目录

[基础篇 1](#_Toc462912577)

[语法 1](#_Toc462912578)

[数据类型 2](#_Toc462912579)

[运算 10](#_Toc462912580)

[流程控制 12](#_Toc462912581)

[序列 14](#_Toc462912582)

[字典 20](#_Toc462912583)

[模块 包 23](#_Toc462912584)

[InPut/OutPut 24](#_Toc462912585)

[序列化 29](#_Toc462912586)

[正则 30](#_Toc462912587)

[函数 38](#_Toc462912588)

[异常处理 51](#_Toc462912589)

[类（面向对象） 55](#_Toc462912590)

[1 面向对象技术 55](#_Toc462912591)

[进阶篇 60](#_Toc462912592)

[网络编程 60](#_Toc462912593)

[Socket 60](#_Toc462912594)

[SocketServer 70](#_Toc462912595)

[Twisted 80](#_Toc462912596)

[进程、线程 80](#_Toc462912597)

[线程 80](#_Toc462912598)

# 基础篇

## 语法

* 缩进统一
* 变量

标识符的第一个字符必须是字母表中的字母(大写或者小写)或是一个下划线(‘\_’);

标识符名称的其他部分可以由字母(大写或小写),下划线(‘\_’)或数字(0-9)组成;

标识符名称对大小写敏感;

变量的命名:书写规范可读性,(user\_name).

例子：

>>> name='song'

>>> name1=name

>>> print name,name1

song song

>>> id(name)

43830248L

>>> id(name1)

43830248L

>>> name='bing'

>>> print name,name1

bing song

>>> id(name)

43830328L

>>> id(name1)

43830248L

>>> name2='song'

>>> id(name2)

43830248L

>>>

>>> a,b,c,d='spam'

>>> a

's'

>>> b

'p'

>>> c

'a'

>>> d

'm'

>>>

>>> a,b,c='spam'

Traceback (most recent call last):

File "<pyshell#29>", line 1, in <module>

a,b,c='spam'

ValueError: too many values to unpack (expected 3)

>>>

>>> string='spam'

>>> a,b,c=string[0],string[1],string[2:]

>>> a

's'

>>> b

'p'

>>> c

'am'

>>> seq=[1,2,3,4]

>>> seq

[1, 2, 3, 4]

>>> a,\*b=seq

>>> a

1

>>> b

[2, 3, 4]

>>> \*a,b=seq

>>> a

[1, 2, 3]

>>> b

4

>>> a,\*b,c=seq

>>> a

1

>>> b

[2, 3]

>>> c

4

>>>

>>> a,\*b='spam'

>>> a

's'

>>> b

['p', 'a', 'm']

>>> a,\*b,c='spam'

>>> a

's'

>>> b

['p', 'a']

>>> c

'm'

>>>

* id()查看变量内存位置

通过上例子查看变量内存地址:

当变量name被赋值字符串song的时候,在内存中创建变量’name’,内存地址为45599800L,同时把变量name重新赋值给变量name1,这时候name1的内存地址和name一样,此时修改变量name的值,重新查看name和name1的内存地址已经发生变化,修改内存存储空间,得出结果是在0-255直接的数值python虚拟机自动做优化,把相同的内容指定到相同的内存空间.

## 数据类型

数据类型划分

* 按特征划分

布尔型:

True False

整型:

长整型 标准整型

分整型:

双精度浮点型 复数

序列类型:

字符串 元组(tuple) 列表(list)

映像类型:

字典(dict)

集合类型:

可变集合(set) 不可变集合(frozenset)

* 按可变性划分

可哈希的,不可变数据类型:

数字类型 不可变集合(frozenset) 字符串(str) 元组(tuple)

可变数据类型:

字典(dict) 列表(list) 可变集合(set)

* 查看属于哪种类型type()

例子:

>> name='songbing'

>>> age=26

>>> wage=5000.02

>>> type(name)

<type 'str'>

>>> type(age)

<type 'int'>

>>> type(wage)

<type 'float'>

>>> list\_t=[1,2,3,4]

>>> type(list\_t)

<type 'list'>

>>> tunple\_t=(1,2,3)

>>> type(tunple\_t)

<type 'tuple'>

>>> dict\_t={'name':'song','age':26}

>>> type(dict\_t)

<type 'dict'>

>>>

* 字符串操作方法

在python有各种各样的string操作函数。在历史上string类在python中经历了一段轮回的历史。在最开始的时候，python有一个专门的string的module，要使用string的方法要先import，但后来由于众多的python使用者的建议，从python2.0开始， string方法改为用S.method()的形式调用，只要S是一个字符串对象就可以这样使用，而不用import。同时为了保持向后兼容，现在的 python中仍然保留了一个string的module，其中定义的方法与S.method()是相同的，这些方法都最后都指向了用S.method ()调用的函数。要注意，S.method()能调用的方法比string的module中的多，比如isdigit()、istitle()等就只能用 S.method()的方式调用。

对一个字符串对象，首先想到的操作可能就是计算它有多少个字符组成，很容易想到用S.len()，但这是错的，应该是len(S)。因为len()是内置函数，包括在\_\_builtin\_\_模块中。python不把len()包含在string类型中，乍看起来好像有点不可理解，其实一切有其合理的逻辑在里头。len()不仅可以计算字符串中的字符数，还可以计算list的成员数，tuple的成员数等等，因此单单把len()算在string里是不合适，因此一是可以把len()作为通用函数，用重载实现对不同类型的操作，还有就是可以在每种有len()运算的类型中都要包含一个len()函数。 python选择的是第一种解决办法。类似的还有str(arg)函数，它把arg用string类型表示出来。

* 字符串大小写转换

>>> s\_str='this IS test'

转换为小写:

>>> print s\_str.lower()

this is test

>>>

转换为大写:

>>> print s\_str.upper()

THIS IS TEST

>>>

大小写互换:

>>> print s\_str.swapcase()

THIS is TEST

>>>

字符串的首字母大写:

>>> print s\_str.capitalize()

This is test

>>>

字符串中所有单词首字母大写:

需要导入string模块

>>> import string

>>> print string.capwords(s\_str)

This Is Test

>>>

# capwords这个模块中的方法。它把s\_str用split()函数分开，然后用capitalize()把首字母变成大写，最后用join()合并到一起。

字符串中所有单词首字母大写:

>>> print s\_str.title()

This Is Test

>>>

* 字符串判断:

S.isalnum() #是否全是字母和数字，并至少有一个字符

S.isalpha() #是否全是字母，并至少有一个字符，不包含空格特殊字符

S.isdigit() #是否全是数字，并至少有一个字符

S.isspace() #是否全是空白字符，并至少有一个字符

S.islower() #S中的字母是否全是小写

S.isupper() #S中的字母是否全是大写

S.istitle() #S是否是首字母大写的

* 字符串格式化:

S.ljust(width,[fillchar])

#输出width个字符，S左对齐，不足部分用fillchar填充，默认的为空格，

fillcachar可以是任意值。

>>> s\_str.ljust(18)

'this IS test '

>>> print s\_str.ljust(18,'0')

this IS test000000

>>>

S.rjust(width,[fillchar]) #右对齐,不足部分用fillchar填充，默认的为空格。

>>> print s\_str.rjust(18)

this IS test

>>> print s\_str.rjust(18,'0')

000000this IS test

>>>

S.center(width, [fillchar]) #中间对齐 ,不足部分用fillchar填充，默认的为空格。

>>> print s\_str.center(18)

this IS test

>>> print s\_str.center(18,'0')

000this IS test000

>>>

S.zfill(width) #把S变成width长，并在右对齐，不足部分只用0补足

>>> print s\_str.zfill(20)

00000000this IS test

>>>

* 字符串搜索替换:

S.find(substr, [start, [end]])

#返回S中出现substr的第一个字母的标号，如果S中没有substr

则返回-1。

start和end作用就相当于在S[start:end]中搜索

>>> print s\_str.find('is')

2

>>> print s\_str.find('a')

-1

>>>

S.index(substr, [start, [end]])

#与find()相同，只是在S中没有substr时，如果没有匹配的字符串会报异常

>>> print s\_str.index('a')

Traceback (most recent call last):

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

ValueError: substring not found

>>> print s\_str.index('is')

2

>>>

S.rfind(substr, [start, [end]])

#返回S中最后出现的substr的第一个字母的标号，如果S中没有substr则返回-1，也就是说从右边算起的第一次出现的substr的首字母标号。

>>> print s\_str.index('is')

2

>>> print s\_str.rfind('a')

-1

>>> print s\_str.rfind('is')

2

>>> print s\_str.rfind('s')

10

>>>

S.rindex(substr, [start, [end]])

#返回substr在字符串中最后出现的位置，如果没有匹配的字符串会报异常。

>>> print s\_str.rindex('d')

Traceback (most recent call last):

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

ValueError: substring not found

>>> print s\_str.rindex('s')

10

>>> print s\_str.rindex('is')

2

>>>

S.count(substr, [start, [end]]) #计算substr在S中出现的次数。

>>> print s\_str.count('s')

2

>>> print s\_str.count('is')

1

>>> print s\_str.count('a')

0

>>>

>>> s\_str='this IS is test'

>>> s\_str.count('is')

2

S.replace(oldstr, newstr, [count])

#把S中的oldstar替换为newstr，count为替换次数。这是替换的通用

形式，还有一些函数进行特殊字符的替换 ，如果不指定替换次数，默认替换所有。

>>> print s\_str.replace('is','at')

that IS test

>>>

重新定义字符串s\_str

>>> s\_str=' 8888this is test8888'

>>> print s\_str

8888this is test8888

>>>

S.strip([chars])

#把S中前后chars中有的字符全部去掉，可以理解为把S前后chars替换为None。

>>> print s\_str.strip()

8888this is test8888

>>> print s\_str.strip('8')

8888this is test

>>> print s\_str.strip().strip('8')

this is test

>>>

S.lstrip([chars])

#返回截掉字符串左边的空格或指定字符后生成的新字符串。

>>> print s\_str.lstrip()

8888this is test8888

>>> print s\_str.lstrip().lstrip('8')

this is test8888

>>>

S.rstrip([chars])

#返回删除字符串末尾的指定字符后生成的新字符串。

>>> print s\_str.rstrip('8')

8888this is test

>>>

做文本处理的时候经常要判断一个文本有没有以一个子串开始，或者结束。Python为此提供了两个函数：  
S.startswith(prefix[, start[, end]]) -> bool  
如果字符串S以prefix开始，返回True，否则返回False。start和end是两个可以缺省的参数。分别是开始比较的位置和结束比较的位置。这个函数也可以写成S[start:end].startswith(prefix)。  
S.endswith(suffix[, start[, end]]) -> bool  
如果字符串S以suffix结束，返回True，否者返回False。与startswith类似，这个函数也可以写成S[start:end].endswith(suffix)。start和end仍然是从左数起。

>>> "fish".startswith("fi")

True

>>> "fish".endswith("sh")

True

>>>

* 字符串分割组合:

S.split([sep, [maxsplit]])

#以sep为分隔符，把S分成一个list。maxsplit表示分割的次数。默认的分割符为空白字符

>>> print s\_str.split()

['8888this', 'is', 'test8888']

>>>

S.rsplit([sep, [maxsplit]]) 和split区别不懂

>>> print s\_str.rsplit()

['8888this', 'is', 'test8888']

>>>

S.splitlines([keepends])

#把S按照行分割符分为一个list，keepends是一个bool值，如果为真每行后而会保留行分割符。

>>> print s\_str.splitlines()

[' 8888this is test8888']

>>>

S.join(seq) #返回通过指定字符连接序列中元素后生成的新字符串。

>>> seq=['aa','bb','cc']

>>> s\_str='-'

>>> print s\_str.join(seq)

aa-bb-cc

>>>

这里再强调一次，字符串对象是不可改变的，也就是说在python创建一个字符串后，你不能把这个字符中的某一部分改变。任何上面的函数改变了字符串后，都会返回一个新的字符串，原字串并没有变。其实这也是有变通的办法的，可以用S=list(S)这个函数把S变为由单个字符为成员的list，这样的话就可以使用S[3]='a'的方式改变值，然后再使用S=" ".join(S)还原成字符串

* 浮点数

#%f 打印浮点数

>>> print ("His height is %f m" %(1.83) )

His height is 1.830000 m

>>> print ("His height is %f m" %(1.1) )

His height is 1.100000 m

>>>

# 打印浮点数，指定小数点位数

>>> print ("His height is %.2f m" %(1.83))

His height is 1.83 m

>>> print ("His height is %.2f m" %(1.83234))

His height is 1.83 m

>>> print ("His height is %.3f m" %(1.83234))

His height is 1.832 m

>>>

#指定占位符宽度

>>> print ("Nmae:%10s Age:%8d Height:%8.2f" %("Aviad",25,1.83))

Nmae: Aviad Age: 25 Height: 1.83

>>>

#指定占位符宽度（左对齐）

>>> print ("Nmae:%-10s Age:%-8d Height:%-8.2f" %("Aviad",25,1.83) )

Nmae:Aviad Age:25 Height:1.83

>>>

#指定占位符（只能用0当占位符）

>>> print ("Nmae:%-10s Age:%08d Height%08.2f" %("ACiad",25,1.83))

Nmae:ACiad Age:00000025 Height00001.83

>>

* format 字符串格式化调用方法

python 2.6 和Python3.0（以及以后的版本）中新的字符串对象的format方法使用主体字符串作为模板，并且接受任意多个表示将要根据模板替换的值的参数。在主体字符串中，花括号通过位置（例如，{1}或者关键字（例如，{food}））指出替换目标及将要插入的参数。

注意：通过位置指定的时候是根据format()里面传入的参数的位置来确定

#通过指定位置

>>> template='{0},{1} and {2}'

>>> template.format('spam','ham','eggs')

'spam,ham and eggs'

0 1

>>> template2='{1},{0},{1}'.format('ceshi','test')

>>> template2

'test,ceshi,test'

#通过类别下标

>>> p=["kk",18]

>>> p

['kk', 18]

>>> template3='{0[0]},{0[1]}'.format(p)

>>> template3

'kk,18'

#通过指定关键字

>>> templage1='{motto},{pork} and {food}'

>>> templage1.format(motto='spam',pork='ham',food='eggs')

'spam,ham and eggs'

#字符串可以创建一个临时字符串的常亮，并且任意的对象类型都可以替换

>>> '{motto},{0} and {food}'.format(42,motto=3.14,food=[1,2])

'3.14,42 and [1, 2]'

#

>>> x='{motto},{0} and {food}'.format(42,motto=3.14,food=[1,2])

>>> x

'3.14,42 and [1, 2]'

>>>

#^、<、>分别是居中、左对齐、右对齐，后面带宽度  
#:号后面带填充的字符，只能是一个字符，不指定的话默认是用空格填充

#指定偏移量

#{0:10}表示一个10位字符宽的字段中的第一个位置参数

#{1:<10}表示第二个位置参数在一个10位字符宽度字段中 左对齐

>>> '{0:10}={1:10}'.format('spam',123.4567)

'spam = 123.4567'

>>> '{1:>10}={1:<10}'.format('spam',123.4567)

' 123.4567=123.4567 '

#精度与类型f

>>> template4='{:.2f}'.format(3.1425165)

>>> template4

'3.14'

#使用逗号默认3位分割

>>> '{:,}'.format(123456789)

'123,456,789'

