

# Python基础

2017年8月18日 星期五 下午11:11

在Python程序中，变量是用一个变量名表示，变量名必须是大小写英文、数字和下划线

在Python中，等号=是赋值语句，可以把任意数据类型赋值给变量，同一个变量可以

常用的转义字符还有：

`\n` 表示换行

`\t` 表示一个制表符

`\\` 表示 `\` 字符本身

如果一个字符串包含很多需要转义的字符，对每一个字符都进行转义会很麻烦。为了避免这种情况，我里面的字符就不需要转义了。例如：

```
r'\(~~)/ \(\~~)/'
```

但是`r'...'`表示法不能表示多行字符串，也不能表示包含`'`和`"`的字符串（为什么？）

如果要表示多行字符串，可以用`'''...'''`表示：

```
'''Line 1
Line 2
Line 3'''
```

上面这个字符串的表示方法和下面的是完全一样的：

```
'Line 1\nLine 2\nLine 3'
```

还可以在多行字符串前面添加 `r`，把这个多行字符串也变成一个raw字符串：

~~Python在后来添加了对Unicode的支持，以Unicode表示的字符串用`u'...'`表示，比如：~~

```
print u'中文'
中文
```

~~注意：不加`u`，中文就不能正常显示。~~

如果中文字符串在Python环境下遇到 `UnicodeDecodeError`，这是因为.py文件保存的格式有问题。可以

```
# -*- coding: utf-8 -*-
```

因为Python把`0`、空字符串`''`和`None`看成 `False`，其他数值和非空字符串都看成 `True`

短路计算：

(\_) 的组合，且不能用数字开头

从反复赋值，而且可以是不同类型的变量

们可以在字符串前面加个前缀 `r`，表示这是一个 raw 字符串，

以在第一行添加注释

1. 在计算 `a and b` 时，如果 `a` 是 `False`，则根据与运算法则，整个结果必定为 `False`，因此返回 `a`；
2. 在计算 `a or b` 时，如果 `a` 是 `True`，则根据或运算法则，整个计算结果必定为 `True`，因此返回 `a`。

所以Python解释器在做布尔运算时，只要能提前确定计算结果，它就不会往后算了，直接返回结果。

由于Python是动态语言，所以

中包含的元素并不要求都必须是同一种数据类型，我们完全可以在list中包含各种数

使用索引时，**千万注意不要越界**

### 添加

`append()`总是把新的元素添加到 `list` 的尾部。

方法是用list的 `insert()`方法，它接受两个参数，第一个参数是索引号，第二个参数

### 删除

`pop()`方法总是删掉list的最后一个元素，并且它还返回这个元素，所以我们执行 `L.pop`

要把Paul踢出list，我们就必须先定位Paul的位置。由于Paul的索引是2，因此，用 `p`

### 替换

对list中的某一个索引赋值，就可以直接用新的元素替换掉原来的元素，list包含的元

**tuple**是另一种有序的列表，中文翻译为“元组”。`tuple` 和 `list` 非常类似，但是，`tuple`一旦创建完毕，同样是表示班里同学的名

用`tuple`表示如下：

```
>>> t = ('Adam', 'Lisa', 'Bart')
```

创建`tuple`和创建`list`唯一不同之处是用`()`替代了`[]`。

现在，这个 `t` 就不能改变了，`tuple`没有 `append()`方法，也没有`insert()`和`pop()`方法。所以，新同学没

获取 `tuple` 元素的方式和 `list` 是一模一样的，我们可以正常使用 `t[0]`，`t[-1]`等索引方式访问元素，但是

创建包含1个元素的 `tuple` 呢？来试试：

```
>>> t = (1)
```

```
>>> print t
```

```
1
```

好像哪里不对！`t` 不是 `tuple`，而是整数1。为什么呢？

因为`()`既可以表示`tuple`，又可以作为括号表示运算时的优先级，结果 (1) 被Python解释器计算出结果 1

正是因为用`()`定义单元素的`tuple`有歧义，所以 Python 规定，单元素 `tuple` 要多加一个逗号“`,`”，这样就

```
>>> t = (1,)
```

