Python基础

# Python中数据类型

在Python中，能够直接处理的数据类型有以下几种：

一、**整数**

Python可以处理任意大小的整数，当然包括负整数，在Python程序中，整数的表示方法和数学上的写法一模一样，例如：1，100，-8080，0，等等。

计算机由于使用二进制，所以，有时候用十六进制表示整数比较方便，十六进制用0x前缀和0-9，a-f表示，例如：0xff00，0xa5b4c3d2，等等。

二、**浮点数**

浮点数也就是小数，之所以称为浮点数，是因为按照科学记数法表示时，一个浮点数的小数点位置是可变的，比如，1.23x10^9和12.3x10^8是相等的。浮点数可以用数学写法，如1.23，3.14，-9.01，等等。但是对于很大或很小的浮点数，就必须用科学计数法表示，把10用e替代，1.23x10^9就是1.23e9，或者12.3e8，0.000012可以写成1.2e-5，等等。

整数和浮点数在计算机内部存储的方式是不同的，整数运算永远是精确的（除法难道也是精确的？是的！），而浮点数运算则可能会有四舍五入的误差。

三、**字符串**

字符串是以''或""括起来的任意文本，比如'abc'，"xyz"等等。请注意，''或""本身只是一种表示方式，不是字符串的一部分，因此，字符串'abc'只有a，b，c这3个字符。

四、**布尔值**

布尔值和布尔代数的表示完全一致，一个布尔值只有True、False两种值，要么是True，要么是False，在Python中，可以直接用True、False表示布尔值（请注意大小写），也可以通过布尔运算计算出来。

布尔值可以用and、or和not运算。

and运算是与运算，只有所有都为 True，and运算结果才是 True。

or运算是或运算，只要其中有一个为 True，or 运算结果就是 True。

not运算是非运算，它是一个单目运算符，把 True 变成 False，False 变成 True。

五、**空值**

空值是Python里一个特殊的值，用None表示。None不能理解为0，因为0是有意义的，而None是一个特殊的空值。

此外，Python还提供了列表、字典等多种数据类型，还允许创建自定义数据类型，我们后面会继续讲到

打印输出

**print**语句可以向屏幕上输出指定的文字。比如输出'hello, world'，用代码实现如下：

>>> print 'hello, world'

# Python中的注释

Python的注释以 # 开头，后面的文字直到行尾都算注释

# Python的变量

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（\_）的组合，且不能用数字开头，比如：

a = 1

变量a是一个整数。

t\_007 = 'T007'

变量t\_007是一个字符串。

在Python中，等号=是赋值语句，可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量，例如：

a = 123 # a是整数

print a

a = 'imooc' # a变为字符串

print a

这种变量本身类型不固定的语言称之为**动态语言**，与之对应的是静态语言。

静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。例如Java是静态语言，赋值语句如下（// 表示注释）：

int a = 123; // a是整数类型变量

a = "mooc"; // 错误：不能把字符串赋给整型变量

和静态语言相比，动态语言更灵活，就是这个原因。

请不要把赋值语句的等号等同于数学的等号。比如下面的代码：

x = 10

x = x + 2

如果从数学上理解x = x + 2那无论如何是不成立的，在程序中，赋值语句先计算右侧的表达式x + 2，得到结果12，再赋给变量x。由于x之前的值是10，重新赋值后，x的值变成12。

最后，理解变量在计算机内存中的表示也非常重要。当我们写：a = 'ABC'时，Python解释器干了两件事情：

1. 在内存中创建了一个'ABC'的字符串；

2. 在内存中创建了一个名为a的变量，并把它指向'ABC'。

也可以把一个变量a赋值给另一个变量b，这个操作实际上是把变量b指向变量a所指向的数据，例如下面的代码：

a = 'ABC'

b = a

a = 'XYZ'

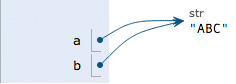
print b

最后一行打印出变量b的内容到底是'ABC'呢还是'XYZ'？如果从数学意义上理解，就会错误地得出b和a相同，也应该是'XYZ'，但实际上b的值是'ABC'，让我们一行一行地执行代码，就可以看到到底发生了什么事：

