# 第一模块

## 1．python解释器说明

Python 解释器

Cpython ：官方版本解释器（C语言开发的）

Ipython ：和Cpython 差不多

Jpython : Java平台上的解释器

PyPy ：Python 写的解释器（采用JIT技术）

IronPython

## 2．python3 安装

linux版本：

#!/bin/bash

#wget https://www.python.org/ftp/python/3.6.3/Python-3.6.3.tgz

#定义...

PYTHON\_PATH="/usr/local/python36"

set -e

set -x

#身份检查

if [ $(id -u) != "0" ]; then

echo "error: user must be an administrator"

exit;

fi

#依赖

yum install gcc gcc-c++ kernel-devel kenel-headers make bzip2 readline readline-devel readline-static openssl openssl-devel openssl-static sqlite-devel gdbm-devel bzip2-devel xz-devel tk-devel bzip2-libs patch vim git ncurses-devel -y

#判断是否有源码包

[ -s Python-3.6.3.tgz ] || exit 1

tar xf Python-3.6.3.tgz

cd Python-3.6.3

./configure --prefix=${PYTHON\_PATH} && make && make install

ln -s ${PYTHON\_PATH}/bin/python3.6 /usr/bin/python3

ln -s ${PYTHON\_PATH}/bin/pip3 /usr/bin/pip3

ln -s ${PYTHON\_PATH}/bin/pyvenv-3.6 /usr/bin/pyvenv3

echo "export PATH="${PYTHON\_PATH}/bin/:$PATH"">>/etc/profile

source /etc/profile

exit 0

## 3．变量

### 3.1 变量定义规则：

变量名只能是 字母、数字或下划线的任意组合

变量名的第一个字符不能是数字

关键字不能声明为变量

>>> import keyword

>>> keyword.kwlist

['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

>>> a,b = (1,2)

>>> a

1

>>> b

2

### 3.2 命令习惯

变量命令：驼峰型，下划线型

函数命令：全小写，或小写+下划线型

类命名： 驼峰式，驼峰型+下划线型

（变量名不易过长，变量名表达明确）

### 3.3 常量

常量命令：全部大写（不是必须的）

## 4. 数据类型

基本数据类型

数 字（integer） ：整数（int）、长整型（long）<python3 没有long类型>、浮点型（float）

字符串（string）： 文本（str）、字节（bytes）

布 尔（bool） ： True/False

数据集

列表（list）

元组（tuple）

字典（dict）：有序字典、无序字典

集合（set）：有序集合、无序集合

## 5. 字符串（string）

字符串格式化： 字符串 %s 整数 %d 浮点 %f 原样打印 %r

input 默认输出的都是字符串

#!/usr/bin/env python

#coding:utf-8

'''字符串格式化：%s 代表字符串，%d 代表 数字，%f 表示浮点数'''

name = input("name:")

age = int(input("age:"))

job = input("job:")

hometown = input("hometown:")

info = """

-------info of %s -----------

Name: %s

Age : %d

Job : %s

Hometown: %s

--------end------------------

""" % (name,name,age,job,hometown)

print(info)

基于字典的字符串格式化

params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}

print("%(pwd)s" % params) # 'secret'

print("%(pwd)s is not a good password for %(uid)s" % params) # 'secret is not a good password for sa'

print("%(database)s of mind, %(database)s of body" % params) # 'master of mind, master of body'

## 6. 运算符

算数运算符：+ - \* / // \*\*

比较运算符： == != > < >= <= #python3 里面已经取消 <>

赋值运算符： = += -= \*= /= %= //=

逻辑运算符：and not or

## 7. 流程控制

### 7.1 if 语句

# 布尔值操作符 and or not 实现多重判断

a = input(">>:")

b = input(">>:")

if a == b:

print('==')

elif a < b:

print(b)

else:

print(a)

### 7.2 while 语句

例1：

while True:

if a == b:

print("==")

break

print("!=")

else:

print('over')

例2：

count = 0

while count <= 10:

if count == 5:

pass

elif 6 <= count <= 8:

print(count\*\*2)

else:

print(count)

count += 1

例3：

while True:

print("永远执行")

while 1:

print("永远执行")

break and continue 说明：

break:用于完全结束一个循环，跳出循环体执行循环后面的语句

continue：终止本次循环，接着执行后面的循环，break 则完全终止循环

## 8. 练习题：

1.简述编译型与解释型语言的区别，且分别列出你知道的那些语言属于编译型，那些属于解释型。

编译型语言在程序执行之前，有一个单独的编译过程，将程序翻译成机器语言，以后执行这个程序的时候，就不用再进行翻译了。编译可以达到最大的优化；

解释型语言，是在运行的时候将程序翻译成机器语言，所以运行速度相对于编译型语言要慢。解释可以达到最大的灵活。

编译型语言： C/C++

解释型语言：java C# python

2. 执行python脚本的两种方式是什么？

环境：linux

1、 python example.py

2、 chmod +x example.py && ./example.py

3.python 单行注释和多行注释分别用什么。

单行注释： #

多行注释： '''注释'''

4.布尔值分别有什么

True / False

5.声明变量注意事项有哪些？

变量只能是字母、数字、下划线的组合

变量开头不能用数字

变量不要使用关键字

变量命名尽量易识别

6.如何查看变量在内存中的地址？

a = 1

id(a)

7.写代码

(1)实现用户输入用户名和密码，当用户名为 seven 且密码为123 时，显示登录成功，否则登录失败！

#coding:utf-8

username = "seven"

password = "123"

login\_user = input("login:")

login\_passwd = input("password:")

if login\_user == username and password == login\_passwd:

print("登录成功")

else:

print("登录失败")

(2)实现用户输入用户名和密码，当用户名为 seven 且密码为123 时，显示登录成功，否则登录失败，失败时允许用户重复输入三次

#coding:utf-8

username = "seven"

password = "123"

num = 0

while num < 3:

login\_user = input("login:")

login\_passwd = input("password:")

num += 1

if login\_user == username and password == login\_passwd:

print("登录成功")

else:

print("登录失败")

(3)实现用户输入用户名和密码，当用户名为 seven 或 alex 且密码为123 时，显示登录成功，否则登录失败，失败时允许用户重复输入三次

#coding:utf-8

username01 = "seven"

username02 = "alex"

password = "123"

num = 0

while num < 3:

login\_user = input("login:").strip()

login\_passwd = input("password:")

num += 1

if login\_user == username01 or login\_user == username02:

if password == login\_passwd:

print("登录成功")

break

else:

print("密码错误")

else:

print("用户名错误、登录失败")

8.写代码

(a)使用while循环实现输出 2-3+4-5+6.....+100 的和

#coding:utf-8

sum = 0

n = 1

while n < 100:

n += 1

mode = n%2

if mode == 0:

sum += n

else:

sum -= n

else:

print("sum:",sum)

(b)使用while循环实现输出 1,2,3,4,5,67,8,9,11,12

#coding:utf-8

n = 0

while n < 12:

n += 1

if n == 10:

continue

else:

print(n)

（d）使用while循环实现输出 1-100内的所有奇数

#coding:utf-8

n = 0

while n < 100:

n += 1

mode = n%2

if mode == 1:

print(n)

(e) 使用while循环实现输出1-100内的所有偶数

#coding:utf-8

n = 0

while n < 100:

n += 1

mode = n%2

if mode == 0:

print(n)

9.现有如下两个变量，请简述 n1 和 n2 是什么关系？

n1 = 123456

n2 = n1

解:

n1,n2 它们都是变量，n1 首先被赋值了一个整数，此刻n1的值在内存中存在了一个内存地址"id(n1)"可以得到n1的值在内存中的地址。当 n2 = n1 时，是n2 的内存指针指向了n1 的值内存地址，因此，此时 n2 = 123456，当n1被重新赋值以后，n1 的内存指针指向了另外的内存地址，但是n2的内存指针在n1被重新赋值时，已经指向了值“123456”，因此，就算n1 的值改变了，n2的内存指针指向的内存地址不变，n2的值也就不变。

## 9. 进制转换

十进制转二进制

>>> bin(342)

'0b101010110' #不看0b

10进制转16进制: hex(16)

ASCII码转换：

>>> ord('a')

97

>>> chr(65)

'A'

8bit = 1bytes = 1B

1kB = 1024B

1BB = 1024YB = 1024\*\*2 ZB = 1024\*\*3 EB = 1024\*\*4 PB = 1024\*\*5 TB = 1024\*\*6 TB = 1024\*\*7 GB = 1024\*\*8 MB = 1024\*\*9 KB = 1024\*\*10 B

## 10．列表（list）

列表是一个数据的集合，集合内可以放任何数据类型，可对集合进行方便的增删改查操作

列表的功能：

创建 li = [] or li = list()

查询 li.index(value)

切片 li[0:3],li[-5,-1],li[0:6:2]

增加 li.append(value),li.insert(index,obj)

修改 li[2] = 3

删除 li.pop() ,li.remove(value), del li[3] ,del li[2:5]

循环 for

排序 li.sort() ,li.reverse()

列表类型内建函数

list.clear() # 清楚列表

list.copy() # 复制列表 l1 = [1,2] l2=l1.copy()

\*list.append(obj) # 向列表中添加一个对象obj

\*list.count(obj) # 返回一个对象obj在列表中出现的次数

list.extend(seq) # 把序列seq的内容添加到列表中

\*list.index(obj,i=0,j=len(list)) # 返回list[k] == obj 的k值,并且k的范围在i<=k<j;否则异常

\*list.insert(index.obj) # 在索引量为index的位置插入对象obj

\*list.pop(index=-1) # 删除并返回指定位置的对象,默认是最后一个对象

list.remove(obj) # 从列表中删除对象obj

list.reverse() # 原地翻转列表

list.sort(func=None,key=None,reverse=False) # 以指定的方式排序列表中成员,如果func和key参数指定,则按照指定的方式比较各个元素,如果reverse标志被置为True,则列表以反序排列

## 11. 深浅copy

import copy

x = copy.copy(y) # make a shallow copy of y #浅拷贝：只拷贝顶级的对象，或者说：父级对象

x = copy.deepcopy(y) # make a deep copy of y #深拷贝：拷贝所有对象，顶级对象及其嵌套对象。或者说：父级对象及其子对象

1.深浅拷贝都是对源对象的复制，占用不同的内存空间

2.如果源对象只有一级目录，那么源对象做任何改动，都不影响深浅拷贝对象

3.如果源对象不止一级目录，那么源对象做任何改动，都要影响浅拷贝，但不影响深拷贝。

4.序列对象的切片是浅拷贝，即只拷贝顶级的对象。

12. 字符串

特性

有序

不可变

字符串类型内建方法

string = 'abc'

string.capitalize() # 字符串首字母大写

string.center(width,[fillchar]) # 字符串居中显示（两边用空格填充）,fillchar 可以用字符填充

string.encode(encoding='utf-8')

string.casefold() # 字符串全小写

string.expandtabs(tabsize=8) # tab符号转为空格 #默认8个空格

string.format() # 字符串格式化 等同于 '%'

\*string.endswith(obj,beg=0,end=len(staring)) # 检测字符串是否已obj结束,如果是返回True #如果beg或end指定检测范围是否已obj结束

\*string.count(str,beg=0,end=len(string)) # 检测str在string里出现次数 f.count('\n',0,len(f)) 判断文件行数

\*string.find(str,beg=0,end=len(string)) # 检测str是否包含在string中,返回索引

\*string.index(str,beg=0,end=len(string)) # 检测str不在string中,会报异常

string.isalnum() # 如果string至少有一个字符并且所有字符都是字母或数字则返回True

string.isalpha() # 如果string至少有一个字符并且所有字符都是字母则返回True

string.isnumeric() # 如果string只包含数字字符,则返回True

string.isspace() # 如果string包含空格则返回True

string.isupper() # 字符串都是大写返回True

string.islower() # 字符串都是小写返回True

string.lower() # 转换字符串中所有大写为小写

string.upper() # 转换字符串中所有小写为大写

string.lstrip() # 去掉string左边的空格

string.rstrip() # 去掉string字符末尾的空格

\*string.replace(str1,str2,num=string.count(str1)) # 把string中的str1替换成str2,如果num指定,则替换不超过num次

\*string.startswith(obj,beg=0,end=len(string)) # 检测字符串是否以obj开头

string.zfill(width) # 返回字符长度为width的字符,原字符串右对齐,前面填充0

\*string.isdigit() # 只包含数字返回True

\*string.split("分隔符") # 把string切片成一个列表

\*":".join(string.split()) # 以:作为分隔符,将所有元素合并为一个新的字符串

## 12. 元组（tuple）

特性

不可变

元组本身不可变

## 13. 字典（dict）

字典是一种key-value 的数据类型

特性

key-value 结构

key必须可 hash、且必须为不可变数据类型、必须唯一

可存放任意多个值、可修改可以不唯一

无序

查找速度快

字典内建方法

dict.clear() # 删除字典中所有元素

\*dict copy() # 返回字典(浅复制)的一个副本

\*dict.fromkeys(seq,val=None) # 创建并返回一个新字典,以seq中的元素做该字典的键,val做该字典中所有键对的初始值

\*dict.get(key,default=None) # 对字典dict中的键key,返回它对应的值value,如果字典中不存在此键,则返回default值

\*dict.items() # 返回一个包含字典中键、值对元组的列表

\*dict.keys() # 返回一个包含字典中键的列表

dict.pop(key[,default]) # 和方法get()相似.如果字典中key键存在,删除并返回dict[key]

dict.setdefault(key,default=None) # 和set()相似,但如果字典中不存在key键,由dict[key]=default为它赋值

\*dict.update(dict2) # 将字典dict2的键值对添加到字典dict

\*dict.values() # 返回一个包含字典中所有值得列表

dict([container]) # 创建字典的工厂函数。提供容器类(container),就用其中的条目填充字典

len(mapping) # 返回映射的长度(键-值对的个数)

hash(obj) # 返回obj哈希值,判断某个对象是否可做一个字典的键值

## 14. 进制转换，字符编码详解

### 1.进制转换

进制转换：

表示各进制的符号：   
b：二进制，o：八进制，d：十进制，x：十六进制   
bin()、oct()、hex()返回值均为字符串，且分别带有0b、0o、0x前缀。

十进制转二进制

>>> bin(10)

'0b1010' #不看0b

10进制转8进制 ：oct（8）

>>> oct(10)

'0o12'

16进制转10进制: hex(16)

>>> hex(10)

'0xa'

二进制不方便阅读。16进制方便阅读，网络编程和数据存储。

字节换算：

8bit = 1bytes = 1B

1kB = 1024B

1BB = 1024YB = 1024\*\*2 ZB = 1024\*\*3 EB = 1024\*\*4 PB = 1024\*\*5 TB = 1024\*\*6 TB = 1024\*\*7 GB = 1024\*\*8 MB = 1024\*\*9 KB = 1024\*\*10 B

### 2.字符编码种类

编码的种类情况

ASCII 占1个字节，只支持英文

GB2312 占2个字节，支持6700+汉字

GBK GB2312的升级版，支持21000+汉字

Shift-JIS 日本字符

ks\_c\_5601-1987 韩国编码

TIS-620 泰国编码

### 3.unicode介绍

Unicode 起到了2个作用：

直接支持全球所有语言，每个国家都可以不用再使用自己之前的旧编码了，用unicode就可以了。(就跟英语是全球统一语言一样)

unicode包含了跟全球所有国家编码的映射关系，为什么呢？后面再讲

Unicode解决了字符和二进制的对应关系，但是使用unicode表示一个字符，太浪费空间。例如：利用unicode表示“Python”需要12个字节才能表示，比原来ASCII表示增加了1倍。

为了解决存储和网络传输的问题，出现了Unicode Transformation Format，学术名UTF，即：对unicode中的进行转换，以便于在存储和网络传输时可以节省空间!

UTF-8： 使用1、2、3、4个字节表示所有字符；优先使用1个字符、无法满足则使增加一个字节，最多4个字节。英文占1个字节、欧洲语系占2个、东亚占3个，其它及特殊字符占4个

UTF-16： 使用2、4个字节表示所有字符；优先使用2个字节，否则使用4个字节表示。

UTF-32： 使用4个字节表示所有字符；

总结：UTF 是为unicode编码 设计 的一种 在存储 和传输时节省空间的编码方案。

### 4.字符怎么存到硬盘上

无论以什么编码在内存里显示字符，存到硬盘上都是2进制。要注意的是，存到硬盘上时是以何种编码存的，再从硬盘是哪个读出来时，就必须以何种编码读出来，否则就乱码了。

### 5.python3 的执行过程

1.解释器找到代码文件，把代码字符串按文件头定义的编码加载到内存，转成unicode

2.把代码字符串按照语法规则进行解释，

3.所有的变量字符都会以unicode编码声明

Python3 里面如果没有申明代码编码，默认使用utf-8的编码格式。

Python的语法里是以Unicode申明的。

Python3就算申明了是utf-8类型。在解释器里面也会把字符串自动转成unicode类型。

### 6.python3编码转换语法

utf-8编码之所以能在windows gbk的终端下显示正常，是因为到了内存里python解释器把utf-8转成了unicode , 但是这只是python3, 并不是所有的编程语言在内存里默认编码都是unicode,比如 万恶的python2 就不是，它的默认编码是ASCII，想写中文，就必须声明文件头的coding为gbk or utf-8, 声明之后，python2解释器仅以文件头声明的编码去解释你的代码，加载到内存后，并不会主动帮你转为unicode,也就是说，你的文件编码是utf-8,加载到内存里，你的变量字符串就也是utf-8, 这意味着什么你知道么？。。。意味着，你以utf-8编码的文件，在windows是乱码。

乱是正常的，不乱才不正常，因为只有2种情况 ，你的windows上显示才不会乱

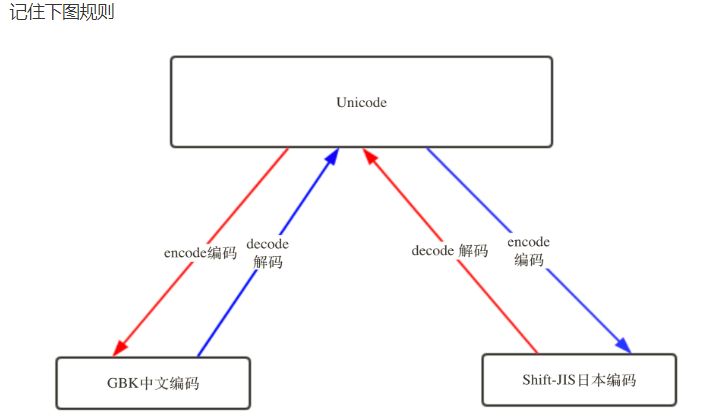
1. 字符串以GBK格式显示
2. 字符串是unicode编码

既然Python2并不会自动的把文件编码转为unicode存在内存里， 那就只能使出最后一招了，你自己人肉转。Py3 自动把文件编码转为unicode必定是调用了什么方法，这个方法就是，decode(解码) 和encode(编码)

UTF-8 --> decode（解码） --> Unicode

Unicode --> encode （编码） --> GBK / UTF-8 ..（把数据存到硬盘上就要从unicode转成utf-8等类型）

#coding：utf-8  
s = **"路飞学城"** # UTF-8类型，当在python2的windows 里面打印的时候，就会出现乱码。因为windows终端的格式是GBK  
s2 = s.decode(**"utf-8"**) #unicode 类型，utf-8 decode解码以后就变成了 unicode，就可以在windows终端打印  
s3 = s2.encode(**"gbk"**) #gbk类型，unicode通过encode编码以后就变成了gbk类型，windows终端也是gbk类型，因此也可以打印  
s4 = s2.encode(**"utf-8"**) #utf-8类型，unicode通过encode编码就变成了utf-8类型，在windows终端打印就乱码。  
**'''  
在python2 里面打印出来的内容，不是unicode 就是 str类型。  
'''**



### 7.进一步字符编码详解

UTF-8（8-bit Unicode Transformation Format）是一种针对Unicode的可变长度字符编码，它可以使用1~4个字节表示一个符号，根据不同的符号而变化字节长度，当字符在ASCII码的范围时，就用一个字节表示，所以是兼容ASCII编码的。

这样显著的好处是，虽然在我们内存中的数据都是unicode，但当数据要保存到磁盘或者用于网络传输时，直接使用unicode就远不如utf8省空间啦！这也是为什么utf8是我们的推荐编码方式。

Unicode与utf8的关系：

一言以蔽之：Unicode是内存编码表示方案（是规范），而UTF是如何保存和传输Unicode的方案（是实现）这也是UTF与Unicode的区别。

总结：一切都是为了节省你的硬盘和流量

### 8.py2中的string编码

在py2中，有两种字符串类型：str类型和unicode类型。注意，这仅仅是两个名字，python定义的两个名字，关键是这两种数据类型在程序运行时在内存地址的是什么？

**str --> Bytes数据**

**unicode ---> unicode数据**

py2 环境下：

>>> name = '中国'

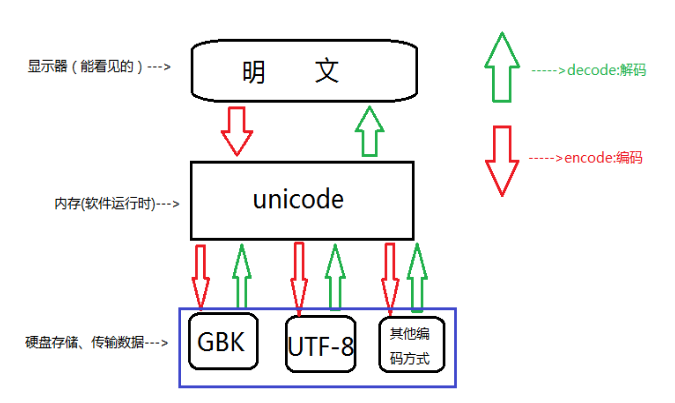
>>> print type(name) #打印name变量的数据类型

<type 'str'>

>>> print repr(name) #打印name的Bytes类型

'\xe4\xb8\xad\xe5\x9b\xbd'

内置函数 repr 可以帮我们在这里显示存储内容。str和unicode这两种数据类型之间是什么关系呢？如何转化？这里就设计到编码（enicode）和解码（decode）了



py2 中encode的编码转换：

>>> name = u'中国' #此时在字符串前面加上 u 就变成了unicdoe类型

>>> print type(name)

<type 'unicode'>

>>> print repr(name)

u'\u4e2d\u56fd'

>>> name2 = name.encode('utf-8') #通过encode编码，就成了 UTF-8 类型

>>> print name2,type(name2)

中国 <type 'str'>

>>> print name2,repr(name2)

中国 '\xe4\xb8\xad\xe5\x9b\xbd'

py2 中的 decode解码转换：

>>> name = "中国"

>>> a = name.decode('utf-8') #将name从str（Bytes）数据解码成unicode

>>> print a,type(a),repr(a)

中国 <type 'unicode'> u'\u4e2d\u56fd'

注意：

>>> name = "中国"

>>> a = name.decode('utf-8') #将name从str（Bytes）数据解码成unicode

>>> b = a.encode('gbk') #将a从unicode类型编码成 GBK方式的Bytes数据

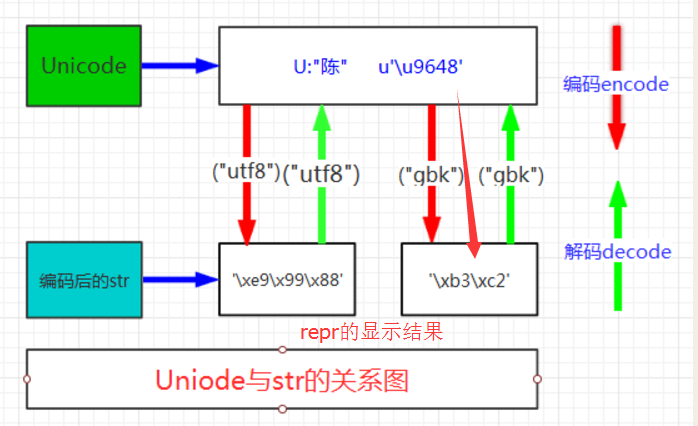
>>> print type(b)

<type 'str'>

>>> print b #出现乱码

א¹

因此无论是utf8还是gbk都只是一种编码规则，一种把unicode数据编码成字节数据的规则，所以utf8编码的字节一定要用utf8的规则解码，否则就会出现乱码或者报错的情况。



Python 2 悄悄掩盖掉了 byte 到 unicode 的转换，只要数据全部是 ASCII 的话，所有的转换都是正确的，一旦一个非 ASCII 字符偷偷进入你的程序，那么默认的解码将会失效，从而造成 UnicodeDecodeError 的错误。py2编码让程序在处理 ASCII 的时候更加简单。你付出的代价就是在处理非 ASCII 的时候将会失败。

### 9.py3的string编码

python3 renamed the unicode type to str ,the old str type has been replaced by bytes.

　　 py3也有两种数据类型：str和Bytes；  str类型存unicode数据，Bytse类型存Bytes数据，与py2比只是换了一下名字而已。

**str -----> unicode数据**

**Bytes -----> Bytes数据**

py3 的encode编码：

>>> name = "中国"

>>> print(type(name))

<class 'str'>

>>> print(name.encode('utf-8'))

b'\xe4\xb8\xad\xe5\x9b\xbd'

>>> print(name.encode('gbk'))

b'\xd6\xd0\xb9\xfa'

py3 的decode解码：

>>> b = name.encode('utf-8')

>>> print(type(b))

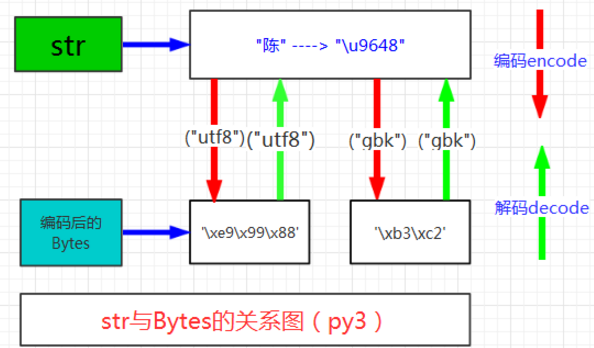
<class 'bytes'>

>>> print(b)

b'\xe4\xb8\xad\xe5\x9b\xbd'

>>> print(b.decode('utf-8'))

中国



py3的编码哲学：

Python 3最重要的新特性大概要算是对文本和二进制数据作了更为清晰的区分，不再会对bytes字节串进行自动解码。文本总是Unicode，由str类型表示，二进制数据则由bytes类型表示。Python 3不会以任意隐式的方式混用str和bytes，正是这使得两者的区分特别清晰。你不能拼接字符串和字节包，也无法在字节包里搜索字符串（反之亦然），也不能将字符串传入参数为字节包的函数（反之亦然）

python2中：当一个str（bytes）类型和unicode类型的字符串相加时可以得到结果：

>>> print 'abc'+u'xyz' #结果：abcxyz

注意：'abc'是str类型（也就是bytes类型），u'xyz' 是unicode类型

python3中：当一个bytes类型和unicode类型相加时会报错：

>>> print(b'abc'+'xyz')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: can't concat str to bytes

注意：b'abc' 是bytes类型，'xyz'是str（也是unicode类型）类型

### 10.文件从磁盘到内存的编码

说到这，才来到我们的重点！

　　抛开执行程序，请问大家，文本编辑器大家都是用过吧，如果不懂是什么，那么word总用过吧，ok，当我们在word上编辑文字的时候，不管是中文还是英文，计算机都是不认识的，那么在保存之前数据是通过什么形式存在内存的呢？yes，就是unicode数据，为什么要存unicode数据，这是因为它的名字最屌：万国码！解释起来就是无论英文，中文，日文，拉丁文，世界上的任何字符它都有唯一编码对应，所以兼容性是最好的。

　　好，那当我们保存了存到磁盘上的数据又是什么呢？

　　答案是通过某种编码方式编码的bytes字节串。比如utf-8 一种可变长编码，很好的节省了空间；当然还有历史产物的gbk编码等等。于是，在我们的文本编辑器软件都有默认的保存文件的编码方式，比如utf8，比如gbk。当我们点击保存的时候，这些编辑软件已经"默默地"帮我们做了编码工作。

　　那当我们再打开这个文件时，软件又默默地给我们做了解码的工作，将数据再解码成unicode,然后就可以呈现明文给用户了！所以，unicode是离用户更近的数据，bytes是离计算机更近的数据。

　　说了这么多，和我们程序执行有什么关系呢？

　　先明确一个概念：py解释器本身就是一个软件，一个类似于文本编辑器一样的软件！

　　现在让我们一起还原一个py文件从创建到执行的编码过程：

打开pycharm，创建hello.py文件，写入

|  |  |
| --- | --- |
| 1  2 | name = **"中国"**  print(name) |

当我们保存的的时候，hello.py文件就以pycharm默认的编码方式保存到了磁盘；关闭文件后再打开，pycharm就再以默认的编码方式对该文件打开后读到的内容进行解码，转成unicode到内存我们就看到了我们的明文；

      而如果我们点击运行按钮或者在命令行运行该文件时，py解释器就会被调用，打开文件，然后解码存在磁盘上的bytes数据成unicode数据，这个过程和编辑器是一样的，不同的是解释器会再将这些unicode数据翻译成C代码再转成二进制的数据流，最后通过控制操作系统调用cpu来执行这些二进制数据，整个过程才算结束。

那么问题来了，我们的文本编辑器有自己默认的编码解码方式，我们的解释器有吗？

当然有啦，py2默认ASCII码，py3默认的utf8，可以通过如下方式查询

|  |  |
| --- | --- |
| 1  2 | import sys  print(sys.getdefaultencoding()) |

　　大家还记得这个声明吗？

|  |  |
| --- | --- |
| 1 | #-\*- coding:utf8 -\*- |

　　是的，这就是因为如果py2解释器去执行一个utf8编码的文件，就会以默认地ASCII去解码utf8，一旦程序中有中文，自然就解码错误了，所以我们在文件开头位置声明　#coding:utf8，其实就是告诉解释器，你不要以默认的编码方式去解码这个文件，而是以utf8来解码。而py3的解释器因为默认utf8编码，所以就方便很多了。

注意：我们上面讲的string编码是在cpu执行程序时的存储状态，是另外一个过程，不要混淆！

### 11.常见的编码问题

1、cmd下的乱码问题

hello.py

#-\*- coding:utf8 -\*-

print ('中国')

　文件保存时的编码也为utf8。

思考：为什么在IDE下用2或3执行都没问题，在cmd.exe下3正确，2乱码呢？

我们在win下的终端即cmd.exe去执行，大家注意，cmd.exe本身也一个软件；当我们python2 hello.py时，python2解释器(默认ASCII编码)去按声明的utf8编码文件，而文件又是utf8保存的，所以没问题；问题出在当我们print'中国'时，解释器这边正常执行，也不会报错，只是print的内容会传递给cmd.exe用来显示，而在py2里这个内容就是utf8编码的字节数据，可这个软件默认的编码解码方式是GBK，所以cmd.exe用GBK的解码方式去解码utf8自然会乱码。

py3正确的原因是传递给cmd的是unicode数据，cmd.exe可以识别内容，所以显示没问题。

明白原理了，修改就有很多方式，比如：

print ('中国')

改成这样后，cmd下用2也不会有问题了。

2、open()中的编码问题

创建一个test文本，保存成utf8：

test.txt

#coding:utf-8

中国非常美

第一种写法：

test.py

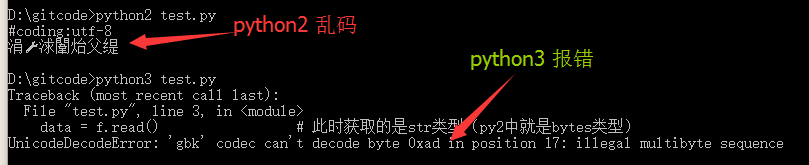
#coding:utf-8

f = open("test.txt",'r')

data = f.read() # 此时获取的是str类型（py2中就是bytes类型）

print(data)

f.close()



解决python2 乱码：

#coding:utf-8

f = open("test.txt",'r')