如果 a 是 True，则整个计算结果必定取决与 b，因此返回 b。  
a；如果 a 是 False，则整个计算结果必定取决于 b，因此返回

居

改是待添加的新元素;

p() 后，会打印出 'Paul'。

pop(2)把Paul删掉

素个数保持不变。

就不能修改了。

没法直接往 tuple 中添加，老同学想退出 tuple 也不行。

不能赋值成别的元素

1，导致我们得到的不是tuple，而是整数 1。

避免了歧义：

```
>>> print t
(1,)
```

Python在打印单元素tuple时，也自动添加了一个“,”，为了更明确地告诉你这是一个tuple。

tuple所谓的“不变”是说，tuple的每个元素，指向永远不变。即指向'a'，就不能改成指向别的这个list本身是可变的！

**Python代码的缩进规则**。具有相同缩进的代码被视为代码块，上面的3，4行 print 语句就构成一个代码块，行这个代码块。

缩进请严格按照Python的习惯写法：**4个空格，不要使用Tab，更不要混合Tab和空格**，否则很容易造成

**elif** 意思就是 **else if**。这样一来，我们就写出了结构非常清晰的一系列条件判断。

**特别注意：**这一系列条件判断会从上到下依次判断，如果某个判断为 True，执行完对应的代码块，后面

Python的 **for** 循环就可以依次把list或tuple的每个元素迭代出来：

```
L = ['Adam', 'Lisa', 'Bart']
for name in L:
    print name
```

**注意：**name 这个变量是在 for 循环中定义的，意思是，依次取出list中的每一个元素，并把元素赋值给

## Python之 break退出循环

用 for 循环或者 while 循环时，如果要在循环体内直接退出循环，可以使用 **break** 语句。

比如计算1至100的整数和，我们用while来实现：

```
sum = 0
x = 1
while True:
    sum = sum + x
    x = x + 1
    if x > 100:
        break
print sum
```

乍一看， while True 就是一个死循环，但是在循环体内，我们还判断了 **x > 100** 条件成立时，用break

现在老师只想统计及格分数的平均分，就要把 **x < 60** 的分数剔除掉，这时，利用 continue，可以做到次循环：

```
for x in L:
    _____
```

'b'，指向一个list，就不能改成指向其他对象，但指向

块（但不包括第5行的print）。如果 if 语句判断为 True，就会执

就因为缩进引起的语法错误。

的条件判断就直接忽略，不再执行了。

name，然后执行for循环体（就是缩进的代码块）。

语句退出循环，这样也可以实现循环的结束。

当  $x < 60$  的时候，不继续执行循环体的后续代码，直接进入下一

```
if x < 60:
    continue
sum = sum + x
n = n + 1
```

## ★ Dict和Set

给定一个名字，就可以直接查到分数。

Python的 **dict** 就是专门干这件事的。用 **dict** 表示“名字”-“成绩”的查找表如下：

```
d = {
    'Adam': 95,
    'Lisa': 85,
    'Bart': 59
}
```

我们把**名字**称为**key**，对应的**成绩**称为**value**，dict就是通过 **key** 来查找 **value**。

花括号 **{ }** 表示这是一个dict，然后按照 **key: value**，写出来即可。最后一个 key: value 的逗号可以省略

由于dict也是集合，**len()** 函数可以计算任意集合的大小：

```
>>> len(d)
3
```

**注意：**一个 key-value 算一个，因此，dict大小为3。

可以简单地使用 **d[key]** 的形式来查找对应的 value，这和 list 很像，不同之处是，**list** 必须使用索引返

```
>>> print d['Adam']
95
```

要避免 **KeyError** 发生，有两个办法：

一是先判断一下 **key** 是否存在，用 **in** 操作符：

```
if 'Paul' in d:
    print d['Paul']
```

如果 'Paul' 不存在，if语句判断为False，自然不会执行 **print d['Paul']**，从而避免了错误。

二是使用dict本身提供的一个 **get** 方法，在Key不存在的时候，返回None：

```
>>> print d.get('Bart')
59
>>> print d.get('Paul')
None
```

