声明空的对象

s=set() #空set(集合是不包含value的字典)

dic=dict() #空字典

s=str()

i=int() #空int是0

lst=list() #空列表

tu=tuple()#空元组

s=str()  
print(type(s))  
i=int()  
print(type(i))  
dic=dict()  
print(type(dic))  
tu=tuple()  
print(type(tu))  
dic=dict()  
print(type(dic))  
se=set()  
print(type(se))

**python**

**day1-01(01)**

**参数**

cpu:主频+核数

显卡：显存（运算能力）+型号（N-GTX 1080 TI，A）+位宽（进出的数据量） 型号看8这个数越大性能越高 10是年代

内存：大小，型号，主频

**目前Python主要应用领域**

1. 云计算:OpenStack
2. WEB开发:Youtube
3. 科学计算:NumPy,SciPy,Matplotlib,Enthoughtlibrarys,pandas
4. 系统运维
5. 金融
6. 图形GUI:PyQT,WxPython,TkInter

**Python在一些公司的应用**

1. Google爬虫
2. YouTuBe
3. Facebook
4. Redhat
5. 豆瓣
6. 知乎

**编译型和解释型**

**day1-02(02)**

**解释器**

**CPython**

当我们从Python官⽅⽹站下载并安装好Python 2.7后，我们就直接获得了一个官⽅版 本的解释器器：CPython。这个解释器器是用C语言开发的，所以叫CPython。在命令行下运 ⾏ python就是启动CPython解释器器。 CPython是使⽤最⼴的Python解释器器。教程的所有代码也都在CPython下执行。

**IPython**

IPython是基于CPython之上的一个交互式解释器器，也就是说，IPython只是在交互⽅式上有所增强，但是执行Python代码的功能和CPython是完全⼀一样的。好比很多国产浏览 器器虽然外观不不同，但内核其实都是调⽤了IE。 CPython⽤>>>作为提示符，⽽IPython⽤In [序号]:作为提示符。

**PyPy**

PyPy是另一个Python解释器器，它的目标是执⾏速度。PyPy采用JIT技术，对Python代 码进⾏动态编译（注意不是解释），所以可以显著提高Python代码的执⾏速度。 绝大部分Python代码都可以在PyPy下运行，但是PyPy和CPython有一些是不同的，这就导致相同的Python代码在两种解释器器下执⾏可能会有不同的结果。如果你的代码要放到 PyPy下执行，就需要了了解PyPy和CPython的不不同点。

**Jython**

Jython是运⾏在Java平台上的Python解释器器，可以直接把Python代码编译成Java字节 码执⾏。 IronPython和Jython类似，只不过IronPython是运⾏在微软.Net平台上的Python解释器，可以直接把Python代码编译成.Net的字节码。

**第一个python程序**

使用记事本 格式设置成UTF-8无bom格式

cd/ windows下跳到跟目录

**变量**

变量量的命名规则: 1, 变量量由字⺟母, 数字,下划线搭配组合⽽而成（将\_当字母用） 2, 不可以用数字开头,更不能是全数字 3,不能是pythond的关键字, 这些符号和字母已经被python占用, 不可以更改 4,不要用中文 5,名字要有意义 6,不要太长 7, 区分大小写 推荐大家使用驼峰体或者下划线命名 驼峰体: 除⾸首字母外的其他每个单词首字母大写 下划线: 每个单词之间用下划线分开

**常量**

在python中不存在绝对的常量. 约定俗成, 所有字母大写就是常量

**注释**

单行注释: # 被注释的内容 多行注释:''' 被注释的内容 ''', """这个也是多⾏注释"""

**python的基本数据类型**

我们⼈人类可以很容易易的分清数字与字符的区别，但是计算机并不不能呀，计算机虽然很强⼤大，但从某种⻆角度上看⼜又很傻，除⾮非你明确的告诉它，1是数字，“汉”是⽂文字，否则它是分不不清1和‘汉’的区别的，因此，在每个编程语⾔言⾥里里都会有⼀个叫数据类型的东东，其实就是对常用的各种数据类型进行了明确的划分，你想让计算机进行数值运算，你就传数字给它，你想让他处理文字，就传字符串类型给他。Python中常用的数据类型有多种，今天我们暂只讲3种， 数字、字符串串、布尔类型(命题).

**int**

**str**

在Python中,凡是⽤用引号引起来的,全是字符串串.

msg = "My name is Alex , I'm 22 years old!" 这个就需要单双引号配合。

msg = """ 今天我想写⾸首⼩小诗，

歌颂我的同桌，

你看他那乌⿊黑的短发，

好像⼀一只炸⽑毛鸡。 """ 想多⾏行行赋值⼀一个字符串串，就需要三引号。

数字类型有 +-\*/ 字符串串有么？字符串串只有 + *\*(+是字符串拼接，\**多次输出)

**bool**

转换成false的数据

0，‘’，None,[],(),{},set()==🡺false

**用户交互**

a=input()得到的是一个字符串

**day1-03(03)**

if ：

elif:

elif:

这种if语句只会执行一个分支

**day02-01(04)**

**作业**

**day02-02(05)**

**while**

**day02-03(06)**

break中断循环 跳到循环外面

continue终止本次循环 继续执行下一次循环

while :

else: (注意else是while条件不成立执行，可能else不执行当中途出现break时)

**day02-04(07)**

**day02-05(08)**

循环求基数

**day02-06(09)**

**格式化输出**

print(name+"今年"+age)# 字符串做加法运算时会出现内存的消耗，每次加一个会产生一个新的字符串

s=”我叫{},我今年{}”.format(“wdd”,18) #format中的值可以是变量名

新玩法

s=”我叫{1},我今年{0}”.format(“wdd”,18)

s=”我叫{name},我今年{age}”.format(name=“wdd”,age=18)

**字符串占位符**

Print(“%s” %())写法字符串里用%s占位然后空格%（）按位置赋值

a=108

s=”梁山有%s将” %（a） 这样也可以用（注意第二个%号前有空格）

如果字符串中没有占位符，%就是%,但是字符串中只要有一个%是占位符所有的%号都是占位符（即使是%2 也是占位符需要用%%转义）

**day02-07(10)**

**算数运算**

幂运算 2\*\*3=8

**比较运算符**

！=和<> 是不等于

**赋值运算**

**逻辑运算**

and (左右两段的值必须是真 运算结果才是真)

or

not

and or not 同时存在 先not 让后and 最后or

重点部分(直接数字比较 没有大于号和小于号)

x or y 如果x==0,那么就是用y，否则是x

print(1 or 2 ) #1

print(2 or 3) #2

print(0 or 3) #3

print(0 or 4) #4

x and y 结果与上面相反

print(1 and 2 ) #2

print(2 and 3) #3

print(0 and 3) #0

print(0 and 4) #0

print(0 or 1 or 3 or 0or 5) #1 0 or 1=1->1 or 3=1->1 or 0=1->1 or 5=1

print(1 or 2 and 3)#1 先and 后or

print(0 or 4 and 3 or 7 or 9 and 6) 0 or 3 or 7 or 6 #3

print(2>3 and 3) #fasle false相当于0

print(2<1 and 4>6 or 3 and 4>5 or 6)#6

**day03-01(11)**

**day03-02(12)**

**作业**

**day03-03（13）**

Ascii->gbk(ANSII就是gbk 2字节)->Unicode(4字节浪费空间)-》utf-8(可变长度的unicode最低2字节)

英文（1个字节）

欧洲文字（2个字节）

中文（3个字节）

**day03-04(14)**

**基本数据类型**

**Int**

bit\_length().计算整数的二进制数长度

i=8

i.bit\_length() #4

**Boolean**

把int转成bool a=0 b=bool(a)#false

True是4个字节 所以1的效率比true效率高点

空字符串为false 空格是True(对于字符串来说，空字符串都是false，非空字符串都是true)

a=None# false 空连空气都不如，是真空

**str**

##### 索引和切片

索引就是下标，切记，小标从0开始（只有数据库是从1开始，其它事从0开始）

字符串倒着取（字符串最后一个位置的index是-1）

切片可以对字符串进行截取 语法 s[起始位置（默认到开头）：结束位置（默认到结尾）：步长（默认是0）]（左开右闭）

S[:]从头到尾都切出来

默认是从左往右切

S[-2:]这样可以切到最后两个字符 不能s[-2:0]这样的话0在左-2在右切不了

s[::-1]将字符串倒序

s="我是梅西，我很慌"  
s1=s[6:2:-3] #倒着取 很西

**总结s[start:end:step] index看成一个环 包括start不包括end start和end不管正负（但start和end必须同为正或同为负）**

**step为正则从左往右切start必须小于end才切出值来**

**step为负则从右往左切start必须大于end才切出值来**

##### 重要概念

切记，字符串是不可变的对象，所以任何操作对原字符串是不会有任何影响的（即一旦给一个字符串赋值了，这个值不能更改，我们重新赋值实际上是将引用指向新的字符串）

##### 操作

s1=s.capitalize() #首字母大写，会返回一个新字符串

print(s1) 这个时候发现将s的首字母变成大写（注意s.capitalize()必须找个变量引用起来因为字符串一旦定义好就不能更改）

tmp=s.lower() #所以字符小写，返回一个新字符串

tmp=s.upper() #所以字符大写，返回一个新字符串

tmp=s.swapcse() #大小写互换 返回一个新字符串

tmp=s.casefold()#转换成小写，和lower（）的区别是lower()对某些字符支持不够好，casefold()支持所有

tmp=s.title() #被特殊字符（空格或中文）隔开的首字母大写（注意这里的特殊字符不会删除）

tmp=s.center(8,”\*”)#居中显示空白部分用\*号显示

tmp=s.strip() #去空格

tmp=s.strip(“指定内容”)#去掉字符串左右两段所有指定的部分（注意strip只去左右两段的内容，即使指定的内容重复话重复的内容也都去了）

tmp=s.lstrip()

tmp=s.rstrip()

tmp=s.replace(“原来的”,”新的”,count)#替换 count默认是所有 （如果指定count则从头开始跟换相应的个数，如果要更换指定的位置先切片）

list=s.split(“,”)#返回一个数组，里面装的是字符串（如果你的刀贴着边切回就会产生空字符）

**day03-05(15)**

**查找**

b=s.satrtswith(“###”)#字符串是否以###开头

b=s.endwith(“###”)#字符串是否以###开头

tmp=s.count(“a”)#a出项了几次

index=s.find(“a”)#返回下标（以字符串出项的第一个位置为准，没有的话就返回-1）

**切片查找**

因为只返回自一个出现的位置

s.find(“a”,3)#从第3个位置开始找

s.index(“a”)#如果内容不存在直接报错（和find的区别）

**条件判断**

s.isalnum()#字符串是否是数字字母组成

s.isdigit()#字符串是否是数字组成，不包括小数点

s.isalpha()#字符串是否是字母组成

s.isnumeric()#即使是汉子数字也认识，数字加汉子组合也认识（和isdigit的区别）

len(s)和s.\_len\_()一样是求长度的（len实际执行的是s.\_len\_()）

**day04-01(16)**

**作业**

**day04-02(17)**

**列表的索引和切片**

和字符串一样

**列表修改与查询**

**增**

list和str是不一样的，lst可以发生改变，所以直接在原来的对象上进行操作

##### 直接在最后面添加

list.append(“aa”)

##### 在指定位置添加，原来的元素向后移动一位

list.insert(1,”刘德华”)

##### 迭代添加(把一个list添加到令一个list后面)

list.extend（lis1）

**删除**

##### 删除最后一个

list.pop()

##### 删除指定位置的元素

List.pop(index)

##### 删除指定元素

list.remove(“aa”) 删除不存在的元素会报错

##### 清空

list.clear()

##### 切片删除

del lst[1:3]

**修改(修改操作靠的是赋值)**

##### 指定元素修改

lst[1]=”aa”

##### 迭代修改

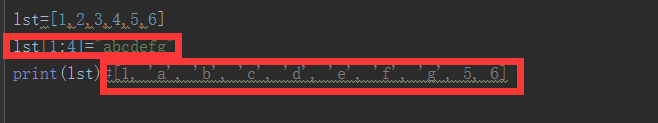
lst = ["太⽩白", "太⿊黑", "五⾊色", "银王", "⽇日天"]

lst[1:4] = "麻花藤" #['太⽩白', '马', '化', '腾', '⽇日天']

##### 切片修改

lst = ["太⽩白", "太⿊黑", "五⾊色", "银王", "⽇日天"]  
print(lst)  
lst[1:4:3] = ["麻花藤"] # 切⽚片修改也OK. 如果步⻓长不不是1, 要注意. 元素的个数#（切片修改，步长不是1需要注意修改个数和赋值个数）  
print(lst)#['太⽩白', '麻花藤', '五⾊色', '银王', '⽇日天']  
lst[1:4] = ["李李嘉诚个⻳龟⼉儿⼦子"] # 如果切⽚片没有步⻓长或者步⻓长是1. 则不不⽤用关⼼心个数  
print(lst)#['太⽩白', '李李嘉诚个⻳龟⼉儿⼦子', '⽇日天']

（注意步长为1和不为1的区别）



**查询**

for循环

**day04-03(18)**

len(list) #长度

c=list.count(“aa”)#返回aa出现的次数

list.sort(reverse=True)#默认是从小到大排序，reverse=True表示翻转

list.reverse()翻转

**列表的嵌套**

当成数组做

**day04-04(19)**

**元祖**

俗称不可变的列表，又被称为只读列表（就把列表的方括号换成圆括号）

元祖可以查询，可以循环，可以切片，但就是不能改

坑：tu=(108) 这是个int(当元祖的个数少于一个需要加，比如（tu=(108，)）空元祖这样定义tu=tuple())

**查询**

tu = (1, "太⽩白", "李李⽩白", "太⿊黑", "怎么⿊黑")

print(tu[0])

print(tu[2])

**切片**

print(tu[2:5]) # 切⽚片之后还是元组

**循环**

# for循环遍历元组

for el in tu:

print(el)

**修改的特殊情况**

# 尝试修改元组

# tu[1] = "⻢马⻁虎疼" # 报错 'tuple' object does not support item assignment

tu = (1, "哈哈", [], "呵呵")

# tu[2] = ["fdsaf"] # 这么改不不⾏行行

tu[2].append("麻花藤") # 可以改了了. 没报错

tu[2].append("王剑林林")

print(tu)

总结：这里元祖的不可变的意思是子元素不可变，而子元素内部的子元素是可以变得，这取决于子元素是否是可变对象

**range**

range(10)

range(1,10,2)

range(10,1,-2)#和切片一样

**day05-01(20)**

**作业**

**day05-02(21)**

**作业2**

type(e)==list#判断元素类型

**day05-03(22)(字典的增删该查)**

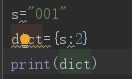
字典（dict）是python中唯一的一个映射类型，他是以{}括起来的键值对组成。

在dict中key是唯一的，在保存的时候根据key来计算一个内存地址。

可哈希意味着是不变的，可以改变的都是不可哈希的

{key1:value1,key2:value2}key可字符串可int

key可以是数字，bool，字符串，元祖(此处没有说a.b.c.d可以作为key)



key不可以是list,dict,set

**字典的相关操作**

**增加**

dict[“a”]=”a”#赋值操作 key可以重复，重复会替换原来的数据

ret=dict.setdefault(“马融”，“aa”)#添加操作，key不可以重复（注意两者区别）,会返回value(两个执行步骤 首先判断字典有没有key，如果没有执行新增 然后用这个key去字典查询，返回查到的结果（返回的是查询的结果，不是给的默认值aa）)

**删除**

##### 根据key删除

ret=dic.pop(“aa”) #返回key对应的value

##### 使用del

del dic[“joy”]

##### 随机删除

ret=dic.popitem() #返回的是元祖，第一个是key，第二个是value

##### 清空字典中的所有内容

dic.clear()

**修改**

dic[“key”]=value

**查询**

##### 索引

print(dic[“aa”])##aa不存在的话会报错

##### get(key)

dic.get(“aaa”,”没有则返回默认值”)##如果key不存在返回none

**day05-04(23)**

**拿到所有的的key**

dic.keys() #拿到所有的key，返回key的集合，像是列表，但不是列表 高仿的列表可以for循环

for key in dic.values():

print(value)

**拿到所有的的key**

dic.values()

**拿到所哟的键值对**

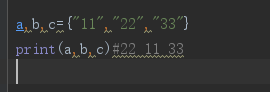
dic.items() #每一项是元祖 获得key item[0] ，value item[1]

**解构**

重点 a=1,2 a是元祖

a,b,c=(“w”,”e”,”r”)

for key,value in dic .items():



**字典的嵌套**

# 字典的嵌套

dic1 = {

"name": "汪峰",

"age": 18,

"wife": {

"name": '章⼦子怡',

"age": 28

},

"children": ['第⼀一个⽑毛孩⼦子', '第⼆二个⽑毛孩⼦子'],

"desc": '峰哥不不会告我吧. 没关系. 我想上头条的'

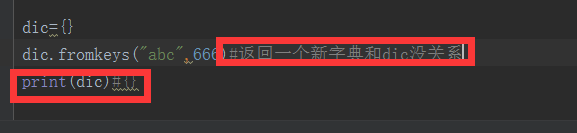
}

print(dic1.get("wife").get("name"))

print(dic1.get("children"))

print(dic1.get("children")[1])

**坑**



**day06-01(24)**

上周内容回顾

**day06-02（25）**

作业讲解

**day06-03（26）**

**得到变量地址**

id（）

**is和==的 区别**

is 用于判断两个变量引用对象是否为同一个， == 用于判断引用变量的值是否相等。类似于Java中的equal()和==。反之，is not 用于判断两个变量是否引用自不同的对象，而 != 用于判断引用变量的值是否不等

总结：id()和is的关系，is是比较id()计算出来的结果，最终我们通过is可以查看两个变量使用的是否是同一个对象（is不光要地址相同还要值相同，==只要值相同就行了）

**小数据池**

把我们使用过的值存储在小数据池中，供其他的变量使用。

小数据池阁数字和字符串使用，其他数据类型不存在

对于数字：-5-256是会被加到小数据池中的，每次使用都是同一个对象

对于字符串：1.如果是纯文字信息和下划线，那么这个对象会被添加到小数据池

2.如果是带有特殊文字符的，那么不会被添加到小数据池，每次都是新的

3.如果是单一字母\*n的情况，a\*20，在20个单位内是可以得。超过20个单位 就不会添加到校数据池中

注意：（一般情况下）:在py文件中，如果你只是单纯的定义一个字符串，那么一般情况下都是会被添加到小数据池中的。我们可以这样认为：在使用字符串的时候，python会帮我们把字符串进行缓存，在下次使用的时候直接指向这个字符串即可，可以节省很多内存。



