

Python3 使用指南

作者：向仔州

if ..else..格式.....	5
if ..else..格式.....	5
强制类型转换.....	6
赋值运算符.....	6
赋值运算符.....	7
While 循环.....	8
break 跳出循环使用.....	10
continue 在循环里不执行 continue 字符后面的语句，跳到循环开始位置重新执行循环.....	10
字符串拼接.....	11
input 函数输入字符.....	11
数据类型转换.....	12
print 函数格式化输出.....	12
转义字符让字符输出格式更工整.....	13
函数定义与使用.....	13
函数定义与使用.....	15
列表 index 函数使用.....	16
修改列表里面的某个元素.....	16
append 使用，给列表增加数据.....	16
extend 使用，两个列表相互插入数据，也就是一个列表数据插入给另一个列表.....	17
remove 使用，删除列表里面某个元素.....	17
pop 使用，默认将列表最后一个元素删除.....	17
del 使用，del 也可以删除列表里面的某个元素.....	18
len 函数统计列表长度.....	18
count 函数，查看列表里面同名的元素出现几次.....	18
sort 函数升序，降序使用.....	18
reverse 函数使列表元素相邻之间进行调换，然后最后的元素放在列表最前面，这叫逆序.....	19
Python 元组用法.....	19
Python 字典用法.....	22
Python 字符串类型.....	25
字符串文本对齐.....	28
字符串拆分和拼接.....	30
字符串切片.....	31
计算字符串里面最大数值.....	32
计算字符串里面最小数值.....	32

计算列表最大值最小值.....	33
计算字典最大值最小值.....	33
列表和元组也可以像字符串那样切片.....	33
增加列表，元组数据长度.....	34
除了合并字符串，字符串还可以两两拼接，列表元组也可以.....	34
in , not in 运算符.....	35
Python 除了 if...else...以外，还增加了 for...else.....	36
Python 实现名片管理系统案例.....	37
Python 实现名片管理系统案例.....	44
函数形参也是变量的引用.....	45
函数形参也是变量的引用.....	46
函数返回值返回多个参数的方法.....	47
全局变量赋值给函数形参，会不会改变函数内部形参的值?.....	50
列表变量之间使用+=符号要注意.....	51
函数形参不赋值的情况下使用(所谓的缺省值)，语句 if not ...使用.....	51
多值参数形参传递，形参变量前 加* 或者 加**	54
Python 的单继承.....	55
子类修改父类的函数叫做方法(函数)重写.....	56
子类重写了父类同名的函数，导致子类对象无法再使用父类同名函数的内容，但是我又想子类使用重写函数的同时也调用父类同名函数的内容，那么只有用 super()来解决了.....	57
Python 多继承.....	59
Python 多态使用.....	60
类属性使用，就是让类里面指定变量永久存放在内存中，用来做记时等应用.....	62
Python 除了类属性，还有类函数，就是类自己调用的函数.....	62
类静态函数(也就是静态方法).....	63
设计模式中的单例模式.....	63
Python 异常.....	65
异常处理处理 try 和 except 关键字，还有 else 和 finally.....	68
python 异常抛出与传递.....	69
主动抛出异常机制 raise 实际应用.....	70
import 关键字使用，就是导入其它 py 文件.....	71
如果 import 导入的 py 文件名太长，就用 as 来给导入的 py 文件取别名替代.....	72
from 文件名 import 内容名.... 这种使用方法.....	73
from...import * 也是导入模块 py 文件的所有内容.....	73
import 导入模块名和系统模块名冲突问题.....	74
import 导入其它模块时，可能会自动执行其它模块的测试代码，如何解决？.....	75
Python 的子模块(py 文件)只是类似于 C 语言的 C 文件，但是 python 还有一种文件叫做包.....	76
Python 读写文件操作.....	78
readline 每次执行读取文件一行内容，而不会把整个文件读完.....	79
readline 每次执行读取文件一行内容，而不会把整个文件读完.....	80
大文件复制粘贴用 readline.....	81

python 读写 csv 文件.....	82
首先了解 python 迭代器.....	82
列表创建的 4 种方法.....	83
Iter(x)内置函数，将列表，字典，字符串这些迭代对象转换成迭代器.....	85
CSV 读取文件实验.....	86
CSV 写入文件实验.....	88
with....as....语句用法.....	89
with....as....语句用法.....	89
列表，字典，集合中取出想要的数据.....	90
lambda 表达式使用.....	90
filter 过滤器使用.....	92
filter 过滤器使用.....	93
集合解析.....	94
使用 enum 增加程序的可读性.....	94
命名元组.....	95
字典无法排序，所以只有将其转换成列表或者元组来排序.....	95
zip 函数使用方法.....	96
Python 寻找列表中出现次数最多的数.....	96
collections 模块的 Counter 类使用.....	96
查询多个字典公共键.....	97
map 函数使用.....	97
map 函数使用.....	98
字典判断 in 和 not in 关键字操作方法.....	99
下面进行公共键值提取案例分析.....	99
dict.keys() 字典 keys 函数使用.....	100
让字典保持有序排列.....	101
collections - OrderedDict 有序字典模块使用.....	101
enumerate 函数使用.....	101
islice 切片函数使用.....	102
记录用户历史数据.....	102
deque 双端队列函数使用.....	102
将字符串里不同的分隔符取消掉.....	103
sum 函数列表相加使用.....	104
re.split 正则表达式使用，字符串分隔多个符号.....	105
查看文件是否存在.....	105
endswith 函数用法.....	105
修改已有字符串的格式.....	106

正则表达式 re.sub 函数使用.....	106
UDP 网络传输数据包拼接处理.....	108
str.join 函数用法.....	109
字符串左右居中对齐.....	110
ljust 左对齐 , rjust 右对齐 , center 居中对齐函数使用.....	110
format 函数使用 , 左对齐 , 右对齐 , 居中.....	111
可迭代对象和迭代器的区别.....	112
迭代器是从可迭代对象里面去获取的.....	113
下面就来提取天气信息 , 用迭代器方式展示.....	113
如何读取二进制文件 , wav 文件案例.....	117
struct.unpack 函数使用.....	117
array 函数使用.....	118
file.tell 总文件大小计算.....	119
array.array.fromfile 函数使用.....	119
临时文件使用.....	120
TemporaryFile 对象使用.....	120
NamedTemporaryFile.....	121
如何读写 json 格式的数据.....	122
Json.load , json.dump 和 json.loads , json.dumps 的区别就是 load , dump 是用来解析文件的....	123
如何读写 excel 文件.....	124

if ..else..格式

```
value = 80  
  
arg = int(input("请输入你的年龄"))  
if arg == value:  
    print("输入正确")  
else:  
    print("输入错误")
```

Input 接受的键盘输入得到的是字符串，这里必须把字符串转换成 int 才能计算所以 int(...)就是强制转换括号里面的内容为 int 类型

```
请输入你的年龄 50  
输入错误  
>>>  
===== RES  
= 请输入你的年龄 80  
输入正确  
>>>
```

If 判断后用冒号 : 结束，就类似 C 语言的 ; 分号执行程序用 tab 包裹，类似 C 语言的 {} 大括号

```
value = 80  
  
arg = int(input("请输入你的年龄"))  
if arg == value:  
    print("输入正确")  
elif arg>80:  
    print("你输入的年龄大了")  
else:  
    print("输入错误")
```

elif 类似 C 语言的 else if

```
= 请输入你的年龄 100  
你输入的年龄大了  
>>>
```

```
value = 80  
  
arg = int(input("请输入你的年龄"))  
if arg == value:  
    print("输入正确")  
elif arg>80:  
    print("你输入的年龄大了")  
elif arg<10:  
    print("你输入的年龄小了")  
else:  
    print("输入错误")
```

elif 类似 C 语言的 else if
可以多次判断
else if
else if

```
请输入你的年龄 5  
你输入的年龄小了  
>>>
```

+ - × ÷ 算术运算符

```
>>> 2**3  
8  
>>> 2*3  
6  
>>> 5/2  
2.5  
>>> 5//2  
2  
>>> 9%5  
4  
>>> 1+2  
3  
>>> 3-2  
1
```

指数

乘法

除法，结果会有小数点

整除

求余

减法

加法

优先级：指数 > 乘除求余整除 > 加减

```
>>> (2+3)*2  
10  
搞不清楚优先级就加()括号
```

强制类型转换

```
>>> a=input("请输入数字")
请输入数字20
>>> a
'20'
>>> b=int(a)
>>> b
20
>>> c=str(b)
>>> c
'20'
>>>
```

这就是前面的问题 input 输入的结果是字符串，所以传入给 a 的也是字符串，a 就成了字符串变量

用 int 强制把 a 转成 int 整形数字

也可以用 str 将整形 b 转回成字符串

```
>>> print("xx xx", "bbbb")
xx xx bbbb
>>> print("xxxx", "bbbb")
xxxxbbbb
>>> print("xxxx", "bbbb", 12)
xxxxbbbb 12
>>>
```

，逗号是将两个字符串用空格隔开

也可以不用逗号两个字符串连接在一起

字符串和整形数字必须用逗号隔开

赋值运算符

```
>>> a = 3
>>> b = 5
>>> a > b
False
>>> a < b
True
>>> a == b
False
>>> a != b
True
>>> a >= b
False
>>> a <= b
True
>>>
```

和 C 语言一样，成立返回 true，不成立返回 false

如果我们要计算一个 a>b，但是 a<c，怎么做？

比如 a = ? , b=100 , c=100

```
a = 200
b = 100
c = 1000
if a>b:
    if a<c:
        print("a = ", a)
    else:
        print("a什么都不是")
>>>
a = 200
>>>
```

Python 的 if 是可以嵌套的，和 C 语言一样

```
a = 2000
b = 100
c = 1000
if a>b:
    if a<c:
        print("a = ", a)
    else:
        print("a什么都不是")
=>>>
a什么都不是
>>>
```

但是 python 给出了更好的方法

```
a = 200
b = 100
c = 1000
if b < a < c:
    print("a的值是", a)
else:
    print("a什么都不是")
>>>
a的值是 200
>>>
```

这就是 python 的变量同时判断左右两边

```
a = 50
b = 100
c = 1000
if b < a < c:
    print("a的值是", a)
else:
    print("a什么都不是")
=>>>
a什么都不是
>>>
a什么都不是
>>>
```

赋值运算符

```
>>> num = 1
```

```
>>> num+=1
```

```
>>> num
```

```
2
```

```
>>> num = 3
```

```
>>> num -= 1
```

```
>>> num
```

```
2
```

```
>>> num = 2
```

```
>>> num *= 2
```

```
>>> num
```

```
4
```

```
>>> num = 4
```

```
>>> num /= 2
```

```
>>> num
```

```
2.0
```

```
>>> num = 4
```

```
>>> num // = 2
```

```
>>> num
```

```
2
```

```
>>> num = 3
```

```
>>> num **=2
```

```
>>> num
```

```
9
```

```
>>> num = 5
```

```
>>> num %= 3
```

```
>>> num
```

```
2
```

逻辑运算符

```
>>> 5 > 3 and 6 > 2
```

```
True
```

```
>>> 5 > 3 and 6 < 2
```

```
False
```

```
>>>
```

```
>>> 5 > 3 or 6 < 2
```

```
True
```

```
>>> 5 > 3 or 6 > 2
```

```
True
```

```
>>> 5 < 3 or 6 < 2
```

```
False
```

```
>>>
```

```
>>> num=1
```

```
>>> num = num + 1
```

```
>>> num
```

```
2
```

```
>>>
```

```
>>> num = 3
```

```
>>> num = num - 1
```

```
>>> num
```

```
2
```

```
...
```

```
>>> num = 2
```

```
>>> num = num * 2
```

```
>>> num
```

```
4
```

```
...
```

```
>>> num = 4
```

```
>>> num = num / 2
```

```
>>> num
```

```
2.0
```

```
...
```

```
>>> num = 4
```

```
>>> num = num // 2
```

```
>>> num
```

```
2
```

```
...
```

```
>>> num = 3
```

```
>>> num = num ** 2
```

```
>>> num
```

```
9
```

```
...
```

```
>>> num = 5
```

```
>>> num = num % 3
```

```
>>> num
```

```
2
```

只有两边都为真，结果才为 true。有一边不为真，那么结果就为 false
也就是 C 语言的 (&& 与) 运算

只要有一边为真，结果就是 true，只有两边都不为真结果才是 false
也就是 C 语言的 (|| 或) 运算

```
>>> not 5 > 3  
False  
>>> not 5 < 3  
True  
>>>
```

not 取反这样用， $5 > 3$ 为真，但是取反就是 false

类似 C 语言的取反！运算符

While 循环

The screenshot shows two windows. On the left is a text editor window titled "test.py - D:\02 我的文档\D" containing the following code:

```
num = 1
while num<=10:
    print(num)
```

On the right is a Python shell window titled "Python 3.6.3 Shell" showing the output of the code:

```
Python 3.6.3 (v3.6.3rc1+dev-049f85d, Mar 27 2018, 16:29:59)
[PyPy 6.1.0+ (6.1.0+dev-049f85d, Mar 27 2018, 16:29:59) on win32]
Type "copyright", "credits" or "license" and hit Enter.  
>>>  
=====
```

A red box highlights the loop condition "num<=10" in the code, and another red box highlights the output "1 2 3 4 5 6 7 8 9 10" in the shell. A callout bubble points to the shell output with the text "和 C 语言一样只要这里为真就一直 while 循环". Another callout bubble points to the shell output with the text "大于 10 为假 while 退出".

用 while 设计判断年龄的游戏，年龄正确退出游戏

The screenshot shows three windows. The top-left is a text editor window titled "test.py - D:\02 我的文档\Desktop\test.py (3.6.3)" containing the following code:

```
arg = 50

while True:
    size = int(input("请输入年龄"))
    if size > 50:
        print("年龄过大")
    elif size < 50:
        print("年龄过小")
    else:
        print("年龄正确")
print("end")
```

The middle-right is a Python shell window titled "Python 3.6.3 (v3.6.3rc1+dev-049f85d, Mar 27 2018, 16:29:59) on win32" showing the interaction:

```
Python 3.6.3 (v3.6.3rc1+dev-049f85d, Mar 27 2018, 16:29:59) on win32
Type "copyright", "credits" or "license" and hit Enter.  
>>>  
=====
```

A red box highlights the user input "请输入年龄 50" in the shell. A callout bubble points to the shell output with the text "我们发现年龄输入正确还是无法退出游戏".

The bottom-left is a text editor window titled "test.py - D:\02 我的文档\Desktop\test.py (3.6.3)" containing the following code:

```
arg = 50

while True:
    size = int(input("请输入年龄"))
    if size > 50:
        print("年龄过大")
    elif size < 50:
        print("年龄过小")
    else:
        print("年龄正确")
        break
print("end")
```

The bottom-right is a Python shell window titled "Python 3.6.3 Shell" showing the interaction:

```
Python 3.6.3 (v3.6.3rc1+dev-049f85d, Mar 27 2018, 16:29:59) on win32
Type "copyright", "credits" or "license" and hit Enter.  
>>>  
=====
```

A red box highlights the user input "请输入年龄 100" in the shell. A callout bubble points to the shell output with the text "和 C 语言一样在 while 循环里面加 break 就可以退出循环".

continue 和 C 语言一样
不执行下面的程序，跳
会循环前面，重新执行

为什么这里没有 5，因
为 num==5 时
continue 被执行，跳过
print 语句，重新循环

while else 组合的无聊程序

如果循环没有被 break
中途退出，后面的 else
会被执行

如果循环被 break 中途
退出，后面的 else 不会
被执行

如果 i 小于 5 就一直执行
Tab 里面的语句

输出i值= 1
输出i值= 1
输出i值= 1
输出i值= 1
输出i值= 1
输出i值= 1

```
i=1
while i<=5:
    print("输出i值= %d" %i)
    i=i+1
```

每次循环 i 都加 1，所以 i 大于 5 时跳出死循环

输出i值= 1
输出i值= 2
输出i值= 3
输出i值= 4
输出i值= 5

记住 python 的 while 循环体内部实现的语句必须用 Tab 空格隔离出来，这和 C 语言的 while 不一样，这是我不喜欢 python 语言的主要原因，过度规范代码。

break 跳出循环使用

```
i=1
while i<=5:
    print("输出i值= %d" %i)
    i=i+1
    break
    print("没有跳出循环")
```

和 C 语言一样，while 里面出现 break 就跳出 while 循环，不执行循环最后位置的代码，而是执行循环外的代码

输出i值= 1
跳出循环

continue 在循环里不执行 continue 字符后面的语句，跳到循环开始位置重新执行循环

```
i=1
while i<=5:
    print("输出i值= %d" %i)
    i=i+1
    continue
    print("没有跳出循环")
```

和 C 语言的 continue 一样，循环执行一半回到循环开始位置

输出i值= 1
输出i值= 2
输出i值= 3
输出i值= 4
输出i值= 5
跳出循环

While 循环嵌套

```
i=1
z=1
while i<=3:
    print("输出i值= %d" %i)
    i=i+1
    while z < 5:
        print("z值输出=%d" %z)
        z=z+1
print("循环嵌套结束")
```

如果 while 循环里面有嵌套的 while 循环，就要执行嵌套的循环 和 C 语言 while 循环嵌套一样

输出i值= 1
z值输出=1
z值输出=2
z值输出=3
z值输出=4
输出i值= 2
输出i值= 3
循环嵌套结束

字符串拼接

```
name = "aaa"  
string = "bbb"  
print(name+string)
```

只需要将两个字符串变量加(+)起来
就实现字符串拼接了

aaabbb

输出一串相同的字符

```
name = "abc"  
  
print(name * 10)
```

一个字符串变量 `* 10` 就会输出 10 个相同的字符串，`* 20` 就会输出 20 个

abcabcabcabcabcabcabcabcabcabc

这种输出方式有什么用？

```
name = "#"  
  
print(name * 50)
```

你想让打印终端有层次感的输出，就用这种方式

字符串变量和整数变量不能进行相互计算

```
name = "abcde"  
num = 10  
print(name + n)
```

字符串变量和整数或者浮点数，或者其它类型的数字变量进行运算会报错

```
Traceback (most recent call last):
  File "D:/02 我的文档/Desktop/py01/py01.py", line 3, in <module>
    print(name + num)
TypeError: can only concatenate str (not "int") to str
```

input 函数输入字符

```
password = input("input number")  
  
print(password)
```

将输入的内容传递给变量

```
input number 10
```

```
password1 = input("input number1")  
password2 = input("input number2")  
print(password1+password2)
```

```
input number1 10  
input number2 20  
10 20
```

我想将两个输入的内容进行运算，为什么成了字符串拼接

这是因为 `input` 输入的都是字符串类型，所以接受 `input` 的变量也是字符串类型

数据类型转换

```
变量 = int(x) //就是将 x 转换成整数给变量
```

```
变量 = float(x) //就是将 x 转化成浮点型给变量
```

```
a = int("123")  
b = int("456")  
print(a+b)
```

将字符串转换成整形

579

现在我们可以解决上面 input 遇到的问题了

```
password1 = input("input number1")  
password2 = input("input number1")  
a = int(password1)  
b = int(password2)  
print(a+b)
```

将字符串转换成整形后问题得到解决

```
input number1 10  
input number1 20  
30
```

print 函数格式化输出

```
name = "名字变量"  
print("格式化输出 %s" %name)
```

%s 输出字符串

```
num = 55  
print("整形数字输出 %d" %num)  
print("数字前面加内容两个0输出 = %04d" %num)
```

%d 输出整形

```
#%04就是输出不到4位就用0来占位，如果输出超过4位，输出多少就是多少  
  
f = 15.28  
print("小数输出 = %f" %f)
```

%f 输出小数

```
格式化输出 名字变量  
整形数字输出 55  
数字前面加内容两个0输出 = 0055  
小数输出 = 15.280000
```

加 . 多少 f 就是保留小数后几位输出

```
f = 15.2825  
print("小数输出 = %f" %f)  
print("保留小数后两位输出 = %.2f" %f)
```

```
小数输出 = 15.282500  
保留小数后两位输出 = 15.28
```

转义字符让字符输出格式更工整

```
print("1 2 3")
print("10 20 30")
```

```
1 2 3
10 20 30
```

没有转义字符，两行
输出是错开排列的

用\t来实现字符工整排列

```
print("1\t2\t3")
print("10\t20\t30")
```

```
1      2      3
10     20     30
```

字符是不是很工整了

用\n在print里面换行

```
print("1234\n56")
```

```
1234
56
```

这就是换行

函数定义与使用

现在定义两个文件，py01, py02。py01文件调用py02文件里面的函数

```
py02.py x
def function1():
    print("输出function1函数内容")
    print(".....")
```

def关键字就是定义函数的
意思，function1就是自己
随意写的函数名

函数名定义后，内容一定要用Tab来隔
离，这Tab就类似C语言的{.....}

import关键字在主程序文件里面导入另外的文件

```
py01.py x
1 ➤ import py02
2
3 py02.function1()
输出function1函数内容
.....
```

我这里导入了py02文
件，因为要使用py02
文件里面的function1

使用py02文件里面的
function1函数

这就是函数定义和函数调用的方法

其实python的函数调用和C语言差不多，而且更加方便，import类似C语言的include包含要调用的头文件，然后(文件名.函数名)这种写法就类似C语言调用头文件里面的函数

为了验证函数是否功能正确，我们可以在主程序文件写一些函数来实验

```
py01.py
1 def fun2():
2     print("函数实验")
3     i=1
4     while i<3:
5         print("函数实验=%d"%i)
6         i=i+1
7
8 fun2()
```

我可以在我自己的文件里面直接定义函数，这样就不需要主程序 import 了

函数实验
函数实验=1
函数实验=2

主程序可以直接调用 fun2 函数，因为函数在本文件定义的

这种方法好处就是把函数调试好以后，再单独建立 python 文件来封装

有一个关键的问题，和 C 语言函数定义问题一样

```
def fun2():
    print("函数实验")
    i=1
    while i<3:
        print("函数实验=%d"%i)
        i=i+1
```

NameError: name 'fun2' is not defined

函数不能在没有定义内容之前就使用，所以这个 func2 只能放在 def fun2 函数的下面

函数形参使用

```
def fun_num(num1, num2):
    print("num1 = %d" %num1)
    print("num2 = %d" %num2)

fun_num(10, 20)
```

这就是函数形参 num1，num2 和 C 语言函数形参一样，只是不需要定义类型

num1 = 10
num2 = 20

函数返回值

```
def fun_num(num1, num2):
    ret = num1+num2
    return ret

num = fun_num(10, 20)
print("return num = %d" %num)
```

和 C 语言一样，直接 return 返回值就是

函数返回计算后的值给变量

return num = 30

```

def fun_num(num1, num2):
    ret = num1+num2
    print("函数return之前")
    return ret
    print("函数return之后")

num = fun_num(10, 20)
print("return num = %d" %num)

```

和 C 语言一样 return 之后的代码不会被执行

函数return之前
return num = 30

函数调用函数，就是函数嵌套使用

```

def test1():
    print("函数test1")

def test2():
    print("函数test2")
    test1()
    print("调用test1函数之后")

test2()

```

函数里面调用另一个函数

函数test2
函数test1
调用test1函数之后

函数嵌套和 C 语言一样

Python 列表用法

在 C 语言中，有 char , int , short , long... 等等，但是 python 在这些数据类型基础上多增加了一些非数字变量的数据类型，非数字变量就是指新增加的这些数据类型除了存放数字，还可以存放字符串，等等.....这些新增的数据类型叫做：列表，元组，字典，字符串。

列表使用

列表是用来存放一串数据

```

value1 = "zhangshan"
value2 = "lishi"
value3 = "wangwu"
print("value1 = %s" %value1)
print("value2 = %s" %value2)
print("value3 = %s" %value3)

```

以前 C 语言或者普通 python 都是一个字符串存放一个变量，下面用列表来改善这种情况

value1 = zhangshan
value2 = lishi
value3 = wangwu

列表使用

```

name = ["zhangshan", "lishi", "wangwu"]
print("name = %s" %name)

name = ['zhangshan', 'lishi', 'wangwu']

```

这种列表有什么意义呢？我 C 语言定义一个 char [][] 二维数组不就行了??