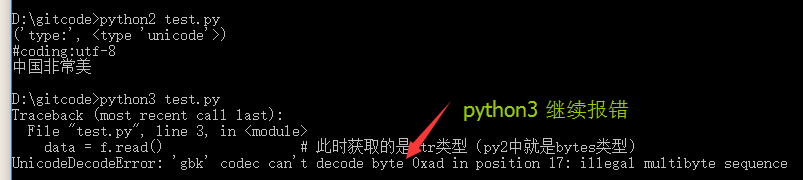
data = f.read() # 此时获取的是str类型（py2中就是bytes类型）

data = data.decode('utf-8') # 变成unicode类型。

print("type:",type(data))

print(data)

f.close()

在python2 下面第一种写法报错，是因为，cmd.exe 默认编码是GBK，而此时是 bytes类型，并且，当执行open函数时，调用的是操作系统打开文件，操作系统用默认的gbk编码去解码utf8的文件，所以乱码，当通过utf-8 解码以后，就变成了unicode，因此就可以正常显示了。

在python3 中 open的正确写法：

test.py

#coding:utf-8

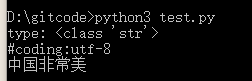
f = open("test.txt",'r',encoding='utf-8')

data = f.read() # 此时获取的是str类型（py3中就是unicode类型）

print("type:",type(data))

print(data)

f.close()



如果你的文件保存的是gbk编码，在win 下就不用指定encoding了。

　　另外，如果你的win上不需要指定给操作系统encoding='utf8'，那就是你安装时就是默认的utf8编码或者已经通过命令修改成了utf8编码。

注意：open这个函数在py2里和py3中是不同的，py3中有了一个encoding＝None参数

# 第二模块：函数编程(函数、装饰器、迭代器、内置方法)

## 第一章 文件操作

### 1.1 三元运算

>>> a = 2

>>> b = 5

>>> val = a if a < b else b

>>> val

2

>>> val = a if 3 > 6 else b

>>> val

5

### 1.2 文件处理（读、写、增加、混合模式）

读文件：

直接以rb模式打开文件，rb是指二进制模式，数据读到内存里直接是bytes格式，如果想看内容，还需要手动decode，因此在文件打开阶段，不需要指定编码

f = open(file=**"兼职白领学生空姐模特护士联系方式.txt"**, mode=**"rb"**)  
data = f.read()  
print(data)  
f.close()

智能检测编码的工具-chardet

>>> import chardet

>>> f = open("/etc/passwd",'rb')

>>> data = f.read()

>>> f.close()

>>> chardet.detect(data)

{'encoding': 'ascii', 'confidence': 1.0, 'language': ''}

写文件：

mode=**"w" 是创建一个新文件。如果有w一个旧文件，就会清空文件里面的所有内容。**

f = open(file=**"兼职白领学生空姐模特护士联系方式.txt"**, mode=**"w"**)  
data = f.read()  
print(data)  
f.close()

追加文件内容：

mode=**'a' 追加文件内容，在最末尾加上一个 ‘\n’另外起一行。**

f = open(**"兼职.txt"**,**'ab'**)  
f.write(**"\n北京 167 55 13523230322"**.encode(**"gbk"**))  
f.close()

混合模式（读写）：

mode=**'r+' 读写模式，先读后写，读的模式打开，支持追加。**

mode=**'w+' 写读模式，以写的模式打开（以创建的模式打开），新写的可以读。（几乎不用）**

### 1.2.1 扩展open函数使用方法

在python中读写文件可以使用内置函数**open()**，而 open 函数在python2 和 python3 中有一定的区别，比如 Python3中可以指定读写文件的编码格式，而 Python2 则不可以，为了同时兼容2和3，我们通常会使用io模块下面的 open 函数，大家可以查文档搞清楚它们之间的区别，培养主动学习能力和查资料的习惯。

方法可以查看后面re练习题 处理手机号：

**import** io  
**import** re  
**with io.open("test",'r',encoding="utf-8") as f:**  
 **for** line **in** f.readlines():  
 res = re.search(**'1[3458]\d{9}'**,line)  
 **if** res:  
 print(res.group())

### 1.3 文件操作其他功能

**def** fileno(self, \*args, \*\*kwargs): *# real signature unknown* 返回文件句柄在内核中的索引值，以后做IO多路复用时可以用到  
  
**def** flush(self, \*args, \*\*kwargs): *# real signature unknown* 把文件从内存buffer里强制刷新到硬盘  
  
**def** readable(self, \*args, \*\*kwargs): *# real signature unknown* 判断是否可读（正在写的文件不可读）  
  
**def** readline(self, \*args, \*\*kwargs): *# real signature unknown* 每次只读一行（根据光标往下读），遇到 \r **or** \n 为止  
  
**def** seek(self, \*args, \*\*kwargs): *# real signature unknown* 把操作文件的光标移到指定位置  
 \*注意seek的长度是按字节算的，字符编码存每个字符所占的字节长度不一样。  
  
**def** seekable(self, \*args, \*\*kwargs): *# real signature unknown* 判断文件是否可以进行seek操作  
  
**def** tell(self, \*args, \*\*kwargs): *# real signature unknown* 返回当前文件操作光标为止

\*注意seek的长度是按字节算的  
  
**def** truncate(self, \*args, \*\*kwargs): *# real signature unknown* 按指定长度截取文件  
 \*按指定长度的话，就从文件开头开始截断指定长度，不指定长度的话，就从当前位置到文件尾部的内容全部去掉。  
  
**def** writable(self, \*args, \*\*kwargs): *# real signature unknown* 判断文件是否可写

### 1.4 修改文件

修改文件两种方式：

1. 直接把要修改的文件内容写到一个新文件，然后在把新文件rename成为一个老文件。
2. 把修改的内容修改以后放到一个列表里面，然后，在把列表里面的内容写到源文件。注意：当要修改的文件，或者说是要替换的字符串的长度大于新字符串的长度时，文件原始的字节长度是不变的，因此 就会出现一部分字符串在修改文件的末尾，因此，此时就要用 文件操作的方法（truncate）来进行截断。因为，当文件修改以后，游标会到修改的那个位置，因此，在游标的位置进行truncate就会把原始文件未被占用的字节长度给截断掉。

例子展示：

文本文件如下：

[root@dev second\_modules]# cat text.log

hello world!

everyday study python

l like python

do you love python?

第一种方法：边读边写到磁盘（新文件）

[root@dev second\_modules]# cat modify\_file\_disk.py

*#!/usr/bin/env python  
#coding:utf-8*f\_name = **"text.log"**f\_new\_name = **"%s.new"** % f\_name  
old\_str = **"python"**new\_str = **"java"**f = open(f\_name,**'r'**,encoding=**"utf-8"**)  
f\_new = open(f\_new\_name,**'w'**,encoding=**"utf-8"**)  
**for** line **in** f:  
 **if** old\_str **in** line:  
 line = line.replace(old\_str,new\_str)  
 f\_new.write(line)  
f.close()  
f\_new.close()

如果要该名称：用到os模块的rename方法即可

第二种方法：先写到内存在写到源文件

[root@dev second\_modules]# cat modify\_file\_free.py

*#!/usr/bin/env python  
#coding:utf-8*f\_name = **"text.log"**old\_str = **"python"**new\_str = **"java"**f = open(f\_name,**'r+'**,encoding=**"utf-8"**)  
list01 = []  
**for** line **in** f:  
 **if** old\_str **in** line:  
 line = line.replace(old\_str,new\_str)  
 list01.append(line)  
values = **''**.join(list01)  
print(values)  
f.seek(0)  
f.write(values)  
**f.truncate()**  
f.close()

**\*\*** 上面红色字体的 truncate的意思就是，列表内容写完以后，假如源文件的字节长度大于现在的字节长度，那么就把现在的字节长度写完以后，进行截断。因为，不管是读还是写都是在源文件的基础上去进行的，源文件的原始字节长度不会改变。如果不进行截断，那么就会多出源文件的一部分字节出来。演示方法去掉truncate即可，并且替换的字节长度小于要替换的字节长度。

### 文本操作练习题

1. 全局替换程序

写一个脚本，运行用户按以下方式执行，即可以对指定文件内容进行全局替换

python your\_script.py old\_str new\_str filename

替换完毕以后打印替换了多少处内容

**'''  
文本例子  
[root@dev second\_modules]# cat text.log  
hello world!  
everyday study python  
l like python  
do you love python?  
'''  
import** sys  
**def** replace\_str(filename,old\_str,new\_str):  
 content\_list = []  
 replace\_times = 0  
 f = open(filename,**'r+'**)  
 **for** line **in** f:  
 **if** old\_str **in** line:  
 line = line.replace(old\_str,new\_str)  
 replace\_times += 1  
 content\_list.append(line)  
 values = **''**.join(content\_list)  
 f.seek(0)  
 f.write(values)  
 f.truncate()  
 **return** replace\_times  
  
**if '\_\_main\_\_'** == \_\_name\_\_:  
 old\_str = sys.argv[1]  
 new\_str = sys.argv[2]  
 filename = sys.argv[3]  
 times = replace\_str(filename,old\_str,new\_str)  
 print(**"replace total numbers:%d"**%times)  
**'''  
执行结果：  
[root@dev second\_modules]# python operate\_file\_replace.py python java text.log  
replace total numbers:3  
'''**

## 第二章 函数

### 2.1 函数-基本介绍

定义：函数是指将一组语句的集合通过一个名字（函数名）封装起来，要想执行这个函数，只需调用其函数名即可。

特性：

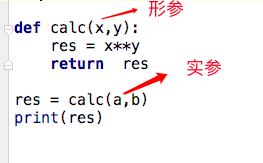
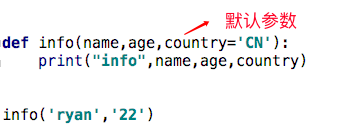
1. 减少重复代码
2. 是程序变的可扩展
3. 是程序变得易维护

### 2.2 函数参数

参数可以让你的函数更灵活，不只做固定的动作，还可以根据调用时传参的不同来决定函数内部的执行流程

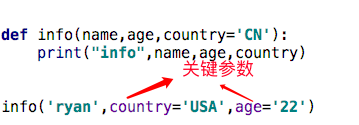
形参：只有在被调用时才分配内存单元，在调用结束时，即可释放所分配的内存单元。因此，形参只在函数内部有效。函数调用结束返回主调用函数后则不能再使用该形参变量

实参：可以是常量，变量，表达式、函数等，无论实参是何种类型的量，在进行函数调用时，它们都必须有确定的值，以便把这些值传送给形参。因此应预先用赋值，输入等办法使参数获得确定值。

默认参数：默认参数必须要放在位置参数后面。

关键参数：在调用函数的时候加上变量名就变成了关键参数。



**非固定参数**：如果函数需要传入多个非固定的参数时就要使用。

def fun01(\*args,\*\*kwargs)

带一个 \* 的非固定参数。

**def** send\_alert(msg,\*users):  
 **for** u **in** users:  
 print(**"报警发送给：%s "**%u)  
  
#如果参数中出现 \*users，传递的参数就可以不再是固定个数，传过来的所有参数打包成元组  
#方式一：  
send\_alert(**'cpu alert'**,**'a1'**,**'a2'**)  
send\_alert(**'cpu alert'**,**'u1'**,**'u2'**,**'u3'**)  
#方式二：  
#如果第二个参数不加 \* 并且是列表，那么发送给 \*users 的时候就会把列表当成是元组的第一个元素：（['u1','u2','u3'],）  
#如果加了 \* 那么就会变成（'u1','u2','u3'）  
send\_alert(**'cpu alert'**,\*[**'u1'**,**'u2'**,**'u3'**])

**def** send\_alter02(msg,\*args,\*\*kwargs):  
 print(msg,args,kwargs)

# \*\*kwargs 接收的是字典参数，**key-value类型**。具体用法如下。 *#方式一：*send\_alter02(**'ryan'**,18,**'SH'**,job=**'IT'**,num=1212)

*#结果：ryan (18, 'SH') {'job': 'IT', 'num': 1212}*

*#方式二：*dic = {**'job'**:**'IT'**,**'num'**:1212}  
send\_alter02(**'ryan'**,**'22'**,dic) *#结果：ryan ('22', {'job': 'IT', 'num': 1212}) {}  
#结果就会出现\*args 的第二种情况，因此应该使用下面这种方式：*send\_alter02(**'ryan'**,25,\*\*dic) *#结果为：ryan (25,) {'job': 'IT', 'num': 1212}*

总结：在使用非关键参数时，一定要注意实参的类型，以及传参时的表示。

### 2.3 函数返回值

**def** func01(user):  
 **while True**:  
 **if** user == **'ryan'**:  
 print(**"welcome"**)  
 **return True  
 else**:  
 print(**"打印错误"**)  
 **return False** print(**"不会打印这句话"**)  
  
res = func01(**'ryan'**)  
**if** res:  
 print(**"返回True"**)  
**else**:  
 print(**"返回False"**)

总结：函数的返回值主要用于一些程序用于接受参数的返回值，不用打印在终端时使用。如果不写return的返回值，默认返回None，如果写了，可以返回任何数据类型（包括 True/False）。并且从上列可以看出，return还可以用于跳出while的循环中断程序执行，就像最后一句话一样不会打印。

注意：同一个缩进代码段只执行第一个return,后面的无效。print 和 return 不要搞混淆print 只会在终端打印，并不会有返回值。

### 2.4局部变量

局部变量，就是定义在函数里的变量，只能在局部生效。  
在函数内部可以引用全局变量。  
如果全局和局部都有一个name变量，函数查找变量的顺序是由内而外的。

在函数内部声明全局变量使用关键字（global），要声明一个局部变量为全局变量，必须要让被声明的局部变量赋值在global后面。

name = **"ryan"** *#全局变量***def** change\_name():  
 **global** age *#声明全局变量*  
 age = 22 *#局部变量赋值，被global声明为全局变量*  
 name = **'cherry'** *#局部变量* print(**"name"**,name)  
change\_name()  
print(age)

### 2.5 嵌套函数

age = 19  
**def** func01():  
 **global** age *#==>等价于 把全局变量age=19 放到此处，然后此处的age还是全局变量* **def** func02():  
 print(age)  
 age = 73 *#==>此处的age还是全局变量，并且 age = 73* func02()  
func01()  
print(age)  
**'''**

**注意，函数内部变量查找是按照一层一层的往上找，并且要注意代码的执行顺序。  
从上面可以看出，在函数内部还是可以修改全局变量的。并且在函数内的局部变量的位置也是很有讲究的。  
注重理解 global 以及局部变量的位置，就可以很好的利用嵌套函数了。但是逻辑稍微复杂，慎用！！！  
'''**

### 2.6 函数作用域

在python中，一个函数就是一个作用域。

所有的局部变量就是放在当前作用域的函数里边。

代码定义完成以后，作用域已经生产，作用域链向上查找

### 2.7 匿名函数

匿名函数  
匿名函数作用：1.节省代码量。2.看着更优雅。

res = **lambda** x,y:x\*\*y *#声明一个匿名函数*res2 = **lambda** x,y: x\*y **if** x < y **else** x - y  
print(res(2,3))  
print(res2(5,3))  
data = list(range(1,10))  
print(list(map(**lambda** x:x\*x,data)))

### 2.8 高阶函数

变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另外一个函数作为参数，这种函数称之为高阶函数。

只需要满足以下任意一个条件，即是高阶函数：

1. 接受一个或多个函数作为输入
2. return 返回另外一个函数

### 2.9 递归

递归的层数是有限的（1000层：sys.getrecursionlimit()）。

对某个变量或者程序需要重复执行时就可以用到递归。

递归的执行过程：

**def** calc(n):  
 n = int(n/2)  
 print(n)  
 **if** n > 0:  
 calc(n)  
 print(n)  
calc(10)  
  
**'''  
执行顺序： calc(10)-->calc(5)-->calc(2)-->calc(1)-->calc(0)-->calc(0)-->calc(1)-->calc(2)-->calc(5)-->calc(10)—>退出  
"calc(0)-->calc(1)-->calc(2)-->calc(5)" 这里的意思是（结合函数的后面两句代码），当n=0时，打印print（0）。但是递归的时候的函数执行  
相当于是一层函数包含着一层函数，因此，退出的时候就会去执行获得 n=0 时 calc(n)里面n的值，此时n=1，这里在补充一点，此时的函数不在是往下递归，  
而是去退出整个函数，因此，在退出一层函数，也就是当 n=1 时，calc(n) 里面n的值是多少，再去打印n值。以此类推。**

递归返回值：

**def** calc(n,count):  
 print(n,count)  
 **if** count < 5:  
 **return** calc(n/2,count+1)  
 **else**:  
 **return** n  
res = calc(100,1)  
print(res)

**'''  
递归的返回值：两个return 缺一不可。因为，如果只有一个else 的return，在返回的时候就会退到上一层，但是上一层没有返回值。因此，就会得到一个None的值。  
如果没有else的return，那么当count=5 以后就开始退出函数了，也就更没有返回值了。这里可以根据上面（特指递归的执行顺序的例子）的的递归顺序去理解，  
之所以会退出，是因为上面的n=0，然后一直往回退。因此，在count=5时返回给上一层以后，count=4 但是 此时没有返回值，所以更不行！  
'''**

递归的特性：

1. 必须要有一个明确的结束条件
2. 每次进入更深一层递归时，问题规模相比上一次递归都应有所减少。
3. 递归效率不高，递归层次过多会导致栈溢出（在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出）

### 2.10内置函数

**数学相关**

* abs(a) : 求取绝对值。abs(-1)
* max(list) : 求取list最大值。max([1,2,3])
* min(list) : 求取list最小值。min([1,2,3])
* sum(list) : 求取list元素的和。 sum([1,2,3]) >>> 6
* sorted(list) : 排序，返回排序后的list。
* len(list) : list长度,len([1,2,3])
* divmod(a,b): 获取商和余数。 divmod(5,2) >>> (2,1)
* pow(a,b) : 获取乘方数。pow(2,3) >>> 8
* round(a,b) : 获取指定位数的小数。a代表浮点数，b代表要保留的位数。round(3.1415926,2) >>> 3.14
* range(a[,b]) : 生成一个a到b的数组,左闭右开。 range(1,10) >>> [1,2,3,4,5,6,7,8,9]

**类型转换**

* int(str) : 转换为int型。int('1') >>> 1
* float(int/str) : 将int型或字符型转换为浮点型。float('1') >>> 1.0
* str(int) : 转换为字符型。str(1) >>> '1'
* bool(int) : 转换为布尔类型。 str(0) >>> False str(None) >>> False
* bytes(str,code) : 接收一个字符串，与所要编码的格式，返回一个字节流类型。bytes('abc', 'utf-8') >>> b'abc' bytes(u'爬虫', 'utf-8') >>> b'\xe7\x88\xac\xe8\x99\xab'
* list(iterable) : 转换为list。 list((1,2,3)) >>> [1,2,3]
* iter(iterable)： 返回一个可迭代的对象。 iter([1,2,3]) >>> <list\_iterator object at 0x0000000003813B00>
* dict(iterable) : 转换为dict。 dict([('a', 1), ('b', 2), ('c', 3)]) >>> {'a':1, 'b':2, 'c':3}
* enumerate(iterable) : 返回一个枚举对象。
* tuple(iterable) : 转换为tuple。 tuple([1,2,3]) >>>(1,2,3)
* set(iterable) : 转换为set。 set([1,4,2,4,3,5]) >>> {1,2,3,4,5} set({1:'a',2:'b',3:'c'}) >>> {1,2,3}
* hex(int) : 转换为16进制。hex(1024) >>> '0x400'
* oct(int) : 转换为8进制。 oct(1024) >>> '0o2000'
* bin(int) : 转换为2进制。 bin(1024) >>> '0b10000000000'
* chr(int) : 转换数字为相应ASCI码字符。 chr(65) >>> 'A'
* ord(str) : 转换ASCI字符为相应的数字。 ord('A') >>> 65

**相关操作**

* eval() : 执行一个表达式，或字符串作为运算（只能处理单行）。 eval('1+1') >>> 2
* exec() : 执行python语句（拿不到返回值）。 exec('print("Python")') >>> Python
* filter(func, iterable) : 通过判断函数fun，筛选符合条件的元素。 filter(lambda x: x>3, [1,2,3,4,5,6]) >>> <filter object at 0x0000000003813828>
* map(func, \*iterable) : 将func用于每个iterable对象。 map(lambda a,b: a+b, [1,2,3,4], [5,6,7]) >>> [6,8,10]
* zip(\*iterable) : 将iterable分组合并。返回一个zip对象。 list(zip([1,2,3],[4,5,6])) >>> [(1, 4), (2, 5), (3, 6)]
* type()：返回一个对象的类型。
* id()： 返回一个对象的唯一标识值。
* hash(object)：返回一个对象的hash值，具有相同值的object具有相同的hash值。 hash('python') >>> 7070808359261009780
* help()：调用系统内置的帮助系统。
* isinstance()：判断一个对象是否为该类的一个实例。
* issubclass()：判断一个类是否为另一个类的子类。
* globals() : 返回当前全局变量的字典。
* next(iterator[, default]) : 接收一个迭代器，返回迭代器中的数值，如果设置了default，则当迭代器中的元素遍历后，输出default内容。
* reversed(sequence) ： 生成一个反转序列的迭代器。 reversed('abc') >>> ['c','b','a']
* vars() : 打印当前变量名以及环境变量
* locals() ：打印函数里面当前的局部变量

### 2.11 函数进阶-命名空间

名称空间共3种，分别如下

* locals: 是函数内的名称空间，包括局部变量和形参
* globals: 全局变量，函数定义所在模块的名字空间
* builtins: 内置模块的名字空间 #查看方法：dir(\_\_builtins\_\_)

**不同变量的作用域不同就是由这个变量所在的命名空间决定的。**

作用域即范围

* 全局范围：全局存活，全局有效
* 局部范围：临时存活，局部有效

查看作用域方法 globals(),locals()

### 2.12 函数进阶-作用域查找

LEGB 代表名字查找顺序: locals -> enclosing function -> globals -> \_\_builtins\_\_

* locals 是函数内的名字空间，包括局部变量和形参
* enclosing 外部嵌套函数的名字空间
* globals 全局变量，函数定义所在模块的名字空间
* builtins 内置模块的名字空间

### 2.13 函数进阶-闭包

关于闭包，即函数定义和函数表达式位于另一个函数的函数体内(嵌套函数)。而且，这些内部函数可以访问它们所在的外部函数中声明的所有局部变量、参数。当其中一个这样的内部函数在包含它们的外部函数之外被调用时，就会形成闭包。也就是说，内部函数会在外部函数返回后被执行。而当这个内部函数执行时，它仍然必需访问其外部函数的局部变量、参数以及其他内部函数。这些局部变量、参数和函数声明（最初时）的值是外部函数返回时的值，但也会受到内部函数的影响。

def outer():

name = 'alex'

def inner():

print("在inner里打印外层函数的变量",name)

return inner

f = outer()

f()

闭包的意义：**返回的函数对象，不仅仅是一个函数对象，在该函数外还包裹了一层作用域，这使得，该函数无论在何处调用，优先使用自己外层包裹的作用域**

### 2.14 装饰器

装饰器是函数，只不过该函数可以具有特殊的含义，装饰器用来装饰函数或类，使用装饰器可以在函数执行前和执行后添加相应操作。

**装饰器原理：**

python允许将方法当做参数传递。

开放封闭原则；

* 封闭：已实现的功能代码块
* 开放：对扩展开发

1，装饰器就是一个函数，至少2层。

2，装饰器的特征是  @ 加上装饰器的名称。添加装饰器的函数会传入到装饰器里面作为参数。装饰器里面函数的返回值，重新赋值给被装饰的函数的函数名。

3，动态参数，可以装饰含有n个参数的函数。

4，函数返回值

5，多装饰器

6，至少三层，三层如何使用

**装饰器好处：**

装饰器的好处就是，不改变源代码而去实现一个新功能。

例1：这是装饰器的一个逐渐进化的过程，用于很好的理解装饰器的用法。

注：函数只有被调用的时候才能被解释器解释，不然，解释器只会把函数的名称放在内存里面，而不去执行函数内部的代码。

*#!/usr/bin/env python3  
#coding:utf-8***def** auth(func):  
 **def** user\_auth():  
 print(**'before'**)  
 func()  
 **return** user\_auth  
**def** one():  
 print(**'one'**)

temp = auth(one)  
one = temp  
  
**def** two():  
 print(**'two'**)  
two = auth(two)  
  
@auth  
**def** three():  
 print(**'three'**)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 print(**'-----one method-----'**)  
 one()  
 print(**'-----two method-----'**)  
 two()  
 print(**'-----three method-----'**)  
 three()  
   
**'''  
[root@js-93 wrap]# python3 exam02.py  
-----one method-----  
before  
one  
-----two method-----  
before  
two  
-----three method-----  
before  
three  
  
代码理解：可以看出三种方式都相当于装饰器的效果，明显的可以看出第三种方法（@auth），简单明了，而且易用性很高。  
整体代码解释：  
当执行整个脚本的时候，是从上到下依次执行的，当执行到函数的时候，这个时候并没有在程序中调用该函数，因此，解释器会把该函数名称先读取到内存中，而不去读取函数里面代码的过程。依次往下读直到13,14行。  
第一种方法：  
当读到13行和14 行的时候，“temp = auth(one)”，这个时候已经调用了函数“auth”,并且传递参数“one”，因此，解释器会去执行该函数。执行auth函数的时候，先执行"return user\_auth"(前面提到过，只有函数被被调用的时候才能执行函数，后面不在提及)，  
然后执行“user\_auth”函数，会打印“before”，然后在调用“fun()”函数。函数“func”在去执行“print('one')”，就得带了后面 “one method”的结果。这里要注意的是，在函数“one（）”函数下面的 temp = auth(one) 以及 one = temp的含义；  
下面两句的含义:  
temp = auth(one)   
one = temp  
这两句的含义，其实等价于 方法2 的 two = auth(two)，也就是 one = auth(one) 此时的one不在执行 print('one')了，因为，one已经被重新定义了，他会去执行auth(one)。  
第二种方法：  
第二种方法和第一种方法理解基本一样。  
第三种方法：（理解于：http://www.cnblogs.com/wupeiqi/articles/4980620.html）  
当写完代码以后，函数未被执行，Python解释器就会从上到下解释代码，步骤如下：  
1. def auth(func): --->将auth函数加载到内存。  
2. @auth   
如果把代码分离，只剩下第三种方法，那么，从代码表面看，解释器仅仅会执行上面2个过程，因为，函数没有被调用之前，其内部代码不会被执行。  
从表面上看解释器确实会执行这两句，但是 @auth 这一句代码里面却有大文章，@函数名 是python的一种语法糖。  
从上例@auth内部会执行一下操作：  
执行auth函数，并将 @auth下面的函数作为auth函数的参数，即：@auth 等价于 auth（three）  
所以，内部会去执行：  
 def auth(three):  
 def user\_auth():  
 print('before')  
 three()  
 return user\_auth  
将执行完的auth函数返回值 赋值给 @auth 下面的 函数的函数名。  
auth函数的返回值是，print brfore 以及 执行，three()  
这就是装饰器的进化过程。  
'''**

### 2.5 装饰器带一个参数

*#!/usr/bin/env python  
#coding:utf-8***def** auth(func):  
 **def** user\_auth(arg):  
 print(**'--before--'**)  
 func(arg)  
 print(**'--after--'**)  
 **return** user\_auth  
  
@auth  
**def** one(arg):  
 print(**'I'**,arg)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 one(**"love you"**)  
**'''  
[root@js-93 wrap]# python3 exam03.py  
--before--  
I love you  
--after--  
运行过程 查看例1，这个主要注意的是，当被装饰的函数带一个参数的时候，装饰器应该怎么写。  
'''**

### 2.6 装饰器之非固定参数

*#主要看多参数的用法。  
#!/usr/bin/env python3  
#coding:utf-8***def** auth(func):  
 **def** user\_auth(\*arg,\*\*kwargs):  
 print(**'--before--'**)  
 func(\*arg,\*\*kwargs)  
 print(**'--after--'**)  
 **return** user\_auth  
  
@auth  
**def** test(\*args,\*\*kwargs):  
 print(**'hello--'**,args,kwargs)  
  
L1 = [1,2,3,4]  
  
info ={  
 **"name"**:**'root'**,  
 **"pwd"**:**'1234'**}  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
  
 print(**'-----List----------'**)  
 test(\*L1)  
 print(**'-----Dict----------'**)  
 test(\*\*info)  
 print(**'-----double----------'**)  
 test(\*L1,\*\*info)  
**'''  
[root@js-93 wrap]# python3 exam04.py  
-----List----------  
--before--  
hello-- (1, 2, 3, 4) {}  
--after--  
-----Dict----------  
--before--  
hello-- () {'name': 'root', 'pwd': '1234'}  
--after--  
-----double----------  
--before--  
hello-- (1, 2, 3, 4) {'name': 'root', 'pwd': '1234'}  
--after--  
'''**

<http://www.cnblogs.com/keep-going2099/articles/6713178.html>

后面总结（包括教程里面的登录认证）

### 2.7 生成器

**列表生成式**

**方法1：**

>>> a

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

>>> b = []

>>> for i in a:

b.append(i+1)

...

>>> b

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

>>> a = b

>>> a

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

方法2：

a = [1,3,4,6,7,7,8,9,11]

for index,i in enumerate(a):

a[index] +=1

print(a)

方法3：

>>> a

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

>>> a = map(lambda x:x+1, a)

>>> a

<map object at 0x101d2c630>

>>> for i in a:print(i)

...

3

5

7

9

11

方法4：

>>> a

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

>>> a = [i+1 for i in a]

>>> a

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

>>> a

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

>>> a = [i if i > 5 else i\*i for i in a]

>>> a

[0, 1, 4, 9, 16, 25, 6, 7, 8, 9]

生成器：用next打印完最后一个过后如果继续打印 会报错（StopIteration）；

>>> a = (i for i in range(10))

>>> a

<generator object <genexpr> at 0x7f58e3ffc0f8>

>>> next(a)

0

>>> next(a)

1

>>> for i in a:

... print(i)

...

2

3

...

