# Python

## Base

### 数据类型

Int

Float

Str

List

### 可变函数和不可变类型

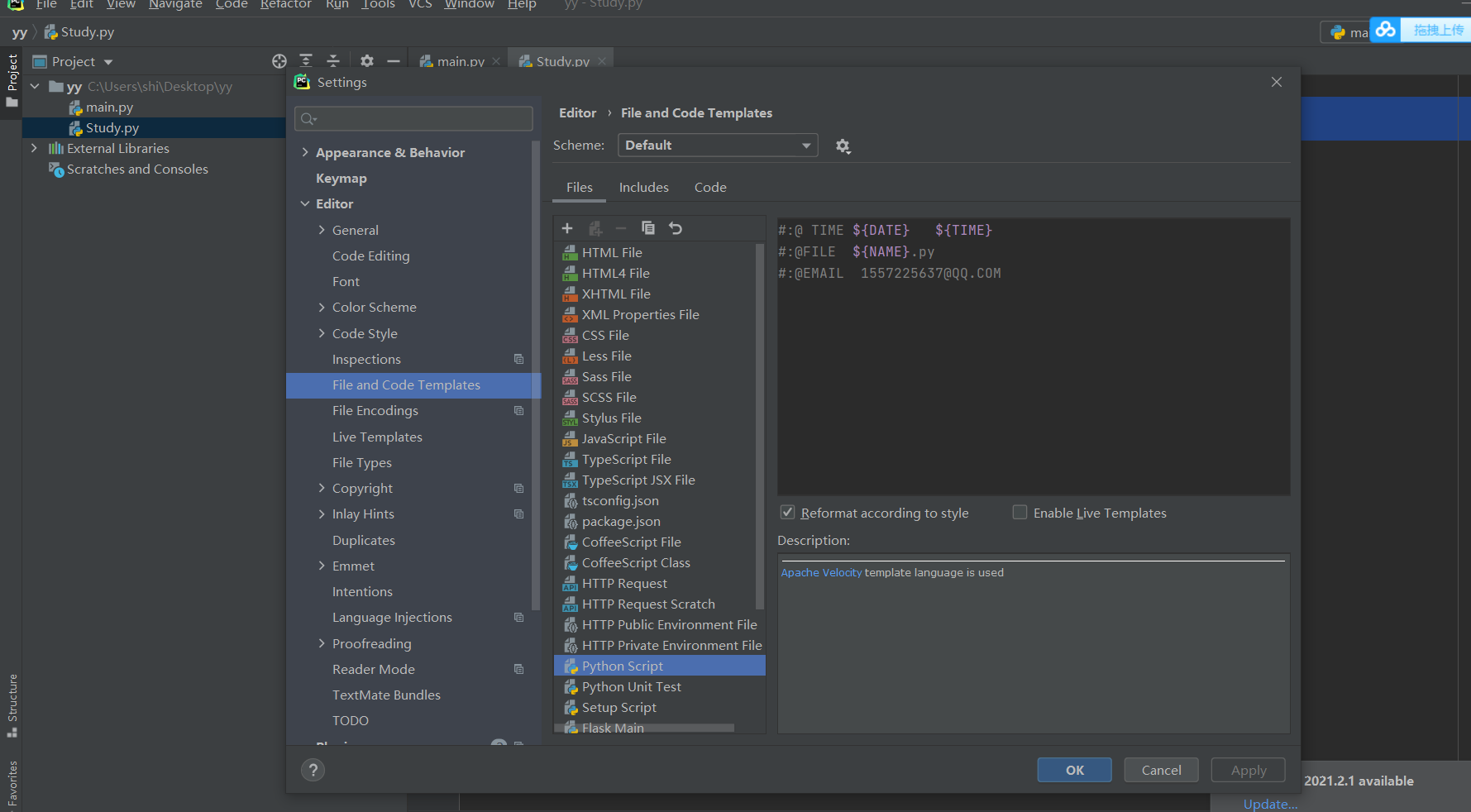
不可变：对象所指向不可以改变

不可变的类型： int str float 元组 tuple

可变类型：对象所指向的内存中的值是可以改变的

类型：dict list

### 配置模板



### 运算符

#除  
print(5/2)  
  
#整除  
print(10//6)  
  
#取模  
print(11%3)  
  
#次方  
print(10\*\*9)

2.5

1

2

1000000000

### 转义符

print('hello 1\nhello2\nhello3\nhello4') # ”转义符\n的使用“直接\n换行  
  
print('hello 1\thello2\thello3\thello4') # ”转义符\t的使用“\t相当于四个空格键  
  
print('hello 1\rhello2\rhello3\thello4') # ”转义符\r的使用“\r前面的所有代码不输出  
  
print('hello11\b\thello22\thello33\thello44') # ”转义符\b的使用“\r前面一格的代码不输出  
  
print('http:\\\\www.baidu.com') # 输出网址双斜杠时要多加两个斜杠  
  
print('老师说：\'你们好\'') # 输出引号要用到斜杠  
  
# 使用原本的字符 不希望字符串中的转义符起到作用，就使用原字符，在字符串之前加上r或R  
  
print(r'hello 1\nhello2\nhello3\nhello4') # 字符串之前加r，让字符串中的转义符失效

\

,/UBH

切片

a=('123456789')  
print(a[2])  
print(a[6])  
print(a[-3])  
print(a[0:6])  
print(a[0:-4])  
print(a[-9:-3])  
print(a[-3:5])  
print(a.count("6"),"统计字符串内有几个6")  
print(len(a),"获取字符串的长度")#,''  
print(a.find("7"),"获取该字符串左边的字符串")  
print(a.index("3"),"获取该字符串的坐标")

print(a[0:]) 冒号后面为空切片到结尾

3

7

7

123456

12345

123456

1 统计字符串内有几个6

9 获取字符串的长度

6 获取该字符串左边的字符串

2 获取该字符串的坐标

### List

#### list内置函数

append，extend，insert，

Remove，pop ，clear，del

Sort ,reverse ,index ,count

定义  
list  
a=[] 空列表  
  
2.符号：  
+ ----->[]+[] 合并  
\* -----> []\*n 相乘  
*in* a *in* [] false true  
*not in* [] alse true  
*is* 地址是否相等  
*not is* 地址是否不相等  
  
  
3.系统中给列表用的函数  
len()  
 sorted() 排序  
max()  
min()  
list（a） 强制转换list类型  
sum 求和  
  
4.列表自身的函数  
append（） 末尾添加  
extend 末尾添加一组元素/两个列表相加  
insert 指定位置插入  
  
  
5.删除  
*del* list[index]  
  
remove 删除指定元素，如果指定元素未找到就报异常  
  
a.pop() 删除a列表内的最后一个元素并返回删除的元素  
  
a.clera 清除列表内的所有元素  
  
6.其他  
  
count（） 指定元素的个数  
  
sort 排序

reverse 翻转

#### 列表[]（增）

##### Random 随机数插入列表并排序

#需求 生成100个 1到9999的随机数、并插入到列表a或者b里面且进行排序，如果生成的数为4为数就放到a，小于4位数就放置b  
*import* random  
big\_list=[]#四位随机数  
little\_list=[]#三位随机数  
b=1 #统计生成四位数的总数量  
*for* i *in* range(101): #循环100次  
 a = random.randint(1,9999) #随机生成1到9999的int值  
 a=str(a) #把int类型转string  
 print("生成随机数第",i,"次") #第几次生成随机数  
 *if* len(a)==4:# 判断生成的值是否为4位数  
 b+=1 #统计生成四位数的总数量  
 print(a)  
 big\_list.append(a)#把值插入列表  
 *for* k *in* range(len(big\_list)):  
 *for* l *in* range(k+1,len(big\_list)):  
 *if* big\_list[k]>big\_list[l]:  
 m=big\_list[k]  
 big\_list[k]=big\_list[l]  
 big\_list[l]=m  
 *else*:  
 little\_list.append(a)  
 *if* i==100:  
 print("共生成随机数",i,"次","值>=1000的生成",b,"次")  
print("big\_list=",big\_list,)  
print("little\_list=",little\_list)

#### 列表[]（查）

#list 表

#类似 其他语言的数组。

#数组：数字的组合 字母的组合 字符串的组合。。。

#符号： 列表：[]

##### 3.3列表函数 index，索引出元素首次出现的下标

list1=*[*1,2,3,4,'王五',5,6*]*print*(*list1.index*(*'王五'*))* #4

##### 3.3.1函数index查找元素5，从第7位之后查找

#index查找元素5，从第7位之后查找

list1=*[*1,2,3,4,'王五',5,6,1,2,3,4,2,2,3,4,5,2,4,5,6,7,8*]*

print*(*list1.index*(*5,7*))* #15

##### 3.4 列表（声明，元素获取）

#声明  
names=['xiaohong','xiaohua','xiaoming','xuaoli','xiaobao','xiaolan']  
name2=[]  
print(names)

['xiaohong', 'xiaohua', 'xiaoming', 'xuaoli', 'xiaobao', 'xiaolan']

#查看是否分配地址  
print(id(names))  
print(id(name2))

2364456873216

2364456783616

names=['xiaohong','xiaohua','xiaoming','xuaoli','xiaobao','xiaolan']  
name2=[]

#元素获取 0~。。。  
print(names[1])  
print(names[2])

xiaohua

xiaoming

##### 3.5查、获取列表中最后一个元素

names=['xiaohong','xiaohua','xiaoming','xuaoli','xiaobao','xiaolan']

#获取最后一个元素  
print(names[-1])

Xiaolan

names=['xiaohong','xiaohua','xiaoming','xuaoli','xiaobao','xiaolan']

#通过获取列表的长度获取最后一个元素,切片从0开始，列表长度从1开始加  
print(names[len(names)-1])

Xiaolan

##### 3.6 用for循环取出列表的所有值

names=['xiaohong','xiaohua','xiaoming','xuaoli','xiaobao','xiaolan']

#用for循环取出列表的所有值  
for cc in names:  
 print(cc)

xiaohong

xiaohua

xiaoming

xuaoli

xiaobao

xiaolan

##### 3.6 把列表内 指定的元素 所有的的下标给输出来

#把列表内 指定的元素 所有的的下标给输出来  
list2='123456123456123456123456' #定义字符串  
*for* i *in* range(len(list2)): #便利字符串元素  
 *if* list2[i]=='3': #判断是否指定元素  
 print(i) #输出指定元素的位置

##### 3.7 用for循环输出字符串

#for循环打印字符串 ，结果每次区一个字符串输出  
for i in 'hello':  
 print(i)

h

e

l

l

o

注意 in后跟的是引起来的字符串不是变量

##### 3.8 用for循环判断 字符串是否在列表内

#查找小明列表里有没有小明 如果列表里面有xiaoming.就停止打印,原理：从names里面取出一个值赋值给 a 变量  
names=['xiaohong','xiaohua','xiaoming','xuaoli','xiaobao','xiaolan']  
for a in names:  
 print(a)  
 if a=='xiaoming':  
 print('找到小明了')  
 break  
 else:  
 print('找不到小明')

循环前两遍没有找到小明，走的else’，第三遍找到了 跳出循环

xiaohong

找不到小明

xiaohua

找不到小明

xiaoming

找到小明了

##### 3.6 简易版 用 if 字符串 in 列表，判断是否在列明内

#简易版查找列表里有没有小明  
if'xiaoming' in names:  
 print('小明在列表里面')  
else:  
 '小明不在里面'

小明在列表里面

#### 列表[]（改）

names = ['l1','l2', 'l3', 'l4', 'l5', 'l6', 'l7', 'l8', '支持华为', 'l10']  
  
# 更改列表内的参数  
# 第一步 用for 循环判断 想要更改的字符串在不在 列表内  
for i in names:  
 if i=='支持华为':  
 print('在列表内')  
 break  
else:  
 print('不在列表内')  
  
  
#简易版判断字符串是否在列表内  
cc='支持华为' in names  
print(cc)  
  
  
  
  
#第二步 index（） 出字符串的位置 去修改  
bb=len(names)  
  
#第三部 用列表的位置改参数  
print(names.index('l10'))  
names[9]='支持小米'  
print(names)

找到元素 直接赋值更改

C:\Shi\Python\python.exe "C:/Shi/PyCharm 2020.3.3/oyo age/基础1/列表/列表 list表（改）.py"

在列表内

True

9

['l1', 'l2', 'l3', 'l4', 'l5', 'l6', 'l7', 'l8', '支持华为', '支持小米']

#### 列表[]（删）

##### 列表dex函数 根据下标删除元素

# 列表内只要有 'l5'或者 'l7' 就删除  
names = ['l1', 'l2', 'l3', 'l4', 'l5', 'l6', 'l7', 'l8', '支持华为', 'l10']  
l = len(names)  
i = 0  
while i < l:  
 if 'l5' in names[i] or 'l7' in names[i]:  
 del names[i]  
 l -= 1  
 i += 1  
print(names)

-------------------删除------------------------

['l1', 'l2', 'l3', 'l4', 'l6', 'l8', '支持华为', 'l10']

#### 列表[]数据类型转换

a="klklk"  
a=list(a)  
print(a)

['k', 'l', 'k', 'l', 'k']

b=[11,12,13,14]

c=tuple(b)

print(c)

(11, 12, 13, 14)

#### 列表切片

1. 和字符串一致
2. names = ['l1', 'l2', 'l3', 'l4', 'l5', 'l6', 'l7', 'l8', '支持华为', 'l10']

print(names[::2])#步长为2切片  
print(names[-5:-1:2])#-5 到-1 步长为2  
print(names[-1::-1])#反向切片  
print(names[-1::-2])#反向切片 步长为2

['l1', 'l3', 'l5', 'l7', '支持华为']

['l6', 'l8']

['l10', '支持华为', 'l8', 'l7', 'l6', 'l5', 'l4', 'l3', 'l2', 'l1']

['l10', 'l8', 'l6', 'l4', 'l2']

#### 列表求最大数

#求列表的最大值  
list=[1,9,3,8,28,3,6,19,4,3,1,0]  
big=0  
*for* i *in* list:  
 *if* i>=big:  
 big=i  
print(big)  
print(max(list))

28

28

#### 列表求最小数

#求列表的最小值  
list=[1,9,3,8,28,3,6,19,4,3,1,0]  
big=9  
*for* i *in* list:  
 *if* i<big:  
 big=i  
print(big)  
print(min(list))

#### 列表更新和迭代

l=[5,8]\*3  
print(l)

[5, 8, 5, 8, 5, 8]

a=3 *in*[1,2,3,5,6]  
print(a)

True

b=[[1,2,9],[3,2,1],[4,5]]  
print(len(b[0]))

print(b[1][3])

3

8

d=[3,2]*in* [1,2,[3,2,1],4,5]  
print(d)

False

#### 列表元素去重复

1.使用集合，集合元素去重

a=[1,2,1,3,13,12,3,4,5,1,2,4,53,2,12,3]

print(set(a))

[1, 2, 3, 13, 12, 4, 5, 53]

2.not in 便利到另一个容器

quchong=*[]*for i in a:  
 if i not in quchong:  
 quchong.append*(*i*)*print*(*quchong*)*

[1, 2, 3, 13, 12, 4, 5, 53]