**编码**

python2中默认使用的是ASCII码，所以不支持中文。如果需要在python2中更改编码需要

在文件的开始编写

# -\*- encoding:utf-8 -\*-

python3中：内存中使用的是unicode码



记住：英文编码之后的结果和源字符串一致，中文编码之后的结果根据编码的不同，编码结

果也不同，我们能看到，一个中文的UTF-8编码是3个字节，一个GBK的中文编码是2个字节。编码之后的类型就是bytes类型。在网络传输和存储的时候我们python是保存和存储的bytes类型，那么在对方接受的时候，也是接受的bytes类型的数据，我么可以使用decode()来进行解码操作，把bytes类型的数据还原回我们熟悉的字符串

**day07-01(27)**

**内容回顾和作业**

坑：字典迭代的过程中涉及到删除（修改 每次循环记录删除的key,循环完在删除）

**day07-02(28)**

**str**

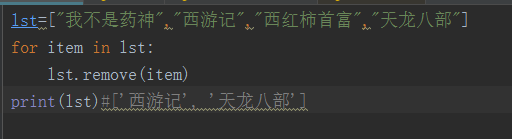
join可以把列表变成字符串，把字符串变成列表.split（）

tmp=str.join(str2)#把字符串str插入到str2中(依次在str2中的每个字符(除了最后一个字符)后面加入str)

s="abc"  
t="你好啊"  
tmp=s.join(t)  
print(tmp) #你abc好abc啊

tmp="\_".join(["alex","wuse","taibai","ritian"])  
print(tmp)#alex\_wuse\_taibai\_ritian

**day07-03(29)**



why？？？

lst中每个元素有一个下标1，2，3，4当我们便利到1并删除1时所有的元素自动向前移

怎么安全删除（记录删除的位置，循环结束在删除）

坑：dic={}  
s=dic.fromkeys("王健林","思聪")（原来不管什么字典都没有影响，产生新字典（深坑，考试））  
print(s)#{'王': '思聪', '健': '思聪', '林': '思聪'}

**set集合（无序，且不重复用的少，去重，不存value的字典，可哈希）**

注意 ：set集合中的元素必须是可hash的，但是set本身是不可hash的，set是可变的（set里面不能装set）

s={“王者”,”王者”,”王者”}

**set去重策略**

1. list=[]
2. 将list转成set()(s=set(lst))
3. 将set转成list(lst=list(s))

**增加**

##### 单个添加

s={“a”,”b”,”c”}

s.add(“c”)#重复的内容不会添加到set集合中去

##### 迭代添加

s = {"刘嘉玲", '关之琳', "王祖贤"}  
s.update("麻花藤") # 迭代更更新  
print(s)#{'王祖贤', '花', '麻', '藤', '刘嘉玲', '关之琳'}  
s.update(["张曼⽟玉", "李李若彤","李李若彤"])  
print(s)#{'王祖贤', '李李若彤', '花', '麻', '藤', '刘嘉玲', '关之琳', '张曼⽟玉'}

总结：迭代添加，遇到str，将str转成list，然后对list中的每个元素迭代添加

**删除**

##### 随机删除

item=s.pop()#随机删除一个

##### 指定删除

s.remove(“aa”)#删除指定，不存在则会报错

##### 清空

s.clear()#清空set集合，需要注意的是set集合是空的，打印出来是set()因为要和dict区分的

**修改**

set集合中的数据没有索引，也没有办法定位一个元素，所以没有办法进行修改。

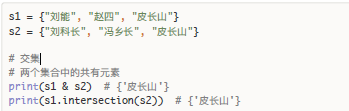
我们可以先删除指定元素在添加来完成修改操作

**查询**

for循环迭代查看

**常用操作**

##### 交集



##### 并集



##### 差集



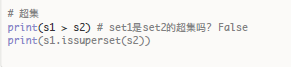
##### 反交集(两个集合中单独存在的数据)



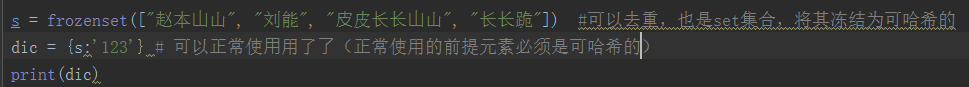
##### 子集



##### 超集



**冻结了得set操作，可哈希**

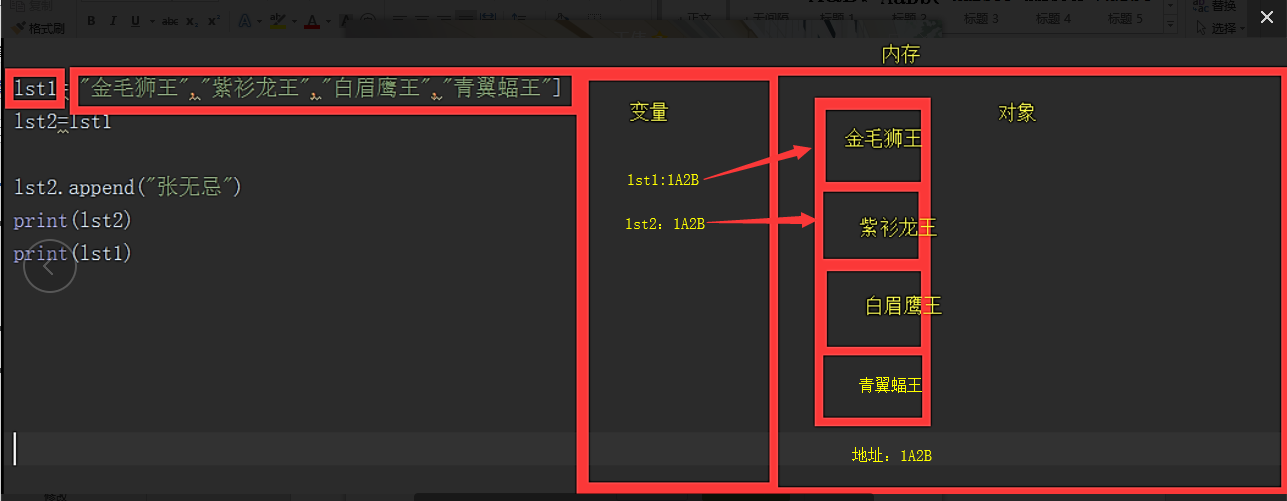


**day07-04(29)浅拷贝**

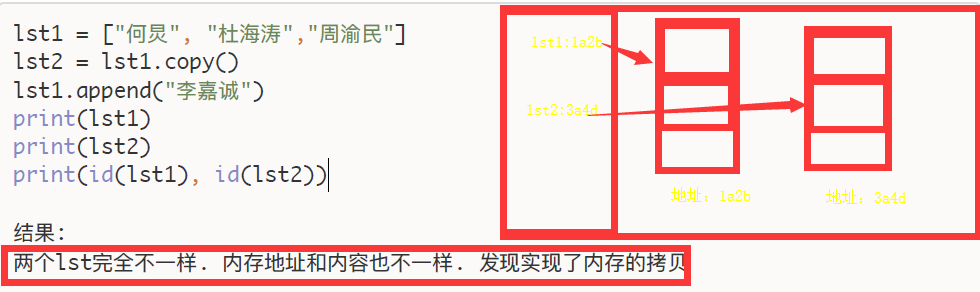


首先，程序执行第一句的时候产生变量lst1和对象，先生成对象，将对象赋值给lst1，实际上将对象的内存地址给lst1，我们程序中操作的是lst1，实际上操作的是对象

lst2=lst1(两个变量指向同一个对象，实际上是引用内存地址的赋值)

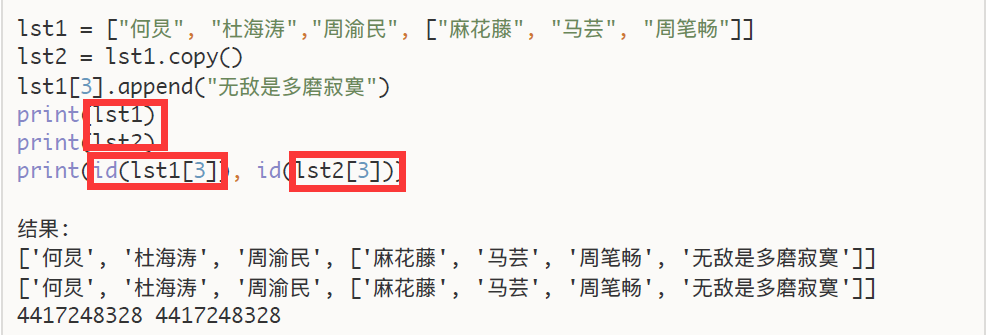


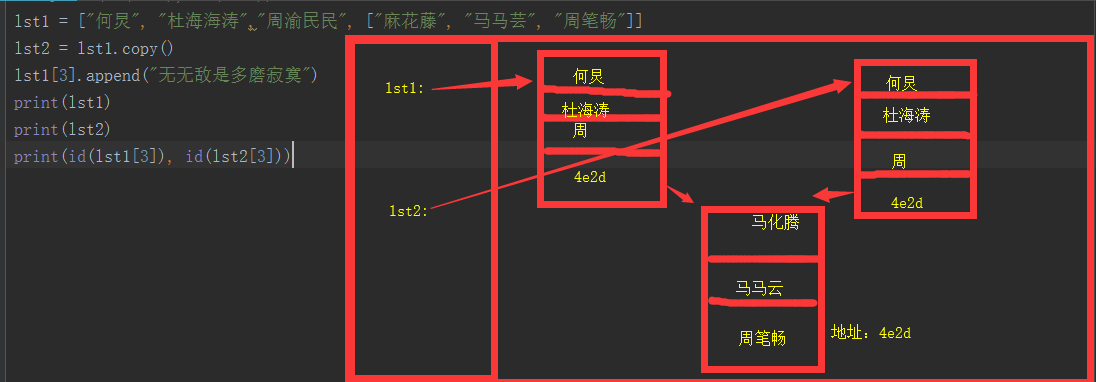
**浅拷贝copy(就变成两个，两个完全的个体，创建新对象)**



浅拷贝的第二种的写法:lst2=lst1[:](切片会产生新的对象)

**day07-05(30)(深拷贝)**





浅拷贝只是拷贝第一层，深拷贝需要import copy模块中的list2=copy.deepcopy(lst1) 进行深度拷贝

**day08-01(31)**

**作业**

**互换**

a,b=b,a

**随机数**

from random import randint

a=rangint(1,20)#产生1-20的随机谁

在python中的random.randint(a,b)用于生成一个指定范围内的整数。其中参数a是下限，参数b是上限，生成的随机数n: a <= n <= b。

**day08-02(32)(初始文件操作)**

从操作系统读文件是全部加载到内存中去还是用到哪里读哪里，当然是用到哪里读到哪里，所以我们通过句柄里实现

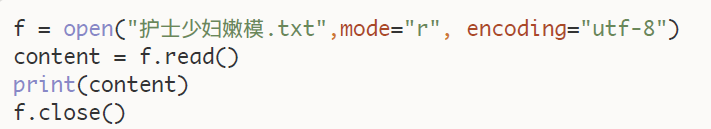
带b的处理是非文本，

文件句柄是一个迭代器

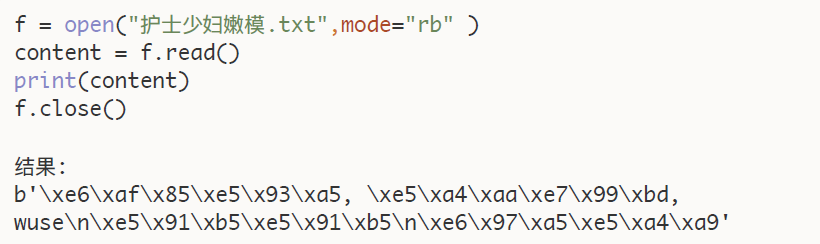
for line in f:##每次读取一行

print(line)

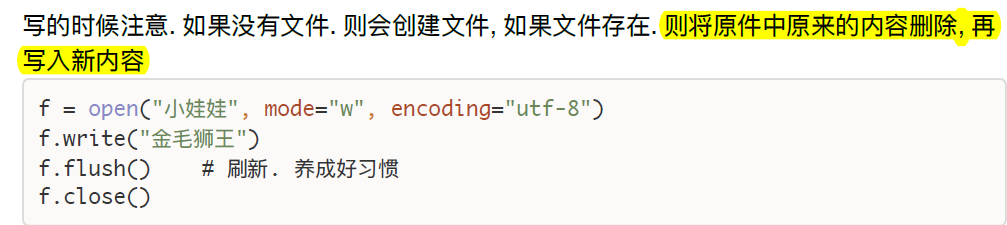
**只读操作(r, rb)**



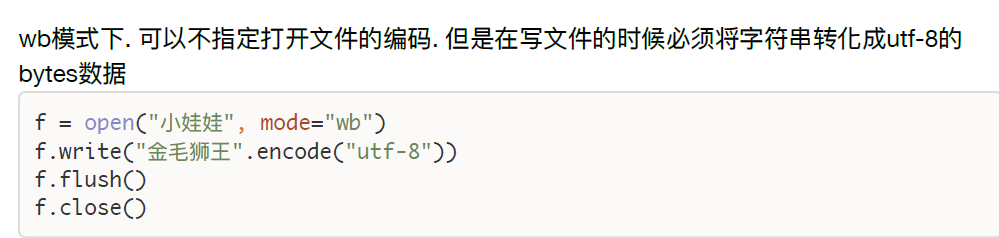
rb. 读取出来的数据是bytes类型, 在rb模式下. 不能选择encoding字符集.



**写模式(w, wb)**



w模式写的时候会清空所有的内容



**追加模式**



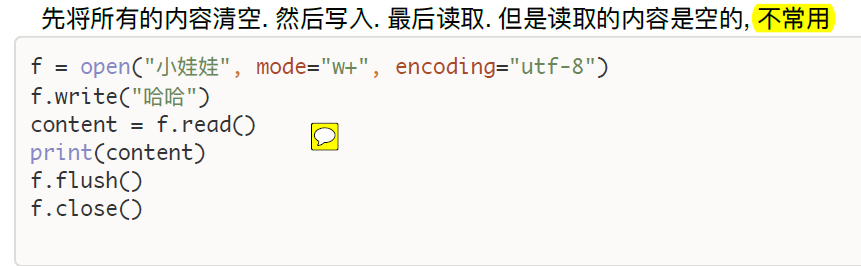
**读写模式(r+, r+b)**

+不是w 是在r上进行扩展在r的基础上写

对于读写模式. 必须是先读. 因为默认光标是在开头的. 准备读取的. 当读完了之后再进⾏写入. 我们以后使⽤用频率最⾼高的模式就是r+（r+模式必须先读后写才是安全的）(如果先写后读的话，会把内容写在第一行，读的时候从第二行开始)

深坑请注意: 在r+模式下. 如果读取了内容. 不论读取内容多少. 光标显⽰的是多少. 再写入或者操作⽂文件的时候都是在结尾进⾏行行的操作.

**写读(w+, w+b)**



如果想用w+先写后读有作用的话，就必须在读之前将光标移到开头（f.seek(0)）

**读取文件的方法**

1. read() 将⽂件中的内容全部读取出来. 弊端: 占内存. 如果⽂文件过⼤大.容易易导致内存崩溃
2. read(n) 读取n个字符. 需要注意的是. 如果再次读取. 那么会在当前位置继续去读⽽而不是从头读, 如果使⽤的是rb模式. 则读取出来的是n个字节
3. readline() 一次读取一行数据, 注意: readline()结尾, 注意每次读取出来的数据都会有⼀

个\n 所以呢. 需要我们使⽤用strip()⽅方法来去掉\n或者空格

4. readlines()将每⼀行形成⼀个元素, 放到一个列表中. 将所有的内容都读取出来. 所以也是 容易易出现内存崩溃的问题.不推荐使⽤用

**光标**

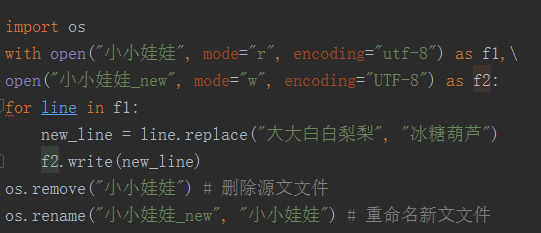
1. seek(n) 光标移动到n位置（距离开头的位置）, 注意, 移动的单位是byte. 所以如果是UTF-8的中⽂文部分要是3的倍数.通常我们使用seek都是移动到开头或者结尾.

移动到开头: seek(0) #用在反复读

移动到结尾: seek(0,2) seek的第二个参数表⽰示的是从哪个位置进⾏行行偏移, 默认是0, 表示开头, 1表⽰示当前位置, 2表⽰示结尾，第一个参数0表示偏移量

**文件修改**





**day09-01(33)**

**作业**

写文件的时候换行，在写的字符串里加/n

line.split() #不指定切割符号的话，会把所有的白色空格作为切割对象

**day09-02(34)(函数的定义)**

**day09-03(35)**

函数名和变量名命名的规则是一样的

每个函数如果在函数中不写return,默认返回None

我们亦可以只写一个return,也是返回None,停止函数的执行

return一个返回值，你在调用方能接受到一个返回值

return 多个返回值（接受方得到的数据是一个元祖），多个值需要用，隔开

**day09-04(36)(函数的参数)**

形参：函数声明的地方，声明出来的变量

实参：函数调用的地方，实际你给函数传递的值

**实参**

**位置参数**

在传参过程中. 系统会默认按照位置把实参赋值到形参



**关键字参数**

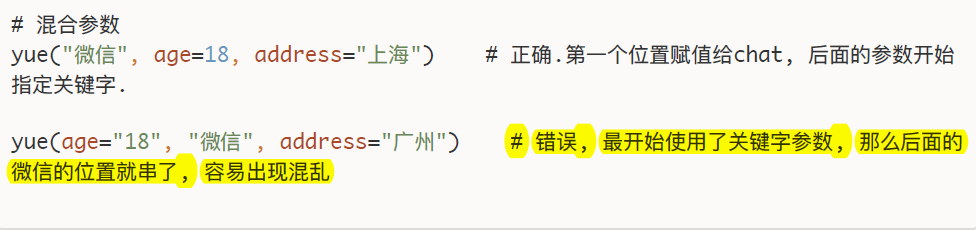
我们不需要记住每个参数的位置. 只要记住每个参数的名字就可以了



**混合参数**

可以把上⾯两种参数混合着使用. 也就是说在调用函数的时候即可以给出位置参数, 也可以指定关键字参数.