## Python中dict的特点

dict的第一个特点是**查找速度快**，无论dict有10个元素还是10万个元素，查找速度都一样。而list的查找

不过dict的查找速度快不是没有代价的，**dict的缺点是占用内存大，还会浪费很多内容**，list正好相反，

由于dict是按key来查找，所以在dict中，key不能重复

。

回对应的元素，而dict使用key：

速度随着元素增加而逐渐下降。

占用内存小，但是查找速度慢。



由于dict是按 key 查找，所以，在 dict 中，key 不能重复。

dict 的第二个特点就是存储的 key-value 序对是**没有顺序**的！这和 list 不一样：

```
d = {
    'Adam': 95,
    'Lisa': 85,
    'Bart': 59
}
```

当我们试图打印这个 dict 时：

```
>>> print d
{'Lisa': 85, 'Adam': 95, 'Bart': 59}
```

打印的顺序不一定是我们创建时的顺序，而且，不同的机器打印的顺序都可能不同，这说明 dict 内部是**无序**的。

dict 的第三个特点是作为 key 的元素必须**不可变**，Python 的基本类型如字符串、整数、浮点数都是不可变的。可以试试用 list 作为 key 时会报什么样的错误。

不可变这个限制仅作用于 key，value 是否可变无所谓：

```
{
    '123': [1, 2, 3], # key 是 str, value 是 list
    123: '123', # key 是 int, value 是 str
    ('a', 'b'): True # key 是 tuple, 并且 tuple 的每个元素都是不可变对象, value 是 bool
}
```

最常用的 key 还是字符串，因为用起来最方便。

dict 是可变的，也就是说，我们可以随时往 dict 中**添加新的 key-value**。比如已有 dict：

```
d = {
    'Adam': 95,
    'Lisa': 85,
    'Bart': 59
}
```

要把新同学 'Paul' 的成绩 72 加进去，用赋值语句：

```
>>> d['Paul'] = 72
```

再看看 dict 的内容：

```
>>> print d
{'Lisa': 85, 'Paul': 72, 'Adam': 95, 'Bart': 59}
```

如果 key 已经存在，则赋值会用新的 value 替换掉原来的 value：

```
>>> d['Bart'] = 60
>>> print d
{'Lisa': 85, 'Paul': 72, 'Adam': 95, 'Bart': 60}
```

## Python 之 遍历 dict

由于 dict 也是一个集合，所以，遍历 dict 和遍历 list 类似，都可以通过 for 循环实现。

直接使用 for 循环可以遍历 dict 的 key：

**无序**的，不能用dict存储有序的集合。

可变的，都可以作为 key。但是list是可变的，就不能作为 key。

value是 boolean

且按使用for循环可以遍历 dict 的 key。

```
>>> d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59 }
>>> for key in d:
...     print key
...
Lisa
Adam
Bart
```

由于通过 key 可以获取对应的 value，因此，在循环体内，可以获取到value的值。Print(key,':',d[key])

## Python中什么是set

dict的作用是建立一组 key 和一组 value 的映射关系，dict的key是不能重复的。

有的时候，我们只想要 dict 的 key，不关心 key 对应的 value，目的就是保证这个集合的元素不会重复。

set 持有一系列元素，这一点和 list 很像，但是set的元素没有重复，而且是无序的，这点和 dict 的 key 类似。

创建 set 的方式是调用 set() 并传入一个 list，list的元素将作为set的元素：

```
>>> s = set(['A', 'B', 'C'])
```

可以查看 set 的内容：

```
>>> print s
set(['A', 'C', 'B'])
```

请注意，上述打印的形式类似 list，但它不是 list，仔细看还可以发现，打印的顺序和原始 list 的顺序不同。

因为set不能包含重复的元素，所以，当我们传入包含重复元素的 list 会怎么样呢？

```
>>> s = set(['A', 'B', 'C', 'C'])
>>> print s
set(['A', 'C', 'B'])
>>> len(s)
3
```