### Tuple

#元组

#类似列表（当成容器）

# 特点：

# 1.定义符号：（）

# 2.和列表大的区别：元组中的内容不可修改，不可添加 ，只能查看

# 3.关键字：tuple‘

#内置1函数 index，count

ti=() #空元组  
print(type(ti)) #输出元组的类型  
  
  
t2=(5) #元组内一个元素 不加逗号  
print(type(t2)) #输出的事int类型 <class 'int'>，如果元组内的元素就一个且不加逗号 和没括号一样  
  
  
t3=(5,6) #元组内两个元素，输出类型  
print(type(t3)) #<class 'tuple'> ,输出的是元组

#### 元组查找函数 index，索引出元素首次出现的下标

#元组查元素 函数 index  
cc=*(*1, 2, 3, 4, '王五', 5, 6, 1, 2, 3, 4, 2, 2, 3, 4, 5, 2, 4, 5, 6, 7, 8*)*#查找3元素 从第7位开始查找1  
print*(*cc.index*(*3,7*))*

### dict

定义：Python字典是一种可变容器模型，且可存储任意类型对象，如字符串、数字、元组等其他容器模型。

字典的每个键值key=>value对用冒号 : 分割，每个对之间用逗号,分割，整个字典包括在花括号{}中 ,格式如下所示：

常用get查找

#### dict内置函数

增加：dict={} 修改 dict{key}=value

删除：dict,Pop , dict.popitem() dict.clear

查：items（） values（） keys（）

a =*{}* #定义空列表  
print*(*type*(*a*))* #输出字典类 dict <class 'dict'>

#### 字典增加元素

a*[*'手机品牌'*]*='华为'  
a*[*'价格'*]*='999'  
a*[*'型号'*]*='mate30'  
a*[*'颜色'*]*='red'  
print*(*a*)*#{'手机品牌': '华为', '价格': '999', '型号': 'mate30', '颜色': 'red'}

#### For循环便利字典,只输出k值

#尝试便利字典 ，便利的结果只有key  
dict1=*{*'石头':99,'月月':88,'小王':22,'小红':78*}*for i in dict1:  
 print*(*i*)*# 石头  
# 月月  
# 小王  
# 小红

#### items() # 字典转元组

#函数items 字典转元组  
dict1=*{*'石头':99,'月月':88,'小王':22,'小红':78*}*print*(*dict1.items*())*#dict\_items([('石头', 99), ('月月', 88), ('小王', 22), ('小红', 78)])

#通过items 转元组 便利，通过元组的下标值做判断  
for n in dict1.items*()*:  
 print*(*n*)* if n*[*1*]*>80:  
 print*(*n*[*1*])*# ('石头', 99)  
# 99  
# ('月月', 88)  
# 88  
# ('小王', 22)  
# ('小红', 78)

#### items()转元组后 拆包(字典)，赋值给变量

dict1=*{*'石头':99,'月月':88,'小王':22,'小红':78*}*for a,b in dict1.items*()*:  
 print*(*a,b*)*# 石头 99  
# 月月 88  
# 小王 22  
# 小红 78

#### 函数values

转元组包列表，只输出values值

#函数values  
dict1=*{*'石头':99,'月月':88,'小王':22,'小红':78*}*# 输出字典的values  
print*(*dict1.values*())*# dict\_values([99, 88, 22, 78])

#### For循环 用values便利 值

#便利出 字典的vaules  
for l in dict1.values*()*:  
 print*(*l*)*# 99  
# 88  
# 22  
# 78

#### keys（for便利key）

dict1=*{*'石头':99,'月月':88,'小王':22,'小红':78*}*for ll in dict1.keys*()*:  
 print*(*ll*)*

#### 列表判断kye是否存在 用in

#找人，找key 用in  
dict1=*{*'石头':99,'月月':88,'小王':22,'小红':78*}*print*(*'小红' in dict1*)*#True

#### 常用get函数，根据key查找值，如果key不存在就返回None

可以设定默认值，找不到就返回默认值

dict1=*{*'石头':99,'月月':88,'小王':22,'小红':78*}*print*(*dict1.get*(*'我'*))* # None  
print*(*dict1.get*(*'我','默认值'*))* #默认值  
print*(*dict1.get*(*'石头'*))* # 99  
# None  
# 99

#### 函数dict.fromkeys 将数据类型转成字典,添加默认值

jj='啊'  
ff=*[*1,2,3,4,5,6,7,8*]*p=dict.fromkeys*(*jj,'默认值'*)*b=dict.fromkeys*(*ff,'默认值'*)*print*(*p*)*print*(*b*)*# {'啊': '默认值'}  
# {1: '默认值', 2: '默认值', 3: '默认值', 4: '默认值', 5: '默认值', 6: '默认值', 7: '默认值', 8: '默认值'}

#### 字典删除元素

##### Del dict1[keyi]

dict1=*{*'石头':99,'月月':88,'小王':22,'小红':78*}*del dict1*[*'石头'*]*print*(*dict1*)*#{'月月': 88, '小王': 22, '小红': 78}

##### pop 根据key 删除键值对,删除成功返回 键值对的值

#字典删除元素 函数pop 根据key 删除键值对， 删除成功返回 键值对的值 ，  
dict1=*{*'石头':99,'月月':88,'小王':22,'小红':78*}*print*(*dict1.pop*(*'月月','找不到要删除的key，默认值'*))* #找到返回 值  
print*(*dict1.pop*(*'是','找不到要删除的key，默认值'*))* #找不到返回默认值  
# 99  
# 88  
# 找不到要删除的key，默认值

##### popitem() #删除最后一建值对 做返回

#删除字典末尾的键值对 函数 popitem()  
dict1=*{*'石头':99,'月月':88,'小王':22,'小红':78*}*print*(*dict1.popitem*())* #删除最后一个值 做返回  
#('小红', 78)

##### clear 清空字数据

# 函数 clear 清空字数据  
dict1=*{*'石头':99,'月月':88,'小王':22,'小红':78*}*print*(*dict1.clear*())*print*(*dict1*)*#{}

### Set

关键字：set

作用：去重复

符号：- & |

#### Set置函数：

增加：add（） update（）

删除：remove（） discart（） pop（） clear（）

#### 集合特性

Python中的集合，是一个无序的、没有重复项的集。它支持数学概念上的集合操作，如交集、并集、补集和差集。集合是可变的，可以在其中添加或删除项。

# 集合用花括号“{}”括起来，并用逗号“,”来分隔其中的项#set（） 创建空集合，只能用set（）

#### 创建集合

#可以使用花括号“{}”创建集合，前提里面必须有参数

s1=set()  
s2={1,2,3,4}  
print(type(s1))  
print(type(s2))

#### 集合列表快速去重

#集合列表快速去重  
list1=[1,2,3,1,2,31,2,4,56,5,4,3,5]  
print(set(list1))  
#{1, 2, 3, 4, 5, 56, 31}

#### 集合增加函数add

#集合增加函数add  
s1=set*()*s1.add*(*'临华'*)*s1.add*(*'开嗓'*)*print*(*s1*)*#{'临华', '开嗓'}

#### 集合删除元素

#### Remove（）存在就删，不存在报key error

#remove如果元素存在则删除，如果不存在就报错key error  
jihe=*{*1,2,3,45,5,6,7,8*}*print*(*jihe.remove*(*2*))*print*(*jihe*)* #{1, 3, 5, 6, 7, 8, 45}  
#print(jihe.remove(999)) #KeyError: 999

#### pop() 一般会从第一个删除,且返回被删除元素

#pop() 一般会从第一个删除,且返回被删除元素  
jihe=*{*1,2,3,45,5,6,7,8*}*fanhui=jihe.pop*()*print*(*jihe*)*print*(*fanhui*)*#{2, 3, 5, 6, 7, 8, 45}

#### clear 清除元素

#clear 清除元素  
jihe=*{*1,2,3,45,5,6,7,8*}*jihe.clear*()*print*(*jihe*)*#set()

#### discard()和remove差不多，删除元素 如果不存在不会报错

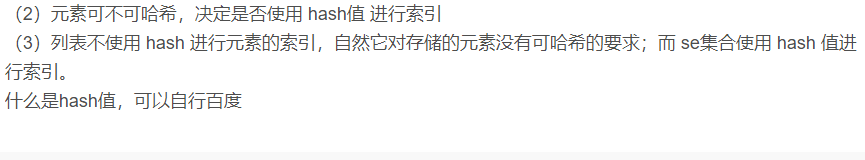
#discard()和remove差不多，删除元素 如果不存在不会报错  
jihe=*{*1,2,3,45,5,6,7,8*}*print*(*jihe.discard*(*12*)*,'不存在输出默认值'*)*

#### 集合之 差集 “-”， 交集“&”， 并集“|”

#集合之 差集 “-”， 交集“&”， 并集“|”  
st1=*{*1,2,3,4*}*st2=*{*1,2,3,4,5,6,7,8*}*print*(*st1-st2*)* #set()  
print*(*st2-st1*)* #{8, 5, 6, 7}  
print*(*st1&st2*)* #{1, 2, 3, 4}  
print*(*st1|st2*)* #{1, 2, 3, 4, 5, 6, 7, 8}

#### Updeat更新集合元素





### Str，list，tuple，dict，set类型转换

str()转 int() list() dict() tuble()

#正转  
# str转int  
a='8'  
print*(*int*(*a*))*# 8  
  
#str转列表  
b='abcd'  
print*(*list*(*b*))*# ['a', 'b', 'c', 'd']  
  
  
#str转集合  
b='abcd'  
print*(*set*(*b*))*# {'c', 'a', 'd', 'b'}  
  
  
#str转元组  
b='abcd'  
print*(*tuple*(*b*))*# ('a', 'b', 'c', 'd')

#元组转列表  
tuple1=*(*1,2,3,4,5*)*print*(*list*(*tuple1*))* #[1, 2, 3, 4, 5]  
  
  
#列表转元组  
list1=*[*1,2,3,4,5,6*]*print*(*tuple*(*list1*))* #(1, 2, 3, 4, 5, 6)  
  
  
#列表转集合  
print*(*set*(*list1*))*#字典转 列表 只保存key值  
dict1=*{*1:'lala',2:'didi',3:'wuwu'*}*print*(*list*(*dict1*))* #[1, 2, 3]

## 函数 def

# 函数：将重复的代码封装到函数里面，使用的时候直接找函数，提高代码的复用性 ，函s数可以增强代码的模块化和提高代码的重复利用率

# 定义函数：

#格式：

def 函数名（[参数，可有可无。。]:

函数体（重复的代码仍在def里了）

#注意1.必须使用关键字def 2.函数体注意缩进 3

# 定义函数，完成随机数的产生

# 带参数的函数  
*'''  
def 函数名（[]）  
 函数  
  
调用：  
 pass  
  
  
  
'''*# 求随机数的函数：产生的个数：  
import random  
  
  
def suijin(nuber): # 形参：形式上的参数  
 for i in range(nuber):  
 ti = random.randint(1, 100)  
 print(ti)  
  
  
# 查看分配内存  
print(suijin)  
  
# 调用  
suijin(5) # 实参：实际参数 具体的值

\

#### 作用域（全局和局部）

在函数中为变量赋值时，默认都是为局部变量赋值

**如果希望在函数内部改变全局变量，则需要使用global关键字声明 a变量为全局变6543**

Python中一红有两种作用域 ：（全局和局部）

**全局作用域的生命周期：全局作用域在程序执行时创建，在程序执行结束是销毁**

所有函数以外的区域都是全局作用域

在全局作用域中定义的变量，全都数据全都输入全局变量，全局变量可以在程序的任意位置访问

**函数作用域（局部）的生命周期：函数作用域在函数调用时创建，在调用结束时销毁**

函数每调用一次就会产生一个新的函数作用域，创建了三个作用域 因为调用了3次

Func()

Func()

Func()

在函数作用域中定义的变量都是局部变量，它只能在函数内部被访问

作用域变量的查找规则（）

当我们使用变量时，会优先在当前作用域寻找该变量，如果有则使用，

如果没有则继续往上一级作用域寻找，以此类推，直到找到全局作用域如果没有就抛出异常

#### 传入列表返回列表

#函数 将列表当作参数放入函数中，返回新列表  
list1=*[*1,2,3,4,5,6,7,8,9,10*]*def aa*(*list1*)*:  
  
 list2=*[]* for i in list1:  
 if i % 2==0:  
 list2.append*(*i*)* return list2  
  
  
bb=aa*(*list1*)*print*(*bb*)* #[2, 4, 6, 8, 10]

#### 判断列表内的最小值

def max(list):  
 a = list[0]  
 for i in list:  
 if a < i:  
 a = i  
 else:  
 print(f'列表内最大值：{a}')  
  
  
list = [1, 4, 2, 4, 78, 5, 3, 23, 1]  
max(list)

列表内最大值：78

#### 判断是不是什么类型:isinstance(变量,类型关键字)

aa = type(list)  
print(type(list))  
print(isinstance(list, aa))

<class 'list'>

True

#### 可变参数def (name,\*agrs)

#可变参数 \*args，和不可变参数 字符串 一起用，不可变参数在实参中，要放前面

可变参数 \*args 是可有可无的 不填实参也不会报错

#定义方式  
#\*n \*是用元组吧实参装进来  
def add*(*\*arg*)*:  
 print*(*type*(*arg*)*,arg*)*add*()*#()  
add*(*1*)*#(1,)  
add*(*1,2*)*#(1, 2)  
add*(*1,2,3,4,5,6*)*#(1, 2, 3, 4, 5, 6)

#### 函数实参为 可变参数+不可变参数一起用

#定义方式  
#\*n \*是用元组吧实参装进来  
def add*(*\*arg*)*:  
 print*(*type*(*arg*)*,arg*)*add*()*#()  
add*(*1*)*#(1,)  
add*(*1,2*)*#(1, 2)  
add*(*1,2,3,4,5,6*)*#(1, 2, 3, 4, 5, 6)

#方法可变参数求 可变实参的和  
def add1*(*name,\*aaa*)*:  
 sum=0  
 for i in aaa:  
 sum+=i  
 print*(*sum*)*add1*(*'实参的累加和是',1,2,3,4*)*

#### 函数实参为 可变参数+默认值

#5赋值给a，b已有值 不予赋值  
def aa(a,b=10):  
 c=a+b  
 print(c)  
  
  
aa(5)  
  
  
print('\*'\*50)  
  
#2赋值给d，9赋值给e 原有值被覆盖  
def bb(d,e=111):  
 f=d+e  
 print(f)  
  
bb(2,9)

#### 通过 字典函数values（），用for 便利出每个values值 使用拆包（values）输出值

#通过 字典函数values（），用for 便利出每个values值 使用拆包（values）输出值  
dict1=*{*'001':*(*'a',11*)*,'002':*(*'b',22*)*,'003':*(*'c',33*)*,'004':*(*'d',44*)*,'005':*(*'e',55*)}*aaa=dict1.values*()*print*(*dict1.values*())*for ii,ee in aaa:  
 print*(*"{}的年龄是{}".format*(*ii,ee*))*# a 11  
# b 22  
# c 33  
# d 44  
# e 55

#### 可变参数实参为列表，\*list 拆列表

#可变参数实参为列表，\*list 拆列表  
list2=*[*1,2,3,4,5,6*]*def list22*(*sun,\*ag*)*:  
 sum1=0  
 for i in ag:  
 sum1+=i  
 print*(*sun,sum1*)*#列表的累加和是 21  
  
list22*(*'列表的累加和是',\*list2*)*#列表的累加和是 21

#### 可变参数实参为字典，\*\*dict 拆列表dict

#可变参数实参为字典，\*\*dict 拆列表dict  
zidian=*{*'001':*(*'a',11*)*,'002':*(*'b',22*)*,'003':*(*'c',33*)*,'004':*(*'d',44*)*,'005':*(*'e',55*)}*def zid*(*va,\*\*aag*)*: #\*\*aag 装包，把拆掉的字典装进来  
 print*(*va,aag.values*())*zid*(*'字典的values是',\*\*zidian*)* #\*\*zidian 拆包的过程，