列表数据与数据之间
用逗号分隔

```
name = ["zhangshan", "lishi", "wangwu"]
print(name[0])
print(name[1])
print(name[2])
```

获取列表里面的数据就和 C 语
言使用数组下标方式一样

```
zhangshan
lishi
wangwu
```

下面我们来说明为什么列表不能用 C 语言二维数组代替

```
name = ["zhangshan", 1.55, 100]
print(name[0])
print(name[1])
print(name[2])
```

列表里面的元素可以是任意的数据
类型，整形，浮点型，字符串。而
C 语言的二维数组只能是一种数据
类型，这就是列表的优势

```
zhangshan
1.55
100
```

列表实现增删查改。根据写入列表里面元素内容搜索出内容在该列表什么位置

列表 index 函数使用

```
name = ["zhangshan", 1.55, 100]
```

```
result = name.index(1.55)
print(result)
result = name.index("zhangshan")
print(result)
```

```
1
0
```

你看 index 根据输入的列表元素值，返回该元素值在列表的第几个下标

修改列表里面的某个元素

```
name = ["zhangshan", 1.55, 100]
name[1] = "李四"
print(name)
```

和 C 语言一样，修改列表里面某个元
素，直接给该元素所在位置的下标重
新赋值就是了

```
['zhangshan', '李四', 100]
```

append 使用，给列表增加数据

```
name = ["zhangshan", 1.55, 100]
name.append("新增数据")
print(name)
```

这种直接使用 append(.) 向
里面写入内容，这个内容只
能默认添加到列表最后

```
['zhangshan', 1.55, 100, '新增数据']
```

如果想在列表中间某个位置插入数据，就要有 insert

```
name = ["zhangshan", 1.55, 100]
name.insert(1, "插入数据")
print(name)
```

```
['zhangshan', '插入数据', 1.55, 100]
```

insert (值 1, 值 2)

值 1: 写入你要插入数据的位置。比如我写 1，那么数据就会插入在 1 下标前面。然后变成下标 1 的数据。

extend 使用，两个列表相互插入数据，也就是一个列表数据插入给另一个列表

```
name = ["zhangshan", 1.55, 100]
list = ["新列表数据", 3.14, 2000]
name.extend(list)
print(name)
```

```
['zhangshan', 1.55, 100, '新列表数据', 3.14, 2000]
```

extend , 指定要插入的列

表，这个列表插在要插入列
表的末尾

remove 使用，删除列表里面某个元素

```
name = ["zhangshan", 1.55, 100]
name.remove(1.55)
print(name)
```

指定元素数值，直接删除

```
['zhangshan', 100]
```

pop 使用，默认将列表最后一个元素删除

```
name = ["zhangshan", 1.55, 100]
name.pop()
print(name)
```

1 个 pop 删除列表一个末尾元素

```
['zhangshan', 1.55]
```

```
name = ["zhangshan", 1.55, 100]
name.pop()
name.pop()
print(name)
```

2 个 pop 删除列表两个末尾元
素，以此类推

```
['zhangshan']
```

如果我们不知道列表里面的内容，但是我知道要删除列表的下标号，怎么删？？

```
name = ["zhangshan", 1.55, 100]
name.pop(1)
print(name)
```

```
['zhangshan', 100]
```

直接在 pop 里面写
入下标名称就是

`del` 使用，`del` 也可以删除列表里面的某个元素

```
name = ["zhangshan", 1.55, 100]
del name[1]
print(name)
['zhangshan', 100]
```

```
name = ["zhangshan", 1.55, 100]
del name[1]
del name
```

`del` 可以将整个变量，列表从内存中删除。

那么到底是用 `pop` 还是用 `del` 呢？记住在删除列表内容时就用列表自身的 `pop` 函数，删除变量时才使用 `del`，比如有些变量使用之后就再也不使用了，这种就可以用 `del` 来删除变量

列表实现统计功能

`len` 函数统计列表长度

```
name = ["zhangshan", 1.55, 100, "李四", "统计", "李四"]
list_len = len(name)
print("列表长度，也就是列表元素个数 = %d" %list_len)
```

```
列表长度，也就是列表元素个数 = 6
```

`count` 函数，查看列表里面同名的元素出现几次

```
name = ["zhangshan", 1.55, 100, "李四", "统计", "李四"]
count = name.count("李四")
print("内容在列表出现次数 = %d" %count)
```

```
内容在列表出现次数 = 2
```

向 `count` 写入你要查找的内容，它就会返回该内容在列表出现的次数

```
name = ["zhangshan", 1.55, 100, "李四", "统计", "李四"]
name.remove("李四")
print(name)
['zhangshan', 1.55, 100, '统计', '李四']
```

如果列表出现了多次相同的内容，`remove` 会删除列表相同内容中第一次出现的内容

`sort` 函数升序，降序使用

```
name = ["axxx", "bzzzz", "cdddddd", "deeeeeeee"]
name.sort()
print(name)
['axxx', 'bzzzz', 'cdddddd', 'deeeeeeee']
```

感觉是列表按照字符串数量进行排序的，字符串少的在前面字符串多的在后面，真是这样吗？

```
name = ["axxx", "bzzzz", "cdfffff", "aaaaaaaa"]  
name.sort()  
print(name)  
['aaaaaaaa', 'axxx', 'bzzzz', 'cdfffff']
```

我将字符串最多的一列第一个字符修改了

其实 sort 是根据字符优先级来排列的，最高优先级先看字符大小，a~z 字符最大的再前面，次优先级遇到相同字符的就看字符串多少，字符串多的再前面，感觉 sort 不是很好用。

```
name = [100, 50, 110, 70, 20]  
name.sort()  
print(name)  
[20, 50, 70, 100, 110]
```

sort 用在数字升序上是很好用的，只是字符串升序不好用

```
name = [100, 50, 110, 70, 20]  
name.sort(reverse=True)  
print(name)  
[110, 100, 70, 50, 20]
```

向 sort 里面写入 True，就使列表降序排列

reverse 函数使列表元素相邻之间进行调换，然后最后的元素放在列表最前面，这叫逆序

```
name = [700, 50, 110, 70, 20, 500]  
name.reverse()  
print(name)  
[500, 20, 70, 110, 50, 700]
```

你看 500 和 20 顺序颠倒了
110 和 70 顺序颠倒了

Python 元组用法

在 C 语言中，有 char , int , short , long.... 等等，但是 python 在这些数据类型基础上多增加了一些非数字变量的数据类型，非数字变量就是指新增加的这些数据类型除了存放数字，还可以存放字符串，等等..... 这些新增的数据类型叫做：列表，元组，字典，字符串。

```
mytuple = ("xxxxzzz", 158, 1.65, -30)  
  
print(mytuple)
```

只是元组里面的元素都是用大括号包括

元组和列表类似，都是类似 C 语
言定义数组的方式来存放数据

获取元组里面数据的方法

```
mytuple = ("xxxxzzz", 158, 1.65, -30)

print(mytuple[0])
print(mytuple[1])
print(mytuple[2])
print(mytuple[3])
```

```
xxxxzzz
158
1.65
-30
```

元组获取里面的数据和列表，C 语言数组类似，都是用下标获取。

既然这样那么元组有什么用呢？

```
mytuple = ("xxxxzzz", 158, 1.65, -30)

mytuple[0] = 50
TypeError: 'tuple' object does not support item assignment
```

元组和列表的区别就是元组里面的内容不能被修改，元组定义之后，数据只有只读权限

index 函数使用，元组和列表一样可以经过内容找到对应的下标号

```
mytuple = ("xxxxzzz", 158, 1.65, -30)

ret = mytuple.index(158)
print("ret = %d" %ret)
```

```
ret = 1
```

获取了数字 158 在元组中的下标位置

count 函数使用，统计元组里面一个内容出现过几次

```
mytuple = ("xxxxzzz", 158, 1.65, -30, 158)

num = mytuple.count(158)
print("num = %d" %num)
```

158 在元组出现过 2 次

```
num = 2
```

```
mytuple = ("xxxxzzz", 158, 1.65, -30, 158)

num = len(mytuple)
print("num = %d" %num)
```

统计元组里面元素个数

```
num = 5
```

元组主要应用场景

- 1.元组做函数参数，或者函数返回值
- 2.元组格式化字符串
- 3.让列表里面内容不可以被修改

2.元组格式化字符串使用方法

```
mytuple = ("小明", 28, 1.75)
```

输出格式化顺序按照元组来排列

```
print("%s 年龄是:%d 身高:%f" %mytuple)
```

小明 年龄是:28 身高:1.750000

等效于这样编写

```
print("%s 年龄是:%d 身高:%f" % ("小明", 28, 1.75))
```

小明 年龄是:28 身高:1.750000

3.让列表里面内容不可以被修改使用方法

```
list = [10, 30, 40, 20]
```

```
yuanzu = tuple(list)
```

```
print(yuanzu)
```

将列表放入 tuple 关键字，这样 tuple 将复制列表的内容生成个元组返回给变量

```
list[0]=20
```

这是我去修改列表，输出没有问题

```
(10, 30, 40, 20)
```

```
[20, 30, 40, 20]
```

```
list = [10, 30, 40, 20]
```

```
yuanzu = tuple(list)
```

```
print(yuanzu)
```

```
list[0]=20
```

```
print(list)
```

```
yuanzu[0] = 100
```

我元组继承了列表的内容，这样元组里面的内容就不能修改了，但是列表的内容可以被修改。所以你要对列表进行数据保护，就用元组继承列表的内容，然后操作元组来计算

```
TypeError: 'tuple' object does not support item assignment
```

Python 字典用法

在 C 语言中，有 char , int , short , long.... 等等，但是 python 在这些数据类型基础上多增加了一些非数字变量的数据类型，非数字变量就是指新增加的这些数据类型除了存放数字，还可以存放字符串，等等.....这些新增的数据类型叫做：列表，元组，字典，字符串

```
zidian = {"name": "小明", "age": 20}  
print(zidian)
```

{'name': '小明', 'age': 20}

字典一个元素由一个冒号的左右两边
值组成，KEY(键) : value(值)，
1 个键值对一个元素，元素之间用逗
号隔离开

```
zidian = {"name": "小明",  
          "age": 20,  
          "height": 175,  
          "weight": 75.5}  
print(zidian)
```

{'name': '小明', 'age': 20, 'height': 175, 'weight': 75.5}

为了好看我们可以把键值
对按照列来排列编写

字典里面键值对用大
括号括起来

注意这里键值打印输出内容，在有些编译器里面顺序是乱的，这不重要，字典是根据键名来获取对应值的

根据键值来获取字典里面键值对应的数据

```
zidian = {"name": "小明",  
          "age": 20,  
          "height": 175,  
          "weight": 75.5}  
  
ret = zidian["name"]  
print("name = %s" %ret)
```

name = 小明

获取字典里面的数据
我们只需要填入键的
标识，字典就会返回
该键对应的值

```
zidian = {"name": "小明",  
          "age": 20,  
          "height": 175,  
          "weight": 75.5}  
  
ret = zidian["name"]  
print("name = %s" %ret)  
value = zidian["age"]  
print("age = %d" %value)
```

name = 小明
age = 20

获取两个键值

字典的增删改查

```
zidian = {"name": "小明",
          "age": 20,
          "height": 175,
          "weight": 75.5}

zidian["name"] = 500
print(zidian)
{'name': 500, 'age': 20, 'height': 175, 'weight': 75.5}
```

填入键值，修改键值对应的

```
zidian = {"name": "小明",
          "age": 20,
          "height": 175,
          "weight": 75.5}

zidian["xzz"] = 1000
print(zidian)
{'name': '小明', 'age': 20, 'height': 175, 'weight': 75.5, 'xzz': 1000}
```

字典里面没有
xzz 键值

修改字典里面键值时，如果这个键值不存在，那么这个键值就会增加到字典后面

```
zidian = {"name": "小明",
          "age": 20,
          "height": 175,
          "weight": 75.5}

zidian.pop("name")
print(zidian)
{'age': 20, 'height': 175, 'weight': 75.5}
```

pop 删除了字典对应的键值和对应的数据

计算字典长度，合并字典

```
zidian = {"name": "小明",
          "age": 20,
          "height": 175,
          "weight": 75.5}

#统计字典里面键值的数量
ret = len(zidian)
print("字典键值数量 = %d" % ret)

#合并字典
newzidian = {"新字典": 0}
newzidian.update(zidian)
print(newzidian)
```

Len 计算字典长度，计算
的元素只有键数量

将 zidian 字典变量添加到新字典
newzidian 变量后面，用 update 函数

```
字典键值数量 = 4
{'新字典': 0, 'name': '小明', 'age': 20, 'height': 175, 'weight': 75.5}
```

如果两个字典里面有相同的键值会出现什么情况？？

```
zidian = {"name": "小明",
          "age": 20,
          "height": 175,
          "weight": 75.5}

#合并字典
newzidian = {"新字典": 0, "age": 10}
newzidian.update(zidian)
print(newzidian)
{'新字典': 0, 'age': 20, 'name': '小明', 'height': 175, 'weight': 75.5}
```

原有的 zidian 里面 age 键
值对会覆盖 newzidian 里
面新 age 键值对

```
zidian = {"name": "小明",
          "age": 20,
          "height": 175,
          "weight": 75.5}

#清空字典
zidian.clear()
print(zidian)
```

clear 清空字典里面所有内容

列表里面加入字典

```
list = [
    {},
    {}
]
print(list)
```

在列表里面加入两个大括号
就是两个字典

在两个字典里面写入每
个人的基本信息

```
list = [
    {"name": "张三", "age": 20, "height": 185},
    {"name": "李四", "age": 30, "height": 170}
]
print(list)
[{'name': '张三', 'age': 20, 'height': 185}, {'name': '李四', 'age': 30, 'height': 170}]
```

```
list = [
    {"name": "张三", "age": 20, "height": 185},
    {"name": "李四", "age": 30, "height": 170}
]
for value in list:
    print(value)
```

for 循环一次获取一行字典里面的信息赋值给
value , 然后打印出来

```
{'name': '张三', 'age': 20, 'height': 185}
{'name': '李四', 'age': 30, 'height': 170}
```

Python 字符串类型

```
str1 = "I love python"
print(str1[3])
```

和数组下标一样 , 用下标
获取字符串

空格也占用 1 个字符的

```
str1 = "I love python"
for char in str1:
    print(char)
```

循环获取字符串变量 , 就是
每一次循环 str1 里面的元素
赋值给 char , str1 自己++
地址 , 直到 str1 地址加到字
符串末尾 , 退出循环



```
str1 = "I love python"

long = len(str1) #计算字符串个数
print("字符串长度 = %d" % long)

num = str1.count("o") #统计字符串某个字符出现的次数
print("o 出现次数 = %d" % num)
num = str1.count("lo") #统计字符串某段字符出现的次数
print("某段字符个数 = %d" % num)
index = str1.index("lo") #查找lo这段字符在字符串什么下标位置
print("索引位置 = %d" % index)
```

字符串长度 = 13
o 出现次数 = 2
某段字符个数 = 1
索引位置 = 2

这就是字符个数计算 , 查找某个字符出现个数和查找字符在字符串的下标位置

```
str1 = "I love python"

index = str1.index("lx") #查找lo这段字符串在
print("索引位置 = %d" %index)
用 index 函数查找的字符串不存在，程序会报错
ValueError: substring not found
```

有些时候我们希望就算是字符串不存在也不要出现运行错误，而是继续运行

```
str1 = "I love python"
ret = str1.find("lov")
print(ret)
ret = str1.find("xo")
print(ret)
```

find 就解决了这个问题
find 搜索字符串里面有对应的字符返回大于1

find 搜索字符串里面没有对应的字符就返回-1

2
-1

这样就是 C 语言的返回值用法，把错误发给服务器或者其它程序去识别

字符串算法

```
str1 = ""
flag = str1.isspace()
print("该字符串是不是空白字符 %s" %flag)
```

isspace 就是判断字符串里面是不是空字符，空字符返回 True，非空字符返回 False

```
str1 = " "
flag = str1.isspace()
print("该字符串是不是空白字符 %s" %flag)
```

该字符串是不是空白字符 False

该字符串是不是空白字符 True

```
str1 = " \t\n\r"
```

这些对齐，换行，回车制表符在字符串里面也属于空白字符

```
flag = str1.isspace()
print("该字符串是不是空白字符 %s" %flag)
```

该字符串是不是空白字符 True

```
str1 = " abc124"
```

只有实实在在字符在字符串
里面才不是空白字符

```
flag = str1.isspace()
```

```
print("该字符串是不是空白字符 %s" %flag)
```

该字符串是不是空白字符 False

```
str1 = "abc124"
```

```
flag = str1.isdecimal()
```

```
print("该字符串是不是数字 %s" %flag)
```

```
str2 = "124"
```

```
flag = str2.isdecimal()
```

```
print("该字符串是不是数字 %s" %flag)
```

该字符串是不是数字 False

该字符串是不是数字 True

isdecimal 只能判断字符
串里面是不是全是数字，
全是数字返回 True，不完
全是数字，比如涵盖了字
符，就返回 False

```
str1 = "hello index"
```

```
flag = str1.startswith("hello")
```

```
print("该字符串是不是根据指定字符串开头的 %s" %flag)
```

startswith 判断字符串是不
根据指定的字符开头的，如
果是返回 True，不是返回 Flase

```
flag = str1.startswith("ello")
```

```
print("该字符串是不是根据指定字符串开头的 %s" %flag)
```

该字符串是不是根据指定字符串开头的 True

该字符串是不是根据指定字符串开头的 False

```
str1 = "hello index"
```

```
flag = str1.endswith("dex")
```

```
print("该字符串是不是根据指定字符串结束 %s" %flag)
```

endswith 判断字符串是不
根据指定的字符结尾，如
果是返回 True，不是返回 Flase

```
flag = str1.startswith("in")
```

```
print("该字符串是不是根据指定字符串结束 %s" %flag)
```

```
该字符串是不是根据指定字符串结束 True  
该字符串是不是根据指定字符串结束 False
```

```
str1 = "hello index"  
print(str1)  
str2 = str1.replace("hello", "world")  
print(str2)
```

replace(参数 1,参数 2)
参数 1：是需要修改的字符串
参数 2：写入修改的字符串
修改后的字符串会返回给新变量

```
hello index  
world index
```

字符串文本对齐

```
poem = ["蹬高黄鹤楼",  
        "百日宴",  
        "一行五百千里",  
        "更上一层楼"]
```

```
print(poem)
```

```
[ '蹬高黄鹤楼', '百日宴', '一行五百千里', '更上一层楼' ]
```

这样横着看字符串列表不舒服，我们用 for 循环让它竖着排列显示

```
poem = ["蹬高黄鹤楼",  
        "百日宴",  
        "一行五百千里",  
        "更上一层楼"]
```

```
for char in poem:  
    print(char)
```

这些字符确实是排列输出的，但是感觉没有居中对齐

```
蹬高黄鹤楼  
百日宴  
一行五百千里  
更上一层楼
```

```
poem = ["蹬高黄鹤楼",  
        "百日宴",  
        "一行五百千里",  
        "更上一层楼"]
```

```
for char in poem:  
    print(char.center(10))
```

center(参数 1)
参数 1: 填入数字，字符串
就会用英文空格填充这行
字符串到这么多个数字

```
蹬高黄鹤楼  
百日宴  
一行五百千里  
更上一层楼
```

意思就是我 center(10) ，蹬高黄鹤楼只有 5 个字符，那么这段字符前后还要加入空格来填充这行字符串，填充到这行字符串是 10 个字符为止

但是填充后看起来每行字符是居中了，但是还是没有对齐

```
poem = ["蹬高黄鹤楼",
        "百日宴",
        "一行五百千里",
        "更上一层楼"]

for char in poem:
    print(" | %s | "%char.center(10))
```

我用竖线来优化一下，看起来字符串确实没有完全对齐

蹬高黄鹤楼	
百日宴	
一行五百千里	
更上一层楼	

```
poem = ["蹬高黄鹤楼",
        "百日宴",
        "一行五百千里",
        "更上一层楼"]
```

ljust 函数就是让字符串左对齐

```
for char in poem:
    print(" | %s | "%char.ljust(10))
```

蹬高黄鹤楼	
百日宴	
一行五百千里	
更上一层楼	

rjust 函数就是让字符串右对齐

```
for char in poem:
    print(" | %s | "%char.rjust(10))
```

蹬高黄鹤楼	
百日宴	
一行五百千里	
更上一层楼	

这种字符串处理方法到底用来干什么的？

```
poem = ["\t\n蹬高黄鹤楼",
        "百日宴",
        "一行五百千里\r",
        "更上一层楼"]

for char in poem:
    print(" | %s | "%char.center(10))
```

正常情况下我们从网页爬来的数据都是这样杂乱无序的，里面包含一些制表符

蹬高黄鹤楼	
百日宴	
一行五百千里	
更上一层楼	

```
poem = ["\t\n蹬高黄鹤楼",
        "百日宴",
        "一行五百千里\r",
        "更上一层楼"]

for char in poem:
    print(" | %s | "%char.strip())
```

strip 删除字符串的前后多余制表符，比如删除\r\t\n

蹬高黄鹤楼	
百日宴	
一行五百千里	
更上一层楼	

这样你爬出来的网页就清晰可见了

字符串拆分和拼接

```
poem = "网页抓取的字符串\r12345\nabcde\t"  
print(poem)
```

这是网页爬取的信息，因为有这些\t\n\r 空白字符串，导致网页的数据输出不完整，而且很乱

12345
abcde

```
poem = "网页抓取的字符串\r12345\nabcde\t"
```

```
list = poem.split()
```

```
print(list)
```

```
['网页抓取的字符串', '12345', 'abcde']
```

split 就是将有制表符的字符串，以制表符作为分割线，把分割的字符串封装成列表(记住是列表)返回

这样就解决了网页爬取的数据有制表符这些乱七八糟的空白字符

split 也就是字符串拆分

合并字符串

```
poem = "网页抓取的字符串\r12345\nabcde\t"
```

```
#list = poem.split()  
result = " ".join(poem)  
print(result)
```

如果直接用join的话
字符串的制表符没有
取消，合并出来的字
符串就很怪

1 2 3 4 5
a b c d e

```
poem = "网页抓取的字符串\r12345\nabcde\t"
```

```
list = poem.split()  
result = " ".join(list)  
print(result)
```

split 拆分后就是完整的字符串列表了，这时候用join
来合并列表里面的每一段字符串是不错的

```
网页抓取的字符串 12345 abcde
```

```
poem = "网页抓取的字符串\r12345\nabcde\t"
```

```
list = poem.split()  
result = " ".join(list)  
print(result)
```

Join 前面“ ”号里面的数据
决定了拼接字符串用什么符号，现在是直接拼接

```
网页抓取的字符串12345abcde
```

```
list = poem.split()  
result = "-".join(list)  
print(result)
```

Join 前面“ ”号里面的数据
决定了拼接字符串用什么符
号，这是用 - 拼接字符串

网页抓取的字符串-12345-abcde

字符串切片

```
strnum = "0123456789"  
  
result = strnum[2:6] #截取下标2到下标6前面下标的字符串  
print(result)  
  
numend = strnum[2:] #截取下标2到末尾的字符串  
print(numend)  
  
numstar = strnum[:5] #截取开始到下标5前面下标的字符串  
print(numstar)
```

截取符号就是[参数 1: 参数 2]

参数 1: 截取字符串的开始下标
参数 2: 截取字符串的最后一个下标，但是记住你写的是最
后一个下标，但是截取的是最后一个前一个下标

2345
23456789
01234

```
strnum = "0123456789"  
  
num = strnum[::2]  
print(num)
```

[:: 参数] 两个冒号表示按
照步长来获取字符串数据，
[:2] 从 0 地址开始，每增加
两个地址，获取一个值

02468

用 C 语言地址方式理解 python 字符串比较方便

```
strnum = "0123456789"  
  
num = strnum[1::2]  
print(num)
```

[:: 参数] 两个冒号表示按照步长来获取字符
串数据，
[参数 1 :: 参数 2]
参数 1 表示获取字符数据的开始地址
参数 2 表示每增加几个地址获取一次数据

13579

```
strnum = "0123456789"  
  
num = strnum[-2::]  
print(num)
```

89

-2 表示从字符串到时候第 2
个数据开始获取 ,然后一直获
取到字符串末尾结束

计算字符串里面最大数值

```
strnum = "abcd12345"  
str = max(strnum) #max函数计算字符串中那个字符串最大  
print("最大字符 = %s" %str)
```

```
strnum2 = "12345"  
str = max(strnum2) #max函数计算字符串中那个字符串最大  
print("最大数字字符 = %s" %str)
```

```
strnum3 = "123a45"  
str = max(strnum3) #max函数计算字符串中那个字符串最大  
print("数字和英文字符串大小对比 = %s" %str)
```

最大字符 = d
最大数字字符 = 5
数字和英文字符串大小对比 = a

计算字符串里面最小数值

```
strnum = "abcd12345"  
str = min(strnum) #min函数计算字符串中那个字符串最小  
print("最小字符 = %s" %str)
```

```
strnum2 = "12345"  
str = min(strnum2) #min函数计算字符串中那个字符串最小  
print("最小数字字符 = %s" %str)
```

```
strnum3 = "123a45"  
str = min(strnum3) #min函数计算字符串中那个字符串最小  
print("数字和英文字符串大小对比 = %s" %str)
```

最小字符 = 1
最小数字字符 = 1
数字和英文字符串大小对比 = 1

计算列表最大值最小值

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

```
print("列表最大值 = %d" %max(list))  
print("列表最小值 = %d" %min(list))
```

```
列表最大值 = 10  
列表最小值 = 1
```

计算字典最大值最小值，记住字典只能对比 key 值不能对比 key 对应的 value 值

```
dict = {"key1": "50",  
        "key2": "40",  
        "key3": "30",}
```

看到没，字典只对比 key 的值，后面的 50,40,30 的值不会对比，按道理说 key1 是 50，值是最大的，为什么 key1 不能是最大值呢？因为对比 key 的话只有 key3 是最大值

```
print("字典最大值 = %s" %max(dict))  
print("字典最小值 = %s" %min(dict))
```

```
字典最大值 = key3  
字典最小值 = key1
```

列表和元组也可以像字符串那样切片

```
list = [1, 2, 3, 4, 5, 6]  
qiebian = list[2:5] #列表切片和字符串一样  
print(qiebian)
```

```
yuanzu = (1, 2, 3, 4, 5, 6)  
yz = yuanzu[2:5] #元组切片和字符串一样  
print(yz)
```

```
[3, 4, 5]  
(3, 4, 5)
```

但是记住字典是不能切片的

增加列表，元组数据长度

```
list = [1, 2, 3, 4, 5]
print(list)
list = list*5#把列表里面的数据重复增加5次
print(list)

yuanzu = (10, 20, 30, 40, 50)
print(yuanzu)
yuanzu = yuanzu*5#把元组里面的数据重复增加5次
print(yuanzu)
```

[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
(10, 20, 30, 40, 50)
(10, 20, 30, 40, 50, 10, 20, 30, 40, 50, 10, 20, 30, 40, 50, 10, 20, 30, 40, 50)

这样就让(列表/元组)的元素在自己(列表/元组)中复制了 5 次，增加到 25 个元素

记住字典不支持 * 号操作符，所以字典无法复制

除了合并字符串，字符串还可以两个拼接，列表元组也可以

```
str1 = "python"
str2 = "java"
str = str1+str2
print(str)
```



+号就是把两个字符串拼接在一起

pythonjava

```
list1 = [1, 2]
list2 = [3, 4]
list = list1+list2
print(list)
```



列表也是一样

[1, 2, 3, 4]

```
yuanzu1 = (1, 2)
yuanzu2 = (3, 4)
yuanzu = yuanzu1+yuanzu2
print(yuanzu)
```



(1, 2, 3, 4)

extend 函数使用

```
list1 = [1, 2]
list2 = [3, 4]
list1.extend(list2)
print(list1)
```

extend 和 + 号不一样，
extend 写入的列表会将原来
list1 的列表内容修改

```
[1, 2, 3, 4]
```

```
list1 = [1, 2]
list2 = [3, 4]
list1.append(list2)
print(list1)
```

append 和 extend 不一样，extend 是将新列表元素
加在现有列表元素后面，而 append 是直接将新列表
元素当成一个新的列表加在现有列表元素后面，那么现
有列表里面就又包含了一个列表

```
[1, 2, [3, 4]]
```

in , not in 运算符

```
str1 = "abcde"
str = "a" in str1 #in标识字符串变量里包含不包含a, 包含返回True
print(str)
```

True
False

in 语法规则: (要寻找的字符串 in 在什么变量里面查找要寻找的字符串)

not in 语法规则一样

```
list1 = [0, 1, 2]
list = 1 in list1 #列表in使用方法
print(list)
```

```
list2 = [0, 1, 2]
list = 1 not in list1#列表not in 使用方法
print(list)
```

True
False

Python 除了 if....else...以外，还增加了 for....else....

```
for num in [1,2,3,4]:  
    print(num)  
print("循环结束")
```

```
1  
2  
3  
4  
循环结束
```

这是我们常规的 for 循环，
把列表的数据都放在 num
变量里面循环执行

```
list = [1,2,3,4]  
for num in list:  
    print(num)  
print("循环结束")
```

```
1  
2  
3  
4  
循环结束
```

这种列表变量写法，和上面一样的

```
list = [1,2,3,4]  
for num in list:  
    print(num)  
  
else:  
    print("循环正常结束就会执行")  
print("循环结束")
```

把列表的元素循环完
就会执行 else 的语句

```
1  
2  
3  
4  
循环正常结束就会执行  
循环结束
```

```
list = [1,2,3,4]  
for num in list:  
    print(num)  
    break  
  
else:  
    print("循环正常结束就会执行")  
print("循环结束")
```

没有把列表元素循环执行
完，中途就 break 跳出循
环了，这样 else 里面的语
句是不会执行的

```
1  
循环结束
```

这就是 forelse...的使用方法

for...else...实际应用案例

```
students = [{"name": "小明"},  
           {"name": "小美"}]  
  
find_name = "小美"  
  
for list in students:  
    if list["name"] == find_name:  
        print("找到了 %s" % find_name)  
  
print("循环结束")
```