* 类型转换：

int(x [,base ]) 将x转换为一个整数

long(x [,base ]) 将x转换为一个长整数

float(x ) 将x转换到一个浮点数

complex(real [,imag ]) 创建一个复数

str(x ) 将对象 x 转换为字符串

repr(x ) 将对象 x 转换为表达式字符串

eval(str ) 用来计算在字符串中的有效Python表达式,并返回一个对象

tuple(s ) 将序列 s 转换为一个元组

list(s ) 将序列 s 转换为一个列表

chr(x ) 将一个整数转换为一个字符

unichr(x ) 将一个整数转换为Unicode字符

ord(x ) 将一个字符转换为它的整数值

hex(x ) 将一个整数转换为一个十六进制字符串

oct(x ) 将一个整数转换为一个八进制字符串

## 运算

* 运算符

|  |  |  |
| --- | --- | --- |
| + | 加 - 两个对象相加 | a + b 输出结果 30 |
| - | 减 - 得到负数或是一个数减去另一个数 | a - b 输出结果 -10 |
| \* | 乘 - 两个数相乘或是返回一个被重复若干次的字符串 | a \* b 输出结果 200 |
| / | 除 - x除以y | b / a 输出结果 2 |
| % | 取模 - 返回除法的余数 | b % a 输出结果 0 |
| \*\* | 幂 - 返回x的y次幂 | a\*\*b 为10的20次方， 输出结果 100000000000000000000 |
| // | 取整除 - 返回商的整数部分 | 9//2 输出结果 4 , 9.0//2.0 输出结果 4.0 |

* 比较运算

|  |  |
| --- | --- |
| == | 等于 - 比较对象是否相等 |
| != | 不等于 - 比较两个对象是否不相等 |
| <> | 不等于 - 比较两个对象是否不相等 |
| > | 大于 - 返回x是否大于y |
| < | 小于 - 返回x是否小于y。所有比较运算符返回1表示真，返回0表示假。这分别与特殊的变量True和False等价。注意，这些变量名的大写。 |
| >= | 大于等于 - 返回x是否大于等于y。 |
| <= | 小于等于 - 返回x是否小于等于y。 |

* 赋值运算

|  |  |  |
| --- | --- | --- |
| = | 简单的赋值运算符 | c = a + b 将 a + b 的运算结果赋值为 c |
| += | 加法赋值运算符 | c += a 等效于 c = c + a |
| -= | 减法赋值运算符 | c -= a 等效于 c = c - a |
| \*= | 乘法赋值运算符 | c \*= a 等效于 c = c \* a |
| /= | 除法赋值运算符 | c /= a 等效于 c = c / a |
| %= | 取模赋值运算符 | c %= a 等效于 c = c % a |
| \*\*= | 幂赋值运算符 | c \*\*= a 等效于 c = c \*\* a |
| //= | 取整除赋值运算符 | c //= a 等效于 c = c // a |

* 逻辑运算

|  |  |  |
| --- | --- | --- |
| and | x and y | 与运算，只有所有都为True，and运算结果才是True： |
| or | x or y | 或运算，只要其中有一个为True，or运算结果就是True： |
| not | not x | 运算是非运算，它是一个单目运算符，把True变成False，False变成True： |

* 成员运算

|  |  |  |
| --- | --- | --- |
| in | 如果在指定的序列中找到值返回 True，否则返回 False。 | x 在 y 序列中 , 如果 x 在 y 序列中返回 True。 |
| not in | 如果在指定的序列中没有找到值返回 True，否则返回 False。 | x 不在 y 序列中 , 如果 x 不在 y 序列中返回 True。 |

* 身份运算

|  |  |  |
| --- | --- | --- |
| is | is是判断两个标识符是不是引用自一个对象 | x is y, 如果 id(x) 等于 id(y) , **is** 返回结果 1 |
| is not | is not是判断两个标识符是不是引用自不同对象 | x is not y, 如果 id(x) 不等于 id(y). **is not** 返回结果 1 |

* 位运算

位运算二进制运算，在计算机识别中，只有0和1 ，计算机的存储单位

1Byte=8bit

1Byte标识 128 64 32 16 8 4 2 1 其中每个字段标识一个bit

数字10的二进制

0 0 0 0 1 0 1 0

|  |  |  |
| --- | --- | --- |
| & | 按位与运算符：参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0 | (a & b) 输出结果 12 ，二进制解释： 0000 1100 |
| | | 按位或运算符：只要对应的二个二进位有一个为1时，结果位就为1。 | (a0 1 | b) 输出=结果 61 ，二进制解释： 0011 1101 |
| ^ | 按位异或运算符：当两对应的二进位相异时，结果为1 | (a ^ b) 输出结果 49 ，二进制解释： 0011 0001 |
| ~ | 按位取反运算符：对数据的每个二进制位取反,即把1变为0,把0变为1 | (~a ) 输出结果 -61 ，二进制解释： 1100 0011， 在一个有符号二进制数的补码形式。 |
| << | 左移动运算符：运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。 | a << 2 输出结果 240 ，二进制解释： 1111 0000 |
| >> | 右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数 | a >> 2 输出结果 15 ，二进制解释： 0000 1111 |

## 流程控制

* 判断

Python程序语言指定任何非0和非空（null）值为true，0 或者 null为false。

条件判断

Python 中if语句用于控制程序的执行，当判断条件成立时(非0)，则执行后面的语句，执行内容可以是是多行，以缩进来区分表示同一范围。

else 为可选语句，当需要在条件不成立是执行内容则可以执行相关语句。

if 语句的判断条件可以用>（大于）、<(小于)、==（等于）、>=（大于等于）、<=（小于等于）

!= (不等于)来表示其关系

如果判断需要多个条件需同时判断时，可以使用 or （或），表示两个条件有一个成立时判断条件成功；使用 and （与）时，表示只有两个条件同时成立的情况下，判断条件才成功。

当if有多个条件时可使用括号来区分判断的先后顺序，括号中的判断优先执行，此外 and 和 or 的优先级低于>（大于）、<（小于）等判断符号，即大于和小于在没有括号的情况下会比与或要优先判断。

1. 判断

if 判断条件：

执行语句……

else：

执行语句……

1. 多重判断

if 判断条件1:

执行语句1……

elif 判断条件2:

执行语句2……

elif 判断条件3:

执行语句3……

else:

执行语句4……

1. 循环

for:

for iterating\_var in sequence:

statements(s)

for iterating\_var in sequence:

statements(s)

else:

执行语句……

当for语句正常执行结束以后，才执行else语句

while:

while 判断条件：

执行语句……

while 判断条件：

执行语句……

else:

执行语句……

当while语句正常执行结束以后，才执行else语句

1. 断点

break

在语句块执行过程中终止循环，并且跳出整个循环

continue

在语句块执行过程中终止当前循环，跳出该次循环，执行下一次循环。

pass

pass是空语句，是为了保持程序结构的完整性

## .序列

序列是Python中最基本的数据结构。序列中的每个元素都分配一个数字—它的位置或索引，第一个索引是0，第二个是1，依次类推。

Python中有6个序列的内置类型，其中最常见的是列表和元组。

序列都可以进行的操作包括索引，切片，加，减，乘，检查成员。

Python中已经内置确定序列的长度以及确定最大和最小的元素的方法。

* 列表 有序的可变集合

列表常用函数：

append()#在列表末尾添加新的对象

>>> name\_list

['this', 'is', 'test']

>>> name\_list.append('songbing')

>>> name\_list

['this', 'is', 'test', 'songbing']

>>>

count()#统计某个元素在列表中出行的次数

>>> name\_list

['this', 'is', 'test', 'songbing', 'is']

>>> name\_list.count('is')

2

>>>

index()#从列表中找出某个值第一个匹配项的索引的位置  
>>> name\_list

['this', 'is', 'test', 'songbing', 'is']

>>> name\_list.index('is')

1

>>>

Pop()#移除列表中的一个元素(默认最后一个元素)，并且返回该元素的值，可以直接指定需要移除的元素的索引值

>>> name\_list

['this', 'is', 'test', 'songbing', 'is']

>>> name\_list.pop()

'is'

>>> name\_list

['this', 'is', 'test', 'songbing']

>>>

>>> name\_list.pop(3)

'songbing'

>>> name\_list

['this', 'is', 'test']

>>>

Insert()#插入对象到列表

>>> name\_list

['this', 'is', 'test']

>>> name\_list.insert(2,'aaaa')

>>> name\_list

['this', 'is', 'aaaa', 'test']

>>>

Remove()#移除列表中某个值的第一个匹配项

>>> name\_list.append('is')

>>> name\_list

['this', 'is', 'aaaa', 'test', 'is']

>>> name\_list.remove('is')

>>> name\_list

['this', 'aaaa', 'test', 'is']

>>>

Reverse()#反向列表中的元素

>>> name\_list

['this', 'aaaa', 'test', 'is']

>>> name\_list.reverse()

>>> name\_list

['is', 'test', 'aaaa', 'this']

>>>

Sort()#对原列表进行排序

>>> name\_list

['is', 'test', 'aaaa', 'this']

>>> name\_list.sort()

>>> name\_list

['aaaa', 'is', 'test', 'this']

>>>

* 列表切片：

>>> name\_list

['this', 'is', 'test']

>>> name\_list[:2]

['this', 'is']

>>> name\_list[1:2]

['is']

>>> name\_list[-2:]

['is', 'test']

>>>

* 列表解析表达式：

处理序列的操作和列表的方法中，Python还包括了跟高级的操作，称作列表解析表达式（list comprehension expression）,从而提供了一种处理像矩阵这样结构的强大工具。

列表解析源自集合的概念，它是一种通过对序列中的每一项运行一个表达式来创建一个新列表的方法，每次一个，从左至右。列表解析是编写在方括号中（

提醒你创建列表这个事实），并且由使用了同一个变量名的（这里是row）表达式和循环结构组成。下面的这个类别解析表达式基本上就是“把矩阵M的每个row中的row[1]，放在一个新的列表中”。其结果就是一个包含了矩阵的第二列的心列表。

>>> M=[[1,2,3],[4,5,6],[7,8,9]]

>>> M

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

#通过for循环取M的，并把值付给row[1]取出每个小列表里面的值

>>> col2=[row[1] for row in M]

>>> col2

[2, 5, 8]

>>>

类似：

>>> for row in M:

... print row[1]

...

2

5

8

复杂列表解析表达式

>>> [row[1]+1 for row in M]

[3, 6, 9]

>>> [row[1] for row in M if row[2] %2 ==0]

[5]

第一个操作，把搜索到的每个元素都加一

第二个操作，把搜索到的每个元素通过if条件语句，使用%取余表达式过滤了奇数作为结果

>>> M

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

>>> diag=[M[i][i] for i in [0,1,2]]

>>> diag

[1, 5, 9]

>>>

>>> doubles=[c\*2 for c in 'spam']

>>> print doubles

['ss', 'pp', 'aa', 'mm']

>>>

>>> [x+y for x in 'abc' for y in 'spam']

['as', 'ap', 'aa', 'am', 'bs', 'bp', 'ba', 'bm', 'cs', 'cp', 'ca', 'cm']

>>>

* 元组 有序不可变集合

元组与列表类似，不同之处在于元组的元素不能修改，不能对元组进行内部操作

元组中的元素值是不允许删除修改增加的，但我们可以使用del语句来删除整个元组

>>> name\_trun=('this','is','test')

>>> name\_trun

('this', 'is', 'test')

>>> del name\_trun

>>> name\_trun

Traceback (most recent call last):

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

NameError: name 'name\_trun' is not defined

要定义一个只有1个元素的tuple，如果你这么定义：

>>> t = (1)

>>> t

1

定义的不是tuple，是1这个数！这是因为括号()既可以表示tuple，又可以表示数学公式中的小括号，这就产生了歧义，因此，Python规定，这种情况下，按小括号进行计算，计算结果自然是1。

所以，只有1个元素的tuple定义时必须加一个逗号,，来消除歧义：

>>> t = (1,)

>>> t

(1,)

Python在显示只有1个元素的tuple时，也会加一个逗号,，以免你误解成数学计算意义上的括号。

最后来看一个“可变的”tuple：

>>> t = ('a', 'b', ['A', 'B'])

>>> t[2][0] = 'X'

>>> t[2][1] = 'Y'

>>> t

('a', 'b', ['X', 'Y'])

这个tuple定义的时候有3个元素，分别是'a'，'b'和一个list。不是说tuple一旦定义后就不可变了吗？怎么后来又变了？

别急，我们先看看定义的时候tuple包含的3个元素：



当我们把list的元素'A'和'B'修改为'X'和'Y'后，tuple变为：



表面上看，tuple的元素确实变了，但其实变的不是tuple的元素，而是list的元素。tuple一开始指向的list并没有改成别的list，所以，tuple所谓的“不变”是说，tuple的每个元素，指向永远不变。即指向'a'，就不能改成指向'b'，指向一个list，就不能改成指向其他对象，但指向的这个list本身是可变的！

理解了“指向不变”后，要创建一个内容也不变的tuple怎么做？那就必须保证tuple的每一个元素本身也不能变。

* 集合

 set是一个无序不重复元素集, 基本功能包括关系测试和消除重复元素.

集合对象还支持union(联合), intersection(交), difference(差)

和sysmmetric difference(对称差集)等数学运算.

>>> x=set('spam')

>>> print x

set(['a', 'p', 's', 'm'])

>>> y=set(['h','a','m'])

>>> print y

set(['a', 'h', 'm'])

>>> x,y

(set(['a', 'p', 's', 'm']), set(['a', 'h', 'm']))

#交集，列出两个集合共有的元素

>>> x & y

set(['a', 'm'])

>>>

#并集，合并两个集合并去重复

>>> x | y

set(['a', 'p', 's', 'h', 'm'])

>>>

#差集，列出在x集合中有，y中没有的元素

>>> x - y

set(['p', 's'])

>>>

两种集合的添加方法

#集合添加 add

>>> x.add('test')

>>> x

set(['a', 'p', 's', 'test', 'm'])

>>>

#集合添加 update

>>> x.update('ceshi')

>>> x

set(['a', 'c', 'e', 'i', 'h', 'm', 'p', 's', 'test'])

>>>

#集合删除元素 remove

>>> x.remove('test')

>>> x

set(['a', 'c', 'e', 'i', 'h', 'm', 'p', 's'])

>>>

Set 集合常用的操作方法

s.add(x)

向 set “s”中增加元素 x

s.remove(x)

从 set “s”中删除元素 x, 如果不存在则引发 KeyError

s.discard(x)

如果在 set “s”中存在元素 x, 则删除，不存在也不引发报错

s.pop()

删除并且返回 set “s”中的一个不确定的元素, 如果为空则引发 KeyError

s.clear()

删除 set “s”中的所有元素

## 字典

字典是另一种可变容器模型，可存储任意类型对象。无序

字典的每个键值(key=value)对用冒号(:)分割，每个对之间用逗号(,)分割，整个字典包括在花括号内({}).

dic={

key1:value1,

key2:value2,

}

字典中键值必须是唯一，但是值则不必

值可以取任何数据类型，但是键必须是不可变的，如字符串，数字，元组

* 字典常用方法

访问字典的值

>>> dict={'name':'song','age':27}

>>> print dict['name']

song

>>>

如果key值不存在返回报错

>>> print dict['job']

Traceback (most recent call last):

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

KeyError: 'job'

>>>

字典添加值

>>> dict['job']='IT'

>>> dict

{'job': 'IT', 'age': 27, 'name': 'song'}

>>>

修改字典的值

>>> dict['name']='bing'

>>> dict

{'job': 'IT', 'age': 27, 'name': 'bing'}

>>>

删除字典里面的值

>>> del dict['job']