面向对象

类：

类名首字母必须大写，如果多个要用驼峰

类可以为空类，类的普通方法只能通过对象来调用

类属性就是属于类所有，可以直接用类名.属性名直接调用，类的属性在内存中只有一份。实例属性就是在\_\_init\_\_()方法中初始化的属性；

创建对象后 对象引用属性会优先引用自己内存里的属性，没有的话引用类属性 分代码上下

可以修改类属性 phone.color='白色',要用类名去更改，更改后 所有对象调用的类初始属性为改后的#面向对象

# #定义类  
# # class 类名:  
# # 特征：属性  
# # 动作：方法

#使用类，创建对象，加括号调用类 不加不调用  
aa=phone*()*

#创建对象 产生空间，且调用类  
a=phone*()*#类体  
print*(*phone.color,'类.color地址：',id*(*phone.color*))*# 打印类属性地址：白色 类.color地址： 2375195613968  
  
  
print*(*a,'打印aaaaaaa'*)*#打印对象属性：<\_\_main\_\_.phone object at 0x0000022904D37FD0> 打印aaaaaaa  
print*(*a.color,'a.color地址：',id*(*a.color*))*#打印 对象引用的类属性，对象属性与类属性地址相同：白色 b.color地址: 2375195613968，  
  
a.color='金色'#在对象的空间里添加color属性，对象属性使用color 时会优先取自己内存中的color  
print*(*a.color,'a.color地址：',id*(*a.color*))*#金色 a.color地址： 2375195614736，修改对象的属性color 地址产生新空间  
a.color='红色'  
print*(*a.color,'a.color地址：',id*(*a.color*))*#金色 a.color地址： 2375195614736，修改对象的属性color 地址产生新空间

#### \_\_init\_\_方法

\_\_Init\_\_方法是类里面的魔术方法，（通常用来给对象属性初始化）调用类的时候会自动执行魔术方法

当调用对象时的步骤

* + - 1. 对象会找有没有引用的这个类，如果有↓
      2. 对象会利用类来申请一模一样的空间(地址不同)，申请的时候会先去类里面找有没有\_\_init\_\_方法，如果有则把对象本身传给 self，进入\_\_init\_\_方法执行动作后 将内存赋值给对象（一般用来初始化类属性）

class ren*()*:  
 def \_\_init\_\_*(*self*)*:  
 self.sex='女' #不要忘了self.  
 name='月月'  
 age='23'  
 stature='160'  
  
  
ren2=ren*()*#对象创建后 类里面的元素不会copy过来，需要重新定义  
print*(*id*(*ren.name*))*#2237494367120  
print*(*id*(*ren2.name*))*#2237494367120  
print*(*ren2.sex*)*

#### \_\_init\_\_方法带参

#类里面的方法只能通过对象调用

#**init 可以加传参，实参填在创建对象的括号里 ，参数优先取 对象本身的 b=a(init参数)**

#**类里面的普通方法也可以带参，实参填在调用对象的括号**

**b.add（普通方法参数）**

#类里面的方法只能通过对象调用  
#init 可以加传参，实参填在创建对象的括号里 ，参数优先取 对象本身的输出  
#类里面的普通方法也可以带参，实参填在调用对象的括号里  
  
class aa*()*:  
 def \_\_init\_\_*(*self,zname,zage*)*:  
 self.name=zname  
 self.age=zage  
 def t1*(*self,sex*)*:  
 print*(*self.name*)* print*(*self.age*)* print*(*sex*)*b=aa*(*'b小红',18*)*b.name='大红'  
  
  
  
  
c=aa*(*'c小名',22*)*b.t1*(*'20'*)*#大红，18，20  
c.t1*(*'99'*)*#c小名，22，99

#### 参数私有化（\_\_name）

**Python中用\_\_两个下划线开头，声明该属性为私有，不能在类地外部被使用或直接访问**

**使用\_\_下划线表示私有属性，对象不能直接调用，要通过方法调调用**

#### 私有方法的使用（def \_\_func()）

class Person5:  
 def \_\_p*(*self*)*:  
 print*(*"这是私有属性"*)* # 内部函数也同样可以任意之间互相调用  
  
  
 def p1*(*self*)*:  
 print*(*"这是p1不是私有方法"*)* def p2*(*self*)*:  
 print*(*"这是p2,可以调用p1,也可以调用私有方法\_\_p"*)* self.p1*()* self.\_\_p*()*# 创建对象  
c1 = Person5*()*c1.p1*()*c1.p2*()*# c1.\_\_p() #不能直接私有函数。报错。注意区分系统自带的函数如\_\_str\_\_，外部可以直接调用的。  
  
'''结果如下：  
这是p1不是私有方法  
这是p2,可以调用p1,也可以调用私有方法\_\_p  
这是p1不是私有方法  
这是私有属性  
'''

#### 类方法@classmethod

类方法的使用：

是类所拥有的方法，需要用修饰器@classmethod来标识其为类方法，对于类方法，第一个参数必须是类对象，一般以cls作为第一个参数，也可以有别的参数。但是第一个必须是类对象，类似类中的def定义的普通方法第一个参数要是self一样的道理

class People*(*object*)*:  
 country = 'china'  
  
 # 类方法，用classmethod来进行修饰,跟普通的方法区别就是可以直接通过类名.方法名的方式调用  
 @classmethod  
 def getCountry*(*cls*)*:  
 return cls.country  
  
 @classmethod  
 def sum*(*cls, a, b*)*:  
 return a + b  
  
  
p = People*()*print*(*p.getCountry*())* # 可以用过实例对象引用  
print*(*People.getCountry*())* # 可以通过类名.方法名的形式调用  
print*(*p.sum*(*10, 11*))*print*(*People.sum*(*10, 11*))*

#输出私有话参数，通过类方法打印 调用  
class ren*()*:  
 \_\_leican=11  
 def \_\_init\_\_*(*self*)*:  
 pass  
  
 @classmethod  
 def leifa*(*cls*)*:#cls 把类本身传到cls里  
 pass  
 cls.\_\_leican=100#修改私有参数  
  
 @classmethod  
 def chacan*(*cls*)*:  
 print*(*cls.\_\_leican*)*#打印私有参数  
  
#静态方法只能访问类的属性和方法，对象的是无法访问的  
#需要装饰器 @classmethod  
#无需self 和cls方法  
#加载时机同类方法w2  
  
 @staticmethod  
 def test*()*:  
 print*(*'静态方法'*)* print*(*ren.\_\_leican*)*a=ren*()*b=ren*()*ren.leifa*()*ren.test*()*#100  
ren.chacan*()*#100

#### 静态方法

需要通过修饰器@staticmethod来进行修饰，静态方法不需要多定义参数

class People*(*object*)*:  
 country = 'china'  
  
 @staticmethod  
 # 静态方法,不用定义参数  
 def getCountry*()*:  
 return People.country  
  
  
print*(*People.getCountry*())*

### 继承

class Person:  
 def \_\_init\_\_*(*self,name,age*)*:  
 self.name=name  
 self.age=age  
  
 def eat*(*self*)*:  
 print*(*self.name+'在吃饭'*)* def running*(*self*)*:  
 print*(*self.name+'正在跑步'*)*item=*[*'1','2','3','4'*]*class Student*(*Person*)*:  
 def \_\_init\_\_*(*self,name,age,clazz,item*)*: #clazz是学生类独有的 ,item 可以传列表  
 super*()*.\_\_init\_\_*(*name,age*)*#super 相当于调用父类的 init方法  
 self.clazz=clazz #clazz是学生类独有的  
class Nurse:  
 pass  
  
  
class Police:  
 pass  
  
s=Student*(*'siyong',22,'一班',item*)*s.running*()*

## 异常处理

### 介绍

程序的错误有两种  
 1：编码错误 不符合 语言的语法

2：运行中出的错 ---异常

程序运行期间要避免异常

程序一旦出现异常就会终止运行

### 捕获异常语法

try：  
 可能出现异常的程序  
except：  
 出现异常后会执行的代码

如果try里面的代码没出现异常 那么execpt不会运行，只有try内的代码出现异常 才会运行

如果异常被try捕捉到 程序就不会被报错终止了

### 捕获不同类型的错误

try:  
 a=int*(*input*(*"请输入一个整数"*))* b=int*(*input*(*'请输入一个整数'*))* print*(*a/b*)*except ValueError:  
 print*(*"输入的非整数"*)*except ZeroDivisionError:  
 print*(*'输入的数不能为0'*)*except :  
 print*(*'未知错误'*)*

### 没有异常执行的代码

a = *int*(*input*("请输入一个整数"))  
 b = *int*(*input*("请输入第二个个整数"))  
 *print*(a/b)  
except:  
 *print*("输入错误 抛出异常")  
else:  
 *print*('没有发上异常执行的代码')

### 捕捉未知异常

把未知异常系统的错误提示给抛出来

try:

a = *int*(*input*("请输入一个整数"))

b = *int*(*input*("请输入第二个个整数"))

*print*(a/b)

except *Exception* as result: *#捕捉未知异常 把未知异常系统的错误提示给抛出来*

*print*(result)

### 无论是否异常都要执行的代码

try:

a = *int*(*input*("请输入一个整数"))

b = *int*(*input*("请输入第二个个整数"))

*print*(a/b)

except :

*print*('出现异常执行的代码')

finally:*#无论是否异常都要执行的代码*

*print*("无论是否异常都要执行的代码")

### 完整的捕捉异常方法

try:

a = *int*(*input*("请输入一个整数"))

b = *int*(*input*("请输入第二个个整数"))

*print*(a/b)

except *ValueError* :

*print*('输入格式错误')

except *Exception* as result:

*print*("捕捉未知异常 抛出")

else:

*print*("代码没有异常发上")

finally:*#无论是否异常都要执行的代码*

*print*("无论是否异常都要执行的代码")

### 主动抛出异常raise Exception("异常描述")

抛出异常后 只要未捕捉，下面代码无法执行，终止执行

*print*("开始")

raise *Exception*("异常描述")

*print*('结束')

开始

### 捕捉主动抛出的异常

try:

a = *int*(*input*("请输入一个整数"))

b = *int*(*input*("请输入第二个个整数"))

c=*input*("请输入运算符 + - \* 、")

if c !='+' and c != '-' and c !='\*' and c!='/':

raise *Exception*('请输入正确的加减乘除')

if a=='+':

*print*(a+b)

elif a=='-':

*print*(a-b)

elif a=='\*':

*print*(a\*b)

elif a=='/':

*print*(a/b)

except *Exception* as result:

*print*(result)

*print*(666)

请输入一个整数5

请输入第二个个整数6

请输入运算符 + - \* 、d

请输入正确的加减乘除

666

## Code

### 随机数列表/排序

#需求 生成100个 1到9999的随机数、并插入到列表a或者b里面且进行排序，如果生成的数为4为数就放到a，小于4位数就放置b  
*import* random  
big\_list=[]#四位随机数  
little\_list=[]#三位随机数  
b=1 #统计生成四位数的总数量  
*for* i *in* range(101): #循环100次  
 a = str(random.randint(1,9999)) #随机生成1到9999的int值  
 print("生成随机数第",i,"次") #第几次生成随机数  
 *if* len(a)==4:# 判断生成的值是否为4位数  
 b+=1 #统计生成四位数的总数量  
 print(a)  
 big\_list.append(a)#把值插入列表  
 *for* k *in* range(len(big\_list)):  
 *for* l *in* range(k+1,len(big\_list)):  
 *if* big\_list[k]>big\_list[l]:  
 m=big\_list[k]  
 big\_list[k]=big\_list[l]  
 big\_list[l]=m  
 *else*:  
 little\_list.append(a)  
 *if* i==100:  
 print("共生成随机数",i,"次","值>=1000的生成",b,"次")  
print("big\_list=",big\_list,)  
print("little\_list=",little\_list)

生成随机数第 100 次

4916

共生成随机数 100 次 值>=1000的生成 93 次

big\_list= ['1044', '1045', '1120', '1160', '1261', '1265', '1292', '1387', '1412', '1415', '1430', '1475', '1890', '1894', '2026', '2100', '2100', '2127', '2349', '2556', '2609', '2807', '2827', '3007', '3195', '3308', '3363', '3570', '3652', '3664', '3867', '3958', '4145', '4372', '4439', '4606', '4916', '4998', '5051', '5087', '5112', '5134', '5163', '5210', '5242', '5306', '5316', '5377', '5401', '5778', '6101', '6125', '6168', '6185', '6352', '6360', '6368', '6535', '6579', '6742', '6926', '6932', '6992', '7105', '7233', '7432', '7987', '8109', '8133', '8149', '8334', '8341', '8367', '8383', '8533', '8679', '8824', '8841', '8939', '9216', '9237', '9267', '9347', '9401', '9506', '9599', '9739', '9768', '9792', '9827', '9949', '9950']

little\_list= ['14', '457', '923', '316', '115', '324', '236', '879', '129']

### 1到4位组成 不重复3位数

for i in range*(*1,5*)*:  
 for j in range*(*1,5*)*:  
 for k in range*(*1,5*)*:  
 if i!=j and j!=k and i!=k:  
 print*(*i,j,k*)*

### 统计字符串内有几个数字

a='1aad21adasd'  
sum=0  
for i in a:  
 if i>="0" and i<="9":  
 sum+=1  
  
print*(*sum*)*

3

### JSON 操作符



### 正泽表达式

import re  
a='sadfghjklk;jkg`12343fdf@878#$3&\*(445q\_\_ABRFGHKL;\_\_'  
print*(*re.findall*(*'.',a*))* #匹配除“\n”之外的任何单个字符  
print*(*re.findall*(*'\d',a*))*# 匹配一个数字字符。等价于[0-9]。  
print*(*re.findall*(*'\D',a*))*# 匹配一个非数字字符。等价于[^0-9]。  
print*(*re.findall*(*'\w',a*))*# 匹配包括下划线的任何单词字符。等价于“[A-Za-z0-9\_]”  
print*(*re.findall*(*'*[*abcd*]*',a*))*print*(*re.findall*(*'*[*a**-**z*]*',a*))*print*(*re.findall*(*'*[*A**-**Z*]*',a*))*print*(*re.findall*(*'*[*A**-**Za**-**z1**-**9*]*',a*))*print*(*re.findall*(*'.*{*3*}*',a*))* #匹配所有字符 每三个  
print*(*re.findall*(*'\d*{*3*}*',a*))* #匹配 三个在一起的数字  
print*(*re.findall*(*'\d*{*3,5*}*',a*))* #匹配 三个 至 5个在一起的数字  
print*(*re.findall*(*'\d*{*2,5*}*?',a*))* #跟上问好 ？ 尽可能匹配2个在一起的数字  
print*(*re.findall*(*'\d*{*2,*}*',a*))* #跟上问好 ？ 尽可能匹配更长在一起的数字  
print*(*re.findall*(*'\d+',a*))* #匹配数字

### Whil 单次循环取多个值!

goods\_list = ["1000020", "1002735", "1000976"]  
color\_list = ["1976", "5406", "532"]  
size\_list = ["7494", "7496", "7491"]

a = 0  
while a < len(goods\_list):  
 goods\_id = goods\_list[a]  
 color\_id = color\_list[a]  
 size\_id = size\_list[a]

