穷次,贪婪匹配  
# [a,b] 匹配中括弧中出现的元素，可以是中括弧中的任意一个元素  
# [a-z] 匹配a~z的任何  
# []字符集的特殊功能，取消元字符的特殊功能（\ ^ - 这三个字符例外）  
# [^] 尖角号放在中括弧中是对括弧中的所又元素取反的意思  
# | 管道符是或者的意思  
# \ 反斜杠后边跟元字符取出特数功能，后边跟普通字符实现特数功能  
# \d 匹配任何十进制数，相当于[0-9]  
# \D 匹配任何非数字字符，相当于[^0-9]  
# \s 匹配任何空白字符，相当于[\t\n\r\f\v]  
# \S 匹配任何非空字符，相当于[^\t\n\r\f\v]  
# \w 匹配任何字母数字字符，相当于[a-zA-Z0-9]  
# \W 匹配任何非字母数字字符，相当于[^a-zA-Z0-9]  
# \b 匹配一个特殊字符的边界，也就是指单词和空格间的位置**ret**=**re.findall(**'w\w{2}l'**,**'helloworld'**)  
print(ret)  
ret**=**re.findall(**'h...o'**,**'helloworld'**)  
print(**'...'**,ret)  
ret**=**re.findall(**'^h...o'**,**'helloworld'**)  
print(**'^^^'**,ret)  
ret**=**re.findall(**'h...o$'**,**'worldhello'**)  
print(**'$$$'**,ret)  
ret**=**re.findall(**'ba\*'**,**'ddbaaaaaa'**)  
print(**'\*\*\*\*\*\*'**,ret)  
ret**=**re.findall(**'a+b'**,**'ddbabaabaaa'**)  
print(**'+++++'**,ret)  
ret**=**re.findall(**'a?b'**,**'ddabbbaab'**)  
print(**'?????'**,ret)  
ret**=**re.findall(**'ab{5}'**,**'ddaaaaabbbbbaab'**)  
print(**'{}{}'**,ret)  
ret**=**re.findall(**'ab{1,3} '**,**'ababbabbbabbbb'**)  
print(**'{}{}'**,ret)  
ret**=**re.findall(**'c[om,n] '**,**'cocm'**)**# ['co', 'cm']**print(**'[om,n]'**,ret)  
ret**=**re.findall(**'c[0-9a-zA-Z] '**,**'cbcacz'**)**# ['cb', 'ca', 'cz']**print(**'a-z '**,ret)  
ret**=**re.findall(**'[a,\*,.,,]'**,**'aa\*.,.\*\*'**)**# ['a', 'a', '\*', '\*', '\*']此时的\* . , 都代表普通的字符**print(**'aaa\*\*\* '**,ret)  
ret**=**re.findall(**'[^4,5]'**,**'456'**)  
print(**'4545'**,ret)  
ret**=**re.findall(**r'I\b'**,**'I am a LI$ST'**)**# 此时的\b是一个特殊字符（比如空格，% $ 等等）的边界**print(**r'\b\b\b'**,ret)  
**######################################**ret**=**re.search(**'wbs'**,**'asdfawbsafdas'**)  
print(ret)**# <\_sre.SRE\_Match object; span=(5, 8), match='wbs'>**print(ret.group())**# 打印出来匹配到的内容**ret**=**re.search(**'wbs\\\\.'**,**'wbs\.aa'**)**# 此时的\是python的转义字符和re模块的转义字符**print(ret.group())  
ret**=**re.findall(**r'wbs\\c'**,**'wbs\caa'**)**# 此时的\是一个re模块中的转义字符**print(ret)

解释\\\\的原因或者前边匹配的时候加r的原因：



首先python解释器将四个斜杠\\\\解释成两个[\\，re](file:///\\，re)模块再将其两个斜杠\\解释成为一个斜杠来匹配字符串，而前边如果加了r的时候，会将其中的字符串全部认为是普通的字符串，所以两个斜杠\\会被re模块解释成一个斜杠\

## ()分组

ret**=**re.search(**'(as)+'**,**'djfeiasas'**).group()**# ()作为一个整体的内容**print(ret)  
ret**=**re.search(**'(as)| 3'**,**'3as'**).group()**# 谁先出来谁先匹配**print(ret)  
print(**'========================'**)  
ret**=**re.search(**'(?P<id>\d{3})'**,**'alsdjf234asdfa'**)**# 想要给正则起名字，必须要用?P, ?P是固定的写法，可以识别<>中的东西**print(ret.group())  
print(ret.group(**'id'**))  
ret**=**re.search(**'(?P<id>\d{3})/(?P<name>\w{3})'**,**'alsdjf234/wbs'**)**# ?P是固定的写法，可以识别<>中的东西**print(ret.group())**# 234/wbs**print(ret.group(**'name'**))# 结果是wbs

正则表达式的方法：

1. findall():所有结果都返回到一个列表里面
2. search():返回匹配的第一个对象object，对象里可以调用group()方法返回结果
3. match():只在字符串开始匹配，返回一个对象
4. split():分割
5. sub():替换
6. compile():编译的规则
7. ret**=**re.match(**'as'**,**'asasas'**)  
   print(ret)**# <\_sre.SRE\_Match object; span=(0, 2), match='as'>**print(ret.group())  
   ret**=**re.split(**'[k,s]'**,**'dkegsd'**)**# 先通过k分，分完后再通过s分，**print(ret)  
   ret**=**re.split(**'[k,s]'**,**'dkegs'**)**#最后一次通过s分开后，最后被分开的是一个空字符串**print(ret)**# ['d', 'eg', '']**ret**=**re.sub(**'w..b'**,**'wbs'**,**'dddwaab'**)**# 替换**print(ret)**# dddwbs**
8. obj**=**re.compile(**'\.com'**)**# 先编译一个规则**ret**=**obj.findall(**'wbs.com'**)**# 用编译的规则匹配**print(ret)