结果显示，set会自动去掉重复的元素，原来的list有4个元素，但set只有3个元素。

## Python之 访问set

由于set存储的是无序集合，所以我们没法通过索引来访问。

访问 set中的某个元素实际上就是判断一个元素是否在set中。

例如，存储了班里同学名字的set：

```
>>> s = set(['Adam', 'Lisa', 'Bart', 'Paul'])
```

我们可以用 in 操作符判断：

Bart是该班的同学吗？

```
>>> 'Bart' in s
True
```

Bill是该班的同学吗？

复，这时，set就派上用场了。

ey很像。

有可能是不同的，因为set内部存储的元素是**无序**的。

```
>>> 'Bill' in s
False
```

bart是该班的同学吗？

```
>>> 'bart' in s
False
```

看来大小写很重要，'Bart' 和 'bart'被认为是两个不同的元素。

## Python之 set的特点

set的内部结构和dict很像，唯一区别是不存储value，因此，判断一个元素是否在set中速度很快。

set存储的元素和dict的key类似，必须是不变对象，因此，任何可变对象是不能放入set中的。

最后，set存储的元素也是没有顺序的。

set的这些特点，可以应用在哪些地方呢？

星期一到星期日可以用字符串'MON', 'TUE', ... 'SUN'表示。

假设我们让用户输入星期一至星期日的某天，如何判断用户的输入是否是一个有效的星期呢？

可以用 if 语句判断，但这样做非常繁琐：

```
x = '???' # 用户输入的字符串
if x != 'MON' and x != 'TUE' and x != 'WED' ... and x != 'SUN':
    print 'input error'
else:
    print 'input ok'
```

注意：if 语句中的...表示没有列出的其它星期名称，测试时，请输入完整。

如果事先创建好一个set，包含'MON' ~ 'SUN'：

```
weekdays = set(['MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT', 'SUN'])
```

再判断输入是否有效，只需要判断该字符串是否在set中：

```
x = '???' # 用户输入的字符串
if x in weekdays:
    print 'input ok'
else:
    print 'input error'
```

这样一来，代码就简单多了。

## Python之 遍历set

由于 set 也是一个集合，所以，遍历 set 和遍历 list 类似，都可以通过 for 循环实现。

直接使用 for 循环可以遍历 set 的元素：

```
>>> s = set(['Adam', 'Lisa', 'Bart'])
>>> for name in s:
...     print name
... 
```



Lisa  
Adam  
Bart

注意: 观察 for 循环在遍历set时, 元素的顺序和list的顺序很可能是不同的, 而且不同的机器上运行的结

## Python之 更新set

由于set存储的是一组不重复的无序元素, 因此, 更新set主要做两件事:

一是把新的元素添加到set中, 二是把已有元素从set中删除。

添加元素时, 用set的add()方法:

```
>>> s = set([1, 2, 3])
>>> s.add(4)
>>> print s
set([1, 2, 3, 4])
```

如果添加的元素已经存在于set中, add()不会报错, 但是不会加进去了:

```
>>> s = set([1, 2, 3])
>>> s.add(3)
>>> print s
set([1, 2, 3])
```

删除set中的元素时, 用set的remove()方法:

```
>>> s = set([1, 2, 3, 4])
>>> s.remove(4)
>>> print s
set([1, 2, 3])
```

如果删除的元素不存在set中, remove()会报错:

```
>>> s = set([1, 2, 3])
>>> s.remove(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 4
```

所以用add()可以直接添加, 而remove()前需要判断。

## 函数

<http://docs.python.org/2/library/functions.html#abs>

### 任务

sum()函数接受一个list作为参数, 并返回list所有元素之和。请计算  $1*1 + 2*2 + 3*3 + \dots + 100*100$ 。

🔴 不会了怎么办

首先, 可以用 while 循环构造出 list。

参考代码:

```
L = []
```

结果也可能不同。



```

x = 1
while x <= 100:
    L.append(x * x)
    x = x + 1
print sum(L)