### random phone

def get\_phone*()*:  
 header\_list=*[*135,136,137,138,139,147,150,151,152,157,158,159,172,178,182,183,184,187,188,198*]* title=random.randint*(*0,len*(*header\_list*)*-1*)* header\_phone=str*(*header\_list*[*title*])* for \_ in range*(*0,8*)*:  
 header\_phone+=str*(*random.randint*(*0,9*))* return header\_phone

### rfind url切片

aa='https://123p3.sogoucdn.com/imgu/2018/04/20180404144123\_595.png'  
bb=aa.rfind('/')  
cc=aa[bb+1:]  
print(cc)

20180404144123\_595.png

## Fuction

### type() 判断数据类型

### isinstance()判断数据类型

>>>a = 2 >>> isinstance (a,int) True

>>> isinstance (a,str) False

>>> isinstance (a,(str,int,list)) # 是元组中的一个返回 True True

### eval()

把字符串取消转换成本质数据类型

### With Open

#### Read

with open*(*'file1.py',encoding="utf-8"*)* as file:  
 print*(*file.read*())*

#### write

#清空文件内所有内容 并写入数据 返回写入字符的lenth  
with open*(*'file1.py','w',encoding="utf-8"*)* as file:  
 print*(*file.write*(*"print('把文件清空然后写入一行数据')"*))*

#### ‘a‘/add

#在文件内追加数据 如果文件不存在就创建个文件  
with open*(*'file1.py','a',encoding="utf-8"*)* as file:  
 print*(*file.write*(*"\nprint('换行追加 在文件内追加数据')"*))*

#### readline

with open*(*'file1.py','r',encoding='utf-8'*)* as file:  
 for i in range*(*2*)*:  
 file\_r=file.readline*()* split\_file=file\_r.split*(*'@'*)*

### str

#### str.upper()

字符转大写

#### str.title()

字符首字母大写

#### str.account(‘str’,n,len(n))

统计字符出现次数 n~n

#### str.find(‘str’,1,len(str))

查找字符串下标 n~n ,找不到返回-1

#### str.index(‘str’,2,len(str))

查找字符串下标 n~n ,找不到后会异常

#### Str.rfind()

返回下表

#### Str. Replace()

字符替换

#### Startswith()

判断字符串以什么开头

#### Endswith()

判断字符串以什么结尾

#### join()

#### Format()

### Type()

### Id()

### Filter

用于过滤序列

第一个参数 是函数 ，第二个参数是可迭代对象

def fun(n):  
 return isinstance(n,str)  
  
  
list1 = [1, 2, 657, 78,'a','adssa']  
  
  
res = filter(fun, list1)  
print(res)  
print(list(res))

<filter object at 0x01B9F170>

['a', 'adssa']

### Map()

将可迭代对象中的 数据取出来 ，一个一个传入到函数中 调用，将返回结果 放到新的对象中

def func(n):  
 if isinstance(n, str):  
 return 1000  
 else:  
 return 9  
  
  
list2 = [1, 2, 3, 4, 'ad', 'dawd', False]  
response = map(func, list2)  
print(list(response))

[9, 9, 9, 9, 1000, 1000, 9]

## Module

### Jsonpath

import jsonpath  
bbb=jsonpath.jsonpath*(*ass,'$.data.token'*)*print*(*\*bbb*)*

### Random

#### randint

import random  
int1=random.randint(1,10000)

#### random.shuffle 将列表内的参数随机打乱

*import* random  
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
random.shuffle(list)  
print (list)

[1, 5, 6, 2, 8, 9, 10, 7, 3, 4]

#### random.choice 随机索引字符串或者列表内的参数

print (random.choice("Pythontab.com"))  
print (random.choice(["python", "tab", "com"]))  
print (random.choice(("python", "tab", "com")))

b

com

tab

phnme\_lsit=["139","138","137","136","135","134","159","158","157","150","151","152","188"]#创建手机号三位开头的列表

*for* i *in* range(1000):  
 print(random.choice(phnme\_lsit))

随机输出1000次列表内的值

#### random.sample(sequence, k)随机获取字符串中任意长度的

cc=''.join(random.sample(string.ascii\_letters,8))  
print(cc)

list1=[1,2,3,4,5,6,7]  
a1=random.sample(list1,2)  
print(a1) #[3, 6]

### String

#### String.ascii\_letters

aa=string.ascii\_letters  
print(aa)

abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ

### Re

#### Re.findall(re,str)

传入正则表达式 返回列表

c="商品号：SP10278Special SP1022Occasion Dresses SP Cabernet #5c2c35 1"  
# b=c.find('/')  
# print(b)  
# print(c[0:b])  
aa='"code": 0,"error": "ccc"'  
  
import re  
#模块  
print(re.findall('"code": (.\*)?', aa))  
  
#大写字母[A-Z],小写字母[a-z]  
print(re.findall("[A-Z]+[\d]+",c))  
#匹配所有数字 每个数字独立  
print(re.findall("[\d]",c))  
#匹配所有数字 集合+  
print(re.findall("[\d]+", c))  
  
#匹配所有字母 每个字母独立  
# print(re.findall("[a-z]",c))  
#匹配所有字母 集合+  
# print(re.findall("[a-z+]",c))  
  
#匹配字母+数字 字母和数字都独立  
print(re.findall("[a-z'\d]']",c))  
  
# print(re.findall("[A-Z]+[\d]+",c))

### Json

#### Json.Dumps（）

字典转str

#### Json.loads()

Str转字典

### Os

#### Os.path.abs()

返回文件绝对路径

#### Os.path.dirname()

返回当前文件所在的文件夹

#### Os.path.join(a,b)

拼接路径

#### Os.listdir(dir)

传入dir 返回路径下所有的文件 与文件夹名称 放入列表内

#### Os.path.exists()

判断路径是否存在

# Custom\_class

## 接口自动化小技巧

Excel 可以+1



## Json.loads 与eval区别

Josn 不会自运算 支持 null  
 eval 会自运算 不支持null

## Excel\_class

import os  
from openpyxl import load\_workbook  
xl\_path=os.path.join*(*os.path.dirname*(*os.path.abspath*(*\_\_file\_\_*))*,'login\_data.xlsx'*)*print*(*xl\_path*)*list\_data=*[]*#加载excle  
open\_exle=load\_workbook*(*'login\_data.xlsx'*)*#选择表单  
shooes\_from=open\_exle*[*'login'*]*#获取表头  
list\_titel=*[]*for list\_data\_titel in list*(*shooes\_from.rows*)[*0*]*:  
 list\_titel.append*(*list\_data\_titel.value*)*#取出用例参数  
for dict\_data in list*(*shooes\_from.rows*)[*1:*]*:  
 # print(dict\_data)  
 value\_dict=*{}* for index in range*(*len*(*dict\_data*))*:  
 # print(index,dict\_data[index],dict\_data[index].value)  
 value\_dict*[*list\_titel*[*index*]]*=dict\_data*[*index*]*.value  
 list\_data.append*(*value\_dict*)*print*(*list\_data*)*

## .Logging\_class

*"""  
@Project : new\_api  
@File : log  
@Author : Shi yong   
@Data : 2022/3/3  
  
二次封装的logger类 把log输出到控制台 和 文件里面  
s"""*import logging  
import my\_path  
  
  
class Log(logging.Logger):  
 def \_\_init\_\_(self, log\_name, level=logging.DEBUG, file=None):  
 *"""  
  
 log创建收集器 getLogger  
 设置收集级别 log.setLevel(logger.INFO)  
  
 设置日志格式 format=logger.Format 独立类  
 创建渠道 handle=logger.StreamHandler  
 渠道绑定格式 handle.setFormat(format)  
  
 收集器添加渠道 self.addHandle(handle)* ***:param*** *log\_name:log\_name* ***:param*** *level:日志级别* ***:param*** *file:文件路径  
 """* super().\_\_init\_\_(log\_name,level)  
 # 创建日志格式  
 fmt = '%(name)s:%(asctime)s:%(funcName)s:%(filename)s:%(lineno)d:%(levelname)s:%(message)s'  
 # 设置日志格式  
 formats = logging.Formatter(fmt)  
 # 创建console收集渠道  
 handle1 = logging.StreamHandler()  
 # 渠道绑定日志格式  
 handle1.setFormatter(formats)  
 # 渠道添加入收集器  
 self.addHandler(handle1)  
  
 if file:  
 file\_handle=logging.FileHandler(file,mode='a', encoding='utf-8')  
 file\_handle.setFormatter(formats)  
 self.addHandler(file\_handle)

### logging 调用使用

#登录case  
@data*(*\*login\_data.all\_data*())*def test01\_login\_success*(*self,case*)*:  
 utlis.new\_login.debug*(*'StartTest,Test\_Case{}:'.format*(*case*[*'case\_name'*])*, case*)* utlis.new\_login.debug*(*'TestData————》{}'.format*(*case*)*,case*)* case*[*'data'*]* =eval*(*case*[*'data'*])* responese=self.loginapi.login*(*self.session,case*[*'data'*])* self.assertEqual*(*case*[*'code'*]*,responese.json*()*.get*(*'code'*))* # self.assertEqual(msg, responese.json().get('msg'))  
 config.Token=responese.json*()*.get*(*'data.token'*)*

## Database

class Database:  
 #创建连结  
 def \_\_init\_\_*(*self,host,port,user,password,database,cursorclass='pymysql.cursors.DictCursor'*)*:  
 self.connect=pymysql.connect*(* host=host,  
 port=port,  
 user=user,  
 password=password,  
 database=database,  
 charset='utf8',  
 cursorclass=cursorclass  
 *)* #获取游标  
 self.cur=self.connect.cursor*()* def get\_fetchall*(*self,sq*)*:  
 self.cur.execute*(*sq*)* return self.cur.fetchall*()* def get\_fetchone*(*self,sq*)*:  
 self.cur.execute*(*sq*)* return self.cur.fetchone*()* def count\_line*(*self,sq*)*:  
 count=self.cur.execute*(*sq*)* return count

## File config

class Config\_file:  
 def \_\_init\_\_*(*self,file\_path*)*:  
 file\_path=os.path.join*(*config.conifg\_file\_dir,'Data\_base\_cf.ini'*)* self.new\_config\_parser=ConfigParser*()* self.red\_file=self.new\_config\_parser.read\_file*(*file\_path,'utf-8'*)* def get\_str*(*self,section,option*)*:  
 return self.new\_config\_parser.get*(*section,option*)* def get\_bool*(*self,section,option*)*:  
 return self.new\_config\_parser.getboolean*(*section,option*)* def get\_float*(*self,section,option*)*:  
 return self.new\_config\_parser.getfloat*(*section,option*)* def get\_int*(*self,section,option*)*:  
 return self.new\_config\_parser.getint*(*section,option*)*

## excel dynamic data

dict=*{*'a':'2',"b":"1","c":"5","d":"1","e":"4","f":1*}*def find\_data*(*case,newd,oldd*)*:  
 for key,value in case.items*()*:  
 if value is not None and isinstance*(*value,str*)*:  
 if value.find*(*oldd*)* !=-1:  
 case*[*key*]*=newd  
 return case  
  
  
dict=find\_data*(*dict,'new','1'*)*print*(*dict*)*

## Function

### type() 与 isinstance()区别：

class A:

pass

class B(A):

pass

isinstance(A(), A) # returns True

type(A()) == A # returns True

isinstance(B(), A) # returns True

type(B()) == A # returns False

### eval()

把字符串取消转换成本质数据类型

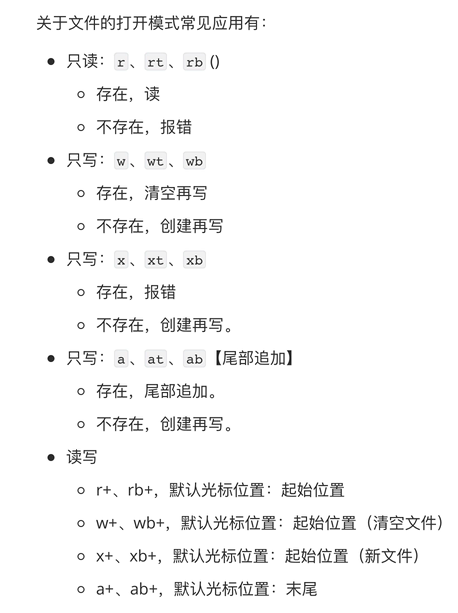
### Jsonpath 返回json value

import jsonpath  
bbb=jsonpath.jsonpath*(*ass,'$.data.token'*)*print*(*\*bbb*)*print*(*bbb*[*0*])*

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJhemF6aWUiLCJhdWQiOiJ1c2VycyIsImlhdCI6MTYzNTA2NzQ3OCwiZXhwIjoxNjM1NjcyMjc4LCJ1c2VyX2lkIjozODU2MDU4fQ.8\_5chIRbszq1813NAQ7j8uW4ASlPNx2IqEOIRx4ix30

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJhemF6aWUiLCJhdWQiOiJ1c2VycyIsImlhdCI6MTYzNTA2NzQ3OCwiZXhwIjoxNjM1NjcyMjc4LCJ1c2VyX2lkIjozODU2MDU4fQ.8\_5chIRbszq1813NAQ7j8uW4ASlPNx2IqEOIRx4ix30

### Open function



调整光标位置

file\_obj.seek(0)

#### Read

读取所有内容

# 读取文件所有内容  
with open('test\_file/222.txt', mode='rt', encoding='utf-8') as f1:  
 print(f1.read())

#### write

# q清空文件 写入一行文件  
with open('test\_file/111.txt', mode='wt', encoding='utf-8') as f:  
 f.write('shiuong')

#### Read /write pohoto or video

# 读取图片二进制编码，读的原始二进制 不存在编码  
with open('test\_file/az.png', mode='rb') as f2:  
 content = f2.read()  
 f2.close()  
# 写入读取到的 文件读的原始二进制 不存在编码  
with open('test\_file/az2.png', mode='wb') as f3:  
 f3.write(content)

#### 打开文件一次 循环写入/flush立即写入文件

# 多用户注册写入  
  
file\_obj = open('test\_file/info.txt', mode='wt', encoding='utf-8')  
while True:  
 name = input('enter name:')  
 if name.upper() == 'Q':  
 break  
 age = input('enter age:')  
  
 data = '{}:{}\n'.format(name, age)  
 # 并未理解写入文件 只是写到了缓冲区  
 file\_obj.write(data)  
 # 立即写入到文件  
 file\_obj.flush()  
file\_obj.close()

#### 末尾追加

file\_obj = open('test\_file/info.txt', mode='a+', encoding='utf-8')  
while True:  
 name = input('enter name:')  
 if name.upper() == 'Q':  
 break  
 age = input('enter age:')  
  
 data = '{}:{}\n'.format(name, age)  
file\_obj.close()

#### readline

读取一行数据

open\_obj = open('test\_file/info.txt', mode='r+', encoding='utf-8')  
print(open\_obj.readline())

#### 读取所有行数据 返回列表

open\_obj = open('test\_file/info.txt', mode='r+', encoding='utf-8')  
print(open\_obj.readlines())

['shiyong:18\n', 'shiyong2:20\n', 'shiyong3:23\n', '追加:1\n', '追加:2\n', '追加:3\n', '追加4:4\n', '追加:5\n']