注意：在使用混合参数的时候，关键字参数必须在位置参数后面



**形参**

**位置参数**

位置参数：按照位置来赋值

**默认值参数**

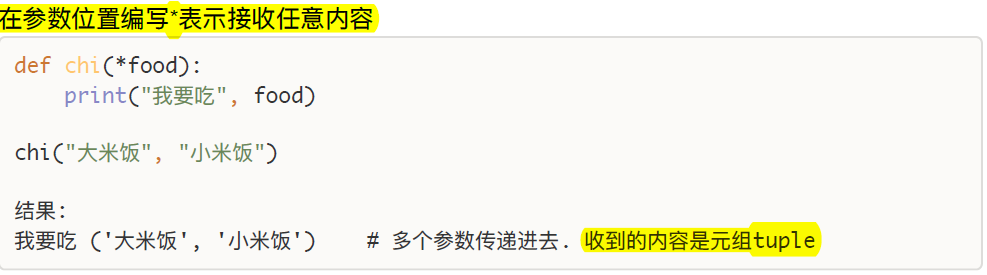
在函数声明的时候，就可以给出函数参数的默认值。在调用的时候可以给出具体的值，也可以不给值，使用默认值



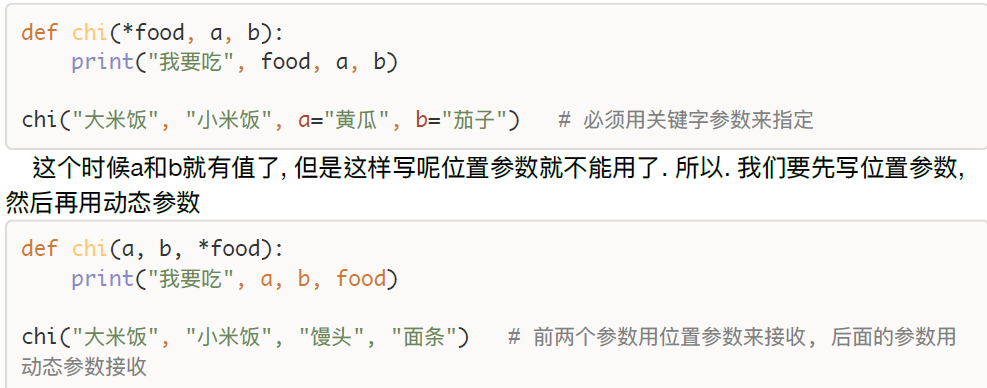
注意：必须先声明位置参数，才能声明默认值参数

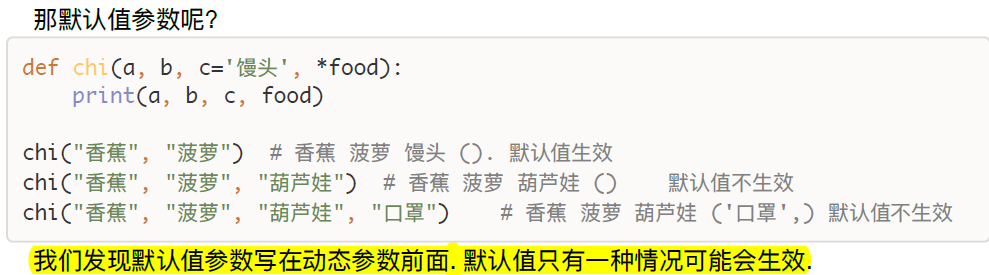
**动态参数**

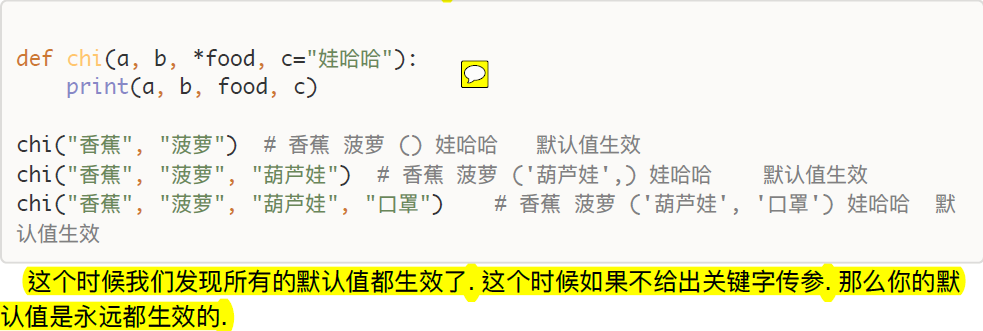
在参数位置编写\*表示接受任意内容



动态参数必须在位置的后面







顺序：位置参数，动态参数，默认值参数

**day09-05(37)(函数的实体应用)**

**day10-01(38)**

**作业**

a if a>b else b 如果a>b成立返回a否则返回b

**day10-02(39)(动态传参)**

**动态接受位置参数**

\* 接收的是元祖

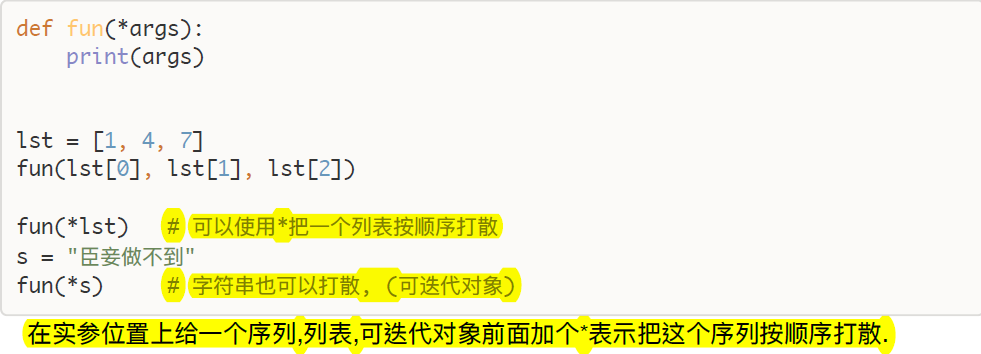
**动态接受关键字参数**

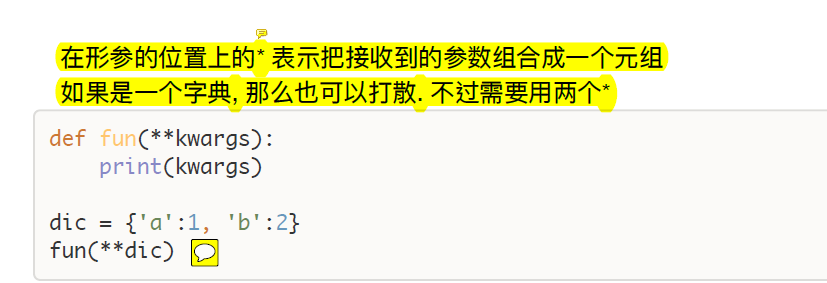
\*\* 接受的是字典

**总结：**



**动态参数的令一种传参方式**





**day10-03(40)命名空间**

在python解释器开始执⾏之后, 就会在内存中开辟一个空间, 每当遇到一个变量的时候, 就把变量名和值之间的关系记录下来, 但是当遇到函数定义的时候, 解释器只是把函数名读入内存, 表示这个函数存在了, ⾄于函数内部的变量和逻辑, 解释器是不关心的. 也就是说一开始的时候函数只是加载进来, 仅此而已, 只有当函数被调用和访问的时候, 解释器才会根据函数内部声明的变量来进行开辟变量的内部空间. 随着函数执⾏行完毕, 这些函数内部变量占⽤的空间也会随着函数执⾏完毕而被清空.

**全局命名空间**

我们直接在py⽂文件中, 函数外声明的变量都属于全局命名空间

**局部命名空间**

在函数中声明的变量会放在局部命名空间

**内置命名空间**

存放python解释器为我们提供的名字, list, tuple, str, int这些都是内置命名空间

**加载顺序**

首先内置命名空间，然后全局命名空间，最后局部命名空间

**取值顺序**

局部命名空间，全局命名空间，内置命名空间

**作用域命名空间:**

1. 全局作⽤用域: 全局命名空间 + 内置命名空间

2. 局部作⽤用域: 局部命名空间

我们可以通过globals()函数来查看全局作用域中的内容, 也可以通过locals()来查看局部作用域中的变量和函数信息（单独显示 他们是一样的 需要放在函数中才有区别）

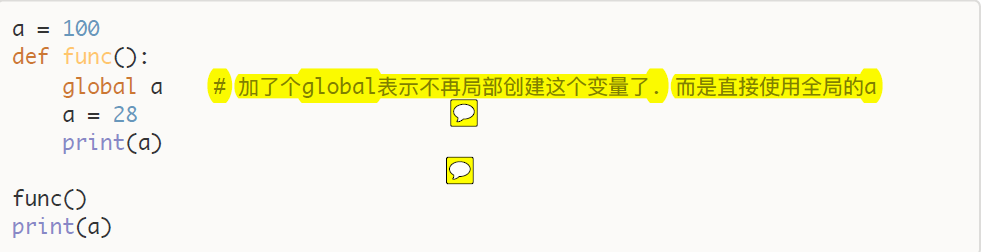
**函数的嵌套**

只要遇⻅了()就是函数的调用. 如果没有()就不是函数的调用



**关键字global和nonlocal**

**global**



加了global就不会在局部创建（不会局部开辟空间来创建）就是说只有一个a 把外面的a拿到里面来了（用的全局的a）,如果全局不存在a就会创建一个全局a

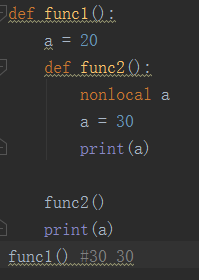
重点理解：

global具有提升变量的范围

如果a事先没有创建，这个程序照样可以执行

**nonlocal(用本地外面的 )**

nonlocal 表⽰在局部作用域中, 调用父级命名空间中的变量.



a = 1  
def fun\_1():  
 a = 2  
 def fun\_2():  
 nonlocal a  
 a = 3  
 def fun\_3():  
 a = 4  
 print(a)  
 print(a)  
 fun\_3()  
 print(a)  
 print(a)  
 fun\_2()  
 print(a)  
  
print(a)  
fun\_1()  
print(a)# 1 2 3 4 3 3 1

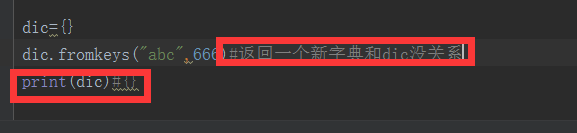
**day11-考试（41）**

print("a"+"b"+1) 报错

print(int("11a")) 报错

列表最后一个元素跟个逗号不影响执行print([1,2,])

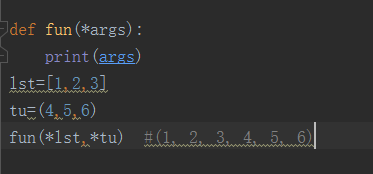
l1=[22,33,44] l2=l1[:] l1与l2的关系是浅拷贝 对的



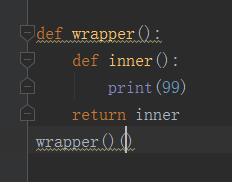
将列表转成元祖（tuple(ll)）

**day12-01(42)**

**作业**



特殊题目

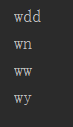
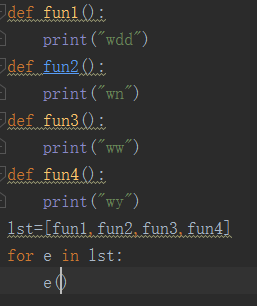


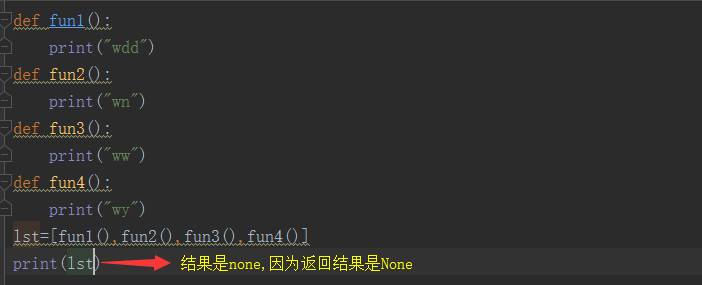
**坑**



**day12-02(43)**

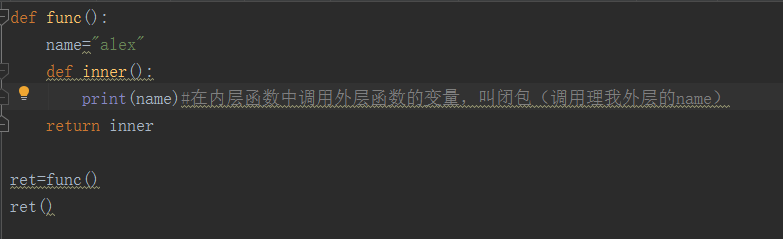
函数名是内存地址，函数名后面加个括号就表示执行





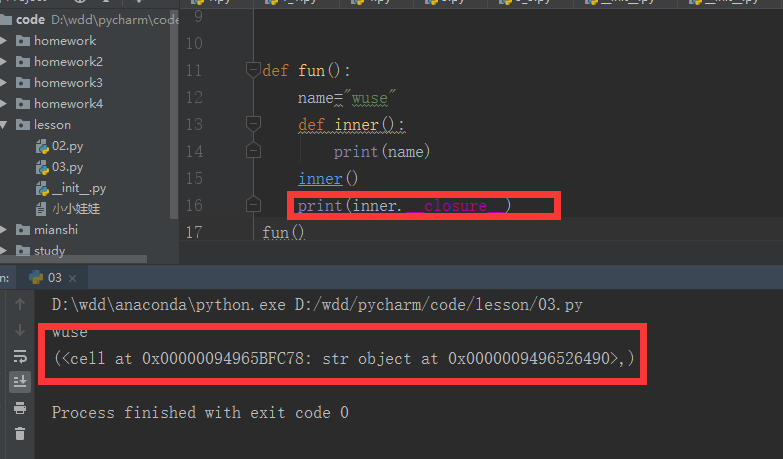
**day12-03(44)(闭包)**

闭包就是内层函数对外层函数（非全局）的变量的引用，叫闭包

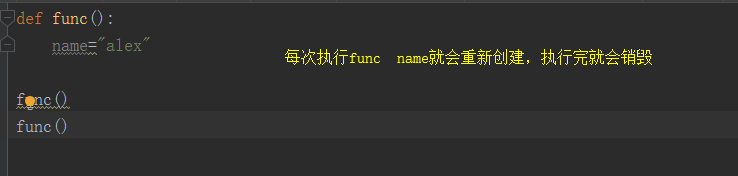


**闭包的检验**

我们可以使⽤\_\_closure\_\_来检测函数是否是闭包. 使⽤函数名.\_\_closure\_\_返回cell就是闭包. 返回None就不是闭包



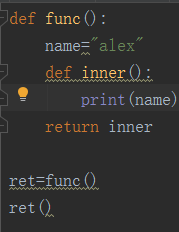
**闭包的作用**



**作用一**

闭包，可以让一个局部变量常驻内存

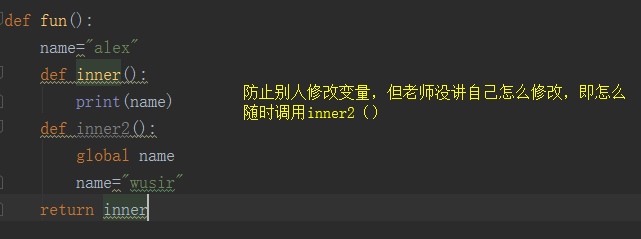
python的机制就是碰到这样的代码（两个不同空间用到同一个变量，就将这个变量常驻内存）就会将局部变量常驻内存



**作用二**

全局写的变量 通过global是可以修改的，这是不安全的（变量写在全局是不安全的）

变量在一个方法里面，别人改不了



**闭包的好处**



爬虫，我们从一个网址上爬图片，第一次需要耗费时间，我们将爬到的内容常驻内存，下次再爬就秒查

**迭代器**

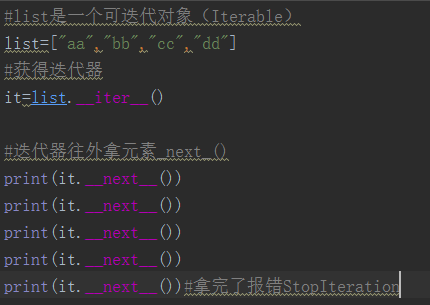
一个班级是迭代对象，班长是迭代器，班长可以一个一个的把人叫出来

可迭代对象：str,list,tuple,set,f,dict(可迭代对象都有一个函数\_\_iter\_\_())

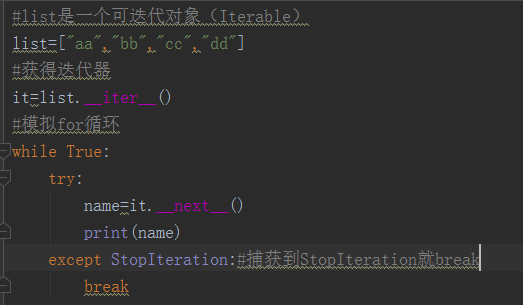
dir(对象)来查看一个对象或数据类型中包含了那些东西print（dir(对象)）

判断某个对象是否有\_\_iter\_\_ print(“\_\_iter\_\_”in dir(s))

迭代器将可迭代对象的内容一个一个输出出来



**模拟for循环迭代器**

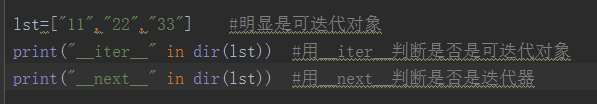
“  
注意： 迭代对象和迭代器是两码子事，迭代器比可迭代对象多了一个\_next\_()

**迭代器和可迭代对象的判断**

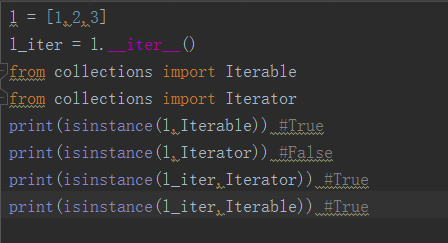
**方式一**

迭代对象的判断 “\_iter\_”in dir(对象) (判断\_iter\_在dir(对象中))