我们寻找小美，找到了输出

找到了 小美
循环结束

```
students = [{"name": "小明"},  
           {"name": "小美"}]  
  
find_name = "张三"  
  
for list in students:  
    if list["name"] == find_name:  
        print("找到了 %s" % find_name)  
  
print("循环结束")
```

我们寻找张三，发现字典列表里面没有张三，直接就循环结束了，什么都不提示

循环结束

```
students = [{"name": "小明"},  
           {"name": "小美"}]  
  
find_name = "张三"  
  
for list in students:  
    if list["name"] == find_name:  
        print("找到了 %s" % find_name)  
        break  
  
else:  
    print("没有找到 %s" % find_name)  
print("循环结束")
```

如果我们加入 break，找到了对应名字的人就 break 跳出循环，else 不会被执行。如果没有找到对应的人就把循环执行完，然后 else 执行提示

没有找到 张三
循环结束

这样 else 里面的语句可以发送给服务器，提示没有这个人存在。

Python 实现名片管理系统案例



```
inputstr=input("请选择执行程序:") #input返回用户输入的字符串
print("你输入的操作是 %s" %inputstr)

if inputstr in ["1","2","3"]:
    pass
elif inputstr == "0":
    pass
else:
    print("输入错误 重新输入")
```

(if 参数 in 列表), 这种语法的判断可以让变量 inputstr 和列表的 3 个参数依次判断，只要列表里面有任何一个字符等于 inputstr 变量的值就执行 if 里面的内容

```
请选择执行程序:2
你输入的操作是 2
```

这时候是写程序框架，暂时不知道判断后要执行些什么函数，所以可以用 pass 占位符先占用 if 里面的内容，如果 if 下面是空内容会执行报错。pass 其实就是空操作。

加入死循环，这样输入错误了可以重新执行输入程序

```
while True:
    inputstr=input("请选择执行程序:") #inp
    print("你输入的操作是 %s" %inputstr)

    if inputstr in ["1","2","3"]:
        pass
    elif inputstr == "0":
        print("跳出死循环")
        break
    else:
        print("输入错误 重新输入")
```

while True 就是死循环

break 就是不管什么循环直接跳出和 C 语言一样

```

inputstr=input("请选择执行程序:") #input返回用户
print("你输入的操作是 %s" %inputstr)

if inputstr in ["1","2","3"]:
    if inputstr == "1": #增加名片到数据库
        pass
    elif inputstr == "2": #在数据库查询所有名片
        pass
    elif inputstr == "3": #查询数据库指定的名片
        pass
    pass

```

修改 main.py 里面的功能，增加增删查改功能

名片管理系统框架搭建完成。

建立 datapro.py 文件来写增删查改数据处理函数

```

datapro.py x
def new_card(): #新增名片函数
    print("新增名片")

def show_all(): #显示所有名片
    print("显示所有名片")

def search_card(): #搜索指定名片
    print("搜索指定名片")

```

数据处理文件
datapro.py

主程序调用数据处理文件里面的函数

数据处理文件主要接受主程序文件传过来的数据

- 接受名称数据进行增删查改
- 接受电话数据进行增删查改
- 接受QQ数据进行增删查改

这就是在 datapro 文件里面实现的数据处理程序

然后 main.py 用 import 加在 datapro 文件就是了，类似 C 语言 include 功能

```

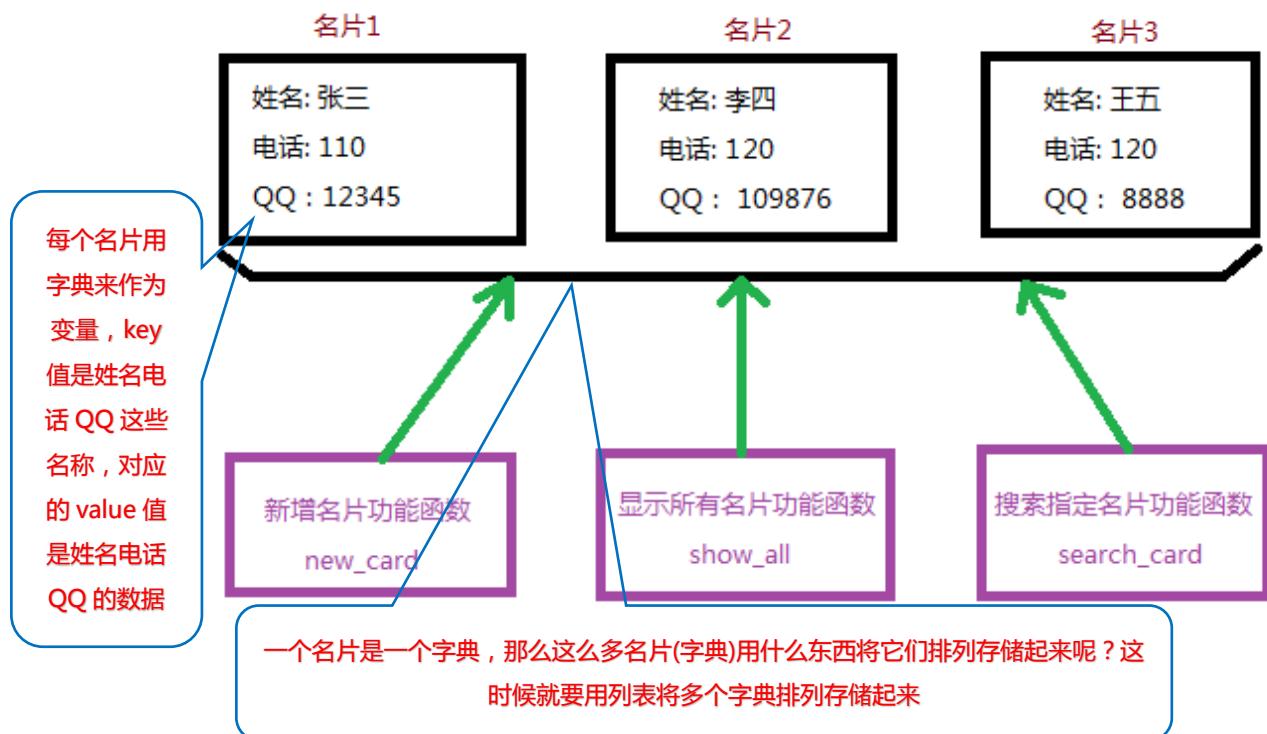
import datapro
# import 类似C语言的include,这样才能使用datapro文件里面的函数

while True:
    inputstr=input("请选择执行程序:") #input返回用户输入的字符串
    print("你输入的操作是 %s" %inputstr)

    if inputstr in ["1","2","3"]:
        if inputstr == "1": #增加名片到数据库
            datapro.new_card();
        elif inputstr == "2": #在数据库查询所有名片
            datapro.show_all();
        elif inputstr == "3": #查询数据库指定的名片
            datapro.search_card();

```

main.py 调用
datapro 里面的
函数，语法为
(文件名.函数名)



修改 datapro 里面的新增名片函数

```
datapro.py
card_list = [] # 定义个空列表来存放名片(字典)

def new_card(): # 新增名片函数
    print("新增名片")
    name = input("请输入姓名")
    phone = input("请输入电话")
    qq = input("请输入QQ")

    card_dict = {"name":name,
                 "phone":phone,
                 "qq":qq}

    card_list.append(card_dict)
    print(card_list)
```

列表用来存放多个名片的字典

得到姓名,电话号码,QQ 号

将姓名,电话号码,QQ 号这些变量转换成字典

用 append 将字典加入列表

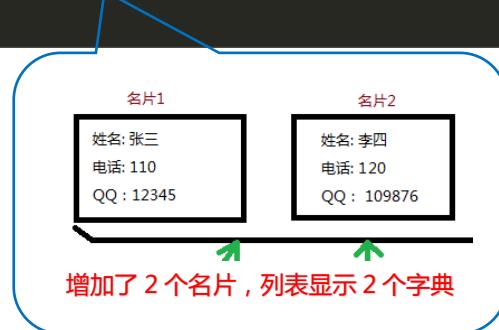
请选择执行程序: 1
你输入的操作是 1
新增名片
请输入姓名张三
请输入电话110
请输入QQ12345

[{'name': '张三', 'phone': '110', 'qq': '12345'}]
名片添加成功

增加了1个名片，列表显示一个字典

```
请选择执行程序: 1
你输入的操作是 1
新增名片
请输入姓名李四
请输入电话120
请输入QQ109876
[{'name': '张三', 'phone': '110', 'qq': '12345'}, {'name': '李四', 'phone': '120', 'qq': '109876'}]
名片添加成功
请选择执行程序:
```

名片增加功能完成



查询列表里面有多少项字典

```
def show_all(): #显示所有名片
    print("显示所有名片")
    for value in card_list:
        print(value)
```

列表新增了多少个名片，
就有多少项字典。那么
for 根据列表里面字典数
目进行循环，每循环 1
次，获取列表一项字典打
印出来

```
显示所有名片
请选择执行程序: 1
你输入的操作是 1
新增名片
请输入姓名张三
请输入电话120
请输入QQ12456
[{'name': '张三', 'phone': '120', 'qq': '12456'}]
名片添加成功
请选择执行程序: 1
你输入的操作是 1
新增名片
请输入姓名李四
请输入电话110
请输入QQ56789
[{'name': '张三', 'phone': '120', 'qq': '12456'}, {'name': '李四', 'phone': '110', 'qq': '56789'}]
名片添加成功
请选择执行程序: 2
你输入的操作是 2
显示所有名片
{'name': '张三', 'phone': '120', 'qq': '12456'}
{'name': '李四', 'phone': '110', 'qq': '56789'}
请选择执行程序:
```

新增两个名片

查询所有名片功能
打印 2 个名片

如果觉得这样打印不舒服，我们可以格式化打印

```
def show_all(): #显示所有名片
    print("显示所有名片")
    for value in card_list:
        print("姓名: %s , 电话 :%s , QQ:%s " %(value["name"],
                                                value["phone"],
                                                value["qq"]))
```

因为 value 获取 card_list 列表里面的元素是某一项字典，这
是 value 是字典变量，所以操作 value 就可以用字典方法

```
显示所有名片  
姓名：张三，电话：110，QQ：123456  
姓名：李四，电话：120，QQ：123456
```

在查询名片之前先确认列表里面有有没有名片

```
def show_all(): #显示所有名片  
    print("显示所有名片")  
    if len(card_list) == 0: #查询列表前先确认列表有没有名片  
        print("当前列表没有名片")  
        return #列表没有名片，下面代码就不会执行  
  
    for value in card_list:  
        print("姓名：%s，电话：%s，QQ：%s" % (value["name"],
```

增加名片查找功能

```
def search_card(): #搜索指定名片  
    print("搜索指定名片")  
    name = input("输入你要搜索的姓名")  
    for value in card_list:  
        if value["name"] == name:  
            print("找到了")  
            break  
    else:  
        print("没有找到 %s" % name)
```

循环完整个列表都没有找到就执行 else

```
请选择执行程序：3  
你输入的操作是 3  
搜索指定名片  
输入你要搜索的姓名张三  
没有找到 张三  
请选择执行程序：1  
你输入的操作是 1  
新增名片  
请输入姓名张三  
请输入电话120  
请输入QQ12345  
[{"name": "张三", "phone": "120", "qq": "12345"}]  
名片添加成功  
请选择执行程序：3  
你输入的操作是 3  
搜索指定名片  
输入你要搜索的姓名张三  
找到了
```



我们在原来的 datapro 功能基础上增加删除名片和修改名片

```
def deal_card():
    name = input("输入要删除的名片名字")
    for value in card_list:
        if value["name"] == name:
            card_list.remove(value)
```

将找到对应名字的字典放入列表，列表会去自动找到对应的字典项用 remove 删除

```
显示所有名片
姓名：李四，电话：110，QQ：56789
姓名：张三，电话：120，QQ：123456
请选择执行程序：4
你输入的操作是 4
输入要删除的名片名字李四
请选择执行程序：2
你输入的操作是 2
显示所有名片
姓名：张三，电话：120，QQ：123456
请选择执行程序：
```

下面我们修改指定名称的名片内容

```
def search_amend_card(name): #传入要修改的名片名字，进行修改
    for value in card_list:
        if value["name"] == name:
            print("找到了")
            value["name"] = input("你要修改成什么名字")
            value["phone"] = input("修改电话")
            value["qq"] = input("修改QQ号")
            break
    else:
        print("没有找到 %s" %name)

def amend():
    name = input("输入你要修改名片的名字")
    search_amend_card(name)
```

名片修改就是先循环列表里面的字典，根据输入的名字找到对应的字典，然后向找到的该项字典写数据

```
显示所有名片
姓名：张三，电话：110，QQ：12345
姓名：李四，电话：120，QQ：56789
请选择执行程序：5
你输入的操作是 5
输入你要修改名片的名字李四
找到了
你要修改成什么名字王二五
修改电话1111111111
修改QQ号5555555
请选择执行程序：2
你输入的操作是 2
显示所有名片
姓名：张三，电话：110，QQ：12345
姓名：王二五，电话：1111111111，QQ：5555555
请选择执行程序：
```

变量做引用的实际意义

```
1 a = 1
2 print("a 地址 = ", id(a)) #id函数就是获取某个变量地址
3 print("1 地址 = ", id(1))
4
```

↑ a 地址 = 8791345521488
↓ 1 地址 = 8791345521488

```
1 a = 1
2 print("a 地址 = ", id(a)) #id函数就是获取某个变量地址
3 print("1 地址 = ", id(1))
4
5 b = a
6 print("b 地址 = ", id(b))
```

我们发现变量 b 也是保存的变量 1 的地址，我觉得这个逻辑和 C 语言变量间赋值有点不一样，C 语言如果是 int b = a，意思就是 b 得到 a 的数值，但是 a 的地址和 b 的地址还是分开的、不是一个地址，但是 python 怎么就不是这样呢？

↑ a 地址 = 8791343489872
↓ 1 地址 = 8791343489872
b 地址 = 8791343489872

这就是 python 的一个特点



证明 python 的变量不占用内存空间，python 的变量只是一个数字地址的标牌而已。

```
1 a = 1
2 print("a 地址 = ", id(a)) # id函数就是获取某个变量地址
3 print("1 地址 = ", id(1))
4 b = a
5 print("b 地址 = ", id(b))
6
7 a = 20
8 print("a 地址 = ", id(a))
9 print("b 地址 = ", id(b))
```

我给 a 重新赋值了，相当于给了 a 一个新的地址。但是 b 的地址会受影响吗？

a 地址 = 8791374553936
1 地址 = 8791374553936
b 地址 = 8791374553936
a 地址 = 8791374554544
b 地址 = 8791374553936

明显 b 的地址不会受影响，除非你重新给 b 赋值

函数形参也是变量的引用

```
def test(num):
    print("num 地址 = ", id(num))

a = 10
print("a 地址 = ", id(a))

test(a)
```

1. a 的地址是 0x8791341524080

2. a 变量传给 num , num 的地址也是 0x8791341524080 和 a 的地址一样，证明了 python 的形参和 C++ 引用一样，是直接操作外部变量地址本身，而不是 C 语言那种只是个形参，只负责表面传值而已，而不去修改地址本身

a 地址 = 8791341524080
num 地址 = 8791341524080

```
1 def test():
2     result = "xxxxxxxx"
3     print("result 地址 = ", id(result))
4     return result
5
6 ret = test()
7 print("ret 地址 = ", id(ret))
8
```

test x
result 地址 = 31204776
ret 地址 = 31204776

函数返回值返回的也是一个地址，而且 python 子函数执行完后，如果有 return，那么内存栈空间不会释放变量地址，这点和 C/C++ 语言子函数不一样

Python 全局变量使用时和 C/C++ 语言全局变量不一样的地方

```
1 num = 10
2
3 def test1():
4     num = 30
5     print("test1 num = %d" % num)
6
7 def test2():
8     print("test2 num = %d" % num)
9
10 test1()
11 test2()
```

test x
test1 num = 30
test2 num = 10

第 1 个 test1 函数修改了全局变量值

运行之后发现 test1 确实是修改了 num 的值

但是 test2 的 num 值怎么没有被修改啊，num 全局变量不是在 test2 运行之前被 test1 修改了吗

这就是 python 和 C 语言全局变量不一样的地方，因为在 python 给全局变量赋值过程中，并不是在赋值，而是在函数内部又新定义了一个和全局变量名一样的局部变量。

```
num = 10
def test1():
    num = 30
    print("test1 num = %d" % num)
```

所以这里不是给全局变量赋值，而是新
定义了一个局部变量，这种定义变量方
法前面已经用过，其实就是 python 定
义局部变量的方法，没有问题

```
def test2():
    print("test2 num = %d" % num)
```

所以当 test1 执行完后，局部变量 num 就被子函数释放了，如
果把局部变量做成返回值，那么就可以保留局部变量的地址，
但是我这里没有做返回值，所以局部变量释放了

```
test1()
test2()
```

这时 test2 调用的是全局变量 num = 10 的值

```
1▶ num = 10
2def test1():
3    global num #将num指明为全局变量
4    num = 30
5    print("test1 num = %d" % num)
6
7def test2():
8    print("test2 num = %d" % num)

0test1()
1test2()
```

用 global 关键字就可以指定 num
为全局变量，而不是局部变量

```
test ×
test1 num = 30
test2 num = 30
```

根据输出结果，问题得到解决

函数返回值返回多个参数的方法

```
def mesg():
    temp = 39
    return temp

result = mesg()
print("函数返回值 = %d" %result)
```

函数返回值 = 39

这是常规的返回值方法，和 C 语言一样只能返回一个值，如果 C 语言函数要返回多个值，就得用 malloc 创建内存，用指针返回，这种很麻烦。

```
def mesg():
    temp = 39
    temp2 = 100
    return (temp,temp2)

result = mesg()
print("函数元组返回值 = ", result)
```

函数元组返回值 = (39, 100)

这就是 python 比 C 语言方便的地方，用元组函数可以返回很多参数。

```
def mesg():
    temp = 39
    temp2 = 100
    return temp,temp2

result = mesg()
print("函数元组返回值 = ", result)
```

函数元组返回值 = (39, 100)

用不用括号看自己习惯

如何获取函数返回的元组里面一堆数据的某个数据

```
def mesg():
    temp = 39
    temp2 = 100
    return temp, temp2

result = mesg()
print("函数元组返回值第1个数据temp = ", result[0])
print("函数元组返回值第2个数据temp2 = ", result[1])
```

用数组下标来获取元组里面某个参数是可以的

```
函数元组返回值第1个数据temp = 39
函数元组返回值第2个数据temp2 = 100
```

如果用数组下标获取元组里面的参数太麻烦，而且下标名称和元组每个元素的名称都不好对应，那么下面有个更简单的方法。

```
def mesg():
    temp = 39
    temp2 = 100
    return temp, temp2
```

你函数返回值返回多少个变量，而且有多少个不同的变量名

```
result1, result2 = mesg()
```

接受返回值可以定义和函数返回值数量相同名字一样的变量，用逗号隔开

```
print("函数元组返回值第1个数据temp = ", result1)
print("函数元组返回值第2个数据temp2 = ", result2)
```

```
函数元组返回值第1个数据temp = 39
```

```
函数元组返回值第2个数据temp2 = 100
```

输出结果也是一样的，而且接受返回值名称和函数返回值名称可以做成一样，方便记忆。

我这里用的 result1/2 表示，你可以改成 temp1,temp2，这种方法你必须知道函数要返回几个参数才行。

Python 中两个数字的交换

```
a = 6
b = 100
print("a = ", a)
print("b = ", b)

a, b = b, a
print("交换后 a = ", a)
print("交换后 b = ", b)
```

用元组的这种方式就可以很方便的进行两个变量值交换，不像 C 语言还要写交换函数

```
a = 6
b = 100
交换后 a = 100
交换后 b = 6
```

全局变量赋值给函数形参，会不会改变函数内部形参的值？

```
def demo(num):  
    num = 100  
    print("num = ", num)  
  
globalNum = 50  
demo(globalNum)  
print("globalNum = ", globalNum)
```

函数形参内
部赋值

外部传入全局参数给
函数形参

```
num = 100  
globalNum = 50
```

和 C 语言一样，如果函数形参在内部赋值了，外部全局变量无法修改函数内部形参

如果函数是列表形参呢？

```
def demo(num_list):  
    num_list = [1, 2, 3]  
    print("num = ", num_list)  
  
globalList = [40, 50, 60]  
demo(globalList)  
print("globalNum = ", globalList)
```

```
num = [1, 2, 3]  
globalNum = [40, 50, 60]
```

列表一样的，全局变量列表不会对已经赋值的函数内部列表有影响

但是有一种函数形参列表会受到影响

```
def demo(num_list):  
    num_list.append(100)  
    print("num = ", num_list)  
  
globalList = [40, 50, 60]  
demo(globalList)  
print("globalNum = ", globalList)
```

如果在函数内部不给列表形参赋值，
这样函数内部的列表就自动引用全局
列表传入进来的地址，所以就会修改
全局列表的值

```
num = [40, 50, 60, 100]  
globalNum = [40, 50, 60, 100]
```

所以 python 形参这个逻辑和 C 语言有点不同。

列表变量之间使用+=符号要注意

```
def demo(num):  
    num+=num #num = num+num  
    print("num = ", num)  
  
globalnum = 100  
demo(globalnum)  
print("globalNum = ", globalnum)  
  
num = 200  
globalNum = 100
```

输出正确

两个 num 相加然后赋值，不会影响全局变量的值上面讲过，这是我们 C 语言将两个变量相加的方法

然后我使用列表试试

```
def demo(num):  
    num+=num #num = num+num  
    print("num = ", num)  
  
globallist = [1, 2, 3]  
demo(globallist)  
print("globalNum = ", globallist)  
  
num = [1, 2, 3, 1, 2, 3]  
globalNum = [1, 2, 3, 1, 2, 3]
```

将列表传入进函数，
让列表变量两个相加

我们发现列表用+=运算并不像数字运算那么简单，列表的+=运算就是列表的 extend 函数使用，将在列表后面增加当前列表里面的数据。而且还会影传入形参的全局列表。

函数形参不赋值的情况下使用(所谓的缺省值)，语句 if not ... 使用

```
def demo(name, flag):  
    value = "男人"  
    if not flag: #如果flag不为真，没有传入参数就是不为真  
        value = "女人"  
    print("%s 是 %s" %(name, value))  
  
demo("小明", True)
```

小明 是 男人

因为 flag 传入的是 True 所以 flag 位真，if not 里面的语句不执行

```
def demo(name, flag):  
    value = "男人"  
    if not flag: #如果flag不为真, 没有传入参数就是不为真  
        value = "女人"  
    print("%s 是 %s" %(name, value))
```

demo("小明", False)

小明 是 女人
flag 传入的是 False , 所以 if not....被执行

以上范例没说明问题啊, 下面来说明问题

```
def demo(name, flag):  
    value = "男人"  
    if not flag: #如果flag不为真, 没有传入参数就是不为真  
        value = "女人"  
    print("%s 是 %s" %(name, value))
```

demo("小明") 因为是真假判断, 我不想传入第 2 个形参

```
Traceback (most recent call last):  
  File "D:/02_我的文档/Desktop/busim_card/test.py", line 7, in <module>  
    demo("小明")  
TypeError: demo() missing 1 required positional argument: 'flag'
```

编译报错, 因为你的函数要求必须写入两个形参, 如果你想不写形参默认 flag 为真怎么办?

```
def demo(name, flag=True):  
    value = "男人"  
    if not flag: #如果flag不为真, 没有传入参数就是不为真  
        value = "女人"  
    print("%s 是 %s" %(name, value))
```

那你就在函数定义的时候先填好一个 flag 的值, 这个值就是所谓的缺省值

demo("小明")

小明 是 男人

你看编译通过成功运行了, 函数默认执行 True , 如果你给函数传入 False , 那么函数会自动将默认的 flag=True 改成 flag=False

```
def demo(name, flag=True):  
    value = "男人"  
    if not flag: #如果flag不为真, 没有传入参数就是不为真  
        value = "女人"  
    print("%s 是 %s" %(name, value))
```

demo("小明", False) 传入 False

小明 是 女人

注意缺省形参一定在函数形参的末尾，不准在函数形参的中间

```
def demo(name, flag=True, blue):  
    blue = 10  
    value = "男人"  
    if not flag: #如果flag不为真，没有传入参数就是不为真
```

```
File "D:/02 我的文档/Desktop/busim_card/test.py", line 1  
    def demo(name, flag=True, blue):  
    ^  
SyntaxError: non-default argument follows default argument
```

这样在缺省值后面定义新形参是编译报错的，一定要让缺省值形参是整个函数最后一个形参

```
▶ def demo(name, blue, flag=True):  
    blue = 10  
    value = "男人"
```

小明 是 男人

编译通过正常运行

还有个问题，就是多形参数传参问题

```
def demo(name, blue, flag=True):  
    value = "男人"  
    if not flag: #如果flag不为真，没有传入参数就是不为真  
        value = "女人"  
    print("%s 是 %s" % (name, value))
```

demo("小明")

```
Traceback (most recent call last):  
  File "D:/02 我的文档/Desktop/busim_card/test.py", line 8, in <module>  
    demo("小明")  
TypeError: demo() missing 1 required positional argument: 'blue'
```

但是我现在就想传一个参数怎么办？

```
def demo(name, blue="", flag=True):  
    value = "男人"  
    if not flag: #如果flag不为真，没有传入参数就是不为真  
        value = "女人"  
    print("%s 是 %s" % (name, value))  
    print("blue = %s" % blue)  
demo("小明")
```

我们给形参一个默认参数，不一定给缺省值，我给个空值

```
小明 是 男人
```

```
blue =
```

编译输出正常了，blue 没有得到参数，所以输出空

下面我们传第 2 个参数试试

```
demo ("小明", False)
```

我传入一个 False

```
小明 是 男人
```

```
blue = False
```

我传第 2 个参数 False 并不是传给缺省值，而是传给第 2 个形参 blue

如果在函数多形参的情况下，我想第 2 个传入的参数就是缺省值怎么办？

```
def demo (name, blue="", flag=True) :
```

```
    value = "男人"
```

```
    if not flag: #如果flag不为真，没有传入参数就是不为真
```

```
        value = "女人"
```

```
    print ("%s 是 %s" %(name, value))
```

```
    print ("blue = %s" %blue)
```

```
demo ("小明", flag=False)
```

我们在传参数给函数的时候，指定缺省值编号就是了，这里缺省值形参是 flag

```
小明 是 女人
```

```
blue =
```

这样就可以了，你也不用每个形参都去写一遍

多值参数形参传递，形参变量前 加* 或者 加**

```
def demo (name, *yuanzu, **keyword) :  
    print (name)  
    print (yuanzu)  
    print (keyword)  
  
    demo (1)
```

```
1  
( )  

```

根据输出很明显，

name 接受的是值

*变量(*yuanzu)接受元组

变量(keyword)接受字典

所以在 python 中

*变量：表示变量接受元组

**变量：表示变量接受字典

其实和 C 语言指针有点像



Python 字典也就是 C 语言的二维数组，1 个元素表示 key 值，key 对应一个元素表示数值

```
def demo(name, *yuanzu, **keyword):
    print(name)
    print(yuanzu)
    print(keyword)
```

```
demo(1,[2, 3, 4, 5],ming="小明")
```