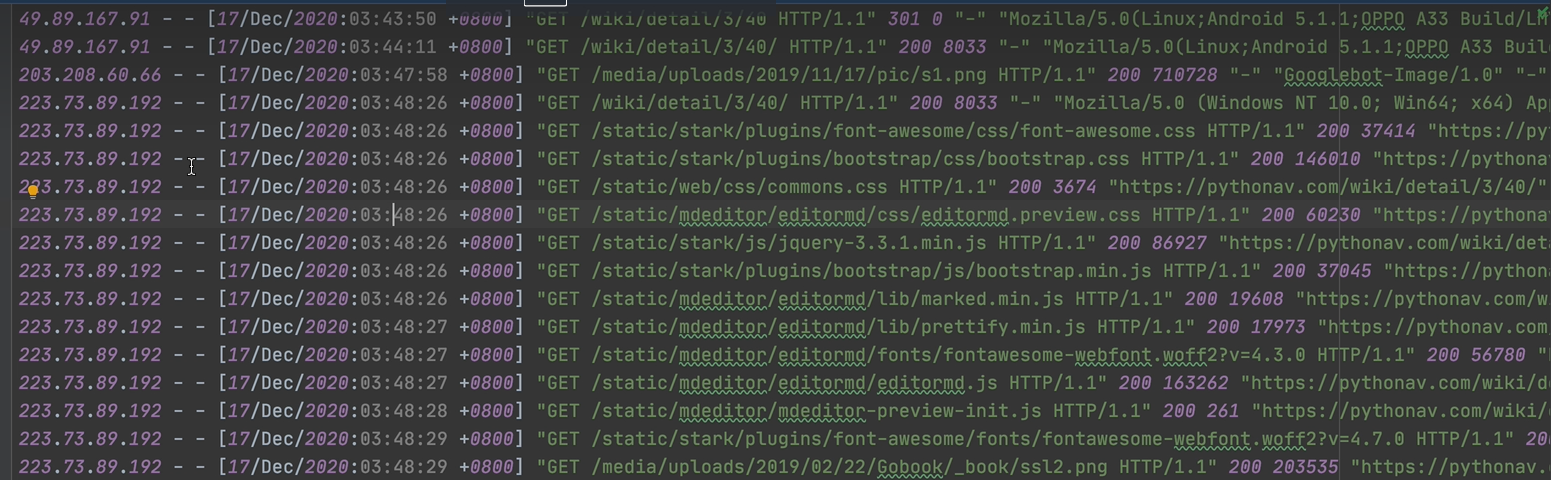
#### 未知行数 遍历所有行

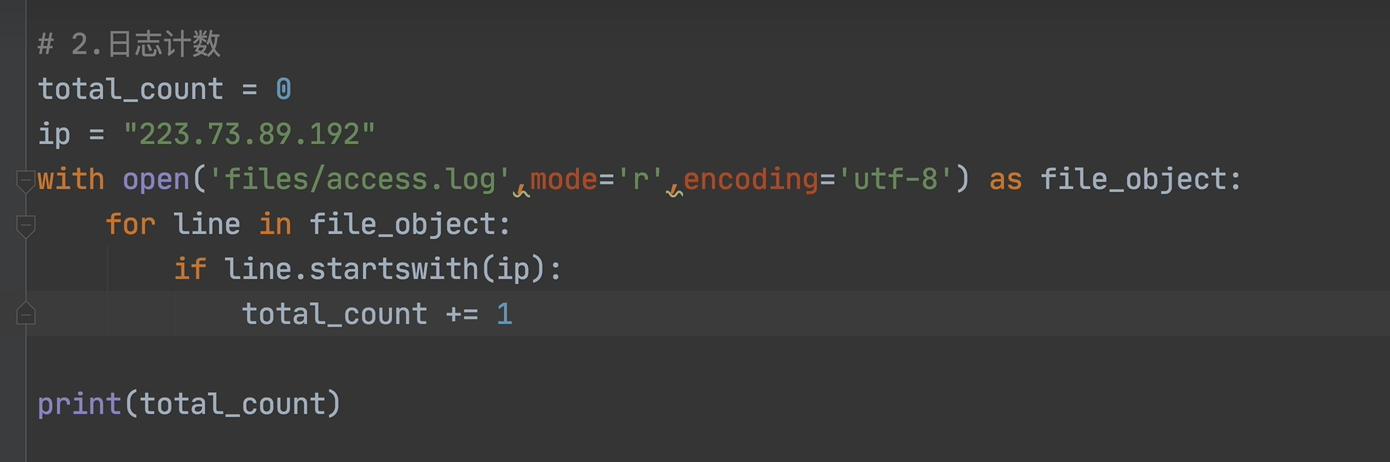
因为 print自动换行 所以用了strip（）

open\_obj = open('test\_file/info.txt', mode='r+', encoding='utf-8')  
for i in open\_obj:  
 print(open\_obj.readline().strip())