9

python3:

>>> range(10)

range(0, 10)

python2:

>>> range(10)

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

range在python3 优于python2 的好处在于，如果数据需要足够大，那么python3中消耗的内存地址则较少。

### 2.8 生成器之斐波拉契数列

著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：0,1, 1, 2, 3, 5, 8, 13, 21, 34, …

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

def fib(max):

n, a, b = 0, 0, 1

while n < max:

print(b)

a, b = b, a + b

n = n + 1

return 'done'

提示：a是斐波拉契的第一个值，b是斐波那契的第二个值，n是计数

要把fib函数变成generator，只需要把print(b)改为yield b就可以了：

def fib(max):

n,a,b = 0,0,1

while n < max:

#print(b)

yield b #

a,b = b,a+b

n += 1

return 'done'

yield b 这里的意思是把函数的执行过程冻结在这一步，并且把b的值 返回给外面的next()

生成器的创建方式：

1. 列表生成式 a = (i for i in range(10))
2. 函数写生成器（yield）

### 2.9 函数写生成器以及send方法

return vs yield 的区别：

return：返回并终止函数

yield：返回数据，并冻结当前函数的执行过程，直到next 唤醒冻结的函数执行过程，继续执行，直到遇到下一个yield。

**def** range2(n):  
  
 count = 0  
 **while** count < n:  
 print(**'count'**,count)  
 count += 1  
 sign = **yield** count**if** sign == **'stop'**:  
 print(**"---sign"**, sign)  
 **break** print(**'sin...'**,sign)  
 **return** 3333  
  
new\_range = range2(3)  
  
*# n1=next(new\_range)*next(new\_range)  
new\_range.send(**None**) #这里就相当于next，只是next发送了一个none的值过去。  
*#new\_range.send("stop")  
#1. 唤醒并继续执行  
#2. 发送一个信息到生成器内部***'''  
python 2  
  
 range = list  
 xrange = 生成器  
  
python 3  
 range = 生成器  
 xrange 没有  
  
生成器的创建方式  
 1. 列表 生成 式（）  
 2. 函数  
  
  
yield vs return  
 return 返回 并中止function  
 yield 返回 数据 ，并冻结当前的执行过程 。。。  
 next 唤醒冻结的函数执行过程，继续执行，直到遇到下一个yield**

**函数有了yield之后  
 1. 函数名加()就变得到了一个生成器，  
 2. return 在生成器里， 代表 生成器的中止， 直接 报错  
  
next  
 唤醒 生成器并继续 执行  
send("stop")  
 #1. 唤醒并继续执行  
 #2. 发送一个信息到生成 器内部  
'''**

### 2.10 迭代器

可以直接作用于for循环的数据类型有以下几种：

一类是集合数据类型，如list、tuple、dict、set、str等；

一类是generator，包括生成器和带yield的generator function。

这些可以直接作用于for循环的对象统称为**可迭代对象**：Iterable。

可以使用isinstance()判断一个对象是否是Iterable对象：

>>> from collections import Iterable

>>> isinstance([], Iterable)

True

>>> isinstance({}, Iterable)

True

>>> isinstance('abc', Iterable)

True

>>> isinstance((x for x in range(10)), Iterable)

True

>>> isinstance(100, Iterable)

False

而生成器不但可以作用于for循环，还可以被next()函数不断调用并返回下一个值，直到最后抛出StopIteration错误表示无法继续返回下一个值了。

**\*可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator。**

可以使用isinstance()判断一个对象是否是Iterator对象：

>>> from collections import Iterator

>>> isinstance((x for x in range(10)), Iterator)

True

>>> isinstance([], Iterator)

False

>>> isinstance({}, Iterator)

False

>>> isinstance('abc', Iterator)

False

生成器都是Iterator对象，但list、dict、str虽然是Iterable，却不是Iterator。

把list、dict、str等Iterable变成Iterator可以使用iter()函数：

>>> isinstance(iter([]), Iterator)

True

>>> isinstance(iter('abc'), Iterator)

True

你可能会问，为什么list、dict、str等数据类型不是Iterator？

这是因为Python的Iterator对象表示的是一个数据流，Iterator对象可以被next()函数调用并不断返回下一个数据，直到没有数据时抛出StopIteration错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过next()函数实现按需计算下一个数据，所以Iterator的计算是惰性的，只有在需要返回下一个数据时它才会计算。

Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的。

小结：

凡是可作用于for循环的对象都是Iterable类型（对象）；

凡是可作用于next()函数的对象都是Iterator类型，它们表示一个惰性计算的序列；

集合数据类型如list、dict、str等是Iterable但不是Iterator，不过可以通过iter()函数获得一个Iterator对象。

Python3的for循环本质上就是通过不断调用next()函数实现的，例如：

for x in [1, 2, 3, 4, 5]:

pass

实际上完全等价于：

# 首先获得Iterator对象:

it = iter([1, 2, 3, 4, 5])

# 循环:

while True:

try:

# 获得下一个值:

x = next(it)

except StopIteration:

# 遇到StopIteration就退出循环

break

## 第三章 常用模块

### 3.1模块介绍、调用、查找等

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。在Python中，一个.py文件就称之为一个模块（Module）。

好处:

1. 最大的好处是大大提高了代码的可维护性
2. 可以重复使用
3. 使用模块还可以避免函数名和变量名冲突

模块分为三种：

* 内置标准模块（又称标准库）执行**help('modules')**查看所有python自带模块列表
* 第三方开源模块，可通过pip install 模块名 联网安装
* 自定义模块

模块调用:

import module

from module import xx #导入模块里面的指定方法

例如：from os import rmdir,makedirs (也可以指定导入两个方法)

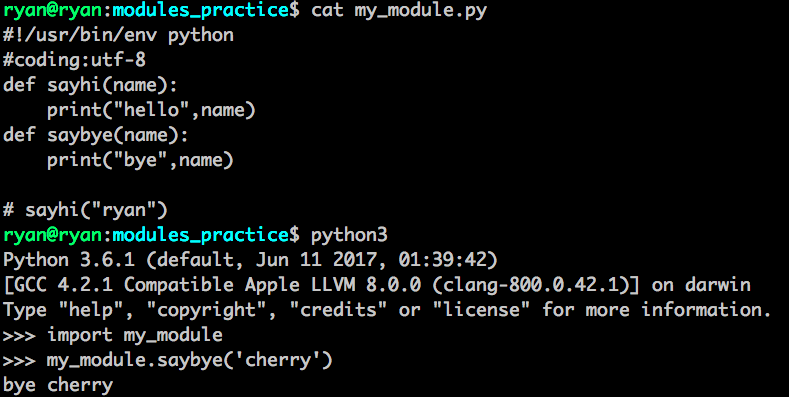
from module.xx.xx import xx as rename #导入指定模块的方法，然后取别名。

from module.xx.xx import \* #调用时不加模块名，只用方法名。（不推荐，可能会冲突）

注意：模块一旦被调用，即相当于执行了另外一个py文件里的代码

### 3.2自定义模块

1.自定模块导入（当前模块所在目录）



2.查看环境的模块

>>> import sys

>>> print(sys.path)

['', '/usr/local/Cellar/python3/3.6.1/Frameworks/Python.framework/Versions/3.6/lib/python36.zip', '/usr/local/Cellar/python3/3.6.1/Frameworks/Python.framework/Versions/3.6/lib/python3.6', '/usr/local/Cellar/python3/3.6.1/Frameworks/Python.framework/Versions/3.6/lib/python3.6/lib-dynload', '/usr/local/lib/python3.6/site-packages']

'/usr/local/lib/python3.6/site-packages' #标准库，第三方和内置库都在里面。

查找模块的方法是从环境变量的第一个元素开始找，找到就停止。（列表有序）

因此要使用上面那个模块（my\_module.py），则需要把它导入到环境变量里面。

import sys

sys.path.append(“自定义路径”)

import 模块(自定义路径下的py文件名)



### 3.3 开源模块的方法

<https://pypi.python.org/pypi/PyTyrion/1.0.1>（开源软件的一个例子<吴佩琪>）

可以用下载源码包安装：

然后执行：python3 setup.py build (编译)

python3 setup.py install （安装）

pip使用

pip freeze # 查看包版本

pip install Package # 安装包 pip install requests

pip show --files Package # 查看安装包时安装了哪些文件

pip show --files Package # 查看哪些包有更新

pip install --upgrade Package # 更新一个软件包

pip uninstall Package # 卸载软件包

pip list # 查看pip安装的包及版本

pip install django==1.5 # 指定版本安装

pip命令默认会连接在国外的python官方服务器下载，速度比较慢，你还可以使用国内的豆瓣源，数据会定期同步国外官网，速度快好多

pip install -i http://pypi.douban.com/simple/ alex\_sayhi（模块名） --trusted-host pypi.douban.com

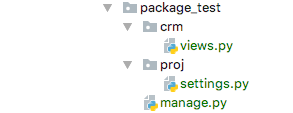
### 3.3 包

包就是文件夹，但该文件夹下必须存在 \_\_init\_\_.py 文件, 该文件的内容可以为空。\_\_int\_\_.py用于标识当前文件夹是一个包。

python2 必须要 \_\_init\_\_.py

python3 可以不用 \_\_init\_\_.py ，但是基本都加上

目录结构：



--------------------分割线-----------------

manage.py

*#!/usr/bin/env python  
#coding:utf-8***from** crm **import** views  
views.sayhi()

--------------------分割线-----------------

setting.py

*#!/usr/bin/env python  
#coding:utf-8*DATABASES = {  
 **'host'**:**'localhost'**}  
  
print(**"in proj/setting.py"**)

--------------------分割线-----------------

*#!/usr/bin/env python  
#coding:utf-8***from** proj **import** settings  
  
**def** sayhi():  
 print(**"hello world"**)

执行结果：

ryan@ryan:package\_test$ python3 manage.py

in proj/setting.py

hello world

注意：可以看到能够调用到 proj下面的settings里面的变量。为什么呢？

因此此时的入口是 manage.py 是和 proj是同一个目录的，但是如果执行views.py文件，那么就会报错。

ryan@ryan:crm$ python3 views.py

Traceback (most recent call last):

File "views.py", line 4, in <module>

from proj import settings

ModuleNotFoundError: No module named 'proj'

因此模块（包）的导入，一定要注意入口文件在哪里，以及要被调用的模块（包）的路径。

如果一定要执行 views.py 文件，那么用下面方法。

### 3.4 相对路径导入模块

第一种方法：直接用相对路径。

*#!/usr/bin/env python  
#coding:utf-8***import** sys  
*#相对路径导入环境变量*sys.path.append(**'../../package\_test'**)

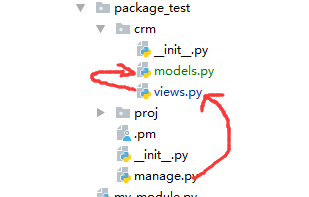
**from** proj **import** settings  
**def** sayhi():  
 print(**"hello world"**)

第二种方法：调用 dir() 下面的方法：“\_\_file\_\_”的属性去获取当前文件所在的绝对路径。

*#!/usr/bin/env python  
#coding:utf-8***import** sys,os *#"\_\_file\_\_"表示的是当前py文件文件名，然后用os.path.abspath获取绝对路径，然后，用os.path.dirname获取相对路径即可，这两个相结合就能导入需要导入的模块路径。*BASE\_DIR = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)

))  
sys.path.append(BASE\_DIR)  
**from** proj **import** settings  
  
**def** sayhi():  
 print(**"hello world"**)

注意：如果在当前目录有A,B两个python文件时，一定不要直接用import（A.py里面导入B.py），如果直接导入，在别的入口文件进行调用A时，就会报错。因此要使用相对路径。例如：



models.py

*#!/usr/bin/env python  
#coding:utf-8*  
**def** model():  
 print(**"crm/model.py"**)

views.py

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

**from** . **import** models  
**def** sayhi():  
 print(**"hello world"**)  
models.model()

manage.py

*#!/usr/bin/env python  
#coding:utf-8***from** crm **import** views  
  
views.sayhi()

导入的时候要用 **from** . **import** models 或者**from** crm **import** models ，虽然，views.py 和 models.py在同一个目录下，执行views.py文件虽然可以执行，但是，当入口文件为manage.py时，就会报错。

在涉及到相对导入时，package所对应的文件夹必须正确的被python解释器视作package，而不是普通文件夹。否则由于不被视作package，无法利用package之间的嵌套关系实现python中包的相对导入。

文件夹被python解释器视作package需要满足两个条件：

1. **文件夹中必须有\_\_init\_\_.py文件，该文件可以为空，但必须存在该文件。**
2. **不能作为顶层模块来执行该文件夹中的py文件（即不能作为主函数的入口）。**
3. **顶层py入口文件放在所有包的最顶层。**

### 3.5 time & datetime模块

在Python中，通常有这几种方式来表示时间：

1. 时间戳
2. 格式化的时间字符串
3. 元组（struct\_time）共九个元素。由于Python的time模块实现主要调用C库，所以各个平台可能有所不同。

时间戳（timestamp）的方式：通常来说，时间戳表示的是从1970年1月1日00:00:00开始按秒计算的偏移量。我们运行“type(time.time())”，返回的是float类型。

元组（struct\_time）方式：struct\_time元组共有9个元素，返回struct\_time的函数主要有gmtime()，localtime()，strptime()。下面列出这种方式元组中的几个元素：

索引（Index） 属性（Attribute） 值（Values）

0 tm\_year（年） 比如2011

1 tm\_mon（月） 1 - 12

2 tm\_mday（日） 1 - 31

3 tm\_hour（时） 0 - 23

4 tm\_min（分） 0 - 59

5 tm\_sec（秒） 0 - 61

6 tm\_wday（weekday） 0 - 6（0表示周日）

7 tm\_yday（一年中的第几天） 1 - 366

8 tm\_isdst（是否是夏令时） 默认为-1

**time模块的方法**

* time.localtime([secs])：将一个时间戳转换为当前时区的struct\_time。secs参数未提供，则以当前时间为准。
* time.gmtime([secs])：和localtime()方法类似，gmtime()方法是将一个时间戳转换为UTC时区（0时区）的struct\_time。
* time.time()：返回当前时间的时间戳。
* time.mktime(t)：将一个struct\_time转化为时间戳。
* time.sleep(secs)：线程推迟指定的时间运行。单位为秒。
* time.asctime([t])：把一个表示时间的元组或者struct\_time表示为这种形式：'Sun Oct 1 12:04:38 2017'。如果没有参数，将会将time.localtime()作为参数传入。
* time.ctime([secs])：把一个时间戳（按秒计算的浮点数）转化为time.asctime()的形式。如果参数未给或者为None的时候，将会默认time.time()为参数。它的作用相当于time.asctime(time.localtime(secs))。
* time.strftime(format[, t])：把一个代表时间的元组或者struct\_time（如由time.localtime()和time.gmtime()返回）转化为格式化的时间字符串。如果t未指定，将传入time.localtime()。
  + 举例：time.strftime("%Y-%m-%d %X", time.localtime()) #输出'2017-10-01 12:14:23'
* time.strptime(string[, format])：把一个格式化时间字符串转化为struct\_time。实际上它和strftime()是逆操作。
  + 举例：time.strptime('2017-10-3 17:54',"%Y-%m-%d %H:%M") #输出 time.struct\_time(tm\_year=2017, tm\_mon=10, tm\_mday=3, tm\_hour=17, tm\_min=54, tm\_sec=0, tm\_wday=1, tm\_yday=276, tm\_isdst=-1)

**datetime模块**

相比于time模块，datetime模块的接口则更直观、更容易调用

**datetime模块定义了下面这几个类：**

* datetime.date：表示日期的类。常用的属性有year, month, day；
* datetime.time：表示时间的类。常用的属性有hour, minute, second, microsecond；
* datetime.datetime：表示日期时间。
* datetime.timedelta：表示时间间隔，即两个时间点之间的长度。
* datetime.tzinfo：与时区有关的相关信息。（这里不详细充分讨论该类，感兴趣的童鞋可以参考python手册）

**我们需要记住的方法仅以下几个：**

1.d=datetime.datetime.now() 返回当前的datetime日期类型

d.timestamp(),d.today(), d.year,d.timetuple()等方法可以调用

2.datetime.date.fromtimestamp(322222) 把一个时间戳转为datetime日期类型

**3.时间运算**

>>> datetime.datetime.now()

datetime.datetime(2017, 10, 1, 12, 53, 11, 821218)

>>> datetime.datetime.now() + datetime.timedelta(4) #当前时间 +4天

datetime.datetime(2017, 10, 5, 12, 53, 35, 276589)

>>> datetime.datetime.now() + #当前时间+4小时

datetime.datetime(2017, 10, 1, 16, 53, 42, 876275)

还支持分钟和秒（datetime.timedelta(minutes=4) datetime.timedelta(seconds=4)）

4.时间替换

>>> d = datetime.datetime.now()

>>> d

datetime.datetime(2017, 12, 19, 13, 22, 39, 157418)

>>> d.replace(month=10)

datetime.datetime(2017, 10, 19, 13, 22, 39, 157418)

>>> d.replace(day=10)

datetime.datetime(2017, 12, 10, 13, 22, 39, 157418)

>>> d.replace (year=2999,month=11,day=30)

datetime.date(2999, 11, 30)

### 3.6 random模块详解

程序中有很多地方需要用到随机字符，比如登录网站的随机验证码，通过random模块可以很容易生成随机字符串

>>> random.randomrange(1,10) #返回1-10之间的一个随机数，不包括10

>>> random.randomint(1,10) #返回1-10之间的一个随机数，包括10

>>> random.randrange(0, 100, 2) #随机选取0到100间的偶数

>>> random.random() #返回一个随机浮点数

>>> random.choice('abce3#$@1') #返回一个给定数据集合中的随机字符

'#'

>>> random.sample('abcdefghij',3) #从多个字符中选取特定数量的字符

['a', 'd', 'b']

#生成随机字符串

>>> import string

>>> ''.join(random.sample(string.ascii\_lowercase + string.digits, 6))

'4fvda1'

#洗牌

>>> a

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

>>> random.shuffle(a)

>>> a

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

### 3.7 os 模块详解

os 模块提供了很多允许你的程序与操作系统直接交互的功能

得到当前工作目录，即当前Python脚本工作的目录路径: os.getcwd() <执行脚本或者说是python解释器的目录>

返回指定目录下的所有文件和目录名:os.listdir()

函数用来删除一个文件:os.remove()

删除多个目录：os.removedirs（r“c：\python”）

检验给出的路径是否是一个文件：os.path.isfile()

检验给出的路径是否是一个目录：os.path.isdir()

判断是否是绝对路径：os.path.isabs()

检验给出的路径是否真地存在:os.path.exists()

返回一个路径的目录名和文件名:os.path.split() e.g os.path.split('/home/swaroop/byte/code/poem.txt') 结果：('/home/swaroop/byte/code', 'poem.txt')

分离扩展名：os.path.splitext() e.g os.path.splitext('/usr/local/test.py') 结果：('/usr/local/test', '.py')

获取路径名：os.path.dirname()

获得绝对路径: os.path.abspath()

获取文件名：os.path.basename()

运行shell命令: os.system() #返回一个状态码，返回正确“0”，不正确 “非0”

读取操作系统环境变量HOME的值:os.getenv("HOME")

返回操作系统所有的环境变量： os.environ

设置系统环境变量，仅程序运行时有效：os.environ.setdefault('HOME','/home/alex')

给出当前平台使用的行终止符:os.linesep Windows使用'\r\n'，Linux and MAC使用'\n'

指示你正在使用的平台：os.name 对于Windows，它是'nt'，而对于Linux/Unix用户，它是'posix'

重命名：os.rename（old， new）

创建多级目录：os.makedirs（r“c：\python\test”）

创建单个目录：os.mkdir（“test”）

获取文件属性：os.stat（file）

修改文件权限与时间戳：os.chmod（file）

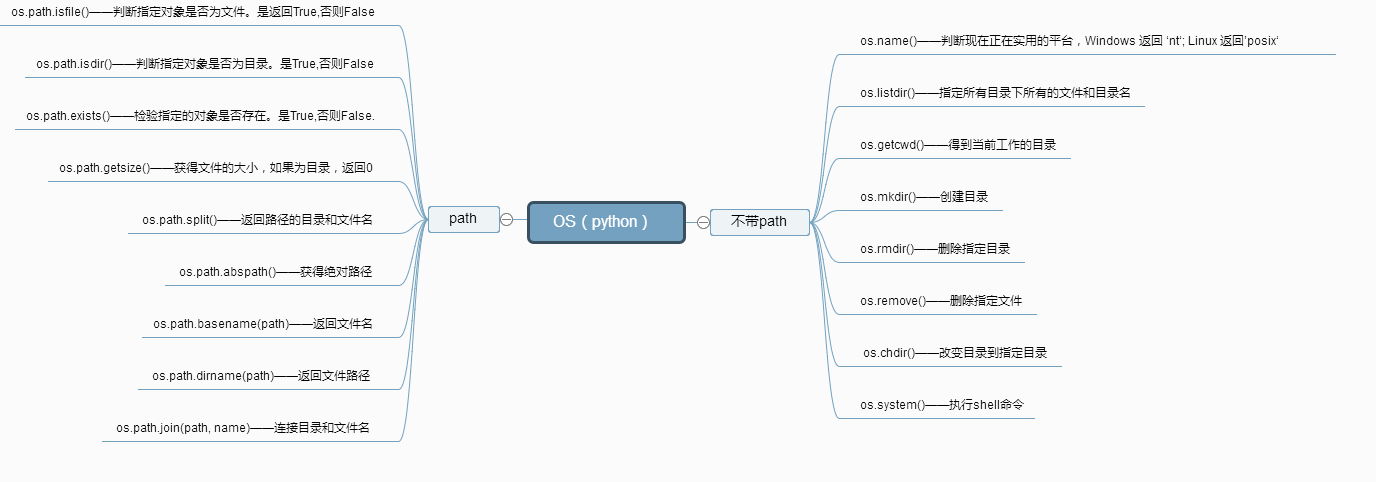
获取文件大小：os.path.getsize（filename）

结合目录名与文件名：os.path.join(dir,filename)

改变工作目录到dirname: os.chdir(dirname)

获取当前终端的大小: os.get\_terminal\_size()

杀死进程: os.kill(10884,signal.SIGKILL)



### 3.8 sys模块

sys.argv 命令行参数List，第一个元素是程序本身路径

sys.exit(n) 退出程序，正常退出时exit(0)

sys.version 获取Python解释程序的版本信息

sys. maxsize 最大的Int值

sys.path 返回模块的搜索路径，初始化时使用PYTHONPATH环境变量的值

sys.platform 返回操作系统平台名称

sys.stdout.write('please:') #标准输出 , 引出进度条的例子， 注，在py3上不行，可以用print代替

val = sys.stdin.readline()[:-1] #标准输入

sys.getrecursionlimit() #获取最大递归层数

sys.setrecursionlimit(1200) #设置最大递归层数

sys.getdefaultencoding() #获取解释器默认编码

sys.getfilesystemencoding #获取内存数据存到文件里的默认编码

### 3.9 shutil 模块

shutil.copyfileobj(fsrc, fdst[, length]) #将文件内容拷贝到另一个文件中，length表示每次读多少然后往fdst文件里面写多少。

shutil.copyfile(src, dst) #拷贝文件

shutil.copymode(src, dst) #仅拷贝权限。内容、组、用户均不变

shutil.copystat(src, dst) #拷贝状态的信息，包括：mode bits, atime, mtime, flags

shutil.copy(src, dst) #拷贝文件和权限

shutil.copy2(src, dst) #拷贝文件和状态信息

shutil.ignore\_patterns(\*patterns)  
shutil.copytree(src, dst, symlinks=False, ignore=None) #递归的去拷贝文件

例如：copytree(source, destination, ignore=ignore\_patterns('\*.pyc', 'tmp\*'))

shutil.rmtree(path[, ignore\_errors[, onerror]]) #递归的去删除文件

shutil.move(src, dst) #递归的去移动文件，相当于重命名目录。

shutil.make\_archive(base\_name, format, ...)

创建压缩包并返回文件路径，例如：zip、tar  
base\_name： 压缩包的文件名，也可以是压缩包的路径。只是文件名时，则保存至当前目录，否则保存至指定路径，  
如：www = > 保存至当前路径  
如： / Users / wupeiqi / www = > 保存至 / Users / wupeiqi /  
format： 压缩包种类，“zip”, “tar”, “bztar”，“gztar”  
root\_dir： 要压缩的文件夹路径（默认当前目录）  
owner： 用户，默认当前用户  
group： 组，默认当前组  
logger： 用于记录日志，通常是logging.Logger对象

用法：shutil.make\_archive(**"/压缩路径/文件名"**,**"格式"**,**"/被压缩的路径/被压缩的文件名"**)

*# 将 /Users/wupeiqi/Downloads/test 下的文件打包放置当前程序目录***import** shutil  
ret = shutil.make\_archive(**"wwwwwwwwww"**, **'gztar'**, root\_dir=**'/Users/wupeiqi/Downloads/test'**)  
  
*# 将 /Users/wupeiqi/Downloads/test 下的文件打包放置 /Users/wupeiqi/目录***import** shutil  
ret = shutil.make\_archive(**"/Users/wupeiqi/wwwwwwwwww"**, **'gztar'**, root\_dir=**'/Users/wupeiqi/Downloads/test')**

注意：shutil 对压缩包的处理是调用 ZipFile 和 TarFile 两个模块来进行的。

Zipfile 压缩解压

import zipfile

# 压缩 (只能压缩文件，不能压缩目录)

z = zipfile.ZipFile('laxi.zip', 'w')

z.write('a.log')

z.write('data.data')

z.close()

# 解压

z = zipfile.ZipFile('laxi.zip', 'r')

z.extractall()

z.close()

tarfile 压缩解压：

import tarfile

# 打包 （递归打包）

tar = tarfile.open('your.tar','w')

tar.add('/home/bbs2', arcname='bbs2.zip')

tar.add('/home/cmdb', arcname='cmdb.zip')

tar.close()

# 解压

tar = tarfile.open('your.tar','r')

tar.extractall() # 可设置解压地址

tar.close()

### 3.10 json & pickle序列化

**什么叫序列化？**

序列化是指把内存里的数据类型转变成字符串，以使其能存储到硬盘或通过网络传输到远程，因为硬盘或网络传输时只能接受bytes

把内存数据转成字符串，叫序列化

把字符转成内存数据类型叫反序列化。

用于序列化的两个模块

* json，用于字符串 和 python数据类型间进行转换
* pickle，用于python特有的类型 和 python的数据类型间进行转换

Json模块提供了四个功能：dumps、dump、loads、load

pickle模块提供了四个功能：dumps、dump、loads、load

只是把数据类型转成字符串存到内存里的意义？

json.dumps json.loads

1. 把你的内存数据通过网络（网络传输类型：bytes）共享给远程其他人
2. 跨语言，跨平台的传递对象。

json和pickle的用法:

json和pickle 的用法基本一样；

**json vs pickle 的区别：**

**JSON:**

优点：跨语言、体积小

缺点：只能支持int\str\list\tuple\dict

区别：dump或者dumps以后是str类型。

**Pickle:**

优点：专为python设计，支持python所有的数据类型（例如函数，）

缺点：只能在python中使用，存储数据占空间大

区别：在dump或者dumps以后是bytes类型，因此打开文件open(‘data.pkl’,’wb’),读文件’rb’

小提示：一般json保存格式为:file.json；pickle 文件的格式为：file.pkl

**dump,dumps,load,loads 的用法：**

*#!/usr/bin/env python  
#coding:utf-8***import** json  
data = {  
 **'roles'**:[  
 {**'role'**:**'monster'**,**'life'**:50},  
 {**'role'**:**'hero'**,**'life'**:80},  
 ]  
}  
**'''dump 转成字符串，然后直接把数据写到磁盘（文件）'''**f = open(**"test.json"**,**'w'**)  
json.dump(data,f)  
f.close()  
  
**'''dumps 仅转成字符串，然后通过文件的写方法，写到文件'''**f = open(**"test.json"**,**'w'**)  
d = json.dumps(data)  
f.write(d)  
f.close()  
  
**'''load 直接读取磁盘内容'''**f = open(**"test.json"**,**'r'**)  
data = json.load(f)  
f.close()  
  
**'''loads 先读取到内存，在从内存读取出来'''**f = open(**"test.json"**,**'r'**)  
d = f.read()  
data = json.loads(d)  
f.close()

特别注意: 一个文件里面只能dump or dumps 一次，不然在load时一定会报错。

### 3.11 shelve模块（序列化）

shelve模块是一个简单的k,v将内存数据通过文件持久化的模块，可以持久化任何pickle可支持的python数据格式。（是pickle的封装）

**import** shelve  
*# 写文件 序列化*f = shelve.open(**"shelve\_file"**)  
names = {**"k1"**:**"v1"**,**"k2"**:**"v2"**}  
list01 = [**'a'**,**'b'**,**'c'**,**'d'**]  
f[**"names"**] = names  
f[**"list01"**] = list01  
f.close()

*#读取文件 反序列化*

f = shelve.open(**"shelve\_file"**)  
print(list(f.keys()),**'up key back value'**,list(f.values()))  
print(f[**'names'**])  
*#删除***del** f[**'names'**]  
*#修改，假如key对应的values是 列表或者字典，那么列表或者字典里面的值是不可以单独修改的。只有一起修改才可以。*f[**'names'**] = [**'x'**,**'y'**,**'z'**] #直接修改文件里面，key对应的value  
f.close()

### 3.12 xml模块

xml是实现不同语言或程序之间进行数据交换的协议，跟json差不多，但json使用起来更简单，不过，古时候，在json还没诞生的黑暗年代，大家只能选择用xml呀，至今很多传统公司如金融行业的很多系统的接口还主要是xml。

xml的格式如下，就是通过<>节点来区别数据结构的:

<?xml version="1.0"?>

<data>

<country name="Liechtenstein">

<rank updated="yes">2</rank>

<year>2008</year>

<gdppc>141100</gdppc>

<neighbor name="Austria" direction="E"/>

<neighbor name="Switzerland" direction="W"/>

</country>

<country name="Singapore">

<rank updated="yes">5</rank>

<year>2011</year>

<gdppc>59900</gdppc>

<neighbor name="Malaysia" direction="N"/>

</country>

<country name="Panama">

<rank updated="yes">69</rank>

<year>2011</year>

<gdppc>13600</gdppc>

<neighbor name="Costa Rica" direction="W"/>

<neighbor name="Colombia" direction="E"/>

</country>

</data>

xml协议在各个语言里的都 是支持的，在python中可以用以下模块操作xml

查找xml文档的内容：

import xml.etree.ElementTree as ET

tree = ET.parse("xmltest.xml") #相当于打开文件

root = tree.getroot() #相当于到达这个tree对象的第一个位置，f.seek(0

print(root.tag) #从 f.seek(0)开始的第一个tag

#遍历xml文档

for child in root:

print(child.tag, child.attrib) #打印tag 以及tag的属性

for i in child:

print(i.tag,i.text) #打印tag标签，以及text（内容）

#只遍历year 节点

for node in root.iter('year'): #只查看year标签

print(node.tag,node.text) #打印year标签，以及year标签里面的值。

修改和删除xml文档内容：

import xml.etree.ElementTree as ET

tree = ET.parse("xmltest.xml")

root = tree.getroot()

#修改

for node in root.iter('year'):

new\_year = int(node.text) + 1 #因为text里面都是字符串，因此，要用到int。

node.text = str(new\_year) #存到xml的tag里面，需要字符串

node.set("updated","yes") # 在该标签里面设置一个属性，并赋值。

tree.write("xmltest.xml") #写到xml里面

#删除node

for country in root.findall('country'): # root.findall('country') 查找xml文档里面的country标签

rank = int(country.find('rank').text) #查到到country以后，在找rank标签。

if rank > 50:

root.remove(country) #删除不符合条件的country的标签，以及子标签。

tree.write('output.xml')

自己创建xml文档：

import xml.etree.ElementTree as ET

new\_xml = ET.Element("namelist") #创建一个根对象（相当于上面的data）

name = ET.SubElement(new\_xml,"name",attrib={"enrolled":"yes"})

age = ET.SubElement(name,"age",attrib={"checked":"no"})

sex = ET.SubElement(name,"sex")

sex.text = '33'

name2 = ET.SubElement(new\_xml,"name",attrib={"enrolled":"no"})

age = ET.SubElement(name2,"age")

age.text = '19'

et = ET.ElementTree(new\_xml) #生成文档对象

et.write("test.xml", encoding="utf-8",xml\_declaration=True)

ET.dump(new\_xml) #打印生成的格式

### 3.13 configparser模块

用法一（查询，判断）：

文件 conf\_test.ini

**[group1]  
k1** = **v1  
k2**:**v2  
  
[group2]  
k1** = **v1**

*#!/usr/bin/env python  
#coding:utf-8***import** configparser  
conf = configparser.ConfigParser() *#生成一个conf对象*conf.read(**"conf.ini"**)  
print(conf.sections())  
print(conf.default\_section) *#打印文件里面的组名（[] 括起来的名称）*print(list(conf[**"bitbucket.org"**].keys())) *#打印当前组的key值，包括default的key值*print(conf[**"bitbucket.org"**][**'user'**]) *#打印key（'user'） 的值。***for** k,v **in** conf[**'bitbucket.org'**].items(): *#循环打印 组['bitbucket.org'] 以及 [DEFAULT]组里面的值* print(k,v)  
  
**if 'User' in** conf[**'bitbucket.org'**]: *#判断某个值是否在组['bitbucket.org'] 和[DEFAULT] 里面。* print(**'in'**)

用法二（读取，增加，删除）：

*#!/usr/bin/env python  
#coding:utf-8***import** configparser  
conf = configparser.ConfigParser() *#生成一个conf对象*conf.read(**"conf\_test.ini"**)  
print(dir(conf))  
  
*#***读取**print(conf.options(**'group1'**)) *#读到组 group1 里面的key result:['k1', 'k2']*print(conf[**'group1'**][**'k2'**]) *#获取对应key的value reslut: v2*item\_list = conf.items(**'group1'**)  
print(item\_list) *#result: [('k1', 'v1'), ('k2', 'v2')]  
  