迭代器的判断 “\_next\_”in dir(对象) （判断\_next\_在dir(对象中)）



**方式二**



迭代器不仅有\_\_iter\_\_方法也有\_\_next\_\_方法（判断是迭代器必须同时又\_\_iter\_\_和\_\_next\_\_）

迭代器一定是可迭代的,迭代器在去拿他的迭代器是它本身，有\_\_next\_\_不一定是迭代器但没有\_\_next\_\_一定不是迭代器

**迭代器的优点**

1. 节省内存
2. 惰性机制
3. 不能反复只能向下执行

**day13-01**

**内容回顾**

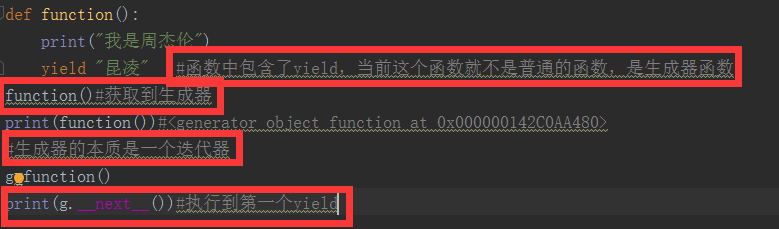
**day13-02(生成器函数)**

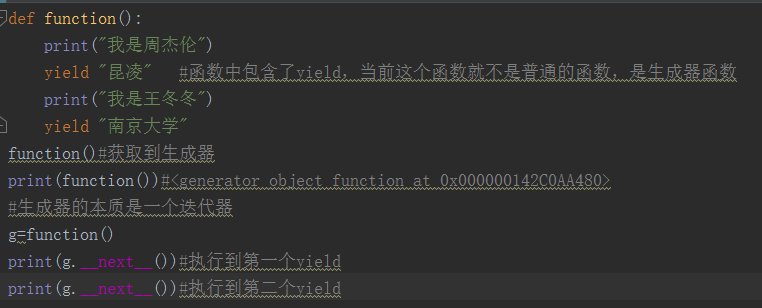
**生成器和生成器函数**

生成器的本质是迭代器

**生成器的三种生成方法**

##### 通过生成器函数





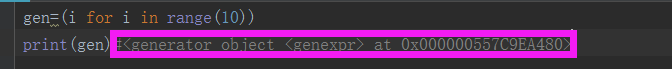
return和yeild的区别：

return 直接返回结果，结束函数的调用

yield 返回结果，可以让函数分段执行（最后一个yield之后如果在进行\_\_next\_\_会报错）

##### 通过生成器表达式创建生成器

（结果 for 变量 in 可迭代对象 if 筛选）



##### 通过数据转换

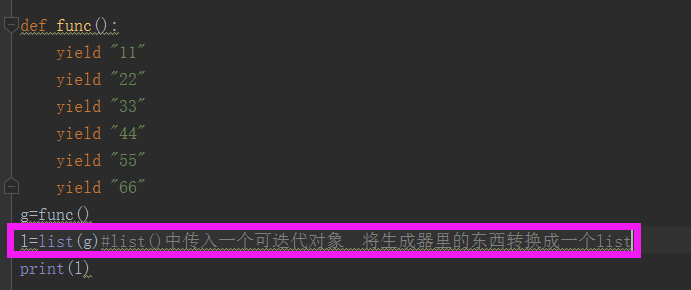
**day13-03(46)(send和推倒式)**



\_\_next\_\_()可以让生成器向下执行一次

send()也可以让生成器向下执行一次，给上一个yield传一个值，第一个不用send()，最后一个也不用send（）

**生成器里的东西转换成列表**



**推倒式（简化输入）**

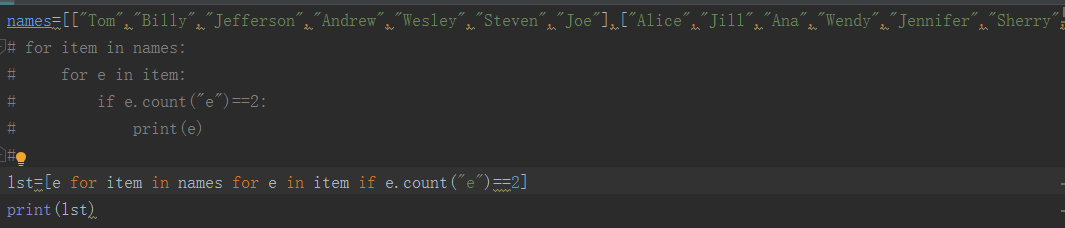
那种类型的推倒式最终给你的就是那种类型

##### 列表推倒式

语法【最总结果变量 for 变量in 可迭代对象 if 条件】

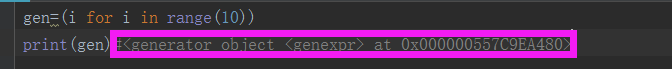


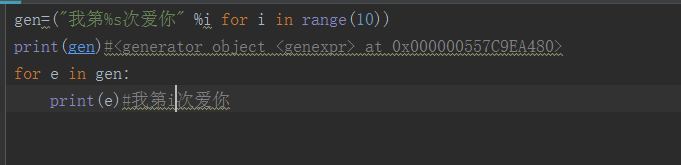
##### 多层for循环如何转成推导式（直接将for循环连起来写）



**day13-04(47)(生成式表达式)**

生成器表达式只要把列表推到式的【】改成（）





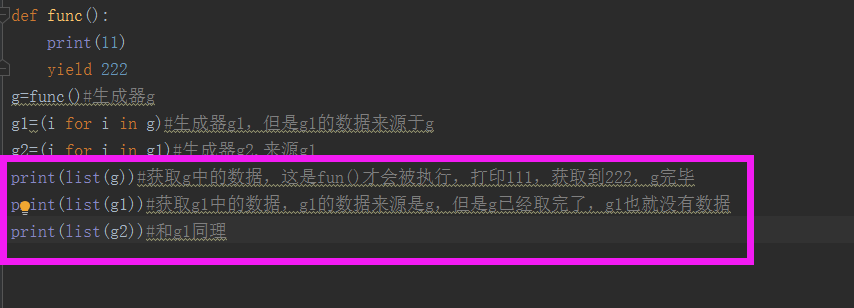
生成器表达式和列表推导式的区别：

1. 列表推倒式比较耗内存，生成器机会不占用内存，使用的时候才分配和使用
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

举个列子：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

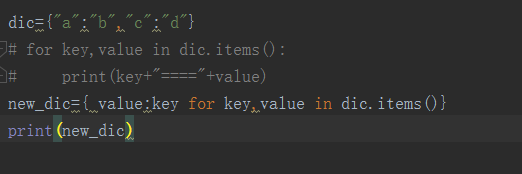
生成器是惰性机制：生成器只有在访问的时候才取值，说白了，你找他要他才给你值，不找他就不会给你值

**深坑**



**day13-05(48)(面试题讲解)**

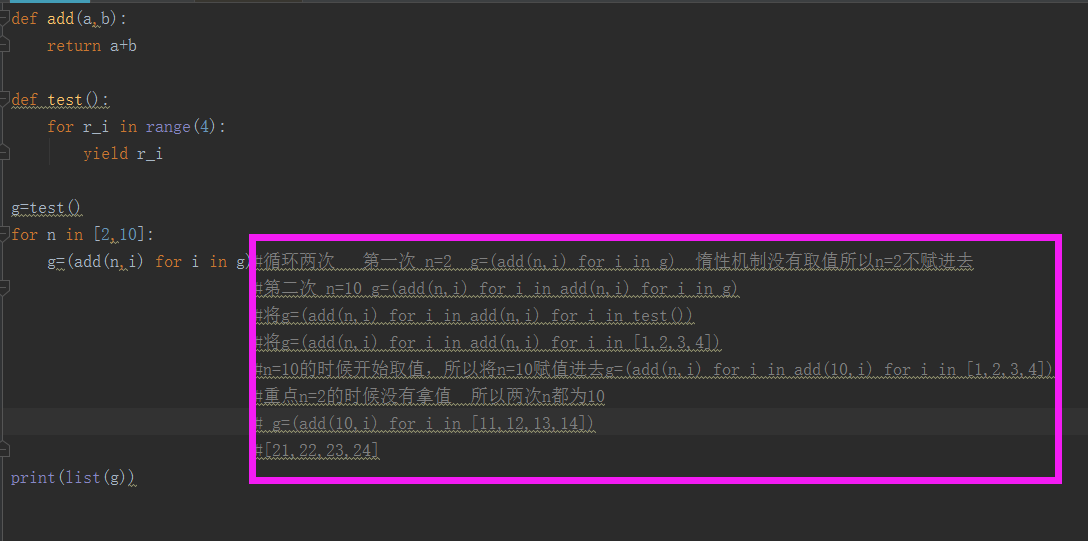
**字典推倒式**



集合推导式：集合推导式自带去重功能

总结：推导式有列表推倒式，字典推导式，集合推导式，没有元祖推导式（元祖不可变，元祖是小括号，小括号是生成器）

**深坑**

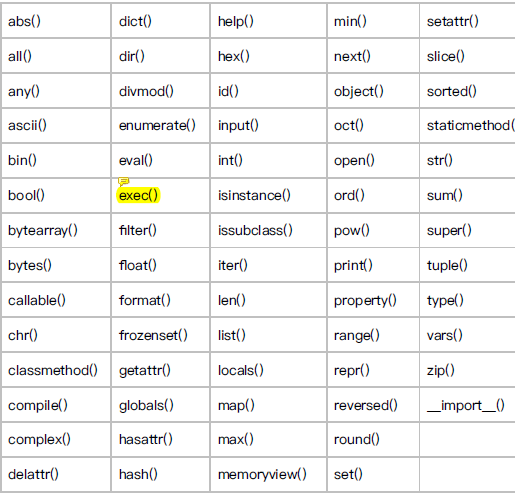


**day14-01(49)**

**内容回顾**

**day14-02,03(50)**

**内置函数**



##### 作用域相关

Globals()

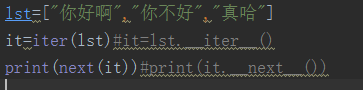
Locals()

##### 迭代器和生成器

Range

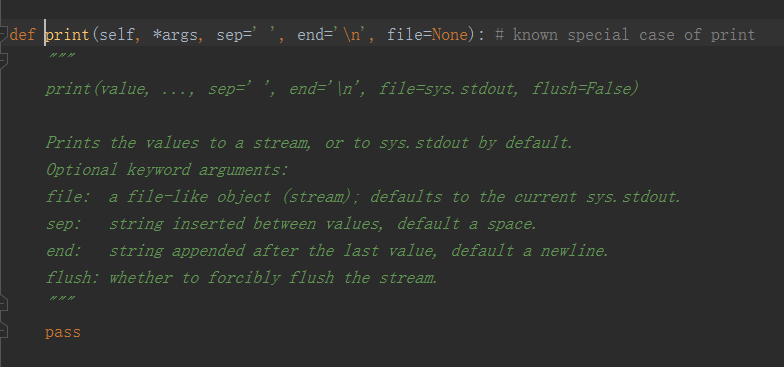
iter=iter(可迭代对象)

tmp=next(迭代器)



##### 其它

###### 输入输出



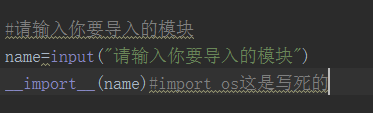
###### 内存相关

hash 算出存储的位置 dict(dict是字典)查找效率非常高，hash表，用空间换的时间，比较耗费内存，数字的hash值就是它本身hash(12)=12,列表不可hash因为列表是可变的，元祖是可hash的

id

###### 文件相关的

###### 模块相关的



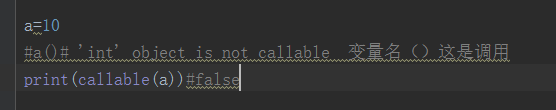
###### 帮助

help(对象)查看对象的全部内容 倾向于用ctrl点进去help(print)

dir()查看内置属性 dir([])查看列表有哪些方法

###### 调用相关

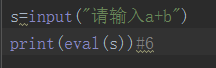
callable（给的参数是否可以被调用）（调用就是 名（））



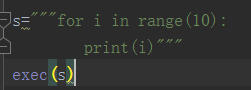
函数名可以被调用

###### 字符串

eval可以动态的执行代码，这个代码必须要有返回值，只能执行简单的



exec（s）执行代码不反回任何内容，支持复杂的（用户向里面敲代码）

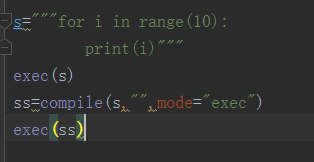




compile将一个字符串的代码编译成字节码







##### 基础数据类型

bin()

hex()

oct()

###### 数学运算

abs()

divmod(a,b)返回元祖（求商和余数）

round()5舍六入

pow(x,y)

sum()

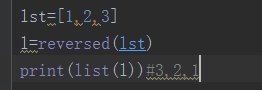
min()

max()

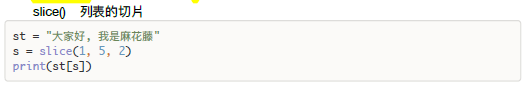
list()

tuple()

reversed()将一个序列翻转，返回翻转序列的迭代器（为啥返回的是迭代器，迭代器的通用性，可以方便转换），不会改变原来的



slice(start,end,step)



##### 字符串

###### str

###### format



###### bytes

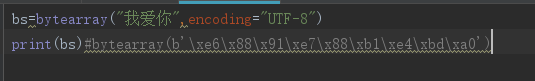
将字符串转成字节码



###### bytearry

深层次的代码中可能用到

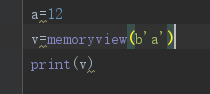
正常一个字符用UTF-8编码是3个字符

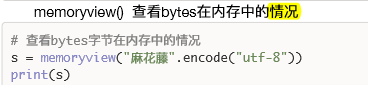


因为一个字符是3个，所以改变其中一个你字就不认识了

###### memoryview

内存效果显示(TypeError: memoryview: a bytes-like object is required, not 'int')





###### ord

返回字符在码表中的位置

根据字求码位



###### chr

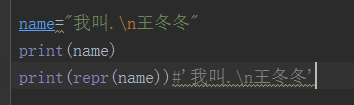
根据码位求字符



###### ascii

###### repr

原样输出,过滤掉转义字但是不过滤%号的转义字符（即字符串的占位效果依然有效）



##### 数据集合

###### dict()

###### set()

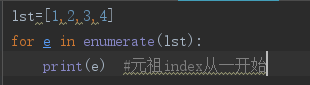
**frozenset()**创建一个冻结的集合，冻结的集合不能进行添加和删除操作

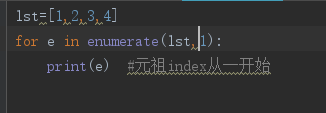
**len()**

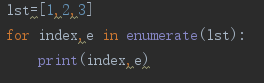
###### sorted()

###### enumerate()获取集合的枚举对象

enumerate(list，start)start表示下表从多少开始（从多少开始是编号，仍然会获取全部）

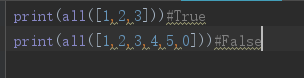






###### all()

可迭代对象中全部是True,结果才是True



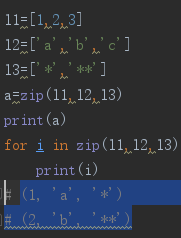
###### any()

可迭代对象中有一个是True,结果就是True



###### zip()用在数据汇总

函数用于将可迭代的对象作为参数，将对象中的对应的元素打包成一个个元组，如果各个对象的可迭代元素个数不一致的话，则返回列表长度为最短对象的长度（多余部分不处理）



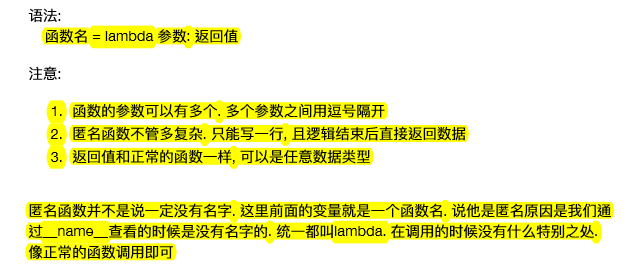
###### filter()

###### map()

**day15-01,02,03,04,05,06(51)**

### 昨日内容回顾lambda

#### 匿名函数

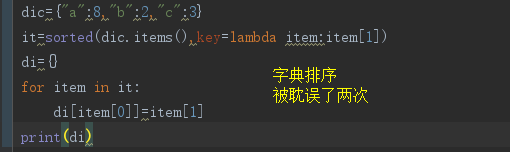




查看最终的函数名：a.\_\_name\_\_

#### Sorted函数

list.sore()是list中的函数而Sorted是内置函数





#### Filter()

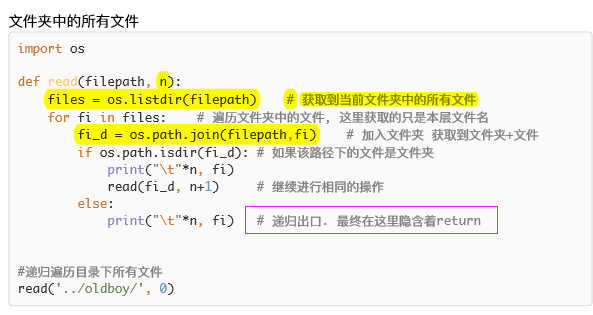


#### map()





#### 递归



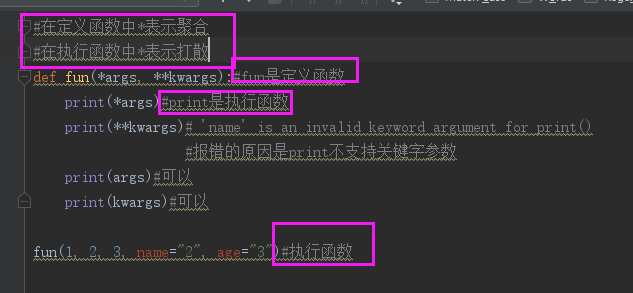
#### 二分查找

# 面向对象

## day16-01(1)

### 内容大纲

## day16-02(2)



### 空间

名称空间：当程序执行时，将变量与值得对应关系存放在一个空间中

临时名称空间：当函数执行时，将函数中的变量以及值得对应关系放到一个空间中，这个空间会随着函数的结束而消失

内置命名空间：python自带的内置函数

### global:

1.在函数中声明一个变量

2.在函数中修改一个全局变量

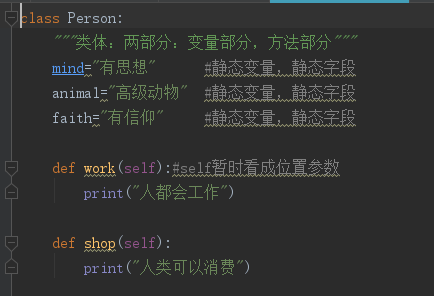
### nonlocal:

在函数中对父级的变量进行修改，但是不能修改全局变量

## day16-03(03)(面向对象初识)

### 类

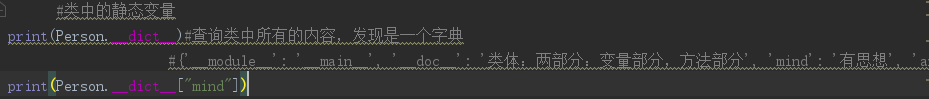
### 类名角度



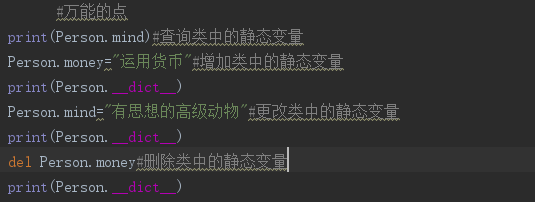
#### 类中的静态变量调用

##### 类名.\_\_dict\_\_（只能做查询不能进行增删改查）

类中的所有内容都以字典的形式存储



##### 万能的点（可以增改查删）

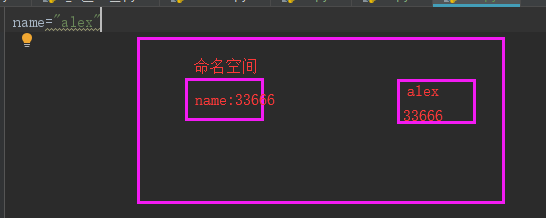


#### 操作类中的方法

##### 类名.function(self)self要传一个地址

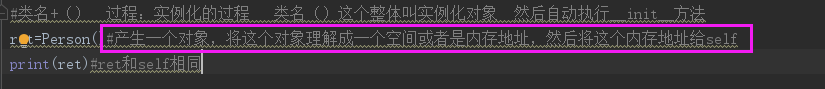
这个说明类方法的执行不需要对象的存在只要地址存在就行（这种方法基本不用）

### 对象的角度

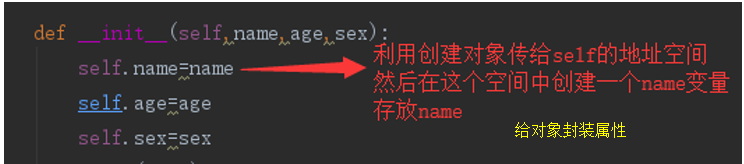


根据变量名找内存地址---》然后根据内存地址找对应的值

#### 理解类方法中的self



产生一个对象，把地址传给类中方法的第一个参数，我们约定成self



创建一个实例化对象分三步走：

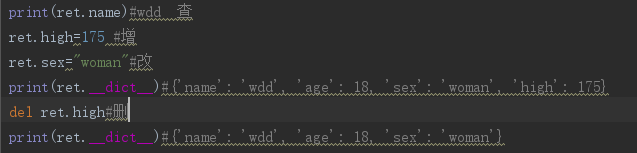
1. 类名+（）产生一个实例（对象，对象空间）
2. 自动执行类中的\_\_init\_\_方法，将对象空间传给\_\_init\_\_的self参数
3. 给对象封装对应的属性

#### 操作对象中的静态变量和对象操作类中的静态变量和方法

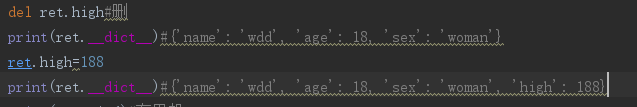
##### \_\_dict\_\_查询对象中的所有内容



##### 万能的.



##### 直接用万能的点增加属性



##### 对象操作类中的静态变量（只能查）



为啥不能增删改：因为增的会在自己实例化的空间中增，改的话也是变相的在自己实例化的空间中增加，删除的话，自己空间中没有，删除报错

##### 对象操作类中的方法

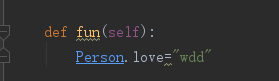


类中方的第一个参数不用传，对象调用的话自动传进去

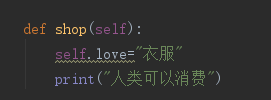
## day17-01(04)

0.5看技术 0.5谈吐，交朋友

### 扩展（在类里面增加静态变量）



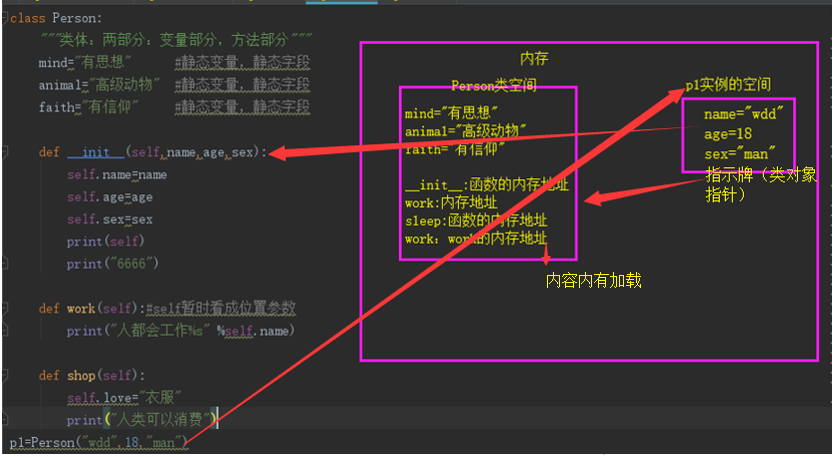
扩展（在类的方法里给对象封装属性）



## day17-02(05)

## day17-03(06)

问题既然可以在类外面封装属性，为啥在\_\_init\_\_方法中进行封装（\_\_init\_\_自动封装，类外面的要调用）



当程序运行时，读到class Person立马成内存中创建一个Person空间（但这个类不用执行）

继续将类静态变量加载到类空间，将方法的地址加载进来，读到创建对象这个代码时，首先碰到Person()在内存中实例一个空间（这个实例空间中只有类的指示牌），然后自动执行\_\_init\_\_将属性加到实例的空间中

注意（类中的静态变量加载到内存了，方法没有，方法只加载了地址）

先从对象空间找然后再找类空间的

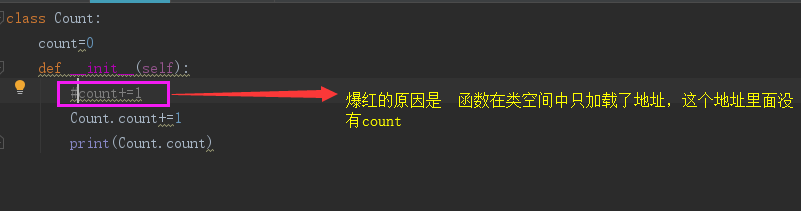
### 查询顺序

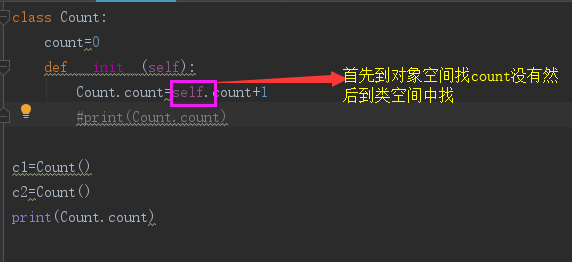
对象.属性:先从对象空间找，如果找不到，再到类空间找，再找不到，再从父类找

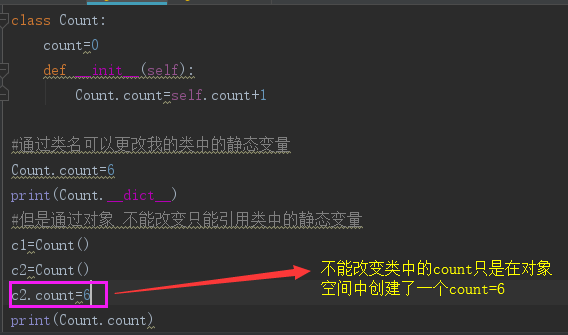
类名.属性:先从本类空间找，如果找不到，再从父类找

## day17-04(07)

### 计算一个类实例化多少对象







## day17-05（08）组合

组合：给一个类的对象封装一个属性，这个属性是令一个类的对象

## day18-01(09)

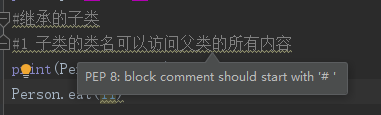
## day18-02（10）继承

面向对象的三大特性：封装，继承，多态

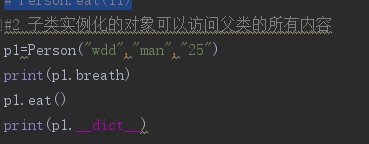
用法

### 访问

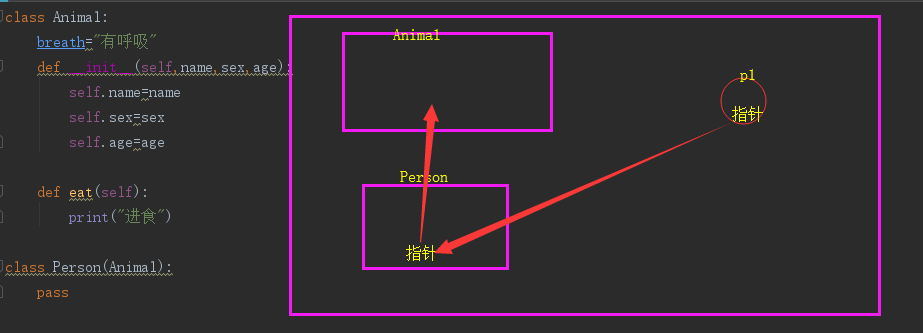
#### 继承父类的子类可以用子类名访问父类的静态变量和方法



#### 子类实例化的对象也可以访问父类的所有方法



#### 内存访问顺序



## day18-03(11)(继承关系中执行方法)

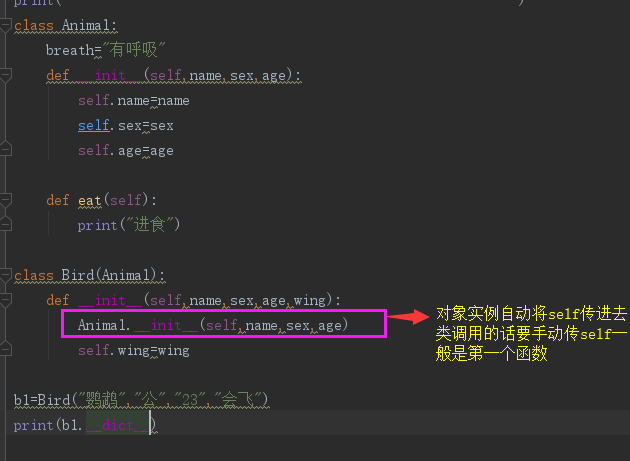
子类中有方法就执行子类的，没有就执行父类的

只执行父类的方法，子类中不要定义与父类同名的方法

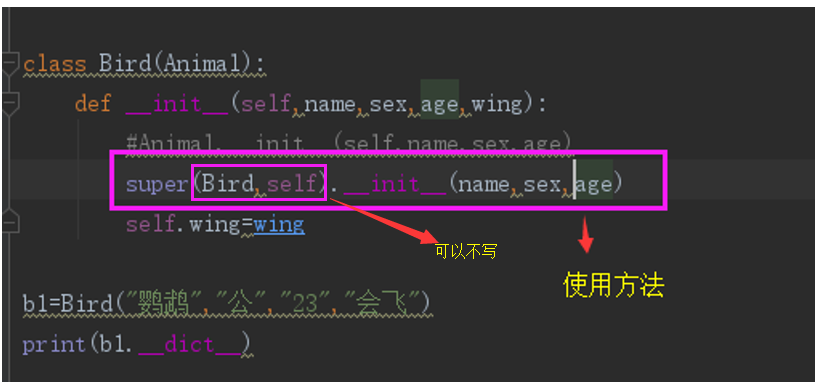
只执行子类的方法，在子类创建这个方法

### 即执行子类的方法，又要执行父类的方法

#### 方案一：调用父类的构造方法



#### 方案二：用super()推荐



## day18-04(12)(继承的顺序，深度优先和广度优先)

### 继承（单继承，多继承）

新式类：凡是继承object类都是新式类（python3中所有的类都是新式类，默认继承object）

经典类：不继承object类都是经典类（python2中既有新式类又有经典类）

单继承：新式类和经典类执行的顺序一样

多继承：新式类和经典类执行的顺序不一样

### 区别

### 新式类：遵循广度优先(python3)(最多继承两个类，3个类就出错)

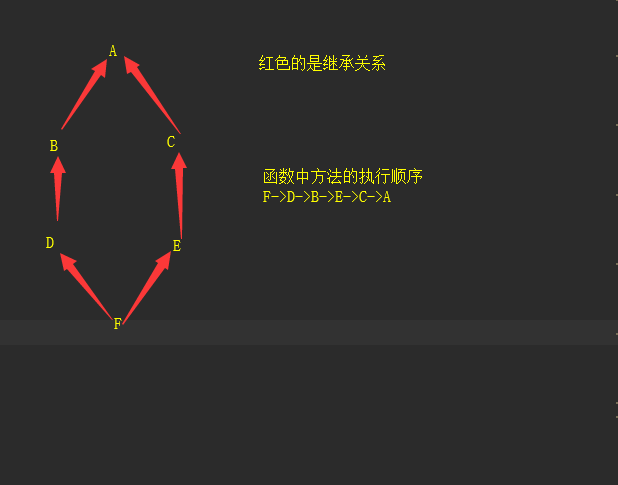
## day19-01(13)

继承的好处：

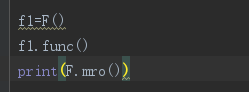
优化代码，节省代码

提高代码的复用性

让类与类之间发生关系



广度优先：沿着一条路走，到最后一个结点时判断是不是还有其他没有走的路径到达最后一个结点，如果有其他路径的话，就换成另一个路径走。

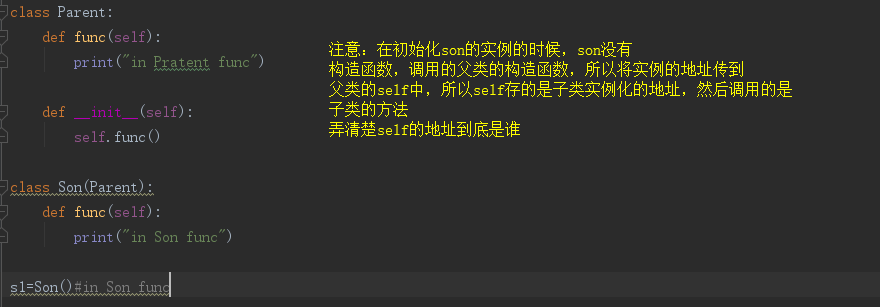
是用这个方法来查看类的中方法的执行顺序

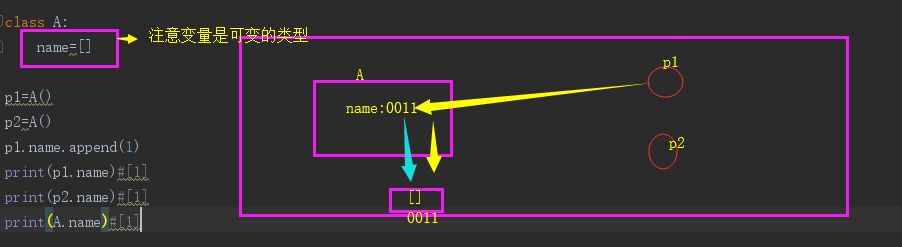
### 经典类：遵循深度优先（python2）（最多继承两个类，3个类就出错）

深度优先是一条路走到黑

## day19-02(14)

### 坑





## day19-03(15)

### 抽象类，接口类

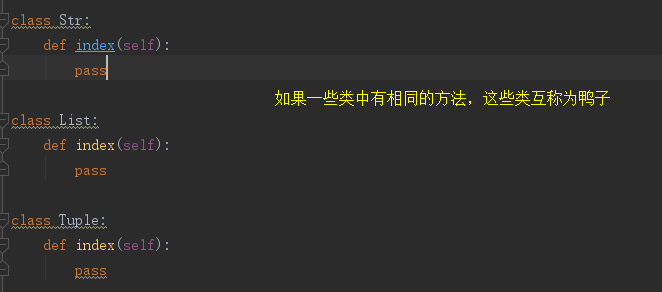
抽象类和接口类就是定义一个规则：



## day19-04(16)面向对象三大特性之封装

python中处处是多态

python中没有多态，他有什么？他有鸭子类型



### 封装

广义的封装：实例化一个对象，给对象空间封装一些属性

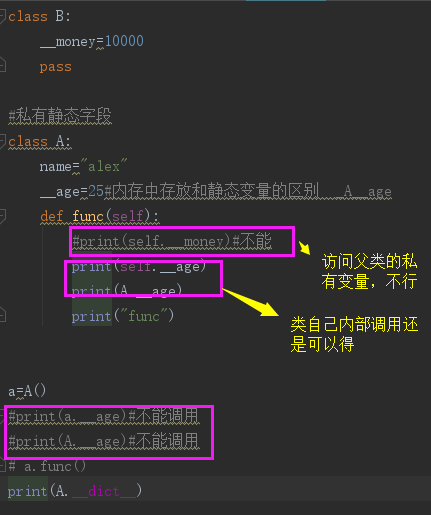
狭义的封装:私有化

私有成员:私有静态字段，私有方法，私有对象属性

### 私有静态字段（对于私有静态字段来说，只能在本类中内部访问，类的外部，派生类均不可访问）

实例化对象不能访问私有静态字段

类名不能访问私有静态字段



其实在编译器的构造中是可以使用的（不推荐）

编译器在碰到私有静态变量时自动在私有变量前加个前缀



### 私有方法（对于私有方法，类外和类的派生类也不能访问）

### 坑（静态的加载到内存中都要在变量名前面加前缀）



所以，执行的是父类中的\_\_fun

## day20-01(17)

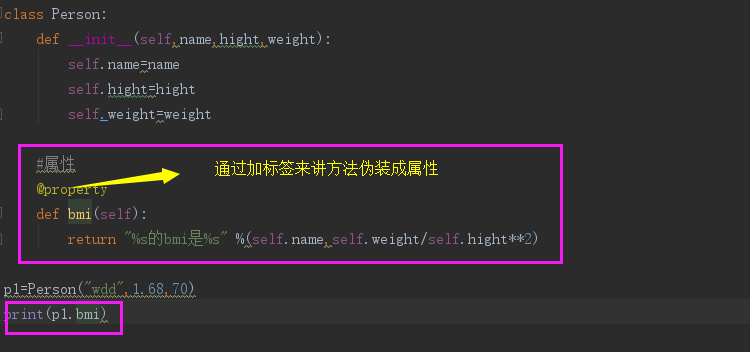
## day20-02(18)属性（目的私有静态变量和属性配套使用，让私有的静态变量在外面也能被获取,修改，删除，增加也是调用age（\_\_age不存在就增加））