```

## Python之编写函数

在Python中，定义一个函数要使用 **def** 语句，依次写出**函数名**、**括号**、括号中的**参数**和**冒号**；然后，在函数体内部写要执行的代码。我们以自定义一个求绝对值的 my\_abs 函数为例：

```

def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x

```

**请注意**，函数体内部的语句在执行时，一旦执行到return时，函数就执行完毕，并将结果返回。因此，函数体内部不用写return语句，如果函数没有return语句，函数执行完毕后也会返回结果，只是结果为 None。

**return None可以简写为return。**

但其实这只是一种假象，Python函数返回的仍然是单一值：

```

>>> r = move(100, 100, 60, math.pi / 6)
>>> print r
(151.96152422706632, 70.0)

```

用print打印返回结果，原来返回值是一个tuple！

但是，在语法上，返回一个tuple可以省略括号，而多个变量可以同时接收一个tuple，按位置赋给对应的变量，这样写起来更方便。

## 任务

一元二次方程的定义是： $ax^2 + bx + c = 0$

请编写一个函数，返回一元二次方程的两个解。

**注意：**Python的math包提供了sqrt()函数用于计算平方根。

**?**不会了怎么办

请参考求根公式： $x = (-b \pm \sqrt{b^2 - 4ac}) / 2a$

**参考代码：**

```

import math
def quadratic_equation(a, b, c):
    t = math.sqrt(b * b - 4 * a * c)
    return (-b + t) / (2 * a), (-b - t) / (2 * a)
print quadratic_equation(2, 3, 0)
print quadratic_equation(1, -6, 5)

```

在缩进块中编写函数体，函数的返回值用 `return` 语句返回。

函数内部通过条件判断和循环可以实现非常复杂的逻辑。

的值，所以，Python的函数返回多值其实就是返回一个tuple，但

## Python之递归函数

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。

举个例子，我们来计算阶乘  $n! = 1 * 2 * 3 * \dots * n$ ，用函数 `fact(n)`表示，可以看出：

$fact(n) = n! = 1 * 2 * 3 * \dots * (n-1) * n = (n-1)! * n = fact(n-1) * n$

所以，`fact(n)`可以表示为  $n * fact(n-1)$ ，只有 $n=1$ 时需要特殊处理。

于是，`fact(n)`用递归的方式写出来就是：

```
def fact(n):
    if n==1:
        return 1
    return n * fact(n - 1)
```

上面就是一个递归函数。可以试试：

```
>>> fact(1)
```

```
1
```

```
>>> fact(5)
```

```
120
```

```
>>> fact(100)
```

```
933262154439441526816992388562667004907159682643816214685929638952175
22375825118521091686400000000000000000000000000000000L
```

如果我们计算`fact(5)`，可以根据函数定义看到计算过程如下：

```
====> 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
```

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每次会减一层栈帧。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。可以试试计算

## Python之定义默认参数

定义函数的时候，还可以有默认参数。

例如Python自带的 `int()` 函数，其实就有两个参数，我们既可以传一个参数，又可以传两个参数：

```
>>> int('123')
```

```
123
```

```
>>> int('123', 8)
```

```
15
```

n

99993229915608941463976156518286253697920827

辑不如递归清晰。

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  
fact(10000)。

int()函数的第二个参数是转换进制，如果不传，默认是十进制 (base=10)，如果传了，就用传入的参数。

可见，**函数的默认参数的作用是简化调用**，你只需要把必须的参数传进去。但是在需要的时候，又可以

我们来定义一个计算  $x$  的  $N$  次方的函数：

```
def power(x, n):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

假设计算平方的次数最多，我们就可以把  $n$  的默认值设定为 2：

```
def power(x, n=2):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

这样一来，计算平方就不需要传入两个参数了：

```
>>> power(5)
25
```

由于函数的参数按从左到右的顺序匹配，所以**默认参数只能定义在必需参数的后面**：

```
# OK:
def fn1(a, b=1, c=2):
    pass
# Error:
def fn2(a=1, b):
    pass
```