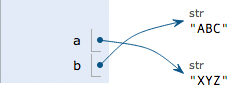
执行a = 'ABC'，解释器创建了字符串  'ABC'和变量 a，并把a指向 'ABC'：

[http://img.mukewang.com/540581030001c11202360058.jpg](http://img.mukewang.com/540581030001c11202360058.jpg)

执行b = a，解释器创建了变量 b，并把b指向 a 指向的字符串'ABC'：

[](http://img.mukewang.com/53fc5e880001399902360084.jpg)

执行a = 'XYZ'，解释器创建了字符串'XYZ'，并把a的指向改为'XYZ'，但b并没有更改：

[](http://img.mukewang.com/53fc5e9f0001b98d02360090.jpg)

所以，最后打印变量b的结果自然是'ABC'了。

# 数据类型

## Python中定义字符串。

字符串可以用''或者""括起来表示。

如果字符串本身包含'怎么办？比如我们要表示字符串 I'm OK ，这时，可以用" "括起来表示：

"I'm OK"

类似的，如果字符串包含"，我们就可以用' '括起来表示：

'Learn "Python" is useful'

如果字符串既包含'又包含"怎么办？

这个时候，就需要对字符串的某些特殊字符进行“转义”，Python字符串用\进行转义。

要表示字符串 Bob said "I'm OK".  
由于 ' 和 " 会引起歧义，因此，我们在它前面插入一个\表示这是一个普通字符，不代表字符串的起始，因此，这个字符串又可以表示为:

'Bob said \"I\'m OK\".'

**注意：**转义字符 \ 不计入字符串的内容中。

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

## Python中raw字符串与多行字符串

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，我们可以在字符串前面加个前缀 r ，表示这是一个 raw 字符串，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：

r'''Python is created by "Guido".

It is free and easy to learn.

Let's start learn Python in imooc!'''

## Python中Unicode字符串

字符串还有一个编码问题。

因为计算机只能处理数字，如果要处理文本，就必须先把文本转换为数字才能处理。最早的计算机在设计时采用8个比特（bit）作为一个字节（byte），所以，一个字节能表示的最大的整数就是255（二进制11111111=十进制255），0 - 255被用来表示大小写英文字母、数字和一些符号，这个编码表被称为ASCII编码，比如大写字母 A 的编码是65，小写字母 z 的编码是122。

如果要表示中文，显然一个字节是不够的，至少需要两个字节，而且还不能和ASCII编码冲突，所以，中国制定了GB2312编码，用来把中文编进去。

类似的，日文和韩文等其他语言也有这个问题。为了统一所有文字的编码，Unicode应运而生。Unicode把所有语言都统一到一套编码里，这样就不会再有乱码问题了。

Unicode通常用两个字节表示一个字符，原有的英文编码从单字节变成双字节，只需要把高字节全部填为0就可以。

因为Python的诞生比Unicode标准发布的时间还要早，所以最早的Python只支持ASCII编码，普通的字符串'ABC'在Python内部都是ASCII编码的。

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。

Unicode字符串除了多了一个 u 之外，与普通字符串没啥区别，转义字符和多行表示法仍然有效：

**转义：**

u'中文\n日文\n韩文'

**多行：**

u'''第一行

第二行'''

**raw+多行：**

ur'''Python的Unicode字符串支持"中文",

"日文",

"韩文"等多种语言'''