什么是特性property

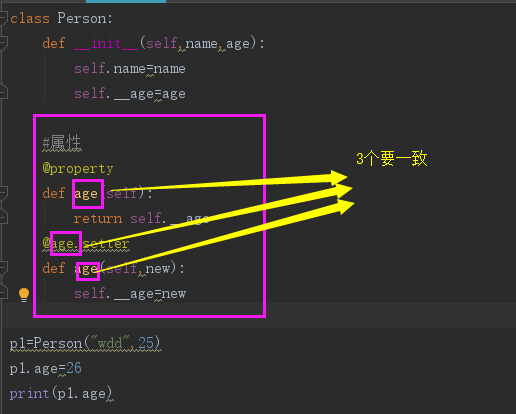
property是一种特殊的属性，访问它时会执行一段功能（函数）然后返回值

将一个类的函数定义成特性后，对象再去使用的时候obj.name，根本无法察觉自己的name是执行了一个函数然后计算出来的，这种特性的使用方式遵循了统一访问原则

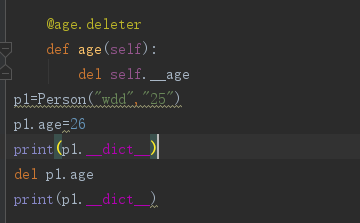
### 通过添加@property将一个方法伪装成一个属性



### [通过@属性.setter](mailto:通过@属性.setter)标签进行属性的更改



### 通过age.deleter删除私有静态变量

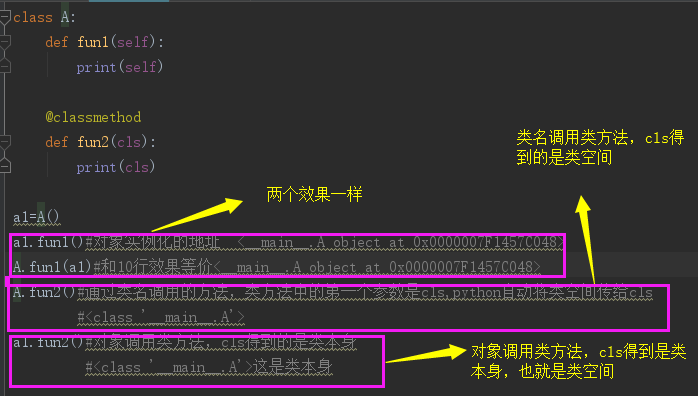


### 总结：

### Property:看着像名词但是需要计算的，比如bmi,周长，面积

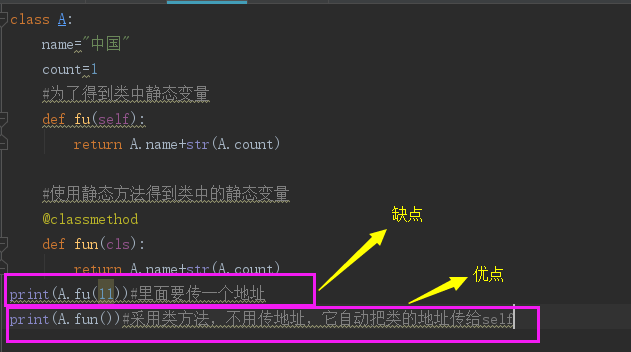
## day20-03(19)(类方法，静态方法)

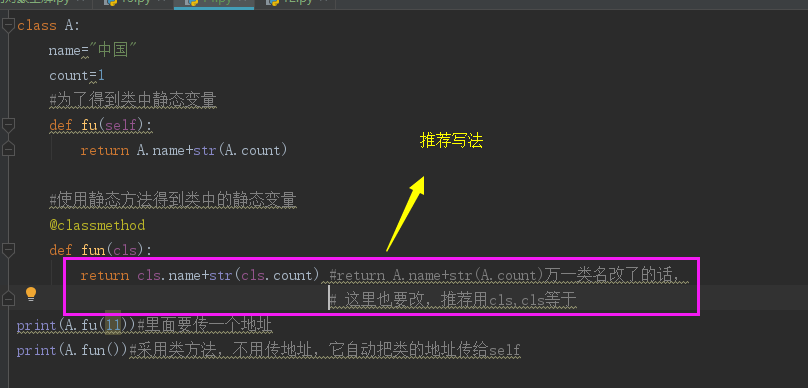
### 类方法（通过类名调用的方法，类方法中的第一个参数是cls,python自动将类空间传给cls）



#### 类方法的应用场景

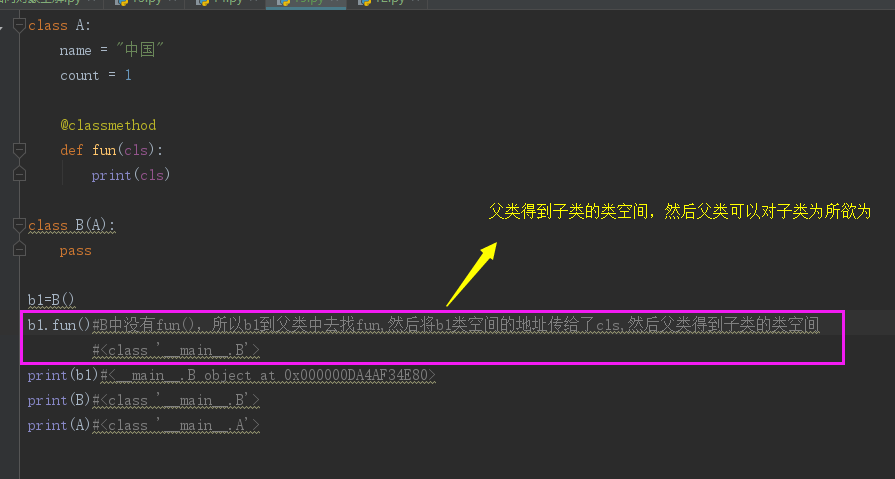
##### 场景一：类中有些方法是不需要传入对象，不要对象的一切东西



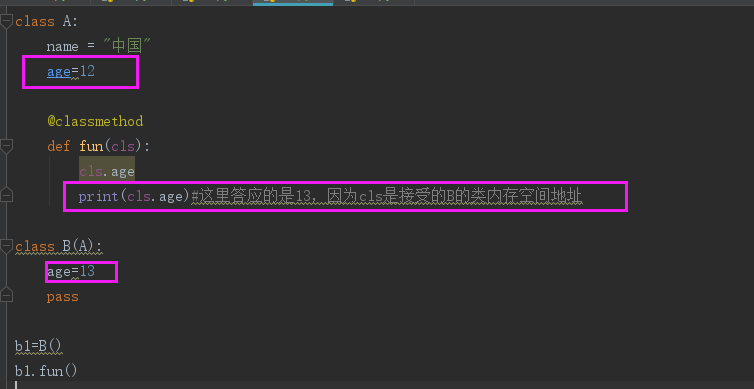


##### 场景二：对象对类中静态变量的更改（不调用类方法，对象是无法修改类中的静态变量的）

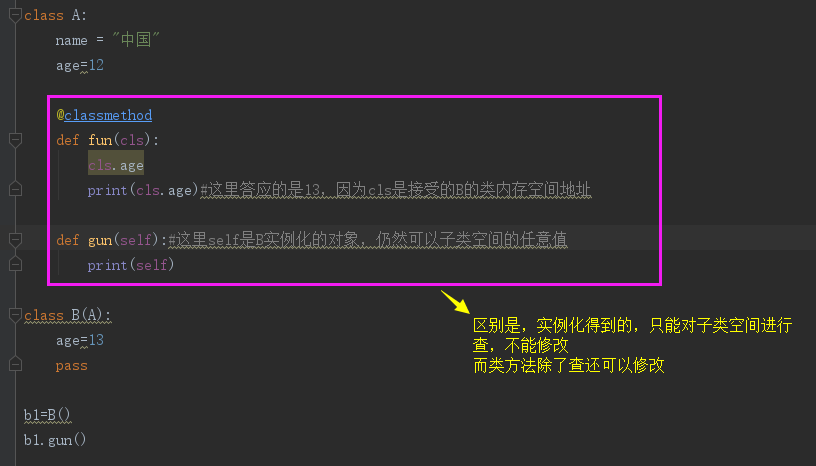
##### 场景三：继承中，父类得到子类的类空间



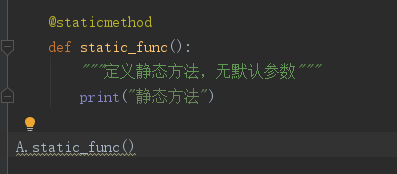
##### 面试题



##### 不通过类方法，也能让我的父类的某个方法得到子类空间里面的任意值（两者区别是）



### 静态方法（通过加标签@staticmethod，定义一个方法不用传self也不用传cls）



#### 优点：

代码块:知道逻辑关系，代码清晰

复用性:任何派生类搜可以调用

## day20-04类多继承c3算法（20）



最后结果【B,C,D,F，E,G,H】

第一步生成3个列表(先按父类继承的顺序递归，然后在写自己的)

【B，D，F，H】，【C，E,G,H】,【D，F，H】 【B,C,D】

第二步对每个列表进行分割（列表中第一个元素是头，其他是尾巴）

【B，|D，F，H】，【C，|E,G,H】,【D，|F，H】 【B,|C,D】

第三步判断第一个列表的头是否在其他列表中的尾巴出现，不出现就吧这个头加到最后结果中,然后对每个列表中删除这个列表的头，出项了话就跳到第二个列表的头，做相同的操作。

【D，|F，H】，【C，|E,G,H】,【D，|F，H】 【C|,D】

第四步 D出现在其他列表的尾巴中，所以我们跳到第二个列表的头，c没有出现在其他列表的尾巴中，我们在最后结果加入C,并在其他列表中删除C

【D，|F，H】，【E,|G,H】,【D，|F，H】 【D|】

第五步，我们从第一个列表开始，头尾D,发现D不出现在其他列表的尾巴中，我们在最后结果中加入D,并在所有的列表中删除D

【F，|H】，【E,|G,H】,【F，|H】 【】

第六步，我们喧杂F,F没有出现在其他列表的尾巴中，我们加入F,并在所有的列表中删除F

【H|】，【E,|G,H】,【H|】 【】

第七步选H,H出项在第二个列表中，我们换成E,E没有出现在其他列表的尾巴中，我们加入E,并在所有的列表中删除E

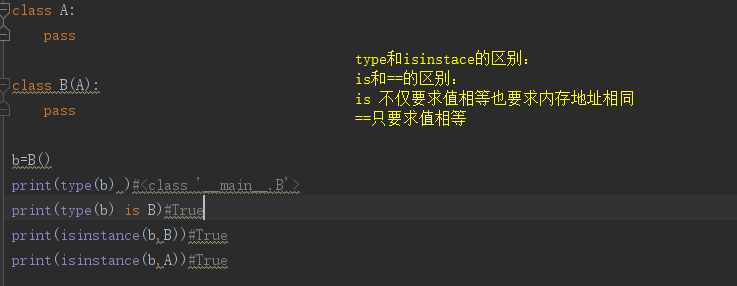
【H|】，【G,|H】,【H|】 【】

最后G,H

## day21-02(内置函数)（21）

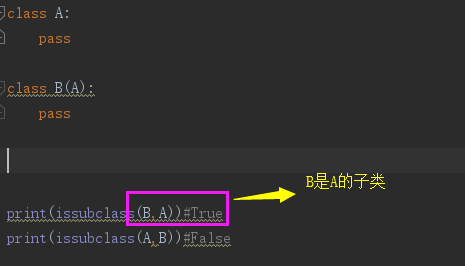
### isinstace（）

判断某个对象是不是这个类型的，与type的区别是，type只能判断到当前类，不能判断到它的父类（值判断当前层），isinstace可以判断到父类（判断到上几层）



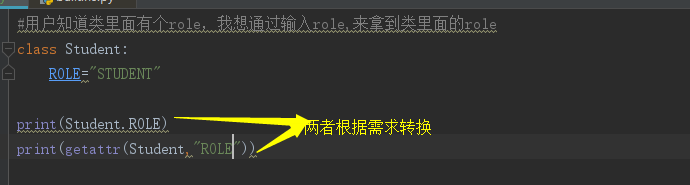
### issubclass()

判断类与类之间的继承关系



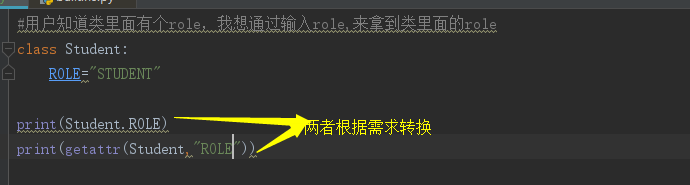
## day21-03（反射）（22）

### 需求（你只能拿到一个字符串版本，从文件里拿或者是交互拿（input/网络传输））

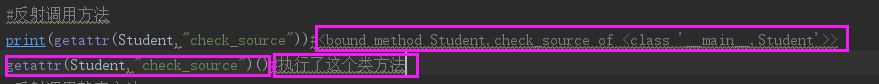


getattr(类名，字符串)#获得第一个参数的命名空间的变量名为第二个参数的值

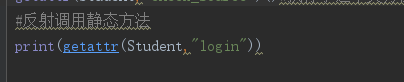
#### 反射调用静态属性



#### 反射调用类方法



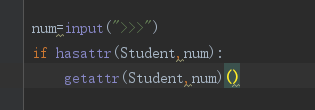
#### 反射调用静态方法



#### 使用流程

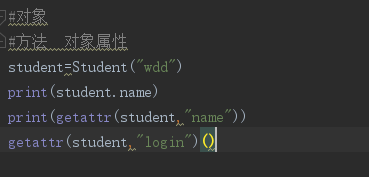
首先判断类里面是否有这个名称存在 hasattr(类名，名称)

有的话就调用getattr(类名，名称)

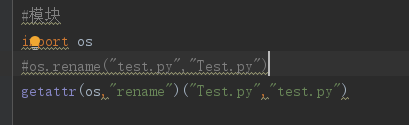


## day21-04(23)(反射的应用)

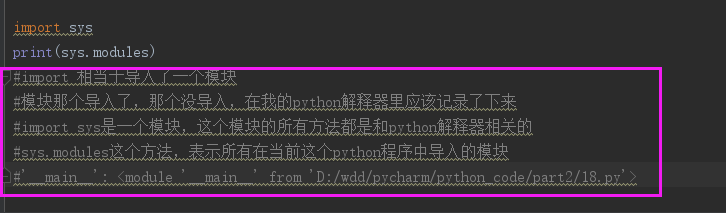
### 使用对象来反射对象属性

、

### 使用模块来反射模块内的属性和方法

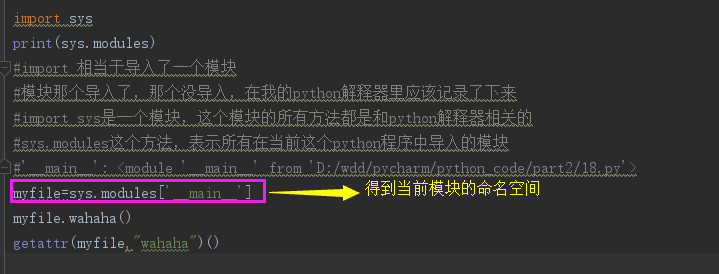


### 反射自己模块的内容 找到自己当前文件所在的命名空间



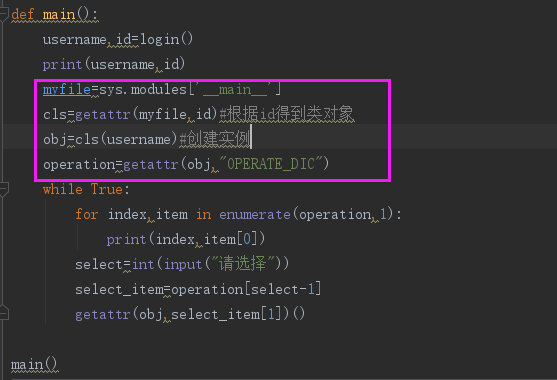
找到main对应的地址就是当前模块的地址

'\_\_main\_\_': <module '\_\_main\_\_' from 'D:/wdd/pycharm/python\_code/part2/18.py'>





### 反射的典型应用（高大尚的代码必备）



class Manager:  
 OPERATE\_DIC=[("创建学生账号","create\_student"),("创建课程","create\_course"),("查看学生信息","check\_student\_info")]  
 def \_\_init\_\_(self,name):  
 self.name=name  
 def create\_student(self):  
 print("创建学生账号")  
 def create\_course(self):  
 print("创建课程")  
 def check\_student\_info(self):  
 print("查看学生信息")  
class Student:  
 OPERATE\_DIC = [("查询课程", "check\_course"),("选择课程", "chose\_course"),("查看已选择课程", "choosed\_course")]  
 def \_\_init\_\_(self,name):  
 self.name=name  
 def check\_course(self):  
 print("查询课程")  
 def chose\_course(self):  
 print("选择课程")  
 def choosed\_course(self):  
 print("查看已选择课程")  
def login():  
 username=input("use")  
 password=input("pas")  
 with open("data.txt",mode="r",encoding="UTF\_8") as f:  
 for line in f:  
 list=line.strip().split("|")  
 if username==list[0] and password==list[1]:  
 print("登陆成功")  
 return list[0],list[2]  
import sys  
def main():  
 username,id=login()  
 print(username,id)  
 myfile=sys.modules['\_\_main\_\_']  
 cls=getattr(myfile,id)#根据id得到类对象  
 obj=cls(username)#创建实例  
 operation=getattr(obj,"OPERATE\_DIC")  
 while True:  
 for index,item in enumerate(operation,1):  
 print(index,item[0])  
 select=int(input("请选择"))  
 select\_item=operation[select-1]  
 getattr(obj,select\_item[1])()  
  
main()

## day21-05(24)(\_\_call\_\_)

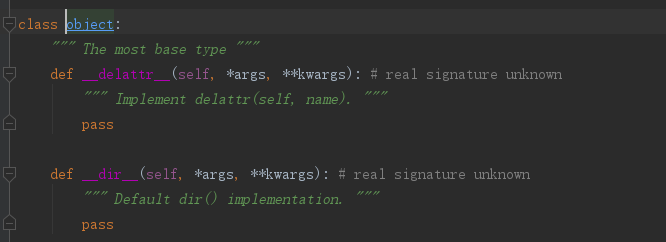
类中看到这个 \_\_名字\_\_ 类中的每一个双下方法都有它自己的特殊意义

#类中的特殊方法

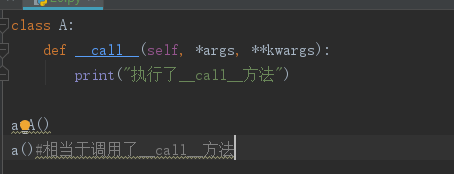
#双下方法

#魔术方法

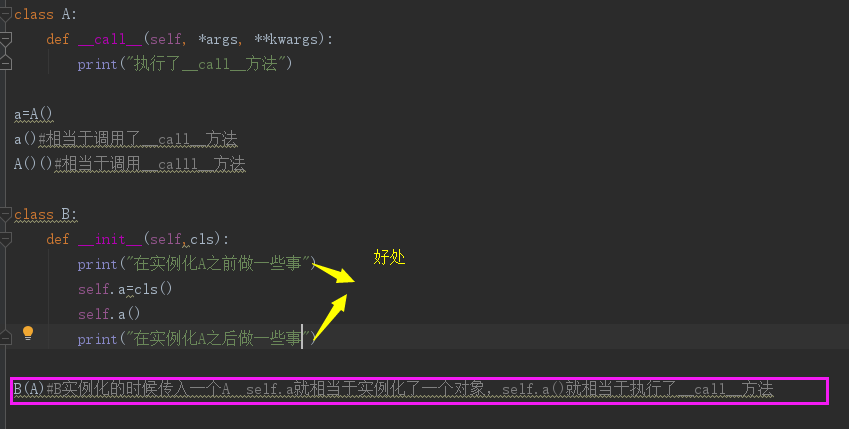
内置方法



### \_\_call\_\_



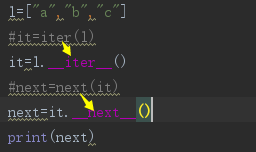
实例化对象后面跟括号等于调用了\_\_call\_\_方法



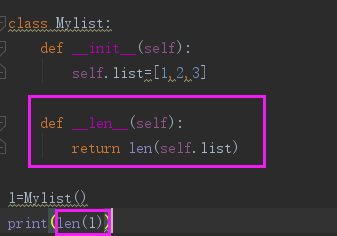
## day21-06(25)(\_\_len\_\_)