传入值
传入元组
传入键值对，ming 是键，小明是值，(记住键不能和某个形参同名)

```
1
(2, 3, 4, 5)
{'ming': '小明'}
```

Python 的单继承

Python 和 C++, JAVA 一样有类继承功能

```
class peple:
    def eat(self):
        print("人吃东西")
    def drink(self):
        print("人喝东西")
    def fight(self):
        print("人打架")
pe = peple()
pe.eat()
pe.drink()
pe.fight()
```

实现人的类

```
class bird:
    def eat(self):
        print("鸟吃东西")
    def drink(self):
        print("鸟喝东西")
    def fight(self):
        print("鸟打架")
    def fly(self):
        print("鸟儿会飞")
pe = bird()
pe.eat()
pe.drink()
pe.fight()
pe.fly()
```

实现鸟的类

```
人吃东西
人喝东西
人打架
```

```
鸟吃东西
鸟喝东西
鸟打架
鸟儿会飞
```

我们发现鸟类就比人类多了一个会飞的执行函数，鸟类的其它函数功能和人类是一样的，然后我在鸟类也重复写人类吃喝打架三个功能，是不是写得太重复了。

所以我用继承来解决这个问题，和 C++, JAVA 思路一样

```
class peple:  
    def eat(self):  
        print("鸟人吃东西")  
    def drink(self):  
        print("鸟人喝东西")  
    def fight(self):  
        print("鸟人打架")  
  
class bird(peple):  
    def fly(self):  
        print("鸟儿会飞")
```

```
鸟人吃东西  
鸟人喝东西  
鸟人打架  
-----  
鸟人吃东西  
鸟人喝东西  
鸟人打架  
鸟儿会飞
```

```
pe = peple() #人类  
pe.eat()  
pe.drink()  
pe.fight()  
print("-----")  
bir = bird() #鸟类可以使用人类里面的函数  
bir.eat()  
bir.drink()  
bir.fight()  
bir.fly()
```

在类括号里面写上继承的父类类名就行，这样 bird 就成了子类

然后创建人类对象调用人类里的函数

创建鸟类也就是子类对象，也可以使用人类的函数

执行效果就达到了，让你省略了相同的代码。用 C 语言理解就是新创建的类包含了以前类的头文件

子类修改父类的函数叫做方法(函数)重写

```
class peple:  
    def eat(self):  
        print("鸟人吃东西")  
    def drink(self):  
        print("鸟人喝东西")  
    def fight(self):  
        print("鸟人打架")
```

```
class bird(peple):  
    def fly(self):  
        print("鸟儿会飞")
```

```
bir = bird()  
bir.eat()
```

我们现在子类调用的父类函数，eat()里面实现的内容太少，我想 eat()多实现点东西

鸟人吃东西

```

class peple:
    def eat(self):
        print("鸟人吃东西")
    def drink(self):
        print("鸟人喝东西")
    def fight(self):
        print("鸟人打架")

class bird(peple):
    def fly(self):
        print("鸟儿会飞")
    def eat(self):
        print("鸟人吃吃吃东西")
        print("鸟人对打")

```

让子类声明和父类相同的函数名，然后在子类重写这个函数名的内容，那么子类对象就会调用子类的 eat 函数

鸟人吃吃吃东西
鸟人对打

这样就是实现了函数重写，但是父类 eat 函数的内容就无法使用了，只能使用子类 eat 的内容，除非你用父类创建个对象，这个对象就只属于父类，那么就可以使用父类的 eat 函数

那么问题来了？我父类 eat() 函数实现的内容少了，我子类实现个函数和 eat 名称不一样但是内容很多的函数，不是就解决了吗。对！！这样是可以解决，但是在那种程序架构写好了的地方，比如 C++ 那种虚函数，在没有实现虚函数里面内容的时候，就有其它人调用这个虚函数名，占个坑，到时候等你去完善这个虚函数。这时候就不能在子类里随便定义个函数名来弥补父类函数的不足，必须用函数重写的方法来解决。

子类重写了父类同名的函数，导致子类对象无法再使用父类同名函数的内容，但是我又想子类使用重写函数的同时也调用父类同名函数的内容，那么只有用 super() 来解决了

```

class peple:
    def eat(self):
        print("鸟人吃东西")
    def drink(self):
        print("鸟人喝东西")
    def fight(self):
        print("鸟人打架")

class bird(peple):
    def fly(self):
        print("鸟儿会飞")
    def eat(self):
        print("鸟人吃吃吃东西-----")
        super().eat()
        print("鸟人对打----")

```

super() 关键字指定的执行函数就是父类的函数

```
bir = bird()  
bir.eat()
```

鸟人吃吃吃吃东西-----
鸟人吃东西
鸟人对打----

这句就是父类的函数

这样就可以让子类重写的函数使用父类的函数，但是同时也会执行子类重写函数的内容

```
class peple:  
    def eat(self):  
        print("鸟人吃东西")  
    def drink(self):  
        print("鸟人喝东西")  
    def fight(self):  
        print("鸟人打架")
```

```
class bird(peple):  
    def fly(self):  
        print("鸟儿会飞")  
    def eat(self):  
        print("鸟人吃吃吃吃东西-----")  
        peple.eat(self)  
        print("鸟人对打----")
```

```
bir = bird()  
bir.eat()
```

鸟人吃吃吃吃东西-----
鸟人吃东西
鸟人对打----

执行结果和上页一样

但是如果用子类名称去调用父类同名函数呢？

```
class bird(peple):  
    def fly(self):  
        print("鸟儿会飞")  
    def eat(self):  
        print("鸟人吃吃吃吃东西-----")  
        bird.eat(self)  
        print("鸟人对打----")  
  
bir = bird()  
bir.eat()
```

用子类名去调用父类同名函数，
会出现递归报错

```
File "D:/02 我的文档/Desktop/busim_card/test.py", line 13, in eat
    bird.eat(self)
```

你看报错了，所以子类不能去调用父类和子类的同名函数

Python 多继承

```
class A:
    def AAA(self):
        print("AAAA CLASS")

class B:
    def BBB(self):
        print("BBBB CLASS")

class C(A, B):
    def CCC(self):
        print("CCCC CLASS")

Cobj = C()
Cobj.AAA()
Cobj.BBB()
Cobj.CCC()
```

Python 的 C 类用括号表示继承了 A 类和 B 类

这样 C 类的对象也可以调用 A
类，B 类的函数，和 C++一样

```
AAAA CLASS
BBBB CLASS
CCCC CLASS
```

执行结果正确

多继承注意事项

```
class A:
    def AAA(self):
        print("AAAA CLASS")
    def BBB(self):
        print("A-BBBB CLASS")

class B:
    def BBB(self):
        print("BBBB CLASS")

class C(A, B):
    def CCC(self):
        print("CCCC CLASS")

Cobj = C()
Cobj.AAA()
Cobj.BBB()
Cobj.CCC()
```

如果我让 A 类也有和 B 类一样的函
数 BBB(self)，那么对象到底执行那
个类里面的 BBB 函数呢？

```
AAAA CLASS
A-BBBB CLASS
CCCC CLASS
```

很显然对象执行 A
类里面的同名函数
BBB(self)

这就是多继承同名函数的问题

```
class C(B, A):  
    def CCC(self):  
        print("CCCC CLASS")
```

如果我让子类继承方式
修改为 B 类继承在前，
A 类继承在后

AAAA CLASS
BBBB CLASS
CCCC CLASS

你看 AB 类同名函数就先执行 B 类里面
的 BBB(self) 函数了

所以这是根据子类继承父类的过程中，括号里面哪个类最靠前那个类的同名函数会被先执行
所以 C++ 要求最好使用单继承不要使用多继承，JAVA 就直接取消了多继承，只使用单继承

Python 多态使用

多态就是不同的子类对象，调用相同名字的父类函数，运行的结果不一样

```
class Dog(object):  
    def __init__(self, name): # 在类创建对象的时候执行构造函数  
        self.name = name  
  
    def game(self):  
        print("执行 %s 狗类里面的函数" % self.name)  
  
class cat(Dog): # 猫类继承了狗类  
    def game(self):  
        print("执行猫类 %s 的 game 函数" % self.name)  
  
class person(object):  
    def __init__(self, name): # 在类创建对象的时候执行构造函数  
        self.name = name  
    def game_dog(self, Dog): # 可以传入 dog 类定义的对象  
        print("%s 和 %s" % (self.name, Dog.name))  
        Dog.game()  
  
wangcai = Dog("旺财")  
xiaoming = person("小明")  
xiaoming.game_dog(wangcai)
```

2. 那么会去执行 Dog
类里面的函数

1. 正常情况下，我传入 xiaoming 对象
的是 Dog 类，

小明 和 旺财
执行 旺财 狗类里面的函数

```

class Dog(object):
    def __init__(self, name): #在类创建对象的时候执行构造函数
        self.name = name

    def game(self):
        print("执行 %s 狗类里面的函数" %self.name)

class cat(Dog): #猫类继承了狗类
    def game(self):
        print("执行猫类 %s 的game函数" %self.name)

class person(object):
    def __init__(self, name): #在类创建对象的时候执行构造函数
        self.name = name

    def game_dog(self, Dog):
        print("%s 和 %s" %(self.name, Dog.name))
        Dog.game()

maomi = cat("小猫咪")
xiaoming = person("小明")
xiaoming.game_dog(maomi)

```

1. 创建的猫类的对象

2. 但是 xiaoming 对象的函数 game_dog 接受的是 cat 类的对象，但是我的形参是 Dog 类啊

这就是多态的好处了，你虽然使用的 game_dog 函数是 Dog 形参，但是你主要传入了猫类对象给 game_dog，同时猫类必须继承了 Dog 类对象，那么根据传入对象的性质不同，比如我这是猫类对象。我会去执行猫类里面和 Dog 类的同名函数 game

```

class Dog(object):
    def __init__(self, name): #在类创建对象的时候执行构造函数
        self.name = name

    def game(self):
        print("执行 %s 狗类里面的函数" %self.name)

class cat(Dog): #猫类继承了狗类
    def game(self):
        print("执行猫类 %s 的game函数" %self.name)

class person(object):
    def __init__(self, name): #在类创建对象的时候执行构造函数
        self.name = name

    def game_dog(self, Dog):
        print("%s 和 %s" %(self.name, Dog.name))
        Dog.game()

maomi = cat("小猫咪")
xiaoming = person("小明")
xiaoming.game_dog(maomi)

```

小明 和 小猫咪
执行猫类 小猫咪 的game函数

这就是多态，只要子类继承了父类，然后子类和父类有同名函数，那么就根据传入的对象属于哪个类，来决定执行哪个类的同名函数。

类属性使用，就是让类里面指定变量永久存放在内存中，用来做计时等应用



所以类也是一个对象，我们有些情况下可以直接操作类里面的函数，而不用非要创建类对象来操作类里面的函数。

```
class tool(object):
    count = 0
    def __init__(self):
        tool.count += 1
t1 = tool()
t2 = tool()
t3 = tool()
print("对象创建了 %d 个" % tool.count)
```

对对象创建了 3 个

执行没有问题，很多人用类属性方法来做单例模式，JAVA 也有这种类属性功能

这就是类属性的用法，直接在类中定义变量，

在构造函数中计算类属性 count 变量，这样你创建几个对象就会执行几次构造函数，count 就会计算几次

有个疑问，如果对象释放了，类属性 count 变量是不是会被清 0 呢？
结果是 count 不会被清 0，因为类属性是在类对象的单独一块内存区的，比如上图中的 class A，它自己有一块内存区域，和创建的对象内存区域是分开的

Python 除了类属性，还有类函数，就是类自己调用的函数

```
class tool(object):
    count = 0
    @classmethod
    def show_tool_count(cls):
        print("tool类函数执行 count数量为 %d" % cls.count)
    def __init__(self):
        tool.count += 1
t1 = tool()
t2 = tool()
t3 = tool()
tool.show_tool_count()
```

类函数(方法)和类属性一样，只有类名可以调用，定义类函数要加@classmethod 关键字

然后任意取类函数名，但是形参必须是 `cls`，因为 `cls` 是本类名的引用

如果类函数向使用本类属性的变量，比如 `count`，就要用 `cls` 来指定

类函数调用直接用类名指定就行了

tool类函数执行 count数量为 3

类静态函数(也就是静态方法)

```
class tool(object):
    count = 0

    @classmethod
    def show_tool_count(cls):
        print("tool类函数执行 count数量为 %d" %cls.count)

    def __init__(self):
        tool.count +=1

    @staticmethod
    def run():
        print("静态方法执行")
```

在函数前面加@staticmethod 就表示下面函数是静态方法

静态方法(函数)不能执行类里面的变量，比如上面的count，也不能执行对象的变量，所以形参没有self和cls，也不能执行类里面的函数，只能执行外部导入的函数

```
t1 = tool()
t2 = tool()
t3 = tool()

tool.show_tool_count()
tool.run()
```

用类名执行静态方法

```
tool类函数执行 count数量为 3
静态方法执行
```

设计模式中的单例模式

```
class musicplay(object):

    def __init__(self):
        print("播放器初始化")

    player1 = musicplay()
    player2 = musicplay()

    print(player1)
    print(player2)
```

我们平时创建的普通类，这种类创建一个对象就分配一个内存空间，再创建个对象就又分配个新的内存空间

所以每个对象地址不一样

```
test ×
播放器初始化
播放器初始化
<__main__.musicplay object at 0x00000000001DD25F8>
<__main__.musicplay object at 0x00000000001DD7D68>
```

单例模式就是要求一个类，只创建一个对象，这个对象是唯一的，如果再用这个类创建第 2 个对象，就让其第 2 个对象创建失败，这样这个类的对象就是在系统中唯一的。

_new_方法使用

```
class musicplay(object):  
  
    def __new__(cls, *args, **kwargs):  
        print("创建new方法对象，分配内存空间")  
    def __init__(self):  
        print("播放器初始化")  
  
player1 = musicplay()  
player2 = musicplay()  
  
print(player1)  
print(player2)
```

_new_方法就是在用类名创建对象时，python 解释器会先执行__new_里面的函数为对象分配空间

但是为什么打印对象的地址是 None 呢？不是__new_分配了空间吗？

创建new方法对象，分配内存空间
创建new方法对象，分配内存空间
None
None

这是因为__new_执行后分配了内存空间，但是分配的空间地址还必须执行其它函数返回给 python 解释器

```
class musicplay(object):  
  
    def __new__(cls, *args, **kwargs):  
        print("创建new方法对象，分配内存空间")  
        return super().__new__(cls)  
  
    def __init__(self):  
        print("播放器初始化")  
  
player1 = musicplay()  
player2 = musicplay()  
  
print(player1)  
print(player2)
```

用父类 object 的__new_来获取子类 musicplay 对象分配的内存空间，返回给 python 解释器

Python 解释器获得最先分配内存地址的引用之后，会自动把地址赋值给__init_的第 1 个参数 self

这样分配的内存就不是 None 了

```
创建new方法对象，分配内存空间  
播放器初始化  
创建new方法对象，分配内存空间  
播放器初始化  
<__main__.musicplay object at 0x0000000001DE76D8>  
<__main__.musicplay object at 0x0000000001E1DA90>
```

这种方法和我以前直接定义对象有什么区别呢？我还要去执行 super().__new__(cls)一下，好麻烦

下面的单例模式就会让 super().__new__(cls)这种做法有用

```
class musicplay(object):  
  
    instance = None # 创建一个类属性变量，设置初始化为None  
  
    def __new__(cls, *args, **kwargs):  
        #1. 判断instance变量是不是None  
        if cls.instance is None: # 判断instance被分配内存空间没有，如果没有执行if语句  
            print("调用父类方法为第1个对象分配内存空间")  
            cls.instance = super().__new__(cls) # 将分配的内存空间给instance  
  
    return cls.instance  
  
def __init__(self):  
    print("播放器初始化")
```

这个就是用来
记录这个类创
建过对象没有

将第1次创建的内存地址返回给__init__

```
17     player1 = musicplay()  
18     player2 = musicplay()  
19  
20     print(player1)  
21     print(player2)
```

所以两个对象地址是一样的，这就是单例模
式，所以没必要创建第2个对象

JAVA 也是这种设计方法，用类属性变量来实现单例模式，上一章节我也提到过。

Python 异常

```
▶ num = int(input("输入一个数")) # 这种在input前面加int就表示必须输入整数  
  
test x  
输入一个数10  
Process finished with exit code 0
```

输入整数是没有问题的

```
▶ num = int(input("输入一个数")) # 这种在input前面加int就表示必须输入整数  
  
test x  
输入一个数a  
Traceback (most recent call last):  
  File "D:/02 我的文档/Desktop/busim_card/test.py", line 1, in <module>  
    num = int(input("输入一个数")) # 这种在input前面加int就表示必须输入整数  
ValueError: invalid literal for int() with base 10: 'a'
```

因为我 input 强制转换的是
int，就是接受数据为 int，
但是你输入字符就会报错

下面我们用异常来捕获错误

```
try:  
    num = int(input("输入一个数"))  
except:  
    print("请输入正确的整数")  
print("-----")
```

加入 try 异常关键字，如果 try 里面的程序执行出错，不会让程序停止执行，直接报错。

而是执行 except 里面的程序，然后还有继续执行下面的程序

```
test  
输入一个数 10  
-----
```

没有错误，不执行 except 里面的程序，直接执行下面的程序

```
输入一个数 a  
请输入正确的整数  
-----
```

如果输入错误，也不会停止程序运行，会先执行 except 的程序，再向下执行

这种异常机制就类似 JAVA 和 C++，可以让程序出现错误的时候继续向下执行，但是会打印出异常让开发者修改，不会停止用户使用程序。然后 C 语言就比较严格，有异常就执行报错，停止执行程序。至于要不要异常这种机制，看使用场景。

except 关键字里面有很多错误类型选项

ZeroDivisionError 就是除 0 错误选项

```
try:  
    num = int(input("输入一个数")) #这种在input前面加int就表示必须输入  
    result = 8/num  
    print("result = %d" %result)  
  
except ZeroDivisionError:  
    print("除0错误")  
  
print("-----")
```

如果输入的是非 0 数字，那么能被 8 整除正常执行

如果输入的是 0，那么被 8 整除的 0 结果还是 0，没有意义，就会执行 ZeroDivisionError 这个关键字里面的异常

```
输入一个数 5  
result = 1  
-----
```

除以的是整数，正常执行

```
输入一个数 0  
除0错误  
-----
```

除 0 没有意义报告异常

```
try:  
    num = int(input("输入一个数")) #  
    result = 8/num  
    print("result = %d" %result)  
  
except ZeroDivisionError:  
    print("除0错误")  
  
print("-----")
```

和上面一样的程序，已经有异常处理了

```

输入一个数 a
Traceback (most recent call last):
  File "D:/02 我的文档/Desktop/busim card/test.py", line 2, in <module>
    num = int(input("输入一个数")) #这种在input前面加int就表示必须输入整数
ValueError: invalid literal for int() with base 10: 'a'

```

这是因为输入 a ,是输入值错误异常 ,而不是运算除 0 异常 ,所以你必须要增加 except 关键字 ,执行 ValueError 异常

ValueError 输入字符错误异常选项

```

try:
    num = int(input("输入一个数")) #这种在input前面加int就表示必须输入整数
    result = 8/num
    print("result = %d" %result)

except ZeroDivisionError:
    print("除0错误")
except ValueError:
    print("输入字符错误")

print("-----")

```

输入一个数 5 ----- 输入正确	输入一个数 0 除0错误 ----- 运算错误	输入一个数 a 输入字符错误 ----- 输入字符错误
--------------------------	----------------------------------	--------------------------------------

如果程序执行的错误不是 ZeroDivisionError 错误 ,也不是 ValueError 错误 ,可能是其它错误 ,但是我记不住那种错误选项 ,怎么办呢 ?

下面用 Exception 来捕获所有错误打印出来

```

try:
    num = int(input("输入一个数")) #这
    result = 8/num
    print("result = %d" %result)

except ZeroDivisionError:
    print("除0错误")
except ValueError:
    print("输入字符错误")
except Exception as ret:
    print("未知错误 %s" %ret)

print("-----")

```

Exception 就是抛出除 ZeroDivisionError 和 ValueError 之外的其它错误 然后将错误赋值给 ret 变量 ,打印出来方便确认

但是我记不住那么多选项所以就用 ZeroDivisionError 选项来模拟错误，请将 ZeroDivisionError 和 ValueError 取消掉

```
try:  
    num = int(input("输入一个数")) #  
    result = 8/num  
    print("result = %d" %result)  
  
except Exception as ret:  
    print("未知错误 %s" %ret)  
  
print("-----")
```

正常输入输出
运算错误
输入字符错误

异常处理处理 try 和 except 关键字，还有 else 和 finally

```
try:  
    num = int(input("输入一个数"))  
    result = 8/num  
    print("result = %d" %result)  
  
except Exception as ret:  
    print("未知错误 %s" %ret)  
else:  
    print("运算成功")  
finally:  
    print("有无异常都会执行")  
print("-----")
```

如果 try 里面的程序执行错误，就不会执行 else 里的程序，如果 try 里面程序执行正确就会执行 else 里面的程序

```

输入一个数 5
result = 1
运算成功
有无异常都会执行

```

try 里面程序正确，执行 else 程序


```

输入一个数 0
未知错误 division by zero
有无异常都会执行
-----
```

try 里面执行错误报出异常，不执行 else 的程序，但是 finally 无论有无异常都会执行


```

输入一个数 a
未知错误 invalid literal for int() with base 10: 'a'
有无异常都会执行
-----
```

try 里面执行错误报出异常，不执行 else 的程序，但是 finally 无论有无异常都会执行

这就是 try , except , else , finally 关键字完整使用

python 异常抛出与传递

python 和 JAVA 一样有异常抛出机制

```

def demo1():
    return int(input("输入整数:"))

print(demo1())

```

这是常规调用程序的方法，返回值
返回输入值

```

test %
输入整数: 1
1
```

输入数字 1 没有问题

```

test %
输入整数:A
Traceback (most recent call last):
  File "D:/02 我的文档/Desktop/busim_card/test.py", line 4, in <module>
    print(demo1())
  File "D:/02 我的文档/Desktop/busim_card/test.py", line 2, in demo1
    return int(input("输入整数:"))
ValueError: invalid literal for int() with base 10: 'A'
```

输入 a 字符报错很正常

但是我们发现第 2 行 return 执行报错的异常向上传递给调用它的函数

为了让调用的函数不会因为异常而终止程序，我们可以在调用函数的主程序创建 try 异常处理机制。

```

def demo1():
    return int(input("输入整数:"))

try:
    print(demo1())
except Exception as ret:
    print("异常类型 %s" %ret)

print("-----")

```

```

输入整数:A
异常类型 invalid literal for int() with base 10: 'A'
-----
```

问题得到解决，因为本函数异常如果不处理，异常有向上传递也就是向调用函数传递的特性，所以我在调用函数的主程序去处理子程序的异常也是可以的，这样主程序可以继续向下执行

主动抛出异常机制 raise 实际应用

- 提示用户 输入密码，如果 长度少于 8，抛出 异常



我们练习下这个程序

```

def input_passwd():
    pwd = input("请输入密码")
    if len(pwd) >= 8: #len函数用来判断用户输入密码长度
        return pwd

    ex = Exception("密码长度不够")
    raise ex

input_passwd()

```

用 raise 关键字将 ex 变量抛出给上一层调用它的函数

如果 if 没有执行，就不会 return 跳出函数，那么下面的 Exception 就会执行，Exception 函数会把自定义的异常错误字符串传递给 ex 变量

```
请输入密码12345678
```

```
Process finished with exit code 0
```

执行正确

```
test.py  
请输入密码12345  
Traceback (most recent call last):  
  File "D:/02 我的文档/Desktop/busim_card/test.py", line 9, in <module>  
    input_passwd()  
  File "D:/02 我的文档/Desktop/busim_card/test.py", line 7, in input_passwd  
    raise ex  
Exception: 密码长度不够
```

输入字符不够，抛出自己定义的异常

如果你不想程序中断执行，就和上一节一样，在主程序里面创建 try

```
def input_passwd():  
    pwd = input("请输入密码")  
    if len(pwd) >= 8: #len函数用来判断用户输入密码长度  
        return pwd  
  
    ex = Exception("密码长度不够")  
    raise ex  
  
try:  
    input_passwd()  
except Exception as ret:  
    print("异常问题是 %s" %ret)  
  
print("-----")
```

```
请输入密码12345  
异常问题是 密码长度不够
```

import 关键字使用，就是导入其它 py 文件

import 就是在本 py 文件中倒入其它文件，然后可以在本文件使用其它 py 文件的函数，全局变量和类，适合模块化开发。

```
test1.py  
1 title1 = "模块1"  
2  
3 def test1():  
4     print("执行test1子函数")  
5  
6 class classt1(object):  
7     def classt1(self):  
9         print("class t1子类操作")
```

test1.py 文件

```
test2.py  
1 title2 = "模块2"  
2  
3 def test2():  
4     print("执行test2子函数")  
5  
6 class classt2(object):  
7     def classt2(self):  
9         print("class t2子类操作")
```

test2.py 文件

```
test.py  
1 import test1 #import就是导入其它py文件  
2 import test2  
  
3 xt1 = test1.title1 #获取test1文件里面的全局变量值  
4 print("test1 py文件里面的全局变量 = %s" %xt1)  
5 test1.test1() #执行test1文件里面的函数  
  
6 clst1 = test1.classt1() #执行test1文件里面的类创建对象  
7 clst1.classt1()
```

主程序用 import 导入同一个目录
下的其它 py 文件

可以使用 test1.py 里面的函数了

```
test1.py文件里面的全局变量 = 模块1  
执行test1子函数  
cl ass t1子类操作
```

这里打印出来的都是 test1.py 文件里面的内容

使用 test2 里面的内容

```
test.py  
import test1 #import就是导入其它py文件  
import test2  
  
xt1 = test1.title1 #获取test1文件里面的全局变量值  
print("test1 py文件里面的全局变量 = %s" %xt1)  
test1.test1()#执行test1文件里面的函数  
  
clst1 = test1.classt1() #执行test1文件里面的类创建对象  
clst1.classt1()  
  
xt2 = test2.title2 #获取test1文件里面的全局变量值  
print("test2 py文件里面的全局变量 = %s" %xt2)  
test2.test2()#执行test1文件里面的函数  
  
clst2 = test2.classt2() #执行test1文件里面的类创建对象  
clst2.classt2()
```

```
test1 py文件里面的全局变量 = 模块1  
执行test1子函数  
cl ass t1子类操作  
test2 py文件里面的全局变量 = 模块2  
执行test2子函数  
cl ass t2子类操作
```