>>> dict

{'age': 27, 'name': 'bing'}

>>>

Python2和Python3中动态初始化字典方法

>>> list(zip(['a','b','c'],[1,2,3]))

[('a', 1), ('b', 2), ('c', 3)]

>>> D=dict(zip(['a','b','c'],[1,2,3]))

>>> D

{'b': 2, 'a': 1, 'c': 3}

Python3

>>> D1={k:v for (k,v) in zip(['a','b','c'],[1,2,3])}

>>> D1

{'b': 2, 'a': 1, 'c': 3}

>>> D2={x:x\*\*2 for x in [1,2,3,4]}

>>> D2

{1: 1, 2: 4, 3: 9, 4: 16}

>>> D3={c:c\*4 for c in 'spam'}

>>> D3

{'a': 'aaaa', 'm': 'mmmm', 'p': 'pppp', 's': 'ssss'}

>>> D4=dict.fromkeys(['a','b','c'],[1,2,3])

>>> D4

{'b': [1, 2, 3], 'a': [1, 2, 3], 'c': [1, 2, 3]}

>>> D5=dict.fromkeys(['a','b','c'],0)

>>> D5

{'b': 0, 'a': 0, 'c': 0}

>>> D6=dict.fromkeys('spam')

>>> D6

{'a': None, 'm': None, 'p': None, 's': None}

#动态的反应在视图对象创建以后的修改

>>> D7={'a':1,'b':2,'c':3}

>>> D7

{'b': 2, 'a': 1, 'c': 3}

>>> k=D.keys()

>>> v=D.values()

>>> list(k)

['a', 'm', 'p', 's']

>>> k=D7.keys()

>>> v=D7.values()

>>> list(k)

['b', 'a', 'c']

>>> list(v)

[2, 1, 3]

>>> del D7['b']

>>> list(k)

['a', 'c']

>>> list(v)

[1, 3]

注意：

字典值可以没有限制地取任何python对象，既可以是标准的对象，也可以是用户定义的，但键不行。

不允许同一个键出现两次。创建时如果同一个键被赋值两次，后一个值会被记住

* 字典中内置方法

|  |  |
| --- | --- |
| 1 | [radiansdict.clear()](http://www.runoob.com/python/att-dictionary-clear.html) |
| 删除字典内所有元素 |
| 2 | [radiansdict.copy()](http://www.runoob.com/python/att-dictionary-copy.html) |
| 返回一个字典的浅复制 |
| 3 | [radiansdict.fromkeys()](http://www.runoob.com/python/att-dictionary-fromkeys.html) |
| 创建一个新字典，以序列seq中元素做字典的键，val为字典所有键对应的初始值 |
| 4 | [radiansdict.get(key, default=None)](http://www.runoob.com/python/att-dictionary-get.html) |
| 返回指定键的值，如果值不在字典中返回default值 |
| >>> dict.get('name') |
| 'bing' |
| >>> dict.get('test') |
| >>> print dict.get('test') |
| None |
| >>> |
| 5 | [radiansdict.has\_key(key)](http://www.runoob.com/python/att-dictionary-has_key.html) |
| 如果键在字典dict里返回true，否则返回false |
| >>> if dict.has\_key('name'): |
| ... print dict['name'] |
| ... |
| bing |
| >>> |
|  |
| >>> if dict.has\_key('test'): |
| ... print dict['test'] |
| ... else: |
| ... print "Not found" |
| ... |
| Not found |
| >>> |
| 6 | [radiansdict.items()](http://www.runoob.com/python/att-dictionary-items.html) |
| 以列表返回可遍历的(键, 值) 元组数组 |
| >>> for key,vaul in dict.items(): |
| ... print key,vaul |
| ... |
| age 27 |
| name bing |
| >>> |
| 7 | [radiansdict.keys()](http://www.runoob.com/python/att-dictionary-keys.html) |
| 以列表返回一个字典所有的键 |
| >>> dict.keys() |
| ['age', 'name'] |
| >>> |
| 8 | [radiansdict.setdefault(key, default=None)](http://www.runoob.com/python/att-dictionary-setdefault.html) |
| 和get()类似, 但如果键不存在于字典中，将会添加键并将值设为default |
| 9 | [radiansdict.update(dict2)](http://www.runoob.com/python/att-dictionary-update.html) |
| 把字典dict2的键/值对更新到dict里 |
| 10 | [radiansdict.values()](http://www.runoob.com/python/att-dictionary-values.html) |
| 以列表返回字典中的所有值 |
| >>> dict.values() |
| [27, 'bing'] |
| >>> |
|  |  |

## 模块 包

在Python中一个.py文件就称为一个模块。

使用模块提高了代码的可维护性，还可以避免函数名和变量名的冲突。

相同名字的函数和变量完全可以分别在不同的模块中，

模块常用导入方法，

为了避免模块名冲突，Python又引入了按目录来组织模块的方法，称为包（Package）。

举个例子，一个abc.py的文件就是一个名字叫abc的模块，一个xyz.py的文件就是一个名字叫xyz的模块。

现在，假设我们的abc和xyz这两个模块名字与其他模块冲突了，于是我们可以通过包来组织模块，避免冲突。方法是选择一个顶层包名，比如mycompany，按照如下目录存放：



引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。现在，abc.py模块的名字就变成了mycompany.abc，类似的，xyz.py的模块名变成了mycompany.xyz。

请注意，每一个包目录下面都会有一个\_\_init\_\_.py的文件，这个文件是必须存在的，否则，Python就把这个目录当成普通目录，而不是一个包。\_\_init\_\_.py可以是空文件，也可以有Python代码，因为\_\_init\_\_.py本身就是一个模块，而它的模块名就是mycompany。



文件www.py的模块名就是mycompany.web.www，两个文件utils.py的模块名分别是mycompany.utils和mycompany.web.utils。

mycompany.web也是一个模块，请指出该模块对应的.py文件。

import os#直接导入模块

Python的from语句让你从模块中导入一个指定的部分到当前命名空间中

from os import \* #不建议使用

from os import system,popen #导入多个方法

import muliprocessing as multi #别名导入方法

模块能够有逻辑的组织Python代码段

模块就是一个保存了Python代码的文件。模块能定义函数，类和变量。模块里也能包含可执行的代码。

一个模块只会被导入一次，不管你执行了多少次import。这样可以防止导入模块被一遍又一遍地执行

当你导入一个模块，Python解析器对模块位置的搜索顺序是：

当前目录

如果不在当前目录，Python 则搜索在 shell 变量 PYTHONPATH 下的每个目录。

如果都找不到，Python会察看默认路径。UNIX下，默认路径一般为/usr/local/lib/python/。

模块搜索路径存储在system模块的sys.path变量中。变量里包含当前目录，PYTHONPATH和由安装过程决定的默认目录。

## InPut/OutPut

IO在计算机中指Input/Output，也就是输入和输出。由于程序和运行时数据是在内存中驻留，由CPU这个超快的计算核心来执行，涉及到数据交换的地方，通常是磁盘、网络等，就需要IO接口。

比如你打开浏览器，访问新浪首页，浏览器这个程序就需要通过网络IO获取新浪的网页。浏览器首先会发送数据给新浪服务器，告诉它我想要首页的HTML，这个动作是往外发数据，叫Output，随后新浪服务器把网页发过来，这个动作是从外面接收数据，叫Input。所以，通常，程序完成IO操作会有Input和Output两个数据流。当然也有只用一个的情况，比如，从磁盘读取文件到内存，就只有Input操作，反过来，把数据写到磁盘文件里，就只是一个Output操作。

IO编程中，Stream（流）是一个很重要的概念，可以把流想象成一个水管，数据就是水管里的水，但是只能单向流动。Input Stream就是数据从外面（磁盘、网络）流进内存，Output Stream就是数据从内存流到外面去。对于浏览网页来说，浏览器和新浪服务器之间至少需要建立两根水管，才可以既能发数据，又能收数据。

由于CPU和内存的速度远远高于外设的速度，所以，在IO编程中，就存在速度严重不匹配的问题。举个例子来说，比如要把100M的数据写入磁盘，CPU输出100M的数据只需要0.01秒，可是磁盘要接收这100M数据可能需要10秒，怎么办呢？有两种办法：

第一种是CPU等着，也就是程序暂停执行后续代码，等100M的数据在10秒后写入磁盘，再接着往下执行，这种模式称为同步IO；

另一种方法是CPU不等待，只是告诉磁盘，“您老慢慢写，不着急，我接着干别的事去了”，于是，后续代码可以立刻接着执行，这种模式称为异步IO。

同步和异步的区别就在于是否等待IO执行的结果。好比你去麦当劳点餐，你说“来个汉堡”，服务员告诉你，对不起，汉堡要现做，需要等5分钟，于是你站在收银台前面等了5分钟，拿到汉堡再去逛商场，这是同步IO。

你说“来个汉堡”，服务员告诉你，汉堡需要等5分钟，你可以先去逛商场，等做好了，我们再通知你，这样你可以立刻去干别的事情（逛商场），这是异步IO。

很明显，使用异步IO来编写程序性能会远远高于同步IO，但是异步IO的缺点是编程模型复杂。想想看，你得知道什么时候通知你“汉堡做好了”，而通知你的方法也各不相同。如果是服务员跑过来找到你，这是回调模式，如果服务员发短信通知你，你就得不停地检查手机，这是轮询模式。总之，异步IO的复杂度远远高于同步IO。

操作IO的能力都是由操作系统提供的，每一种编程语言都会把操作系统提供的低级C接口封装起来方便使用，Python也不例外。我们后面会详细讨论Python的IO编程接口。

读写文件前，我们先必须了解一下，在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）。

* 文件读写

读写文件是最常见的I0操作，Python内置了读写文件的函数读写文件前，需要先了解，在磁盘上读写文件的功能是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以读写文件就是请求操作系统打开一个文件对象然后通过操作系统提供的接口从这个文件对象中读取数据，或者 把数据写入这个文件对象

* Open函数

file object = open(file\_name [, access\_mode][, buffering])

* file\_name：file\_name变量是一个包含了你要访问的文件名称的字符串值。
* access\_mode：access\_mode决定了打开文件的模式：只读，写入，追加等。所有可取值见如下的完全列表。这个参数是非强制的，默认文件访问模式为只读(r)。
* buffering:如果buffering的值被设为0，就不会有寄存。如果buffering的值取1，访问文件时会寄存行。如果将buffering的值设为大于1的整数，表明了这就是的寄存区的缓冲大小。如果取负值，寄存区的缓冲大小则为系统默认。

|  |  |
| --- | --- |
| r | 以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。 |
| rb | 以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。 |
| r+ | 打开一个文件用于读写。文件指针将会放在文件的开头。 |
| rb+ | 以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。 |
| w | 打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。 |
| wb | 以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。 |
| w+ | 打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。 |
| wb+ | 以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。 |
| a | 打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。 |
| ab | 以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。 |
| a+ | 打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写。 |
| ab+ | 以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。 |

文件被打开后，就有一个file对象，可以得到该文件的各种信息

|  |  |
| --- | --- |
| file.closed | 返回true如果文件已被关闭，否则返回false。 |
| file.mode | 返回被打开文件的访问模式。 |
| file.name | 返回文件的名称。 |
| file.softspace | 如果用print输出后，必须跟一个空格符，则返回false。否则返回true。 |
| f.flush() | 刷新输出缓存 |

* Python 读写方法

使用read()和write()方法来读取和写入文件

write()方法可将任何字符串写入一个打开的文件。需要重点注意的是，Python字符串可以是二进制数据，而不是仅仅是文字。

write()方法不会在字符串的结尾添加换行符('\n')：

read（）方法从一个打开的文件中读取一个字符串。需要重点注意的是，Python字符串可以是二进制数据，而不是仅仅是文字。

**fileObject.read([count]);**

在这里，被传递的参数是要从已打开文件中读取的字节计数。该方法从文件的开头开始读入，如果没有传入count，它会尝试尽可能多地读取更多的内容，很可能是直到文件的末尾。

f.read([count])  
读出文件，如果有count，则读出count个字节。  
f.readline()  
读出一行信息。  
f.readlines()  
读出所有行，也就是读出整个文件的信息。

f=open('file\_test.txt','w')  
f.write('This test file')  
f.close()  
  
f=open('file\_test.txt','r')  
print f.read()  
f.close()

* 文件定位

tell()方法告诉你文件内的当前位置；换句话说，下一次的读写会发生在文件开头这么多字节之后。

seek（offset [,from]）方法改变当前文件的位置。Offset变量表示要移动的字节数。From变量指定开始移动字节的参考位置。

如果from被设为0，这意味着将文件的开头作为移动字节的参考位置。如果设为1，则使用当前的位置作为参考位置。如果它被设为2，那么该文件的末尾将作为参考位置。

# f.seek(offset[,where])  
# 把文件指针移动到相对于where的offset位置。where为0表示文件开始处，这是默认值 ；1表示当前位置；2表示文件结尾。  
# f.tell()  
# 获得文件指针位置。

f=open('file\_test.txt','r')  
for i in f.readlines():  
 print len(i)  
print f.tell()  
f.seek(15)  
print f.read()  
f.close()

读取，写入文件后需要调用close()方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的：

每次都这么写实在太繁琐，所以，Python引入了with语句来自动帮我们调用close()方法：

with open('/path/to/file', 'r') as f:

print f.read()

* fileinput 函数

【 基本格式 】

fileinput.input([files[, inplace[, backup[, bufsize[, mode[, openhook]]]]]])

【 默认格式 】

fileinput.input (files=None, inplace=False, backup='', bufsize=0, mode='r', openhook=None)

files: #文件的路径列表，默认是stdin方式，多文件['1.txt','2.txt',...]

inplace: #是否将标准输出的结果写回文件，默认不取代

backup: #备份文件的扩展名，只指定扩展名，如.bak。如果该文件的备份文件已存在，则会自动覆盖。

bufsize: #缓冲区大小，默认为0，如果文件很大，可以修改此参数，一般默认即可

mode: #读写模式，默认为只读

openhook: #该钩子用于控制打开的所有文件，比如说编码方式等;

【 常用函数】

fileinput.input() #返回能够用于for循环遍历的对象

fileinput.filename() #返回当前文件的名称

fileinput.lineno() #返回当前已经读取的行的数量（或者序号）

fileinput.filelineno() #返回当前读取的行的行号

fileinput.isfirstline() #检查当前行是否是文件的第一行

fileinput.isstdin() #判断最后一行是否从stdin中读取

fileinput.close() #关闭队列

#查看文件内容  
for line in fileinput.input('test.txt'):  
 print line,

#文件内容替换  
#backup备份，只有当inplace参数为1的时候backup参数才能生效  
 for line in fileinput.input('test.txt',backup='.bak',inplace=1):  
 print line.rstrip().replace('This','HTIS')

#文件处理  
import sys  
for line in fileinput.input(r'test.txt'):  
 sys.stdout.write('==>')  
 sys.stdout.write(line)

## 序列化

我们把变量从内存中变成可存储或传输的过程称之为序列化

序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即unpickling。

Python提供两个模块来实现序列化：cPickle和pickle。这两个模块功能是一样的，区别在于cPickle是C语言写的，速度快，pickle是纯Python写的，速度慢，跟cStringIO和StringIO一个道理。用的时候，先尝试导入cPickle，如果失败，再导入pickle：

**try**:  
 **import** cPickle **as** pickle  
**except** ImportError:  
 **import** pickle

pickle 支持的存储类型

所有Python支持的 [原生类型](http://woodpecker.org.cn/diveintopython3/native-datatypes.html) : 布尔, 整数, 浮点数, 复数, 字符串, bytes (字节串)对象, 字节数组, 以及 None .