#***增加** *#方法一：*conf.add\_section(**"group3"**)  
conf[**"group3"**][**'name'**] = **'ryan'**conf[**"group3"**][**'age'**] = **'18'***# 方法二：*conf.add\_section(**"group4"**)  
conf.set(**"group4"**,**'address'**,**'SH'**)  
conf.set(**"group4"**,**'PHONE'**,**'1234'**)  
f = open(**"conf\_test\_new.ini"**,**'w'**)  
conf.write(f)  
f.close()  
  
*#***删除**conf02 = configparser.ConfigParser() *#生成一个conf对象*conf02.read(**"conf\_test\_new.ini"**)  
*#删除一个key值*conf02.remove\_option(**"group4"**,**"address"**)  
f = open(**"conf\_test\_new.ini"**,**'w'**)  
conf02.write(f)  
f.close()  
*#删除一个section*conf02.remove\_section(**"group4"**)  
f = open(**"conf\_test\_new.ini"**,**'w'**)  
conf02.write(f)  
f.close()

### 3.14 hashlib 加密模块

1.HASH介绍

简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

HASH主要用于信息安全领域中加密算法，他把一些不同长度的信息转化成杂乱的128位的编码里,叫做HASH值.也可以说，hash就是找到一种数据内容和数据存放地址之间的映射关系

2.MD5

**什么是MD5算法**

MD5讯息摘要演算法（英语：MD5 Message-Digest Algorithm），一种被广泛使用的密码杂凑函数，可以产生出一个128位的散列值（hash value），用于确保信息传输完整一致。MD5的前身有MD2、MD3和MD4。

**MD5功能**

输入任意长度的信息，经过处理，输出为128位的信息（数字指纹）；  
不同的输入得到的不同的结果（唯一性）；

**MD5算法的特点**

压缩性：任意长度的数据，算出的MD5值的长度都是固定的

容易计算：从原数据计算出MD5值很容易

抗修改性：对原数据进行任何改动，修改一个字节生成的MD5值区别也会很大

强抗碰撞：已知原数据和MD5，想找到一个具有相同MD5值的数据（即伪造数据）是非常困难的。

**MD5算法是否可逆？**

MD5不可逆的原因是其是一种散列函数，使用的是hash算法，在计算过程中原文的部分信息是丢失了的。

MD5用途

防止被篡改：

防止直接看到明文：

防止抵赖（数字签名）

MD5与SHA-1的简单比较

SHA-1 比MD5 被强行攻击时强度更大。

SHA-1的运行速度比MD5慢。

加密相关的操作，3.x里代替了md5模块和sha模块，主要提供 SHA1, SHA224, SHA256, SHA384, SHA512 ，MD5 算法

*#!/usr/bin/env python  
#coding:utf-8***import** hashlib

m = hashlib.md5()  
m.update(**b"Hello"**)  
m.update(**b"It's me"**)  
print(m.digest())  
m.update(**b"It's been a long time since last time we ..."**)  
  
print(m.digest()) *#2进制格式hash*print(len(m.hexdigest())) *#16进制格式hash***'''  
def digest(self, \*args, \*\*kwargs): # real signature unknown  
 """ Return the digest value as a string of binary data. """  
 pass  
  
def hexdigest(self, \*args, \*\*kwargs): # real signature unknown  
 """ Return the digest value as a string of hexadecimal digits. """  
 pass  
  
'''  
import** hashlib  
  
*# ######## md5 ########*hash = hashlib.md5()  
hash.update(**'admin'**)  
print(hash.hexdigest())  
  
*# ######## sha1 ########*hash = hashlib.sha1()  
hash.update(**'admin'**)  
print(hash.hexdigest())  
  
*# ######## sha256 ########*hash = hashlib.sha256()  
hash.update(**'admin'**)  
print(hash.hexdigest())  
  
  
*# ######## sha384 ########*hash = hashlib.sha384()  
hash.update(**'admin'**)  
print(hash.hexdigest())  
  
*# ######## sha512 ########*hash = hashlib.sha512()  
hash.update(**'admin'**)  
print(hash.hexdigest())

### 3.15 subprocess 模块

**三种执行命令的方法**

* subprocess.run(\*popenargs, input=None, timeout=None, check=False, \*\*kwargs) #官方推荐
* subprocess.call(\*popenargs, timeout=None, \*\*kwargs) #跟上面实现的内容差不多，另一种写法
* subprocess.Popen() #上面各种方法的底层封装

**run()方法**

**标准写法**

subprocess.run(['df','-h'],stderr=subprocess.PIPE,stdout=subprocess.PIPE,check=True)

check=True：表示如果shell命令返回的状态码不为“0”直接报错，如果不加，报错不直接显示出来。

**涉及到管道|的命令需要这样写**

subprocess.run('df -h|grep disk1',shell=True) #shell=True的意思是这条命令直接交给系统去执行，不需要python负责解析

subprocess.run('df -h|grep vda1',stderr=subprocess.PIPE,stdout=subprocess.PIPE,shell=True)

**call()方法**

#执行命令，返回命令执行状态 ， 0 or 非0

>>> retcode = subprocess.call(["ls", "-l"])

#执行命令，如果命令结果为0，就正常返回，否则抛异常。

>>> subprocess.check\_call(["ls", "-l"])

0

#接收字符串格式命令，返回元组形式，第1个元素是执行状态，第2个是命令结果

>>> subprocess.getstatusoutput('ls /bin/ls')

(0, '/bin/ls')

#接收字符串格式命令，并返回结果

>>> subprocess.getoutput('ls /bin/ls')

'/bin/ls'

#执行命令，并返回结果，注意是返回结果，不是打印，下例结果返回给res

>>> res=subprocess.check\_output(['ls','-l'])

>>> res

b'total 0\ndrwxr-xr-x 12 alex staff 408 Nov 2 11:05 OldBoyCRM\n'

**Popen()方法**

常用参数：

* args：shell命令，可以是字符串或者序列类型（如：list，元组）
* stdin, stdout, stderr：分别表示程序的标准输入、输出、错误句柄
* preexec\_fn：只在Unix平台下有效，用于指定一个可执行对象（callable object），它将在子进程运行之前被调用
* shell：同上，shell = True
* cwd：用于设置子进程的当前目录
* env：用于指定子进程的环境变量。如果env = None，子进程的环境变量将从父进程中继承。

下面这2条语句执行会有什么区别？

a=subprocess.run('sleep 10',shell=True,stdout=subprocess.PIPE)

a=subprocess.Popen('sleep 10',shell=True,stdout=subprocess.PIPE)

区别是Popen会在发起命令后立刻返回，而不等命令执行结果。这样的好处是什么呢？

如果你调用的命令或脚本 需要执行10分钟，你的主程序不需卡在这里等10分钟，可以继续往下走，干别的事情，每过一会，通过一个什么方法来检测一下命令是否执行完成就好了。

Popen调用后会返回一个对象，可以通过这个对象拿到命令执行结果或状态等，该对象有以下方法

poll()

Check if child process has terminated. Returns returncode。#检查子进程是否已经终止。如果终止返回状态码

wait()

Wait for child process to terminate. Returns returncode attribute.

terminate()终止所启动的进程（非强制）Terminate the process with SIGTERM

kill() 杀死所启动的进程 Kill the process with SIGKILL

communicate()与启动的进程交互，发送数据到stdin,并从stdout接收输出，然后等待任务结束

>>> a = subprocess.Popen('python3 guess\_age.py',stdout=subprocess.PIPE,stderr=subprocess.PIPE,stdin=subprocess.PIPE,shell=True)

>>> a.communicate(b'22')

(b'your guess:try bigger\n', b'')

send\_signal(signal.xxx)发送系统信号

pid 拿到所启动进程的进程号

### 3.16 logging模块

很多程序都有记录日志的需求，并且日志中包含的信息即有正常的程序访问日志，还可能有错误、警告等信息输出，python的logging模块提供了标准的日志接口，你可以通过它存储各种格式的日志，logging的日志可以分为 debug(), info(), warning(), error() and critical()5个级别，下面我们看一下怎么用。

**最简单用法**

import logging

logging.warning("user [alex] attempted wrong password more than 3 times")

logging.critical("server is down")

输出

WARNING:root:user [alex] attempted wrong password more than 3 times

CRITICAL:root:server is down

日志级别

| **Level** | **When it’s used** |
| --- | --- |
| DEBUG(10) | Detailed information, typically of interest only when diagnosing problems. |
| INFO(20) | Confirmation that things are working as expected. |
| WARNING(30) | An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected. |
| ERROR(40) | Due to a more serious problem, the software has not been able to perform some function. |
| CRITICAL(50) | A serious error, indicating that the program itself may be unable to continue running. |

DEBUG<INFO<WARNING<ERROR<CRITICAL

如果想把日志写到文件里，也很简单

import logging

logging.basicConfig(filename='example.log',level=logging.INFO)

logging.debug('This message should go to the log file')

logging.info('So should this')

logging.warning('And this, too')

其中下面这句中的level=loggin.INFO意思是，把日志纪录级别设置为INFO，也就是说，只有比日志是INFO或比INFO级别更高的日志才会被纪录到文件里，在这个例子， 第一条日志是不会被纪录的，如果希望纪录debug的日志，那把日志级别改成DEBUG就行了。

logging.basicConfig(filename='example.log',level=logging.INFO)

自定义日志格式

感觉上面的日志格式忘记加上时间啦，日志不知道时间怎么行呢，下面就来加上!

import logging

logging.basicConfig(format='%(asctime)s %(message)s', datefmt='%m/%d/%Y %I:%M:%S %p')

logging.warning('is when this event was logged.')

#输出

12/12/2010 11:46:36 AM is when this event was logged.

除了加时间，还可以自定义一大堆格式，下表就是所有支持的格式

| **%(name)s** | **Logger的名字** |
| --- | --- |
| %(levelno)s | 数字形式的日志级别 |
| %(levelname)s | 文本形式的日志级别 |
| %(pathname)s | 调用日志输出函数的模块的完整路径名，可能没有 |
| %(filename)s | 调用日志输出函数的模块的文件名 |
| %(module)s | 调用日志输出函数的模块名 |
| %(funcName)s | 调用日志输出函数的函数名 |
| %(lineno)d | 调用日志输出函数的语句所在的代码行 |
| %(created)f | 当前时间，用UNIX标准的表示时间的浮 点数表示 |
| %(relativeCreated)d | 输出日志信息时的，自Logger创建以 来的毫秒数 |
| %(asctime)s | 字符串形式的当前时间。默认格式是 “2003-07-08 16:49:45,896”。逗号后面的是毫秒 |
| %(thread)d | 线程ID。可能没有 |
| %(threadName)s | 线程名。可能没有 |
| %(process)d | 进程ID。可能没有 |
| %(message)s | 用户输出的消息 |

**日志同时输出到屏幕和文件**

如果想同时把log打印在屏幕和文件日志里，就需要了解一点复杂的知识 了

Python 使用logging模块记录日志涉及四个主要类，使用官方文档中的概括最为合适：

* **logger提供了应用程序可以直接使用的接口；**
* **handler将(logger创建的)日志记录发送到合适的目的输出；**
* **filter提供了细度设备来决定输出哪条日志记录；**
* **formatter决定日志记录的最终输出格式。**

**每个组件的主要功能**

**logger**

每个程序在输出信息之前都要获得一个Logger。Logger通常对应了程序的模块名，比如聊天工具的图形界面模块可以这样获得它的Logger：

LOG=logging.getLogger(”chat.gui”) #logger对象

而核心模块可以这样：

LOG01=logging.getLogger(”chat.kernel”)

还可以绑定handler和filters

Logger.setLevel(lel):指定最低的日志级别，低于lel的级别将被忽略。debug是最低的内置级别，critical为最高

Logger.addFilter(filt)、Logger.removeFilter(filt):添加或删除指定的filter

Logger.addHandler(hdlr)、Logger.removeHandler(hdlr)：增加或删除指定的handler

Logger.debug()、Logger.info()、Logger.warning()、Logger.error()、Logger.critical()：可以设置的日志级别

#### **handler**

handler对象负责发送相关的信息到指定目的地。Python的日志系统有多种Handler可以使用。有些Handler可以把信息输出到控制台，有些Handler可以把信息输出到文件，还有些 Handler可以把信息发送到网络上。如果觉得不够用，还可以编写自己的Handler。可以通过addHandler()方法添加多个多handler

Handler.setLevel(lel):指定被处理的信息级别，低于lel级别的信息将被忽略

Handler.setFormatter()：给这个handler选择一个格式

Handler.addFilter(filt)、Handler.removeFilter(filt)：新增或删除一个filter对象

每个Logger可以附加多个Handler。接下来我们就来介绍一些常用的Handler：

**logging.StreamHandler** 使用这个Handler可以向类似与sys.stdout或者sys.stderr的任何文件对象(file object)输出信息。

**logging.FileHandler**和StreamHandler 类似，用于向一个文件输出日志信息。不过FileHandler会帮你打开这个文件

**logging.handlers.RotatingFileHandler**

这个Handler类似于上面的FileHandler，但是它可以管理文件大小。当文件达到一定大小之后，它会自动将当前日志文件改名，然后创建 一个新的同名日志文件继续输出。比如日志文件是chat.log。当chat.log达到指定的大小之后，RotatingFileHandler自动把 文件改名为chat.log.1。不过，如果chat.log.1已经存在，会先把chat.log.1重命名为chat.log.2。。。最后重新创建 chat.log，继续输出日志信息。它的函数是：

RotatingFileHandler( filename[, mode[, maxBytes[, backupCount]]])

其中filename和mode两个参数和FileHandler一样。

* + maxBytes用于指定日志文件的最大文件大小。如果maxBytes为0，意味着日志文件可以无限大，这时上面描述的重命名过程就不会发生。
  + backupCount用于指定保留的备份文件的个数。比如，如果指定为2，当上面描述的重命名过程发生时，原有的chat.log.2并不会被更名，而是被删除。

**logging.handlers.TimedRotatingFileHandler**

这个Handler和RotatingFileHandler类似，不过，它没有通过判断文件大小来决定何时重新创建日志文件，而是间隔一定时间就 自动创建新的日志文件。重命名的过程与RotatingFileHandler类似，不过新的文件不是附加数字，而是当前时间。它的函数是：

TimedRotatingFileHandler( filename [,when [,interval [,backupCount]]])

其中filename参数和backupCount参数和RotatingFileHandler具有相同的意义。

interval是时间间隔。

when参数是一个字符串。表示时间间隔的单位，不区分大小写。它有以下取值：

* + 1. S 秒
    2. M 分
    3. H 小时
    4. D 天
    5. W 每星期（interval==0时代表星期一）
    6. midnight 每天凌晨

#### **formatter 组件**

日志的formatter是个独立的组件，可以跟handler组合

fh = logging.FileHandler("access.log")

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

fh.setFormatter(formatter) #把formmater绑定到fh上

#### filter 组件

如果你想对日志内容进行过滤，就可自定义一个filter

class IgnoreBackupLogFilter(logging.Filter):

"""忽略带db backup 的日志"""

def filter(self, record): #固定写法

return "db backup" not in record.getMessage()

注意filter函数会返加True or False，logger根据此值决定是否输出此日志

然后把这个filter添加到logger中

logger.addFilter(IgnoreBackupLogFilter())

下面的日志就会把符合filter条件的过滤掉

logger.debug("test ....")

logger.info("test info ....")

logger.warning("start to run db backup job ....")

logger.error("test error ....")

日志总的操作：

*#coding:UTF-8***import** logging  
**from** logging **import** handlers  
  
*#Filter过滤日志内容，可以有多个。***class** IgnoreBackupLogFilter(logging.Filter):  
 *"""忽略带db backup 的日志"""* **def** filter(self, record): *#固定写法  
 #如果在message里面包含了 "db backup" 就不返回。* **return "db backup" not in** record.getMessage()  
  
*#1.生成logger对象*logger = logging.getLogger(**"web"**)  
*#1.1 设置日志级别，如果不设置，默认的级别是 >= warning*logger.setLevel(logging.DEBUG)  
*#l.1 把filter对象添加到logger中*logger.addFilter(IgnoreBackupLogFilter())  
  
*#2.生成handler对象  
#输出到屏幕，并且设置日志级别*ch = logging.StreamHandler()  
ch.setLevel(logging.INFO)  
  
*#输出到文件，并设置日志级别  
# fh = logging.FileHandler('web.log')  
# fh = handlers.RotatingFileHandler( "web.log",maxBytes=10,backupCount=3)  
# fh.setLevel(logging.WARNING)*fh = handlers.TimedRotatingFileHandler( **"web.log"**,when=**"S"**,interval=5,backupCount=3)  
  
*#2.1 把handler对象绑定到logger*logger.addHandler(ch)  
logger.addHandler(fh)  
  
  
*#3.生成formatter对象*file\_formatter = logging.Formatter(**'%(asctime)s - %(name)s - %(levelname)s - %(message)s'**)  
console\_formatter = logging.Formatter(**'%(asctime)s - %(name)s - %(levelname)s - %(lineno)d - %(message)s'**)  
  
*#3.1 把formatter对象绑定到handler对象*ch.setFormatter(console\_formatter)  
fh.setFormatter(file\_formatter)  
  
*#console : INFO  
#global : DEBUG ,global default level : warning  
#file :Warning  
#全局设置为DEBUG后， console handler 设置为INFO, 如果输出的日志级别是debug, 那就不会在屏幕上打印  
  
#日志格式等级：数字越大，等级越高***'''  
CRITICAL = 50  
FATAL = CRITICAL  
ERROR = 40  
WARNING = 30  
WARN = WARNING  
INFO = 20  
DEBUG = 10  
NOTSET = 0  
'''***#代码运行优先级，先查看global的日志等级，在查看局部的等级。*logger.warning(**"test log debug"**)  
logger.info(**"test log info"**)  
logger.error(**" db backup test log info"**)

### 3.17 re模块

正则表达式就是字符串的匹配规则，在多数编程语言里都有相应的支持，python里对应的模块是re

模式 描述

^ 匹配字符串的开头

$ 匹配字符串的末尾。

. 匹配任意字符，除了换行符，当re.DOTALL标记被指定时，则可以匹配包括换行符的任意字符。

[...] 用来表示一组字符,单独列出：[amk] 匹配 'a'，'m'或'k'

[^...] 不在[]中的字符：[^abc] 匹配除了a,b,c之外的字符。

re\* 匹配0个或多个的表达式。

re+ 匹配1个或多个的表达式。

re? 匹配0个或1个由前面的正则表达式定义的片段，非贪婪方式

re{ n}

re{ n,} 精确匹配n个前面表达式。

re{ n, m} 匹配 n 到 m 次由前面的正则表达式定义的片段，贪婪方式

a| b 匹配a或b

(re) G匹配括号内的表达式，也表示一个组

(?imx) 正则表达式包含三种可选标志：i, m, 或 x 。只影响括号中的区域。

(?-imx) 正则表达式关闭 i, m, 或 x 可选标志。只影响括号中的区域。

(?: re) 类似 (...), 但是不表示一个组

(?imx: re) 在括号中使用i, m, 或 x 可选标志

(?-imx: re) 在括号中不使用i, m, 或 x 可选标志

(?#...) 注释.

(?= re) 前向肯定界定符。如果所含正则表达式，以 ... 表示，在当前位置成功匹配时成功，否则失败。但一旦所含表达式已经尝试，匹配引擎根本没有提高；模式的剩余部分还要尝试界定符的右边。

(?! re) 前向否定界定符。与肯定界定符相反；当所含表达式不能在字符串当前位置匹配时成功

(?> re) 匹配的独立模式，省去回溯。

\w 匹配字母数字

\W 匹配非字母数字

\s 匹配任意空白字符，等价于 [\t\n\r\f].

\S 匹配任意非空字符

\d 匹配任意数字，等价于 [0-9].

\D 匹配任意非数字

\A 匹配字符串开始 相当于 ^ 以什么开始。

\Z 匹配字符串结束，如果是存在换行，只匹配到换行前的结束字符串。相当于$

\z 匹配字符串结束

\G 匹配最后匹配完成的位置。

\b 匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。

\B 匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。

\n, \t, 等. 匹配一个换行符。匹配一个制表符。等

\1...\9 匹配第n个分组的子表达式。

\10 匹配第n个分组的子表达式，如果它经匹配。否则指的是八进制字符码的表达式。

'(?P<name>...)' 分组匹配 re.search("(?P<province>[0-9]{4})(?P<city>[0-9]{2})(?P<birthday>[0-9]{4})","371481199306143242").**groupdict**("city") 结果{'province': '3714', 'city': '81', 'birthday': '1993'}

**re的匹配语法有以下几种**

* re.match 从头开始匹配
* re.search 匹配包含
* re.findall 把所有匹配到的字符放到以列表中的元素返回
* re.split 以匹配到的字符当做列表分隔符
* re.sub 匹配字符并替换
* re.fullmatch 全部匹配

**re.compile(pattern, flags=0)**

Compile a regular expression pattern into a regular expression object, which can be used for matching using its match(), search() and other methods, described below.

The sequence

prog = re.compile(pattern)

result = prog.match(string)

is equivalent to

result = re.match(pattern, string)

but using re.compile() and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

疑问：complie用在什么地方，为什么要用这个呢？

主要用于要多次去用每个正则表达是去匹配不同的字符串。如果把正则每次都写到match or search 等语法当中，那么每次去匹配的时候，python解释器都会去翻译这个正则匹配写到内存（多次），用到compile之后，只需要写入一次，以后需要调用该正则匹配的时候，python解释器就不用每次都进行匹配，直接调用就行。

**re.match(pattern, string, flags=0)**

从起始位置开始根据模型去字符串中匹配指定内容，匹配单个

* pattern 正则表达式
* string 要匹配的字符串
* flags 标志位，用于控制正则表达式的匹配方式

import re

obj = re.match('\d+', '123uuasf')

if obj:

print obj.group()

|  |  |
| --- | --- |
| **匹配对象方法** | **描述** |
| group(num=0) | 匹配的整个表达式的字符串，group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组。 |
| groups() | 返回一个包含所有小组字符串的元组，从 1 到 所含的小组号。 |

正则表达式可以包含一些可选标志修饰符来控制匹配的模式。修饰符被指定为一个可选的标志。多个标志可以通过按位 OR(|) 它们来指定。如 re.I | re.M 被设置成 I 和 M 标志

|  |  |
| --- | --- |
| **修饰符** | **描述** |
| re.I | 使匹配对大小写不敏感 |
| re.L | 做本地化识别（locale-aware）匹配 |
| re.M | 多行匹配，影响 ^ 和 $ |
| re.S | 使 . 匹配包括换行在内的所有字符 |
| re.U | 根据Unicode字符集解析字符。这个标志影响 \w, \W, \b, \B. |
| re.X | 该标志通过给予你更灵活的格式以便你将正则表达式写得更易于理解。用于注释 |

**re.search(pattern, string, flags=0)**

根据模型去字符串中匹配指定内容，匹配单个

import re

obj = re.search('\d+', 'u123uu888asf')

if obj:

print obj.group()

**re.findall(pattern, string, flags=0)**

match and search均用于匹配单值，即：只能匹配字符串中的一个，如果想要匹配到字符串中所有符合条件的元素，则需要使用 findall。

import re

obj = re.findall('\d+', 'fa123uu888asf')

print obj

**re.sub(pattern, repl, string, count=0, flags=0)**

用于替换匹配的字符串

>>>re.sub('[a-z]+','sb','^ASabc123',)

>>> re.sub('\d+','|', 'alex22wupeiqi33oldboy55',count=2)

'alex|wupeiqi|oldboy55'

相比于str.replace功能更加强大

**re.split(pattern, string, maxsplit=0, flags=0)**

>>>s='9-2\*5/3+7/3\*99/4\*2998+10\*568/14'

>>>re.split('[\\*\-\/\+]',s)

['9', '2', '5', '3', '7', '3', '99', '4', '2998', '10', '568', '14']

>>> re.split('[\\*\-\/\+]',s,3)

['9', '2', '5', '3+7/3\*99/4\*2998+10\*568/14']

**re.fullmatch(pattern, string, flags=0)**

整个字符串匹配成功就返回re object, 否则返回None

re.fullmatch('\w+@\w+\.(com|cn|edu)',"alex@oldboyedu.cn")

### re 练习题

1.打印文件中合法的手机号。

test文件：

姓名 地区 身高 体重 电话  
况咏蜜 北京 171 48 13651054608  
叶梓萱 上海 171 49 18042432324  
杜姗姗1 北京 167 49 13324523342  
杜姗姗2 北京 167 49 23324523342  
杜姗姗3 北京 167 49 33324523342

**import** io  
**import** re  
pattern = re.compile(**'1[3458]\d{9}'**)  
**with** io.open(**"test"**,**'r'**,encoding=**"utf-8"**) **as** f:  
 **for** line **in** f.readlines():  
 res = re.findall(pattern,line)  
 **if** res:  
 print(res)

2.验证邮箱是否合法。

email\_info 文件：

yang IT ryan@bakheet.cn  
Gou Teacher cherry@qq.com  
Ping doctor [ping@bakheet.cn](mailto:ping@bakheet.cn)

pattern = re.compile(**'(\w+@\w+.(com|cn))'**)  
**with** io.open(**"email\_info"**,**'r'**,encoding=**"utf-8"**) **as** f:  
 **for** line **in** f.readlines():res = re.findall(pattern,line)  
 **if** res:  
 print(res)

# 第三模块 第一章 类

## 3.1 编程范式

编程是程序员用特定的语法+数据结构+算法组成的代码来告诉计算机如何执行任务的过程，一个程序是程序员为了得到一个任务结果而编写的一组指令的集合，正所谓条条大路通罗马，实现一个任务的方式有很多种不同的方式， 对这些不同的编程方式的特点进行归纳总结得出来的编程方式类别，即为编程范式。 不同的编程范式本质上代表对各种类型的任务采取的不同的解决问题的思路， 大多数语言只支持一种编程范式，当然也有些语言可以同时支持多种编程范式。 两种最重要的编程范式分别是面向过程编程和面向对象编程

## 3.2 面向过程编程

面向过程：核心是过程二字，过程指的是解决问题的步骤，设计一条流水线，机械式的思维方式  
优点：复杂的问题流程化，进而简单化  
缺点：可扩展性差

## 3.3 面向对象编程

面向对象：核心就是对象二字，对象就是特征与技能的结合体  
优点：可扩展性强  
缺点：编程复杂度高  
应用场景：用户需求经常变化，互联网应用，游戏，企业内部应用  
  
  
类就是一系列对象相似的特征与技能的结合体  
强调：站在不同的角度，得到的分类是不一样的  
  
在现实世界中：一定先有对象，后有类  
在程序中：一定得先定义类，后调用类来产生对象

**面向对象介绍**

学习面向对象过程中会遇到一些名词，我们先解释下

**名词解释**

**类：**一个类即是对一类拥有相同属性的对象的抽象、蓝图、原型、模板。在类中定义了这些对象的都具备的属性（variables(data)）、共同的方法

**属性：**人类包含很多特征，把这些特征用程序来描述的话，叫做属性，比如年龄、身高、性别、姓名等都叫做属性，一个类中，可以有多个属性

**方法：**人类不止有身高、年龄、性别这些属性，还能做好多事情，比如说话、走路、吃饭等，相比较于属性是名词，说话、走路是动词，这些动词用程序来描述就叫做方法。

**实例(对象)：**一个对象即是一个类的实例化后实例，一个类必须经过实例化后方可在程序中调用，一个类可以实例化多个对象，每个对象亦可以有不同的属性，就像人类是指所有人，每个人是指具体的对象，人与人之前有共性，亦有不同

**实例化：**把一个类转变为一个对象的过程就叫实例化

**面向对象3大特性**

注意，此处只需知道这几个名词，后面会专门再对着代码学习这几大特性

**Encapsulation 封装**

在类中对数据的赋值、内部调用对外部用户是透明的，这使类变成了一个胶囊或容器，里面包含着类的数据和方法

**Inheritance 继承**

一个类可以派生出子类，在这个父类里定义的属性、方法自动被子类继承

**Polymorphism 多态**

多态是面向对象的重要特性,简单点说:**“一个接口，多种实现”**，指一个基类中派生出了不同的子类，且每个子类在继承了同样的方法名的同时又对父类的方法做了不同的实现，这就是同一种事物表现出的多种形态。

编程其实就是一个**将具体世界进行抽象化的过程**，多态就是抽象化的一种体现，把一系列具体事物的共同点抽象出来, 再通过这个抽象的事物, 与不同的具体事物进行对话。

对不同类的对象发出相同的消息将会有不同的行为。比如，你的老板让所有员工在九点钟开始工作, 他只要在九点钟的时候说：“开始工作”即可，而不需要对销售人员说：“开始销售工作”，对技术人员说：“开始技术工作”, 因为“员工”是一个抽象的事物, 只要是员工就可以开始工作，他知道这一点就行了。至于每个员工，当然会各司其职，做各自的工作。

**多态允许将子类的对象当作父类的对象使用**，某父类型的引用指向其子类型的对象,调用的方法是该子类型的方法。这里引用和调用方法的代码编译前就已经决定了,而引用所指向的对象可以在运行期间动态绑定

## 3.4 初识类

**class** First(object): #创建类  
 **def** func(selt): #创建类中的函数  
 print(**'hello word!'**)  
obj = First() #根据类first创建对象obj  
  
**class** 关键字，表示要创建类  
self 特殊参数，必填  
创建对象，类名称后加括号即可  
注：类中的函数第一个参数必须是self  
  
类中定义的函数叫做 “方法”  
好处：  
1，节省内存  
2，根据情况可以改动方法的特征

例子２：创建学生对象

站在路飞学院的角度，大家都是学生  
在现实世界中：  
 对象1：王二丫  
 特征：  
 学校='luffycity'  
 名字='王二丫'  
 性别='女'  
 年龄=18  
 技能：  
 学习  
 吃饭  
 睡觉  
  
 对象2：李三炮  
 特征：  
 学校='luffycity'  
 名字='李三炮'  
 性别='男'  
 年龄=38  
 技能：  
 学习  
 吃饭  
 睡觉  
  
 对象3：张铁蛋  
 特征：  
 学校='luffycity'  
 名字='张铁蛋'  
 性别='男'  
 年龄=48  
 技能：  
 学习  
 吃饭  
 睡觉  
  
 总结现实中路飞学院的学生类：  
 相似的特征  
 学校='luffycity'  
  
 相似的技能  
 学习  
 吃饭  
 睡觉

程序实现：  
#先定义类  
**class** LuffyStudent:  
 school=**'luffycity'  
  
 def** learn(self):  
 print(**'is learning'**)  
  
 **def** eat(self):  
 print(**'is sleeping'**)  
  
  
#后产生对象  
stu1=LuffyStudent()  
stu2=LuffyStudent()  
stu3=LuffyStudent()  
print(stu1)  
print(stu2)  
print(stu3)

执行结果：LuffyStudent 这个类产生的对象的内存地址

<\_\_main\_\_.LuffyStudent object at 0x10dbdb9b0>

<\_\_main\_\_.LuffyStudent object at 0x10dbdbb70>

<\_\_main\_\_.LuffyStudent object at 0x10dbdbba8>

## 3.5 如何使用类

重点：

1. 类名.\_\_dict\_\_ 打印类的名称空间 ，类的名称空间(\_\_dict\_\_)返回的是一个字典

*#先定义类***class** LuffyStudent:  
 school=**'luffycity'** *#数据属性* **def** learn(self): *#函数属性* print(**'is learning'**)  
 **def** eat(self): *#函数属性* print(**'is sleeping'**)

#打印名称空间  
print(LuffyStudent.\_\_dict\_\_) *#打印类的名称空间*print(LuffyStudent.\_\_dict\_\_[**'school'**]) *#打印类的数据属性的名称空间*print(LuffyStudent.\_\_dict\_\_[**'learn'**]) *#打印方法（函数，技能）的名称空间*