这是 test1 和 test2 文件里面的内容都被主程序使用了。

如果 import 导入的 py 文件名太长，就用 as 来给导入的 py 文件取别名替代

```
import test1 as ts1  
import test2 as ts2  
  
xt1 = ts1.title1 #获取test1文件里面的全局变量值  
print("test1 py文件里面的全局变量 = %s" %xt1)  
ts1.test1()#执行test1文件里面的函数  
clst1 = ts1.classt1() #执行test1文件里面的类创建对象  
clst1.classt1()  
  
xt2 = ts2.title2 #获取test1文件里面的全局变量值  
print("test2 py文件里面的全局变量 = %s" %xt2)  
ts2.test2()#执行test1文件里面的函数  
clst2 = ts2.classt2() #执行test1文件里面的类创建对象  
clst2.classt2()
```

如果调用的 py 文件名太长，那么下面调用函数前就要敲很长一段 py 文件名，这里用 as 给导入的 py 文件取个短名来替代

这样就使用短名来替代长文件名的 py 文件，提高编程效率

```
test1 py文件里面的全局变量 = 模块1  
执行test1子函数  
cl ass t1子类操作  
test2 py文件里面的全局变量 = 模块2  
执行test2子函数  
cl ass t2子类操作
```

执行结果是一样的

这就是 as 在 import 里面的使用方法，把长文件名修改为短名来替代，提高编程效率

from 文件名 import 内容名.... 这种使用方法

```
from test1 import test1  
from test2 import test2
```

这个意思就是导入 test1.py 文件里的
test1 函数，test2 也是如此

```
test1()  
test2()
```

这样主程序直接调用函数名就是了，不用去
先指定 py 文件名.函数

执行test1子函数
执行test2子函数

```
from test1 import classt1  
from test2 import classt2
```

调用指定的类也是
这样操作

```
cls1 = classt1()  
cls2 = classt2()  
cls1.classt1()  
cls2.classt2()
```

cl ass t1子类操作
cl ass t2子类操作

这样写有什么意义呢？虽然调用函数和类方便了，但是你不知道这些函数和类来自于哪个文件。

import 是将 py 文件所有内容都导入进主程序文件，用文件名.文件内容的方式执行

from...import 是指定 py 文件里面某一段内容导入，直接执行函数名，变量名，类名就可以

from...import * 也是导入模块 py 文件的所有内容

```
from test1 import *  
from test2 import *  
  
cls1 = classt1()  
cls2 = classt2()  
cls1.classt1()  
cls2.classt2()
```

from test1 import * 和 import test1 的区别是
import test1 会在调用 py 文件里面的内容时，在
内容前面加文件名.内容。
而 from test1 import * 就是可以直接使用模块的
函数名，不需要加前缀

cl ass t1子类操作
cl ass t2子类操作

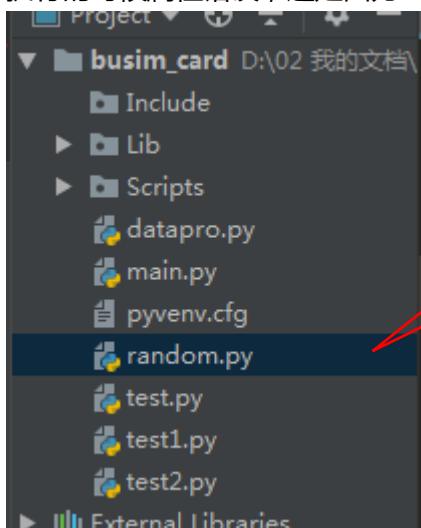
import 导入模块名和系统模块名冲突问题

```
random.py
1 import random
2
3 print(random.__file__)
4
5 rand = random.randint(0,10)
6 print("随机数范围 = %d" %rand)
```

本来系统有一个 random 的 py 文件，我自己又创建了一个 random 的 py 文件

```
Traceback (most recent call last):
  File "D:/02 我的文档/Desktop/busim_card/random.py", line 1, in <module>
    import random
  File "D:/02 我的文档/Desktop/busim_card/random.py", line 5, in <module>
    rand = random.randint(0,10)
AttributeError: module 'random' has no attribute 'randint'
```

执行的时候属性错误，这是因为 random 模块的加载顺序问题



程序运行时，python 解释器会先去找工程里面的 import 加载的同名文件，这里找到了我自己创建的 random 文件，这时候，解释器就不会再去系统寻找 random 文件，所以运行出错

```
random.py
1 import random
2
3 print(random.__file__)
4
5 rand = random.randint(0,10)
6 print("随机数范围 = %d" %rand)
```

我把工程里面的 random.py 删除掉，在另外文件里面使用 random

打印库文件搜索路径

```
D:\pycharm\lib\random.py
随机数范围 = 4
```

你看执行没有问题了，因为这时 python 解释器搜索的库文件在 pycharm 内置的路径下

import 导入其它模块时，可能会自动执行其它模块的测试代码，如何解决？



```
title1 = "模块1"

def test1():
    print("执行test1子函数")

class classt1(object):
    def classt1(self):
        print("class t1子类操作")

test1() #test1子模块执行测试代码
```

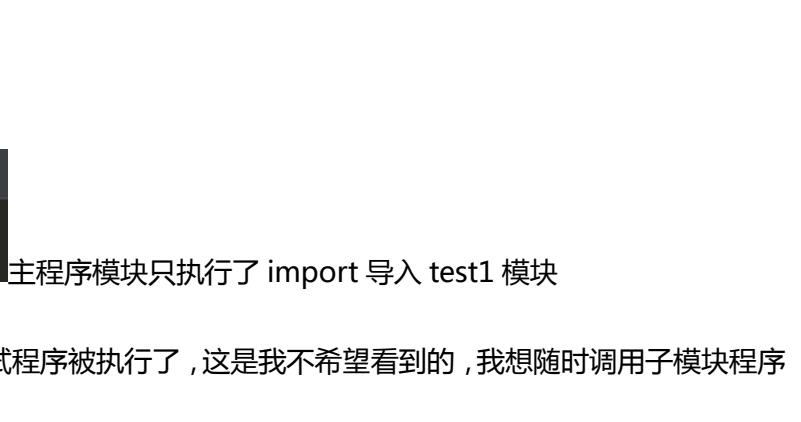
在子模块测试程序



执行test1子函数
执行没有问题

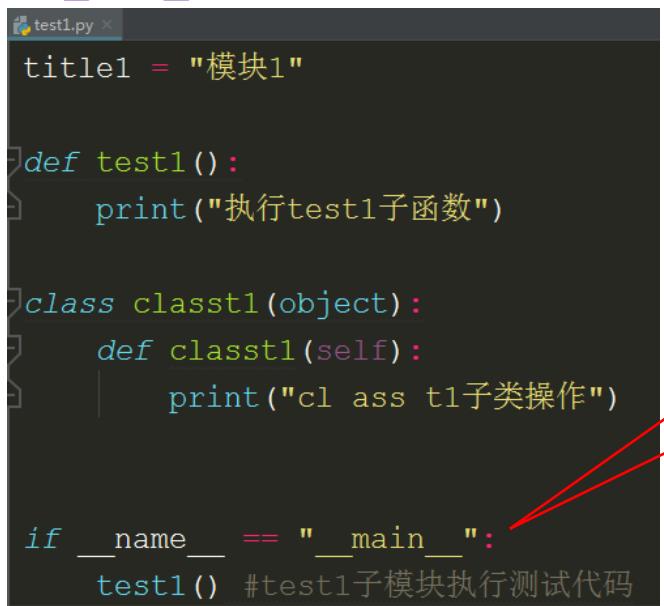
1 | import test1

主程序模块只执行了 import 导入 test1 模块



执行test1子函数
但是子模块的测试程序被执行了，这是我不希望看到的，我想随时调用子模块程序时，子模块程序才执行，怎么解决呢？

查询__name__里面的值来决定现在是在执行 test1 子模块程序，还是在执行 test 主程序

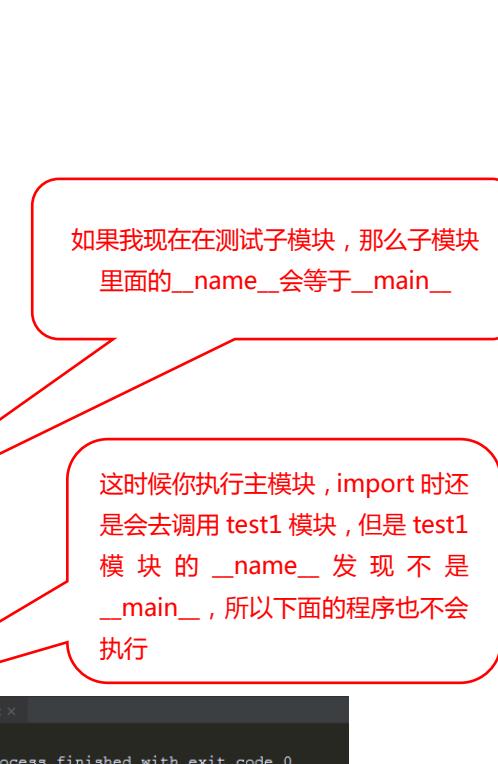


```
title1 = "模块1"

def test1():
    print("执行test1子函数")

class classt1(object):
    def classt1(self):
        print("class t1子类操作")

if __name__ == "__main__":
    test1() #test1子模块执行测试代码
```



如果我现在在测试子模块，那么子模块里面的__name__会等于__main__

这时候你执行主模块，import 时还是会去调用 test1 模块，但是 test1 模块的 __name__ 发现不是 __main__，所以下面的程序也不会执行

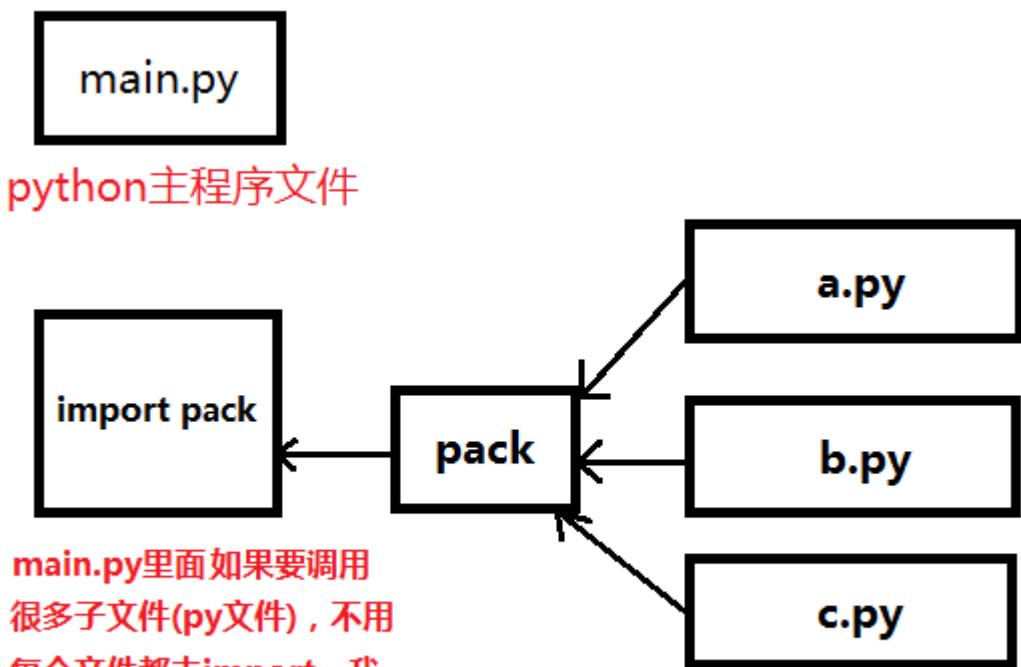
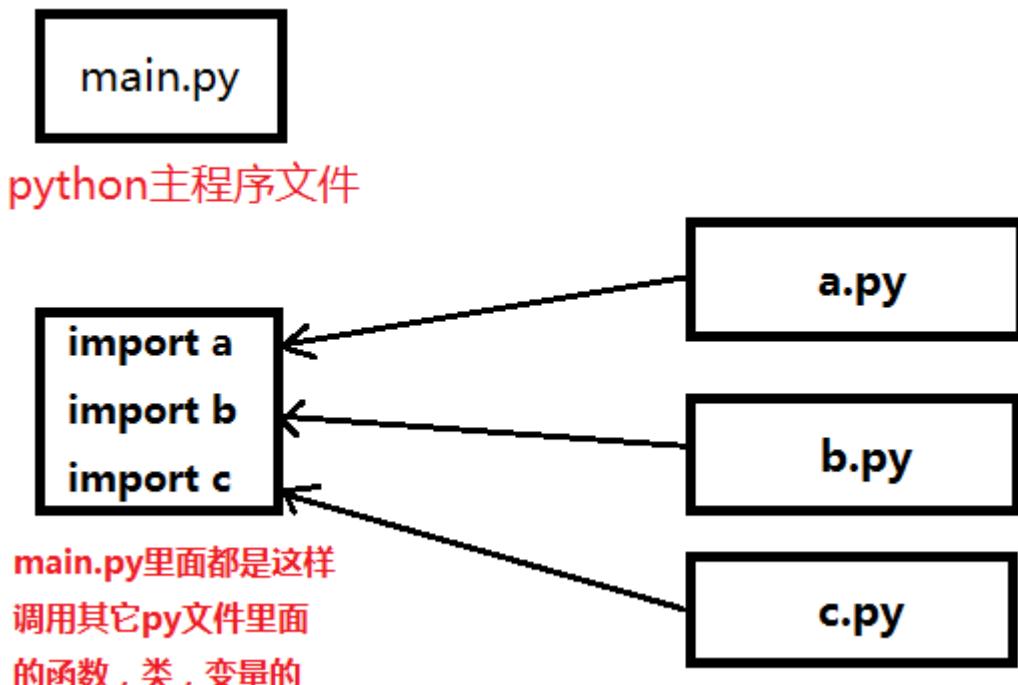


```
1 | import test1
```

```
test x
Process finished with exit code 0
```

Python 的子模块(py 文件)只是类似于 C 语言的 C 文件，但是 python 还有一种文件叫做包

什么是包，包就类似于 C 语言的静态库.a 或者动态库.so，把几个 C 文件汇总到一个文件里面



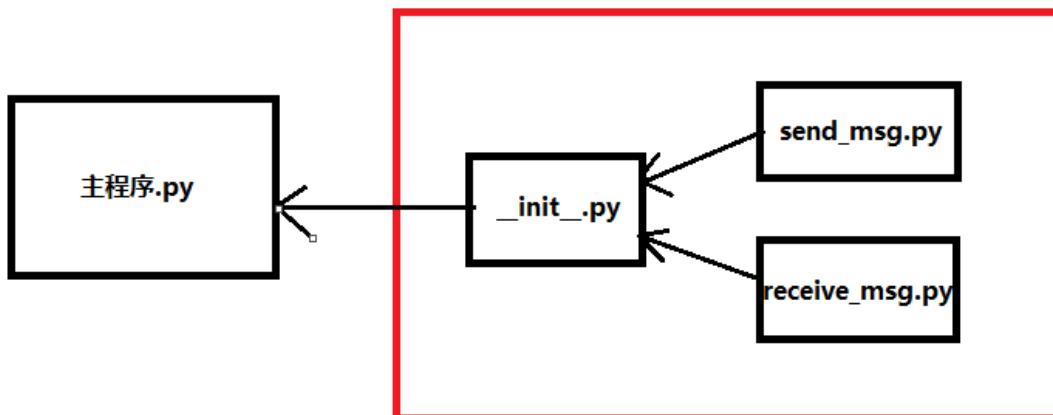
所以包管理就是将其余的 py 文件归纳进包，然后主程序只要 import 包，就可以调用其它 py 文件里面的子函数，不用调用每个 py 文件主程序都要去写 import，如果 100 个 py 文件，未必你就去写 100 个 import，这样的话文件看起就很不舒服，打字也很累。

定义个发送函数的 py 文件

```
send_msg.py x
def send(text):
    print("发送 %s" %text)
```

定义个接受函数的 py 文件

```
receive_msg.py x
def receive():
    return "接受程序receive_msg执行"
```

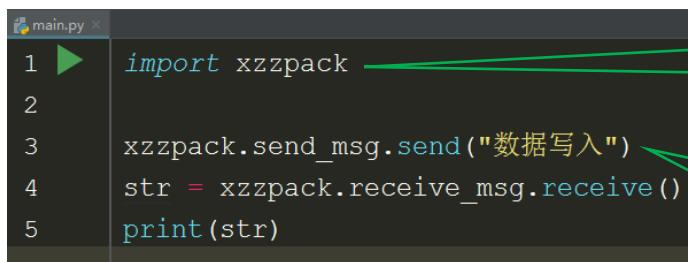
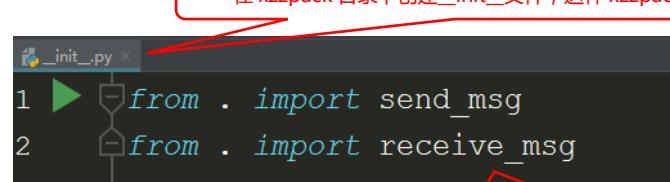
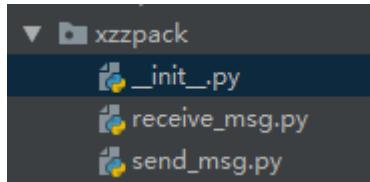


所有文件放在xzzpack目录下



但是想让 xzzpack 目录发生功效，必须在 xzzpack 目录下建立一个_init_.py 文件，这个文件是特殊文件，名字必须一样，文件必须要有。

from . import 模块名 使用方法



发送 数据写入
接受程序receive_msg执行

Python 读写文件操作

读文件操作



```
file = open("D:\\02 我的文档\\Desktop\\xzz123.txt", 'r') #1. 打开文件
text = file.read() #2. 读文件
file.close() #3. 关闭文件一定要做

print(text) #打印文件内容
```

这是文本文件里面的内容
执行没有问题

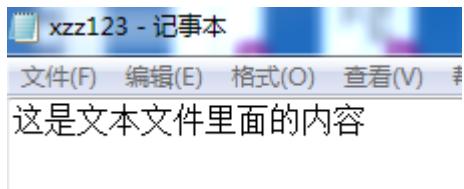
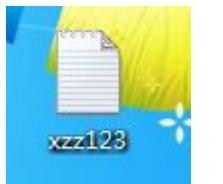
open("D:\\02 我的文档\\Desktop\\xzz123.txt" , 'r') //这种双斜杠是一种寻找文件路径的方法

如果你觉得双斜杠太麻烦，就改成下面这样

open(r "D:\\02 我的文档\\Desktop\\xzz123.txt" , 'r') //前面加 r 就可以改成单斜杠

记住：你 read 一次文件后，文件的所有内容都读取完了，这时候文件指针在文件的末尾

文件的写入



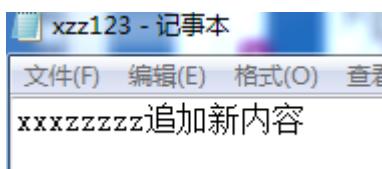
```
file = open(r"D:\02 我的文档\Desktop\xzz123.txt", 'w') #1.打开文件
file.write("xxxxxxxx") #2.写文件
file.close() #3.关闭文件一定要做
```



这种直接 write 写入的方式将文件原本内容会进行覆盖

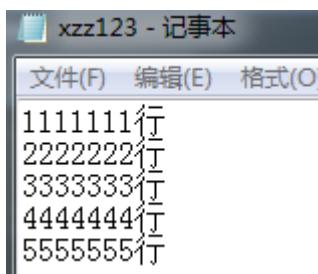
用 'a' 打开文件，就将文件指针移动到文件末尾，在文件末尾增加新内容

```
file = open(r"D:\02 我的文档\Desktop\xzz123.txt", 'a') #1.打开文件
file.write("追加新内容") #2.写文件
file.close() #3.关闭文件一定要做
```



你看，前面内容没有被覆盖掉，后面内容增加了

readline 每次执行读取文件一行内容，而不会把整个文件读完



这是文本的 5 行内容

```
file = open(r"D:\02 我的文档\Desktop\xzz123.txt", 'r') #1.打开文件
text = file.readline() #一次读取一行内容
file.close() #3.关闭文件一定要做

print(text)
```

1111111行

执行一次 readline 读取一行文本内容给 text 变量

```
file = open(r"D:\02 我的文档\Desktop\xzz123.txt", 'r') #1.打开文件  
file.readline() #读取第1行  
text = file.readline() #执行第2次读取第2行  
file.close() #3.关闭文件一定要做  
print(text)
```

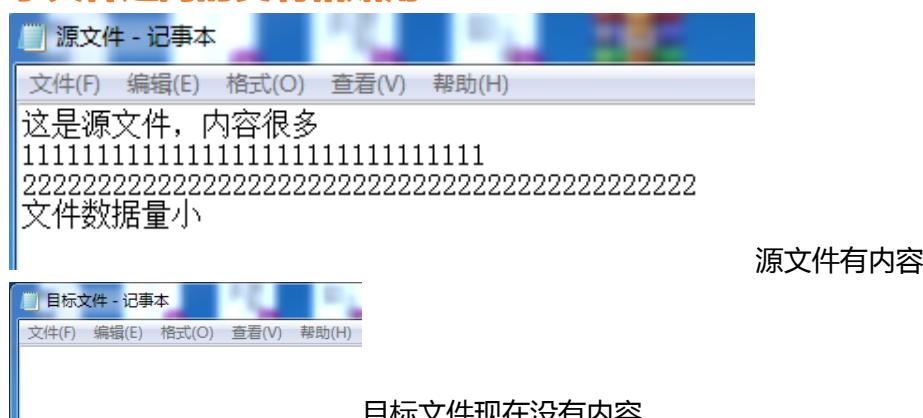
222222行 执行 2 次 readline 就读取第 2 行内容

```
file = open(r"D:\02 我的文档\Desktop\xzz123.txt", 'r') #1.打开文件  
  
while True:  
    text = file.readline() #每次循环都读取一行  
    print(text)  
  
    if not text:#如果text里面没有内容就跳出死循环  
        break  
  
file.close() #3.关闭文件一定要做
```

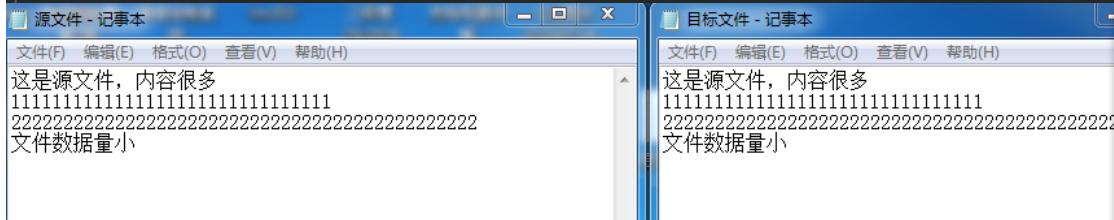
死循环，读取到文
本最后一行没有内
容就会返回空数据
给 text 变量

1111111行
2222222行
3333333行
4444444行
5555555行
执行没有问题

小文件之间的复制粘贴用 read



现在将源文件内容复制给目标文件

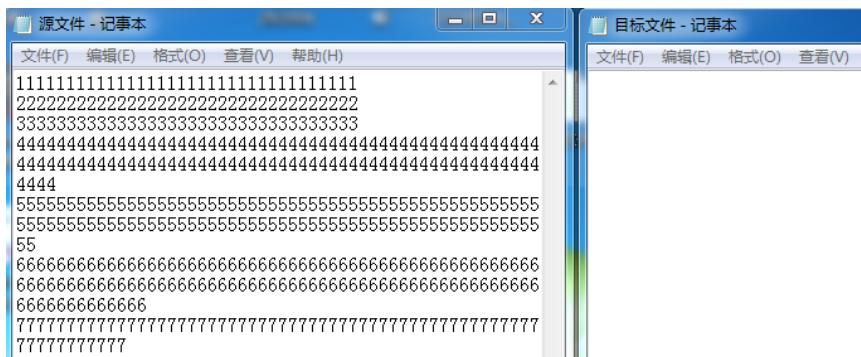


你看目标文件有源文件内容了。

大文件复制粘贴用 readline

为什么用 `readline` 呢？`read` 函数是将整个文件内容复制到内存后才执行后面的程序。

而 `readline` 是复制文件一行内容，执行下面其它程序，然后返回来再执行一次 `readline`，如此循环，这样每次执行 `readline` 只有一行数据，内存压力就要小很多。



执行前的源文件和目标文件

```
file_read = open(r"D:\02 我的文档\Desktop\源文件.txt", 'r') #1.打开文
file_write = open(r"D:\02 我的文档\Desktop\目标文件.txt", 'w')

while True:
    text = file_read.readline() #每次获取源文件数据一行内容
    if not text:
        break

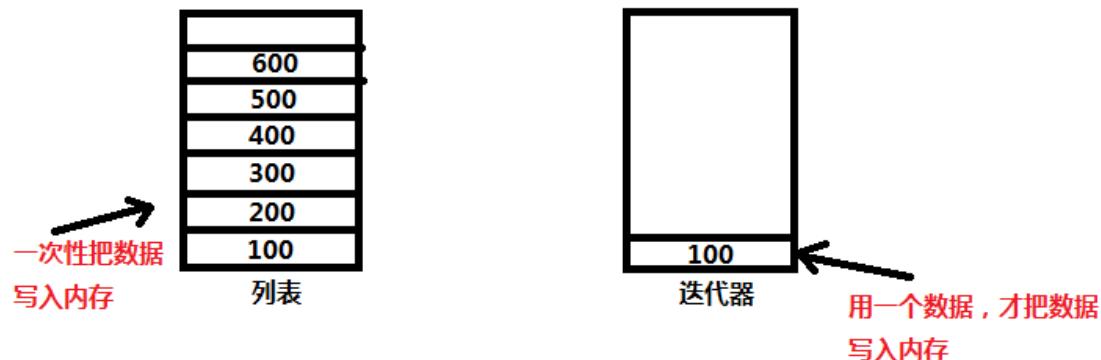
    file_write.write(text) #每次都写一行进目标文件

file_read.close() #3.关闭文件一定要做
file_write.close()
```

为什么源文件 open 的时候写文件用的是 ‘w’ 标识符，怎么每次覆盖文件没有把第一行覆盖掉呢？不是 ‘a’ 才是追加内容吗？这是因为用的 readline，每次都是写入一行就会跳到下一行。所以后面写的内容不会覆盖第 1 行内容。

Python 读写 csv 文件

首先了解 python 迭代器



我们知道 python 列表就像 C 语言数组一样，如果数组元素很多，会一次性将栈空间撑爆造成内存溢出。而迭代器就是用某个数据的时候，才将数据写入内存，不用的时候不写，不像 C 语言数组一次性定义了很多，那么不管用不用都必须让数组占用到内存空间。

```
liebiao1 = [100, 200, 300, 400, 500, 600] #根据列表元素个数就会提前占用内存空间
print(liebiao1[1])
liebiao2 = [1, 2, "aaa", 3.14, 3+4j] #列表里面可以放不同类型的元素
print(liebiao2[2])
print(liebiao2[4])
```