由任何原生类型组成的列表，元组，字典和集合。

由任何原生类型组成的列表，元组，字典和集合组成的列表，元组，字典和集合(可以一直嵌套下去，直至[Python支持的最大递归层数](http://docs.python.org/3.1/library/sys.html#sys.getrecursionlimit) ).

函数，类，和类的实例(带警告)。

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
d=dict(name="bob",age=20,score=88)  
print(d)  
  
  
import pickle  
  
#序列化存储  
f=open('dump.pkl','wb')  
pickle.dump(d,f)  
f.close()  
  
#序列化导出  
f1=open('dump.pkl','rb')  
d1=pickle.load(f1)  
f1.close()  
print(d1)

## 正则

#!/usr/bin/env python  
# -\*- coding:utf-8 -\*-   
# Author: songbing  
# Mail : [songbing513@163.com](mailto:songbing513@163.com)

import re  
#match尝试从字符串的开始匹配，从字符串的一个开始处匹配，如果开始处不匹配，则不匹配  
#第一个参数是正则表达式，这里为"(\w+)\s"，如果匹配成功，则返回一个Match，否则返回一个None；  
#第二个参数表示要匹配的字符串；  
#第三个参数是标致位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。  
text="JGood is a handsome boy, he is cool, clever, and so on..."  
m=re.match(r"(\w+)\s",text)  
print 'match'  
if m:  
 print m.group(0),'\n',m.group(1)  
else:  
 print "Not match"

bt='bat bet bit'  
print("match 方法")  
m=re.match('bat',bt)  
  
if m is not None:  
 print(m.group())  
else:  
 print("Not found bat")  
  
n=re.match('bit',bt)  
if n is not None:  
 print(n.group())  
else:  
 print("Not found bit")

输出结果：

match 方法

bat

Not found bit  
#search函数会在字符串内查找模式匹配,只到找到第一个匹配然后返回，如果字符串没有匹配，则返回None  
#匹配整个字符  
print 'search'  
m1=re.search(r'\shan(ds)ome\s',text)  
if m:  
 print m1.group(0),m1.group(1)  
else:  
 print "Not search"

bt1=' bat bet bit'  
print("\nsearch 方法")  
m1=re.search('bat',bt1)  
if m1 is not None:  
 print(m1.group())  
else:  
 print('m1 not found')

输出结果：

search 方法

bat

#findall()函数查询字符串中某个正则表达式模式全部的非重复出现情况  
#与search函数执行字符串搜索类似，但是不同之处，findall()返回一个列表，列表将包含所有成功的匹配部分，从左向右按顺序排列  
# 如果没有搜索到，返回空列表  
f=re.findall('bit',bt1)  
print(f)  
输出结果:

['bit']

f1=re.findall('car','carry the barcardi to the car')  
print(f1)

输出结果：

['car', 'car', 'car']

s='This and that.'  
s1=re.findall(r'th\w+ and th\w+',s,re.I)  
s1\_2=re.findall(r'(th\w+) and (th\w+)',s,re.I)  
s1\_3=re.findall(r'(th\w+)',s,re.I)  
print(s1)  
print(s1\_2)  
print(s1\_3)

输出结果：

['This and that']

[('This', 'that')]

['This', 'that']

#sub()和subn()搜索和替换功能：  
#将某字符串中所有匹配正常表达式的部分进行某种形式替换  
#其中subn返回一个表示替换的总数，替换后的字符串和表示替换总数的数字两个元素，组成一个元组返回  
#re.sub('原有字符串','替换字符串','字符串')  
  
r=re.sub('X','Mr. smith','attn: X\nDear X,\n')  
print(r)  
  
r1=re.subn('X','Mr. smith','attn: X\nDear X,\n')  
print(r1)

输出结果：

attn: Mr. smith

Dear Mr. smith,

('attn: Mr. smith\nDear Mr. smith,\n', 2)

#split通过指定分隔符对字符串进行分割

#str.split(str="", num=string.count(str)).

#str -- 分隔符，默认为空格。

#num -- 分割次数。

#返回分割后的字符串列表

str = "Line1-abcdef Line2-abc Line4-abcd";  
print(str.split())  
print(str.split(' ',1))  
print(str.split(' ',2))

输出结果：

['Line1-abcdef', 'Line2-abc', 'Line4-abcd']

['Line1-abcdef', 'Line2-abc Line4-abcd']

['Line1-abcdef', 'Line2-abc', 'Line4-abcd']

注意分割的区别，后面加上分割次数以后

注意空格，和标红的部分  
DATE=(  
 'Moutain View, CA 94040',  
 'Sunnyvale, CA',  
 'Los Altos, 94023',  
 'Cupertino 95014',  
 'Palo Alto CA',  
)  
  
for datum in DATE:  
 print(re.split(',\s|(?=\s(?:\d{5}|[A-Z]{2}))\s',datum))

输出结果：

['Moutain View', 'CA', '94040']

['Sunnyvale', 'CA']

['Los Altos', '94023']

['Cupertino', '95014']

['Palo Alto', 'CA']

#正则匹配邮箱

#mail=input("请输入你的邮箱地址:")  
mail='songbing513@163.com'  
#取出名字  
print(re.split('@',mail)[0])  
  
#邮箱匹配  
#一 不能以 . - \_ | 特殊字符串 开始,和结束  
#二 不能是特殊字符  
#三 只有一个@  
#四 只有一个.  
#五 结尾是@xxx.xxx模式

匹配邮箱

综合目前国内常用的邮箱，大概通用的规则包括：

1、[^\.\_]，不能以下划线和句点开头。

2、[\w\.]+，包括字母、数字。而对句点及下划线各提供商有差别，对此有效性不做更严格的判断。

3、@是必须的。

4、(?:[A-Za-z0-9]+\.)+[A-Za-z]+$，@后以xxx.xxx结尾，考虑到多级域名，会有这种情况xxx.xxx.xxx如xxx@yahoo.com.cn

p=re.compile('[^\W\.\_-][\w\.-]+@(?:[a-zA-Z0-9]+\.)+[a-zA-Z]+$')  
  
if p.match(mail):  
 print("邮箱匹配正确")  
else:  
 print("邮箱匹配失败")

#扩展符合

print(re.findall(r'(?i)yes','yes? Yes. YES'))

输出结果:

['yes', 'Yes', 'YES']

print(re.findall(r'(?im)(^th[\w\s]+)',"""  
This line is the first,  
another line,  
that line,it's the beat,  
theres is test.  
"""))

(?aiLmsux) 后可以紧跟着 'a'，'i'，'L'，'m'，'s'，'u'，'x' 中的一个或多个字符，只能在正则表达式的开头使用

每一个字符对应一种匹配标志：re-A（只匹配 ASCII 字符），re-I（忽略大小写），re-L（区域设置），re-M（多行模式）, re-S（. 匹配任何符号），re-X（详细表达式），包含这些字符将会影响整个正则表达式的规则

print(re.findall(r'(?im)(^th[\w]+)',"""  
This line is the first,  
another line,  
that line,it's the beat,  
theres is test.  
"""))

print(re.findall(r'(?im)(^th[\w]{2})',"""  
This line is the first,  
another line,  
that line,it's the beat,  
theres is test.  
"""))

print(re.findall(r'(?im)(th[\w\s]+)',"""  
This line is the first,  
another line,  
that line,it's the beat,  
theres is test.  
"""))

print(re.findall(r'(?:\w+\.\*\w+\.com)','http://google.com http://www.google.com http://code.google.com'))  
print(re.findall(r'(?:\w+\.)\*(\w+\.com)','http://google.com http://www.google.com http://code.google.com'))

输出结果:

['This line is the first', 'that line', 'theres is test']

['This', 'that', 'theres']

['This', 'that', 'ther']

['This line is the first', 'ther line', 'that line', 'the beat', 'theres is test']

['google.com', 'www.google.com', 'code.google.com']

['google.com', 'google.com', 'google.com']



(?P...) 命名组，通过组的名字（name）即可访问到子组匹配的字符串

(?P=name) 反向引用一个命名组，它匹配指定命名组匹配的任何内容

(?#...) 注释，括号中的内容将被忽略

(?=...) 前向肯定断言。如果当前包含的正则表达式（这里以 ... 表示）在当前位置成功匹配，则代表成功，否则失败。一旦该部分正则表达式被匹配引擎尝试过，就不会继续进行匹配了；剩下的模式在此断言开始的地方继续尝试。

举个栗子：love(?=FishC) 只匹配后边紧跟着 "FishC" 的字符串 "love"

(?!...) 前向否定断言。这跟前向肯定断言相反（不匹配则表示成功，匹配表示失败）。

举个栗子：FishC(?!\.com) 只匹配后边不是 ".com" 的字符串 "FishC"

(?<=...) 后向肯定断言。跟前向肯定断言一样，只是方向相反。

举个栗子：(?<=love)FishC 只匹配前边紧跟着 "love" 的字符串 "FishC"

(?<!...) 后向否定断言。跟前向肯定断言一样，只是方向相反。

举个栗子：(?<!FishC)\.com 只匹配前边不是 "FishC" 的字符串 ".com"

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

## 函数

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。

函数能提高应用的模块性，和代码的重复利用率。

函数代码块以 def 关键词开头，后接函数标识符名称和圆括号()。

任何传入参数和自变量必须放在圆括号中间。圆括号之间可以用于定义参数。

函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。

函数内容以冒号起始，并且缩进。

return [表达式] 结束函数并选择性地返回一个值给调用方。不带表达式的return相当于返回 None。

函数中默认可以访问全局变量，默认不能修改全局变量，加global声明以后可以修改

函数中可以直接修改全局中的列表，字典的复杂的数据类型

* 函数文档字符串

def functionname():

#定义函数文档字符串  
 "函数介绍"  
 print "This is test!"

#调用函数  
functionname()

#查看函数文档字符串  
print functionname.\_\_doc\_\_

#查看函数帮助  
help(functionname)

1. 必备参数

必备参数必须以正确的顺序传入函数。调用时的数量必须和声明时的一至。

1. 关键字参数

关键字参数和函数调用关系紧密，函数调用使用关键字参数来确定传入的参数值。

使用关键字参数允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值。

>>> def funciont1(arg1,arg2):

... print arg1,arg2

...

>>> funciont1(1,2)

1 2

>>>

1. 默认参数

调用函数时，缺省参数的值如果没有传入，则被认为是默认值。下例会打印默认的arg2，如果arg2没有被传入

>>> def functiont2(arg1,arg2=35):

... print(arg1,arg2)

...

>>> functiont2(1,2)

1 2

>>> functiont2(1)

1 35

>>>

4 不定长参数

需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，和上述几种参数不同，声明时不会命名

def functionname([formal\_args,] \*var\_args\_tuple ):

"函数\_文档字符串"

function\_suite

return [expression]

加了星号（\*）的变量名会存放所有未命名的变量参数。选择不多传参数也可

>>> def function(\*arg):

... print arg

...

>>> function(1)

(1,)

>>> function(2)

(2,)

>>> function(1,2,3)

(1, 2, 3)

>>> function()

()

>>>

在python中函数参数的定义主要有四种方式：

1. F(arg1,arg2,...)

这是最常见的定义方式，一个函数可以定义任意个参数，每个参数间用逗号分割，用这种方式定义的函数在调用的的时候也必须在函数名后的小括号里提供个数相等的 值（实际参数），而且顺序必须相同，也就是说在这种调用方式中，形参和实参的个数必须一致，而且必须一一对应，也就是说第一个形参对应这第一个实参。例 如：

def a(x,y):

print x,y

调用该函数，a(1,2)则x取1，y取2，形参与实参相对应，如果a(1)或者a(1,2,3)则会报错。

1. F(arg1,arg2=value2,...)

这种方式就是第一种的改进版，提供了默认值

def a(x,y=3):

print x,y

调用该函数，a(1,2)同样还是x取1，y取2，但是如果a(1)，则不会报错了，这个时候x还是1，y则为默认的3。上面这俩种方式，还可以更换参数位置，比如a(y=8,x=3)用这种形式也是可以的。

1. F(\*arg1)

上面俩个方式是有多少个形参，就传进去多少个实参，但有时候会不确定有多少个参数，则此时第三种方式就比较有用，它以一个\*加上形参名的方式来表示这个函数 的实参个数不定，可能为0个也可能为n个。注意一点是，不管有多少个，在函数内部都被存放在以形参名为标识符的tuple中。

>>> def a(\*x):

if len(x)==0:

print 'None'

else:

print x

>>> a(1)

(1,) #存放在元组中

>>> a()

None

>>> a(1,2,3)

(1, 2, 3)

>>> a(m=1,y=2,z=3)

Traceback (most recent call last):

File "<pyshell#16>", line 1, in -toplevel-

a(m=1,y=2,z=3)

TypeError: a() got an unexpected keyword argument 'm'

1. F(\*\*arg1)

形参名前加俩个\*表示，参数在函数内部将被存放在以形式名为标识符的dictionary中，这时调用函数的方法则需要采用arg1=value1,arg2=value2这样的形式。

>>> def a(\*\*x):

if len(x)==0:

print 'None'

else:

print x

>>> a()

None

>>> a(x=1,y=2)

{'y': 2, 'x': 1} #存放在字典中

>>> a(1,2) #这种调用则报错

Traceback (most recent call last):

File "<pyshell#25>", line 1, in -toplevel-

a(1,2)

TypeError: a() takes exactly 0 arguments (2 given)

上面介绍了四种定义方式，接下来看函数参数在调用过程中是怎么被解析的,其实只要记住上面这四种方法优先级依次降低，先1，后2，再3，最后4，也就是先把方式1中的arg解析，然后解析方式2中的arg=value，再解析方式3，即是把多出来的arg这种形式的实参组成个tuple传进去，最后把剩下的key=value这种形式的实参组成一个dictionary传给带俩个星号的形参，也就方式4。

>>> def test(x,y=1,\*a,\*\*b):

print x,y,a,b

>>> test(1)

1 1 () {}

>>> test(1,2)

1 2 () {}

>>> test(1,2,3)

1 2 (3,) {}

>>> test(1,2,3,4)

1 2 (3, 4) {}

>>> test(x=1,y=2)

1 2 () {}

>>> test(1,a=2)

1 1 () {'a': 2}

>>> test(1,2,3,a=4)

1 2 (3,) {'a': 4}

>>> test(1,2,3,y=4)

Traceback (most recent call last):

File "<pyshell#52>", line 1, in -toplevel-

test(1,2,3,y=4)

TypeError: test() got multiple values for keyword argument 'y'

* 例子

#!/usr/bin/env python  
# -\*- coding:utf-8 -\*-   
# Author: songbing  
# Mail : songbing513@163.com  
  
#默认函数  
def sayHi():  
 print "你好，我叫宋兵，你叫什么名字"  
  
#定义形参  
def sayHi2(name):  
 print "你好%s,很高兴认识你" % name  
  
#定义默认参数  
def sayHi3(name="liapan"):  
 print "hello %s ,nice to me tee you" % name  
  
def sayHi4(name,age,job="IT",wage=5000):  
 print "my name is %s ,my %d year old,my jobs is %s ,my wage is %s" % (name,age,job,wage)