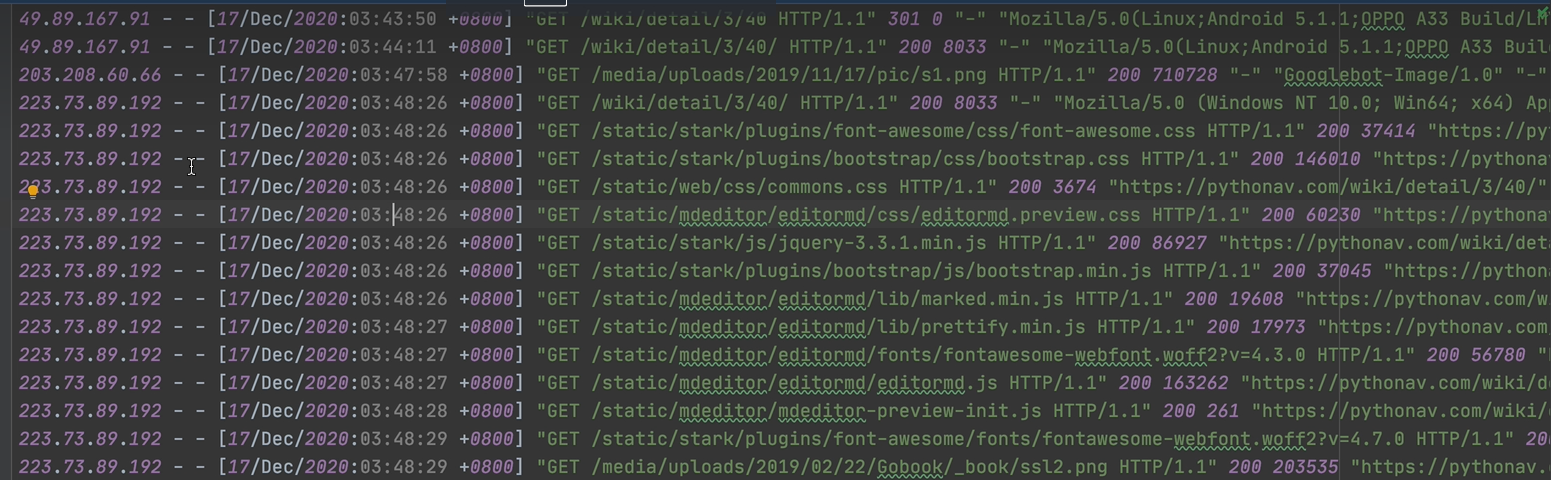
#### 日志计数

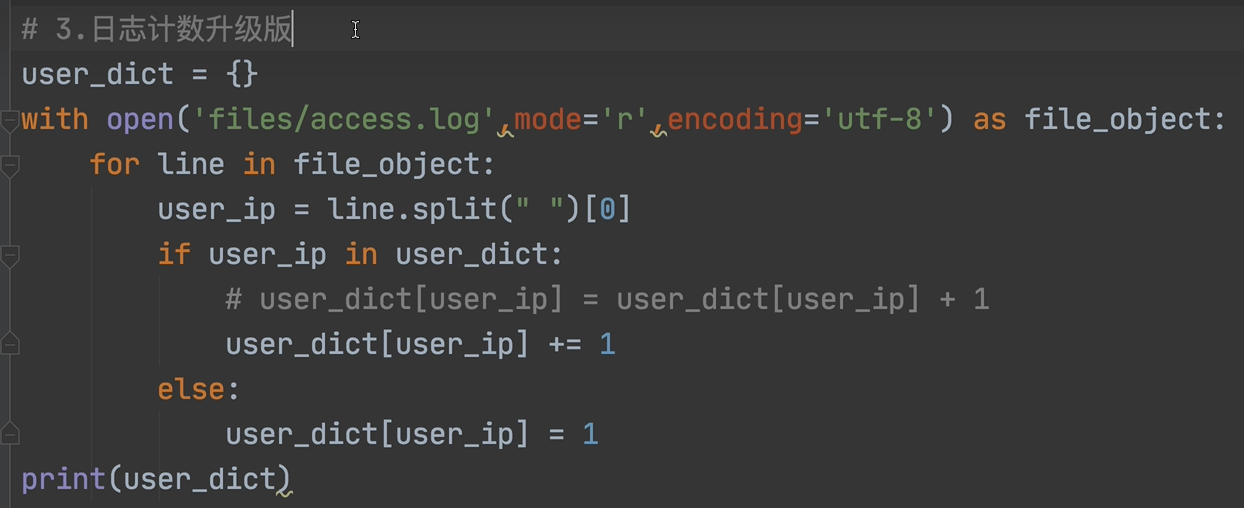
统计日志文件内 ip出现的次数



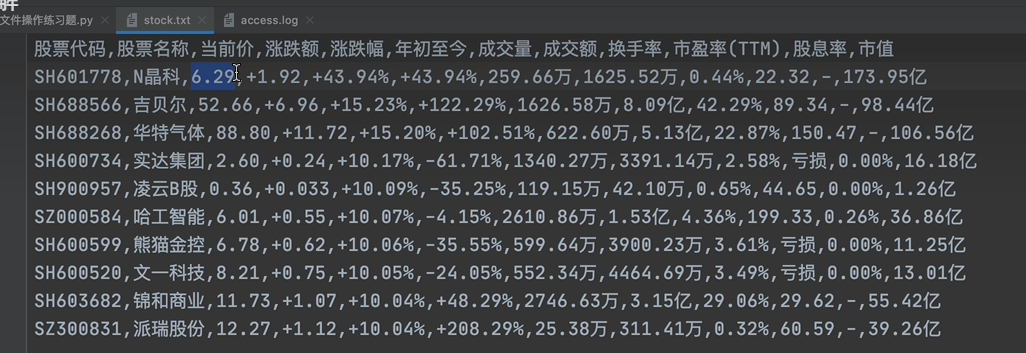


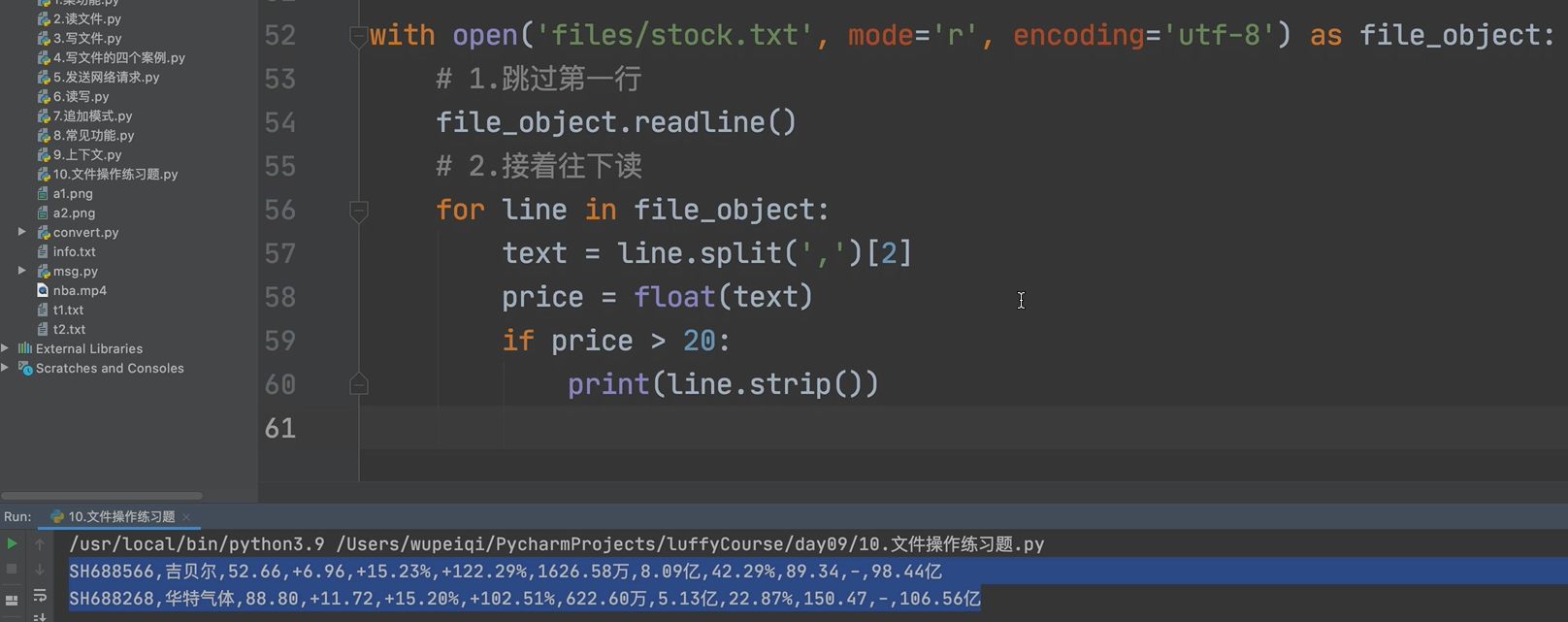
#### 统计日志中出现ip的次数



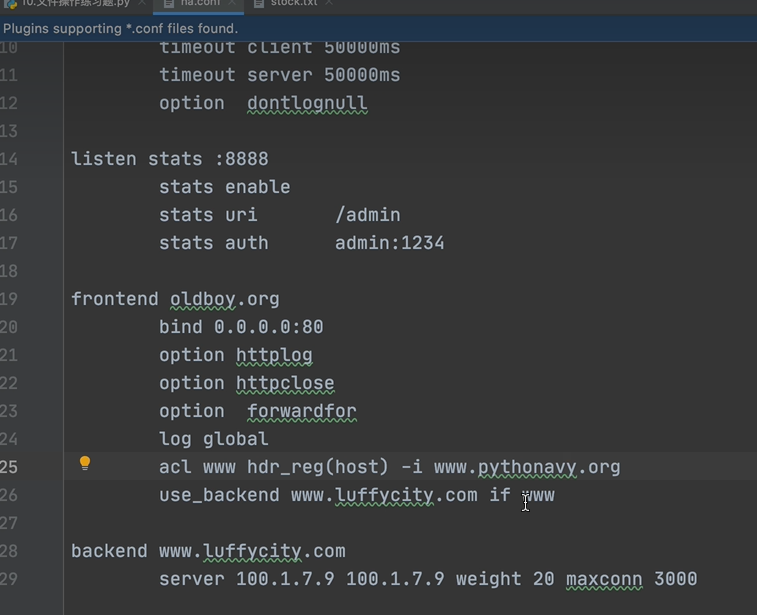


#### 统计出股票 当前价大于20的信息





读取文件内容 替换文件内容



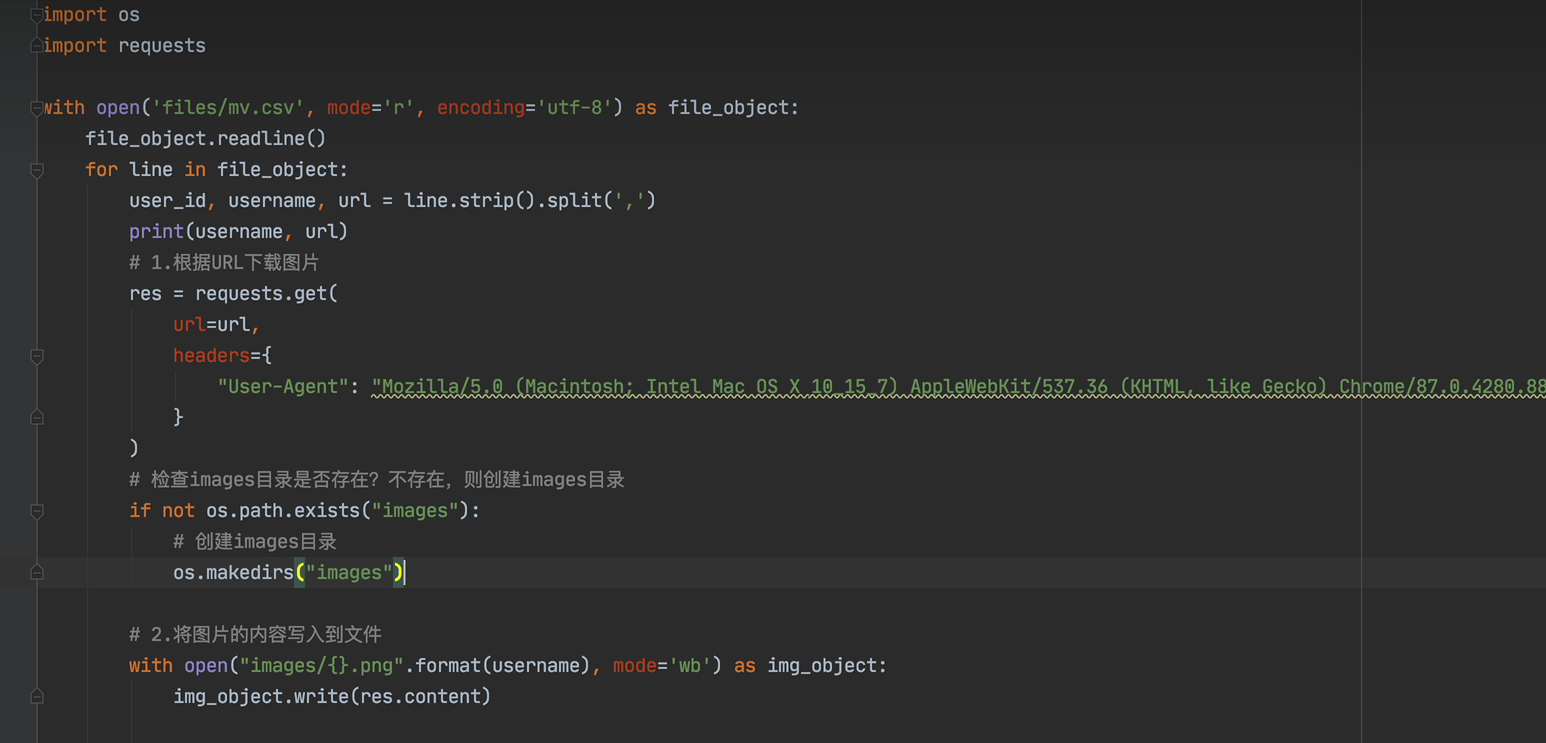


#### 重命名替换原文件



1

Csv文件读取



### string

#### str.upper() 字符转大写

#### str.title() 字符首字母大写

#### str.account(‘str’,n,len(n)) 统计字符出现次数 n~n

#### str.find(‘str’,1,len(str)) 查找字符串下标 n~n ,找不到返回-1

#### str.index(‘str’,2,len(str)) 查找字符串下标 n~n ,找不到后会异常

print(a.upper()) #upper 将字符串中的所有小写字符串转大写  
  
print(a.swapcase()) #将字符串 的大小写反转  
  
print(a.title()) #将字符串内的每个字符首字符大写  
  
print(a.capitalize()) # 将字符串的首字符大写 只能转一个  
  
print(a.count('h',1,len(a))) #统计 字符穿出现的次数，从 n 到n直接出现的次数  
  
print(a.find('wangl',1,len(a))) #查找字符串 并索引出下标，从n到n,如果找到返回下标，找不到返回-1  
  
print(a.index('wang',2,len(a))) #和find 一样 只不过找不到后会异常

#### rfind

从左边开始查找字符串 ：图片url的文件名

aa='https://123p3.sogoucdn.com/imgu/2018/04/20180404144123\_595.png'  
bb=aa.rfind('/')  
cc=aa[bb+1:]  
print(cc)

20180404144123\_595.png

#### Replace

替换字符串，

s1='shixiantao'  
s2=s1.replace('shixiantao','humingyuehaha')  
print(s2)

### Type（）

### Id()

### In not in

### Sorted()

aa=[{'name':'shi','age':3},{'name':'hu','age':7},{'name':'a','age':9}]  
a=sorted(aa,key=lambda i:i['age'],reverse=True)  
print(a)

[{'name': 'a', 'age': 9}, {'name': 'hu', 'age': 7}, {'name': 'shi', 'age': 3}]

## Module

### Random.randin(int,int)

#随机数  
import random  
t1=random.randint(1,100)  
print(t1)

### random.choice

phnme\_lsit=["139","138","137","136","135","134","159","158","157","150","151","152","188"]#创建手机号三位开头的列表

*for* i *in* range(1000):  
 print(random.choice(phnme\_lsit))

**random.shuffle**

将列表内的参数随机打乱

*import* random  
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
random.shuffle(list)  
print (list)

[1, 5, 6, 2, 8, 9, 10, 7, 3, 4]

**random.sample(sequence, k)**

随机获取字符串中任意长度的

cc=''.join(random.sample(string.ascii\_letters,8))  
print(cc)

# Outo case

### Add goods

加车各品类商品成功

加车促销商品成功

同一商品加车多个

未登录加车

加车失败 未添加goods

加车失败 未选择color

加车失败 未选择 size

Update cart

修改商品数量

修改商品color

修改商品siez

修改商品 save for later

修改商品 save for later 到catr

save for later 修改商品数量

未登录修改 商品信息

# Test\_develop

## Virtualenv

### 安装

#### 1.安装包

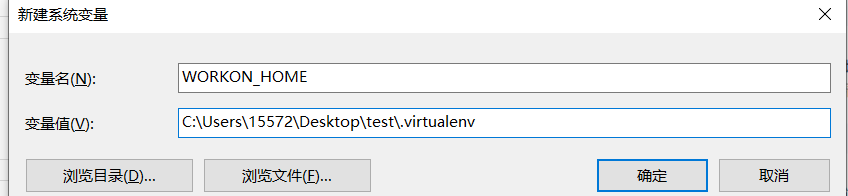
Pip install virtualen

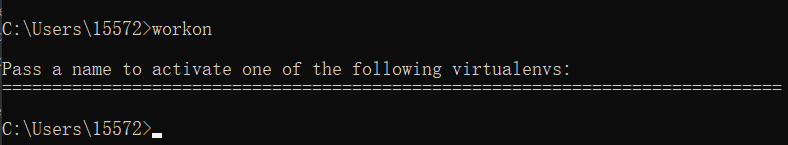
pip install virtualenvwrapper-win

#### 2.创建 虚拟环境家目录

创建文件夹 里面装虚拟环境

#### 3.配置虚拟环境家目录 环境变量





### Virtualenv 使用命令



退出虚拟环境 exit（）

指定python版本创建虚拟环境

····· mkvirtualenv -p "C:\shi\python\_3.7.9\python.exe" vir2



## Pipenv

1.安装

1.安装包 pip install pipenv

2. cd 到项目路径 pipenv --python 3.7 创建指定python版本虚拟环境

### Command

Pipenv install 恢复依赖包 前提项目中含pipfile文件

Pipenv graph 查看虚拟环境包列表

Pipenv –venv 查看当前虚拟环境

Pipenv --rm 删除虚拟环境

列出虚拟环境列表 workon

显示虚拟环境python解释器所在路径 pipenv --py

## 推导式

#### 普通列推导式

list1 = [i for i in range(10)]  
print(list1)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

#### If判断列表推导式

list2 = [i for i in range(10) if i % 2 != 0]  
print(list2)

[1, 3, 5, 7, 9]

#### 双层if 推导式

list3 = [(i, j \* 5) for i in 'abcdef' for j in range(5)]  
print(list3)

[('a', 0), ('a', 5), ('a', 10), ('a', 15), ('a', 20), ('b', 0), ('b', 5), ('b', 10), ('b', 15), ('b', 20), ('c', 0), ('c', 5), ('c', 10), ('c', 15), ('c', 20), ('d', 0), ('d', 5), ('d', 10), ('d', 15), ('d', 20), ('e', 0), ('e', 5), ('e', 10), ('e', 15), ('e', 20), ('f', 0), ('f', 5), ('f', 10), ('f', 15), ('f', 20)]

### 字典推导式

#### 推导生成字典

data ={"syt" + str(i):"123456" for i in range(20)}  
print(data)

#### '获取字典中的key是小写的键值对'

'获取字典中的key是小写的键值对'  
dict1 = {"a": 10, "B": 20, "C": True, "D": "hello world", "e": "python教程"}  
dict1 = {key: dict1.get(key) for key, value in dict1.items() if key.islower()}  
print(dict1)

{'a': 10, 'e': 'python教程'}

#### '将字典里的所有key设置为小写'

dict2 = {"a": 10, "B": 20, "C": True, "D": "hello world", "e": "python教程"}  
dict2 = {key.lower(): value for key, value in dict2.items()}  
print(dict2)

**{'a': 10, 'b': 20, 'c': True, 'd': 'hello world', 'e': 'python教程'}**

#### "将字典内 所有key为小写字母的 value赋值为error"

dict3 = {"a": 10, "B": 20, "C": True, "D": "hello world", "e": "python教程"}  
dict3 = {key: value if key.islower() else 'error' for key, value in dict3.items()}  
print(dict3)

{'a': 10, 'B': 'error', 'C': 'error', 'D': 'error', 'e': 'python教程'}

#### 将cookie 字符串修改成字典

'将cookie 字符串修改成字典'  
cookies = "anonymid=jy0ui55o-u6f6zd; depovince=GW; \_r01\_=1; JSESSIONID=abcMktGLRGjLtdhBk7OVw;" \  
 " ick\_login=a9b557b8-8138-4e9d-8601-de7b2a633f80; \_ga=GA1.2.1307141854.1562980962;" \  
 " \_gid=GA1.2.201589596.1562980962; \_c1=-100; first\_login\_flag=1; ln\_uact=18323008898;" \  
  
dict4 = {i.split('=')[0]: i.split('=')[1] for i in cookies.split(';')}  
print(dict4)

**{'anonymid': 'jy0ui55o-u6f6zd', ' depovince': 'GW', ' \_r01\_': '1', ' JSESSIONID': 'abcMktGLRGjLtdhBk7OVw', ' ick\_login': 'a9b557b8-8138-4e9d-8601-de7b2a633f80'}**

## 生成器

不占用内存 数据用一次取一次

### 推导式生成 生成器

a = (i for i in range(1, 101))  
print(next(a))  
print(next(a))

1

2

### Yield

方法内 Yield 返回生成器对象 调用一次方法取一次

# yield 返回生成器对象 每用next取一次值就返回下一个值 取完再取会报错  
def gen\_fun():  
 yield [i for i in range(10)]  
 print('hellow python')  
 yield 200  
 yield 300  
  
 # 返回生成器对象  
gen\_obj = gen\_fun()  
print(gen\_obj)  
print(next(gen\_obj))  
print(next(gen\_obj))  
print(next(gen\_obj))

<generator object gen\_fun at 0x01462A70>

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

hellow python

200

300

### 迭代器

# iter 把可迭代对象转换成 迭代器

#### 可迭代对象和迭代器的区别

'可迭代对象 内部实现了\_\_iter\_\_方法,迭代器内部除了iter方法外 还实现了next方法'

'生成器是迭代器的一种 ，生成器器 和迭代器都可进行迭代'

'生成器内部还实现了 send方法 可与生成器进行交互'

a=[ i for i in range(10)]  
print(a)  
b=iter(a)  
print(next(b))  
print(next(b))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

0

1

## 递归

函数自己调用自己

### 递归阶乘

def jiecheng(n):  
 if n==1:  
 return 1  
 else:  
 return n\*jiecheng(n-1)  
  
print(jiecheng(5))

120

### If 判断阶乘

sum=1  
def if\_jc(n):  
 global sum  
 for i in range(1,n+1):  
 sum\*=i  
 return sum  
  
print(if\_jc(5))

、

### 斐波那契数列

def fbn(c):  
 if c==1:  
 return 1  
 elif c==2:  
 return 2  
 else:  
 return fbn(c-1)+fbn(c-2)

## 纯函数

特点

传入参数相同 保证输出相同（变量都是从函数作用域内取）

无函数副作用 不会与作用域外部的变量 对内部参数发生修改 称之为纯函数

## Lambda

格式

值=（Lambda 参数:表达式）（入参）

res2 = lambda a, b: a + b  
res2(5, 6)  
print(res2(5, 6))

11

res3 = (lambda c, d: c \* d)(5, 6)  
print(res3)

30

list2 = [1, 6, 765, 54, 3, 67, 987, 3, 543, 657, 78, 99]  
list3 = filter((lambda n: n > 90), list2)  
print(list(list3))

[765, 987, 543, 657, 99]

## 偏函数

'''偏函数'''  
  
from functools import partial  
import random  
  
l1 = [1, 34, 456, 765, 889, 8, 43, 45, 54, 765, 35, 765, 2]  
l2 = [324, 4, 556, 677, 354]  
l3 = [23, 44, 6, 4, 35, 667, 56, 87, 98]  
new\_partial=partial(filter, lambda n: n > 100)  
l4=new\_partial(l1)  
print(list(l4))

[456, 765, 889, 765, 765]

## 闭包

### 闭包的三个必要条件

1.函数内部 嵌套函数

2.函数中 返回内层函数的名称

3.内层函数 引用了外层函数的变量

### 闭包的作用

实现数据锁定，提高稳定性

## 装饰器

装饰器就是闭包函数的应用，装饰器不用严格遵守闭包规则 ，可引用全局变量

装饰器的特点 ：在不修改原函数代码的情况下 扩展代码功能

装饰器逻辑：a函数扩展 b函数代码，b是个闭包函数，当a装饰好后 调用a函数

会先运行闭包函数把 原函数a传入b函数内，然后b函数返回 b内部函数 ，a接

收b返回的内部函数，此时a函数就是 b内部的函数，

### 无参装饰器

def check(fun):  
  
 def login():  
 username = 'shiyong'  
 password = '123456'  
 ac = input('input a account:')  
 pwd = input('input a password:')  
 if ac == username and pwd == password:  
 print('登录成功')  
 fun()  
 else:  
 print('密码 或 账号错误')  
  
 return login  
  
  
@check #add\_cat=login()  
def add\_cat():  
 print('这是一个网站')  
  
  
add\_cat()

input a account:shiyong

input a password:123456

登录成功

这是一个网站

### 有参装饰器

def add(func):  
 def fun(a,b):  
 print('计算 两个数相乘：a+b={}'.format(a+b))  
 func(a,b)  
 return fun  
  