## 任务

请定义一个 greet() 函数，它包含一个默认参数，如果没有传入，打印 'Hello, world.'，如果传入，打印

**?**不会了怎么办

默认参数的默认值可以设定为 'world'

参考代码：

```
def greet(name='world'):
    print 'Hello, ' + name + ' .'
greet()
greet('Bart')
```

## Python之定义可变参数

如果想让一个函数能接受任意个参数，我们就可以定义一个可变参数：

```
def fn(*args):
```

传入额外的参数来覆盖默认参数值。

] 'Hello, xxx.'

```
print args
```

可变参数的名字前面有个 \* 号，我们可以传入0个、1个或多个参数给可变参数：

```
>>> fn()
()
>>> fn('a')
('a',)
>>> fn('a', 'b')
('a', 'b')
>>> fn('a', 'b', 'c')
('a', 'b', 'c')
```

可变参数也不是很神秘，Python解释器会把传入的一组参数组装成一个tuple传递给可变参数，因此，在定义可变参数的目的也是为了简化调用。假设我们要计算任意个数的平均值，就可以定义一个可变参数

```
def average(*args):
    ...
```

这样，在调用的时候，可以这样写：

```
>>> average()
0
>>> average(1, 2)
1.5
>>> average(1, 2, 2, 3, 4)
2.4
```

## 任务

请编写接受可变参数的 average() 函数。

? 不会了怎么办

可变参数 args 是一个tuple，当0个参数传入时，args是一个空tuple。

参考代码：

```
def average(*args):
    sum = 0.0
    if len(args) == 0:
        return sum
    for x in args:
        sum = sum + x
    return sum / len(args)
print average()
print average(1, 2)
print average(1, 2, 2, 3, 4)
```

## 对list进行切片

取前3个元素，用一行代码就可以完成切片：

```
>>> L[0:3]
['Adam', 'Lisa', 'Bart']
```

在函数内部，直接把变量 `args` 看成一个 `tuple` 就好了。

：



```
L = ['Adam', 'Lisa', 'Bart', 'Paul']
```

L[0:3]表示，从索引0开始取，直到索引3为止，但不包括索引3。即索引0，1，2，正好是3个元素。

如果第一个索引是0，还可以省略：

```
>>> L[:3]
['Adam', 'Lisa', 'Bart']
```

也可以从索引1开始，取出2个元素出来：

```
>>> L[1:3]
['Lisa', 'Bart']
```

只用一个：，表示从头到尾：

```
>>> L[:]
['Adam', 'Lisa', 'Bart', 'Paul']
```

因此，L[:]实际上复制出了一个新list。

切片操作还可以指定第三个参数：

```
>>> L[::2]
['Adam', 'Bart']
```

第三个参数表示每N个取一个，上面的 L[::2] 会每两个元素取出一个来，也就是隔一个取一个。

把list换成tuple，切片操作完全相同，只是切片的结果也变成了tuple。

## 任务

range()函数可以创建一个数列：

```
>>> range(1, 101)
[1, 2, 3, ..., 100]
```

请利用切片，取出：

1. 前10个数；
2. 3的倍数；
3. 不大于50的5的倍数。

**?**不会了怎么办

要取出3, 6, 9可以用::3的操作，但是要确定起始索引。

参考代码：

```
L = range(1, 101)
print L[:10]
print L[2::3]
print L[4:50:5]
#python2.7
```

Python3里range本身就是一种数据类型了，是一种迭代器。所以可以直接输出，其实不输出的是range(2,5)。可以用list(range(\*\*))转化成列表。

## 倒序切片

对于list 既然Python支持L[-1]取倒数第一个元素 那么它同样支持倒数切片 试试·

是原样输出的，是处理过的。你可以试试`range(1+1,5)`输

为了使用，通常 Python 又对 list 的取倒数第几个元素，加上了倒序切片的功能，代码如下：

```
>>> L = ['Adam', 'Lisa', 'Bart', 'Paul']
>>> L[-2:]
['Bart', 'Paul']
>>> L[: -2]
['Adam', 'Lisa']
>>> L[-3: -1]
['Lisa', 'Bart']
>>> L[-4: -1: 2]
['Adam', 'Bart']
```