#定义局部变量和全局变量，在函数内声明全局变量后，此变量在全局可用

#在函数外部定义的变量函数内部可以调用修改，但是之作用于函数内部  
def sayHi5():  
 global user\_name  
 print user\_name  
 user\_name='liapan'  
 print 'my name is %s ' % user\_name  
  
  
def sayHi6(x,y):  
 if x>y:  
 return x/y  
 else:  
 return x\*y

def sayHi7(\*args):  
 print args  
  
def sayHi8(\*\*kwargs):  
 print kwargs  
 if kwargs.has\_key('name'):  
 print kwargs['name']  
  
  
#函数调用  
sayHi()  
  
#定义实参  
user\_name="宋兵"  
sayHi2(user\_name)  
  
#未定义函数的参数是，调用默认参数  
sayHi3()  
  
#定义多个默认参数  
sayHi4("songbing",26)  
  
#定义局部，全局变量  
sayHi5()  
  
#定义函数的返回值  
test1=sayHi6(10,2)  
test2=sayHi6(2,10)  
  
print test1,test2  
  
#定义函数，不固定的多个参数（元组）  
sayHi7(1,2,3,4,5)  
  
#定义函数多个参数，根据传入的参数，取对应的值（字典）  
sayHi8(name="songbing",age=26,job='IT')  
  
a=False

* 变量作用域

一个程序的所有的变量并不是在哪个位置都可以访问的。访问权限决定于这个变量是在哪里赋值的。

变量的作用域决定了在哪一部分程序你可以访问哪个特定的变量名称。两种最基本的变量作用域如下：

全局变量

局部变量

作用域变量定义

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。

局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。调用函数时，所有在函数内声明的变量名称都将被加入到作用域中。

* 例子：

全局变量在局部变量可以调用

name='song'  
def funciton\_t():  
 print name  
  
funciton\_t()

输出结果

song

全局变量在局部变量中可以被修改，但是修改不影响全局变量，只在局部变量中生效

def function\_t1():  
 name='bing'  
 print name  
  
function\_t1()  
print name

输出结果:

bing

song

如果需要把在局部变量中修改的全局变量，在全局中生效，需要把变量为全局变量，使用global声明变量

def funciton\_t2():  
 global name  
 name='bing'  
 print name  
funciton\_t2()  
print name

输出结果:

bing

bing

* 匿名函数

lambda只是一个表达式，函数体比def简单很多。

lambda的主体是一个表达式，而不是一个代码块。仅仅能在lambda表达式中封装有限的逻辑进去。

lambda函数拥有自己的命名空间，且不能访问自有参数列表之外或全局命名空间里的参数。

虽然lambda函数看起来只能写一行，却不等同于C或C++的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。

lambda函数的语法只包含一个语句，如下：

lambda [变量]：表达式

lambda [arg1 [,arg2,.....argn]]:expression

sum=lambda arg1,arg2: arg1+arg2  
print "相加后的值:",sum(10,20)

输出结果

相加后的值: 30

lambda x: x \* x

匿名函数的作用等同于下面的函数

def f(x):

return x \* x

* 递归

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数

递归函数的优点是定义简单，逻辑清晰。理论上，所有递归函数都可以写成循环的方式，但是循环的逻辑不如递归清晰。

使用递归函数需要注意防止栈溢出。在计算中，函数调用通过栈(stack)这种数据结构实现的，

每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。

由于栈的大小不是无限的，所以递归调用的次数过多，会导致栈溢出。

#递归函数  
def fact(n):  
 if n==1:  
 return 1  
 return n\*fact(n-1)  
print fact(5)  
  
# ===> fact(5)  
# ===> 5 \* fact(4)  
# ===> 5 \* (4 \* fact(3))  
# ===> 5 \* (4 \* (3 \* fact(2)))  
# ===> 5 \* (4 \* (3 \* (2 \* fact(1))))  
# ===> 5 \* (4 \* (3 \* (2 \* 1)))  
# ===> 5 \* (4 \* (3 \* 2))  
# ===> 5 \* (4 \* 6)  
# ===> 5 \* 24  
# ===> 120

* 迭代器

如果给定一个list或tuple，我们可以通过for循环来遍历这个list或tuple，这种遍历我们称为迭代（Iteration）。

如何判断一个对象是可迭代对象呢？方法是通过collections模块的Iterable类型判断：

#导入模块

from collections import Iterable  
print isinstance('abc',Iterable)  
print isinstance([1,2,3],Iterable)  
print isinstance(123,Iterable)

输出结果：

True

True

False

Iter(list)把列表变成迭代

迭代的调用方法next()

* 生成器

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器（Generator）

* map zip 匿名函数 sorted() filter() reduce()

#!/usr/bin/env python  
# -\*- coding:utf-8 -\*-   
# Author: songbing  
# Mail : songbing513@163.com  
  
#把用户名首字母大写  
name\_list=['adam', 'LISA', 'barT']  
print(name\_list)  
  
#循环方式  
for i in name\_list:  
 print i.capitalize()  
  
#列表表达式方式  
a=[x.capitalize() for x in name\_list ]  
print a  
  
#map 和匿名函数  
b=map(lambda x:x.capitalize(),name\_list)  
print b

匿名函数

larmbda x:x.capitalize()中larmbda x指定变量，

x.capitalize()对此变量操作的表达式

map()函数

接收两个参数，一个是函数，一个是序列，map将传入的函数依次作用到序列的每个元素，并把结果作为新的list返回

将序列中的每个元素作用到函数中并把函数处理结果返回新的list

def f(x):  
 return x\*x  
r=map(f,list(range(1,10)))  
print(list(r))

结果：

[1, 4, 9, 16, 25, 36, 49, 64, 81]

Filter()函数

接收两个参数，一个是函数，一个是序列，filter把序列中的每个元素交给函数执行，如果执行结果为真，把结果作为新list返回

num\_list=range(10)  
print filter(lambda x:x>=5,num\_list)

zip()函数

把两个列表按照最短类别合并

a=range(10)  
b=range(10,20)  
print zip(a,b)

输出结果：

[(0, 10), (1, 11), (2, 12), (3, 13), (4, 14), (5, 15), (6, 16), (7, 17), (8, 18), (9, 19)]

如果两个列表长度不一致时，zip只列出相同长度的合并集

使用map函数时候，可以列出所有合并集，但是需要指定None

a=range(10)  
b=range(10,15)  
print zip(a,b)  
print map(None,a,b)

输出结果：

[(0, 10), (1, 11), (2, 12), (3, 13), (4, 14)]

[(0, 10), (1, 11), (2, 12), (3, 13), (4, 14), (5, None), (6, None), (7, None), (8, None), (9, None)]

Reduce()函数

reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算

from functools import reduce  
def add(x,y):  
 return x+y  
  
print(reduce(add,[1,3,5,7,9]))

结果:

25

使用匿名函数实现

print(reduce(lambda x,y:x+y,[1,2,3,4]))

* 装饰器

把一个函数当做参数然后返回一个替代版函数

在不改变原有参数调用方式的情况下，增加函数的内容

#!/usr/bin/env python  
# -\*- coding:utf-8 -\*-   
# Author: songbing  
# Mail : songbing513@163.com  
  
def log(func):  
 def wrapper(\*args,\*\*kw):  
 print 'call %s():' % func.\_\_name\_\_  
 return func(\*args,\*\*kw)

#此处返回函数主体名，并不实际调用函数  
 return wrapper

#return func(\*args,\*\*kw)作用是把函数执行结果返回给wrapper函数

# return wrapper作用是只把函数主体名返回给log函数，并不实际执行wrapper函数  
#@log即装饰器作用等同于log(now('song','26'))

@log  
def now(name,age):  
 print '2016-6-22'  
 print name,age

#调用函数  
now('name','26')  
查看函数主体名，通过查看发现实际上now函数调用的主体函数是wrapper  
print now.\_\_name\_\_

输出结果：

call now():

2016-6-22

name 26

wrapper

## 异常处理

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
  
'''  
try来运行代码，如果执行出错，则后续代码不会继续执行，  
而是直接跳转至错误处理代码，即except语句块，执行完except后，  
如果有finally语句块，则执行finally语句块，至此，执行完毕  
语句无论是否发生异常都将执行finally  
'''  
try:  
 print('try....')  
 r=10/0  
 print('result:',r)  
except ZeroDivisionError as e:  
 print('except:',e)  
finally:  
 print('finally....')  
  
print('END')  
  
print('############################')  
  
'''  
  
如果发生了不同类型的错误，应该由不同的except语句块处理。  
可以有多个except来捕获不同类型的错误  
  
'''  
  
try:  
 print('try....')  
 r=10/int('a')  
 print('result:',r)  
except ValueError as e:  
 print('ValueError:',e)  
except ZeroDivisionError as e:  
 print('ZeroDivisionError',e)  
finally:  
 print('finally....')  
print('END')  
  
print('#############################')  
  
'''  
如果没有错误发生，可以在except语句块后面加一个else，  
当没有错误发生时，会自动执行else语句  
'''  
  
  
try:  
 print('try...')  
 r=10/int('2')  
 print('result:',r)  
except ValueError as e:  
 print('ValueError',e)  
except ZeroDivisionError as e:  
 print('ZeroDivisionError',e)  
else:  
 print('no error！')  
finally:  
 print('finally....')  
  
print('END')

输出结果：

try....

except: division by zero

finally....

END

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

try....

ValueError: invalid literal for int() with base 10: 'a'

finally....

END

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

try...

result: 5.0

no error！

finally....

END

* 常用的标准异常

|  |  |
| --- | --- |
| BaseException | 所有异常的基类 |
| SystemExit | 解释器请求退出 |
| KeyboardInterrupt | 用户中断执行(通常是输入^C) |
| Exception | 常规错误的基类 |
| StopIteration | 迭代器没有更多的值 |
| GeneratorExit | 生成器(generator)发生异常来通知退出 |
| StandardError | 所有的内建标准异常的基类 |
| ArithmeticError | 所有数值计算错误的基类 |
| FloatingPointError | 浮点计算错误 |
| OverflowError | 数值运算超出最大限制 |
| ZeroDivisionError | 除(或取模)零 (所有数据类型) |
| AssertionError | 断言语句失败 |
| AttributeError | 对象没有这个属性 |
| EOFError | 没有内建输入,到达EOF 标记 |
| EnvironmentError | 操作系统错误的基类 |
| IOError | 输入/输出操作失败 |
| OSError | 操作系统错误 |
| WindowsError | 系统调用失败 |
| ImportError | 导入模块/对象失败 |
| LookupError | 无效数据查询的基类 |
| IndexError | 序列中没有此索引(index) |
| KeyError | 映射中没有这个键 |
| MemoryError | 内存溢出错误(对于Python 解释器不是致命的) |
| NameError | 未声明/初始化对象 (没有属性) |
| UnboundLocalError | 访问未初始化的本地变量 |
| ReferenceError | 弱引用(Weak reference)试图访问已经垃圾回收了的对象 |
| RuntimeError | 一般的运行时错误 |
| NotImplementedError | 尚未实现的方法 |
| SyntaxError | Python 语法错误 |
| IndentationError | 缩进错误 |
| TabError | Tab 和空格混用 |
| SystemError | 一般的解释器系统错误 |
| TypeError | 对类型无效的操作 |
| ValueError | 传入无效的参数 |
| UnicodeError | Unicode 相关的错误 |
| UnicodeDecodeError | Unicode 解码时的错误 |
| UnicodeEncodeError | Unicode 编码时错误 |
| UnicodeTranslateError | Unicode 转换时错误 |
| Warning | 警告的基类 |
| DeprecationWarning | 关于被弃用的特征的警告 |
| FutureWarning | 关于构造将来语义会有改变的警告 |
| OverflowWarning | 旧的关于自动提升为长整型(long)的警告 |
| PendingDeprecationWarning | 关于特性将会被废弃的警告 |
| RuntimeWarning | 可疑的运行时行为(runtime behavior)的警告 |
| SyntaxWarning | 可疑的语法的警告 |
| UserWarning | 用户代码生成的警告 |

1. Logging 模块
2. Assert 断言
3. Pdb 调试器

## 类（面向对象）

### 1 面向对象技术

* **类(Class):**用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
* **类变量：**类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
* **数据成员：**类变量或者实例变量用于处理类及其实例对象的相关的数据。
* **方法重写：**如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称为方法的重写。
* **实例变量：**定义在方法中的变量，只作用于当前实例的类。
* **继承：**即一个派生类（derived class）继承基类（base class）的字段和方法。继承也允许把一个派生类的对象作为一个基类对象对待。例如，有这样一个设计：一个Dog类型的对象派生自Animal类，这是模拟"是一个（is-a）"关系（例图，Dog是一个Animal）。
* **实例化：**创建一个类的实例，类的具体对象。
* **方法：**类中定义的函数。
* **对象：**通过类定义的数据结构实例。对象包括两个数据成员（类变量和实例变量）和方法
* 创建类

使用class语句来创建一个新类，class之后为类的名称并以冒号结尾，如下实例:

class ClassName:  
 '类的帮助信息' #类文档字符串  
 class\_suite #类体

类的帮助信息可以通过ClassName.\_\_doc\_\_查看。

class\_suite 由类成员，方法，数据属性组成。

类实例：

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
class Employee:  
 '所有员工的基类'  
 empCount=0  
  
 def \_\_init\_\_(self,name,salary):  
 self.name=name  
 self.salary=salary  
 Employee.empCount+=1  
  
 def displayCount(self):  
 print("Total Employee %d" % Employee.empCount)  
  
 def displayEmployee(self):  
 print("Name: ",self.name,"Salary: ",self.salary)

要创建一个类的实例，你可以使用类的名称，并通过\_\_init\_\_方法接受参数。  
emp1=Employee('Zara',2000)  
emp2=Employee('Manni',5000)

可以使用点(.)来访问对象的属性。使用如下类的名称访问类变量:  
emp1.displayEmployee()  
emp1.displayEmployee()  
print("Total Employy %d" % Employee.empCount)

empCount变量是一个类变量，它的值将在这个类的所有实例之间共享。你可以在内部类或外部类使用Employee.empCount访问。

\_\_init\_\_()方法是一种特殊的方法，被称为类的构造函数或初始化方法，当创建了这个类的实例时就会调用该方法

* 访问限制

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线\_\_，在Python中，实例的变量名如果以\_\_开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问，就确保了外部代码不能随意修改对象内部的状态，这样通过访问限制的保护，代码更加健壮。

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
class Employee:  
 '所有员工的基类'  
 empCount=0  
  
 def \_\_init\_\_(self,name,salary):  
 # self.name=name  
 # self.salary=salary  
 self.\_\_name=name  
 self.\_\_salary=salary  
 Employee.empCount+=1  
  
 def displayCount(self):  
 print("Total Employee %d" % Employee.empCount)  
  
 def displayEmployee(self):  
 print("Name: ",self.\_\_name,"Salary: ",self.\_\_salary)  
  
  
emp1=Employee('Zara',2000)  
emp2=Employee('Manni',5000)  
  
emp1.displayEmployee()  
emp1.displayEmployee()  
print("Total Employy %d" % Employee.empCount)

如果外部代码需要获取name和salary可以增加获取的方法，或者增加修改name和salary的方法

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
class Employee:  
 '所有员工的基类'  
 empCount=0  
  
 def \_\_init\_\_(self,name,salary):  
 # self.name=name  
 # self.salary=salary  
 self.\_\_name=name  
 self.\_\_salary=salary  
 Employee.empCount+=1  
  
#增加外部获取属性的方法  
 def get\_name(self):  
 return self.\_\_name  
   
 def get\_salary(self):  
 return self.\_\_salary  
   
#增加外部修改属性的方法  
 def set\_name(self,name):  
 self.\_\_name=name  
   
 def set\_salary(self,salary):  
 self.\_\_salary=salary

def displayCount(self):  
 print("Total Employee %d" % Employee.empCount)  
  
 def displayEmployee(self):  
 print("Name: ",self.\_\_name,"Salary: ",self.\_\_salary)  
  
emp1=Employee('Zara',2000)  
emp2=Employee('Manni',5000)  
  
emp1.displayEmployee()  
emp1.displayEmployee()  
print("Total Employy %d" % Employee.empCount)

* 继承

当定义一个class的时候，可以从某个现有的class继承，新的class称为子类（Subclass），而被继承的class称为基类、父类或超类（Base class、Super class）。

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
  
#创建类Animal 并有一个run的方法  
class Animal(object):  
  
 def \_\_init\_\_(self,name,hair):  
 self.name=name  
 self.hair=hair  
  
 def colour(self):  
 print("%s skin color is %s" % (self.name,self.hair))  
  
 def run(self):  
 print("%s is running...." % self.name)  
  
  
#创建dog和cat类的时，可以直接从Animal类继承

#并且可以添加自己的属性和方法  
class Dog(Animal):  
 def cry(self):  
 print("%s 叫声汪汪" % self.name)  
  
class Cat(Animal):  
 pass  
  
#对于dog和cat,Animal就是父类，dog就是Animal的子类  
#子类自动继承父类的属性和方法  
  
  
dog=Animal("旺财","black")  
dog.colour()  
  
dog1=Dog("小黑",'black')  
dog1.colour()  
dog1.run()  
dog1.cry()

* **\_\_slots\_\_方法**

限制类的属性，只允许对实例添加固定的属性，

\_\_slots\_\_定义的属性仅对当前类起作用，对继承的子类是不起作用的

>>> class Student(object):

... \_\_slots\_\_=("name","age")

...