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。可以在第一行添加注释

# -\*- coding: utf-8 -\*-

目的是告诉Python解释器，用UTF-8编码读取源代码。然后用Notepad++ 另存为... 并选择UTF-8格式保存。

## Python中整数和浮点数的运算

Python支持对整数和浮点数直接进行四则混合运算，运算规则和数学上的四则运算规则完全一致。

基本的运算：

1 + 2 + 3 # ==> 6

4 \* 5 - 6 # ==> 14

7.5 / 8 + 2.1 # ==> 3.0375

使用括号可以提升优先级，这和数学运算完全一致，注意只能使用小括号，但是括号可以嵌套很多层：

(1 + 2) \* 3 # ==> 9

(2.2 + 3.3) / (1.5 \* (9 - 0.3)) # ==> 0.42145593869731807

和数学运算不同的地方是，Python的整数运算结果仍然是整数，浮点数运算结果仍然是浮点数：

1 + 2 # ==> 整数 3

1.0 + 2.0 # ==> 浮点数 3.0

但是整数和浮点数混合运算的结果就变成浮点数了：

1 + 2.0 # ==> 浮点数 3.0

为什么要区分整数运算和浮点数运算呢？这是因为整数运算的结果永远是精确的，而浮点数运算的结果不一定精确，因为计算机内存再大，也无法精确表示出无限循环小数，比如 0.1 换成二进制表示就是无限循环小数。

那整数的除法运算遇到除不尽的时候，结果难道不是浮点数吗？我们来试一下：

11 / 4 # ==> 2

令很多初学者惊讶的是，Python的整数除法，即使除不尽，结果仍然是整数，余数直接被扔掉。不过，Python提供了一个求余的运算 % 可以计算余数：

11 % 4 # ==> 3

如果我们要计算 11 / 4 的精确结果，按照“整数和浮点数混合运算的结果是浮点数”的法则，把两个数中的一个变成浮点数再运算就没问题了：

11.0 / 4 # ==> 2.75

## Python中布尔类型

我们已经了解了Python支持布尔类型的数据，布尔类型只有True和False两种值，但是布尔类型有以下几种运算：

**与运算**：只有两个布尔值都为 True 时，计算结果才为 True。

True and True # ==> True

True and False # ==> False

False and True # ==> False

False and False # ==> False

**或运算**：只要有一个布尔值为 True，计算结果就是 True。

True or True # ==> True

True or False # ==> True

False or True # ==> True

False or False # ==> False

**非运算**：把True变为False，或者把False变为True：

not True # ==> False

not False # ==> True

布尔运算在计算机中用来做条件判断，根据计算结果为True或者False，计算机可以自动执行不同的后续代码。

在Python中，布尔类型还可以与其他数据类型做 and、or和not运算，请看下面的代码：

a = True

print a and 'a=T' or 'a=F'

计算结果不是布尔类型，而是字符串 'a=T'，这是为什么呢？

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，所以：

True and 'a=T' 计算结果是 'a=T'

继续计算 'a=T' or 'a=F' 计算结果还是 'a=T'

**要解释上述结果，又涉及到 and 和 or 运算的一条重要法则：短路计算。**

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；如果 a 是 True，则整个计算结果必定取决与 b，因此返回 b。

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；如果 a 是 False，则整个计算结果必定取决于 b，因此返回 b。

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。

# Python—List

## Python创建list

Python内置的一种数据类型是列表：list。list是一种有序的集合，可以随时添加和删除其中的元素。

比如，列出班里所有同学的名字，就可以用一个list表示：

>>> ['Michael', 'Bob', 'Tracy']

list是数学意义上的有序集合，也就是说，list中的元素是按照顺序排列的。

## Python按照索引访问list

由于list是一个有序集合，所以，我们可以用一个list按分数从高到低表示出班里的3个同学：

>>> L = ['Adam', 'Lisa', 'Bart']

那我们如何从list中获取指定第 N 名的同学呢

**需要特别注意的是**，索引从 0 开始，也就是说，第一个元素的索引是0，第二个元素的索引是1，以此类推。

因此，要打印第一名同学的名字，用 L[0]:

>>> print L[0]

Adam

要打印第二名同学的名字，用 L[1]:

>>> print L[1]

Lisa

要打印第四名同学的名字，用 L[3]:

>>> print L[3]

4 (most recent call last):

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

IndexError: list index out of range

报错了！IndexError意思就是索引超出了范围，因为上面的list只有3个元素，有效的索引是 0，1，2。

所以，使用索引时，**千万注意不要越界**。

## Python之倒序访问list

List的倒序检索：最后一个为L[-1]，倒数第二个为 L[-2] …..