结果：

第一个print 是打印类的名称空间，

{'\_\_module\_\_': '\_\_main\_\_', 'school': 'luffycity', 'learn': <function LuffyStudent.learn at 0x10bf058c8>, 'eat': <function LuffyStudent.eat at 0x10bf05840>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'LuffyStudent' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'LuffyStudent' objects>, '\_\_doc\_\_': None}

luffycity

<function LuffyStudent.learn at 0x10bf058c8>

类是，一系列对象相似特征与技能的结合体  
特征用变量去表示，技能用函数去表示。  
查看类的名称空间

#查print(LuffyStudent.school) *#LuffyStudent.\_\_dict\_\_['school']*print(LuffyStudent.learn) *#LuffyStudent.\_\_dict\_\_['learn']*

结果：

luffycity

<function LuffyStudent.learn at 0x10ccbc8c8>

#增

LuffyStudent.county=**'China'**print(LuffyStudent.\_\_dict\_\_)  
print(LuffyStudent.county)

结果：

{'\_\_module\_\_': '\_\_main\_\_', 'school': 'luffycity', 'learn': <function LuffyStudent.learn at 0x105c798c8>, 'eat': <function LuffyStudent.eat at 0x105c79840>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'LuffyStudent' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'LuffyStudent' objects>, '\_\_doc\_\_': None, 'county': 'China'}

China

*# 删***del** LuffyStudent.county

*# 改*LuffyStudent.school=**'Luffycity'**print(LuffyStudent.school)

结果：

Luffycity

## 3.6 如何使用类 和 类的 \_\_init \_\_ 方法

**\_\_init\_\_方法用来为对象定制对象自己独有的特征**

**class** LuffyStudent:  
 school=**'luffycity'** *# stu1,'王二丫','女',18* **def** \_\_init\_\_(self,name,sex,age):  
 self.Name=name  
 self.Sex=sex  
 self.Age=age  
  
 **def** learn(self):  
 print(**'is learning'**)  
  
 **def** eat(self):  
 print(**'is sleeping'**)

根据上面的类进行如下操作：

### 操作1：未被实例化时-查看类的名称空间 print(LuffyStudent.\_\_dict\_\_) #打印类的名称空间

结果：

Luffycity

### 操作2: 产生对象，\_\_init\_\_ 实例化的步骤

stu1=LuffyStudent(**'王二丫'**,**'女'**,18) *#LuffyStudent.\_\_init\_\_(stu1,'王二丫','女',18)*print(stu1.\_\_dict\_\_) #打印对象 stu1 的名称空间

结果：

{'Name': '王二丫', 'Sex': '女', 'Age': 18}

上面方法简介：

加上\_\_init\_\_方法后，实例化的步骤  
self 相当于 stu1 （实例化的对象的名称）  
1、先产生一个空对象stu1  
2、LuffyStudent.\_\_init\_\_(stu1,**'王二丫'**,**'女'**,18) #初始化

### 操作3：查对象属性

print(LuffyStudent.\_\_dict\_\_) *#类的名称空间*print(stu1.\_\_dict\_\_) *#实例化对象自己独有的属性*print(stu1.Name) #查看对象单独的属性  
print(stu1.Sex)  
print(stu1.Age)

结果：

{'\_\_module\_\_': '\_\_main\_\_', 'school': 'luffycity', '\_\_init\_\_': <function LuffyStudent.\_\_init\_\_ at 0x10ee2a8c8>, 'learn': <function LuffyStudent.learn at 0x10ee2a840>, 'eat': <function LuffyStudent.eat at 0x10ee2a7b8>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'LuffyStudent' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'LuffyStudent' objects>, '\_\_doc\_\_': None}

{'Name': '王二丫', 'Sex': '女', 'Age': 18}

王二丫

女

18

### 操作4：改变对象是属性的值

print(**"未改变对象属性之前:"**,stu1.\_\_dict\_\_)  
stu1.Name=**'李二丫'**print(**"改变对象属性之后："**,stu1.\_\_dict\_\_)  
print(stu1.Name)

结果：

未改变对象属性之前: {'Name': '王二丫', 'Sex': '女', 'Age': 18}

改变对象属性之后： {'Name': '李二丫', 'Sex': '女', 'Age': 18}

李二丫

### 操作5： 删除对象的属性

**del** stu1.Name  
print(stu1.\_\_dict\_\_)

结果：

{'Sex': '女', 'Age': 18}

### 操作6：增加对象的属性

stu1.class\_name=**'python开发'**print(stu1.\_\_dict\_\_)

结果：

{'Name': '王二丫', 'Sex': '女', 'Age': 18, 'class\_name': 'python开发'}

## 3.7 属性查找与绑定方法

情景介绍：

*'''  
在现实世界中：  
 对象1：王二丫  
 特征：  
 学校='luffycity'  
 名字='王二丫'  
 性别='女'  
 年龄=18  
 技能：  
 学习  
 吃饭  
 睡觉  
  
 对象2：李三炮  
 特征：  
 学校='luffycity'  
 名字='李三炮'  
 性别='男'  
 年龄=38  
 技能：  
 学习  
 吃饭  
 睡觉  
  
'''*

#对象：特征与技能的结合体  
#类：类是一系列对象相似的特征与相似的技能的结合体

x=**'global'  
class** LuffyStudent:  
 school=**'luffycity'  
  
 def** \_\_init\_\_(self,name,sex,age):  
 self.Name=name  
 self.Sex=sex  
 self.Age=age  
  
 **def** learn(self,x):  
 print(**'%s is learning %s'** %(self.Name,x))  
  
 **def** eat(self):  
 print(**'%s is sleeping'** %self.Name)  
  
  
*#后产生对象*stu1=LuffyStudent(**'王二丫'**,**'女'**,18)  
stu2=LuffyStudent(**'李三炮'**,**'男'**,38)  
stu3=LuffyStudent(**'张铁蛋'**,**'男'**,48)

根据上面代码做如下操作:

### 操作1：查看对象的私有特征

print(stu1.\_\_dict\_\_)  
print(stu2.\_\_dict\_\_)  
print(stu3.\_\_dict\_\_)

结果：

{'Name': '王二丫', 'Sex': '女', 'Age': 18}

{'Name': '李三炮', 'Sex': '男', 'Age': 38}

{'Name': '张铁蛋', 'Sex': '男', 'Age': 48}

### 操作2：查看类中的数据属性：是所有对象所共有的

print(LuffyStudent.school,id(LuffyStudent.school))  
print(stu1.school,id(stu1.school))  
print(stu2.school,id(stu2.school))  
print(stu3.school,id(stu3.school))

结果：

luffycity 4408710832

luffycity 4408710832

luffycity 4408710832

luffycity 4408710832

结论：类中的数据属性的ID 和对象的数据属性ID相同，因此，类中的数据属性就是 类和对象所共有的。

### 操作3：查看类中的函数属性

类中的函数属性:是绑定给对象使用的，绑定到不同的对象是不同的绑定方法，对象调用绑定方式时，会把对象本身当作第一个传入，传给self

print(stu1.learn)  
stu1.learn(1) *#LuffyStudent.learn(stu1,1)*print(stu2.learn)

结果：

<**bound method** LuffyStudent.learn of <\_\_main\_\_.LuffyStudent object at 0x109ca9d30>>

王二丫 is learning 1

<**bound method** LuffyStudent.learn of <\_\_main\_\_.LuffyStudent object at 0x109ca9d68>>

结论：**bound method** 绑定方法。相当于不同对象调用一种功能，但是各自执行不同的方法。类中定义的函数，是绑定给对象使用的。是谁在调，那就是谁在使用这个功能。

### 操作4：对象属性的查找顺序

查找顺序：先从自己的名称空间里面找自己的属性，然后在从类里面找属性(类还包括父类等)，类里面没有直接报错。

stu1.x=**'from stu1'**LuffyStudent.x=**'from Luffycity class'**print(LuffyStudent.\_\_dict\_\_)  
print(stu1.\_\_dict\_\_)  
print(**"属性："**,stu1.x)

结果：

{'\_\_module\_\_': '\_\_main\_\_', 'school': 'luffycity', '\_\_init\_\_': <function LuffyStudent.\_\_init\_\_ at 0x10d5ce8c8>, 'learn': <function LuffyStudent.learn at 0x10d5ce840>, 'eat': <function LuffyStudent.eat at 0x10d5ce7b8>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'LuffyStudent' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'LuffyStudent' objects>, '\_\_doc\_\_': None, 'x': 'from Luffycity class'}

{'Name': '王二丫', 'Sex': '女', 'Age': 18, 'x': 'from stu1'}

**属性： from stu1**

## 3.8 补充

### 补充1：怎么定义程序中的类和对象

一般情况下，程序中的类和对象是怎么定义出来的：先找现实世界中的角度，找到到现实世界中的对象，由现实世界中的对象总结出现实世界中的类，到程序当中定义程序中的类，在实例化成程序中的对象，进而去程序当中去用。

但是：

* 站的角度不同，定义出的类是截然不同的；
* 现实中的类并不完全等于程序中的类，比如现实中的公司类，在程序中有时需要拆分成部门类，业务类等；
* 有时为了编程需求，程序中也可能会定义现实中不存在的类，比如策略类，现实中并不存在，但是在程序中却是一个很常见的类。

### 补充2：python一切皆对象

python一切皆对象，在python3里统一类类与类型的概念

比如：列表是 list 的类产生的，自定义的函数是 class 定义产生的。因此，python一切节对象

## 3.9 练习

### 练习1：编写一个学生类，产生一堆学生对象

要求：  
有一个计数器（属性），统计总共实例了多少个对象

**class** Student:  
 school=**'luffycity'** count=0  
  
 **def** \_\_init\_\_(self,name,age,sex):  
 self.name=name  
 self.age=age  
 self.sex=sex  
 Student.count+=1  
  
 **def** learn(self):  
 print(**'%s is learing'** %self.name)  
  
print(**"未绑定实例之前："**,Student.count)  
stu1=Student(**'alex'**,**'male'**,38)  
stu2=Student(**'jinxing'**,**'female'**,78)  
stu3=Student(**'egon'**,**'male'**,18)  
print(**"绑定实例以后："**,Student.count)  
print(stu1.\_\_dict\_\_)

结果：

未绑定实例之前： 0

绑定实例以后： 3

{'name': 'alex', 'age': 'male', 'sex': 38}

结论：每触发一次 \_\_init\_\_ 方法，就相当于实例化出了一个对象。就相当于一个类创建了一个对象，如果，count 是自己的属性，那么别的对象就感受不到，那就会形成 count = 1 ,因此，就要把count 设置为类的属性，那样，在创建一个对象的时候，对象本身的count就自增一次，最后，在调用 Student.count 的时候，就可以查看到创建打了多少次对象。

### 练习2：模仿LoL定义两个英雄类

要求：  
英雄需要有昵称、攻击力、生命值等属性；  
实例化出两个英雄对象；  
英雄之间可以互殴，被殴打的一方掉血，血量小于0则判定为死亡。

**class** Garen:  
 camp=**'Demacia'  
 def** \_\_init\_\_(self,nickname,life\_value,aggresivity):  
 self.nickname=nickname  
 self.life\_value=life\_value  
 self.aggresivity=aggresivity  
 **def** attack(self,enemy):  
 enemy.life\_value-=self.aggresivity  
 *#r1.life\_value-=g1.aggresivity***class** Riven:  
 camp = **'Noxus'  
 def** \_\_init\_\_(self, nickname, life\_value, aggresivity):  
 self.nickname = nickname  
 self.life\_value = life\_value  
 self.aggresivity = aggresivity  
 **def** attack(self, enemy):  
 enemy.life\_value -= self.aggresivity  
  
g1=Garen(**'草丛伦'**,100,30)  
r1=Riven(**'可爱的锐雯雯'**,80,50)  
  
print(r1.life\_value)  
g1.attack(r1)  
print(r1.life\_value)

## 3.10 继承

继承就是解决类与类之间的代码冗余问题，也就是代码重复的问题。

### 什么是继承？

继承指的是类与类之间的关系，**是一种什么“是”什么的关系**，继承的功能之一就是用来解决代码重用问题

继承是一种创建新类的方式，在python中，新建的类可以继承一个或多个父类，父类又可以称为基类或超类，新建的类称为派生类或子类

### 2.python中类的继承分为：单继承和多继承

**class** ParentClass1:  
 **pass  
  
class** ParentClass2:  
 **pass  
  
class** SubClass1(ParentClass1):  
 **pass  
  
class** SubClass2(ParentClass1,ParentClass2):  
 **pass**print(SubClass1.\_\_bases\_\_) *#查看继承的父类*print(SubClass2.\_\_bases\_\_)

结果：

(<class '\_\_main\_\_.ParentClass1'>,)

(<class '\_\_main\_\_.ParentClass1'>, <class '\_\_main\_\_.ParentClass2'>)

**查看继承的父类：\_\_bases\_\_**

### 3. 经典类与新式类

1.只有在python2中才分新式类和经典类，python3中统一都是新式类

2.在python2中，没有显式的继承object类的类，以及该类的子类，都是经典类

3.在python2中，显式地声明继承object的类，以及该类的子类，都是新式类

4.在python3中，无论是否继承object，都默认继承object，即python3中所有类均为新式类

提示：如果没有指定基类，python的类会默认继承object类，object是所有python类的基类，它提供了一些常见方法（如\_\_str\_\_）的实现。

>>> ParentClass1.\_\_bases\_\_

(<class 'object'>,)

>>> ParentClass2.\_\_bases\_\_

(<class 'object'>,)

### 4. 继承与抽象（先抽象再继承）

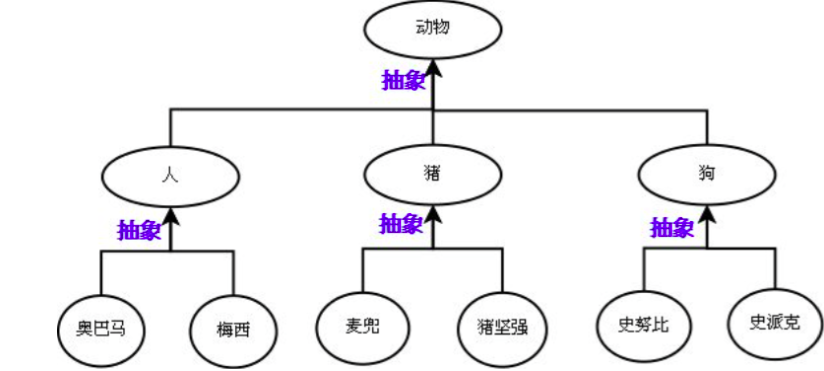
抽象即抽取类似或者说比较像的部分。

抽象分成两个层次：

1.将奥巴马和梅西这俩对象比较像的部分抽取成类；

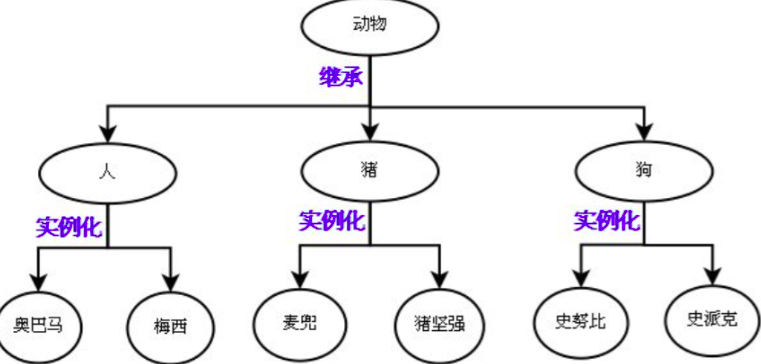
2.将人，猪，狗这三个类比较像的部分抽取成父类。

抽象最主要的作用是划分类别（可以隔离关注点，降低复杂度）



继承：是基于抽象的结果，通过编程语言去实现它，肯定是先经历抽象这个过程，才能通过继承的方式去表达出抽象的结构。

抽象只是分析和设计的过程中，一个动作或者说一种技巧，通过抽象可以得到类



### 5.继承与重用性

在开发程序的过程中，如果我们定义了一个类A，然后又想新建立另外一个类B，但是类B的大部分内容与类A的相同时

我们不可能从头开始写一个类B，这就用到了类的继承的概念。

通过继承的方式新建类B，让B继承A，B会‘遗传’A的所有属性(数据属性和函数属性)，实现代码重用

class Hero:

def \_\_init\_\_(self,nickname,aggressivity,life\_value):

self.nickname=nickname

self.aggressivity=aggressivity

self.life\_value=life\_value

def move\_forward(self):

print('%s move forward' %self.nickname)

def move\_backward(self):

print('%s move backward' %self.nickname)

def move\_left(self):

print('%s move forward' %self.nickname)

def move\_right(self):

print('%s move forward' %self.nickname)

def attack(self,enemy):

enemy.life\_value-=self.aggressivity

class Garen(Hero):

pass

class Riven(Hero):

pass

g1=Garen('草丛伦',100,300)

r1=Riven('锐雯雯',57,200)

print(g1.life\_value) #结果:300

r1.attack(g1)

print(g1.life\_value) #结果:243

提示：用已经有的类建立一个新的类，这样就重用了已经有的软件中的一部分设置大部分，大大节省了编程工作量，这就是常说的软件重用，不仅可以重用自己的类，也可以继承别人的，比如标准库，来定制新的数据类型，这样就是大大缩短了软件开发周期，对大型软件开发来说，意义重大.

### 6.继承的属性查找

**class** Foo:  
 **def** f1(self):  
 print(**'from Foo.f1'**)  
  
 **def** f2(self):  
 print(**'from Foo.f2'**)  
 self.f1() *# self 现在是 b的对象，然后 得到 b.f1()，那么又从对象b本身开始找***class** Bar(Foo):  
 **def** f1(self):  
 print(**'from Bar.f1'**)  
  
b=Bar()  
print(b.\_\_dict\_\_) *#查看对象本身的 \_\_init\_\_ 方法*b.f2()

结果：

{}

from Foo.f2

from Bar.f1

结论： 之所以为\_\_dict\_\_ 为空是没有定义本身的初始化方法。

查找顺序 1）本身查看（\_\_init\_\_）里面查找。2）去 Bar 里面查找。3）父类查找。4）得到self.f1() 相当于 b.f1() 。5）然后又从本身查找，回到了（1）

## 3.11 派生

当然子类也可以添加自己新的属性或者在自己这里重新定义这些属性（不会影响到父类），需要注意的是，一旦重新定义了自己的属性且与父类重名，那么调用新增的属性时，就以自己为准了。

**class** Hero:  
 **def** \_\_init\_\_(self,nickname,life\_value,aggresivity):  
 self.nickname=nickname  
 self.life\_value=life\_value  
 self.aggresivity=aggresivity  
 **def** attack(self,enemy):  
 enemy.life\_value-=self.aggresivity  
**class** Garen(Hero):  
 camp=**'Demacia'  
 def** attack(self,enemy):  
 print(**'from Garen Class'**)  
**class** Riven(Hero):  
 camp=**'Noxus'**g=Garen(**'草丛伦'**,100,30)  
r=Riven(**'锐雯雯'**,80,50)  
*# print(g.camp)  
# g.attack(r)  
# print(r.life\_value)*g.attack(r)

结果：

from Garen Class

结论：查找顺序还是和继承一样。但是当子类中有了属性以后，那么，就会用子类当中的属性，如果没有就会继承父类中的属性。这个就是派生，也就是，子类中可以重新定义父类的方法，但是在对象调用该方法时，如果有就调用子类本身的属性，如果没有就调用父类的属性。

## 3.12 继承实现原理

python到底是如何实现继承的，对于你定义的每一个类，python会计算出一个方法解析顺序(MRO)列表，这个MRO列表就是一个简单的所有基类的线性顺序列表，例如：

>>> F.mro() #等同于F.\_\_mro\_\_

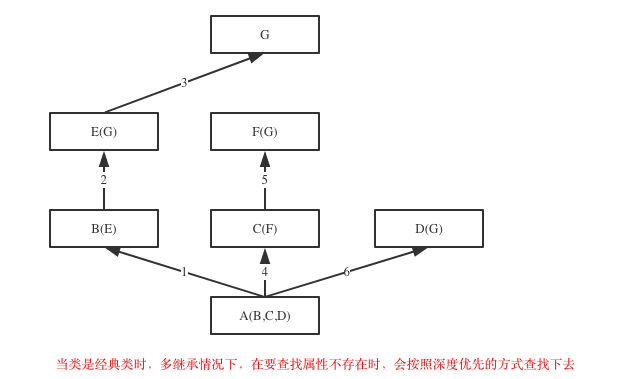
[<class '\_\_main\_\_.F'>, <class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>,

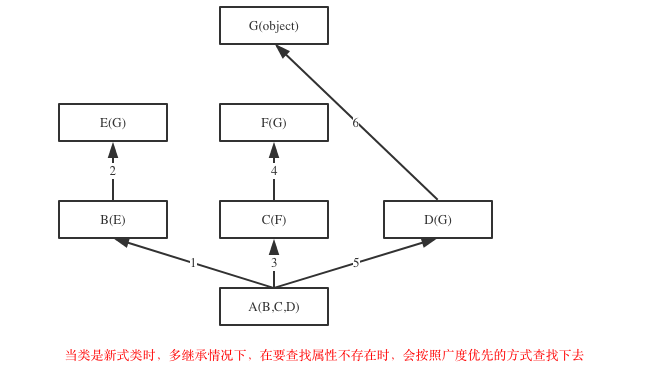
<class '\_\_main\_\_.E'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]

为了实现继承,python会在MRO列表上从左到右开始查找基类,直到找到第一个匹配这个属性的类为止。而这个MRO列表的构造是通过一个C3线性化算法来实现的。我们不去深究这个算法的数学原理,它实际上就是合并所有父类的MRO列表并遵循如下三条准则:

1. 子类会先于父类被检查
2. 多个父类会根据它们在列表中的顺序被检查
3. 如果对下一个类存在两个合法的选择,选择第一个父类

在Java和C#中子类只能继承一个父类，而Python中子类可以同时继承多个父类，如果继承了多个父类，那么属性的查找方式有两种，分别是：深度优先和广度优先





示例代码：

class A(object):

def test(self):

print('from A')

class B(A):

def test(self):

print('from B')

class C(A):

def test(self):

print('from C')

class D(B):

def test(self):

print('from D')

class E(C):

def test(self):

print('from E')

class F(D,E):

# def test(self):

# print('from F')

pass

f1=F()

f1.test()

print(F.\_\_mro\_\_) #只有新式才有这个属性可以查看线性列表，经典类没有这个属性

#新式类继承顺序:F->D->B->E->C->A—>object

#经典类继承顺序:F->D->B->A->E->C

#python3中统一都是新式类

#pyhon2中才分新式类与经典类

## 3.13 在子类中重用父类的属性

**在子类派生出的新的方法中重用父类的方法，有两种实现方式**

### 方式一：指名道姓（不依赖继承）

**class** Hero:  
 **def** \_\_init\_\_(self,nickname,life\_value,aggresivity):  
 self.nickname=nickname  
 self.life\_value=life\_value  
 self.aggresivity=aggresivity  
 **def** attack(self,enemy):  
 enemy.life\_value-=self.aggresivity  
  
**class** Garen(Hero):  
 camp=**'Demacia'  
 def** attack(self,enemy):  
 Hero.attack(self,enemy) *#指名道姓，不以来继承，相当于用了另外一个类的方法。* print(**'from Garen Class'**)  
  
**class** Riven(Hero):  
 camp=**'Noxus'**g=Garen(**'草丛伦'**,100,30)  
r=Riven(**'锐雯雯'**,80,50)  
  
print(r.life\_value)  
g.attack(r)  
print(r.life\_value)

结果：

80

from Garen Class

50

初始化的时候，在调用父类的 \_\_init\_\_ 方法。

**class** Hero:  
 **def** \_\_init\_\_(self,nickname,life\_value,aggresivity):  
 self.nickname=nickname  
 self.life\_value=life\_value  
 self.aggresivity=aggresivity  
 **def** attack(self,enemy):  
 enemy.life\_value-=self.aggresivity  
  
**class** Garen(Hero):  
 camp=**'Demacia'  
 def** \_\_init\_\_(self,nickname,life\_value,aggresivity,weapon):  
 *# self.nickname=nickname  
 # self.life\_value=life\_value  
 # self.aggresivity=aggresivity* Hero.\_\_init\_\_(self,nickname,life\_value,aggresivity)  
 self.weapon=weapon  
  
 **def** attack(self,enemy):  
 Hero.attack(self,enemy) *#指名道姓* print(**'from Garen Class'**)  
  
  
g=Garen(**'草丛伦'**,100,30,**'金箍棒'**)  
print(g.\_\_dict\_\_)

结果：{'nickname': '草丛伦', 'life\_value': 100, 'aggresivity': 30, 'weapon': '金箍棒'}

结论：

上面两种方法可以这样使用，但是不依赖于继承原理。是一种指名道姓的使用方法，相当于直接指定某个类的某方法Hero.\_\_init\_\_(self,nickname,life\_value,aggresivity) 这里就相当于上面子类里面注释掉的self.属性 的内容。因此，应该有更好的方法去重用父类的方法。

### 方式二：super() (依赖继承)

**class** Hero:  
 **def** \_\_init\_\_(self,nickname,life\_value,aggresivity):  
 self.nickname=nickname  
 self.life\_value=life\_value  
 self.aggresivity=aggresivity  
 **def** attack(self,enemy):  
 enemy.life\_value-=self.aggresivity  
  
**class** Garen(Hero):  
 camp=**'Demacia'  
  
 def** \_\_init\_\_(self,nickname,life\_value,aggresivity,weapon):  
 *# super(Garen,self).\_\_init\_\_(nickname,life\_value,aggresivity)* super().\_\_init\_\_(nickname,life\_value,aggresivity) *#super qu mro中找* self.weapon=weapon  
  
 **def** attack(self,enemy):  
 super().attack(enemy) *#实际上是这样用的 super(Garen,self).attack(enemy)* print(**'from Garen Class'**)  
**class** Riven(Hero):  
 camp=**'Noxus'**g=Garen(**'草丛伦'**,200,50,**'金箍棒'**)  
print(Garen.\_\_mro\_\_)  
print(g.\_\_dict\_\_)  
  
r=Riven(**'锐雯雯'**,80,50)  
g.attack(r)  
print(r.life\_value)

结果：

(<class '\_\_main\_\_.Garen'>, <class '\_\_main\_\_.Hero'>, <class 'object'>)

{'nickname': '草丛伦', 'life\_value': 200, 'aggresivity': 50, 'weapon': '金箍棒'}

from Garen Class

30

结论：在子类中要重用父类的方法有两种方式，但是，根据python class的特性，用到super是最好的方法，因为它依赖于继承。为什么？因为，只要是一个子类，那么它就会继承一个父类，他就会有一个 \_\_mro\_\_ 列表，super 就是从mro列表当中去找属性。

### 三：super（）的查找顺序

**class** A:  
 **def** f1(self):  
 print(**'from A'**)  
 super().f1() *#此时位置在mro列表中的 class\_A 位置，因为super会沿着 mro列表往后找，因此，会找到class\_B***class** B:  
 **def** f1(self):  
 print(**'from B'**)  
  
**class** C(A,B):  
 **pass**print(C.mro())  
c=C()  
c.f1()

结果：

[<class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.B'>, <class 'object'>]

from A

from B

结论：super 的查找顺序是按照 \_\_mro\_\_ 列表往后找，就像上例一样，在A中super().f1() 此时，虽然 A没有继承B，但是，在super 这个方式，是由c对象触发的，因此，super的查找方法，会按照，c对象的 \_\_mro\_\_ 列表往后找。因此，就会打印 from B;

A 和 B 无关乎于继承，只是在C中继承了A,B 导致了后面super的查找顺序。该例子一般是特例。

## 3.13 组合

软件重用的重要方式除了继承之外还有另外一种方式，即：组合

组合指的是，在一个类中以另外一个类的对象作为数据属性，称为类的组合

**组合与继承都是有效地利用已有类的资源的重要方式。但是二者的概念和使用场景皆不同。**

代码示例：下面的操作都基于此代码完成；

**class** People:  
 school=**'luffycity'  
 def** \_\_init\_\_(self,name,age,sex):  
 self.name=name  
 self.age=age  
 self.sex=sex  
  
**class** Teacher(People):  
 **def** \_\_init\_\_(self,name,age,sex,level,salary,):  
 super().\_\_init\_\_(name,age,sex)  
 self.level=level  
 self.salary=salary  
  
 **def** teach(self):  
 print(**'%s is teaching'** %self.name)  
  
**class** Student(People):  
 **def** \_\_init\_\_(self, name, age, sex, class\_time,):  
 super().\_\_init\_\_(name,age,sex)  
 self.class\_time=class\_time  
 **def** learn(self):  
 print(**'%s is learning'** % self.name)  
  
**class** Course:  
 **def** \_\_init\_\_(self,course\_name,course\_price,course\_period):  
 self.course\_name = course\_name  
 self.course\_price = course\_price  
 self.course\_period = course\_period  
 **def** tell\_info(self):  
 print(**'课程名<%s> 课程价钱<%s> 课程周期<%s>'** %(self.course\_name,self.course\_price,self.course\_period))

### 操作1：类的定义，及自己理解的组合

为何要这样定义该类：因为，老师以及学生都有共同的属性，例如：姓名，年龄，性别等。但是他们之间又有不同的属性，例如：老师授课，学生听课等。因此，就要定义一个People类，然后在定义各自的类，这样就减少了代码的重复性。

其次，定义一个课程类的原因是，因为，老师和学生之间是有一个交集的关系，如果在各自的类里面定义一门课程的名字等属性，那么又会重复写代码。并且，课程有多个，因此，单独的去写一个course的类，在把这个类实例化成具体的课程，把这个实例化后的课程，赋值给老师或者学生对象，这个就是下面将要讲的组合。

先表述一下自己了解的组合：在类A 与类B里面，类A没有x的属性，但是，类B里面有x属性对应的想要的内容。因此，就会 M=A（），先实例化对象M（类A的对象），N=B（），然后，M.x = N，这就是组合,这样就可以在M这个对象里面，有了x 的属性，去访问N 里面的属性或方法。（对象（A）.属性=对象（B））。顺便提一下，组合应该是一种什么有什么的现象。比如，飞机可以飞，如果人的这个对象去组合飞机的对象，那就得到了人可以飞，这个不是现实生活中的对象应该具有的类的特征。所以，不可以。

### 操作2：组合的使用

teacher1=Teacher(**'alex'**,18,**'male'**,10,3000,) *# 实例化教师1对象*teacher2=Teacher(**'egon'**,28,**'male'**,30,3000,) *# 实例化教师2对象*python=Course(**'python'**,3000,**'3mons'**) *# 实例化课程对象*linux=Course(**'linux'**,2000,**'4mons'**) *# 实例化课程对象*teacher1.course=python *#相当于给教师1，添加一个属性*teacher2.course=python *#相当于给教师1，添加一个属性*print(**"实例化的对象："**,python)  
print(**"实例化对象的属性："**,python.course\_name)  
print(**"教师1的课程对象："**,teacher1.course)  
print(**"教师2的课程对象："**,teacher2.course)  
print(**"教师1的课程名称："**,teacher1.course.course\_name)  
print(**"教师2的课程名称："**,teacher2.course.course\_name)

teacher1.course.tell\_info()  
print(teacher1.course.\_\_dict\_\_)

结果：

实例化的对象： <\_\_main\_\_.Course object at 0x00000215C37641D0>

实例化对象的属性： python

教师1的课程对象： <\_\_main\_\_.Course object at 0x00000215C37641D0>

教师2的课程对象： <\_\_main\_\_.Course object at 0x00000215C37641D0>

教师1的课程名称： python

教师2的课程名称： python

课程名<python> 课程价钱<3000> 课程周期<3mons>

{'course\_name': 'python', 'course\_price': 3000, 'course\_period': '3mons'}

结论：可以看到，实例化的对象都是内存地址，只有调用该属性的时候，才可以得到它的值。

因此可以看出组合. 对象.属性 = 内存地址 。（赋值的是一个内存地址）

### 操作3：学生学习多门课程的组合使用

student1=Student(**'张三'**,28,**'female'**,**'08:30:00'**) *# 实例化一个student1的对象*python=Course(**'python'**,3000,**'3mons'**) *# 实例化课程对象*linux=Course(**'linux'**,2000,**'4mons'**) *# 实例化课程对象*student1.course1=python *# 给学生添加了一个python课程属性*student1.course2=linux *# 给学生添加了一个linux课程的属性*student1.course1.tell\_info()  
student1.course2.tell\_info()

结果：

课程名<python> 课程价钱<3000> 课程周期<3mons>

课程名<linux> 课程价钱<2000> 课程周期<4mons>

上面的操作还可以用列表形式添加课程：

结论：用法和操作2一样。

student1=Student(**'张三'**,28,**'female'**,**'08:30:00'**) *# 实例化一个student1的对象*python=Course(**'python'**,3000,**'3mons'**) *# 实例化课程对象*linux=Course(**'linux'**,2000,**'4mons'**) *# 实例化课程对象*

student1.courses=[]  
student1.courses.append(python)  
student1.courses.append(linux)

student1.courses=[]  
student1.courses.append(python)  
student1.courses.append(linux)  
  
**for** num **in** range(len(student1.courses)):  
 *# print(student1.courses[num].\_\_dict\_\_)  
 # print(student1.courses[num].course\_name)* student1.courses[num].tell\_info()

结果：

课程名<python> 课程价钱<3000> 课程周期<3mons>

课程名<linux> 课程价钱<2000> 课程周期<4mons>

操作4：还可以在上面示例代码中添加一些日期类，用于打印生日等。操作也是组合类似于上面的2，3 操作一样。

总结：

1. 继承的方式

通过继承建立了派生类与基类之间的关系，它是一种'是'的关系，比如白马是马，人是动物。

当类之间有很多相同的功能，提取这些共同的功能做成基类，用继承比较好，比如老师是人，学生是人

1. 组合的方式

用组合的方式建立了类与类之间的关系，它是一种‘有’的关系,比如教授有生日，教授教python和linux课程，教授有学生s1、s2、s3...