  
  
@add  
def add\_number(a,b):  
 print('计算 两个数相加：a+b={}'.format(a+b))  
  
add\_number(5,9)

计算 两个数相乘：a+b=14

计算 两个数相加：a+b=14

### 可有可无参装饰器

def common\_func(func):  
 def fun(\*args, \*\*kwargs):  
 print('这是扩展的功能')  
 func(\*args, \*\*kwargs)  
  
 return fun  
  
  
@common\_func  
def goods\_list(number):  
 print('这是列表页第：{} 页'.format(number))  
  
  
@common\_func  
def home\_page():  
 print('这是网站首页')  
  
  
goods\_list(9)  
print('-------------这是两个函数的分割线----------------')  
home\_page()

这是扩展的功能

这是列表页第：9 页

-------------这是两个函数的分割线----------------

这是扩展的功能

这是网站首页

### 可以装饰类的装饰器

类装饰器必须要返回 对象

def common\_func(func):  
 def fun(\*args, \*\*kwargs):  
 print('这是扩展的功能')  
 return func(\*args, \*\*kwargs)  
  
 return fun  
  
  
@common\_func  
class Page:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
  
new\_page = Page('shiyong')  
print(new\_page)

这是扩展的功能

<\_\_main\_\_.Page object at 0x01E29EF0>

### 单例对象装饰器

def only\_object(cls):  
 object\_dict = {}  
  
 def fun(\*args, \*\*kwargs):  
 if cls in object\_dict:  
 return object\_dict[cls]  
 else:  
 object\_dict[cls] = cls(\*args, \*\*kwargs)  
 return object\_dict[cls]  
  
 return fun  
  
  
@only\_object  
class Room:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
  
@only\_object  
class Room2:  
 def \_\_init\_\_(self, age):  
 self.age = age  
  
  
t1 = Room('shiyong')  
t2 = Room()  
print(id(t1))  
print(id(t2))  
print(t2.name)  
t3 = Room2()  
t4 = Room2()  
print(id(t3))  
print(id(t4))

27917040

27917040

shiyong

27972752

27972752

18

### 类作为装饰器

class Call:  
 def \_\_init\_\_(self, func):  
 self.fun = func  
  
 def \_\_call\_\_(self, \*args, \*\*kwargs):  
 print('这是类装饰器里面的功能，饭前洗洗手')  
 self.fun()  
  
  
@Call  
def eat():  
 print('开始吃饭')  
  
  
eat()

这是类装饰器里面的功能，饭前洗洗手

开始吃饭

## 魔术方法

### \_\_\_\_New\_\_\_\_单例对象类

Python 内置魔术方法 ，创建对象时第一个调用的方法 会返回一个对象

可以在类内部重写 控制对象返回 实现 单例对象类

*"""  
  
对象唯一 任何时候创建对象都是第一个对象  
"""*class Room:  
 only\_object = None  
  
 def \_\_new\_\_(cls, \*args, \*\*kwargs):  
 if not cls.only\_object:  
 cls.only\_object = object.\_\_new\_\_(cls)  
 return cls.only\_object  
 else:  
 return cls.only\_object  
  
  
t1 = Room()  
t1.name = 'shiyong'  
t2 = Room()  
print(t1.name)  
print(t2.name)

shiyong

shiyong

### \_\_\_str\_\_\_,\_\_\_repr\_\_\_

*"""  
\_\_\_str\_\_方法  
\_\_\_repr\_\_方法  
  
  
print（）,str(),format()会调用\_\_\_str\_\_\_方法,找不到\_\_\_str\_\_\_方法会调用 \_\_\_repr\_\_\_方法，必须返回str类型值  
"""*class T1:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 def \_\_str\_\_(self):  
 print('调用了 \_\_\_str\_\_\_方法')  
 return self.name  
  
 def \_\_repr\_\_(self):  
 print('调用了 \_\_\_repr方法\_\_\_')  
 return 'aaaaa'  
  
  
new\_t1 = T1('shiyong')  
print(new\_t1)  
str(new\_t1)  
format(new\_t1)  
print(repr(new\_t1))

### \_\_\_call\_\_

*"""  
  
如果有call方法是可被调用  
"""*class Call:  
 def \_\_init\_\_(self):  
 print('运行了init方法')  
  
 def \_\_call\_\_(self):  
 print('调用了 类的call方法')  
  
  
t1 = Call()  
t1()

### \_\_dict\_\_ ,

类调用\_\_dict\_\_时可查看 类里面的所有属性，创建类时 每个类里面会自动添加\_\_dict\_\_属性，继承时 子类不会有\_\_dict\_\_属性

继承类时 子类不会有\_\_dict\_\_属性，因为子类里

class A:  
 pass  
  
  
class B(A):  
 pass  
  
  
a=A()  
print(A.\_\_dict\_\_)  
print(B.\_\_dict\_\_)  
print(a.\_\_dict\_\_)

{'\_\_module\_\_': '\_\_main\_\_', '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'A' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'A' objects>, '\_\_doc\_\_': None}

{'\_\_module\_\_': '\_\_main\_\_', '\_\_doc\_\_': None}

{}

### \_\_slots\_\_

限制类对象的属性 ，对象只能添加指定属性，节约内存 创建对象时所有对象属性指向\_\_slots\_\_内存

大量创建对象的时候使用 节约内存

class A:  
 \_\_slots\_\_ = ['name']  
   
  
a=A()  
a.name='shiyong'  
a.age='100'

### \_\_enter\_\_,\_\_exit\_\_实现数据库上下文管理

With 里面\_\_\_Enter\_\_\_方法 \_\_exit\_\_方法 whit语句运行完成后 自动运行\_\_exit\_\_方法

import pymysql  
  
  
class Db:  
 def \_\_init\_\_(self, data\_conf):  
 self.con = pymysql.connect(\*\*data\_conf)  
 self.cursor = self.con.cursor()  
  
 def \_\_enter\_\_(self):  
 return self.cursor  
  
 def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):  
 self.con.close()  
 self.cursor.close()  
  
  
db\_conf\_data = dict(  
 host='rm-uf60nj0t33i3601vx3o.mysql.rds.aliyuncs.com',  
 port=3306,  
 user='root',  
 password='shi1557225637\_',  
 database='table\_one',  
 charset='utf8'  
)  
  
sq = "SELECT \* FROM `users` WHERE name='shiyong'"  
with Db(db\_conf\_data) as db:  
 db.execute(sq)  
 print(db.fetchone())

(13, 'shiyong', 16628554195, '1557225637@qq.com', '123456', 7401.99)

### \_\_getattr\_\_,\_\_getattribute\_\_,setattr\_\_,\_\_delattr\_\_

class Attribute:  
  
 # def \_\_init\_\_(self):  
 # self.name = 'shiyong'  
  
 # 访问不到属性时 调用此方法  
 def \_\_getattr\_\_(self, item):  
 print('调用了 \_\_getattr\_\_ 方法')  
  
 # 查找属性时调用此方法  
 def \_\_getattribute\_\_(self, item):  
 print('调用了 \_\_getattribute\_\_ 方法')  
 return super().\_\_getattribute\_\_(item)  
  
 # 设置属性时 调用此方法  
 def \_\_setattr\_\_(self, key, value):  
 print('调用了 \_\_setattr\_\_ 方法')  
 if key == 'name':  
 object.\_\_setattr\_\_(self, key, 'shiyong')  
 else:  
 object.\_\_setattr\_\_(self, key, value)  
  
 # 删除属性时 调用了此方法  
 def \_\_delattr\_\_(self, item):  
 print(' 调用了 \_\_delattr\_\_ 方法')  
 if item == 'name':  
 pass  
  
  
attr = Attribute()  
attr.name='shiyong'  
attr.name='az'  
print(attr.name)  
  
  
print(attr.age)  
  
del attr.name  
print(attr.name)

调用了 \_\_setattr\_\_ 方法

调用了 \_\_setattr\_\_ 方法

调用了 \_\_getattribute\_\_ 方法

shiyong

调用了 \_\_getattribute\_\_ 方法

调用了 \_\_getattr\_\_ 方法

None

调用了 \_\_delattr\_\_ 方法

调用了 \_\_getattribute\_\_ 方法

Shiyong

## 描述器\_\_get set delte\_\_

当类里面包含 get set delte 任意一个方法时 此类为描述器类

当描述器类对象为 其他类属性时 修改描述器对象属性操作会被覆盖 调用特殊属性方

class M:  
  
 def \_\_get\_\_(self, instance, owner):  
 print('描述器对象访问属性时\_\_get\_\_ 方法被调用')  
 print('self:', self)  
 print('instance:', instance)  
 print('owner;', owner)  
 return self.value  
  
 def \_\_set\_\_(self, instance, value):  
 print('描述器对象设置属性时\_\_set\_\_ 方法被调用')  
 print('self:', self)  
 print('instance:', instance)  
 print('value;', value)  
 self.value = value  
  
 def \_\_delete\_\_(self, instance):  
 print('删除描述器对象时 \_\_del\_\_ 方法被调用 ')  
 self.value = None  
  
  
class Common:  
 m = M()  
  
  
common = Common()  
common.m = 'shiyong'  
del common.m  
print(common.m)  
print(type(common.m))

描述器对象设置属性时\_\_set\_\_ 方法被调用

self: <\_\_main\_\_.M object at 0x012AF0B0>

instance: <\_\_main\_\_.Common object at 0x012F0670>

value; shiyong

删除描述器对象时 \_\_del\_\_ 方法被调用

描述器对象访问属性时\_\_get\_\_ 方法被调用

self: <\_\_main\_\_.M object at 0x012AF0B0>

instance: <\_\_main\_\_.Common object at 0x012F0670>

owner; <class '\_\_main\_\_.Common'>

None

描述器对象访问属性时\_\_get\_\_ 方法被调用

self: <\_\_main\_\_.M object at 0x012AF0B0>

instance: <\_\_main\_\_.Common object at 0x012F0670>

owner; <class '\_\_main\_\_.Common'>

<class 'NoneType'>

## 伪多态

多个子类继承一个父类，子类里面对父类的方法进行了重写 实现传不同对象实现不同功能

遵循 开放封闭原则 允许通过子类扩展功能 不允许修改父类

class Base:  
 def func(self):  
 print('动物类型 会走路')  
  
  
class Dog(Base):  
 def func(self):  
 print(' 这个是狗 可以跑步')  
  
  
class Cat(Base):  
 def func(self):  
 print(' 这个是猫 可以走猫步')  
  
  
class Pig(Base):  
 def func(self):  
 print('动物类型 慢慢走路')  
  
  
class Ya(Base):  
 pass  
  
  
  
a = Dog()  
b = Cat()  
c = Pig()  
d = Ya()  
a.func()  
b.func()  
c.func()  
d.func()

这个是狗 可以跑步

这个是猫 可以走猫步

动物类型 慢慢走路

动物类型 会走路

## Orm模型

## 描述器初步实现

class CharFiled:  
  
 def \_\_init\_\_(self, max\_length=10):  
 self.max\_length = max\_length  
  
 def \_\_get\_\_(self, instance, owner):  
 return self.value  
  
 def \_\_set\_\_(self, instance, value):  
 if isinstance(value, str):  
 if len(value) > self.max\_length:  
 raise ValueError('value length must less than {} '.format(self.max\_length))  
 else:  
 self.value = value  
 else:  
 raise TypeError('value type must is str ')  
  
 def \_\_delete\_\_(self, instance):  
 self.value = None  
  
  
class IntFiled:  
  
 def \_\_get\_\_(self, instance, owner):  
 return self.value  
  
 def \_\_set\_\_(self, instance, value):  
 if isinstance(value, int):  
 self.value = value  
 else:  
 raise TypeError('value type must is int ')  
  
 def \_\_delete\_\_(self, instance):  
 self.value = None  
  
  
class Model(object):  
 user = CharFiled(max\_length=20)  
 pwd = CharFiled(max\_length=10)  
 age = IntFiled()  
  
model = Model()  
model.user = 'shiyong'  
model.pwd = 'password'  
model.age = '18'  
print(model.user)  
print(model.pwd)  
print(model.age)  
print(type(model.age))

shiyong

password

18

<class 'int'>

## 元类

所有类都是 由type方法创建的 type就是元类

所有类都继承于 object ，object为顶级类

metaclass父亲制定元类 ，子类也会继承 父类继承的元类

*"""  
自定义元类  
指定元类后 会先把当前类加载完成 然后把当前类 anme 属性 方法 传入元类new方法参数 通过type 创建类，重写new方法  
类名 属性 方法 当做入参 return 返回  
  
"""*# 自定义元类 必须继承type类  
class MyTest(type):  
 def \_\_new\_\_(cls, name, base, attrs, \*args, \*\*kwargs):  
 print('自定义元类')  
 print(attrs)  
 return super().\_\_new\_\_(cls, name, base, attrs)  
  
  
# 指定元类后 会先把当前类加载完成 然后把当前类 anme 属性 方法 传入元类new方法参数 创建类 r  
class Test(metaclass=MyTest):  
 aa='shiyong'  
 def eat(self):  
 print('吃吃吃')

## Orm模型数据存储

### 1.创建orm数据类型类

# 字段父类  
class BaseFiled:  
 pass  
  
  
class IntFiled(BaseFiled):  
 def \_\_get\_\_(self, instance, owner):  
 return self.value  
  
 def \_\_set\_\_(self, instance, value):  
 if isinstance(value, int):  
 self.value = value  
 else:  
 raise TypeError('value type must is int')  
  
 def \_\_delete\_\_(self, instance):  
 self.value = None  
  
  
class StrFiled(BaseFiled):  
 def \_\_get\_\_(self, instance, owner):  
 return self.value  
  
 def \_\_set\_\_(self, instance, value):  
 if isinstance(value, str):  
 self.value = value  
 else:  
 raise TypeError('value type must is str')  
  
 def \_\_delete\_\_(self, instance):  
 self.value = None  
  
  
class Boolean(BaseFiled):  
 def \_\_get\_\_(self, instance, owner):  
 return self.value  
  
 def \_\_set\_\_(self, instance, value):  
 if isinstance(value, bool):  
 self.value = value  
 else:  
 TypeError('value type must is bool')  
  
 def \_\_delete\_\_(self, instance):  
 self.value = None

### 2.创建 模型类 元类/模型类/模型类父类

from model\_type import IntFiled, StrFiled, Boolean, BaseFiled  
  
  
class FileMeatClass(type):  
 def \_\_new\_\_(cls, name, base, dic, \*args, \*\*kwargs):  
 t\_name = name.lower() # 将类名转化为小写，对应表的名称  
 fields = {} # 定义 模型字段 属性容器  
 for k, v in list(dic.items()):  
 # 如果模型字段 属性值是否为字段类型,把字段类型属性存储起来  
 if isinstance(v, BaseFiled):  
 fields[k] = v  
  
 dic['t\_name'] = t\_name # 表名存储  
 dic['fields'] = fields # 字段属性存储  
 return super().\_\_new\_\_(cls, name, base, dic)  
  