注意倒序检索也不能越界！！

## Python之添加新元素

**append()； 在集合最后边添加新的元素；**

**L.insert(0, 'Paul')； 把元素插入到定义的位置。**

现在，班里有3名同学：

>>> L = ['Adam', 'Lisa', 'Bart']

今天，班里转来一名新同学 Paul，如何把新同学添加到现有的 list 中呢？

第一个办法是用 list 的 append() 方法，把新同学追加到 list 的末尾：

>>> L = ['Adam', 'Lisa', 'Bart']

>>> L.append('Paul')

>>> print L

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

**append()**总是把新的元素添加到 list 的尾部。

如果 Paul 同学表示自己总是考满分，要求添加到第一的位置，怎么办？

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，第二个参数是待添加的新元素：

>>> L = ['Adam', 'Lisa', 'Bart']

>>> L.insert(0, 'Paul')

>>> print L

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

**L.insert(0, 'Paul')** 的意思是，'Paul'将被添加到索引为 0 的位置上（也就是第一个），而原来索引为 0 的Adam同学，以及后面的所有同学，都自动向后移动一位。

## Python从list删除元素

L.pop() 删除集合的最后一个元素

L.pop(元素下标) 删除指定下标的元素

## Python中替换元素

L[要替换的下标]=要替换的元素（也可用倒序索引）。

## Python之创建tuple

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，就不能修改了。

同样是表示班里同学的名称，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[ ]。

## Python之创建单元素tuple

tuple和list一样，可以包含 0 个、1个和任意多个元素。

包含多个元素的 tuple，前面我们已经创建过了。

包含 0 个元素的 tuple，也就是空tuple，直接用 ()表示：

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，导致我们得到的不是tuple，而是整数 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“,”，这样就避免了歧义：

>>> t = (1,) #单个tuple需在元素后加逗号

>>> 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。

多元素 tuple 加不加这个额外的“,”效果是一样的：

>>> t = (1, 2, 3,)

>>> print t

(1, 2, 3)

## Python之“可变”的tuple

前面我们看到了tuple一旦创建就不能修改。现在，我们来看一个“可变”的tuple：

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

**注意**到 t 有 3 个元素：**'a'，'b'**和一个list：**['A', 'B']**。list作为一个整体是tuple的第3个元素。list对象可以通过 t[2] 拿到：

>>> L = t[2]

然后，我们把list的两个元素改一改：

>>> L[0] = 'X'

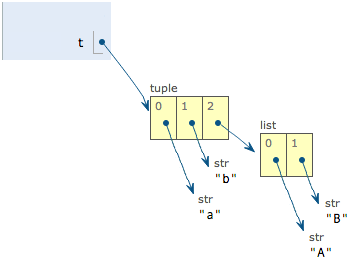
>>> L[1] = 'Y'

再看看tuple的内容：

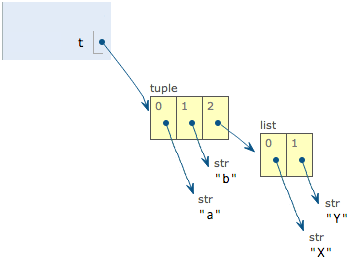
>>> print t

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

tuple包含的3个元素存储分析：

[](http://img.mukewang.com/540538d400010f4603500260.jpg)

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

[](http://img.mukewang.com/540538e9000110c003500260.jpg)

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就必须保证tuple的每一个元素本身也不能变。

# Python—判断、循环

## Python之if语句

If示例，注意缩进

age = 20

if age >= 18:

print 'your age is', age

print 'adult'

else:

print 'age is smaller than 18'

print 'END

'