执行正确，现在从列表现象我们引出迭代器来思考，列表和迭代器有什么区别。

其实迭代器和我上面给的图片逻辑一样，具有延时加载功能，就是说迭代器执行一句代码才在内存加载一个数据，不会像列表那样提前就把元素数量在内存上加载完。

迭代器有两个功能，迭代器对象和可迭代对象

可迭代对象使用 Iterable

```
liebiao = [100, 200, 300, 400, 500, 600]
```

from....import...就是加载外部模块里面的某一个对象或者函数

```
from collections.abc import Iterable
```

1.isinstance 函数就是判断某个对象

2.是不是适合转换后做以下这个对象

```
print(isinstance(liebiao, Iterable))
```

True

返回的 True 表示列表适合做迭代对象

迭代器使用 Iterator

```
liebiao = [100, 200, 300, 400, 500, 600]
```

加载 abc 模块里面的迭代器

```
from collections.abc import Iterable, Iterator
```

```
print(isinstance(liebiao, Iterator))
```

False

返回 false 证明，列表不能做成迭代器使用

列表创建的 4 种方法

```
a = [50, 30, 20]
```

```
b = [1, 2, 5.6, "strrrr", 3+5j]
```

```
print(a)
```

```
print(b)
```

```
[50, 30, 20]
```

```
[1, 2, 5.6, 'strrrr', (3+5j)]
```

这是第 1 种创建列表的常规方法，我前面列表章节讲过第 16 页

```
a = list(range(10))
```

#用range函数自动生成10个数返回给list，list再将列表返回给a，a就成了列表对象了
print(a)

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

这是第 2 种创建列表的方法，省去了手敲数据进列表，而且除了 range 生成有规律的数，还可以用随机值函数库来生成，看自己选择

第3种创建列表方法，用列表推导式创建列表

```
a = [x for x in range(10)]  
print(a)
```

这种推导式写法是什么意思

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

列表推导式规则

列表变量 = [语句 for 变量值 in 函数]

语句：是for循环一次就会执行一次的语句

变量值：就是循环一次，把函数返回的值赋值给变量值

函数：可以是其它语句

例子：a = [x for x in range(10)]

1步：a = [x for x in 10]

第1次循环 a = [x for 1 in 10]

然后把1赋值给语句的x变量 a = [x for 1 in 10]

然后把x的1放入列表赋值给a列表对象

第2次循环 a = [x for 2 in 10]

然后把2赋值给语句x变量 a = [x for 2 in 10]

然后把x的2放入列表，赋值给a列表对象

后面过程一次类推.....

```
a = [x*2 for x in range(10)]  
print(a)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

这个更明显的解释了列表推导式，执行一次循环后，把 for 里面的 x 值赋值给前面的 x*2，x*2 会执行一次，所以每次 x 的值都会平方后给列表

```
b = (x for x in range(1,10))  
print(type(b)) #任何变量或者对象，你不知道是什么类型就用type查看
```

```
<class 'generator'>
```

从打印结果看，这个 b 是生成器。

```
from collections.abc import Iterable, Iterator  
  
b = (x for x in range(1,10))  
print(type(b))  
print(isinstance(b, Iterator)) #b这个对象时迭代器  
print(next(b)) #因为b是迭代器，所以直接用next函数获取数据  
print(next(b)) #如果要多次获取数据就多次执行next  
print(next(b)) #如果多次手敲打next太麻烦就用for循环
```

写入 10 个数据自动创建迭代器存储，和列表推导式一个逻辑

```
main.c  
<class 'generator'>  
True  
1  
2  
3
```

那么迭代对象和迭代器有什么不一样呢？

```
liebiao = [100, 200, 300, 400, 500, 600]
```

```
print(next(liebiao))
```

迭代对象直接用 next 函数获取

数据会报错

```
main
Traceback (most recent call last):
  File "D:/02 我的文档/Desktop/busim_card/main.py", line 3, in <module>
    print(next(liebiao))
TypeError: 'list' object is not an iterator
```

你看执行报错了，记住只要是可以用 for 循环遍历修改数据，查询数据的列表，字典，字符串等等...都属于迭代对象，而不是迭代器。

Iter(x)内置函数，将列表，字典，字符串这些迭代对象转换成迭代器

```
liebiao = [100, 200, 300, 400, 500, 600]
```

```
b = iter(liebiao) #用iter内置函数将迭代对象列表转换成迭代器
print(next(b))
print(next(b))
print(next(b))
```

100

200

300 执行结果正确

几条迭代器和可迭代对象的原则

1. 可迭代对象包含迭代器。

2. 如果一个对象拥有`__iter__`方法，其是可迭代对象；如果一个对象拥有`next`方法，其是迭代器。

3. 定义可迭代对象，必须实现`__iter__`方法；定义迭代器，必须实现`__iter__`和`next`方法。

可迭代对象（`list`, `set`, `dict`）可以重复迭代，只能使用`for`循环；迭代器只能迭代一次，什么意思呢？

```
liebiao = [100, 200, 300, 400, 500, 600]
```

```
b = iter(liebiao) #用iter内置函数将迭代对象列表转换成迭代器
```

```
print(next(b)) #输出100
```

```
print(next(b)) #输出200
```

```
print(next(b)) #输出300
```

```
print(next(b)) #输出400
```

```
print(next(b)) #输出500
```

```
print(next(b)) #输出600
```

```
print(next(b)) #输出700
```

你看我执行了 7 次

```
100
200
300
400
500
600
Traceback (most recent call last):
File "D:/02 我的文档/Desktop/busim_card/main.py", line 10, in <module>
    print(next(b)) #输出700
StopIteration
```

next 执行到 7 次的时候就报异常

这就是迭代器只能迭代一次的原因，比如你有 10 个数据在内存，迭代器的 next 执行一次，你内存的数据就会在内存少一个，所以这就是迭代器的好处，执行完了节省内存。

```
liebiao = [100, 200, 300, 400, 500, 600]
```

```
b = iter(liebiao) #用iter内置函数将迭代对象列表转换成迭代器
print(next(b)) #输出100
print(next(b)) #输出200
print(next(b)) #输出300
print(next(b)) #输出400
print(next(b)) #输出500
print(next(b)) #输出600
b = iter(liebiao) #用iter内置函数将迭代对象列表转换成迭代器
print(next(b)) #输出100
```

如果你想再次使用迭代器，你就得给迭代器重新添加迭代对象

```
100
200
300
400
500
600
100
```

迭代器 next 重新在内存加载迭代对象，然后依次获取数据

可迭代对象就不会像迭代器那样每次执行 next 就在内存消失一个数据。可迭代对象就是列表，元组，字典，字符串，可以用 for 循环重复使用，但是这些数据会在内存一直存放着。这就是迭代器和可迭代对象的实际区别。

CSV 读取文件实验

Windows 下创建 txt 文件，然后每个字符串用逗号分割开，记住逗号要用英文输入法

文件名(N): xzz

保存类型(T): CSV (逗号分隔)

保存为 csv 文件

写好 txt 后保存，用 excel 打开->数据->分列设置文件格式

CSV 文件的数据都是以逗号作为两个数据的分割

```
书名,作者,出版社,原作名,译者,出版年,页数,定价,ISBN
计算机程序构造和解释,harolable,IPC,涵涵,韩寒,1995,153,50,20151050
重构,harolable,IPC,涵涵,韩寒,2000,500,20,20170808
架构师,“全栈工程师,真全栈少年”,大陆,NB工程师,真全栈少年,2019,150,100,20191010
```

但是若遇到一个数据要用逗号来说明两个事物，但是它本身是一个数据，那么就用引号包裹起来

```
import csv
```

需要使用 csv 的函数，倒入 csv 功能模块

```
rf = open(r'D:\02 我的文档\Desktop\test.csv')
```

打开 CSV 文件

```
reader = csv.reader(rf)
```

将 csv 的对象 rf 放入 csv.reader 读取函数中，然后会自动将 CSV 文件每一行数据做成列表迭代器

```
print(next(reader))
```

然后用 next 读取迭代器每一行数据，这个前面章节讲过，读取完了就不能再读取了，需要重新加载

```
print(next(reader))
```

```
print(next(reader))
```

```
print(next(reader))
```

读取 CSV 文件程序

```
[‘书名’, ‘作者’, ‘出版社’, ‘原作名’, ‘译者’, ‘出版年’, ‘页数’, ‘定价’, ‘ISBN’]  
[‘计算机程序构造和解释’, ‘harolable’, ‘IPC’, ‘涵涵’, ‘韩寒’, ‘1995’, ‘153’, ‘50’, ‘20151050’]  
[‘重构’, ‘harolable’, ‘IPC’, ‘涵涵’, ‘韩寒’, ‘2000’, ‘500’, ‘20’, ‘20170808’]  
[‘架构师’, ‘全栈工程师,真全栈少年’, ‘大陆’, ‘NB工程师’, ‘真全栈少年’, ‘2019’, ‘150’, ‘100’, ‘20191010’]
```

你看 CSV 文件读取出来的数据，每一行用列表包裹，用逗号分隔数据项。

```
import csv
```

```
rf = open(r'D:\02 我的文档\Desktop\test.csv')
```

```
reader = csv.reader(rf)
```

```
for book in reader:
```

如果你觉得用手敲 next 太累，就用 for 循环，
for 循环会自动将 reader 的数据赋值给 book，
还可自动计算迭代器长度

```
    print(book)
```

```
[‘书名’, ‘作者’, ‘出版社’, ‘原作名’, ‘译者’, ‘出版年’, ‘页数’, ‘定价’, ‘ISBN’]  
[‘计算机程序构造和解释’, ‘harolable’, ‘IPC’, ‘涵涵’, ‘韩寒’, ‘1995’, ‘153’, ‘50’, ‘20151050’]  
[‘重构’, ‘harolable’, ‘IPC’, ‘涵涵’, ‘韩寒’, ‘2000’, ‘500’, ‘20’, ‘20170808’]  
[‘架构师’, ‘全栈工程师,真全栈少年’, ‘大陆’, ‘NB工程师’, ‘真全栈少年’, ‘2019’, ‘150’, ‘100’, ‘20191010’]
```

csv.reader 函数默认的分隔符是逗号所以你可以直接用，但是如果你的 csv 文件分隔符是分号或者其它符号，看下面怎么 cozy

```
书名;作者;出版社;原作名;译者;出版年;页数;定价;ISBN  
计算机程序构造和解释;harolable;IPC;涵涵;韩寒;1995;153;50;20151050  
重构;harolable;IPC;涵涵;韩寒;2000;500;20;20170808  
架构师;“全栈工程师,真全栈少年”;大陆;NB工程师;真全栈少年;2019;150;100;20191010
```

```
import csv
```

```
rf = open(r'D:\02 我的文档\Desktop\test.csv')
```

```
reader = csv.reader(rf)
```

```
for book in reader:
```

```
    print(book)
```

这是上一页的传统写法

```
[‘书名’, ‘作者’, ‘出版社’, ‘原作名’, ‘译者’, ‘出版年’, ‘页数’, ‘定价’, ‘ISBN’]  
[‘计算机程序构造和解释’, ‘harolable’, ‘IPC’, ‘涵涵’, ‘韩寒’, ‘1995’, ‘153’, ‘50’, ‘20151050’]  
[‘重构’, ‘harolable’, ‘IPC’, ‘涵涵’, ‘韩寒’, ‘2000’, ‘500’, ‘20’, ‘20170808’]  
[‘架构师’, ‘全栈工程师,真全栈少年’, ‘大陆’, ‘NB工程师’, ‘真全栈少年’, ‘2019’, ‘150’, ‘100’, ‘20191010’]
```

你看输出的字符串两边还有单引号

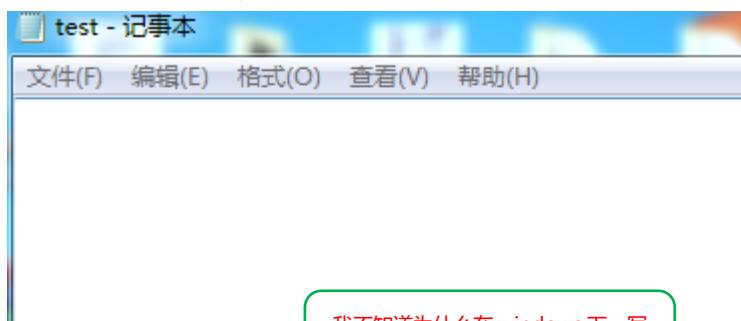
```
import csv  
rf = open(r'D:\02 我的文档\Desktop\test.csv')  
reader = csv.reader(rf, delimiter=';')  
for book in reader:  
    print(book)
```

在 reader 用 delimiter 指定分隔符号

```
[‘书名,作者,出版社,原作名,译者,出版年,页数,定价,ISBN’]  
[‘计算机程序构造和解释,harolable,IPC,涵涵,韩寒,1995,153,50,20151050’]  
[‘重构,harolable,IPC,涵涵,韩寒,2000,500,20,20170808’]  
[‘架构师,”全栈工程师,真全栈少年”,大陆,NB工程师,真全栈少年,2019,150,100,20191010’]
```

你看每段字符的单引号就没有了。所以如果不是逗号分隔的 CSV 文件，记得指定分隔符。

CSV 写入文件实验



```
import csv  
wf = open('D:\\02 我的文档\Desktop\\test.csv', 'w')  
writer = csv.writer(wf, delimiter=';')  
writer.writerow(['xxx', 'yyy', 'zzz'])  
writer.writerow([123, 456, 789])  
writer.writerow([3.14, 5.08, 3+4j])  
wf.flush()
```

我不知道为什么在 windows 下，写文件打开方式要加两个\\

分隔符用分号

写第 1 行

写第 2 行

写完之后一定要用 open 返回的对象执行 flush 把数据保存在磁盘上

```
xxx;yyy;zzz  
123;456;789  
3.14;5.08;(3+4j)
```

你看写入 CSV 文件的数据是用分号分隔的

with....as....语句用法

1. 我们常规打开操作文件的方法

```
file = open("/tmp/foo.txt")
data = file.read()
file.close()
```

这里有两个问题。一是可能忘记关闭文件句柄；二是文件读取数据发生异常，没有进行任何处理。

2.如果我加入异常处理

```
file = open("/tmp/foo.txt")
try:
    data = file.read()
finally:
    file.close()
```

虽然这段代码运行良好，但是太冗长了。

所以下面用 with....as...可以偷懒简化文件操作步骤

```
with open("/tmp/foo.txt") as file:
    data = file.read()
```

这样 open 返回的文件对象还是赋值给 as 后面的变量 file

在文件执行结束后会自动 close 关闭文件，如果有异常 with 会在运行的时候弹出异常，这样很方便

下面用 CSV 文件读写来做个案例

比如我网络爬虫爬取的一大堆乱七八糟的图书数据保存在 CSV 文件。

```
书名;作者;出版社;原作名;译者;出版年;页数;定价;ISBN
计算机程序构造和解释;harolable;IPC;涵涵;韩寒;1995;153;50;20151050
重构;harolable;IPC;涵涵;韩寒;2000;500;20;20170808
架构师;“全栈工程师,真全栈少年”;大陆,NB工程师;真全栈少年;2019;150;100;20191010
```

这就是我爬取的数据在 test.csv 文件

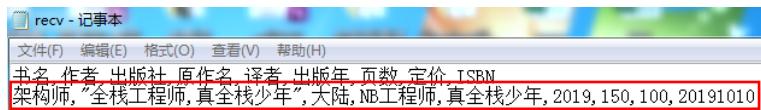


我要自动建立一个 recv.csv 文件来获取已爬取数据里面大于 100 元的书籍

```
import csv
with open('D:\\02 我的文档\\Desktop\\test.csv') as rf:
    reader = csv.reader(rf) #读取csv文件的所有数据
    headers = next(reader) #用迭代器把csv文件的头部数据读出来，就是那些标题
    with open('D:\\\\02 我的文档\\Desktop\\\\recv.csv','w') as wf:
        writer = csv.writer(wf) #将要写入csv文件的变量创建出来
        writer.writerow(headers) #将第4行变量读出来的标题写入新的csv文件
        for book in reader: #循环读取test文件每一行数据
            price = book[-2] #获取当前行book的倒数第2列数据给price变量
            if int(price) >= 80: #如果在test文件发现有大于80价格的书
                writer.writerow(book) #book获取的是test文件的数据，写入新csv文件
```

Process finished with exit code 0

执行成功



大于 80 元的书分类出来了。

列表，字典，集合中取出想要的数据

过滤掉列表[3,9,-1,10,20,-2...]中的负数，将正数全部取出来

```
data = [3, 9, -1, 10, 20, -2]
res = []
for x in data:
    if x >= 0:
        res.append(x)
print(res)
```

[3, 9, 10, 20] 你看筛选出正数了，但是这种 for 循环的执行效率很差，代码冗余。

用列表解析来筛选正数

```
data = [3, 9, -1, 10, 20, -2]
res = [x for x in data if x >= 0]
print(res)
```

[3, 9, 10, 20]

res = [x for x in data if x >= 0]

1. 根据data列表数据个数的多少进行循环多少次

2. [x for x in data if x >= 0] 将每个data的数据赋值给x

3. [x for x in data if x >= 0] 将x得到的值进行判断

4. [x for x in data if x >= 0] 将判断出来的值赋值给第1项x，这样循环几次，第一项x就是列表了
里面存放的是筛选出来的数

这种列表筛选方式执行效率高，代码简洁

lambda 表达式使用

lambda的意义在于，有些简单的语句用函数来写很占用内存空间，这种情况可以交给lambda来完成

语法 lambda: 参数列表 : 表达式

```

def test():
    return 100

print(test)
print(test())

```

我们一般都是用函数来实现返回值

```

func=lambda : 100
print(func)
print(func())

```

然后调用函数，打印 function 表示
这个 test 是函数变量

lambda 不填入参数，只填入表达式，那么 lambda 就
返回函数地址，这是 lambda 无参数写法

```

<function test at 0x00000000027B9AE8>
100
<function <lambda> at 0x00000000027B9A60>
100

```

这种 lambda 写起代码比函数方便，而且只需效率高

所以 lambda 有两种写法，一种匿名函数写法，一种计算数据写法

```

def add(a, b):
    return a + b

```

这是普通函数写法效率低

```

result = add(1, 2)
print("function = %d" % result)

```

参数 表达式

```

fn = lambda a, b:a + b
print("lambda = %d" % fn(5, 10))

```

按照规则(lambda 参数:表达式) 返回的就是一个函数地址

```

function = 3
lambda = 15

```

这就是 lambda 的函数形式

```

def add(a, b, c=100):
    return a + b + c

```

传入默认形参值 100

```

result = add(1, 2)
print("function = %d" % result)

```

lambda 也可以传入默认形参

```

fn = lambda a, b, c=100:a + b + c
print("lambda = %d" % fn(5, 10))

```

```

function = 103
lambda = 115

```

```
func = lambda *args: args  
print(func(1, 2, 3, 4, 5))
```

如果 lambda 参数用的*args，那么返回的函数就是可以传入任意多个参数的函数，函数返回值是元组

```
(1, 2, 3, 4, 5)
```

```
func = lambda **kwargs: kwargs  
print(func(name='zidian', arg=30, xzz=3.15))
```

Lambda **kwargs 就是传入无数个字典

```
{'name': 'zidian', 'arg': 30, 'xzz': 3.15}
```

```
func = lambda a, b: a if a > b else b  
print(func(1000, 500))
```

```
1000
```

Python 三目运算符，if 运算成立，返回 if 前面的 a，否则返回 else 后面的 b

filter 过滤器使用

返回生成器变量 = filter(匿名函数，过滤的值放入的变量：过滤条件，要过滤的列表变量)

```
data = [3, 9, -1, 10, 20, -2]  
g = filter(lambda x: x >= 0, data)  
  
print(next(g))  
print(next(g))  
print(next(g))  
print(next(g))  
print(next(g))
```

循环结束后，filter 会把为真的数据封装到生成器里面返回

循环获取列表里面的值

每次循环都用 lambda 去获取 data 里面的数据，然后判断，如果为真返回给 filter

```
3  
9  
10  
20  
Traceback (most recent call last):  
File "D:/02 我的文档/Desktop/busim_ca  
    print(next(g))  
StopIteration
```

生成器的数据必须用迭代器获取，我们知道迭代器是获取一个数据少一个

```
data = [3, 9, -1, 10, 20, -2]

g = filter(lambda x: x >= 0, data)
lst = list(g)
print(lst)
```

将生成器变量在迭代器消耗
之前，使用 list 函数转换成
列表，就可以一直保存

```
[3, 9, 10, 20]
```

用 filter 过滤执行速度也很快，但是没有列表解析来的快

字典解析

筛选出字典{'LiLei': 79, 'Jim': 88, 'Lucy': 92...}中值高于90的项

基础回顾，字典的 Items 函数

Items 循环返回字典里面的个数内容，返回后是元组形式

```
dict = {'Google': 10, 'Runoob': 20, 'taobao': 30}
val = dict.items()
print(val)

for value in dict.items():
    print(value)
```

Items 方式适合用于 for 循环

```
dict_items([('Google', 10), ('Runoob', 20), ('taobao', 30)])
('Google', 10)
('Runoob', 20)
('taobao', 30)
```

字典解析案例

```
dict = {'LiLei': 79, 'Jim': 88, 'Lucy': 92}

ret = {k:v for k,v in dict.items() if v >= 90}
print(ret)
```

大于 90 将这项字典返回给
k:v,然后返回字典类型到 ret

和列表解析差不多，先用 items 计算循环次数，
每次循环把 dict 的字典 key 和 value 放入 k,v

然后取出这次循环
字典 k, v 里面的 v
值是否大于 90

```
{'Lucy': 92}
```

你看，根据 value 值，提取出了对应的人名

集合解析

筛除集合中 {3,77,89,6,32,20...} 筛选出能被3整除的数

```
gather = {3, 77, 89, 6, 32, 20} #大括号是集合  
ret = {x for x in gather if x % 3 == 0}  
print(ret)
```

{3, 6} 集合解析和列表差不多，筛选出来返回的类型是集合类型

使用 enum 增加程序的可读性

C/C++语言有 define , JAVA 有 enum 来给数字取名 , python 也有 enum

```
from enum import IntEnum  
  
class studentEnum(IntEnum):  
    NAME = 0  
    AGE = 1  
    SEX = 2  
    EMAIL = 3  
  
yuanzu = ('真全栈', 16, 'mmm', '163@xzz.com')
```

使用枚举变量关键字，所以导入 enum 里面的 IntEnum

IntEnum 关键字就是 int 类型

给每个 int 类型的值赋值上可读的名称

使用类枚举来获取元组里面的数据

这里传入的就是 2，其余几个以此类推

```
真全栈  
16  
mmm  
163@xzz.com
```

命名元组

```
from collections import namedtuple  
student = namedtuple('Student', ['name', 'age', 'sex', 'email'])  
s2 = student('真全栈', '16', 'mmm', 'www.abs@163.com')  
print(s2)  
  
Student(name='真全栈', age='16', sex='mmm', email='www.abs@163.com')
```

你看打印出来的数据，每个元组都有对应的名字，但是记住这不是字典，这是元组

```
from collections import namedtuple  
  
student = namedtuple('Student', ['name', 'age', 'sex', 'email'])  
s2 = student('真全栈', '16', 'mmm', 'www.abs@163.com')  
print(s2.name)  
print(s2.age)
```

这样就和枚举是一样的效果了。用名字获取元组的对应的值

字典无法排序，所以只有将其转换成列表或者元组来排序

```
keyvalue = {'Lilei':20, 'xzz':10, 'ttt':30, 'Cddd':50, 'jav':40, 'yeall':60}  
  
lst = [(v, k) for k, v in keyvalue.items()]  
  
print(lst)  
print(sorted(lst))
```

这就是字典排序的方法

如果我想大值在前小值在后呢？

```
keyvalue = {'Lilei':20, 'xzz':10, 'ttt':30, 'Cddd':50, 'jav':40, 'yeall':60}  
  
lst = [(v, k) for k, v in keyvalue.items()]  
print(sorted(lst))  
  
lnum = sorted(lst, reverse=True)  
print(lnum)
```

```
[(10, 'xzz'), (20, 'Lilei'), (30, 'ttt'), (40, 'jav'), (50, 'Cddd'), (60, 'yeall')]  
[(60, 'yeall'), (50, 'Cddd'), (40, 'jav'), (30, 'ttt'), (20, 'Lilei'), (10, 'xzz')]
```

你看 reverse 之后就是大值在前，小值在后

zip 函数使用方法

```
za = zip([1, 2, 3], [4, 5, 6])  
print(next(za))  
print(next(za))  
print(next(za))
```

zip 是将每个列表最小的数据放入一个元组

(1, 4)
(2, 5)
(3, 6)

zip 返回的是生成器，只能用 next 访问

```
keyvalue = {'Lilei': 20, 'xzz': 10, 'ttt': 30, 'Cddd': 50, 'jav': 40, 'yeall': 60}  
  
za = zip(keyvalue.values(), keyvalue.keys())  
print(next(za))  
print(next(za))  
print(next(za))
```

如果字典放入 zip，那么 zip 返回的生成器就是一串字典的元组

```
(20, 'Lilei')  
(10, 'xzz')  
(30, 'ttt')
```

```
keyvalue = {'Lilei': 20, 'xzz': 10, 'ttt': 30, 'Cddd': 50, 'jav': 40, 'yeall': 60}  
  
za = zip(keyvalue.values(), keyvalue.keys())  
lst = list(za)  
print(lst)  
print(sorted(lst))
```

所以 zip 也可以将字典转化成列表，然后排序

```
[(20, 'Lilei'), (10, 'xzz'), (30, 'ttt'), (50, 'Cddd'), (40, 'jav'), (60, 'yeall')]  
[(10, 'xzz'), (20, 'Lilei'), (30, 'ttt'), (40, 'jav'), (50, 'Cddd'), (60, 'yeall')]
```

Python 寻找列表中出现次数最多的数

collections 模块的 Counter 类使用

```
from collections import Counter  
  
lst = [10, 11, 12, 10, 13, 17, 18, 19, 17, 20, 20, 15, 30, 12, 12, 16, 17, 0, 2, 0, 0]  
  
d = Counter(lst)  
print(d)
```

这是一对随机数

用 Counter 计算出每个数据在列表出现多少次，然后用 collections.Counter 返回

```
print("10出现次数=%d" % d[10])
```

同下标指定字典 K 值，然后获取对应的数

这就是表示 12 在列表中出现了 3 次

```
Counter({12: 3, 17: 3, 0: 3, 10: 2, 20: 2, 11: 1, 13: 1, 18: 1, 19: 1, 15: 1, 30: 1, 16: 1, 2: 1})  
10出现次数=2
```

```

from collections import Counter

lst = [10, 11, 12, 10, 13, 17, 18, 19, 17, 20, 20, 15, 30, 12, 12, 16, 17, 0, 2, 0, 0]

d = Counter(lst)
print(d)
print(type(d))
print("10出现次数=%d" % d[10])

yuanzulst = d.most_common(1)
print(yuanzulst)
yuanzulst = d.most_common(2)
print(yuanzulst)
yuanzulst = d.most_common(4)
print(yuanzulst)

```

collections.most_common 函数就是获取 d 列表里面 1 个出现次数最多的数

collections.most_common 函数就是获取 d 列表里面 2 个出现次数最多的数

collections.most_common 函数就是获取 d 列表里面 4 个出现次数最多的数