### \_\_len\_\_（len(obj)自动调用对象类的\_\_len\_\_,\_\_len\_\_方法return的值就是len函数的返回值）

举个例子



证明了：python的内置函数和类的内置方法是有奸情的

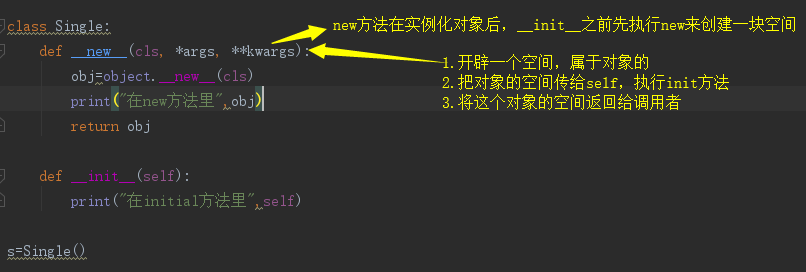


## day21-07(26)(\_\_new\_\_)

### 区别

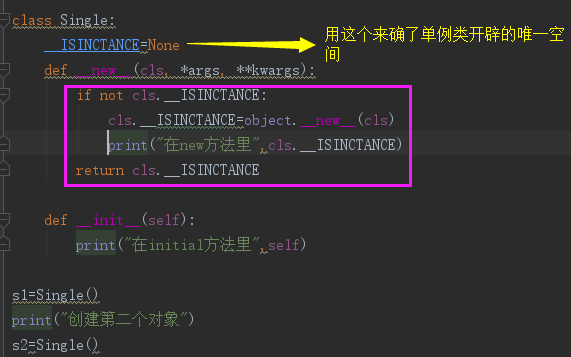
\_\_new\_\_构造方法

\_\_init\_\_初始化方法



### 单例类（如果一个类，从头到尾只能有一个实例，那么这个类就是一个单例类）

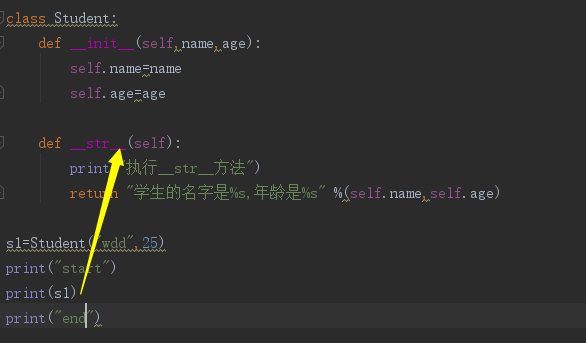
单例类只要开辟一个空间了



## day21-08(27)(\_\_str\_\_)

通过打印内存地址显示地址里的内容

Print(地址)#如果类中实现了\_\_str\_\_方法的话，就调用这个方法，没有实现的话就调用object的\_\_str\_\_方法返回内存地址



str(obj),相当于执行obj.\_\_str\_\_方法（作用是把一个类转成字符串，如果这个类实现了\_\_str\_\_方法就执行，没有实现的话就执行Object的\_\_str\_\_返回一个内存地址）



字符串的格式化%s



总结：所有的双下方法，没有需要你在外部调用的，而是总有一些其他的内置函数特殊分语法，来自动触发这些双下方法

## day22-01(面向对象复习)(28)

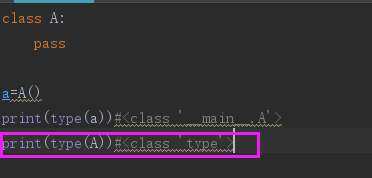
### 什么时候用面向对象

1. 代码量大，功能多的时候
2. 处理比较复杂的角色之间的关系

### 创建一个类

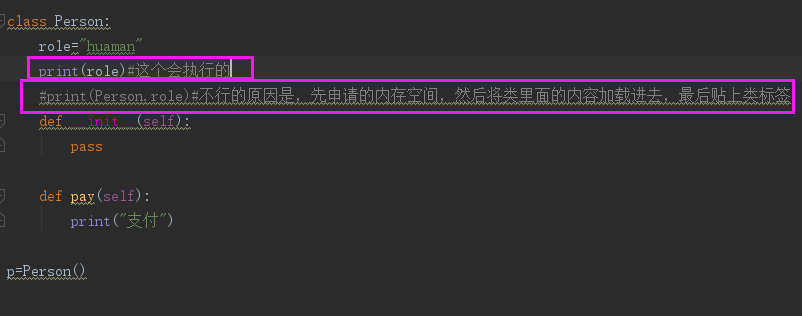
class 类名 语法级别的python解释器读到这句话的时候，会创建类，使用type创建类

创建抽象类的class A(metaclass=ABCMeta)(ABCMeta创建了这个A类，那么ABCMeta就是A的元类)，只不过创建普通类的时候省略了括号和里面的内容。完整的创建普通类是这样class B(metaclass=type)（type创建了B,type就是普通的类的元类）。某一个类的type是type

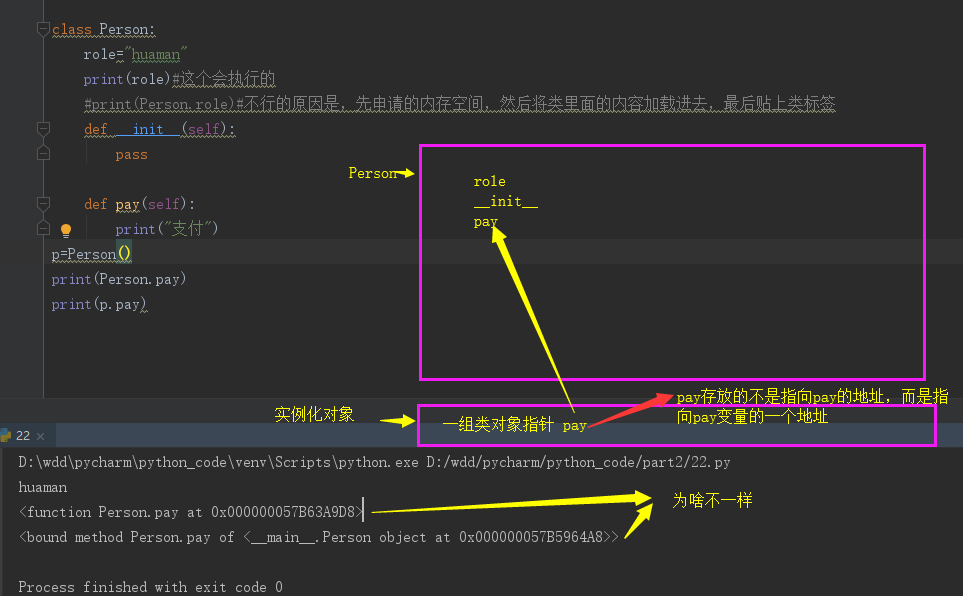


### 坑一

类是什么时候被加载的，以及类名是什么时候生效的

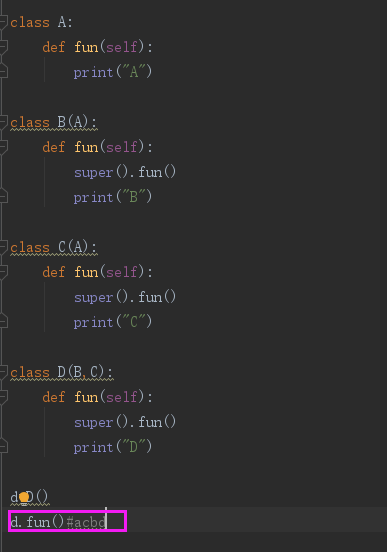


### 坑二



举个例子：我知道你家的地址，我怕记不住，我写在本子上，当别人问我你家在哪里的时候，我说在本子上，从本子里找你家的地址

### 继承



## da22-02(29)(抽象类和接口类)

### 抽象类和接口类

抽-象类的方法是可以实现的，只能单继承

接口类可以多继承，但是接口类的所有方法不应该被实现

## day22-03(30)(多态和鸭子类型)

多个子类去继承父类，那么每个子类都是这个父类的一种形态

Python中不要声明的原因是所有的类都是object的子类

## day22-04(31)(封装)

广义的封装

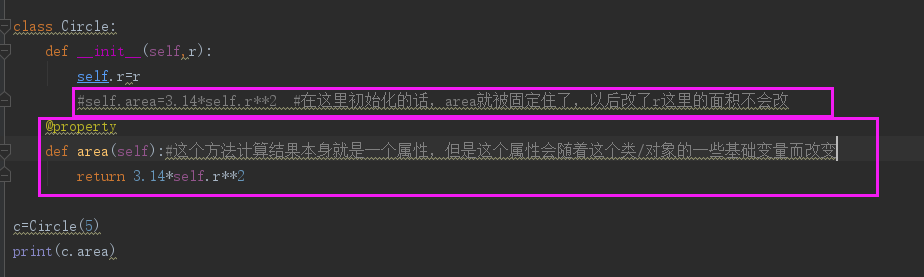
狭义的封装（私有） 为什么在类的内部可以使用双下划线访问：在类的内部，你知道在那个类中

## day22-05(32)(装饰器)

property装饰器函数，内置函数，帮助你将类中的方法伪装成属性，特性

@方法名.setter 装饰器，修改被property装饰的属性的时候调用这个装饰器的方法

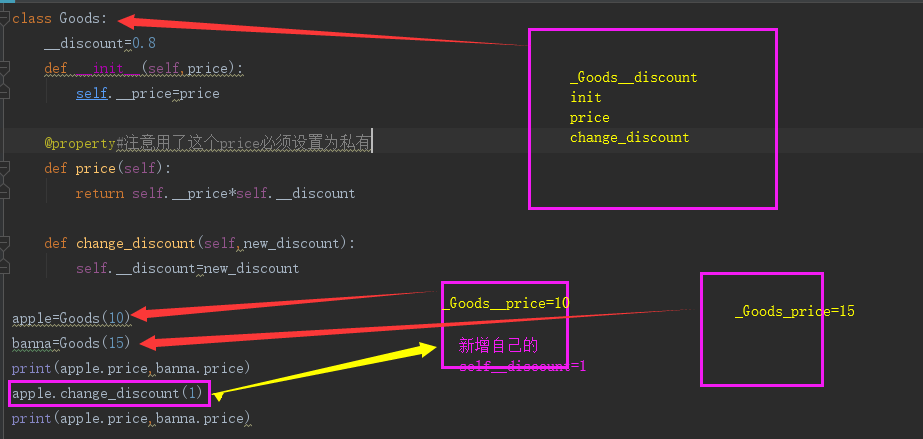
@方法名.deleter装饰器，当要删除被property装饰的属性的时候会调用这个



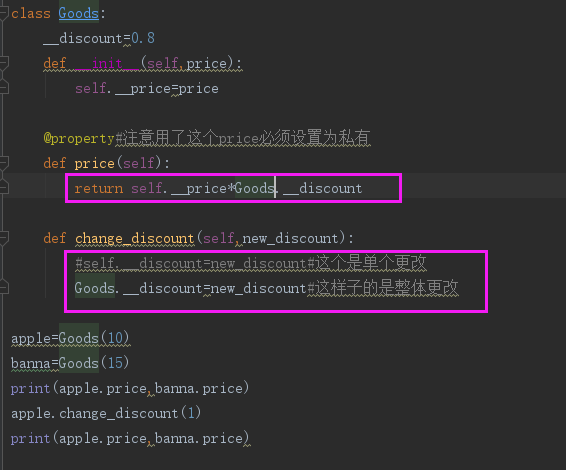
## day22-06(33)(类方法和静态方法)

### 类方法应用场景

类方法不用对象命名空间的内容，而用到了类命名空间中的变量（静态属性），或者类方法和静态方法。



粉红色的部分表示在自己的对象空间中新增了自己的\_\_discount，然后实例化本身在自己的空间中找到，就不需要到类空间去找了



#### 最终使用类方法

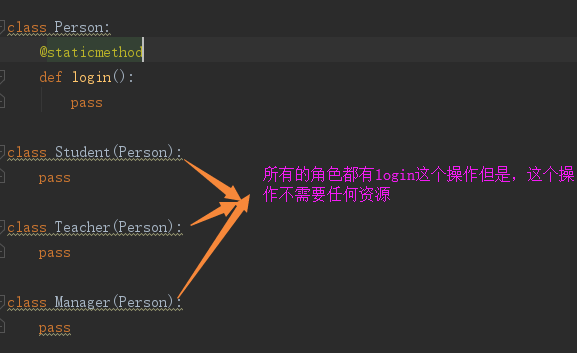


### 静态方法的应用场景

类里面的方法既不需要用到self中的资源，也不用cls中的资源，相当于一个普通函数，但是你由于某种原因，还是要把这个方法放在类中，这个时候就将这个方法变为静态方法

某种原因：

1. 你完全想用面向对象编程，所有的函数都必须写到类里面
2. 某个功能确确实实是这个类的方法，但是确确实实没有用到和这个类有关系的资源



## day22-07(34)(List中的insert(0,15)和pop(n))

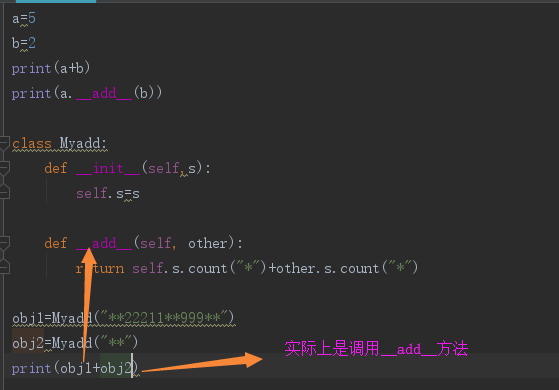
pop和insert效率很低（尽量不要想列表中插入和删除但是可以用append和pop()，这才能提高时间质量）

## day22-08(35)(反射)

过

## day22-09(36)(内置方法的复习)

a+b=c实际上是调用类内的\_\_add\_\_



## day22-10(37)(repr方法）

repr方法对应\_\_repr\_\_方法

### repr和str的区别

\_\_str\_\_:str(obj),要求必须实现了\_\_str\_\_,要求这个方法的返回值必须是字符串str类型（print,%s，str）

\_\_repr\_\_:是\_\_str\_\_的备胎，如果有\_\_str\_\_方法，那么（print,%s,str）都去执行\_\_str\_\_方法，并且使用\_\_str的返回值，没有的话那么（print,%s,str）都会执行repr方法repr只管repr(obj),%r

两个只能实现一个的话，我们优先实现\_\_repr，他的功能多点

重点：在子类中使用\_\_str\_\_，先找子类的\_\_str\_\_。没有的话要向上找，只要父类不是object,就执行父类的\_\_str\_\_，但是如果除了object之外的父类都没有\_\_str\_\_方法，就执行子类的\_\_repr\_\_方法，如果子类也没有，还要继续向上找父类的\_\_repr\_\_方法，一直找不到在执行object类中的\_\_str\_\_方法

## day22-11(总结38)

## day23-01(总结39)

双下方法的触发条件：内置函数，特殊语法，语法糖，内置模块

\_\_len\_\_ 内置函数len

\_\_call\_\_ 特殊语法 对象（）

\_\_new\_\_ 特殊语法，构造方法，在实例化的时候\_\_init\_\_之前调用

\_\_init\_\_ 特殊语法，初始化方法，在实例化的时候\_\_new\_\_之后调用

\_\_repr\_\_ repr %r

\_\_str\_\_ str ,%s ,print

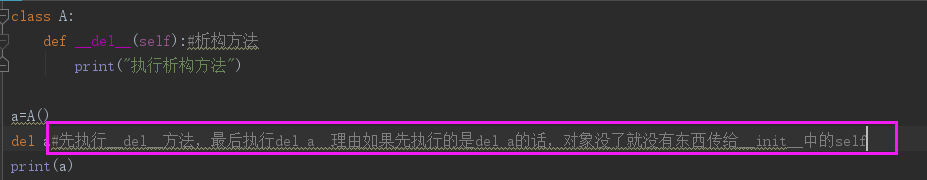
## day23-02(40)

析构方法：

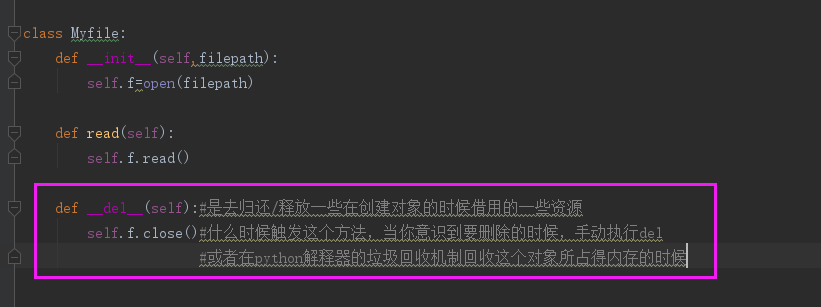
## day23-03(41)

### 析构方法：释放一个空间之前执行（某对象借用操作系统的资源，还要通过析构方法归还回去，比如文件资源，网络资源）

del A的对象



### 应用场景



什么事后需要实现\_\_del\_\_方法，什么时候不需要

说明：

Python解释器在内部就能搞定的事儿

Python解释器申请一块空间，这是操作系统分配给你的

在这一块空间之内左右的事儿，归你的python解释器来管理（所以仅在这块空间创立的对象的类不需要实现\_\_del\_\_方法）

但是f=open(),python没有权利去操作硬盘里的文件，所有的硬件都在操作系统中管理，需要通过操作系统，操作系统替你打开文件，然后给你一个文件操作符（句柄），然后这个文件俺操作符拿到内存中存起来，当你不用文件的时候，删除这个文件操作符，但是操作系统给为你提供的资源没有关闭，需要你给出关闭指令

del f 是释放的是python解释器存储的内存

f.close()是释放的操作系统打开的资源

补充说明：withopen没有open安全，因为当with的时候出现异常的话，就会中断在异常哪里，有三个异常withopen处理不了

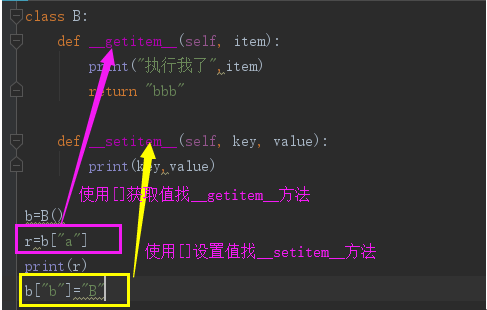
## day23-04(item系列)(42)