记住倒数第一个元素的索引是-1。倒序切片包含起始索引，不包含结束索引。

## 对字符串切片

字符串 'xxx' 和 Unicode 字符串 u'xxx' 也可以看成是一种 list，每个元素就是一个字符。因此，字符串也可以切片。

```
>>> 'ABCDEFGH'[:3]
'ABC'
>>> 'ABCDEFGH'[-3:]
'EFG'
>>> 'ABCDEFGH'[: :2]
'ACEG'
```

在很多编程语言中，针对字符串提供了很多各种截取函数，其实目的就是字符串切片。Python 没有专门的函数。

## 什么是迭代

在 Python 中，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，这种遍历我们成为迭代。

在 Python 中，迭代是通过 **for ... in** 来完成的，而很多语言比如 C 或者 Java，迭代 list 是通过下标完成的，

```
for (i=0; i<list.length; i++) {
    n = list[i];
}
```

可以看出，Python 的 for 循环抽象程度要高于 Java 的 for 循环。

因为 Python 的 for 循环不仅可以用在 list 或 tuple 上，还可以作用在其他任何可迭代对象上。

因此，迭代操作就是对于一个集合，无论该集合是有序还是无序，我们用 for 循环总是可以依次取出集合中的每个元素。

**注意：**集合是指包含一组元素的数据结构，我们已经介绍的包括：

1. **有序集合**：list，tuple，str 和 unicode；
2. **无序集合**：set
3. **无序集合并且具有 key-value 对**：dict

而迭代是一个动词，它指的是一种操作，在 Python 中，就是 for 循环。

迭代与按下标访问数组最大的不同是，后者是一种具体的迭代实现方式，而前者只关心迭代结果，根本不在乎如何迭代。

可以用切片操作，只是操作结果仍是字符串：

对字符串的截取函数，只需要切片一个操作就可以完成，非常简

为迭代（Iteration）。

比如Java代码：

合的每一个元素。

不关心迭代内部是如何实现的。

## 索引迭代

Python中，**迭代永远是取出元素本身，而非元素的索引。**

对于有序集合，元素确实是有索引的。有的时候，我们确实想在 for 循环中拿到索引，怎么办？

方法是使用 **enumerate() 函数**：

```
>>> L = ['Adam', 'Lisa', 'Bart', 'Paul']
>>> for index, name in enumerate(L):
...     print index, '-', name
...
0 - Adam
1 - Lisa
2 - Bart
3 - Paul
```

使用 enumerate() 函数，我们可以在for循环中同时绑定索引index和元素name。但是，这不是 enumerate(L)

**['Adam', 'Lisa', 'Bart', 'Paul']**

变成了类似：

**[(0, 'Adam'), (1, 'Lisa'), (2, 'Bart'), (3, 'Paul')]**

因此，迭代的每一个元素实际上是一个tuple：

```
for t in enumerate(L):
    index = t[0]
    name = t[1]
    print index, '-', name
```

如果我们知道每个tuple元素都包含两个元素，for循环又可以进一步简写为：

```
for index, name in enumerate(L):
    print index, '-', name
```

这样不但代码更简单，而且还少了两条赋值语句。

可见，索引迭代也不是真的按索引访问，而是由 enumerate() 函数自动把每个元素变成 (index, element)

## 迭代dict的value

我们已经了解了**dict对象**本身就是可**迭代对象**，用 for 循环直接迭代 dict，可以每次拿到dict的一个key

如果我们希望迭代 dict 对象的value，应该怎么做？

dict 对象有一个 **values() 方法**，这个方法把dict转换成一个包含所有value的list，这样，我们迭代的就

```
d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59 }
print d.values()
# [85, 95, 59]
for v in d.values():
    print v
# 85
# 95
# 59
```