```

Counter({12: 3, 17: 3, 0: 3, 10: 2, 20: 2, 11: 1, 13: 1, 18: 1, 19: 1, 15: 1, 30: 1, 16: 1, 2: 1})
<class 'collections.Counter'>
10出现次数=2
[(12, 3)]
[(12, 3), (17, 3)]
[(12, 3), (17, 3), (0, 3), (10, 2)]

```

查询多个字典公共键

map 函数使用

map(function , iterable ,)

function: 传入一个写好的函数

iterable: 将该列表或者字典放入**function**指定的函数去计算

```

def square(x):
    return x+1

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

```

准备一个函数

准备一个列表

```

ret = map(square, lst)

```

将列表每一个数放入函数去运算，然后返回生成器，列表有多少个元素，就生成多少个生成器

```

print(ret)
xzz = list(ret)
print(xzz)

```

打印确认是生成器

为了方便观察把生成器转换成列表

输出结果都是列表被 map 指定的函数计算后的结果

```

<map object at 0x0000000001E2DBA8>
[2, 3, 4, 5, 6]

```

如果 map 指定的函数是两个参数，那么就要指定两个列表

```

def square(x, y):
    return x+y

lst1 = [1, 2, 3, 4, 5]
lst2 = [2, 2, 2, 2, 2]
ret = map(square, lst1, lst2)

print(ret)
xzz = list(ret)
print(xzz)

```

map 指定的 function 是两个参数
那么 map 后面传入的 iterable 也必须是两个参数，所以建立两个列表
输出结果是两个列表相对应的元素下标计算的值
 $lst1[0]+lst2[0], lst1[1]+lst2[1], \dots$

<map object at 0x00000000001E1DEF0>
[3, 4, 5, 6, 7]

为了写代码方便你可以直接在 map 里面用 lambda 来表示函数

```

lst1 = [1, 2, 3, 4, 5]

ret = map(lambda x:x+1, lst1)
print(ret)          函数 1 个形参
xzz = list(ret)    传入形参
print(xzz)          进行计算

```

输出结果和上面的 def square(x):x+1 结果是一样的

<map object at 0x000000000006976D8>
[2, 3, 4, 5, 6]

```

lst1 = [1, 2, 3, 4, 5]
lst2 = [2, 2, 2, 2, 2]
ret = map(lambda x,y:x+y, lst1, lst2)

print(ret)
xzz = list(ret)
print(xzz)

```

<map object at 0x0000000000214DC18>
[3, 4, 5, 6, 7]

all 函数使用

all(iterable)

iterable : 只能传入元组或列表

元组里面只要有一个元素为空，就返回False，如果所有元素不为空就返回True

列表里面只要有一个元素为空，就返回False，如果所有元素不为空就返回True

```

ret = all(['a', 'b', 'c', 'd'])    # 列表list, 元素都不为空或0
print(ret) #返回True

ret = all(['a', 'b', '', 'd'])      # 列表list, 存在一个为空的元素
print(ret) #返回False

ret = all(('a', 'b', 'c', 'd'))    # 元组tuple, 元素都不为空或0
print(ret) #返回True

ret = all(('a', 'b', '', 'd'))      # 元组tuple, 存在一个为空的元素
print(ret) #返回False

ret = all((0, 1, 2, 3))           # 元组tuple, 存在一个为0的元素
print(ret) #返回False

```

```

True
False
True
False
False

```

字典判断 in 和 not in 关键字操作方法

key in dict

key: 写入某个字符

dict: 字典

如果key字符在dict字典里面有就返回True

key not in dict

key: 写入某个字符

dict: 字典

如果key字符在dict字典里面有就返回False

```
zidian = {'A':10, 'B':10, 'C':10}
```

```

True
False

```

```
ret = 'A' in zidian #字符A在字典里面有返回True
print(ret)
```

```
ret = 'Z' in zidian #字符Z在字典里面没有返回False
print(ret)
```

not in 做法相反，自己试验

下面进行公共键值提取案例分析

足球比赛统计每一轮球员进球个数

第1轮 {'小明':1, '小红':2, '李四':1}

第2轮 {'小明':2, 'C罗':3, '张三':5}

第3轮 {'小明':7, '梅西':6, '王五':3}

计算出每1轮都有进球的球员这里很明显是小明，看看怎么计算

就是要筛选出每一轮都有没有上一轮相同名字的球员进球

```

zidian1 = {'小明':1, '小红':2, '李四':1}
zidian2 = {'小明':2, 'C罗':3, '张三':5}
zidian3 = {'小明':7, '梅西':6, '王五':3}

hb = [zidian1, zidian2, zidian3]
print(hb)
print(hb[0])
print(hb[1:])
ret =[k for k in hb[0] if all(map(lambda d: k in d, hb[1:]))]
print("筛选结果 = %s" %ret)

```

2. hb[1:]拿出第 1 组的后 1 组字典和 k 进行 in 对比，如果第 2 组里面有小明 lambda 返回列表字符，然后 all 发现是列表字符不为空返回 True

```
[{'小明': 1, '小红': 2, '李四': 1}, {'小明': 2, 'C罗': 3, '张三': 5}, {'小明': 7, '梅西': 6, '王5': 3}]  
{'小明': 1, '小红': 2, '李四': 1}  
[{'小明': 2, 'C罗': 3, '张三': 5}, {'小明': 7, '梅西': 6, '王5': 3}]  
筛选结果 = ['小明']
```

另一种简单的方法求出几个字典里面同名的键值

dict.keys() 字典 keys 函数使用

```
zidian = {'A': 10, 'B': 10, 'C': 10}  
print(zidian)  
ret = zidian.keys() #将字典的键返回给新列表，键对应的数值就扔去  
print(ret)  
{'A': 10, 'B': 10, 'C': 10}  
dict_keys(['A', 'B', 'C'])
```

你看 ret 返回的就只有字典的键了，然后放入列表

```
zidian1 = {'小明': 1, '小红': 2, '李四': 1}  
zidian2 = {'小明': 2, 'C罗': 3, '张三': 5}  
zidian3 = {'小明': 7, '梅西': 6, '王5': 3}
```

```
s1 = zidian1.keys()  
s2 = zidian2.keys()  
print(s1)  
print(s2)  
s = s1 & s2
```

因为只有列表才能用&
交集符号，所以把字典
的键取出来

用交集符号& 筛选出两组列表的相同
数据/字符，返回相同数据

```
dict_keys(['小明', '小红', '李四'])  
dict_keys(['小明', 'C罗', '张三'])  
{'小明'}
```

你看这就筛选出每场比赛同名的进球人员了

```
zidian1 = {'小明': 1, '小红': 2, '李四': 1}  
zidian2 = {'小明': 2, 'C罗': 3, '小红': 5}  
zidian3 = {'小明': 7, '梅西': 6, '小红': 3}  
  
s1 = zidian1.keys()  
s2 = zidian2.keys()  
s3 = zidian3.keys()  
  
s = s1 & s2 & s3  
print(s)
```

我增加了小红每轮
进球同名的成员

```
{'小明', '小红'}
```

取三组列表的交集也是可以的。

让字典保持有序排列

```
d = {} #创建一个空字典  
d['c'] = 1  
d['b'] = 6  
d['a'] = 3  
lst = d.keys() #将字典转换成列表  
print(lst)
```

```
dict_keys(['c', 'b', 'a'])
```

这个字典是根据创建数据顺序有序排列的

但是有时候从网络获取的数据放入字典是无序排列的

```
d = {} #创建一个空字典  
d['c'] = 1  
d['b'] = 6  
d['a'] = 3
```

```
dict_keys(['b', 'c', 'a'])
```

这种 c , b , a 就是无序排列的，和字典创建顺序不一样

collections - OrderedDict 有序字典模块使用

```
from collections import OrderedDict  
  
od = OrderedDict()  
  
od['c'] = 1  
od['b'] = 6  
od['a'] = 3  
lst = od.keys() #将字典转换成列表  
print(lst)
```

导入 collections 里面的 orderedDict 有序字典模块

用 OrderedDict 创建有序的空字典

```
odict_keys(['c', 'b', 'a'])
```

enumerate 函数使用

enumerate(sequence, [start=0])

sequence: 传入需要填加序号的列表，元组，字符串

start: 默认从0开始给列表增加序号，当然可以指定开始值

```
plays = list("abcdefghijklmnopqrstuvwxyz") #用list生成一个有数的列表
print(plays)
ret = enumerate(plays, 2)
print(ret) #enumerate返回的是枚举对象
```

```
lst = list(ret) #用list将其枚举变量进行转换，生成元组列表
print(lst)
```

列表生成没加序号

```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
<enumerate object at 0x0000000002930558>
[(2, 'a'), (3, 'b'), (4, 'c'), (5, 'd'), (6, 'e'), (7, 'f'), (8, 'g')]
```

用 enumerate 给列表加序号

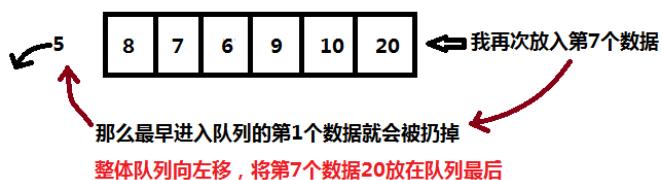
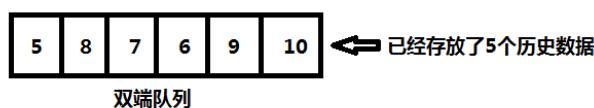
现在我想输入序号就能获取对应的字符

islice 切片函数使用

```
from itertools import islice
val = [0, 8, 6, 5, 10, 20, 30, 7, 6, 3, 2]
print(val)
ret = islice(val, 3, 6)
#islice获取列表从3元素开始到6元素之前也就是5元素的所有数据
#这里是3~5元素下标
print(list(ret))
```

```
[0, 8, 6, 5, 10, 20, 30, 7, 6, 3, 2]
[5, 10, 20]
```

记录用户历史数据



deque 双端队列函数使用

deque(iterator, maxlen)

iterator: 指定队列里面放那种类型的数据，列表，元组，字符串？

maxlen: 指定创建存放多少个元素的队列

```

from collections import deque
q = deque([], 6) # 创建列表类型的队列，队列容量为6
q.append(10) # append是右端入队，appendleft是左端入队
q.append(20)
q.append(30)
q.append(40)
q.append(50)
q.append(60)
print(q)
q.append(70)
print(q)

```

加入队列库

向队列放入 6 个数据没有问题

如果队列数据超过前面申请的 6 个，再向队列放入一个数据，那么就会把队列前面一个数据挤出掉

```

deque([10, 20, 30, 40, 50, 60], maxlen=6)
deque([20, 30, 40, 50, 60, 70], maxlen=6)

```

你看数据 10 被挤出去了，后面加入的 70 数据放在队列最后

这种双端队列就适合做浏览器，历史浏览数据存储。给出一个空间，如果后面的访问超出这个空间就让最早的浏览记录清除掉。

将字符串里不同的分隔符取消掉

```

str = "ab;cd|efg,hijk|lm\opq"
res = str.split(';')
print(res)
res2 = str.split('|')
print(res2)

```

split 每次执行只能写入一种分隔符号

```

['ab', 'cd|efg,hijk|lm\opq']
['ab;cd', 'efg,hijk', 'lm\opq']

```

分隔后默认用逗号区分开

```

str = "ab;cd|efg,hijk|lm\opq"
lst = list(map(lambda s:s.split('|'), str.split(';')))
print(lst)
[[['ab'], ['cd', 'efg,hijk', 'lm\opq']]]

```

然后得到分号筛选的结果，将|号筛选出来

先将分号筛选出来

这样筛选出来是个二维列表

sum 函数列表相加使用

sum(iterable ,start)

iterable: 放入列表，元组，集合

start: 指定相加参数，可以是数字，列表

用 sum 函数将两个列表相加变成一维列表

map 函数在 97 页，split 函数在 30 页

```
str = "ab;cd|efg,hijk|lm\opq"  
lst = list(map(lambda s:s.split(' | '), str.split(';')))  
print(lst)  
  
retlst = sum(lst, [])  
print(retlst)
```

写入[]，指定列表相加

```
[['ab'], ['cd', 'efg,hijk', 'lm\\opq']]  
['ab', 'cd', 'efg,hijk', 'lm\\opq']
```

二维列表变成一维列表了

封装字符串分隔函数

```
str = "ab;cd|efg,hijk|lm\opq"  
  
def my_split(str,seps):  
    res = [str]  
    for sep in seps:  
        t = []  
        list(map(lambda ss:t.extend(ss.split(sep)),res))  
        res = t  
    return res  
  
lst = my_split(str,';|')  
print(lst)
```

如果我传入' ;| '，第1次循环获取分号

在 t 空列表里面增加分号
分隔的字符串

将传入的字符串赋值给 ss

```
['ab', 'cd', 'efg,hijk', 'lm\\opq']
```

这就是分隔后的字符串用逗号隔开

re.split 正则表达式使用，字符串分隔多个符号

```
import re
line = 'aaa bbb ccc;ddd    eee,fff | ggg(hhh) 11.11'
res = re.split(r';|,',line) #按照分号分隔字符串
print(res)

res = re.split(r'\s',line) #按空格分隔字符串
print(res)
res = re.split(r'[ ]',line) #按列表空格分隔字符串
print(res)
res = re.split(r'[\s]',line,1) #列表空格只分隔第一个空格，后面空格不分割
print(res)
res = re.split(r'[,;]',line) #多个符号分隔
print(res)
```

```
['aaa bbb ccc', 'ddd    eee,fff | ggg(hhh) 11.11']
['aaa', 'bbb', 'ccc;ddd', '', '', 'eee,fff', '|', 'ggg(hhh)', '11.11']
['aaa', 'bbb', 'ccc;ddd', '', '', 'eee,fff', '|', 'ggg(hhh)', '11.11']
['aaa', 'bbb ccc;ddd    eee,fff | ggg(hhh) 11.11']
['aaa bbb ccc', 'ddd    eee', 'fff | ggg(hhh) 11.11']
```

所以用正则表达式分隔多符号的字符串最方便

```
import re
str = "ab;cd|efg,hijk|lm\opq"
res = re.split(r'[;|]',str)
print(res)
```

```
['ab', 'cd', 'efg,hijk', 'lm\\opq']
```

查看文件是否存在

endswith 函数用法

```
str = "aaa.py"
ret = str.endswith('.py')
print(ret)

ret = str.endswith('.sh')
print(ret)
```

比如有个文件叫 aaa.py

用 endswith 查看该文件名是否用 py 结尾，如果是返回 True，不是返回 False

True
False

```
str = "aaa.py"
ret = str.endswith('.py')
print(ret)
```

可以用元组的方式传入多个参数，只要满足其中一个参数，返回值就是 True

```
ret = str.endswith(('.sh', '.py'))
print(ret)
```

True
True

这里的文件只要满足.sh 结尾或者.py 结尾都返回 True

修改已有字符串的格式

```
log = "2016-05-21 10:36:26 status unpacked python3\n" \
      "2017-08-31 20:36:56 status unpacked java\n" \
      "2018-06-15 06:07:08 status unpacked C++\n"

print(log)
2016-05-21 10:36:26 status unpacked python3
2017-08-31 20:36:56 status unpacked java
2018-06-15 06:07:08 status unpacked C++
```

这是服务器获取的 log 文件的日志，现在我想把时间 2016-05-21 这一项颠倒过来显示，就是 21-05-2016 这种显示方式，该怎么做？

正则表达式 re.sub 函数使用

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

pattern: 正则表达式，写入字符串模式

repl: 写入要被替换的字符串位置，也可以写函数

string: 填入要被sub处理的字符串

count: 匹配次数，默认全部替换

```
import re
st = "hello 2019"
st = re.sub("([0-9]+)", "danshengou", st)
print(st)

hello danshengou
```

匹配所有连续出现的数字，把 2019 替换成

这就是替换进去的字符

写入要替换的字符串原始变量

替换成功

```

import re
st = "hello aabbbaa"
st = re.sub("(a{2})", "z", st)
print(st)

```

连续出现 2 次的
a，替换成 z

hello zbbz

下面来解决把 2016-05-21 这一项颠倒过来显示，就是 21-05-2016 这种显示方式

```

import re
log = "2016-05-21 10:36:26 status unpacked python3\n" \
      "2017-08-31 20:36:56 status unpacked java\n" \
      "2018-06-15 06:07:08 status unpacked C++\n"
print(log)

ret = re.sub(r'(\d{4})-(\d{2})-(\d{2})', r'\3-\2-\1', log)
print(ret)

```

\d 表示数字，这里指连续出现的 4 个数字

这里指连续出现的 2 个数字

按照第 1 个参数取出数据后，数据是按照 \d 排列的，这里是让它们把数据排列方式按照\3\2\1 排列

2016-05-21 10:36:26 status unpacked python3
 2017-08-31 20:36:56 status unpacked java
 2018-06-15 06:07:08 status unpacked C++

21-05-2016 10:36:26 status unpacked python3
 31-08-2017 20:36:56 status unpacked java
 15-06-2018 06:07:08 status unpacked C++

用横杠表示这 3 组数字是用横杠连接起来的，这样 sub 执行的时候自然能找到文本匹配这种横杠，连续出现数字规则的数据

执行 re.sub 之后，数据是按照\3\2\1 排列了，正则表达式的 r 是表示忽略反斜杠，不加 r 会出问题

第 2 个案例，我们修改中间时间的顺序

```

import re
log = "2016-05-21 10:36:26 status unpacked python3\n" \
      "2017-08-31 20:36:56 status unpacked java\n" \
      "2018-06-15 06:07:08 status unpacked C++\n"
print(log)

ret = re.sub(r'(\d{2}):(\d{2}):(\d{2})', r'\2+\1+\3', log)
print(ret)

```

```
2016-05-21 10:36:26 status unpacked python3  
2017-08-31 20:36:56 status unpacked java  
2018-06-15 06:07:08 status unpacked C++  
  
2016-05-21 [36+10+26] status unpacked python3  
2017-08-31 [36+20+56] status unpacked java  
2018-06-15 [07+06+08] status unpacked C++
```

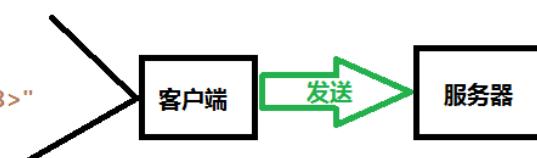
时间顺序已经重新排列了

这就是正则表达式强大之处

UDP 网络传输数据包拼接处理

自定义的 UDP 网络协议数据规则

```
hwDetect:      "<0112>"  
gxDepthBits:  "<32>"  
gxResolution: "<1024x768>"  
fullalpha:    "<1>"  
lodDist:      "<100.0>"
```



["<0112>" , "<32>" , "<1024x768>" , "<1>" , "<100.0>"]
这种‘，’逗号分隔的就是发送的时候做成了5个数据包，但是现在要求发送给服务器的数据先后顺序不能混乱，但是这种‘，’逗号分隔的数据包在 UDP 里面发送，UDP是不分先后顺序的，所以很可能数据包达到服务器顺序错误

"<0112><32><1024x768><1><100.0>"

把多个‘，’逗号分隔的数据包拼接成一个数据包不就得了

```
s1 = 'abcdefg'  
s2 = '12345'  
ret = s1+s2  
print(ret)
```

这是字符串章节讲过的常规
拼接字符串方法

abcdefg12345

下面来处理 UDP 字符串问题

```
str = [ "<0112>" , "<32>" , "<1024x768>" , "<1>" , "<100.0>" ]  
ret = ''  
  
for p in str:  
    ret += p  
  
print(ret)
```

把这些单个‘，’逗号
隔离开的字符串拼接
成1个字符串

<0112><32><1024x768><1><100.0> 执行成功，拼接没有问题

但是这里有个效率问题，只要用到 for 循环处理字符拼接，文本拷贝，都会降低 python 的效率

```
str = ["<0112>", "<32>", "<1024x768>", "<1>", "<100.0>"]
ret = ''  
  
for p in str:  
    ret += p  
    print(ret)
```

我们把每次循环打印出来看看

```
<0112>  
<0112><32>  
<0112><32><1024x768>  
<0112><32><1024x768><1>  
<0112><32><1024x768><1><100.0>
```

你看每次循环都是申请内存，复制数据，粘贴数据，和释放内存，效率很低

如果列表数据很长，那么这种情况就很严重

str.join 函数用法

str.join(iterable)

iterable: 放入可迭代对象，比如字符串，列表，元组

```
str = ["<0112>", "<32>", "<1024x768>", "<1>", "<100.0>"]
ret = ';' .join(str)
print(ret)
```

传入要合并的变量

合并后每个字段用符号分隔开

```
<0112>;<32>;<1024x768>;<1>;<100.0>
```

你看合并成一个字符串了，但是用分号分隔开的，如果想完全没有符号，直接合并成上页那样

```
str = ["<0112>", "<32>", "<1024x768>", "<1>", "<100.0>"]
ret = '' .join(str)
print(ret)
print(type(ret))
```

直接输入空，就是没有符号分隔

```
<0112><32><1024x768><1><100.0>
<class 'str'>
```

这不就对了

字符串左右居中对齐

```
{  
    "lodDist":100.0,  
    "SmallCull":0.04,  
    "DistCull":500.0,  
    "trilinear":40,  
    "xzz":477  
}  
变化后  
"lodDist":100.0,  
"SmallCull":0.04,  
"DistCull":500.0,  
"trilinear":40,  
"xzz":477
```

字符串很整齐

我们就是要将格式混乱的字符串转换成右边这种很整齐的字符串

ljust 左对齐 , rjust 右对齐 , center 居中对齐函数使用

```
str = 'abcd'  
print("====%s====" % str)  
  
ret = str.ljust(20)  
print("====%s====" % ret)  
ret = str.ljust(20, '+')  
print("====%s====" % ret)  
  
ret = str.rjust(20)  
print("====%s====" % ret)  
ret = str.rjust(20, '+')  
print("====%s====" % ret)  
  
ret = str.center(20)  
print("====%s====" % ret)
```

ljust(width)
width: 指定字符串长度
ljust(width[, fillchar])
width: 指定字符串长度
fillchar: 填充什么符号

这里写 20 是直接字符串左移 20 个字符，空的地方用空格填充

这里写 20 也是直接字符串左移 20 个字符，空的地方用+号填充

这里写 20 也是直接字符串右移 20 个字符，空的地方用空格填充

一样的和 ljust 翻过来理解

字符串居中

```
====abcd====  
====abcd=====  
====abcd+++++++=====  
====          abcd====  
====+++++++=abcd====  
====      abcd      ===
```

format 函数使用，左对齐，右对齐，居中

```
str = 'abcd'
print("====%s====" %str)

ret = format(str, '<20') # <20 字符串右对齐偏移20个空格
print("====%s====" %ret)
ret = format(str, '>20') # <20 字符串左对齐偏移20个空格
print("====%s====" %ret)
ret = format(str, '^20') # ^20 字符串居中对齐20个空格
print("====%s====" %ret)
```

```
====abcd====
====abcd          ===
====           abcd===
====      abcd      ===
```

这就是 ljust , rjust , center , format 几种对齐函数的用法

```
{
    "lodDist":100.0,
    "SmallCull":0.04,
    "DistCull":500.0,
    "trilinear":40,
    "xzz":477
}
```



字符串很整齐

现在我们来解决这个案例

```
xdic = {"lodDist":100.0,
         "SmallCull":0.04,
         "DistCull":500.0,
         "trilinear":40,
         "xzz":477}

print(xdic)
{'lodDist': 100.0, 'SmallCull': 0.04, 'DistCull': 500.0, 'trilinear': 40, 'xzz': 477}
```

1. 我们先要让字典键值的字符串宽度一样，那么我们必须先找到字符串最宽的哪一个，然后其余字符串宽度填充成和字符串最宽的那个一样。

```
key = xdic.keys() #keys() 把字典每一个键获取出来
print(key)
```

```
mret = map(len, key) #对每一个键长度取len, 赋值给mret
print(list(mret)) #这里是迭代器所以要转换成list才能看
```

```
mret = map(len, key)
#因为第一次map迭代完了，所以这里max需要使用就得在生成一次
num = max(mret) #获取键字符串，长度里面最长的那个数
print(num)
```

```
dict_keys(['lodDist', 'SmallCull', 'DistCull', 'trilinear', 'xzz'])  
[7, 9, 8, 9, 3]  
9
```

```
for k in xdic:  
    print(k.ljust(num), ':', xdic[k])
```

```
lodDist : 100.0  
SmallCull : 0.04  
DistCull : 500.0  
trilinear : 40  
xzz : 477
```

这样输出就整齐了。

可迭代对象和迭代器的区别

北京:15~20

重庆:30~35

深圳:20~25

.....



网络API抓取天气数据放入list列表变量中

这种方式用户体验就很差，感觉就是点击一下页面，页面就卡死了，等很久页面才正常。

北京:15~20

重庆:30~35

深圳:20~25

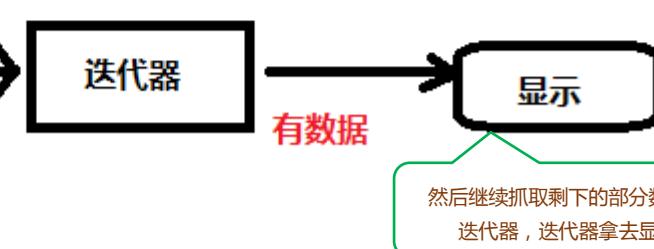


广西:15~20

青岛:20~25

上海:25~30

.....



这种不停的用网络把数据分块抓取，一部分一部分显示，这样效果就很好。

当然这种方法只适用于类似天气这种显示需求的情况。

```

from collections import Iterable, Iterator
lst = [1, 2, 3, 4, 5] # 定义个列表，列表是可迭代对象

for x in lst:
    print(x)

flg = isinstance(lst, Iterable) # 查看列表是不是迭代对象，能返回True
print(flg)
flg = issubclass(list, Iterable) # list列表是否能迭代，能返回True
print(flg)
flg = issubclass(str, Iterable) # 字符串是否能迭代，能返回True
print(flg)
flg = issubclass(dict, Iterable) # 字典是否能迭代，能返回True
print(flg)

```

这就是查看该对象是否是可迭代对象的方法

```

1
2
3
4
5
True
True
True
True
False

```

迭代器是从可迭代对象里面去获取的

```

lst = [1, 2, 3, 4, 5] # 定义个列表，列表是可迭代对象
it = iter(lst)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))

1
2
3
4
5

```

所以迭代器里面的 data next 完了后会报错，就是因为 next 会消耗迭代器内存，最后迭代器内存数据获取完了，内存就会消失。

所以迭代器对象是由可迭代对象经过 iter 生成的。

可迭代对象会占用内存，但是迭代器对象使用之后不会占用内存

下面就来提取天气信息，用迭代器方式展示

```
http://wthrcdn.etouch.cn/weather_mini?city=北京
```

这是一个天气的网站，只能下载天气数据，网页无法显示

```

import requests
url = 'http://wthrcdn.etouch.cn/weather_mini?city=北京'
ret = requests.get(url)
print(ret.text)

```