**3．总结：**当类之间有显著不同，并且较小的类是较大的类所需要的组件时，用组合比较好

## 3.14 抽象类

* 接口与归一化设计
* 抽象类

### 1.什么是接口

接口指的是：自己提供给使用者来调用自己功能的方式\方法\入口。

### 2. 为何要用接口

接口提取了一群类共同的函数，可以把接口当做一个函数的集合。

然后让子类去实现接口中的函数。

这么做的意义在于归一化，什么叫归一化，就是只要是基于同一个接口实现的类，那么所有的这些类产生的对象在使用时，从用法上来说都一样。

归一化的好处在于：

1. 归一化让使用者无需关心对象的类是什么，只需要的知道这些对象都具备某些功能就可以了，这极大地降低了使用者的使用难度。
2. 归一化使得高层的外部使用者可以不加区分的处理所有接口兼容的对象集合
   1. 就好象linux的泛文件概念一样，所有东西都可以当文件处理，不必关心它是内存、磁盘、网络还是屏幕（当然，对底层设计者，当然也可以区分出“字符设备”和“块设备”，然后做出针对性的设计：细致到什么程度，视需求而定）。
   2. 再比如：我们有一个汽车接口，里面定义了汽车所有的功能，然后由本田汽车的类，奥迪汽车的类，大众汽车的类，他们都实现了汽车接口，这样就好办了，大家只需要学会了怎么开汽车，那么无论是本田，还是奥迪，还是大众我们都会开了，开的时候根本无需关心我开的是哪一类车，操作手法（函数调用）都一样

### **3. 模仿java 中的 interface**

方法1：可以借助第三方模块：<http://pypi.python.org/pypi/zope.interface>

方法2：也可以使用继承，其实继承有两种用途

一：继承基类的方法，并且做出自己的改变或者扩展（代码重用）：实践中，继承的这种用途意义并不很大，甚至常常是有害的。因为它使得子类与基类出现强耦合。

二：声明某个子类兼容于某基类，定义一个接口类（模仿java的Interface），接口类中定义了一些接口名（就是函数名）且并未实现接口的功能，子类继承接口类，并且实现接口中的功能

### 4．抽象类

与java一样，python也有抽象类的概念但是同样需要借助模块实现，抽象类是一个特殊的类，它的特殊之处在于只能被继承，不能被实例化

### 5． 为什么要有抽象类

如果说类是从一堆对象中抽取相同的内容而来的，那么抽象类就是从一堆类中抽取相同的内容而来的，内容包括数据属性和函数属性。

　 比如我们有香蕉的类，有苹果的类，有桃子的类，从这些类抽取相同的内容就是水果这个抽象的类，你吃水果时，要么是吃一个具体的香蕉，要么是吃一个具体的桃子。。。。。。你永远无法吃到一个叫做水果的东西。

从设计角度去看，如果类是从现实对象抽象而来的，那么抽象类就是基于类抽象而来的。

　 从实现角度来看，抽象类与普通类的不同之处在于：**抽象类中只能有抽象方法（没有实现功能），该类不能被实例化，只能被继承，且子类必须实现抽象方法。**这一点与接口有点类似，但其实是不同的，即将揭晓答案

### 6．在python中实现抽象类

**import** abc  
**class** Animal(metaclass=abc.ABCMeta): *#只能被继承，不能被实例化* all\_type=**'animal'  
  
 @abc.abstractmethod**  
 **def** run(self): *#之定义名字，不实现代码* **pass**

**@abc.abstractmethod**  
 **def** eat(self):  
 **pass***# animal=Animal() #这里调用会报错 TypeError，抽象类只能被继承不能被调用***class** People(Animal):  
 **def** run(self):  
 print(**'people is running'**)  
 **def** eat(self):  
 print(**'people is eating'**)  
 **def** sleep(self):  
 print(**'people is sleep'**)  
  
**class** Pig(Animal):  
 **def** run(self):  
 print(**'people is walking'**)  
 **def** eat(self):  
 print(**'people is eating'**)  
  
**class** Dog(Animal):  
 **def** run(self):  
 print(**'people is walking'**)  
  
 **def** eat(self):  
 print(**'people is eating'**)  
  
  
peo1=People()  
pig1=Pig()  
dog1=Dog()  
  
peo1.eat()  
pig1.eat()  
dog1.eat()  
  
print(peo1.all\_type)  
peo1.sleep()  
print(peo1.all\_type)

结果：

people is eating

people is eating

people is eating

animal

people is sleep

animal

结论：根据结果和对象调用方法可以看出，实例化后的对象，调用的方法都是一样的，这样的好处在于，用户体验较好。

在抽象类Animal，只能被继承，不能被实例化。并且，在抽象类中定义的方法，只定义名字，不实现代码的逻辑程序。并且，在抽象类中定义的方法名，要在继承它的子类中也存在。在子类中去实现该方法的逻辑程序。但是子类里面可以有别的方法。

在抽象类中定义的数据属性，可以在子类中获取到。

### 7. 抽象类与接口

抽象类的本质还是类，指的是一组类的相似性，包括数据属性（如all\_type）和函数属性（如 run，eat），而接口只强调函数属性的相似性。

抽象类是一个介于类和接口直接的一个概念，同时具备类和接口的部分特性，可以用来实现归一化设

## 3.15 多态与多态性

### 1. 多态

多态指的是一类事物有多种形态

动物有多种形态：人，狗，猪

示例代码：下面的多态性将会用到该代码。

import abc

class Animal(metaclass=abc.ABCMeta): #同一类事物:动物

@abc.abstractmethod

def talk(self):

pass

class People(Animal): #动物的形态之一:人

def talk(self):

print('say hello')

class Dog(Animal): #动物的形态之二:狗

def talk(self):

print('say wangwang')

class Pig(Animal): #动物的形态之三:猪

def talk(self):

print('say aoao')

文件有多种形态：文本文件，可执行文件

import abc

class File(metaclass=abc.ABCMeta): #同一类事物:文件

@abc.abstractmethod

def click(self):

pass

class Text(File): #文件的形态之一:文本文件

def click(self):

print('open file')

class ExeFile(File): #文件的形态之二:可执行文件

def click(self):

print('execute file')

### 2．多态性

多态性是指在不考虑实例类型的情况下使用实例，多态性分为静态多态性和动态多态性

### 3. 动态多态性

用到上面多态的示例代码：

*# 动态多态性*peo1=People()  
dog1=Dog()  
pig1=Pig()  
  
  
peo1.talk()  
dog1.talk()  
pig1.talk()

结果：

say hello

say wangwang

say aoao

结论：不考虑对象的类型，只要是继承了某个类的方法，就可以按照类的标准进行使用，这个就是动态多态性。

### 4. 静态多态性

静态多态性：如任何类型都可以用运算符+进行运算

peo1=People()  
dog1=Dog()  
pig1=Pig()  
cat1=Cat()**def** func(animal):  
 animal.talk()  
  
func(peo1)  
func(pig1)  
func(dog1)  
func(cat1)

结果：

say hello

say aoao

say wangwang

say miamiao

结论：不用考虑对象具体的类型，直接调用一个方法就行。

### 5. 为什么要用多态性（多态性的好处）

其实大家从上面多态性的例子可以看出，我们并没有增加什么新的知识，也就是说python本身就是支持多态性的，这么做的好处是什么呢？

1.增加了程序的灵活性

　　以不变应万变，不论对象千变万化，使用者都是同一种形式去调用，如func(animal)

2.增加了程序额可扩展性

　 通过继承animal类创建了一个新的类，使用者无需更改自己的代码，还是用func(animal)去调用

## 3.16 鸭子类型

Python崇尚鸭子类型，即‘如果看起来像、叫声像而且走起路来像鸭子，那么它就是鸭子’

python程序员通常根据这种行为来编写程序。例如，如果想编写现有对象的自定义版本，可以继承该对象，也可以创建一个外观和行为，但与它无任何关系的全新对象，后者通常用于保存程序组件的松耦合度。

例1：利用标准库中定义的各种‘与文件类似’的对象，尽管这些对象的工作方式像文件，但他们没有继承内置文件对象的方法

**class** Disk:  
 **def** read(self):  
 print(**'disk read'**)  
 **def** write(self):  
 print(**'disk write'**)  
  
**class** Text:  
 **def** read(self):  
 print(**'text read'**)  
 **def** write(self):  
 print(**'text write'**)  
  
disk=Disk()  
text=Text()  
  
disk.read()  
disk.write()  
  
text.read()  
text.write()

结论：二者都像鸭子,二者看起来都像文件,因而就可以当文件一样去用

例2：序列类型有多种形态：字符串，列表，元组，但他们之间没有直接的继承关系

#str,list,tuple都是序列类型

s=str('hello')

l=list([1,2,3])

t=tuple((4,5,6))

#我们可以在不考虑三者类型的前提下使用s,l,t

s.\_\_len\_\_()

l.\_\_len\_\_()

t.\_\_len\_\_()

len(s)

len(l)

len(t)

结论：上面的字符串，列表，元组，都是同一种事物（序列）的三种形态。他们都是序列类型，因此不管他们是什么类型，但他们都有一个len的方法。上面就是python鸭子类型的一种体现。

## 3.17 封装

* 掌握封装，封装与扩展性
* 掌握property

### 引子

从封装本身的意思去理解，封装就好像是拿来一个麻袋，把小猫，小狗，小王八，还有alex一起装进麻袋，然后把麻袋封上口子。照这种逻辑看，封装=‘隐藏’，这种理解是相当片面的。

### 2. 如何隐藏属性

在python中用双下划线开头的方式将属性隐藏起来（设置成私有的）

#其实这仅仅这是一种变形操作

#类中所有双下划线开头的名称如\_\_x都会自动变形成：\_类名\_\_x的形式：

class A:

\_\_N=0 #类的数据属性就应该是共享的,但是语法上是可以把类的数据属性设置成私有的如\_\_N,会变形为\_A\_\_N

def \_\_init\_\_(self):

self.\_\_X=10 #变形为self.\_A\_\_X

def \_\_foo(self): #变形为\_A\_\_foo

print('from A')

def bar(self):

self.\_\_foo() #只有在类内部才可以通过\_\_foo的形式访问到.

#A.\_A\_\_N是可以访问到的，即这种操作并不是严格意义上的限制外部访问，仅仅只是一种语法意义上的变形

示例代码：

**class** A:  
 \_\_x=1 *#在类被定义的时候已经变形为：\_A\_\_x* **def** \_\_init\_\_(self,name):  
 self.\_\_name = name *#变形为：\_A\_\_name* **def** \_\_foo(self): *#变形为： \_A\_\_foo* print(**'A.foo'**)  
  
 **def** tell\_info(self):  
 print(**"隐藏的属性x:"**,self.\_\_x)  
 print(**"隐藏的属性name:"**,self.\_\_name)  
  
 **def** bar(self):  
 print(**'A.bar'**)  
 self.\_\_foo() *#变形为： self.\_A\_\_foo()***class** B(A):  
 **def** \_\_foo(self): *#变形为： \_B\_\_foo* print(**'B.foo'**)

先得到结论，在去总结结论的特点：

**'''  
封装不是单纯意义上的隐藏。  
变量名或者方法名前面加上双下划线（“\_\_”）就可以实现隐藏。就类定义阶段就会发生变形。  
这种变形的特点：  
 1、在类外部无法直接obj.\_\_AttrName  
 2、在类内部是可以直接使用：obj.\_\_AttrName (在定义阶段已经正确访问了)  
 3、子类无法覆盖父类\_\_开头的属性  
'''**

### 操作1：查看隐藏的属性和正确调用隐藏属性

print(A.\_\_dict\_\_)

结果：

{'\_\_module\_\_': '\_\_main\_\_', **'\_A\_\_x'**: 1, '\_\_init\_\_': <function A.\_\_init\_\_ at 0x0000012F7CFCAAE8>, '\_A\_\_foo': <function A.\_\_foo at 0x0000012F7CFCAB70>, 'bar': <function A.bar at 0x0000012F7CFCABF8>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'A' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'A' objects>, '\_\_doc\_\_': None}

可以看到 “\_\_x” 已经被变形了。

调用隐藏的属性

obj=A(**'ryan'**)  
**'''错误调用隐藏的方法'''**print(obj.\_\_x) *#AttributeError: 'A' object has no attribute '\_\_x'***'''**

**正确的调用隐藏的方法：**

**但是在python中，如果在类里面已经设置好了隐藏属性，那在外部就尽量  
不要改变，如果要求调用或者改变它，那就直接不要“\_\_”就可以了**

**'''**print(obj.\_A\_\_x)

结论：因此得到特点1，在外部无法直接访问 obj.\_\_AttrNmae 。如果一定要访问使用：obj.\_类名\_\_属性

### 操作2：在内部可以正常使用隐藏的属性。

obj=A(**'ryan'**)  
obj.tell\_info()

结果：

隐藏的属性x: 1

隐藏的属性name: ryan

结论：因此，在类内部是可以直接访问带有 “\_\_”的属性的，因为隐藏属性在定义结束以后（类所有内容读取到内存以后）才会被变形，但是，在类内部就可以正常使用隐藏的属性

### 操作3：类定义完成以后设置的“\_\_”的属性正常访问。

obj=A(**'ryan'**)  
print(obj.\_\_dict\_\_)  
obj.\_\_age = 18  
print(obj.\_\_dict\_\_)

结果：

{'\_A\_\_name': 'ryan'}

{'\_A\_\_name': 'ryan', '\_\_age': 18}

结论: 类里面的隐藏只有在类被定义的时候才会发生变形。如果在类被定义完成以后，已经变形结束了，此时如果在进行一次 “\_\_X” 属性添加，那就是正常访问，并未被变形。

### 操作4：子类无法覆盖父类的 “\_\_”开头的属性

例如，示例代码里面的 \_\_foo 方法。

在父类 A类里面 \_\_foo(self): 变形为： \_A\_\_foo

在子类 B类里面 \_\_foo(self): 变形为： \_B\_\_foo

他们在被定义后变形的名称已经发生改变，并且不同，因此，子类无法覆盖父类带 “\_\_”开头的方法。

示例1：

**class** A:  
 **def** foo(self):  
 print(**'A.foo'**)  
  
 **def** bar(self):  
 print(**'A.bar'**)  
 self.foo() *#b.foo()***class** B(A):  
 **def** foo(self):  
 print(**'B.foo'**)  
  
b=B()  
b.bar()

结果：

A.bar

B.foo

结论：这个就是前面的继承时代码的执行过程，此时在 self.foo() 是因为这个由 b这个对象触发的，因此，就会先从本身-->自身类里面去找，找到即结束！

那要求self.foo() 一定使用父类的 foo() 方法呢？

**class** A:  
 **def** \_\_foo(self): *# \_A\_\_foo()* print(**'A.foo'**)  
  
 **def** bar(self):  
 print(**'A.bar'**)  
 self.\_\_foo() *#self.\_A\_\_foo()***class** B(A):  
 **def** \_\_foo(self): *#\_B\_\_foo()* print(**'B.foo'**)  
  
b=B()  
b.bar()

结果：

A.bar

A.foo

结论：之所会这样，是因为，在A类里面，self.\_\_foo() 在类被定义的时候就已经把它给变形成为了 self.\_A\_\_foo() 因此，此时调用就是调用A类里面的 \_\_foo

## 3.18 封装不是单纯意义的隐藏

封装有两种：1.封装数据属性。2.封装函数方法。

### 1.封装数据属性及自己理解的意义所在

封装数据属性：明确的区分内外，控制外部对隐藏的属性的操作行为

**class** People:  
 **def** \_\_init\_\_(self,name,age):  
 self.\_\_name=name  
 self.\_\_age=age  
  
 **def** tell\_info(self): *#外部查看内部隐藏属性的一个接口* print(**'Name:<%s> Age:<%s>'** %(self.\_\_name,self.\_\_age))  
  
 **def** set\_info(self,name,age): *#外部修改内部隐藏属性一个接口* **if not** isinstance(name,str): *#对修改属性时，进行值的判断* print(**'名字必须是字符串类型'**)  
 **return  
 if not** isinstance(age,int): *#对修改属性时，进行值的判断* print(**'年龄必须是数字类型'**)  
 **return** self.\_\_name=name  
 self.\_\_age=age  
  
p=People(**'egon'**,18)  
p.tell\_info() *#查看内部隐藏属性*p.set\_info(**'Ryan'**,28)  
p.tell\_info()

运行结果：

Name:<egon> Age:<18>

Name:<Ryan> Age:<28>

结论：在做这个操作的时候，自己发现了一个问题，在set\_info 的时候，在传入的参数，以及修改属性值的时候，其实跟 \_\_init\_\_（self） 下面的代码不都差不多吗？那为啥，不直接重新赋值一个对象就可以进行修改操作呢。这个想法一想起来就一发不可收拾，但是最后意识到却很幼稚。比如，一个网页在注册好用户以后，再去修改用户名或者性别等，不就相当于重新注册了吗？这样显然是不可以的，因此，类内部要对外提供一个修改隐藏属性的接口，让它可以修改类内部的隐藏属性。我想，这就是隐藏属性或方法的一个意义所在。

### 2. 封装方法：隔离复杂度

**class** ATM:  
 **def** \_\_card(self):  
 print(**'插卡'**)  
 **def** \_\_auth(self):  
 print(**'用户认证'**)  
 **def** \_\_input(self):  
 print(**'输入取款金额'**)  
 **def** \_\_print\_bill(self):  
 print(**'打印账单'**)  
 **def** \_\_take\_money(self):  
 print(**'取款'**)  
  
 **def** withdraw(self):  
 self.\_\_card()  
 self.\_\_auth()  
 self.\_\_input()  
 self.\_\_print\_bill()  
 self.\_\_take\_money()  
  
a=ATM()  
a.withdraw()

直接结果：

插卡

用户认证

输入取款金额

打印账单

取款

结论总结：对于ATM项目来说，用户不在乎你怎么调用的，用户只需要执行取款的功能即可。这样就可以在内部把各个逻辑进行划分。

## 3.19 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

property 可以封装一些通过计算后，在访问的数据类型等封装在函数里面，通过 “对象.属性” 访问。

### property用法例子

*'''  
BMI指数（bmi是计算而来的，但很明显它听起来像是一个属性而非方法，如果我们将其做成一个属性，更便于理解）  
成人的BMI数值：过轻：低于18.5;正常：18.5-23.9;过重：24-27;肥胖：28-32;非常肥胖, 高于32  
体质指数（BMI）=体重（kg）÷身高^2（m）  
EX：70kg÷（1.75×1.75）=22.86  
'''***class** People:  
 **def** \_\_init\_\_(self,name,weight,height):  
 self.name=name  
 self.weight=weight  
 self.height=height  
  
 @property  
 **def** bmi(self):  
 **return** self.weight / (self.height \*\* 2)

p=People(**'cherry'**,65,1.73)print(p.bmi)

p.height=1.82  
print(p.bmi)

执行结果：

21.71806

19.62323

结论：property是一个装饰器，它是让一个方法相当于一个属性来使用。并且他是一组算法（程序）的返回，并不会传入一些参数进去。但是这个被property装饰后的方法不可以像数据属性一样去赋值。

### 2．property 的扩展

**class** People:  
 **def** \_\_init\_\_(self,name):  
 self.\_\_name=name  
  
 @property  
 **def** name(self):  
 **return** self.\_\_name  
  
 @name.setter  
 **def** name(self,val):  
 **if not** isinstance(val,str):  
 print(**'名字必须是字符串类型'**)  
 **return** self.\_\_name=val  
  
 @name.deleter  
 **def** name(self):  
 print(**'不允许删除'**)  
  
p=People(**'egon'**)  
print(**"1)打印name："**,p.name)  
p.name=**'EGON'**p.name=123  
print(**"2)打印name"**,p.name)  
  
**del** p.name

执行结果：

1)打印name： egon

名字必须是字符串类型

2)打印name EGON

不允许删除

结论：后面的name.setter 或者 name.deleter 都是现有 name被property装饰了以后才会有。如果name改名为 A，那setter 就要改写成为 A.setter。该例子其实和上面例子的解释对应，property是给一个把一个方法（函数）作为属性去调用，但是在属性调用的过程中，又不可以传入参数，因此，才会有后面的 setter and delete 。

### 3. 为什么要用property

将一个类的函数定义成特性以后，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**遵循了统一访问的原则**

ps：面向对象的封装有三种方式:

【public】

这种其实就是不封装,是对外公开的

【protected】

这种封装方式对外不公开,但对朋友(friend)或者子类(形象的说法是“儿子”,但我不知道为什么大家 不说“女儿”,就像“parent”本来是“父母”的意思,但中文都是叫“父类”)公开

【private】

这种封装对谁都不公

## 3.20 绑定方法与非绑定方法

在类内部定义的函数，分为两大类：

一：绑定方法:绑定给谁，就应该由谁来调用，谁来调用就会把调用者当作第一个参数自动传入

绑定到对象的方法：在类内定义的没有被任何装饰器修饰的

绑定到类的方法：在类内定义的被装饰器classmethod修饰的方法

二：非绑定方法:没有自动传值这么一说了，就类中定义的一个普通工具，对象和类都可以使用

非绑定方法：不与类或者对象绑定

**class** Foo:  
 **def** \_\_init\_\_(self,name):  
 self.name=name  
  
 **def** tell(self): *#绑定给对象* print(**'名字是%s'** %self.name)  
  
 @classmethod *#绑定给类* **def** func(cls): *#cls=Foo* print(cls)  
  
 @staticmethod *#非绑定方法，类和对象都可以使用（普通函数）* **def** func1(x,y):  
 print(**"x + y ="**,x+y)

f=Foo(**'egon'**)  
  
print(**"类的普通函数："**,Foo.tell) *# 类的一个普通函数 调用：Foo.tell(f)*print(**"绑定到对象的方法："**,f.tell) *# 绑定到对象的方法 调用：f.tell()*print(**"绑定到类的方法："**,Foo.func) *# 绑定到类的方法 Foo.func() #等价于 print(Foo)*print(**"非绑定方法,给类使用:"**,Foo.func1) *# 非绑定方法，给类使用*print(**"非绑定方法,给使用:"**,f.func1) *# 非绑定方法，给对象使用*Foo.func1(1,2) *#非绑定方法，该怎么传参就怎么传参*f.func1(1,3) *#非绑定方法，该怎么传参就怎么传参*

执行结果：

类的普通函数： <function Foo.tell at 0x10e4f5840>

绑定到对象的方法： <bound method Foo.tell of <\_\_main\_\_.Foo object at 0x10e4f3c50>>

绑定到类的方法： <bound method Foo.func of <class '\_\_main\_\_.Foo'>>

非绑定方法,给类使用: <function Foo.func1 at 0x10e4f5950>

非绑定方法,给使用: <function Foo.func1 at 0x10e4f5950>

x + y = 3

x + y = 4

## 3.21 绑定方法与非绑定方法的应用

文件示例：

name=**'alex'**age=18  
sex=**'female'**

代码示例：

**import** settings  
**import** hashlib  
**import** time  
  
**class** People:  
 **def** \_\_init\_\_(self,name,age,sex):  
 self.id=self.create\_id()  
 self.name=name  
 self.age=age  
 self.sex=sex  
  
 **def** tell\_info(self): *#绑定到对象的方法* print(**'Name:%s Age:%s Sex:%s'** %(self.name,self.age,self.sex))  
  
 @classmethod  
 **def** from\_conf(cls):  
 obj=cls(  
 settings.name,  
 settings.age,  
 settings.sex  
 )  
 **return** obj  
  
 @staticmethod  
 **def** create\_id():  
 m=hashlib.md5(str(time.time()).encode(**'utf-8'**))  
 **return** m.hexdigest()  
  
*# p=People('egon',18,'male')  
  
#绑定给对象，就应该由对象来调用，自动将对象本身当作第一个参数传入  
# p.tell\_info() #People('egon',18,'male').tell\_info()  
  
#绑定给类，就应该由类来调用，自动将类本身当作第一个参数传入*

#在用户使用时自动去文件里面去调用一些参数，调用参数本身是通过类去调用文件里面的信息，因此，就要使用类的绑定方法。p=People.from\_conf() *#from\_conf(People)*p.tell\_info()  
  
*#非绑定方法，不与类或者对象绑定，谁都可以调用，没有自动传值一说*p1=People(**'egon1'**,18,**'male'**)  
p2=People(**'egon2'**,28,**'male'**)  
p3=People(**'egon3'**,38,**'male'**)  
  
print(p1.id)  
print(p2.id)  
print(p3.id)

## 3.22 反射

反射：通过字符串映射到对象的属性

### 1. 反射的用法

示例代码：

**class** People:  
 country=**'China'  
  
 def** \_\_init\_\_(self,name,age):  
 self.name=name  
 self.age=age  
  
 **def** talk(self):  
 print(**'%s is talking'** %self.name)  
  
obj=People(**'egon'**,18)  
  
*#判断对象有没有这个属性  
#hasatrr(obj,属性)*print(**"hasattr:"**,hasattr(obj,**'name'**)) *#判断是否有 obj.name 这个属性，本质上是去判断对象的\_\_dict\_\_ 下面是否有这个属性 obj.\_\_dict\_\_['name']*print(**"hasattr:"**,hasattr(obj,**'talk'**)) *#obj.talk  
  
#获取对象的属性  
#getattr(obj,'属性','默认值')*print(**"getattr 对象属性:"**,getattr(obj,**'namexxx'**,**None**))  
print(**"getattr 对象绑定方法:"**,getattr(obj,**'talk'**,**None**))  
  
*#设置对象属性  
# setattr(obj,'属性','value')*setattr(obj,**'sex'**,**'male'**) *#obj.sex='male'*print(**"setattr 设置对象属性"**,obj.sex)  
  
delattr(obj,**'age'**) *#本质上是 del obj.age*print(**"delattr 删除U对象属性"**,obj.\_\_dict\_\_)  
  
print(getattr(People,**'country'**)) *#判断People类中是否有country的属性：People.country*

执行结果：

hasattr: True

hasattr: True

getattr 对象属性: None

getattr 对象绑定方法: <bound method People.talk of <\_\_main\_\_.People object at 0x100b74dd8>>

setattr 设置对象属性 male

delattr 删除U对象属性 {'name': 'egon', 'sex': 'male'}

China

### 2. 反射的应用

**class** Service:  
 **def** run(self):  
 **while True**:  
 inp=input(**'>>: '**).strip() *#cmd='get a.txt'* cmds=inp.split() *#cmds=['get','a.txt']  
 # print(cmds)* **if** hasattr(self,cmds[0]):  
 func=getattr(self,cmds[0])  
 func(cmds)  
  
 **def** get(self,cmds):  
 print(**'get.......'**,cmds[1])  
  
 **def** put(self,cmds):  
 print(**'put.......'**,cmds[1])  
  
obj=Service()  
obj.run()

执行结果：

>>: get a.txt

get....... a.txt

>>: put a.txt

put....... a.txt

## 3.23 内置方法

来源：http://www.cnblogs.com/linhaifeng/articles/6204014.html

### 1. isinstance(obj,cls)和issubclass(sub,super)

**isinstance(obj,cls)检查是否obj是否是类 cls 的对象**

1 class Foo(object):

2 pass

3

4 obj = Foo()

5

6 isinstance(obj, Foo)

**issubclass(sub, super)检查sub类是否是 super 类的派生类**

1 class Foo(object):

2 pass

3

4 class Bar(Foo):

5 pass

6

7 issubclass(Bar, Foo)

### \_\_setitem\_\_,\_\_getitem,\_\_delitem\_\_

item系列,类似于字典的操作。

**class** Foo: *#Dict* **def** \_\_init\_\_(self,name):  
 self.name=name  
  
 **def** \_\_getitem\_\_(self, item): *#item='namexxx'  
 # print('getitem...')* **return** self.\_\_dict\_\_.get(item)  
  
 **def** \_\_setitem\_\_(self, key, value):  
 *# print('setitem...')  
 # print(key,value)* self.\_\_dict\_\_[key]=value  
  
 **def** \_\_delitem\_\_(self, key):  
 *# print('delitem...')  
 # print(key)* **del** self.\_\_dict\_\_[key]  
  
obj=Foo(**'egon'**)  
print(obj.\_\_dict\_\_)  
  
*# 查看属性：  
# obj.属性名*print(obj[**'namexxx'**]) *#obj.name*print(obj[**'name'**]) *#obj.name  
  
# 设置属性：*obj[**'sex'**]=**'male'** *# 等同于 obj.sex='male'*print(**"\_\_dict\_\_:"**,obj.\_\_dict\_\_)  
print(**"sex:"**,obj.sex)  
  
*# 删除属性***del** obj[**'name'**] *#等同于 del obj.name*print(obj.\_\_dict\_\_)

### 3. \_\_str\_\_

改变对象的字符串显示\_\_str\_\_,\_\_repr\_\_

\_\_str\_\_方法： 触发打印对象操作，直接执行这个方法下面的内容。

示例代码：

**class** People:  
 **def** \_\_init\_\_(self,name,age):  
 self.name=name  
 self.age=age  
  
 **def** \_\_str\_\_(self):  
 *# print('====>str')* **return '<name:%s,age:%s>'** %(self.name,self.age)  
  
obj=People(**'egon'**,18)  
print(obj) *#res=obj.\_\_str\_\_()*

执行结果：

<name:egon,age:18>

其余的内置方法总结，查看http://www.cnblogs.com/linhaifeng/articles/6204014.html

## 3.24 元类介绍

了解名称空间：http://blog.csdn.net/xinguimeng/article/details/51002259

### 1. 知识储备

exec：三个参数

参数一：字符串形式的命令

参数二：全局作用域（字典形式），如果不指定，默认为globals()

参数三：局部作用域（字典形式），如果不指定，默认为locals()

#把exec当成一个函数来看待去理解下面的代码  
g={  
 **'x'**:1,  
 **'y'**:2  
}  
  
l={}  
  
exec(**"""  
global x,m  
x=10  
m=100  
  
z=3  
"""**,g,l)  
  
print(**"globals："**,g)  
print(**"locals:"**,l)

执行结果：

globals： {'x': 10, 'y': 2,….}

locals: {'z': 3}

### 2. 引子（python一切皆对象）

一切皆对象，对象可以怎么用？  
1、都可以被引用，x=obj  
2、都可以当作函数的参数传入  
3、都可以当作函数的返回值  
4、都可以当作容器类的元素，l=[func,time,obj,1]