### item系列和对象使用[]访问值由联系

#### 使用[]访问值



#### 使用【】设置值

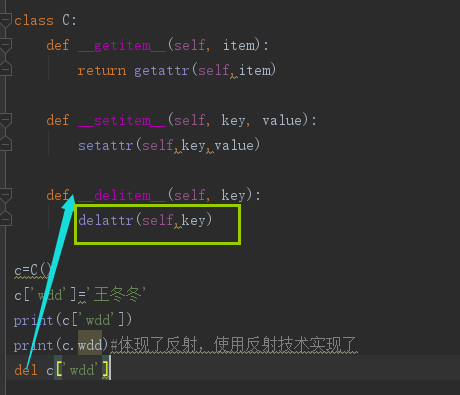


#### 使用[]设置值和使用[]访问值（利用反射实现）

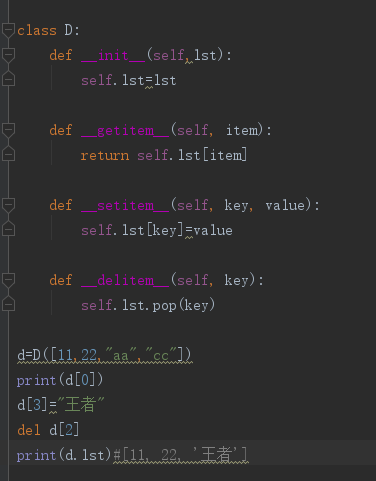


#### 什么时候使用这种组合（在内置的模块中，有一些特殊的方法要求对象必须实现这两个）

#### 使用【】删除值



#### 一个简单的应用实例（通过对象操作对象里的数据类型）



#### 作业

类

么一个对象都是一副扑克牌

我想查看这个对象 来查看整副牌

我想从这一副牌中随机抽取一张牌

我想完成打乱这副牌的顺序的功能

## day23-05(42)(hash方法)

### hash算法（hash(obj),obj内部必须实现了\_\_hash\_\_方法）

hash是一个算法，能够把某一个要存在内存里的值通过一些列计算，保证不同值得hash结果是不一样的，对同一个值在多次执行python代码时候hash值是不同的，但是对同一个值，在同一次执行python代码的时候hash值永远不变

print(hash('abc'))

### 字典寻址快的原因

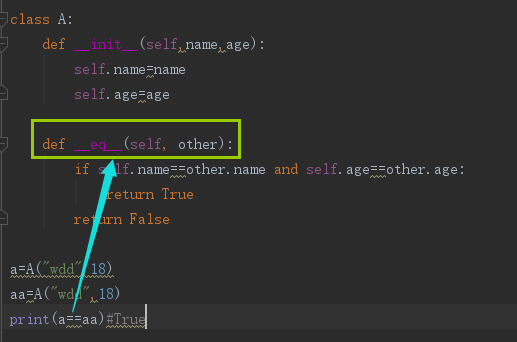
{“key”，“value”} 首先计算hash(key)，找到地址，在这个地址存value,查询的时候，我们根据key，再次计算hash(key)来到对应的地址找value

### Set集合

Hash不是万能，如果再一次寻址过程中两个对象计算出两个相同的hash地址的话，就会判断这两个值一样不，不一样的话进行第二次hash寻址，这样就不会因为hash值重复而覆盖

## day23-06(eq方法)(43)

### ==调用对象函数中的\_\_eq\_\_



## day23-07(面试题)(44)

一个类（姓名，性别，年龄，新的部门）对1000个员工，如果几个员工对象的姓名和性别相同，这是一个人，对人去重



## day23-08(初识模块)(45)

模块的分类：

1. 内置模块（安装python解释器的时候安装的）
2. 自定义模块（你自己写的功能，如果你写的代码是一个通用的功能）
3. 第三方模块/扩展模块（没有安装在python解释器，比如）

什么是模块：

1. 有的功能开发者自己无法完成，这样的话需要借助已经实现的函数/类来完成这些功能
2. 你实现不了的功能都由别人替你实现了

哪些用到模块:

1. 和时间打交道
2. 1000取随机数
3. 和网络通信

为什么要有模块：

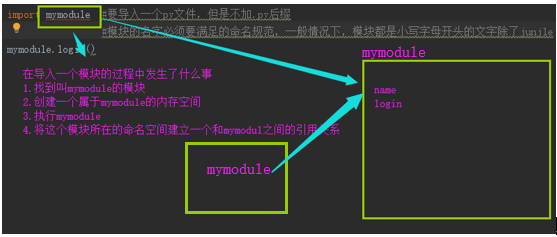
1. 分类，管理方法
2. 节省内存
3. 提供更多的功能

模块怎么用？在哪里用

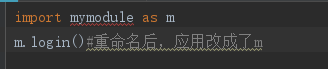
## day23-09(模块的导入和使用)（46）

模块不会被重复导入，即使你导入了多次

### 在导入一个模块的过程中发生了什么事



### 模块的重命名（原来的模块名酒不好用了）

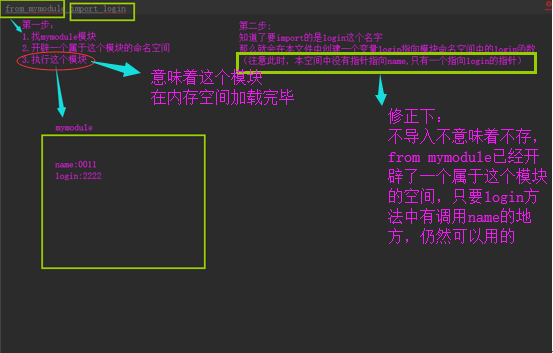


### 模块的导入顺序

先导入内置模块，在导入第三饭模块，最后导入自定义模块

## day24-01，02(47)

## day24-03(48)



不导入不意味着不存，而是没有建立文件到模块中其他名字的引用

### 在使用from module import AA时本模块和被调用模块同时存在名为AA的函数

由于from module import AA定义在开头，所以我的命名空间中有个名为login的函数，当我在我的模块中重新定义一个函数的时候，相当于覆盖掉之前的login函数

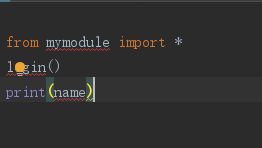
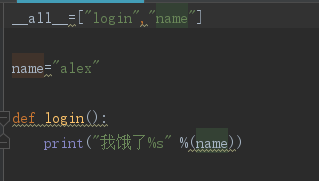
### from module import AA as BB是把AA重命名BB

### 导入多个变量名重命名

from module import login as l,name as n

from module import \*(所有调用模块的引用，在本空间都有个同名的引用)

### \_\_all\_\_控制 import \*的导入（不使用的话，默认导入所有的变量名，使用了话，只导入\_\_all\_\_里面的）



## day24-04(49)(把模块当成脚本运行)

### 运行一个py文件的两种形式

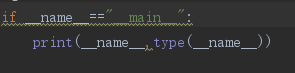
Import my\_module（以模块的形式运行）

直接右键运行或者cmd运行（这就是以脚本的形式运行）

### print(\_\_name\_\_)

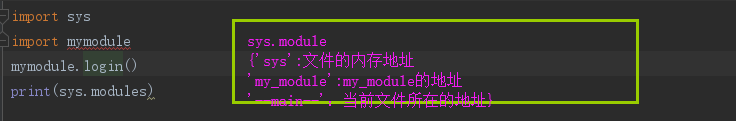
\_\_name\_\_当你以脚本的形式运行时\_\_name\_\_就是\_\_main\_\_,当你以导入模块的形式运行时就是导入的模块名

### 控制执行自己模块时，导入模块中的print语句不打印，但是单独执行导入模块时，导入模块的print语句打印



提示：所有不在函数和类中封装的内容都应该写在if \_\_name\_\_==”\_\_main\_\_”：下面

### sys.module记录所有的地址

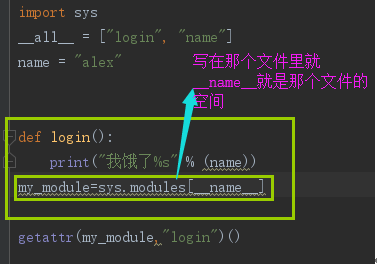


我执行的是那个文件，\_\_main\_\_就存那个文件的地址

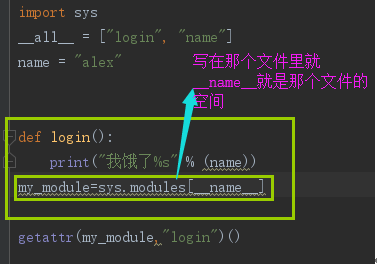
### 坑

我自己写的脚本里涉及到获取当前文件所在的地址的时候用\_\_name\_\_(因为python会把当前空间的地址赋给\_\_name),这样的我我调用这个模块的执行到获取模块空间地址的时候不会报错（如果你用的是my\_module=sys.modules["\_\_main\_\_"]的话，传过来的\_\_main\_\_是你执行的那个文件的地址

）



## day24-05(50)(反射本模块中的内容)

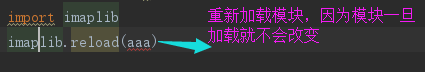


day24-06(51)(模块的其他知识)

### pyc文件

当一个文件作为一个脚本被导入的时候，就会在这个文件所在目录的\_\_pycache\_\_下生成一个编译了好的pyc文件，为了之后导入这个文件的时候直接读这个编译好的pyc文件，可以节省一些导入时候的时间

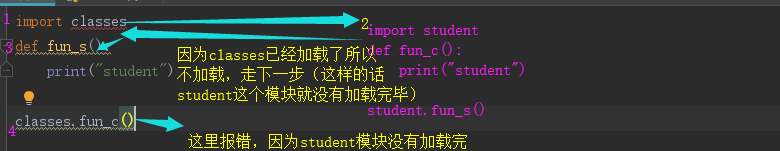
### 重新加载模块



## day24-07(51)(模块总结)

## day24-08(52)(模块的循环引入)

两个模块一个学生模块一个班级模块，学生模块中导入班级，班级模块中导入学生，这就是模块的循环导入



结论：在模块的导入中，不要让你的模块存在循环导入的问题。如果发生循环导入就会报错

## day24-09(包)（53）

包:集合了一组py文件，提供了一组复杂功能

为什么会有包：当提供的功能比较复杂，一个py文件写不下的时候

### 导入包中的模块：

#### 直接导入模块

Import 包.包模块

包.包.模块.变量

推荐下面的方式

From 包.包 import 模块

模块.变量

#### 复杂的导入（导入包，然后包下面的内容可以用）

##### 模块包中的绝对导入(缺点：目录一移动都要改)

导入包相当于执行了包下面的\_\_init\_\_.py

所以我们可以在\_\_init\_\_下做处理（不管在那个文件导入那个模块，From 包.包 import 模块 粉色的部分都要写全路径）

##### 模块包的相对导入（缺点：只能当作模块执行）

使用.代替当前目录

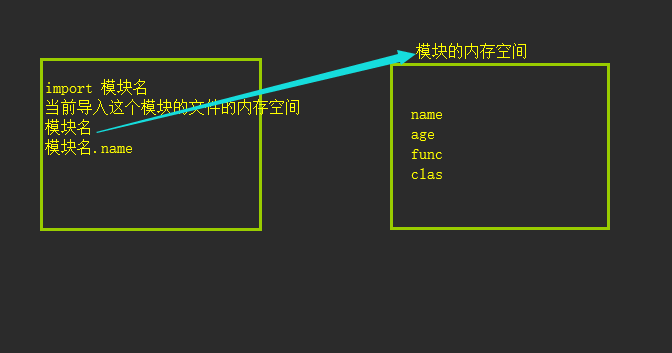
注意（使用了相对模块的只能被当作模块执行，不能被当作脚本（直接运行run）执行）

## day24-10(54)(总结)

import模块名

导入这个模块中的某个名字之后，这个名字可以直接使用

模块名.名字



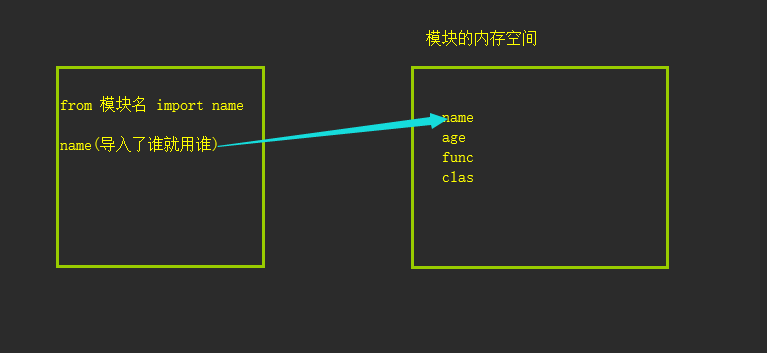
From 模块 import 名字

导入这个模块中的某个名字之后，这个名字就可以直接使用了

名字是变量 直接用

名字是函数 函数名（）就是调用

名字是类名 类名()就是实例化



### Sys.path

一个自定义模块能够被导入，就看sys.path列表中有没有这个模块所在的绝对路径

### 包

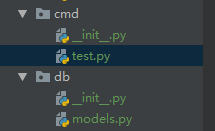
从包中导入模块，注意这个包所在的目录是否在sys.path中

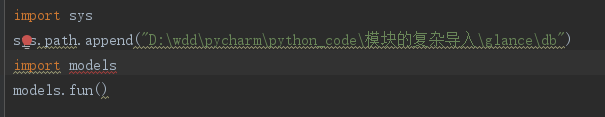
重点：如果是直接导入一个包，那么相当于执行了这个包中的\_\_init\_\_文件，并不会帮你把这个包下面的其他包以及py文件自动导入到内存，如果你希望直接导入包之后，所有的这个包下面的其他包以及py文件都能直接通过包来引用，那么你要自己处理\_\_init\_\_

## day24-11(55)(模块导入练习)

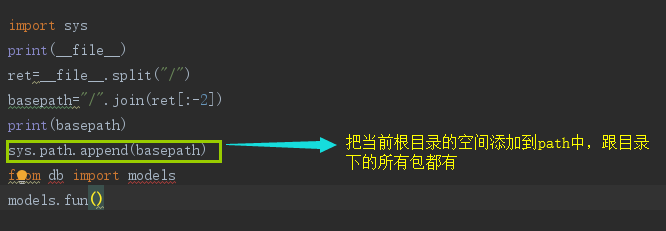
### 新增测试方法（这是死板的）

发现导入一个模块失败的时候（使用sys.path.append(“这个模块的地址”）

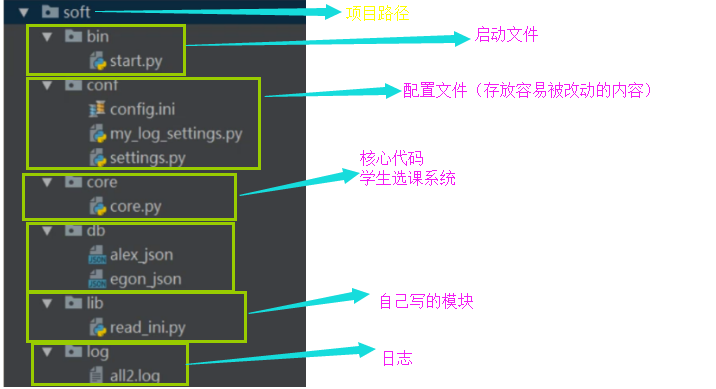




### 推荐灵活写法



## day24-12(56)(项目开发规范)



## day24-13(选课系统开发)(57)

作业

## day25-01(58)（常用模块）

## day25-02(59)(内容回顾)

## day25-03(60)(正则表达式的概念)

什么是正则表达式

一种匹配字符串的规则

正则表达式能做什么

可以制定一个规则，来确定某一个字符串是否符合规则

帮助学习的工具: http://tool.oschina.net/regex/

## day25-04(61)(正则表达式的基本语法)

### 字符组

字符组[ ]在某一个字符的位置上能出现的内容

[abc]表示在某一个字符的位置上能出现的内容是a或b或c

匹配所有的数字【0-9】（注意匹配范围只能正着写，~~不能倒着写【9-4】~~）

匹配所有的小写字母【a-z】

匹配所有的大写字符【A-Z】

[0-9][A-Z][a-z]这是三个字符

既能匹配数字又能匹配字符[0-9a-zA-Z]

【0-9abc】这也是一个字符

匹配18位身份证号

【1-9】【0-9】【0-9】【0-9】【0-9】【0-9】【0-9】【0-9】【0-9】【0-9】【0-9】【0-9】【0-9】【0-9】【0-9】【0-9】【0-9】【0-9Xx】

### 元字符

元字符:用一个字符串来匹配内容

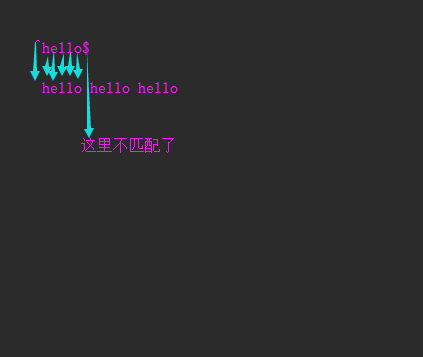
|  |  |  |
| --- | --- | --- |
| . | 匹配除换行符以外的任意字符 | 如果匹配2.3呢  将.转义\.  \d\.\d |
| \w=[0-9a-zA-Z\_] | 匹配字母或数字或下划线 | Word |
| \s==【\n \t】 回车空格制表符 | 匹配任意的空白符 | space |
| \d=[0-9] | 匹配数字 | digit |
| \n | 匹配一个换行符 |  |
| \t | 匹配一个制表符 |  |
| \b | 匹配一个单词的边界（边界部分前后） | 字符\b表示字符后面的是边界  \b字符 表示字符前面的是边界 |
| ^ | 匹配字符串的开始 | 只从开头匹配 |
| $ | 匹配字符串的结尾 | 只匹配结尾的 |
| \D | 匹配非数字 |  |
| \S | 匹配非空白符 |  |
| a|b | 匹配字符a或字符b | 注意如果a和b有重合的地方，把长的一个写下前面（原理是一旦前面的满足就不会匹配后面的）比如abc|ab |
| () | 匹配括号内的表达式，也表示一个组 |  |
| [..] | 匹配字符组中的字符 |  |
| [^..] | 匹配除了字符组中字符的所有字符 |  |
| \W | 匹配非字母或数字或下划线 |  |
|  |  |  |
|  |  |  |

\n 匹配回车

\t 匹配制表符

【\d\D】一对大小写匹配所有的字符串

^hello$和hello hello hello的匹配结果（这样写的话，不仅要求内容一样还要求长度一样）



规则：从头开始去匹配一个字符串，能匹配到多个符合规则的就是多条结果，带着^号的只匹配到一个结果

### 量词

### 特殊的用法和现象