```
{
    "data": {"yesterday": {"date": "17日星期六", "high": "高温 32℃", "fx": "北风", "low": "低温 19℃", "f1": "<![CDATA[3-4级]]>", "type": "晴"}, "city": "北京", "forecast": [{"date": "18日星期天", "high": "高温 32℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 21℃", "fengxiang": "北风", "type": "晴"}, {"date": "19日星期一", "high": "高温 31℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 23℃", "fengxiang": "南风", "type": "多云"}, {"date": "20日星期二", "high": "高温 27℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 20℃", "fengxiang": "南风", "type": "小雨"}, {"date": "21日星期三", "high": "高温 31℃", "fengli": "<![CDATA[3-4级]]>", "low": "低温 21℃", "fengxiang": "西北风", "type": "多云"}, {"date": "22日星期四", "high": "高温 30℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 22℃", "fengxiang": "北风", "type": "晴"}], "ganmao": "各项气象条件适宜，发生感冒机率较低。但请避免长期处于空调房间中，以防感冒。", "wendu": "28"}, "status": 1000, "desc": "OK"}
}
```

这就是打印获取的天气数据,其实是一个 json 格式的字符串

```

import requests
url = 'http://wthrcdn.etouch.cn/weather_mini?city=北京'
ret = requests.get(url)
print(ret.text)
print("====")
res = ret.json()
print(res)

```

我们将 json 字符串转换成字典

```
{
    "data": {"yesterday": {"date": "17日星期六", "high": "高温 32℃", "fx": "北风", "low": "低温 19℃", "f1": "<![CDATA[3-4级]]>", "type": "晴"}, "city": "北京", "forecast": [{"date": "18日星期天", "high": "高温 32℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 21℃", "fengxiang": "北风", "type": "晴"}, {"date": "19日星期一", "high": "高温 31℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 23℃", "fengxiang": "南风", "type": "多云"}, {"date": "20日星期二", "high": "高温 27℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 20℃", "fengxiang": "南风", "type": "小雨"}, {"date": "21日星期三", "high": "高温 31℃", "fengli": "<![CDATA[3-4级]]>", "low": "低温 21℃", "fengxiang": "西北风", "type": "多云"}, {"date": "22日星期四", "high": "高温 30℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 22℃", "fengxiang": "北风", "type": "晴"}], "ganmao": "各项气象条件适宜，发生感冒机率较低。但请避免长期处于空调房间中，以防感冒。", "wendu": "28"}, "status": 1000, "desc": "OK"}
=====

{"data": {"yesterday": {"date": "17日星期六", "high": "高温 32℃", "fx": "北风", "low": "低温 19℃", "f1": "<![CDATA[3-4级]]>", "type": "晴"}, "type": "晴", "city": "北京", "forecast": [{"date": "18日星期天", "high": "高温 32℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 21℃", "fengxiang": "北风", "type": "晴"}, {"date": "19日星期一", "high": "高温 31℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 23℃", "fengxiang": "南风", "type": "多云"}, {"date": "20日星期二", "high": "高温 27℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 20℃", "fengxiang": "南风", "type": "小雨"}, {"date": "21日星期三", "high": "高温 31℃", "fengli": "<![CDATA[3-4级]]>", "low": "低温 21℃", "fengxiang": "西北风", "type": "多云"}, {"date": "22日星期四", "high": "高温 30℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 22℃", "fengxiang": "北风", "type": "晴"}], "ganmao": "各项气象条件适宜，发生感冒机率较低。但请避免长期处于空调房间中，以防感冒。", "wendu": "28"}, "status": 1000, "desc": "OK"}
}
```

现在转换成字典了，有些是字典的 key 对应的值是列表，所以看起来很乱。

```

ret = requests.get(url)

res = ret.json()
print(res)

out = ret.json()['data']['city'] #根据字典的key输出对应的值
print("====")
print(out)

```

```
{
    "data": {"yesterday": {"date": "17日星期六", "high": "高温 32℃", "fx": "北风", "low": "低温 19℃", "f1": "<![CDATA[3-4级]]>", "type": "晴"}, "city": "北京", "forecast": [{"date": "18日星期天", "high": "高温 32℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 21℃", "fengxiang": "北风", "type": "晴"}, {"date": "19日星期一", "high": "高温 31℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 23℃", "fengxiang": "南风", "type": "多云"}, {"date": "20日星期二", "high": "高温 27℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 20℃", "fengxiang": "南风", "type": "小雨"}, {"date": "21日星期三", "high": "高温 31℃", "fengli": "<![CDATA[3-4级]]>", "low": "低温 21℃", "fengxiang": "西北风", "type": "多云"}, {"date": "22日星期四", "high": "高温 30℃", "fengli": "<![CDATA[<3级]]>", "low": "低温 22℃", "fengxiang": "北风", "type": "晴"}], "ganmao": "各项气象条件适宜，发生感冒机率较低。但请避免长期处于空调房间中，以防感冒。", "wendu": "29"}, "status": 1000, "desc": "OK"}
=====

北京

```

获取 data 字典里面的 key 是 city 对应的值，这个值就是北京。

所以为什么要把网络请求数据转换成字典，就是因为这样才好用 python 字典语法筛选数据。

我们用迭代器来获取天气数据

```
from collections.abc import Iterable, Iterator
import requests

class WeatherIterator(Iterator): #迭代器
    def __init__(self, city):
        self.city = city #把城市变量赋值给内部定义的city变量
        self.index = 0 #定义个index变量

    def __next__(self):
        if self.index == len(self.city):
            #发现index经过下面累加后==city大小表示迭代完了
            raise StopIteration #抛出异常
        dcity = self.city[self.index] #将当前访问的城市赋值给变量
        self.index += 1 #每次执行next发现没有迭代完就+1
        return self.get_weather(dcity) #调用类里面的get_weather共有函数

    def get_weather(self, xcity):
        url = 'http://wthrcdn.etouch.cn/weather_mini?city=' + xcity
        #这里城市值xcity可以随意改变
        ret = requests.get(url)
        data = ret.json()['data']['forecast'][0] #获取json里面指定城市的当天气温
        return xcity, data['high'], data['low']

w = WeatherIterator(['北京', '上海', '广州'])

for x in w:
    print(x)
```

```
('北京', '高温 32℃', '低温 21℃')
('上海', '高温 31℃', '低温 25℃')
('广州', '高温 32℃', '低温 24℃')
```

这就是三个城市的气温，可以无限循环刷新获取。

但是我无限循环刷新获取试试

```
w = WeatherIterator(['北京', '上海', '广州'])

for x in w:
    print(x)
    print("-----")
    for x in w:
        print(x)

('北京', '高温 32℃', '低温 21℃')
('上海', '高温 32℃', '低温 26℃')
('广州', '高温 30℃', '低温 25℃')
-----
```

```
class WeatherIterable(Iterable): #可迭代对象
    def __init__(self, city):
        self.city = city
    def __iter__(self):
        return WeatherIterator(self.city)
w = WeatherIterable(['北京', '上海', '广州'])
for x in w:
    print(x)
print("-----")
for x in w:
    print(x)
```

```
('北京', '高温 32℃', '低温 21℃')
('上海', '高温 32℃', '低温 26℃')
('广州', '高温 30℃', '低温 25℃')
-----
('北京', '高温 32℃', '低温 21℃')
('上海', '高温 32℃', '低温 26℃')
('广州', '高温 30℃', '低温 25℃')
```

这样就可以无限的创建迭代器，消耗迭代器，创建迭代器，消耗迭代器这样循环下去。

可以无限刷新天气数据，而且不占用内存。

如何读取二进制文件，wav 文件案例

Endian	File offset (bytes)	Field name	Field Size (bytes)	
big	0	ChunkID	4	The "RIFF" chunk descriptor
little	4	ChunkSize	4	
big	8	Format	4	
big	12	Subchunk1ID	4	
little	16	Subchunk1Size	4	
little	20	AudioFormat	2	
little	22	NumChannels	2	The "fmt" sub-chunk
little	24	SampleRate	4	
little	28	ByteRate	4	describes the format of the sound information in the data sub-chunk
little	32	BlockAlign	2	
little	34	BitsPerSample	2	
big	36	Subchunk2ID	4	The "data" sub-chunk
little	40	Subchunk2Size	4	
little	44	data	Subchunk2Size	Indicates the size of the sound information and contains the raw sound data



我要打开，读取，操作桌面的 将进酒.wav 文件。

```
f = open(r"D:\02 我的文档\Desktop\将进酒.wav",'rb') # rb 表示打开读取二进制文件
info = f.read(44) #读取wav文件前面44字节数据
print(info)

b'RIFF\xbc\xW\x03WAVEfmt \x10\x00\x00\x01\x00\x02\x00D\xac\x00\x00\x10\xb1\x02\x00\x00'
```

打开后输出二级制数据，因为 wav 文件是二进制文件

如何将这些二进制数据解析出来呢？

struct.unpack 函数使用

struct.pack(fmt, v1, v2...) //返回v1, v2对应的二进制数

fmt: 要求转换的格式

→ 返回的数据用什么方式

v1, v2: 传入的参数要和fmt格式一致

存储?下面可以选择

= 本机

> 大端

< 小段

然后再次在符号后面填入

I : v1, v2传入的是unsigned int数

L : v1, v2传入的是unsigned long数

更多符号请查表

```

import struct
    格式填 >I，就是返回值是大端存储，v1, v2 填入的是 unsigned int 十进制数

ret = struct.pack('>I', 4042322160)
print(ret)

res = struct.unpack('>I', b'\xf0\xf0\xf0\xf0')
print(res)
    填入要转换的十进制数
    将十进制转成了二进制
    将二进制转成 python 数据类型
b'\xf0\xf0\xf0\xf0'
(4042322160,) ←

```

```

import struct

f = open(r"D:\02 我的文档\Desktop\将进酒.wav", 'rb') # rb 表示打开读取二进制文件
info = f.read(44) # 读取 wav 文件前面 44 字节数据
print(info)
    'h' 表示获取的数据只占 2 字
    声道数在 22 字节和 23 字节，所以我们切片获取两字节声道数

ret = struct.unpack('h', info[22:24]) # 获取声道数
print(ret)
    采样率是 4 字节，所以用 "i" 格式来获取，i 表示 int，然后切片 24 到 27 的数据

ret = struct.unpack('i', info[24:28]) # 获取采样率
print(ret)
    位宽 16，两字节，切片 34,35

ret = struct.unpack('h', info[34:36]) # 位宽
print(ret)

```

endian	File offset (bytes)	field name	F
big	0	ChunkID	
little	4	ChunkSize	
big	8	Format	
big	12	Subchunk1ID	
little	16	Subchunk1Size	
big	20	AudioFormat	
little	22	NumChannels	
little	24	SampleRate	
little	28	ByteRate	
little	32	BlockAlign	
little	34	BitsPerSample	
big	36	Subchunk2ID	
little	40	Subchunk2Size	
little	44	data	

b'RIFF\xbcxW\x03WAVEfmt \x10\x00\x00
(2,) 声道数
(44100,) 采样率
(16,) 位宽

一般不写 >,<,= 符号，那么默认存放就是小端模式。

大端小端模式请查阅我的嵌入式文档

如何处理 wav 头 44 字节后面的 data 音频数据流

array 函数使用

array.array 静态分配方法

array.array(typecode , [initial])

typecode: 表示要申请数组里面的元素类型是什么，

'i' 表示int , 'I' 表示unsigned int

'f' 表示float , 更多的查百度

[initial]: 初始化数组元素个数

```

import array
buf = array.array('i', [10, 23, 45])
print(buf)
print(type(buf))
print(buf[0])
print(buf[1])
print(buf[2])

```

```
array('i', [10, 23, 45])
<class 'array.array'>
10
23
45
```

array.array动态分配方法

```
array.array( typecode , [initial] )
```

typecode: 表示要申请数组里面的元素类型是什么，

'i' 表示int , 'I' 表示unsigned int

'f' 表示float , 更多的查百度

[initial]: 初始化数组元素个数

例子buf = array.array ('h' , []) //这就是动态分配

一个空间给buf变量，但是是多大的空间呢？要根据后

面赋值确定。这里的 'h' 就是存放short类型的数据

怎么使用动态数组，看下面案例

解析 wav 文件取消 wav 头 44 字节后面的音频数据

```
import array

f = open(r"D:\02 我的文档\Desktop\将进酒.wav",'rb') # rb 表示打开读取二进制文件
info = f.read(44) #读取wav文件前面44字节数据
print(info)

f.seek(0,2)
size = f.tell()
print("文件总大小 = %d" %size)

datalen = (f.tell()-44)
print("data数据长度 = %d" %datalen)
#每个数据是8位的，这个data长度按照8位算的，所以下面还要将数据按照位宽16位字节来计算

databytelen = int((f.tell()-44)/2)
print("data数据字节长度 = %d" %databytelen) #这就是有多少个16位的数据
print(type(databytelen)) #int(float) 将float转换成int

buf = array.array('h',[]) #生成一个缓冲区
```

这里就是申请了一个空的动态数组

```
f.seek(44) #上面将文件指针指到了文件结尾，现在把指针指向44字节位置，因为后面就是data段
buf.fromfile(f,databytelen)
```

fromfile 就是将文件里面数据传给缓存 buf，这样 buf 就自动分配有大小了。语法 fromfile(文件，大小)

```
print(len(buf))
```

你会发现 buf 分配的大小和前两行计算出 data 数据大小一样，
这就证明 fromfile 动态分配内存成功

```
print(buf[5])
print(buf[50])
print(buf[20])
```

```
b'RIFF\xbc\xW\x03WAVEfmt \x10\x00\x00\x00\x01'
文件总大小 = 56064196
data数据长度 = 56064152
data数据字节长度 = [28032076]
<class 'int'>
[28032076]
-2
-1
0
```

如果你觉得缓存 buf 没有收到数据，可以把 buf 打印出来看下

```
f.seek(44) #上面将文件指针指到了文件结尾，现在把指针
buf.fromfile(f, databytelen)

print(len(buf))

print(buf) #输出buf里面的每个short 16位音频数据
```

```
main x

6315, -1981, 8294, 127, 11207, 2280, 10478, 520, 7000, -2367, 5630, -1016, 58
-1848, -842, -3804, -3012, -3586, -3652, -4229, -4731, -3378, -5557, -1552, -
-1479, -12580, -385, -11741, -742, -12673, -1061, -13325, 663, -11170, 2066,
8737, -4282, 5634, -5127, 225, -8228, -4956, -11132, -9160, -13640, -9789, -1
-15122, -9732, -12311, -9137, -9428, -7780, -9204, -6047, -11612, -6531, -151
```

事实证明 buf 收到文件的数据的。

array 动态数组和列表的区别就是 列表是固定分配长度的数组 而且里面的元素类型是可以不一样的。但是 array 分配的动态数组，里面的元素类型必须全部一样，这样处理动态数组的效率比列表高。

临时文件使用



这种情况用临时文件保存数据是最好的方法，你一旦执行close，文件就自动删除了

TemporaryFile 创建的临时文件不会在目录下显示文件

```
from tempfile import TemporaryFile #TemporaryFile创建的临时文件没有名字

f = TemporaryFile(mode='w+b') #创建一个临时文件f, 该文件没有名字
f.write(b"123456789"*1000)
#写入临时文件的数据要用前缀b, 表示转换成二进制写入, 前缀u就是unicode码写入

f.seek(0) #从文件最开始位置读取数据, 这里要将指针移动到文件开头
ret = f.read(5) #先读取文件5个数据显示, 这样节省内存
print(ret)
ret = f.read(5) #再读取文件后5个数据
print(ret)
ret = f.read(50) #再接着读取文件后50个数据
print(ret)
f.close() #关闭文件, 自动销毁临时文件

b'12345'
b'67891'
b'23456789123456789123456789123456789123456'
```

为什么不一次读取 1000 个数据，这是因为
一次性读 1000 个数据太占用内存，所以改
成每次需要处理几个数据就读几个，轮着
来，把 1000 个数据分次数读完

挨着挨着把临时文件数据读出来，前面这个 b 自己想办法解决掉。

NamedTemporaryFile 创建的对象，会在目录下显示临时文件名

```
from tempfile import NamedTemporaryFile
#NamedTemporaryFile创建的临时文件在目录中有文件产生

f = NamedTemporaryFile(mode='w+b')
print(f.name) #显示该临时文件在系统下的文件路径
f.write(b"123456789"*1000)
f.seek(0)
ret = f.read(8)
print(ret)
ret = f.read(8)
print(ret)
f.close()

C:\Users\xzz1346\AppData\Local\Temp\tmpfpbkvw22
b'12345678'
b'91234567'
```

当 close 之后，临时文件销毁

如何读写 json 格式的数据

Json.dumps 函数使用

```
import json  
lst = [1, 2, 'abcde', {'name': 'xzz', 'age': '28'}]  
js = json.dumps(lst) Json.dumps 将列表字典转换成 json 格式的字符串  
print(type(js))  
print(js)  
  
<class 'str'>  
[1, 2, "abcde", {"name": "xzz", "age": "28"}] 你看输出字符串带双引号，这就是 json
```

格式的字符串

```
import json  
zidian = {'b': None, 'a': 5, 'c': "abc"} 写一个字典  
js = json.dumps(zidian)  
print(type(js)) 转成 json 后 None 变成 null 正确  
print(js)  
  
<class 'str'>  
{"b": null, "a": 5, "c": "abc"}
```

```
<class 'str'>  
{"b": null, "a": 5, "c": "abc"}
```

转换成 json 格式的字符串，发现这些空格没有意义，想直接消除掉

```
import json  
zidian = {'b': None, 'a': 5, 'c': "abc"}  
js = json.dumps(zidian, separators=[',', ':'])  
print(type(js)) 转换 json 的时候加入 separators 参数，指定消除符号的格式  
print(js)  
  
<class 'str'>  
{"b":null,"a":5,"c":"abc"} 这样就消除无用的空格了
```

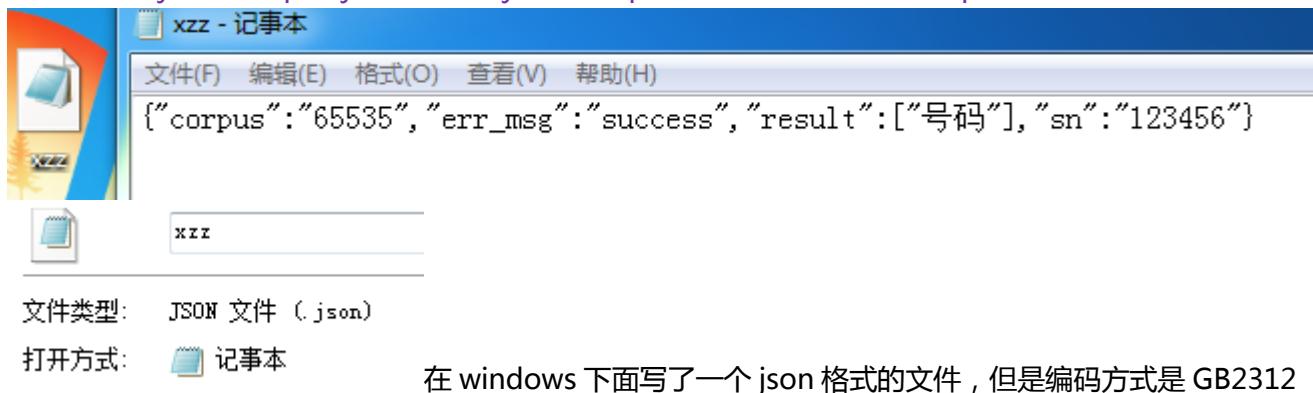
Json.loads 函数用法

```
import json
zidian = {'b':None, 'a':5, 'c':"abc"}
js = json.dumps(zidian, separators=[',', ' :'])
print(js)

py = json.loads(js) #将json格式的字符转换成python需要的字典或者其它格式
print(py)
```

{ "b":null,"a":5,"c":"abc"}
{'b': None, 'a': 5, 'c': 'abc'} 将 json 格式转换成了字典

Json.load , json.dump 和 json.loads , json.dumps 的区别就是 load , dump 是用来解析文件的



```
import json
f = open(r"D:\02 我的文档\Desktop\xzz.json", encoding='GB2312')

ret = json.load(f) load 就是将 json 文件解析为  
python 字典格式
print(ret)

str=ret["corpus"] #返回值是字符串
print("获取corpus数据 = %s" %str)

{'corpus': '65535', 'err_msg': 'success', 'result': ['号码'], 'sn': '123456'}
获取corpus数据 = 65535
```

如何读写 excel 文件

	A	B	C	D	E	F	G
1	类型	星期1	星期2	星期3	星期4	星期5	
2	数据1	10	20	30	40	50	
3	数据2	100	200	300	400	500	
4	数据3	1000	2000	3000	4000	5000	
5	数据4	10000	20000	30000	40000	50000	
6	数据5	100000	200000	300000	400000	500000	
7							

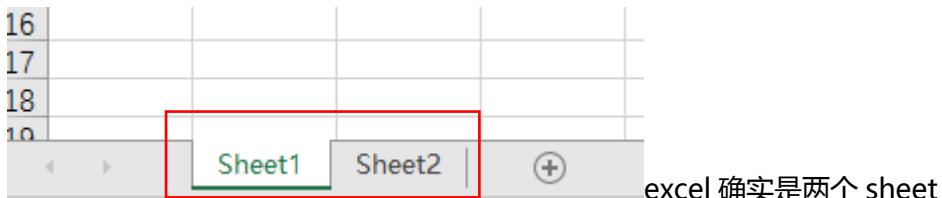
我们读取 excel 数据，然后处理数据后又写会该 excel

需要安装第三方库，xlrd，xlwt

```
import xlrd #读取excel内容的库
book = xlrd.open_workbook(r"D:\02 我的文档\Desktop\xzz.xlsx", 'r')
ret = book.sheets()#获取excel有多少个sheet表
print(ret)

[<xlrd.sheet.Sheet object at 0x000000000002D68CF8>, <xlrd.sheet.Sheet object at 0x000000000002D5CFD0>]
```

从打印结果 excel 应该有两个 sheet，因为这里出现了两个地址



```
import xlrd #读取excel内容的库
book = xlrd.open_workbook(r"D:\02 我的文档\Desktop\xxx.xlsx")
ret = book.sheets()#获取excel有多少个sheet表
print(ret)

sheets = book.sheet_by_index(0) #获取sheet1的对象
row = sheets.nrows
print("sheet1行数 = %d" %row)

sheets2 = book.sheet_by_index(1) #获取sheet2的对象
row = sheets2.nrows
print("sheet2行数 = %d" %row)
```

赋值 0 就是获取 excel 的 sheet1 对象

因为只有 6 行数据，所以获取行数为 6

赋值 1 就是获取 excel 的 sheet2 对象

```
[<xlrd.sheet.Sheet obj  
sheet1行数 = 6  
sheet2行数 = 2]
```

```
sheets = book.sheet_by_index(0) #获取sheet1的对象  
col = sheets.ncols #获取列个数  
print("sheet1列数 = %d" %col)  
  
sheets2 = book.sheet_by_index(1) #获取sheet2的对象  
col = sheets2.ncols #获取列个数  
print("sheet2列数 = %d" %col)
```

```
[<xlrd.sheet.Sheet  
sheet1列数 = 6  
sheet2列数 = 3]
```

这是 excel 的列数

```
sheets = book.sheet_by_index(0) #获取sheet1的对象  
col = sheets.ncols #获取列个数  
print("sheet1列数 = %d" %col)  
  
cell = sheets.cell(0, 0)  
print("0行0列数据类型是 : %s" %cell) #第0行0列数据是文本类型  
print("0行0列数据类型编号是 : %s" %cell.ctype) #文本类型数字是1  
print("0行0列数据是 : %s" %cell.value) #获取excel 0行0列表格的值
```

```
[<xlrd.sheet.Sheet object at 0  
sheet1列数 = 6  
0行0列数据类型是 : text:'类型'  
0行0列数据类型编号是 : 1  
0行0列数据是 : 类型
```

	A	B	C	D	E	F	
1	类型	星期1	星期2	星期3	星期4	星期5	
2	数据1	10	20	30	40	50	
3	数据2	100	200	300	400	500	
4	数据3	1000	2000	3000	4000	5000	
5	数据4	10000	20000	30000	40000	50000	
6	数据5	100000	200000	300000	400000	500000	
7							
8							

```
cell = sheets.cell(1, 1) #访问1行1列  
print("1行1列数据类型是 : %s" %cell) #第1行1列数据是数字类型  
print("1行1列数据类型编号是 : %s" %cell.ctype) #数字类型编号为2  
print("1行1列数据是 : %s" %cell.value) #获取excel 1行1列表格的值
```

```

[<xlrdf.sheet.Sheet object at 0x0000000002E8A80>
sheet1列数 = 6
1行1列数据类型是 : number:10.0
1行1列数据类型编号是 : 2
1行1列数据是 : 10.0

```

A	B	C	D	E	F	
1	类型	星期1	星期2	星期3	星期4	星期5
2	数据1	10	20	30	40	50
3	数据2	100	200	300	400	500
4	数据3	1000	2000	3000	4000	5000
5	数据4	10000	20000	30000	40000	50000
6	数据5	100000	200000	300000	400000	500000
7						
8						

```

cell = sheets.row(1) #访问一行数据
print(cell)
cell = sheets.row(2) #访问一行数据
print(cell)

```

```

[text:'数据1', number:10.0, number:20.0, number:30.0, number:40.0, number:50.0]
[text:'数据2', number:100.0, number:200.0, number:300.0, number:400.0, number:500.0]

```

创建个 excel 文件，向里面写数据

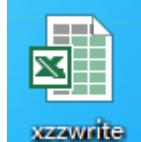
```
import xlwt #向excel写数据的库
```

```

wbook = xlwt.Workbook(encoding = 'utf-8') #创建一个excel文件，编码方式是utf8
wsheet = wbook.add_sheet('sheet1') #给excel添加sheet表
wsheet.write(0,0,'姓名') #给xzzsheet1表增加单元格，我加在0行0列，赋值为字符串

wbook.save(r"D:\02 我的文档\Desktop\xzzwrite.xls") #保持后缀一定是xls而不是xlsx

```



A	B
1	姓名
2	
3	

这就成功写入了数据

下面我再多写几个

```
import xlwt #向excel写数据的库
```

```

wbook = xlwt.Workbook(encoding = 'utf-8') #创建一个excel文件，编码方式是utf8
wsheet = wbook.add_sheet('sheet1') #给excel添加sheet表
wsheet.write(0,0,'姓名') #给xzzsheet1表增加单元格，我加在0行0列，赋值为字符串
wsheet.write(1,1,500) #给xzzsheet1表增加单元格，我加在1行1列，赋值为数值
wsheet.write(2,2,36.3) #给xzzsheet1表增加单元格，我加在2行2列，赋值为小数

wbook.save(r"D:\02 我的文档\Desktop\xzzwrite.xls") #保持后缀一定是xls而不是xlsx

```

A	B	C
1	姓名	
2		500
3		36.3
4		

数据写入正确。如果要重复执行，需要删除原有的 excel 再执行生成新 excel