>>> s=Student()

>>> s.name='song'

>>> s.age=26

>>> s.score=99

Traceback (most recent call last):

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

AttributeError: 'Student' object has no attribute 'score'

>>>

由于score没有在\_\_slots\_\_中所以不能绑定属性

# 进阶篇

## 网络编程

### Socket

使用socket时需要指定Socket Family（地址簇），包括以下几种：

**socket.AF\_UNIX       只能够用于单一的Unix系统进程间通信**

**socket.AF\_INET      用于主机之间的网络通信**

**socket.AF\_INET6    IPv6通信**

若想实现主机之间的通信，我们就得使用socket.AF\_INET

**确认地址簇后，还需要指定socket 数据类型**

socket.SOCK\_STREAM     流式socket, for TCP

socket.SOCK\_DGRAM       数据报式socket, for UDP

socket.SOCK\_RAW    原始套接字，普通的套接字无法处理ICMP、IGMP等网络报文，而SOCK\_RAW可以；其次，SOCK\_RAW也可以处理特殊的IPv4报文；此外，利用原始套接字，可以通过IP\_HDRINCL套接字选项由用户构造IP头。

socket.SOCK\_RDM   是一种可靠的UDP形式，即保证交付数据报但不保证顺序。SOCK\_RAM用来提供对原始协议的低级访问，在需要执行某些特殊操作时使用，如发送ICMP报文。SOCK\_RAM通常仅限于高级用户或管理员运行的程序使用。

socket.SOCK\_SEQPACKET      可靠的连续数据包服务

**我们主要用的一般是SOCK\_STREAM (for TCP)和SOCK\_DGRAM（for UDP）.**

**套接字的实例具有以下方法：**

1. s.bind(address) 将套接字绑定到地址。address地址的格式取决于地址族。在AF\_INET下，以元组（host,port）的形式表示地址。
2. s.listen(backlog)   开始监听传入连接。backlog指定在拒绝连接之前，操作系统可以挂起的最大连接数量。该值至少为1，大部分应用程序设为5就可以了。
3. s.connect(address)  连接到address处的套接字。一般，address的格式为元组（hostname,port），如果连接同一台机器上的服务器，可以将hostname设为‘localhost’。如果连接出错，返回socket.error错误。
4. s.connect\_ex(adddress)  功能与connect(address)相同，但是成功返回0，失败返回errno的值。
5. s.accept() 接受连接并返回（conn,address）,其中conn是新的套接字对象，可以用来接收和发送数据。address是连接客户端的地址。
6. s.close()  关闭套接字。
7. s.fileno()  返回套接字的文件描述符。
8. s.getpeername() 返回连接套接字的远程地址。返回值通常是元组（ipaddr,port）。
9. s.getsockname()  返回套接字自己的地址。通常是一个元组(ipaddr,port)
10. s.getsockopt(level,optname[.buflen]) 返回套接字选项的值。
11. s.gettimeout() 返回当前超时期的值，单位是秒，如果没有设置超时期，则返回None。
12. s.recv(bufsize[,flag])  接受套接字的数据。数据以字符串形式返回，bufsize指定要接收的最大数据量。flag提供有关消息的其他信息，通常可以忽略。
13. s.recvfrom(bufsize[.flag])  与recv()类似，但返回值是（data,address）。其中data是包含接收数据的字符串，address是发送数据的套接字地址。
14. s.send(string[,flag])  将string中的数据发送到连接的套接字。返回值是要发送的字节数量，该数量可能小于string的字节大小。
15. s.sendall(string[,flag])  将string中的数据发送到连接的套接字，但在返回之前会尝试发送所有数据。成功返回None，失败则抛出异常。
16. s.sendto(string[,flag],address)  将数据发送到套接字，address是形式为（ipaddr，port）的元组，指定远程地址。返回值是发送的字节数。该函数主要用于UDP协议。
17. s.setblocking(flag)  如果flag为0，则将套接字设为非阻塞模式，否则将套接字设为阻塞模式（默认值）。非阻塞模式下，如果调用recv()没有发现任何数据，或send()调用无法立即发送数据，那么将引起socket.error异常。
18. s.setsockopt(level,optname,value)   设置给定套接字选项的值。
19. s.settimeout(timeout)   设置套接字操作的超时期，timeout是一个浮点数，单位是秒。值为None表示没有超时期。一般，超时期应该在刚创建套接字时设置，因为它们可能用于连接的操作（如connect()）普通的非套接字实例的函数
20. getdefaulttimeout()返回默认的套接字超时时间（以秒为单位）。None表示不设置任何超时时间。
21. gethostbyname(hostname)   将主机名（如“www.baidu.com”）转换为IPv4地址，IP地址将以字符串的形式返回，如“8.8.8.8”。不支持IPv6
22. gethostname() 返回本地机器的主机名。

出现网络联机错误Socket error #11001  
  
表示您的计算机无法连上服务器，请检查您的Proxy设定以及Proxy相关账号，或暂时取消您防毒软件的「个人防火墙」。  
  
· 出现网络联机错误Socket error #11004  
  
应该是网络联机状态问题，请用户检查网络联机是否正常。  
  
· 出现网络联机错误Socket error #10060 Connection Timed Out  
  
表示您与服务器发生联机逾时错误，请检查您的Proxy相关设定，以及Proxy Server是否已将SSL 443 port打开。  
  
· 出现网络联机错误Socket error #10061  
  
表示服务器忙碌中无法与您建立联机，请稍后再试。  
  
· 出现网络联机错误Socket error #10022  
  
请您上网更新Windows操作系统组件。Windows NT version 4.0的用户请下载安装 Service Pack 5(含)以上。  
  
· 出现Internal Server Error  
  
主机忙碌中，请您稍后再试。  
  
· 上传时出现Socket error #10022 Invalid argument.  
  
请上网更新Windows操作系统组件(windows update)。Windows NT version 4.0的用户请下载安装 Service Pack 5(含)以上。  
  
· 上传时出现socket error #10054 Connection reset by peer  
  
原因为连接被防火樯或proxy中断"或因为您有安装ip分享器请将ip分享器先拿掉，直接将计算机接adsl的线路后重新上传。  
  
· 上传时出现socket error #10057 Connection reset by peer  
  
原因为系统文件被破坏，请进行系统的杀毒，关闭杀毒软件、防火墙。

#### 1 Socket 类型

套接字格式：

socket(family,type[,protocal]) 使用给定的地址族、套接字类型、协议编号（默认为0）来创建套接字。

|  |  |
| --- | --- |
| socket类型 | 描述 |
| socket.AF\_UNIX | 只能够用于单一的Unix系统进程间通信 |
| socket.AF\_INET | 服务器之间网络通信 |
| socket.AF\_INET6 | IPv6 |
| socket.SOCK\_STREAM | 流式socket , for TCP |
| socket.SOCK\_DGRAM | 数据报式socket , for UDP |
| socket.SOCK\_RAW | 原始套接字，普通的套接字无法处理ICMP、IGMP等网络报文，而SOCK\_RAW可以；其次，SOCK\_RAW也可以处理特殊的IPv4报文；此外，利用原始套接字，可以通过IP\_HDRINCL套接字选项由用户构造IP头。 |
| socket.SOCK\_SEQPACKET | 可靠的连续数据包服务 |
| 创建TCP Socket： | s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM) |
| 创建UDP Socket： | s=socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM) |

#### 2 Socket 函数

注意点:

1）TCP发送数据时，已建立好TCP连接，所以不需要指定地址。UDP是面向无连接的，每次发送要指定是发给谁。

2）服务端与客户端不能直接发送列表，元组，字典。需要字符串化repr(data)。

|  |  |  |
| --- | --- | --- |
| socket函数 | | 描述 |
| 服务端socket函数 | | |
| s.bind(address) | 将套接字绑定到地址, 在AF\_INET下,以元组（host,port）的形式表示地址. | |
| s.listen(backlog) | 开始监听TCP传入连接。backlog指定在拒绝连接之前，操作系统可以挂起的最大连接数量。该值至少为1，大部分应用程序设为5就可以了。 | |
| s.accept() | 接受TCP连接并返回（conn,address）,其中conn是新的套接字对象，可以用来接收和发送数据。address是连接客户端的地址。 | |
| 客户端socket函数 | | |
| s.connect(address) | 连接到address处的套接字。一般address的格式为元组（hostname,port），如果连接出错，返回socket.error错误。 | |
| s.connect\_ex(adddress) | 功能与connect(address)相同，但是成功返回0，失败返回errno的值。 | |
| 公共socket函数 | | |
| s.recv(bufsize[,flag]) | 接受TCP套接字的数据。数据以字符串形式返回，bufsize指定要接收的最大数据量。flag提供有关消息的其他信息，通常可以忽略。 | |
| s.send(string[,flag]) | 发送TCP数据。将string中的数据发送到连接的套接字。返回值是要发送的字节数量，该数量可能小于string的字节大小。 | |
| s.sendall(string[,flag]) | 完整发送TCP数据。将string中的数据发送到连接的套接字，但在返回之前会尝试发送所有数据。成功返回None，失败则抛出异常。 | |
| s.recvfrom(bufsize[.flag]) | 接受UDP套接字的数据。与recv()类似，但返回值是（data,address）。其中data是包含接收数据的字符串，address是发送数据的套接字地址。 | |
| s.sendto(string[,flag],address) | 发送UDP数据。将数据发送到套接字，address是形式为（ipaddr，port）的元组，指定远程地址。返回值是发送的字节数。 | |
| s.close() | 关闭套接字。 | |
| s.getpeername() | 返回连接套接字的远程地址。返回值通常是元组（ipaddr,port）。 | |
| s.getsockname() | 返回套接字自己的地址。通常是一个元组(ipaddr,port) | |
| s.setsockopt(level,optname,value) | 设置给定套接字选项的值。 | |
| s.getsockopt(level,optname[.buflen]) | 返回套接字选项的值。 | |
| s.settimeout(timeout) | 设置套接字操作的超时期，timeout是一个浮点数，单位是秒。值为None表示没有超时期。一般，超时期应该在刚创建套接字时设置，因为它们可能用于连接的操作（如connect()） | |
| s.gettimeout() | 返回当前超时期的值，单位是秒，如果没有设置超时期，则返回None。 | |
| s.fileno() | 返回套接字的文件描述符。 | |
| s.setblocking(flag) | 如果flag为0，则将套接字设为非阻塞模式，否则将套接字设为阻塞模式（默认值）。非阻塞模式下，如果调用recv()没有发现任何数据，或send()调用无法立即发送数据，那么将引起socket.error异常。 | |
| s.makefile() | 创建一个与该套接字相关连的文件 | |

#### 3 socket编程思路

1. 第一步是创建socket对象。调用socket构造函数。如：

socket = socket.socket( family, type )

family参数代表地址家族，可为AF\_INET或AF\_UNIX。AF\_INET家族包括Internet地址，AF\_UNIX家族用于同一台机器上的进程间通信。

type参数代表套接字类型，可为SOCK\_STREAM(流套接字)和SOCK\_DGRAM(数据报套接字)。

2. 第二步是将socket绑定到指定地址。这是通过socket对象的bind方法来实现的：

socket.bind( address )

由AF\_INET所创建的套接字，address地址必须是一个双元素元组，格式是(host,port)。host代表主机，port代表端口号。如果端口号正在使用、主机名不正确或端口已被保留，bind方法将引发socket.error异常。

3. 第三步是使用socket套接字的listen方法接收连接请求。

socket.listen( backlog )

backlog指定最多允许多少个客户连接到服务器。它的值至少为1。收到连接请求后，这些请求需要排队，如果队列满，就拒绝请求。

4. 第四步是服务器套接字通过socket的accept方法等待客户请求一个连接。

connection, address = socket.accept()

调 用accept方法时，socket会时入“waiting”状态。客户请求连接时，方法建立连接并返回服务器。accept方法返回一个含有两个元素的 元组(connection,address)。第一个元素connection是新的socket对象，服务器必须通过它与客户通信；第二个元素 address是客户的Internet地址。

5. 第五步是处理阶段，服务器和客户端通过send和recv方法通信(传输 数据)。服务器调用send，并采用字符串形式向客户发送信息。send方法返回已发送的字符个数。服务器使用recv方法从客户接收信息。调用recv 时，服务器必须指定一个整数，它对应于可通过本次方法调用来接收的最大数据量。recv方法在接收数据时会进入“blocked”状态，最后返回一个字符 串，用它表示收到的数据。如果发送的数据量超过了recv所允许的，数据会被截短。多余的数据将缓冲于接收端。以后调用recv时，多余的数据会从缓冲区 删除(以及自上次调用recv以来，客户可能发送的其它任何数据)。

6. 传输结束，服务器调用socket的close方法关闭连接。

python编写client的步骤：

1. 创建一个socket以连接服务器：socket = socket.socket( family, type )

2.使用socket的connect方法连接服务器。对于AF\_INET家族,连接格式如下：

socket.connect( (host,port) )

host代表服务器主机名或IP，port代表服务器进程所绑定的端口号。如连接成功，客户就可通过套接字与服务器通信，如果连接失败，会引发socket.error异常。

3. 处理阶段，客户和服务器将通过send方法和recv方法通信。

4. 传输结束，客户通过调用socket的close方法关闭连接。



#### 4 Socket C/S实例

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

注意：接收信息encode('utf8') 和屏幕输出信息decode('utf8')指定编码格式标红的地方，如果不加此函数，socketserve接收或者回复的信息是b''，

是字节信息，不能进行通信

#encode将字符串变成字节

#decode将字节变成字符串

通过b将字符串变成字节

Socket Server实例（单线程实例）

#单线程循环接收发送信息

Python 2

服务端

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import socket  
from time import ctime  
  