但凡符合上面4个用法就是对象。

*#类也是对象,Foo=type(....)***class** Foo:  
 **pass**obj=Foo()  
print(type(obj))  
print(type(Foo))  
  
**class** Bar:  
 **pass**print(type(Bar))

执行结果：

<class '\_\_main\_\_.Foo'>

<class 'type'>

<class 'type'>

**因此，产生类的类称之为元类，默认所以用class定义的类，他们的元类是type**

### 3．定义类的两种方式之一（class）

**class** Chinese: *#Chinese=type(...)，Chinese实际上是调用了一个类，实例化后得到的一个对象* country=**'China'  
 def** \_\_init\_\_(self,name,age):  
 self.name=name  
 self.age=age  
  
 **def** talk(self):  
 print(**'%s is talking'** %self.name)  
  
print(Chinese)  
obj=Chinese(**'egon'**,18)  
print(obj,obj.name,obj.age)

执行结果：

<class '\_\_main\_\_.Chinese'>

<\_\_main\_\_.Chinese object at 0x10e642828> egon 18

### 4. 定义类的两种方式之二 （type）

**定义类的三要素:类名，类的基类们，类的名称空间(类的名称空间：\_\_dict\_\_)**

class\_name=**'Chinese1'** *#类名*class\_bases=(object,) *#类的基类们  
  
#类的主体*class\_body=**"""  
country='China'  
  
def \_\_init\_\_(self,namem,age):  
 self.name=namem  
 self.age=age  
  
def talk(self):  
 print('%s is talking' %self.name)  
"""**class\_dic={} *#相当于类的 \_\_dict\_\_*exec(class\_body,globals(),class\_dic)  
*#exec(类的主体，全局作用域，局部作用域)*print(class\_dic) *#相当于在class 创建类时，print()*Chinese1=type(class\_name,class\_bases,class\_dic)  
print(Chinese1)  
  
obj1=Chinese1(**'egon'**,18)  
print(obj1,obj1.name,obj1.age)

执行结果：

{'country': 'China', '\_\_init\_\_': <function \_\_init\_\_ at 0x108143048>, 'talk': <function talk at 0x1082f98c8>}

<class '\_\_main\_\_.Chinese1'>

<\_\_main\_\_.Chinese1 object at 0x1082f7ba8> egon 18

结论：该执行结果，和方式一的直接结果一样的。

## 3.25 自定义元类控制类的创建

用元类去控制，类的创建的一个规则

**class** Mymeta(type):  
 **def** \_\_init\_\_(self,class\_name,class\_bases,class\_dic):  
 **if not** class\_name.istitle():  
 **raise** TypeError(**'类名的首字母必须大写'**)  
  
 **if '\_\_doc\_\_' not in** class\_dic **or not** class\_dic[**'\_\_doc\_\_'**].strip():  
 **raise** TypeError(**'必须有注释，且注释不能为空'**)  
  
 super().\_\_init\_\_(class\_name,class\_bases,class\_dic)  
  
**class** Chinese(object,metaclass=Mymeta): *# metaclass 默认等于 type  
 '''  
 中文人的类  
 '''* country=**'China'  
  
 def** \_\_init\_\_(self,namem,age):  
 self.name=namem  
 self.age=age  
  
 **def** talk(self):  
 print(**'%s is talking'** %self.name)

## 3.26 自定义元类控制类的实例化行为

### 引入 \_\_call\_\_ 方法

**class** Mymeta(type):  
 **def** \_\_init\_\_(self,class\_name,class\_bases,class\_dic):  
 **if not** class\_name.istitle():  
 **raise** TypeError(**'类名的首字母必须大写'**)  
  
 **if '\_\_doc\_\_' not in** class\_dic **or not** class\_dic[**'\_\_doc\_\_'**].strip():  
 **raise** TypeError(**'必须有注释，且注释不能为空'**)  
  
 super(Mymeta,self).\_\_init\_\_(class\_name,class\_bases,class\_dic)  
  
 **def** \_\_call\_\_(self, \*args, \*\*kwargs): *#obj=Chinese('egon',age=18)  
 # print(self) #self=Chinese  
 # print(args) #args=('egon',)  
 # print(kwargs) #kwargs={'age': 18}  
  
 #第一件事：先造一个空对象obj* obj=object.\_\_new\_\_(self)  
 *#第二件事：初始化obj* self.\_\_init\_\_(obj,\*args,\*\*kwargs)  
 *#第三件事：返回obj* **return** obj  
  
**class** Chinese(object,metaclass=Mymeta):  
 *'''  
 中文人的类  
 '''* country=**'China'  
  
 def** \_\_init\_\_(self,namem,age):  
 self.name=namem  
 self.age=age  
  
 **def** talk(self):  
 print(**'%s is talking'** %self.name)  
  
obj=Chinese(**'egon'**,age=18) *#Chinese.\_\_call\_\_(Chinese,'egon',18)*print(obj.\_\_dict\_\_)

## 3.27 单例模式

单例模式,对象内部特征一样用单列模式

### 1.单列模式的引入

**class** MySQL:  
 **def** \_\_init\_\_(self):  
 self.host=**'127.0.0.1'** self.port=3306  
  
 **def** conn(self):  
 **pass  
  
 def** execute(self):  
 **pass**obj1 = MySQL()  
obj2 = MySQL()  
obj3 = MySQL()  
print(**"obj1的内存地址"**,id(obj1))  
print(**"obj2的内存地址"**,id(obj2))  
print(**"obj3的内存地址"**,id(obj3))  
  
**'''  
执行结果：  
obj1的内存地址 2024612338096  
obj2的内存地址 2024612338152  
obj3的内存地址 2024612338208  
'''**

结论：可以看到上面的三个对象（obj1,2,3）他们调用的方式一样，参数一样，但是申请的内存空间不一样。因此，就可以使用单例模式，**单例模式就是让，参数一样，调用方式一样的不同对象，在创建时不要新申请内存空间**

### 实现单例模式1

**class** MySQL:  
 \_\_instance=**None** *#\_\_instance=obj1* **def** \_\_init\_\_(self):  
 self.host=**'127.0.0.1'** self.port=3306  
  
 @classmethod  
 **def** singleton(cls):  
 **if not** cls.\_\_instance:  
 obj=cls()  
 cls.\_\_instance=obj  
 **return** cls.\_\_instance  
  
 **def** conn(self):  
 **pass  
  
 def** execute(self):  
 **pass**obj1 = MySQL.singleton()  
obj2 = MySQL.singleton()  
obj3 = MySQL.singleton()  
print(**"obj1的内存地址"**,id(obj1))  
print(**"obj2的内存地址"**,id(obj2))  
print(**"obj3的内存地址"**,id(obj3))  
  
**'''  
执行结果：  
obj1的内存地址 2374233135760  
obj2的内存地址 2374233135760  
obj3的内存地址 2374233135760  
'''**

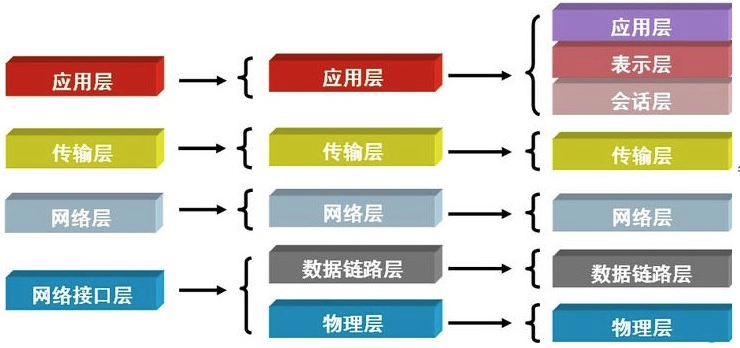
可以看到三个实例化的对象 obj1,2,3 的内存地址空间都是一样的。

### 实现方式2：元类实现单列模式

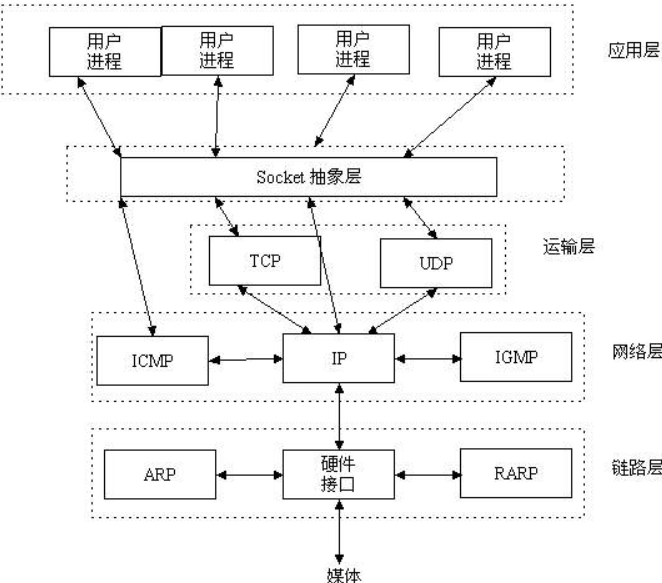
**class** Mymeta(type):  
 **def** \_\_init\_\_(self,class\_name,class\_bases,class\_dic):  
 **if not** class\_name.istitle():  
 **raise** TypeError(**'类名的首字母必须大写'**)  
  
 **if '\_\_doc\_\_' not in** class\_dic **or not** class\_dic[**'\_\_doc\_\_'**].strip():  
 **raise** TypeError(**'必须有注释，且注释不能为空'**)  
  
 super(Mymeta,self).\_\_init\_\_(class\_name,class\_bases,class\_dic)  
 self.\_\_instance=**None  
  
 def** \_\_call\_\_(self, \*args, \*\*kwargs): *#obj=Chinese('egon',age=18)* **if not** self.\_\_instance:  
 obj = object.\_\_new\_\_(self)  
 self.\_\_init\_\_(obj,\*args, \*\*kwargs)  
 self.\_\_instance=obj  
  
 **return** self.\_\_instance  
  
  
**class** Mysql(object,metaclass=Mymeta):  
 *'''  
 mysql xxx  
 '''* **def** \_\_init\_\_(self):  
 self.host=**'127.0.0.1'** self.port=3306  
  
 **def** conn(self):  
 **pass  
  
 def** execute(self):  
 **pass**obj1=Mysql()  
obj2=Mysql()  
obj3=Mysql()  
  
print(obj1 **is** obj2 **is** obj3) #res ：True

# 第三模块 第二章 网络编程

## 1.OSI七层



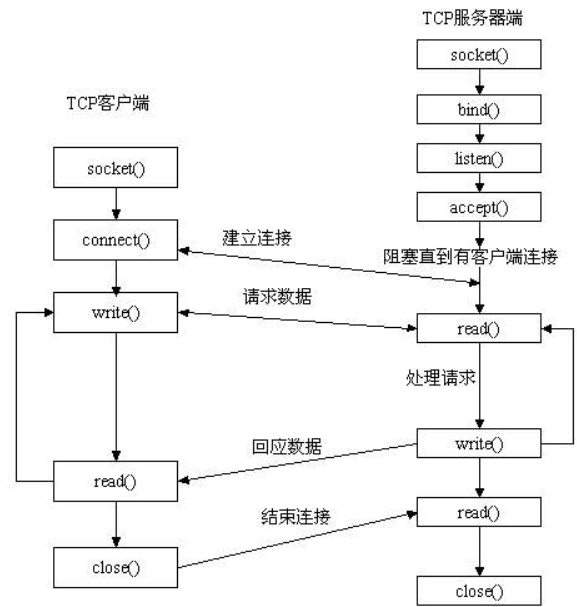
## 2. socket 层



## socket 是什么

Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。

## 4. 套接字工作流程



## 5. 简单的套接字通信

C/S 模式：

### 1.服务端

**import** socket  
  
#1、买手机  
phone=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  
# phone= <socket.socket fd=3, family=AddressFamily.AF\_INET, type=SocketKind.SOCK\_STREAM, proto=0, laddr=('0.0.0.0', 0)>  
  
#2、绑定手机卡  
phone.bind((**'127.0.0.1'**,8080)) #0-65535:0-1024给操作系统使用  
  
#3、开机  
phone.listen(5) #listen 表示最大挂起的连接数  
  
#4、等电话链接  
print(**'starting...'**)  
conn,client\_addr = phone.accept()  
# **conn**: <socket.socket fd=4, family=AddressFamily.AF\_INET, type=SocketKind.SOCK\_STREAM, proto=0, laddr=('127.0.0.1', 8080), raddr=('127.0.0.1', 63377)>  
# **client\_addr**: ('127.0.0.1', 63377)  
  
#5、收，发消息  
data=conn.recv(1024) #1、单位：bytes 2、1024代表最大接收1024个bytes  
print(**'客户端的数据:'**,data)  
conn.send(**"服务端返回："**.encode(**"utf-8"**)+data.upper()) #发送给客户端  
  
#6、挂电话  
conn.close()  
  
#7、关机  
phone.close()

### 2.客户端

**import** socket  
  
#1、买手机  
phone=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  
# print(phone)  
  
#2、拨号  
phone.connect((**'127.0.0.1'**,8080))  
  
  
#3、发，收消息  
phone.send(**'hello'**.encode(**'utf-8'**))  
data=phone.recv(1024)  
print(data.decode(**'utf-8'**))  
  
#4、关闭  
phone.close()

## 6. 在简单套接字基础上加上通信循环

### 1.服务端

**import** socket  
  
phone=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  
phone.bind((**'127.0.0.1'**,8084)) *#0-65535:0-1024给操作系统使用*phone.listen(5)  
  
print(**'starting...'**)  
conn,client\_addr=phone.accept()  
print(client\_addr)  
  
**while True**: *#通信循环* data=conn.recv(1024)  
 print(**'客户端的数据'**,data.decode(**'utf-8'**))  
  
 conn.send(**"服务端返回："**.encode(**'utf-8'**)+data)  
  
conn.close()  
phone.close()

### 2.客户端

**import** socket  
  
phone=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  
phone.connect((**'127.0.0.1'**,8084))  
**while True**:  
 msg=input(**'>>: '**).strip()  
 phone.send(msg.encode(**'utf-8'**))  
 data=phone.recv(1024)  
 print(data.decode(**'utf-8'**))  
  
phone.close()

## 7．socket实现ssh

服务端：

'''  
\_\_title\_\_ = 'ex04.py'  
\_\_author\_\_ = 'ryan'  
\_\_mtime\_\_ = '2018/1/17'  
ssh ftp 服务端  
'''  
*import* os  
*import* socket  
*import* struct  
*import* json  
*import* subprocess  
  
*if* os.name == 'posix':  
 coding = 'utf-8'  
*elif* os.name == 'nt':  
 coding = 'gbk'  
  
ip\_port = ('127.0.0.1', 8099)  
server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
server.bind(ip\_port)  
server.listen(5)  
  
*while True*:  
 conn, client\_addr = server.accept()  
 *while True*:  
 *try*:  
 cmd = conn.recv(8096)  
 *if not* cmd: *continue* res = subprocess.Popen(cmd.decode(coding), shell=*True*, stdout=subprocess.PIPE, stderr=subprocess.PIPE)  
  
 stdout = res.stdout.read()  
 stderr = res.stderr.read()  
  
 res\_dic = {'total\_size': len(stdout) + len(stderr)}  
  
 res\_json = json.dumps(res\_dic)  
  
 res\_bytes = res\_json.encode(coding)  
  
 conn.send(struct.pack('i', len(res\_bytes)))  
  
 conn.send(res\_bytes)  
  
 conn.send(stdout)  
 conn.send(stderr)  
  
 *except* ConnectionResetError:  
 *break* conn.close()  
server.close()

客户端：

'''  
\_\_title\_\_ = ''  
\_\_author\_\_ = 'ryan'  
\_\_mtime\_\_ = '2018/1/16'  
ssh ftp 客户端  
'''  
  
*import* os  
*import* socket  
*import* struct  
*import* json  
  
*if* os.name == 'posix':  
 coding = 'utf-8'  
*elif* os.name == 'nt':  
 coding = 'gbk'  
  
ip\_port = ('127.0.0.1', 8099)  
client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
client.connect(ip\_port)  
  
*while True*:  
 cmd = input('>>:')  
 *if not* cmd: *continue* client.send(cmd.encode(coding))  
  
 recv\_res = client.recv(4)  
 header\_size = struct.unpack('i', recv\_res)[0]  
  
 header\_types = client.recv(header\_size)  
  
 header\_json = header\_types.decode(coding)  
  
 header\_dic = json.loads(header\_json)  
  
 total\_size = header\_dic['total\_size']  
  
 recv\_size = 0  
 recv\_data = b''  
 *while* recv\_size < total\_size:  
 res = client.recv(1024)  
 recv\_data += res  
 recv\_size += len(res)  
 print(recv\_data.decode(coding))  
client.close()

## 8．具体实现客户端和服务端的文件传输

查看自己的github

<https://github.com/ryan-yang-2049/oldboy_python_study/tree/master/third_module/practice/ftp>

包含了 命令执行，以及文件上传下载，断点续传，进度条等。

# 第四模块 多进程多线程

## 1.什么是进程

进程：正在进行的一个过程或者说是一个任务。而负责执行任务则是cpu。

## 2.开启进程的两种方式

### 2.1 multiprocessing 模块介绍

Python提供了multiprocessing。 multiprocessing模块用来开启子进程，并在子进程中执行我们定制的任务（比如函数），该模块与多线程模块threading的编程接口类似。multiprocessing模块的功能众多：支持子进程、通信和共享数据、执行不同形式的同步，>提供了Process、Queue、Pipe、Lock等组件。

需要再次强调的一点是：与线程不同，进程没有任何共享状态，进程修改的数据，改动仅限于该进程内。

### 2.2 开启子进程的两种方式

方式1：

*from* multiprocessing *import* Process  
*import* time  
*def* task(*name*):  
 print('%s is running'%*name*)  
 time.sleep(3)  
 print('%s is done'%*name*)  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 p=Process(target=task,args=('子进程1',))  
 # Process(target=task,kwargs={'name':'子进程2'})  
 p.start() # 仅仅只是给操作系统发送了一个信号。  
 print('主')

方式2：

*from* multiprocessing *import* Process  
*import* time  
*class* MyProcess(Process):  
 *def \_\_init\_\_*(self,*name*):  
 super().\_\_init\_\_()  
 self.name = *name  
 def* run(self):  
 print('%s is running'%self.name)  
 time.sleep(1)  
 print('%s is done'%self.name)  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 p=MyProcess('子进程1')  
 p.start()

### 2.3 练习题

1、思考开启进程的方式一和方式二各开启了几个进程？

都开启了2个进程，一个主进程，一个子进程

2、进程之间的内存空间是共享还是隔离的？

进程没有任何共享状态，进程修改的数据，改动仅限于该进程内。

## 3. 查看进程的pid

查看进程的pid 用os模块的 getpid(),查看父进程的pid 用 os.getppid()

*from* multiprocessing *import* Process  
*import* time,os  
*def* task(*name*):  
 print('%s is running，pid is %s'%(*name*,os.getpid()))  
 time.sleep(3)  
 print('%s is done,parent pid is ==> %s'%(*name*,os.getppid()))

*if* \_\_name\_\_ == '\_\_main\_\_':  
 p=Process(target=task,args=('子进程1',))  
 p.start() # 仅仅只是给操作系统发送了一个信号。  
 print('主',os.getpid(),os.getppid()) # 这个是主进程，当然 运行print 的主主进程是pycharm或者终端 ,windows 查看进程 tasklist |findstr pycharm

## 4．查看对象的其他属性和方法

### 4.1 Process对象的join方法

在主进程运行过程中如果想并发的执行其他任务，我们可以开启子进程，此时主进程的任务与子进程的任务分两种情况。

情况一: 在主进程的任务与子进程的任务彼此独立的情况下，主进程的任务先执行完成，主进程还需要等待子进程执行完毕，然后统一回收资源。

情况二：如果主进程的任务在执行到某一个阶段时，需要等待子进程执行完毕后才能继续执行，就需要有一种机制能够让主进程检测子进程是否运行完毕，在子进程执行完毕后才继续执行，否则一直阻塞，这就是join方法的作用。

### 4.2 进程并发执行

*from* multiprocessing *import* Process  
*import* time,os  
*def* task(*name*,*n*):  
 print('%s is running，pid is %s'%(*name*,os.getpid()))  
 time.sleep(*n*)  
 print('%s is done,parent pid is ==> %s'%(*name*,os.getppid()))

*if* \_\_name\_\_ == '\_\_main\_\_':  
 start = time.time()  
 p1=Process(target=task,args=('子进程1',2,))  
 p2=Process(target=task,args=('子进程2',5,))  
 p3=Process(target=task,args=('子进程3',3))  
 process\_list = [p1,p2,p3]  
 *for* i *in* process\_list:  
 i.start()  
  
 *for* pj *in* process\_list:  
 pj.join()  
  
 print('主',time.time()-start)

### 4.3 进程串行执行

*from* multiprocessing *import* Process  
*import* time,os  
*def* task(*name*,*n*):  
 print('%s is running，pid is %s'%(*name*,os.getpid()))  
 time.sleep(*n*)  
 print('%s is done,parent pid is ==> %s'%(*name*,os.getppid()))  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 start = time.time()  
 p1=Process(target=task,args=('子进程1',5,))  
 p2=Process(target=task,args=('子进程2',3,))  
 p3=Process(target=task,args=('子进程3',2))  
  
 p1.start()  
 p1.join()  
 p2.start()  
 p2.join()  
 p3.start()  
 p3.join()  
 print('主',time.time()-start)

串行并行总结：用join去控制进程执行的状态。

### 4.4 is\_alive 和 terminate

is\_alive：判断进程是否存活

terminate：杀死进程，发送信号给操作系统，操作系统需要一段时间才能够杀死进程

name：查看进程的名称，可以在创建对象的时候指定进程名称

*from* multiprocessing *import* Process  
*import* time,os  
*def* task(*name*):  
 print('%s is running，pid is %s'%(*name*,os.getpid()))  
 time.sleep(3)  
 print('%s is done,parent pid is ==> %s'%(*name*,os.getppid()))  
   
*if* \_\_name\_\_ == '\_\_main\_\_':  
 p=Process(target=task,name='myprocess-1',args=('子进程1',))  
 p.start()  
 p.terminate() # 杀死进程,发送信号给操作系统，操作系统需要一段时间才能够杀死进程。  
 time.sleep(3)  
 print(p.is\_alive())  
 print('主')  
 print(p.name)

## 5.守护进程

守护进程：主进程创建子进程，然后将该子进程设置成守护主进程的子进程。当主进程运行结束后，不管子进程是否进行完成，子进程都会结束。

守护进程需要强调两点：

其一：守护进程会在主进程代码执行结束后就终止

其二：守护进程内无法再开启子进程，否则抛出异常。

如果我们有两个任务需要执行，那么开一个主进程和一个子进程分别去执行就OK了，如果子进程的任务在主进程任务结束后就没有必要存在了，那么该子进程应该在开启前就设置成守护进程。主进程代码运行结束，守护进程随即终止。

### 5.1 验证主进程、子进程和守护进程之间的关系。

*from* multiprocessing *import* Process  
*import* time  
*def* task(*name*,*n*):  
 print('%s is running'%*name*)  
 time.sleep(*n*)  
 print('%s is done' % *name*)  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 p = Process(target=task,args=('子进程1',6))  
 p2 = Process(target=task,args=('子进程2',6))  
 p.daemon = *True* p2.daemon = *True* p.start()  
 p.join()  
 p2.start()  
 print("主")  
 time.sleep(5)  
 print("主over")

通过上面程序的验证：当子进程设置成为主进程的守护进程以后，不管子进程是否运行结束，子进程都会随着主进程的结束而结束。但是，当子进程有了join方法以后，就会先执行玩子进程，在运行主进程。

### 5.2 验证守护进程无法开启子进程

*from* multiprocessing *import* Process  
*import* time  
*def* task(*name*):  
 print('%s is running'%*name*)  
 time.sleep(2)  
 p=Process(target=time.sleep,args=(3,))  
 p.start()  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 p = Process(target=task,args=('子进程1',))  
 p.daemon = *True* p.start()  
 p.join()  
  
 print("主")

抛出异常：AssertionError: daemonic processes are not allowed to have children

### 5.3 练习题

*from* multiprocessing *import* Process  
*import* time  
  
*def* foo():  
 print(123)  
 time.sleep(10)  
 print("end123")  
  
*def* bar():  
 print(456)  
 # time.sleep(1)  
 print("end456")  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 p1=Process(target=foo)  
 p2=Process(target=bar)  
 p1.daemon=*True* p1.start()  
 p2.start()  
 p2.join()  
 print("main-------")

结果：

123

456

end456

main-------

一个主进程有两个子进程，其中一个位守护进程，并且第二个子进程使用了join方法，第二个子进程运行结束需要时间，因此，主进程结束会等待 第二个进程结束以后，才结束主进程，因此，守护进程就有了运行的时间。但是，当主进程结束以后，守护进程需要的时间更长，但他是守护进程，因此，直接结束。

## 6．互斥锁

### 6.1 互斥锁

进程之间数据不共享，但是共享同一套文件系统，所以访问同一个文件，或同一个终端，是没有问题的，而共享带来竞争，竞争带来的结果就是错乱。

互斥锁的优缺点和原理：

互斥锁的原理就是，并发修改同一块共享数据的操作，变成串行，数据安全。

优缺点：由并发变成了串行牺牲了运行效率，但是避免了竞争，提高了数据安全。

### 6.2 开启和不开启互斥锁比较

1）不开启互斥锁

*from* multiprocessing *import* Process,Lock  
*import* time  
*def* task(*name*,mutex):  
 print('%s run 1' %*name*)  
 time.sleep(1)  
 print('%s run 2' %*name*)  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 mutex = Lock()  
 *for* i *in* range(3):  
 p=Process(target=task,args=("进程 %s "%i,mutex))  
 p.start()

运行结果：

进程 0 run 1

进程 1 run 1

进程 2 run 1

进程 0 run 2

进程 1 run 2

进程 2 run 2

2） 开启互斥锁

*from* multiprocessing *import* Process,Lock  
*import* time  
*def* task(*name*,*mutex*):  
 *mutex*.acquire()  
 print('%s run 1' %*name*)  
 time.sleep(1)  
 print('%s run 2' %*name*)  
 *mutex*.release()  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 mutex = Lock()  
 *for* i *in* range(3):  
 p=Process(target=task,args=("进程 %s "%i,mutex))  
 p.start()

结果：

进程 1 run 1

进程 1 run 2

进程 0 run 1

进程 0 run 2

进程 2 run 1

进程 2 run 2

开启和不开启互斥锁可以看出：当开启的时候，只有某个子进程执行完成以后，才会执行第二个子进程。这样就从并发操作变成了串行操作。

### 6.3 模拟抢票软件

假设车票余额在一个文件里：

db.txt

{"count": 3}

模拟抢票.py

*from* multiprocessing *import* Process,Lock  
*import* time,json  
  
*def* search(*name*):  
 '''  
 查看剩余票的余额  
 '''  
 time.sleep(1)  
 dic = json.load(open('db.txt','r',encoding='utf-8'))  
 print('<%s> 查看到的剩余票数位: 【%s】'%(*name*,dic['count']))  
  
*def* get(*name*):  
 '''  
 开始进行抢票操作  
 '''  
 time.sleep(1)  
 dic = json.load(open('db.txt','r',encoding='utf-8'))  
  
 *if* dic['count'] >0:  
 dic['count'] -= 1  
 time.sleep(1)  
 json.dump(dic,open('db.txt','w',encoding='utf-8'))  
 print('<%s> 购票成功' %*name*)  
 *else*:  
 print('<%s> 购票失败' % *name*)  
  
*def* task(*name*,*mutex*):  
 search(*name*)  
 *mutex*.acquire()  
 get(*name*)  
 *mutex*.release()  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 mutex = Lock()  
 *for* i *in* range(5):  
 p=Process(target=task,args=('路人 %s'%i,mutex))  
 p.start()

运行结果：

<路人 0> 查看到的剩余票数位: 【3】

<路人 3> 查看到的剩余票数位: 【3】

<路人 4> 查看到的剩余票数位: 【3】

<路人 1> 查看到的剩余票数位: 【3】

<路人 2> 查看到的剩余票数位: 【3】

<路人 0> 购票成功

<路人 3> 购票成功

<路人 4> 购票成功

<路人 1> 购票失败

<路人 2> 购票失败

### 6.4 总结

加锁可以保证多个进程修改同一块数据时，同一时间只能有一个任务可以进行修改，即串行地修改，没错，速度是慢了，但牺牲了速度却保证了数据安全。

虽然可以用文件共享数据实现进程间的通信，但问题是：

1、效率低（共享数据基于文件，而文件是硬盘上的数据）

2、需要自己加锁处理

因此我们最好寻找一种解决方案能够兼顾：

1、效率高（多个进程共享一块内存数据

2、帮我们处理号锁问题。

这就是multiprocessing 模块为我们提供基于消息的IPC通信机制：队列和管道。

队列和管道都是将数据存放于内存中，而队列又是基于（管道+锁）实现的，可以让我们从复杂的锁问题中解脱出来，因而队列才是进程间通信的最佳选择。

我们应尽量避免使用共享数据，尽可能使用消息传递和队列，避免处理复杂的同步和锁问题，而且进程数目增多时，往往获得更好的可扩展性。

## 7 multiprocessing 中的队列使用

队列和栈的进出方式：

队列：先进先出 （像水管一样，先进去的最先流出去）

栈：先进后出 （像砌墙一样，先堆上去的就是最下面，取的时候，先砌的也是最后取）

*from* multiprocessing *import* Process,Queue  
q=Queue(3)  
# put 放入消息队列，get 获取消息队列里面的内容。  
#put ,get ,put\_nowait,get\_nowait,full,empty  
q.put(1)  
q.put(2)  
q.put(3)  
print(q.full()) #满了  
q.put(4) #再放就阻塞住了  
  
print(q.get())  
print(q.get())  
print(q.get())  
print(q.empty()) #空了  
# print(q.get()) #再取就阻塞住了

## 8 生产者和消费者模型

### 1.为什么要使用生产者消费者模型

生产者指的是生产数据的任务，消费者值得是处理数据的任务，在并发编程中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等着消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这个问题，于是引入了生产者和消费者模式。

### 2.什么是生产者和消费者模式

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

这个阻塞队列就是用来给生产者和消费者解耦的。

### 3.生产者消费者程序例子

*from* multiprocessing *import* Process,Queue  
*import* time  
  
*def* producer(*q*):  
 *for* i *in* range(5):  
 res = "包子%s"%i  
 time.sleep(0.5)  
 print("生产者生产了: %s"%res)  
 *q*.put(res)  
  
*def* consumer(*q*):  
 *while True*:  
 res = *q*.get()  
 *if* res *is None*:*break* time.sleep(1)  
 print("消费者吃了: %s"%res)  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 # 容器  
 q = Queue()  
 # 生产者们  
 p1 = Process(target=producer,args=(q,))  
 p2 = Process(target=producer,args=(q,))  
 p3 = Process(target=producer,args=(q,))  
  
 # 消费者们  
 c1 = Process(target=consumer,args=(q,))  
 c2 = Process(target=consumer,args=(q,))  
  
 p1.start()  
 p2.start()  
 p3.start()  
  
 c1.start()  
 c2.start()  
  
 p1.join()  
 p2.join()  
 p3.join()  
 q.put(*None*)  
 q.put(*None*)  
 print("主")

有多少个生产者不用管，只需要注意有多少个消费者。每一个消费者要对应一个结束信号。

### 4 生产者消费者模型总结

1、程序中有两类角色

一类负责生产数据（生产者）

一类负责处理数据（消费者）

2、引入生产者消费者模型为了解决的问题是

平衡生产者与消费者之间的速度差

程序解开耦合

3、让一个实现生产者消费者

生产者<------------>队列<------------->消费者

以后不用multiprocessing 这个实现队列，如果一定要使用这个，那就是所有的生产者消费者都在这个机器上，就是集中式的体现，会有很多安全性的问题。队列的软件：RabbitMQ