  
# 所有模型类 父类  
class BaseModel(metaclass=FileMeatClass):  
 *"""  
 所有模型类的父类  
 1.实现指定元类  
 2.模型类属性初始化  
  
 """* def \_\_init\_\_(self, \*\*kwargs):  
 for k, v in kwargs.items():  
 setattr(self, k, v)  
  
 def save(self):  
 dict\_fields = {}  
 t\_name = self.t\_name  
 fields = self.fields  
 for k in fields.keys():  
 dict\_fields[k] = getattr(self,k)  
  
 print(dict\_fields)  
  
  
# 利用元类 实现模型类  
class User(BaseModel):  
 *"""  
 用户模型类  
 """* user\_name = StrFiled()  
 pwd = StrFiled()  
 age = IntFiled()  
 live = Boolean()  
 pass  
  
  
xiaoming = User(user\_name='xiaoming', pwd='lb123456', age=18, live=True)  
print(xiaoming.age)  
xiaoming.save()

## 内存管理

### 小整数池

Python会事先创建好整数 -5 ~256 的整数池，当我们创建的变量值符合 -5~256时不会 新创建一块内存 变量会直接指向小整数池这块内存

# 小整数池  
a=5  
b=6  
c=5  
print(id(a))  
print(id(b))  
print(id(c))

1596648000

1596648016

1596648000

### 大整数池

创建字符串类型为 整数 或者 字母 或者 下划线时 会存储到大整数池里，当有新的字符变量时 切、且值满足大整数池条件 就会查找值是否已存在 如果已存在会指向同一值

"""  
大整数池  
"""  
aa = 'abc123'  
bb = 'abc123'  
print(id(aa))  
print(id(bb))

22544128

22544128

### 浅copy

使用数据对象直接使用copy方法 copy时 只copy浅层数据 ，不会copy深层数据 ，浅copy 时 深层数据 属于引用关系

import copy  
  
li = [1, 2, 3]  
  
list1 = [4, 5, 6, li]  
  
list2 = list1  
  
list3 = list1.copy()  
  
list4 = copy.deepcopy(list1)  
  
  
print(id(li),li)  
print(id(list1),list1,'lsit1')  
print(id(list2),list2,'lsit2')  
print(id(list3),list3,'lsit3')  
print(id(list3),list3,'lsit4')  
list1.append('aa')  
li.append('cc')  
  
print(id(list1),list1,'lsit1')  
print(id(list2),list2,'lsit2')  
print(id(list3),list3,'lsit3')  
print(id(list4),list4,'lsit4')

1596648000

1596648016

1596648000

22544128

22544128

### 深copy

深copy需要 导入 copy模块 用copy.deepcopy方法copy ，会吧深参数据也copy出一份 不再是引用关系 一般用于列表嵌套

import copy  
  
li = [1, 2, 3]  
  
list1 = [4, 5, 6, li]  
  
list2 = list1  
  
list3 = copy.deepcopy()

### 垃圾回收机制

垃圾回收机制主要是 引用计数为主 标记清除 和隔代回收为辅

#### 1.引用计数：

变量引用计数 当变量引用计数为0时会自动清除；

缺点：容易造成内存泄漏

#### 2.标记清除：

标记清除起引用计数的辅助作用 比如我们外部的变量引用已经清除 但是他在内部有一些相互引用 还会存在内部 ，比如循环引用

如果没被全局变量直接引用 就会被清除

#### 3.分代回收

也是 辅助引用计数 监测计数频率

当我们内存里面数据对象数量创建达到700个时之后 ，那么这个时候 分代回收机制就会处罚自动监测 监测这些对象 引用计数为0的就会进行删除，然后监测到计数不为0的就会放第代链表里面，放完后 第一代链表 没有数据对象了，当我们在创建新的对象对象时 都会放到第1代链表里 当1代链表再次达到700个时候 再次进行监测 引用0的直接清除 不为0的继续放到迭代链表里， 当地一代链表监测10次后 那么地二代链表会扫描一次 第二代链表监测 不为0的会方到 第三代链表里面，然后当第二代链表 扫描10次后 这时候第三代链表会自动扫描清除

## threading多线程

### threading模块常用方法

#### threading.Thread（fun）

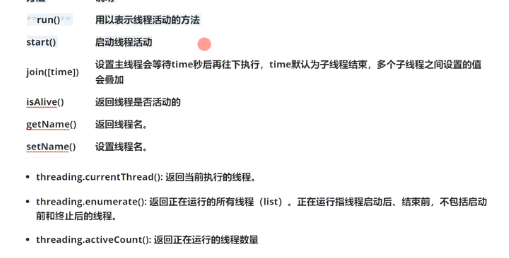
#### run()

#### start()

#### join()

#### getName()

#### setName()



### 并发概念：

任务数大于cpu核时，cpu 在多个任务中 以0.01秒不断的切换运行，看上去是同时运行的 实际是在交替运行 只是执行的速度太快了

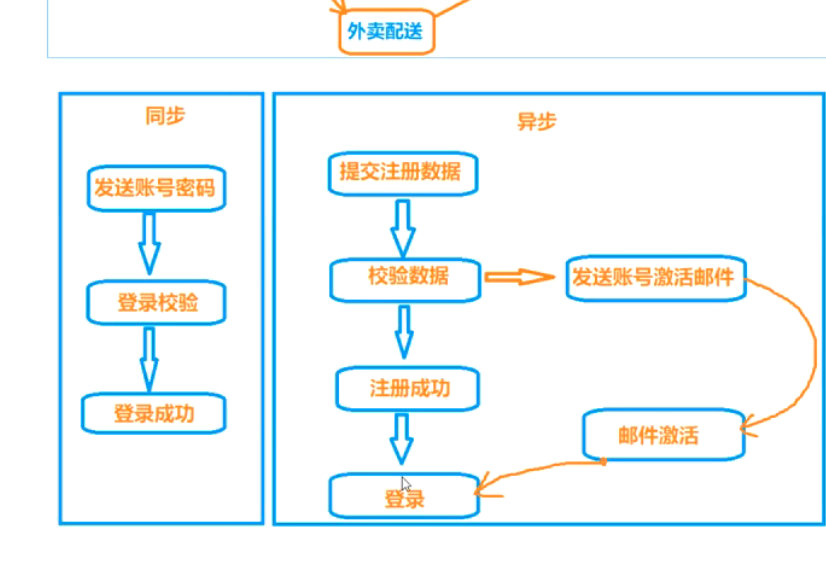
### 并行概念：

任务数小于cpu核时 每个核心分配一个任务 是真正意义上的并行

### 并发时遇到的问题

1. 多个线程修改全局变量时 会出现资源竞争问题

### 同步与异步概念



### 单线程 与多线程哪个种效率高

#### 1.网络IO密集型：

网络io密集型的时候 cpu使用率较低 大多数时间在等待 磁盘 与网络响应 ，此时用多线程效率比较高

#### cpu 密集型

cpu资源使用比较高 多个任务cpu交替执行反而 耽搁时间 效率不如单线程

### python线程可以并行吗

不可以 gil全局锁 每个线程执行的时候都要获取 gil锁

### 多线程二次封装 计算接口平均响应时间

import threading  
import requests  
import time  
  
url = 'https://api-t-9.azazie.com/1.0/user/login'  
  
data = {'email': 'shiyong1111@tetx.com',  
 'password': '123456'}  
  
  
count =0  
  
class MyThread(threading.Thread):  
 def \_\_init\_\_(self, url, data):  
 self.data = data  
 self.url = url  
 super().\_\_init\_\_()  
  
 # 重写了run方法 不需要去线程组对象入参 取方法 直接就运行了  
 def run(self):  
 global count  
 for i in range(5):  
 count+=1  
 res = requests.post(url=url,json=data)  
 print(threading.current\_thread(),res.status\_code)  
 print('发送的第：{} 次请求'.format(i+1))  
  
# 循环创建5个线程对象,放入列表  
threads = [MyThread(url, data) for i in range(5)]  
start\_time = time.time()  
# 遍历 线程对象运行  
for i in threads:  
 i.start()  
  
# 遍历线程对象 同时每个线程都join等下  
for j in threads:  
 j.join()  
  
ent\_etime = time.time()  
print('平均响应时间为： {}.'.format((ent\_etime-start\_time)/25))

### 互斥锁/线程锁

#### 线程锁的优缺点

##### 优点

：锁的好处就是保证被锁住的代码能够完整的被执行

##### 缺点：

锁使用错误可能会造成死锁

阻止了并发运行 效率大幅度下降

#### Threading.Lock（）

创建线程锁对象

Thread1 = Threading.Lock（）

#### Thread1.acquire() 获取锁

#### Thread1.release()释放锁

import threading  
import time  
  
a = 0  
  
  
def func1():  
 global a  
  
 for i in range(1000000):  
 lock1.acquire()  
 a += 1  
 lock1.release()  
  
  
def func2():  
 global a  
  
 for i in range(1000000):  
 lock1.acquire()  
 a += 1  
 lock1.release()  
  
  
lock1 = threading.Lock()  
start\_time = time.time()  
t1 = threading.Thread(target=func1)  
t2 = threading.Thread(target=func2)  
t1.start()  
t2.start()  
t1.join()  
t2.join()  
  
end\_time = time.time()  
print(end\_time - start\_time)  
  
print(a)

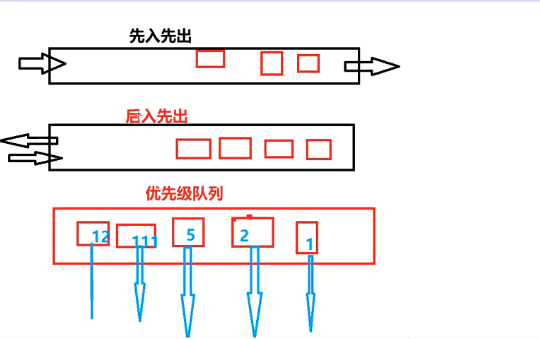
1.0876176357269287

2000000

## 队列

队列内部已经实现了 多线程的锁元语 能够在多线程中直接使用 可以使用队列来实现 线程同步

队列有 3种队列 1.先入先出 2.先入后出 3.优先级



### 1.FIFO先入先出

#### Put 函数

"""  
put 函数  
  
# put 往队列里添加对象 当等待未Ture时 队列已满添加不会报错 因为在等待  
que = queue.Queue(3)  
que.put(11)  
que.put(22)  
que.put(33)  
  
# 当block 参数为False时 不在等待如果任务数 已满再添加就会报错  
# ue.put(44, False) = que.put\_nowait(44)  
  
que.put(44, False)  
"""

#### Get 函数

que = queue.Queue(3) # def \_\_init\_\_(self, maxsize=0) 最大传入任务数  
que.put(11) # def put(self, item, block=True, timeout=None) 是否等待 超时  
que.put(22)  
que.put(33)  
print(que.get())  
print(que.get())  
print(que.get())  
# print(que.get(block=False)) # que.get(block=False) = que.get\_nowait() 队列为空取 不等的会报错  
  
# 获取当前 队列中的未取任务数  
print(que.qsize())  
  
# 判断当前队列是否已满  
print(que.full())

#### qsize()/full()/empty() 函数

# 获取当前 队列中的未取任务数  
print(que.qsize())  
  
# 判断当前队列是否已满  
print(que.full())  
  
# 判断当前队列是否为空  
print(que.empty())

#### Que.task\_done()

# 向队列发送一个信号 告诉队列一个任务已经完成 ，每个get（）都会调用都会得到一个任务，  
que.task\_done()  
que.task\_done()  
que.task\_done()  
# que.join()  
print('111111')

#### Queue.join()

等待队列 任务完成后 在直接别的操作

Queue.join() 实际上意味着等到队列为空，再执行别的操作

### Lifo后入先出

# 后入先出队列  
que = queue.LifoQueue()  
que.put(11)  
que.put(22)  
que.put(33)  
  
print(que.get())

### PriorityQueue 优先级

存入任务时 传入元祖

que = queue.PriorityQueue()  
que.put((11,2))  
que.put((22,1))  
que.put((99,3))  
print(que.get())  
print(que.get())  
print(que.get())

## process进程

什么是进程：

一个程序代码运行 +用到的资源 就是进程 它是操作系统分配资源的基本单元

不仅可以通过多线程完成 任务 多进程也是可以的

进程与线程的区别

系统会给每个进程分配一快内存空间 ，而线程是在进程空间里面 再分配小的空间

多个线程 变量不能共享

### 多进程资源共享

Start（）

Join（）

进程共享队列需要用 from multiprocessing import Process, Queue队列

# import queue  
import os  
import time  
from multiprocessing import Process, Queue  
from multiprocessing import current\_process  
import requests  
  
headers1 = {"Content-Type": "application/json", "x-token": "",  
 "x-countryCode": 'us'}  
  
  
def add\_data():  
 *"""  
 生成数据 添加入队列* ***:return****:  
 """* url = 'https://api-t-9.azazie.com/1.0/user/register'  
 data = [[url, "syt" + str(j) + '@tetx.com', '123456'] for j in range(20)]  
 for i in data:  
 que.put(i)  
 print(current\_process().name + "--end", que.qsize())  
  
  
def func(queue):  
 *"""  
 如果 队列不为空 就调用注册接口* ***:param*** *queue:* ***:return****:  
 """* while queue.qsize() > 0:  
 print('(ID{},JOB({}))'.format(os.getpid(), queue.qsize()))  
 li\_data = queue.get()  
 data = {"email": li\_data[1], "password": li\_data[2]}  
 print(data)  
 res = requests.post(url=li\_data[0], json=data, headers=headers1)  
 print(res.json()['code'])  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 que = Queue()  
 li\_que = []  
  
 add\_data()  
  
 st = time.time()  
 for i in range(4):  
 process1 = Process(target=func, args=(que,))  
 li\_que.append(process1)  
  
 for i in li\_que:  
 i.start()  
  
 for i in li\_que:  
 i.join()  
 en = time.time()  
 print(en - st)  
 print(1111)

### 进程池pool

# 导入进程池 和 队列模块  
from multiprocessing import Pool, Manager  
import os  
import time  
  
a = 0  
  
  
# 创建任务  
def func(que):  
 print('开始运行任务 ，进程id：{}'.format(os.getpid()))  
 print(que.get())  
 time.sleep(0.5)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # 创建进程池可用队列  
 que = Manager().Queue()  
 Manager()  
  
 # 队列添加 10000个任务  
 for i in range(10):  
 que.put(i)  
  
 # 创建进程池最大可用4个进程  
 pool = Pool(2)  
  
 # 如果队列任务大于0  
 for i in range(que.qsize()):  
 # 每次循环都会用 空闲出来的进程去调用目标，传参为不定参数 传入元祖  
 pool.apply(func=func, args=(que,))  
  
 # 关闭进程池 pool不再接收新的请求  
 pool.close()  
 # 等待所有进程完成 必须在close 之后  
 pool.join()  
  
 print('进程结束继续运行之后代码')

## gevent协程

## 进程 线程 协成对比

### 进程

是cpu资源的分配，切换起来占用资源比较高

### 线程

线程的切换条件是获取gil 全局解释器锁 或者 执行到了需要时间等待 或者 执行时间到了一定阈值

是系统资源的调度 线程需要资源一般 效率一般

因为 python解释器 gil锁的缘故 python线程不会并行 也是并发

除非 当线程的任务是 文件写入操作时 此时不需要用到python解释器 这个时候其他线程就可以同时执行其他任务

### 协成

协成切换条件是 遇到协成耗时等待的情况下 用gevent.seelp

是任务资源很小 被进程包含 效率很高 ，他运行的过程就是 当某一个协成空闲时 会做其他任务