HOST='127.0.0.1'  
PORT=8001  
BUFSIZE=1024  
ADDR=(HOST,PORT)  
  
tcpser=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  
tcpser.bind((ADDR))  
tcpser.listen(5)  
  
while True:  
 print "waiting connect ......"  
 conn,addr=tcpser.accept()  
 print addr  
 while True:  
 fdata=conn.recv(BUFSIZE)  
 if not fdata:  
 break  
 else:  
 print '[%s] %s ' % (ctime(),fdata)  
 sdata=conn.send('[%s] %s' % (ctime(),fdata))  
  
 conn.close()  
tcpser.close()

客户端

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import socket  
  
HOST='127.0.0.1'  
PORT=8001  
BUFSIZE=1024  
ADDR=(HOST,PORT)  
class Myclient():  
 def \_\_init\_\_(self):  
 self.tcpcli=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  
 self.tcpcli.connect((ADDR))  
  
 while True:  
 sdata=raw\_input(">")  
 if not sdata:  
 print "Not allowed to empty,please re-enter!"  
 continue  
 self.tcpcli.send(sdata)  
 fdata=self.tcpcli.recv(BUFSIZE)  
 print fdata  
 self.tcpcli.close()  
  
if \_\_name\_\_=="\_\_main\_\_":  
 client=Myclient()

Python 3

服务端

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import socket  
from time import ctime  
  
HOST='127.0.0.1' #定义IP地址  
PORT=12345 #定义端口  
BUFSIZE=1024 #定义发送字节  
ADDR=(HOST,PORT) #地址池  
  
sock=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM) #配置一个socket实例  
sock.bind(ADDR) #绑定socket到指定地址池  
sock.listen(5) #监听客户端连接  
  
  
while True:  
 try:  
 print('connnect.....................')  
 tcpClientSock,addr=sock.accept()  
 print('waiting for connection')  
 print('connect from',addr)  
 while True:  
 print('data......................')  
 data=tcpClientSock.recv(BUFSIZE)  
 s='Hi,you send me: [%s] %s' % (ctime(),data.decode('utf8'))  
 tcpClientSock.send(s.encode('utf8'))  
 print([ctime()],':',data.decode('utf8'))  
 except socket.error as e:  
 print(e)  
 tcpClientSock.close()  
 break  
 if not data:  
 break  
tcpClientSock.close()  
sock.close()

客户端

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
  
import socket  
  
class TcpClient:  
 HOST='127.0.0.1' #指定服务端地址  
 PORT=12345 #指定服务端端口  
 BUFSIZE=1024 #指定发送字节  
 ADDR=(HOST,PORT) #指定地址池

def \_\_init\_\_(self):  
 self.client=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM) #创建客户端实例  
 self.client.connect(self.ADDR) #连接服务端  
  
 while True:  
 data=input('>')  
 if not data:  
 break  
 self.client.send(data.encode('utf8'))  
 data=self.client.recv(self.BUFSIZE)  
 if not data:  
 break  
 print(data.decode('utf8'))  
  
if \_\_name\_\_=='\_\_main\_\_':  
 client=TcpClient()

### SocketServer

#多线程循环接收发送信息

Python 2

服务端

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import SocketServer  
from time import ctime  
  
class MyTCPHandler(SocketServer.BaseRequestHandler):  
 #继承BaseRequestHandler基类，然后必须重新handle方法，并在handle方法里面实现与客户端的所有交互  
  
 def handle(self):  
 while True:  
 data=self.request.recv(1024)#设置接收1024字节数据  
 if not data:  
 break  
 else:  
 print '[%s] %s' % (ctime(),data)  
 self.request.sendall('[%s] %s' % (ctime(),data))  
  
if \_\_name\_\_=="\_\_main\_\_":  
 HOST,PORT='127.0.0.1',8002  
 ADDR=(HOST,PORT)  
 server=SocketServer.ThreadingTCPServer(ADDR,MyTCPHandler)  
 server.serve\_forever()

客户端

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
  
import socket  
  
HOST='127.0.0.1'  
PORT=8002  
BUFSIZE=1024  
ADDR=(HOST,PORT)  
  
class Myclient():  
 def \_\_init\_\_(self):  
 self.tcpcli=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  
 self.tcpcli.connect((ADDR))  
  
 while True:  
 sdata=raw\_input(">")  
 if not sdata:  
 print "Not allowed to empty,please re-enter!"  
 continue  
 self.tcpcli.send(sdata)  
 fdata=self.tcpcli.recv(BUFSIZE)  
 print fdata  
 self.tcpcli.close()  
  
if \_\_name\_\_=="\_\_main\_\_":  
 client=Myclient()

Python 3

服务端

注意标红的地方：不能直接引用变量以下两种方式错误

fdata='ceshi'  
#self.request.sendall('%s' % fdata.encode('utf8'))  
#self.request.sendall('%s.encode(utf8)' % fdata)

正确方式，直接通过变量把需要发送的数据配置好，直接用过send调用变量

需要指定encode(utf8)把字符串转换为字节

fdata='%s %s' % (ctime(),sdata.decode('utf8'))  
self.request.sendall(fdata.encode('utf8'))

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import socketserver  
from time import ctime  
class MyTcpServer(socketserver.BaseRequestHandler):  
 def handle(self):  
 self.bufsize=1024  
 while True:  
 print('waiting connect......')  
 sdata=self.request.recv(self.bufsize)  
 print(sdata)  
 if not sdata:  
 break  
 else:  
 print('[%s] %s ' % (ctime(),sdata.decode('utf8')))  
  
 fdata='%s %s' % (ctime(),sdata.decode('utf8'))  
 self.request.sendall(fdata.encode('utf8'))  
  
if \_\_name\_\_=="\_\_main\_\_":  
 HOST,PORT='127.0.0.1',8003  
 ADDR=(HOST,PORT)  
 server=socketserver.ThreadingTCPServer(ADDR,MyTcpServer)  
 server.serve\_forever()

客户端

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import socket  
  
  
class MyClient():  
 def \_\_init\_\_(self):  
 self.HOST='127.0.0.1'  
 self.PORT=8003  
 self.SIZE=1024  
 self.tcpClient=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  
 self.tcpClient.connect((self.HOST,self.PORT))  
  
 while True:  
 sdata=input('>')  
 if not sdata:  
 print("Not allowed to empty,please re-enter!")  
 continue  
 self.tcpClient.sendall(sdata.encode('utf8'))  
 fdata=self.tcpClient.recv(self.SIZE)  
 print(fdata.decode('utf8'))  
 self.tcpClient.close()  
  
if \_\_name\_\_=="\_\_main\_\_":  
 client=MyClient()

#### 1 SocketServer

SocketServer简化了网络服务器的编写。它有4个类：TCPServer，UDPServer，UnixStreamServer，UnixDatagramServer。这4个类是同步进行处理的，另外通过ForkingMixIn和ThreadingMixIn类来支持异步。

创建服务器的步骤:

首先，你必须创建一个请求处理类，它是BaseRequestHandler的子类并重载其handle()方法。

其次，你必须实例化一个服务器类，传入服务器的地址和请求处理程序类。最后，调用handle\_request()(一般是调用其他事件循环或者使用select())或serve\_forever()。

集成ThreadingMixIn类时需要处理异常关闭。daemon\_threads指示服务器是否要等待线程终止，要是线程互相独立，必须要设置为True，默认是False。

无论用什么网络协议，服务器类有相同的外部方法和属性。

该模块在python3中已经更名为socketserver。

#### 2 服务器类型

5种类型：BaseServer，TCPServer，UnixStreamServer，UDPServer，UnixDatagramServer。 注意：BaseServer不直接对外服务。

#### 3 服务器对象

class SocketServer.BaseServer：这是模块中的所有服务器对象的超类。它定义了接口，如下所述，但是大多数的方法不实现，在子类中进行细化。

BaseServer.fileno()：返回服务器监听套接字的整数文件描述符。通常用来传递给select.select(), 以允许一个进程监视多个服务器。

BaseServer.handle\_request()：处理单个请求。处理顺序：get\_request(), verify\_request(), process\_request()。如果用户提供handle()方法抛出异常，将调用服务器的handle\_error()方法。如果self.timeout内没有请求收到， 将调用handle\_timeout()并返回handle\_request()。

BaseServer.serve\_forever(poll\_interval=0.5): 处理请求，直到一个明确的shutdown()请求。每poll\_interval秒轮询一次shutdown。忽略self.timeout。如果你需要做周期性的任务，建议放置在其他线程。

BaseServer.shutdown()：告诉serve\_forever()循环停止并等待其停止。python2.6版本。

BaseServer.address\_family: 地址家族，比如socket.AF\_INET和socket.AF\_UNIX。

BaseServer.RequestHandlerClass：用户提供的请求处理类，这个类为每个请求创建实例。

BaseServer.server\_address：服务器侦听的地址。格式根据协议家族地址的各不相同，请参阅socket模块的文档。

BaseServer.socketSocket：服务器上侦听传入的请求socket对象的服务器。

服务器类支持下面的类变量：

BaseServer.allow\_reuse\_address：服务器是否允许地址的重用。默认为false ，并且可在子类中更改。

BaseServer.request\_queue\_size

请求队列的大小。如果单个请求需要很长的时间来处理，服务器忙时请求被放置到队列中，最多可以放request\_queue\_size个。一旦队列已满，来自客户端的请求将得到 “Connection denied”错误。默认值通常为5 ，但可以被子类覆盖。

BaseServer.socket\_type：服务器使用的套接字类型; socket.SOCK\_STREAM和socket.SOCK\_DGRAM等。

BaseServer.timeout：超时时间，以秒为单位，或 None表示没有超时。如果handle\_request()在timeout内没有收到请求，将调用handle\_timeout()。

下面方法可以被子类重载，它们对服务器对象的外部用户没有影响。

BaseServer.finish\_request()：实际处理RequestHandlerClass发起的请求并调用其handle()方法。 常用。

BaseServer.get\_request()：接受socket请求，并返回二元组包含要用于与客户端通信的新socket对象，以及客户端的地址。

BaseServer.handle\_error(request, client\_address)：如果RequestHandlerClass的handle()方法抛出异常时调用。默认操作是打印traceback到标准输出，并继续处理其他请求。

BaseServer.handle\_timeout()：超时处理。默认对于forking服务器是收集退出的子进程状态，threading服务器则什么都不做。

BaseServer.process\_request(request, client\_address) :调用finish\_request()创建RequestHandlerClass的实例。如果需要，此功能可以创建新的进程或线程来处理请求,ForkingMixIn和ThreadingMixIn类做到这点。常用。

BaseServer.server\_activate()：通过服务器的构造函数来激活服务器。默认的行为只是监听服务器套接字。可重载。

BaseServer.server\_bind()：通过服务器的构造函数中调用绑定socket到所需的地址。可重载。

BaseServer.verify\_request(request, client\_address)：返回一个布尔值，如果该值为True ，则该请求将被处理，反之请求将被拒绝。此功能可以重写来实现对服务器的访问控制。默认的实现始终返回True。client\_address可以限定客户端，比如只处理指定ip区间的请求。 常用。

#### 4 请求处理器

处理器接收数据并决定如何操作。它负责在socket层之上实现协议（i.e., HTTP, XML-RPC, or AMQP)，读取数据，处理并写反应。可以重载的方法如下：

setup(): 准备请求处理. 默认什么都不做，StreamRequestHandler中会创建文件类似的对象以读写socket.

handle(): 处理请求。解析传入的请求，处理数据，并发送响应。默认什么都不做。常用变量：self.request，self.client\_address，self.server。

finish(): 环境清理。默认什么都不做，如果setup产生异常，不会执行finish。

通常只需要重载handle。self.request的类型和数据报或流的服务不同。对于流服务，self.request是socket 对象；对于数据报服务，self.request是字符串和socket 。可以在子类StreamRequestHandler或DatagramRequestHandler中重载，重写setup()和finish() ，并提供self.rfile和self.wfile属性。 self.rfile和self.wfile可以读取或写入，以获得请求数据或将数据返回到客户端。

#### 5 ftp服务器

Socketserver ftp功能

ftp 服务端

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import SocketServer  
import os  
  
class MyTCPHandler(SocketServer.BaseRequestHandler):  
 def handle(self):  
 while True:  
 instrunction=self.request.recv(1024).strip() #设置接收信息，并去除空格  
 if not instrunction:break #判断是否为空，如果为空跳出循环  
 instrunction=instrunction.split('|') #根据|进行分割字符串  
 print "输出分割以后的字符串列表",instrunction  
 if hasattr(self,instrunction[0]): #判断类中是否有这个方法  
 func=getattr(self,instrunction[0]) #调用类中的方法  
 func(instrunction)  
  
 def FileTransfer(self,msg):  
 print "---filetransfer----",msg  
 if msg[1]=='get':  
 print "client wants to download file:",msg[2]  
 if os.path.isfile(msg[2]):  
 file\_size=os.path.getsize(msg[2])  
 file\_size=int(file\_size)  
 res="ready|%s" % file\_size  
 else:  
 res="file doesn't exist"  
  
 send\_confirmation="FileTransfer|get|%s" % res  
 self.request.send(send\_confirmation)  
 feedback=self.request.recv(1024)  
 print "准备下载1111111111111111111111111111111"  
 if feedback=="FileTransfer|get|recv\_ready":  
 f=file(msg[2],'rb')  
 send\_szie=0  
 while not file\_size==send\_szie:  
 print "输出文件大小和发送文件大小",file\_size,send\_szie  
 print "输出格式",type(file\_size),type(send\_szie)  
 if file\_size-send\_szie>1024:  
 data=f.read(1024)  
 send\_szie+=1024  
 print "计算文件22222222222222222222222222222"  
 else:  
 print "计算文件111111111111111111111111111"  
 data=f.read(file\_size-send\_szie)  
 send\_szie+=(file\_size-send\_szie)  
 print "计算以后的文件大小1111111",send\_szie  
 print "发送文件111111111111111111111111111"  
 self.request.send(data)  
 else:  
 print "----send file:%s done-----" % msg[2]  
 else:  
 print "出错"  
  
if \_\_name\_\_=="\_\_main\_\_":  
 HOST,PORT='127.0.0.1',9002  
 server=SocketServer.ThreadingTCPServer((HOST,PORT),MyTCPHandler)  
 server.serve\_forever()

ftp 客户端

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import socket  
import os  
class FtpClinet(object):  
 def \_\_init\_\_(self,host,port):  
 self.sock=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  
 self.sock.connect((host,port))  
  
 def start(self):  
 self.interactive()  
  
 def interactive(self):  
 while True:  
 user\_input=raw\_input(">>").strip()  
 if len(user\_input)==0:continue  
 user\_input=user\_input.split()  
  
 if hasattr(self,user\_input[0]):  
 func=getattr(self,user\_input[0])  
 func(user\_input)  
 else:  
 print "\033{31;1mWrong cmd usage \033[0m"  
  
 def get(self,msg):  
 print "--get func--",msg  
 if len(msg)==2:  
 file\_name=msg[1]  
 instrunction="FileTransfer|get|%s" % file\_name  
 self.sock.send(instrunction)  
 feedback=self.sock.recv(1024)  
 print "-->",feedback  
 if feedback.startswith("FileTransfer|get|ready"):  
 file\_size=int(feedback.split("|")[-1])  
 print "接收文件大小",file\_size,type(file\_size)  
 self.sock.send("FileTransfer|get|recv\_ready")  
 recv\_size=0  
 f=file("client\_recv/%s" % os.path.basename(file\_name),'wb')  
 print "--->",file\_name  
 while not file\_size==recv\_size:  
 if file\_size-recv\_size>1024:  
 data=self.sock.recv(1024)  
 recv\_size+=len(data)  
 print "下载文件222222222222222222222222222"  
 else:  
 data=self.sock.recv(file\_size-recv\_size)  
 recv\_size+=(file\_size-recv\_size)  
 f.write(data)  
 print file\_size,recv\_size  
 else:  
 print "---recv file:%s----" % file\_name  
 f.close()  
 else:  
 print "2222222222222222222222222222222222222222222222"  
 print feedback  
 else:  
 print "\-33[021;1mWrong cmd usage \033[0m"  
  
if \_\_name\_\_=="\_\_main\_\_":  
 f=FtpClinet('127.0.0.1',9002)  
 f.start()