## 9 Joinablequeue

# JoinableQueue 实现生产者和消费者  
*from* multiprocessing *import* Process,JoinableQueue  
*import* time,random,os  
*def* consumer(*q*,*name*):  
 *while True*:  
 res=*q*.get()  
 time.sleep(random.randint(1,3))  
 print('\033[43m%s 吃 %s\033[0m' %(*name*,res))  
 *q*.task\_done() #发送信号给q.join()，说明已经从队列中取走一个数据并处理完毕了  
  
*def* producer(*q*,*name*,*food*):  
 *for* i *in* range(3):  
 time.sleep(random.randint(1,3))  
 res='%s%s' %(*food*,i)  
 *q*.put(res)  
 print('\033[45m%s 生产了 %s\033[0m' %(*name*,res))  
 *q*.join() #等到消费者把自己放入队列中的所有的数据都取走之后，生产者才结束  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 q=JoinableQueue() #使用JoinableQueue()  
  
 #生产者们:即厨师们  
 p1=Process(target=producer,args=(q,'egon1','包子'))  
 p2=Process(target=producer,args=(q,'egon2','骨头'))  
 p3=Process(target=producer,args=(q,'egon3','泔水'))  
  
 #消费者们:即吃货们  
 c1=Process(target=consumer,args=(q,'alex1'))  
 c2=Process(target=consumer,args=(q,'alex2'))  
 c1.daemon=*True* c2.daemon=*True* #开始  
 p1.start()  
 p2.start()  
 p3.start()  
 c1.start()  
 c2.start()  
  
 p1.join()  
 p2.join()  
 p3.join()

## 10 什么是线程

在传统操作系统中，每个进程有一个地址空间，而且默认就有一个控制线程

线程顾名思义，就是一条流水线工作的过程（流水线的工作需要电源，电源就相当于cpu），而一条流水线必须属于一个车间，一个车间的工作过程是一个进程，车间负责把资源整合到一起，是一个资源单位，而一个车间内至少有一条流水线。

**所以，进程只是用来把资源集中到一起（进程只是一个资源单位，或者说资源集合），而线程才是cpu上的执行单位。**

多线程（即多个控制线程）的概念是，在一个进程中存在多个线程，多个线程共享该进程的地址空间，相当于一个车间内有多条流水线，都共用一个车间的资源。

## 11 线程与进程的区别

1. 同一个进程内的多个线程共享该进程内的地址资源
2. 创建线程的开销要远小于创建进程的开销（创建一个进程，就是创建一个车间，涉及到申请空间，而且在该空间内建至少一条流水线，但创建线程，就只是在一个车间内造一条流水线，无需申请空间，所以创建开销小）

## 12 开启线程的两种方式

### 1 threading 模块介绍

multiprocess模块的完全模仿了threading模块的接口，二者在使用层面，有很大的相似性

### 2 开启线程的两种方式

方式一：

从资源的角度看，该程序运行的是一个主进程和一个线程

从程序的角度看，该程序运行的是一个主线程和一个子线程

*import* time  
*import* random  
*from* threading *import* Thread  
*def* piao(*name*):  
 print('%s piaoing' %*name*)  
 time.sleep(random.randrange(1,5))  
 print('%s piao end'%*name*)  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 t1 = Thread(target=piao,args=('egon',))  
 t1.start()  
 print("主")

方式二：

*from* threading *import* Thread  
*import* time  
  
*class* MyThread(Thread):  
 *def \_\_init\_\_*(self,*name*):  
 super().\_\_init\_\_()  
 self.name=*name  
 def* run(self):  
 time.sleep(2)  
 print('%s say hello' % self.name)  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 t = MyThread('egon')  
 t.start()  
 print('主线程')

## 13 多线程与多进程的区别

### 1 开进程的开销远大于开线程

*import* time  
*from* threading *import* Thread  
*from* multiprocessing *import* Process  
  
*def* piao(*name*):  
 print('%s piaoing' %*name*)  
 time.sleep(2)  
 print('%s piao end'%*name*)  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 t1\_start = time.time()  
 t1 = Thread(target=piao,args=('egon',))  
 t1.start()  
 print("开启线程一共运行了：",time.time()-t1\_start) #0.0005  
 p1\_start = time.time()  
 p1 = Process(target=piao,args=('egon',))  
 p1.start()  
 print("开启进程一共运行了：", time.time() - p1\_start) #0.033  
 print("主")

运行结果：

egon piaoing

开启线程一共运行了： 0.0005016326904296875

开启进程一共运行了： 0.039025306701660156

主

egon piaoing

egon piao end

egon piao end

可以看到开线程的时间明显小于开进程的时间，因为，开进程还要去申请内存地址空间等等。

### 2 同一进程内的多个线程共享该进程的地址空间

*import* time  
*from* threading *import* Thread  
*from* multiprocessing *import* Process  
  
n=100  
*def* task():  
 *global* n  
 n=0  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 t1 = Thread(target=task())  
 t1.start()  
  
 p1 = Process(target=task)  
 p1.start()  
 p1.join()  
  
 print("主",n)

当开启线程时，最后打印的n 的值为 0 。因为线程共享该进程的地址空间，当在函数内修改了n的值以后，也修改了全局的n的值。

当开启进程时，其实，有两个进程在同时运行，俗称主进程与子进程，因此，虽然他们初始的时候获取的n的值都是100，但是，主进程运行的时候，是获取自己的地址空间，因此 最后打印的还是 100.

### 3 所有子线程与主线程（主进程）是同一个PID号，子进程的PPID是主进程的PID

*import* time,os  
*from* threading *import* Thread  
*from* multiprocessing *import* Process  
  
*def* task(*name*):  
 print("%s PID号: %s,PPID号：%s"%(*name*,os.getpid(),os.getppid()))  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 p1 = Process(target=task,args=('子进程',))  
 p1.start()  
 p1.join()  
 t1 = Thread(target=task,args=('子线程1',))  
 t1.start()  
 t2 = Thread(target=task,args=('子线程2',))  
 t2.start()  
 print("主进程 PID号: %s,PPID号：%s"%(os.getpid(),os.getppid()))

结果：

子进程 PID号: 68240,PPID号：73452  
子线程1 PID号: 73452,PPID号：13160  
子线程2 PID号: 73452,PPID号：13160  
主进程 PID号: 73452,PPID号：13160

## 14 Thread对象的其他属性或方法

Thread实例对象的方法

# isAlive(): 返回线程是否活动的。

# getName(): 返回线程名。

# setName(): 设置线程名。

threading模块提供的一些方法：

# threading.currentThread(): 返回当前的线程变量。

# threading.enumerate(): 返回一个包含正在运行的线程的list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。

# threading.activeCount(): 返回正在运行的线程数量，与len(threading.enumerate())有相同的结果。

*from* threading *import* Thread,currentThread,active\_count,enumerate  
*import* time  
*def* task():  
 print('%s is running'%currentThread().getName())  
 time.sleep(2)  
 print('%s is running'%currentThread().getName())  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 t1 = Thread(target=task,name="Thread-test-1")  
 t1.start()  
 t2 = Thread(target=task,name="Thread-test-2")  
 t2.start()  
 t1.setName('子线程1') # 设置子线程名称  
 print("查看存活的线程总数", active\_count()) # 查看存活的线程总数  
 t1.join() # 让子线程先执行完毕后在执行后面的  
 currentThread().setName('Main Thread name') # 设置主线程名称  
 # 查看线程是否存活  
 print(t1.isAlive())  
 print("主线程",currentThread().getName()) # 主线程名称：MainThread  
 # enumerate 以列表形式获取当前活跃的线程对象。[<\_MainThread(MainThread, started 78460)>, <Thread(Thread-test-1, started 76708)>]  
 print(enumerate())

结果：

Thread-test-1 is running

Thread-test-2 is running

查看存活的线程总数 3

Thread-test-2 is running

子线程1 is running

False

主线程 Main Thread name

[<\_MainThread(Main Thread name, started 138132)>]

## 15 守护线程

无论是进程还是线程，都遵循：守护xxx会等待主xxx运行完毕以后被销毁。

需要强调的是：运行完毕并非终止运行。

1、对主进程来说，运行完毕指的是主进程代码运行完毕

2、对主线程来说，运行完毕指的是主线程所在的进程内所有非守护线程统统运行完毕，主线程才算运行完毕

详细解释：

1、主进程在其代码结束后就已经算运行完毕了（守护进程在此时就被回收）,然后主进程会一直等非守护的子进程都运行完毕后回收子进程的资源(否则会产生僵尸进程)，才会结束，

2、主线程在其他非守护线程运行完毕后才算运行完毕（守护线程在此时就被回收）。因为主线程的结束意味着进程的结束，进程整体的资源都将被回收，而进程必须保证非守护线程都运行完毕后才能结束。

解释的第二句话：主线程的结束是在当前主线程和非守护线程结束以后才会结束，守护线程会在非守护线程结束以后才会被回收。

### 1 守护线程的验证一

*from* threading *import* Thread  
*import* time  
*def* sayhi(*name*):  
 time.sleep(2)  
 print('%s say hello' %*name*)  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 t1=Thread(target=sayhi,args=('egon',))  
 t1.setDaemon(*True*) #必须在t.start()之前设置 等价于 t.daemon=True  
 print('主线程')  
 print("t1 是否存活：%s "%(t1.is\_alive()))

运行结果：

主线程

t1 是否存活：False

### 2 守护线程的验证二

主线程就代表了主进程的生命周期

*from* threading *import* Thread  
*import* time  
*def* sayhi(*name*):  
 time.sleep(2)  
 print('%s say hello' %*name*)  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 t1=Thread(target=sayhi,args=('egon',))  
 t1.setDaemon(*True*) #必须在t.start()之前设置 等价于 t.daemon=True  
  
 t2=Thread(target=sayhi,args=('alex',))  
 t1.start()  
 t2.start()  
  
 print('主线程')  
 print("t1 是否存活：%s , t2 是否存活：%s"%(t1.is\_alive(),t2.is\_alive()))

运行结果：

主线程

t1 是否存活：True , t2 是否存活：True

alex say hello

egon say hello

## 16 线程的互斥锁

线程的互斥锁和进程的互斥锁一样，都是保证数据安全，但是牺牲了运行效率。

*from* threading *import* Thread,Lock  
*import* time  
n = 100  
  
*def* task():  
 *global* n  
 mutex.acquire()  
 temp = n  
 time.sleep(0.1)  
 n = temp - 1  
 mutex.release()  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 mutex = Lock()  
 t\_l = []  
 *for* i *in* range(90):  
 t = Thread(target=task)  
 t\_l.append(t)  
 t.start()  
 print(t\_l)  
 *for* t *in* t\_l: # 因主线程和子线程都是并发执行的，因此要先执行完子线程以后，主线程才能获取的子线程处理以后的值。  
 t.join()  
  
 print("主",n)

## 17 GIL全局解释器锁

GIL本质就是一把互斥锁，既然是互斥锁，所有互斥锁的本质都一样，都是将并发运行变成串行，以此来控制同一时间内共享数据只能被一个任务所修改，进而保证数据安全。

可以肯定的一点是：保护不同的数据的安全，就应该加不同的锁。

只有CPYTHON解释器才有GIL

**结论：在Cpython解释器中，同一个进程下开启的多线程，同一时刻只能有一个线程执行，无法利用多核优势**

### 1 GIL与Lock

首先，我们需要达成共识：锁的目的是为了保护共享的数据，同一时间只能有一个线程来修改共享的数据

然后，我们可以得出结论：保护不同的数据就应该加不同的锁。

最后，问题就很明朗了，GIL 与Lock是两把锁，保护的数据不一样，前者是解释器级别的（当然保护的就是解释器级别的数据，比如垃圾回收的数据），后者是保护用户自己开发的应用程序的数据，很明显GIL不负责这件事，只能用户自定义加锁处理，即Lock

示例：

from threading import Thread,Lock

import os,time

def work():

global n

lock.acquire()

temp=n

time.sleep(0.1)

n=temp-1

lock.release()

if \_\_name\_\_ == '\_\_main\_\_':

lock=Lock()

n=100

l=[]

for i in range(100):

p=Thread(target=work)

l.append(p)

p.start()

for p in l:

p.join()

print(n) #结果肯定为0，由原来的并发执行变成串行，牺牲了执行效率保证了数据安全，不加锁则结果可能为99

### 2.计算密集型

如果并发的多个任务是计算密集型：多进程效率高

*from* multiprocessing *import* Process  
*from* threading *import* Thread  
*import* os,time  
*def* work():  
 res=0  
 *for* i *in* range(100000000):  
 res\*=i  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 l=[]  
 print(os.cpu\_count()) #本机为4核  
 start=time.time()  
 *for* i *in* range(4):  
 # p=Process(target=work) #耗时21s多  
 p=Thread(target=work) #耗时62多  
 l.append(p)  
 p.start()  
 *for* p *in* l:  
 p.join()  
 stop=time.time()  
 print('run time is %s' %(stop-start))

### 3 IO密集型

如果并发的多任务是I/O密集型：多线程效率高

*from* multiprocessing *import* Process  
*from* threading *import* Thread  
*import* threading  
*import* os,time  
*def* work():  
 time.sleep(2)  
 print('===>')  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 l=[]  
 print(os.cpu\_count()) #本机为4核  
 start=time.time()  
 *for* i *in* range(400):  
 p=Process(target=work) #耗时52s多,大部分时间耗费在创建进程上  
 # p=Thread(target=work) #耗时2s多  
 l.append(p)  
 p.start()  
 *for* p *in* l:  
 p.join()  
 stop=time.time()  
 print('run time is %s' %(stop-start))

**应用：**

多线程用于IO密集型，如socket，爬虫，web

多进程用于计算密集型，如金融分析

### 4 GIL与互斥锁的自我理解

如果要保证程序的数据安全，一定要加锁，GIL是再有锁的情况下才能够体现数据安全的一个现象。GIL 是全局的解释器锁，如果要保证自己程序的数据安全，那就要对自己的数据处理的地方，加一个锁。通俗的讲：GIL和Lock 是两把锁，一个锁线程，一个锁数据，锁线程是让一个线程去执行，锁数据是让数据安全，锁的目的是为了保护共享的数据，同一时间只能有一个线程来修改共享的数据，然后，我们可以得出结论：保护不同的数据就应该加不同的锁。

进程是资源单位，线程是执行单位。主线程的运行周期就是进程的运行周期，一个程序只是多线程的情况下，一个主线程就代表了 执行该程序的进程。

## 18 死锁

所谓死锁：是只两个或两个以上的进程或线程在执行过程中，因争夺资源造成的一种互相等待的现象。若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程成为死锁。

### 1 程序死锁的例子

造成下面程序的原因是，两个线程在对方身上都有本线程需要的锁，因此，造成了死锁。

*from* threading *import* Thread,Lock  
*import* time  
mutexA=Lock()  
mutexB=Lock()  
  
*class* MyThread(Thread):  
 *def* run(self):  
 self.func1()  
 self.func2()  
 *def* func1(self):  
 mutexA.acquire()  
 print('\033[41m%s 拿到A锁\033[0m' %self.name)  
  
 mutexB.acquire()  
 print('\033[42m%s 拿到B锁\033[0m' %self.name)  
 mutexB.release()  
  
 mutexA.release()  
  
 *def* func2(self):  
 mutexB.acquire()  
 print('\033[43m%s 拿到B锁\033[0m' %self.name)  
 time.sleep(2)  
  
 mutexA.acquire()  
 print('\033[44m%s 拿到A锁\033[0m' %self.name)  
 mutexA.release()  
  
 mutexB.release()  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 *for* i *in* range(10):  
 t=MyThread()  
 t.start()

### 2 递归锁

解决上面死锁的办法就是：递归锁。在Python中为了支持在同一线程中多次请求同一资源，python提供了可重入锁RLock。这个RLock内部维护着一个Lock和一个counter变量，counter记录了acquire的次数，从而使得资源可以被多次require。直到一个线程所有的acquire都被release，其他的线程才能获得资源。上面的例子如果使用RLock代替Lock，则不会发生死锁，二者的区别是：递归锁可以连续acquire多次，而互斥锁只能acquire一次

通俗的讲：可以连续acquire多次，每次acquire一次，计数器就 +1 ，如果继续acquire 计数器继续+1，当遇到release的时候，计数器就 -1 ，这期间所有其他线程只能等待，等待该线程释放所有的锁，也就是计数器的值为0，

*from* threading *import* Thread,RLock  
*import* time  
  
mutexA=mutexB=RLock() # 创建递归锁对象  
  
*class* MyThread(Thread):  
 *def* run(self):  
 self.func1()  
 self.func2()  
 *def* func1(self):  
 mutexA.acquire()  
 print('\033[41m%s 拿到A锁\033[0m' %self.name)  
  
 mutexB.acquire()  
 print('\033[42m%s 拿到B锁\033[0m' %self.name)  
 mutexB.release()  
  
 mutexA.release()  
  
 *def* func2(self):  
 mutexB.acquire()  
 print('\033[43m%s 拿到B锁\033[0m' %self.name)  
 time.sleep(2)  
  
 mutexA.acquire()  
 print('\033[44m%s 拿到A锁\033[0m' %self.name)  
 mutexA.release()  
  
 mutexB.release()  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 *for* i *in* range(10):  
 t=MyThread()  
 t.start()

## 19 信号量

### 1 信号量介绍

信号量也是一把锁，可以指定信号量为5，对比互斥锁同一时间只能有一个任务抢到锁去执行，信号量同一时间可以有5个任务拿到锁去执行，如果说互斥锁是合租房屋的人去抢一个厕所，那么信号量就相当于一群路人争抢公共厕所，公共厕所有多个坑位，这意味着同一时间可以有多个人上公共厕所，但公共厕所容纳的人数是一定的，这便是信号量的大小

*from* threading *import* Thread,Semaphore,currentThread  
*import* time,random  
  
sm=Semaphore(3)  
  
*def* task():  
 # sm.acquire()  
 # print("%s running"%currentThread().getName())  
 # time.sleep(random.randint(1, 3))  
 # sm.release()  
 # 上面四句等价于下面的三句  
 *with* sm:  
 print("%s running" % currentThread().getName())  
 time.sleep(random.randint(1,3))  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 *for* i *in* range(10):  
 t=Thread(target=task)  
 t.start()

### 2 查看信号量是否数据安全

*from* threading *import* Thread,Semaphore,currentThread  
*import* time,random  
  
sm=Semaphore(2)  
n=10  
*def* task():  
 # sm.acquire()  
 # print("%s running"%currentThread().getName())  
 # time.sleep(random.randint(1, 3))  
 # sm.release()  
 #等价于下面  
 *global* n  
 *with* sm:  
 print("%s running" % currentThread().getName())  
 n -= 10  
 print("inner",n)  
 time.sleep(2)  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 t\_l = []  
 *for* i *in* range(3):  
 t=Thread(target=task)  
 t.start()  
 t\_l.append(t)  
  
 *for* t *in* t\_l:  
 t.join()  
 print('n',n)

结果：

Thread-1 running

inner 0

Thread-2 running

inner -10

Thread-3 running

inner -20

n -20

从结果可以看出，信号量的数据是安全的。因为在多线程中有一个全局的GIL解释器锁，会保证在某一个时刻只能有一个线程去执行，只要不死锁，那就线程安全。

## 20 Event 事件

线程的一个关键特性是每个线程都是独立运行且状态不可预测。如果程序中的其 他线程需要通过判断某个线程的状态来确定自己下一步的操作,这时线程同步问题就会变得非常棘手。为了解决这些问题,我们需要使用threading库中的Event对象。对象包含一个可由线程设置的信号标志，它允许线程等待某些事件的发生。在初始情况下，Event对象中的信号标志被设置为假，如果有线程等待一个Event对象，而这个Event对象的标志为假，那么这个线程将会被一直阻塞直至该设置为真。一个线程如果将一个Event对象的信号标志设置为真，它将唤醒所有等待这个Event对象的线程。如果一个线程等待一个已经被设置为真的Event对象，那么它将忽略这个时间，继续执行。

from threading import Event

event.isSet()：返回event的状态值；

event.wait()：如果 event.isSet()==False将阻塞线程；

event.set()： 设置event的状态值为True，所有阻塞池的线程激活进入就绪状态， 等待操作系统调度；

event.clear()：恢复event的状态值为False。

### 示例1

*from* threading *import* Thread,Event  
*import* time  
event = Event()  
event2 = Event()  
# event.wait() 开始等待  
# event.set() 等待结束  
  
*def* student(*name*):  
 print("学生 %s 正在听课"%*name*)  
 event.wait() #可以设置超时时间 event.wait(2)  
 print("学生 %s 课间活动"%*name*)  
 event2.wait()  
 print("放学回家")  
  
*def* teacher(*name*):  
 print("老师 %s 正在上课"%*name*)  
 time.sleep(6)  
 event.set()  
  
*def* go\_home():  
 event.wait()  
 print("还有10分钟放学")  
 time.sleep(10)  
 event2.set()  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 stu1 = Thread(target=student,args=('alex',))  
 stu2 = Thread(target=student,args=('wpq',))  
 stu3 = Thread(target=student,args=('ryan',))  
 t1 = Thread(target=teacher,args=('egon',))  
 g1 = Thread(target=go\_home)  
  
 stu1.start()  
 stu2.start()  
 stu3.start()  
 t1.start()  
 g1.start()

运行结果：

学生 alex 正在听课

学生 wpq 正在听课

学生 ryan 正在听课

老师 egon 正在上课

学生 alex 课间活动

学生 wpq 课间活动

学生 ryan 课间活动

还有10分钟放学

放学回家

放学回家

放学回家

### 示例2：模拟socket远程链接

check是否再线，然后，再判断是否链接。

*from* threading *import* Thread,Event,currentThread  
*import* time  
event = Event()  
  
*def* conn():  
 n=0  
 *while not* event.is\_set():  
 *if* n == 3:  
 print('%s try many times'%currentThread().getName())  
 *return* n+=1  
 print("%s try %s"%(currentThread().getName(),n))  
 event.wait(0.5)  
 print("%s in connected"%currentThread().getName())  
  
  
*def* check():  
 print("%s is checking"%currentThread().getName())  
 time.sleep(5)  
 event.set()  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 t = Thread(target=check, name='check-connect-status')  
 t.start()  
 *for* i *in* range(2):  
 t = Thread(target=conn,name='conn-%s'%i)  
 t.start()

结果：

check-connect-status is checking

conn-0 try 1

conn-1 try 1

conn-0 try 2

conn-1 try 2

conn-0 try 3

conn-1 try 3

conn-1 try many times

conn-0 try many times

### 示例3：链接mysql

*from* threading *import* Thread,Event  
*import* threading  
*import* time,random  
*def* conn\_mysql():  
 count=1  
 *while not* event.is\_set():  
 *if* count > 3:  
 # raise TimeoutError('链接超时')  
 *return* print('<%s>第%s次尝试链接' % (threading.current\_thread().getName(), count))  
 event.wait(0.5)  
 count+=1  
 print('<%s>链接成功' %threading.current\_thread().getName())  
  
  
*def* check\_mysql():  
 print('\033[45m[%s]正在检查mysql\033[0m' % threading.current\_thread().getName())  
 # time.sleep(random.randint(2,4))  
 time.sleep(1)  
 event.set()  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 event=Event()  
 conn1=Thread(target=conn\_mysql)  
 conn2=Thread(target=conn\_mysql)  
 check=Thread(target=check\_mysql)  
  
 conn1.start()  
 conn2.start()  
 check.start()

结果：

<Thread-1>第1次尝试链接

<Thread-2>第1次尝试链接

[Thread-3]正在检查mysql

<Thread-1>第2次尝试链接

<Thread-2>第2次尝试链接

<Thread-1>第3次尝试链接

<Thread-2>第3次尝试链接

<Thread-2>链接成功

<Thread-1>链接成功

从示例2和示例3可以看出，两个线程是同时进行的，当wait等待的时间的总秒数大于检查返回结果的时间，就可以成功链接（假设可以链接的情况下）

## 21 定时器

定时器，指定n秒以后执行某操作

*from* threading *import* Timer  
  
*def* task(*name*):  
 print("hello %s"%*name*)  
  
t=Timer(5,task,args=('ryan',))  
t.start()

Timer(秒数，程序名称，参数)

### 1 示例：验证码

*from* threading *import* Timer  
*import* random  
  
*class* Code:  
 *def \_\_init\_\_*(self):  
 self.make\_cache()  
  
 *def* make\_cache(self,*interval*=5):  
 self.cache = self.make\_code()  
 print(self.cache)  
 self.t=Timer(*interval*,self.make\_cache)  
 self.t.start()  
  
 *def* make\_code(self,*n*=4):  
 res = ''  
 *for* i *in* range(*n*):  
 s1=str(random.randint(0,9))  
 s2=chr(random.randint(65,90))  
 res+=random.choice([s1,s2])  
 *return* res  
  
 *def* check(self):  
 *while True*:  
 code = input("请输入你的验证码>>:").strip()  
 *if* code.upper() == self.cache:  
 print("验证码输入正确")  
 self.t.cancel()  
 *break*obj=Code()  
obj.check()

## 22 线程queue

### 1 线程之--->队列

队列 --->先进先出

*import* queue  
q=queue.Queue(3)  
# 队列存数据，因为上面设置的队列的大小是3，因此，只能存放3次数据,如果存的数据超过了设置的大小，则程序会卡在原地  
q.put('first')  
q.put(2)  
q.put('third')  
'''  
放入数据的次数超过了存设置的大小会抛出异常。  
q.put(4)  
q.put(4,block=False) # 默认block的值是True，如果设置为False以后，运行程序会抛出异常 "queue.Full"  
q.put(4,block=True,timeout=3) # 如果 timeout=3 3秒以内没有去做，也抛出异常 "queue.Full"  
q.put\_nowait(4) # 也抛出异常 "queue.Full"  
'''  
  
# 队列取数据，取的时候也一样，如果取得值超过了队列存的值，程序也会卡在原地  
print("取数据1",q.get())  
print("取数据2",q.get())  
print("取数据3",q.get())  
'''  
取数据的次数超过了放数据的次数，也会抛出异常  
print("取数据4",q.get(block=False,timeout=3)) # 和存数据一样的理解：抛出异常 "queue.Empty"  
print(q.get\_nowait()) # 当队列里面为空时，抛出异常 "queue.Empty"  
'''

结果：

取数据1 first

取数据2 2

取数据3 third

### 2 线程之--->堆栈

堆栈 ---> 后进先出

*import* queue  
q = queue.LifoQueue(3) # 后进先出 --> 堆栈  
q.put('first')  
q.put(2)  
q.put('third')  
  
print("取数据1",q.get())  
print("取数据2",q.get())  
print("取数据3",q.get())

结果：

取数据1 third

取数据2 2

取数据3 first

### 3 线程之--->优先级队列

*import* queue  
q = queue.PriorityQueue(3) # 优先级队列  
q.put((10,'one')) # q.put((优先级，数据)) 数字越小，优先级越高  
q.put((40,'two'))  
q.put((30,'three'))  
  
print("取数据1",q.get())  
print("取数据2",q.get())  
print("取数据3",q.get())

结果：

取数据1 (10, 'one')

取数据2 (30, 'three')

取数据3 (40, 'two')

## 23 多线程实现并发的套接字通信

### 1 服务端

*import* socket  
*from* threading *import* Thread  
  
*def* communicate(*conn*):  
 *while True*:  
 *try*:  
 data = *conn*.recv(104)  
 *if not* data:*break  
 conn*.send(data.upper())  
 *except* ConnectionResetError:  
 *break  
 conn*.close()  
  
*def* server(*ip\_port*):  
 server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 server.bind(*ip\_port*)  
 server.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR,1)  
 server.listen(5)  
 *while True*:  
 conn,client\_addr = server.accept()  
 t=Thread(target=communicate,args=(conn,))  
 t.start()  
 server.close()  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 ip\_port = ('127.0.0.1',8081)  
 server(ip\_port)

### 2 客户端

*import* socket  
client = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  
client.connect(('127.0.0.1',8081))  
  
*while True*:  
 msg = input(">>:").strip()  
 *if not* msg:*continue* client.send(msg.encode('utf-8'))  
 data = client.recv(1024)  
 print(data.decode('utf-8'))

## 24 进程池和线程池

### 1 进程池与线程池

在刚开始学多进程或多线程时，我们迫不及待地基于多进程或多线程实现并发的套接字通信，然而这种实现方式的致命缺陷是：服务的开启的进程数或线程数都会随着并发的客户端数目地增多而增多，这会对服务端主机带来巨大的压力，甚至于不堪重负而瘫痪，于是我们必须对服务端开启的进程数或线程数加以控制，让机器在一个自己可以承受的范围内运行，这就是进程池或线程池的用途，例如进程池，就是用来存放进程的池子，本质还是基于多进程，只不过是对开启进程的数目加上了限制

### 2 介绍

官网：https://docs.python.org/dev/library/concurrent.futures.html

concurrent.futures模块提供了高度封装的异步调用接口

ThreadPoolExecutor：线程池，提供异步调用

ProcessPoolExecutor: 进程池，提供异步调用

Both implement the same interface, which is defined by the abstract Executor class.

### 3 基本方法

1、submit(fn, \*args, \*\*kwargs) 异步提交任务

2、map(func, \*iterables, timeout=None, chunksize=1) 取代for循环submit的操作

3、shutdown(wait=True)

相当于进程池的pool.close()+pool.join()操作

wait=True，等待池内所有任务执行完毕回收完资源后才继续

wait=False，立即返回，并不会等待池内的任务执行完毕

但不管wait参数为何值，整个程序都会等到所有任务执行完毕

submit和map必须在shutdown之前

4、result(timeout=None) 取得结果

5、add\_done\_callback(fn) 回调函数

### 4 简单总结

进程池：ProcessPoolExecutor，线程池：ThreadPoolExecutor 他们的接口一模一样，学会一个另一个自然会用，唯一需要知道是，什么时候用线程池，什么时候用进程池。

进程池和线程池的本质还是开进程或者开线程

进程：计算密，想用多核处理，用进程

线程：I/O密集型，用多线程，例如：socket

线程池/进程池：池对数目加以限制，保证机器以一个可承受的范围，以一个健康的状态保证程序的进行。

### 5 进程池

*from* concurrent.futures *import* ProcessPoolExecutor  
*import* os,random,time  
  
*def* task(*arg*):  
 print('name: %s,PID: %s run'%(*arg*,os.getpid()))  
 time.sleep(random.randint(1,3))  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 # 进程池  
 pool = ProcessPoolExecutor(4) # 如果不指认进程池的个数，默认使用CPU的个数。  
 *for* i *in* range(10):  
 pool.submit(task,'egon%s'%i) # 提交任务的方式叫异步调用。提交完任务，不用再等着任务提交执行结果的返回。  
 pool.shutdown() # 等待所有进程池里面的进程结束以后再去执行主进程  
 print("主")

简单总结：再同一时刻，该程序只有4个进程在使用。使用了shutdown以后，只有当进程池里面的进程完毕以后，才去执行进程池后面的主进程的程序

### 6 线程池

*from* concurrent.futures *import* ThreadPoolExecutor  
*import* os,random,time  
*from* threading *import* currentThread  
  
*def* task():  
 print('name: %s,PID: %s run'%(currentThread().getName(),os.getpid()))  
 time.sleep(random.randint(1,3))  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 pool = ThreadPoolExecutor(5) # 如果不指认进程池的个数，默认使用CPU的个数。  
 *for* i *in* range(10):  
 pool.submit(task) # 提交任务的方式叫异步调用。提交完任务，不用再等着任务提交执行结果的返回。  
 pool.shutdown(wait=*True*) # 等待所有线程池里面的进程结束以后再去执行主进程, 默认值为True  
 print("主")

总结：同一时刻只有5个线程在执行。

### 7 线程池回调函数

## 25 协程

### 1 协程概念

### 2 gevent 的使用

### 3 asyncio 的使用

### 4 应用场景

## 26 IO模型

### 1 IO模型介绍

### 2 阻塞IO

### 3 非阻塞IO

### 4 多路复用IO

### 5 异步IO

# 第四模块 第八章 MySQL相关