如果仔细阅读Python的文档，还可以发现 dict除了values()方法外，还有一个 itervalues() 方法，用：

rate() 的特殊语法。实际上，enumerate() 函数把：

nt) 这样的tuple，再迭代，就同时获得了索引和元素本身。

。

是 dict的每一个 value：

tenvalues() 方法替代 values() 方法，迭代效果完全一样。

如未仔细阅读Python的文档，还可以发现，dict除了 `values()` 方法外，还有 `itervalues()` 方法，用

```
d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59 }
print d.itervalues()
# <dictionary-valueiterator object at 0x106adbb50>
for v in d.itervalues():
    print v
# 85
# 95
# 59
```

那这两个方法有何不同之处呢？

1. `values()` 方法实际上把一个 dict 转换成了包含 value 的list。
  2. 但是 `itervalues()` 方法不会转换，它会在迭代过程中依次从 dict 中取出 value，所以 `itervalues()` 方法
  3. 打印 `itervalues()` 发现它返回一个 `<dictionary-valueiterator>` 对象，这说明在Python中，**for 循环**可
- 何可迭代对象都可以作用于for循环，而内部如何迭代我们通常并不用关心。

如果一个对象说自己可迭代，那我们就直接用 for 循环去迭代它，可见，迭代是一种抽象的数据操作，

## 迭代dict的key和value

我们了解了如何**迭代** dict 的**key**和**value**，那么，在一个 for 循环中，能否同时迭代 key和value？答案  
首先，我们看看 dict 对象的 **items()** 方法返回的值：

```
>>> d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59 }
>>> print d.items()
[('Lisa', 85), ('Adam', 95), ('Bart', 59)]
```

可以看到，`items()` 方法把dict对象转换成了包含tuple的list，我们对这个list进行迭代，可以同时获得ke

```
>>> for key, value in d.items():
...     print key, ': ', value
...
Lisa : 85
Adam : 95
Bart : 59
```

和 `values()` 有一个 `itervalues()` 类似，`items()` 也有一个对应的 `iteritems()`，`iteritems()` 不把dict转换  
占用额外的内存。

## 生成列表

要生成list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]，我们可以用`range(1, 11)`：

```
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

但如果要生成[1x1, 2x2, 3x3, ..., 10x10]怎么做？方法一是循环：

```
>>> L = []
>>> for x in range(1, 11):
...     L.append(x * x)
```

iteritems() 方法返回 values() 方法，返回双元素 tuple。

法比 values() 方法节省了生成 list 所需的内存。

可作用的迭代对象远不止 list, tuple, str, unicode, dict等，任

它不对迭代对象内部的数据有任何要求。

是肯定的。

key和value：

成list，而是在迭代过程中不断给出 tuple，所以， iteritems() 不



```
...  
>>> L  
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

但是循环太繁琐，而列表生成式则可以用一行语句代替循环生成上面的list：

```
>>> [x * x for x in range(1, 11)]  
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

这种写法就是Python特有的列表生成式。利用列表生成式，可以以非常简洁的代码生成 list。写列表生成式时，把要生成的元素  $x * x$  放到前面，后面跟 for 循环，就可以把list创建出来

## 条件过滤

列表生成式的 for 循环后面还可以加上 if 判断。例如：

```
>>> [x * x for x in range(1, 11)]  
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

如果我们只想要偶数的平方，不改动 range()的情况下，可以加上 if 来筛选：

```
>>> [x * x for x in range(1, 11) if x % 2 == 0]  
[4, 16, 36, 64, 100]
```

有了 if 条件，只有 if 判断为 True 的时候，才把循环的当前元素添加到列表中。

## 多层表达式

for循环可以嵌套，因此，在列表生成式中，也可以用多层 for 循环来生成列表。

对于字符串 'ABC' 和 '123'，可以使用两层循环，生成全排列：

```
>>> [m + n for m in 'ABC' for n in '123']  
['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']
```

翻译成循环代码就像下面这样：

```
L = []  
for m in 'ABC':  
    for n in '123':  
        L.append(m + n)
```