使用ftplib编写ftp客户端

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import ftplib  
import os  
import socket  
  
  
HOST='127.0.0.1'  
FILE='1.gz'  
bufsize=1024  
file\_handle=open(FILE,'wb').write  
print file\_handle  
  
def main():  
 try:  
 f=ftplib.FTP() #实例化对象  
 except (socket.error,socket.gaierror) as e:  
 print 'error: cannont reach "%s"' % HOST  
 return  
 print '\*\*\* Connected to host "%s"' % HOST  
  
 try:  
 f.connect(HOST,21)  
 f.login("song","123456") #登录接口，如果是非21端口需要先使用f.connect('127.0.0.1','port')登录  
 f.dir()#显示目录内容  
 except ftplib.error\_perm:  
 print 'Eorror: canont login "anonymous"'  
 f.quit()  
 return  
 print '\*\*\* logged is as "song"'  
  
  
 try:  
 f.retrbinary('RETR %s' % (FILE),file\_handle,bufsize) #下载文件 上传文件ftp.storbinaly("STOR filename.txt",file\_handel,bufsize)  
 except ftplib.error\_perm:  
 print 'Error: canoncet read file "%s"' % FILE  
 os.unlink(FILE)  
 else:  
 print '\*\*\* Downloadwd "%s" to CWD' % FILE  
 f.quit()  
if \_\_name\_\_=="\_\_main\_\_":  
 main()

### Twisted

## 进程、线程

### 线程

#### 1 Thread模块

* thread 模块函数

thread.start\_new\_thread ( function, args[, kwargs] )

* function - 线程主体函数。
* args - 传递给线程函数的参数,他必须是个tuple类型,没有参数,空数组
* kwargs - 可选参数。

allocate\_lock()

* 分配LockType锁对象。

exit()

* 给线程退出指令
* LockType锁对象方法

acquire(wait=None)

* 尝试获取锁对象

locked()

* 如获取锁了锁对象则返回True 否则返回False

release()

* 释放锁

单线程执行循环

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
from time import sleep,ctime  
  
def loop():  
 print "start loop 0 at:",ctime()  
 sleep(4)  
 print "loop 0 done at:",ctime()  
  
def loop1():  
 print "start loop 1 at:",ctime()  
 sleep(2)  
 print "loop1 done at:",ctime()  
  
def main():  
 print "starting at:",ctime()  
 loop()  
 loop1()  
 print "all DONE at:",ctime()  
  
if \_\_name\_\_=="\_\_main\_\_":  
 main()

#单线程循环中连续执行两个循环，一个循环必须在另个循环执行完成以后开始运行，总共消耗的时间是每个循环所用的时间之和。

使用thread模块实现简单多线程

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import thread  
from time import sleep,ctime  
  
def loop(new):  
 print "start loop 0 at:",ctime()  
 sleep(4)  
 print "loop 0 done at:",ctime()  
 print new

def loop1(new):  
 print "start loop 1 at:",ctime()  
 print new  
 sleep(2)  
 print "loop 1 done at:",ctime()  
  
def main():  
 print "All start at:",ctime()  
 thread.start\_new\_thread(loop,(111111111111111,))  
 thread.start\_new\_thread(loop1,(22222222222222222,))  
 sleep(6)  
 print "All done at:",ctime()  
  
if \_\_name\_\_=="\_\_main\_\_":  
 main()

#两个循环并发执行，总的运行时间与执行慢的线程相关

线程和锁

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import thread  
from time import sleep,ctime  
  
loops=[4,2]  
  
def loop(nloop,nsec,lock):  
 print "start loop",nloop,'at:',ctime()  
 sleep(nsec)  
 print "loop",nloop,'done at:',ctime()  
 lock.release()#释放锁  
 print "释放锁"  
  
def main():  
 print "staring at:",ctime()  
 locks=[]  
 nloops=range(len(loops))  
  
 for i in nloops:  
 print "分配锁对象"  
 lock=thread.allocate\_lock()#分配lockType锁对象  
 print "获取锁对象"  
 lock.acquire()#获取锁对象  
 locks.append(lock)  
 print "输出列表信息",locks  
  
  
 for i in nloops:

print "派生新线程"   
 thread.start\_new\_thread(loop,(i,loops[i],locks[i]))  
  
 for i in nloops:  
 while locks[i].locked():  
 pass  
 #print "成功获取锁对象" #如果获取锁对象返回Ture,否则返回False  
 else:  
 print "获取锁对象失败"  
  
 print "all DONE at:",ctime()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

#### 2 Threading模块

* threading模块对象

Thread 线程类，这是我们用的最多的一个类，你可以指定线程函数执行或者继承自它都可以实现子线程功能；

Timer与Thread类似，但要等待一段时间后才开始运行；

Lock 锁原语，这个我们可以对全局变量互斥时使用；

RLock 可重入锁，使单线程可以再次获得已经获得的锁；

Condition 条件变量，能让一个线程停下来，等待其他线程满足某个“条件”；

Event 通用的条件变量。多个线程可以等待某个事件发生，在事件发生后，所有的线程都被激活；

Semaphore为等待锁的线程提供一个类似“等候室”的结构；

BoundedSemaphore 与semaphore类似，但不允许超过初始值；

Queue：实现了多生产者（Producer）、多消费者（Consumer）的队列，支持锁原语，能够在多个线程之间提供很好的同步支持。

* **其中Thread类**

Thread对象属性属性：

name 线程名称

ident 线程的标识符

daemon 布尔标志，表示这个线程是否是守护线程

thread 对象方法：

start(self) 开始线程执行

join(self, timeout=None) 程序挂起，直到线程结束，如果给出timeout，则最多阻塞timeout秒

run(self) 定义线程的功能函数

getName(self) 返回线程的名字,该方法已经弃用thread.name代替

setName(self, name) 设置线程的名字,该方法已经弃用thread.name代替

setDaemon(self, daemonic) 把线程的daemon标志设为daemonic

isDaemon(self) 返回线程的daemon标志

is/setDaemon在python3中被弃用设置属性thread.daemon

isAlive(self) 布尔标志，表示这个线程是否还在运行中从2.6以后被弃用

* 创建线程的方法

在Python中我们主要是通过thread和threading这两个模块来实现的，其中Python的threading模块是对thread做了一些包装的，可以更加方便的被使用，所以我们使用threading模块实现多线程编程。一般来说，使用线程有两种模式，一种是创建线程要执行的函数，把这个函数传递进Thread对象里，让它来执行；另一种是直接从Thread继承，创建一个新的class，把线程执行的代码放到这个新的 class里。

将函数传递进Thread对象

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import threading  
from time import sleep,ctime  
  
def thread\_fun(num):  
 for n in range(0,int(num)):  
 print "I com from %s,num: %s time: %s" % (threading.currentThread().getName(),n,ctime())  
 sleep(3)  
  
  
  
def main(thread\_num):  
 thread\_lists=list()  
 #创建线程对象  
 for i in range(0,thread\_num):  
 thread\_name="thread\_%s" %i  
 thread\_lists.append(threading.Thread(target=thread\_fun,name=thread\_name,args=(20,)))  
  
 #启动所有线程  
 for thread in thread\_lists:  
 thread.start()  
 print "启动线程成功"  
   
 #主线程中等待所有子线程退出  
 for thread in thread\_lists:  
 thread.join()  
  
"""  
　　参数target是一个可调用对象（也称为活动[activity]），在线程启动后执行；  
　　参数name是线程的名字。默认值为“Thread-N“，N是一个数字。  
　　参数args和kwargs分别表示调用target时的参数列表和关键字参数  
  
"""  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main(3)

继承自threading.Thread类

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import threading  
  
  
class MyThread(threading.Thread):  
 def \_\_init\_\_(self):  
 threading.Thread.\_\_init\_\_(self)  
  
 def run(self):  
 print "I am %s " % self.name  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 for i in range(0,5):  
 my\_thread=MyThread()  
 my\_thread.start()

* 互斥锁

Python编程中，引入了对象互斥锁的概念，来保证共享数据操作的完整性。每个对象都对应于一个可称为” 互斥锁” 的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。在Python中我们使用threading模块提供的Lock类。

* 未加锁情况

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
#互斥锁现象重现  
  
  
import threading  
import time  
  
counter=0  
  
class MyThread(threading.Thread):  
 def \_\_init\_\_(self):  
 threading.Thread.\_\_init\_\_(self)  
  
 def run(self):  
 global counter  
 time.sleep(1)  
 counter+=1  
 print "I am %s,set counter:%s" %(self.name,counter)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 for i in range(0,500):  
 my\_thread=MyThread()  
 my\_thread.start()

输出结果：

I am Thread-487,set counter:478I am Thread-484,set counter:479I am Thread-483,set counter:480

I am Thread-486,set counter:481I am Thread-485,set counter:482

I am Thread-492,set counter:483

I am Thread-491,set counter:484

I am Thread-488,set counter:485

I am Thread-489,set counter:486

I am Thread-490,set counter:487

I am Thread-496,set counter:488

I am Thread-494,set counter:489I am Thread-497,set counter:490I am Thread-493,set counter:491

I am Thread-498,set counter:492

I am Thread-495,set counter:493

I am Thread-499,set counter:494

I am Thread-500,set counter:495

造成这个问题的原因：

"""  
从中我们已经看出了这个全局资源(counter)被抢占的情况，  
问题产生的原因就是没有控制多个线程对同一资源的访问，  
对数据造成破坏，使得线程运行的结果不可预期。  
这种现象称为“线程不安全”。  
  
"""

* 添加互斥锁以后的情况

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
"""  
需要添加一个互斥锁变量mutex = threading.Lock()，  
然后在争夺资源的时候之前我们会先抢占这把锁mutex.acquire()，  
对资源使用完成之后我们在释放这把锁mutex.release()  
"""  
  
import threading  
import time  
mutex=threading.Lock()  
  
counter=0  
  
class MyThread(threading.Thread):  
 def \_\_init\_\_(self):  
 threading.Thread.\_\_init\_\_(self)  
  
 def run(self):  
 global counter  
 time.sleep(1)  
 if mutex.acquire():  
 counter+=1  
 print "I am %s, set counter: %s" % (self.name,counter)  
 mutex.release()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 for i in range(0,500):  
 my\_thread=MyThread()  
 my\_thread.start()

* 同步阻塞

当一个线程调用Lock对象的acquire()方法获得锁时，这把锁就进入“locked”状态。因为每次只有一个线程1可以获得锁，所以如果此时另一个线程2试图获得这个锁，该线程2就会变为“block“同步阻塞状态。直到拥有锁的线程1调用锁的release()方法释放锁之后，该锁进入“unlocked”状态。线程调度程序从处于同步阻塞状态的线程中选择一个来获得锁，并使得该线程进入运行（running）状态。

进一步考虑

通过对公共资源使用互斥锁，这样就简单的到达了我们的目的，但是如果我们又遇到下面的情况：

* 1、遇到锁嵌套的情况该怎么办，这个嵌套是指当我一个线程在获取临界资源时，又需要再次获取；
* 2、如果有多个公共资源，在线程间共享多个资源的时候，如果两个线程分别占有一部分资源并且同时等待对方的资源；

上述这两种情况会直接造成程序挂起，即死锁，下面我们会谈死锁及可重入锁RLock。

* 死锁的形成
* 使用threading模块实现多线程编程四[使用Lock互斥锁]我们已经开始涉及到如何使用互斥锁来保护我们的公共资源了，现在考虑下面的情况–  
  如果有多个公共资源，在线程间共享多个资源的时候，如果两个线程分别占有一部分资源并且同时等待对方的资源，这会引起什么问题？
* 死锁概念  
  所谓死锁： 是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。 由于资源占用是互斥的，当某个进程提出申请资源后，使得有关进程在无外力协助下，永远分配不到必需的资源而无法继续运行，这就产生了一种特殊现象死锁。

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
#死锁重现  
  
  
import threading  
counterA=0  
counterB=0  
  
mutexA=threading.Lock()  
mutexB=threading.Lock()  
  
  
class MyThread(threading.Thread):  
 def \_\_init\_\_(self):  
 threading.Thread.\_\_init\_\_(self)  
  
 def run(self):  
 self.fun1()  
 self.fun2()  
  
 def fun1(self):  
 global mutexA,mutexB  
 if mutexA.acquire():  
 print "I am %s,get res: %s" % (self.name,"ResA")  
  
 if mutexB.acquire():  
 print "I am %s,get res:%s" %(self.name,"ResB")  
 mutexB.release()  
 mutexA.release()  
  
 def fun2(self):  
 global mutexA,mutexB  
 if mutexB.acquire():  
 print "I am %s,get res: %s" % (self.name,"ResB")  
  
  
 if mutexA.acquire():  
 print "I am %s,get res: %s" %(self.name,"ResA")  
 mutexA.release()  
  
 mutexB.release()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 for i in range(0,200):  
 my\_thread=MyThread()  
 my\_thread.start()

* 可重入锁Rlock

如果一个线程遇到锁嵌套的情况该怎么办，这个嵌套是指当我一个线程在获取临界资源时，又需要再次获取。

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import threading  
import time  
  
couter=0  
  
mutex=threading.Lock()  
  
  
class MyThread(threading.Thread):  
 def \_\_init\_\_(self):  
 threading.Thread.\_\_init\_\_(self)  
  
 def run(self):  
 global couter,mutex  
 time.sleep(1)  
 if mutex.acquire():  
 couter+=1  
 print "I am %s,set couter: %s" % (self.name,couter)  
 if mutex.acquire():  
 couter+=1  
 print "I am %s,set couter %s" %(self.name,couter)  
 mutex.release()  
 mutex.release()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 for i in range(0,100):  
 my\_thread=MyThread()  
 my\_thread.start()

输出结果：

I am Thread-1,set couter: 1

程序直接挂起，这种情况就形成了简单的死锁

在Python中为了支持在同一线程中多次请求同一资源，python提供了“可重入锁”：threading.RLock。这个RLock内部维护着一个Lock和一个counter变量，counter记录了acquire的次数，从而使得资源可以被多次require。直到一个线程所有的acquire都被release，其他的线程才能获得资源。上面的例子如果使用RLock代替Lock，则不会发生死锁：

#!/usr/bin/env python  
# -\*- coding: UTF-8 -\*-  
  
import threading  
import time  
  
couter=0

#将Lock替换成RLock  
#mutex=threading.Lock()  
mutex=threading.RLock()  
  
class MyThread(threading.Thread):  
 def \_\_init\_\_(self):  
 threading.Thread.\_\_init\_\_(self)  
  
 def run(self):  
 global couter,mutex  
 time.sleep(1)  
 if mutex.acquire():  
 couter+=1  
 print "I am %s,set couter: %s" % (self.name,couter)  
 if mutex.acquire():  
 couter+=1  
 print "I am %s,set couter %s" %(self.name,couter)  
 mutex.release()  
 mutex.release()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 for i in range(0,100):  
 my\_thread=MyThread()  
 my\_thread.start()