**注意:**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（但不包括第5行的print）。如果 if 语句判断为 True，就会执行这个代码块。

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，否则很容易造成因为缩进引起的语法错误。

**注意**: if 语句后接表达式，然后用:表示代码块开始。

如果你在Python交互环境下敲代码，还要特别留意缩进，并且退出缩进需要多敲一行回车：

## Python之 if-elif-else

要避免嵌套结构的 if ... else ...，我们可以用 if ... 多个elif ... else ... 的结构，一次写完所有的规则：

if age >= 18:

print 'adult'

elif age >= 6:

print 'teenager'

elif age >= 3:

print 'kid'

else:

print 'baby'

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，后面的条件判断就直接忽略，不再执行了。

## Python之 for循环

list或tuple可以表示一个有序集合。如果我们想依次访问一个list中的每一个元素呢？比如 list：

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 name，然后执行for循环体（就是缩进的代码块）。

## Python之 while循环

和 for 循环不同的另一种循环是 while 循环，while 循环不会迭代 list 或 tuple 的元素，而是根据表达式判断循环是否结束。

比如要从 0 开始打印不大于 N 的整数：

N = 10

x = 0

while x < N:

print x

x = x + 1

while循环每次先判断 x < N，如果为True，则执行循环体的代码块，否则，退出循环。

在循环体内，x = x + 1 会让 x 不断增加，最终因为 x < N 不成立而退出循环。

如果没有这一个语句，**while循环在判断 x < N 时总是为True**，就会无限循环下去，变成死循环，所以要特别留意while循环的退出条件。

## 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

在循环体内，判断了 x > 100 条件成立时，用break语句退出循环，这样也可以实现循环的结束。

## Python之 continue继续循环

在循环过程中，可以用break退出当前循环，还可以用continue跳过后续循环代码，继续下一次循环。

计算平均分：

L = [75, 98, 59, 81, 66, 43, 69, 85]

想要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，直接进入下一次循环：

for x in L:

if x < 60:

continue

sum = sum + x

n = n + 1

## Python之 多重循环

在循环内部，还可以嵌套循环，我们来看一个例子：

for x in ['A', 'B', 'C']:

for y in ['1', '2', '3']:

print x + y

x 每循环一次，y 就会循环 3 次

# Python—dict/set

## Python-dict 基本用法

用法类似于java的map，写法为java—map的json写法

d = {

'Adam': 95,

'Lisa': 85,

'Bart': 59

}

* 可以简单地使用 d[key] 的形式来查找对应的 value
* len(d) #函数返回dict的长度
* 判断 key 是否存在，用 in 操作符：

if 'Paul' in d:

print d['Paul']

* dict供的一个 get 方法，获取value值，在Key不存在的时候，返回None：

>>> print d.get('Bart')

59

>>> print d.get('Paul')

None

* Python更新dict

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

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

## 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的查找速度随着元素增加而逐渐下降。

不过**dict的缺点是占用内存大，还会浪费很多内容**，list正好相反，占用内存小，但是查找速度慢。

* **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：

dict内部是**无序**的，不能用dict存储有序的集合。

* **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，都可以作为 key。但是list是可变的，就不能作为 key。由于dict是按 key 查找，所以，在一个dict中，key不能重复。

## 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：

>>> 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。

## Python--set

* **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很像。**

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'])

* **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。

>>> 'B' in s #此方法严格区分大小写，如果是小写b，则返回false

True

>>> s = set([1, 2, 3])

>>> s.add(4) #set添加元素

>>> print s

set([1, 2, 3, 4])

删除set中的元素时，用set的remove()方法：

>>> s = set([1, 2, 3, 4])

>>> s.remove(4) #删除set中的已有元素，删之前需判断元素是否存在，不存在的话会报错

>>> print s

set([1, 2, 3])

## Python之 set的特点的应用

**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。

常用来判断对象是否包含在set里。

## 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果也可能不同。

# Python之函数

## 单返回参数函数

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，在缩进块中编写函数体，函数的返回值用 return语句返回。

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，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

**return None可以简写为return。**

如果没有return语句，函数执行完毕后也会返回结果，只是结果为 None。

## 多个返回参数函数

比如在游戏中经常需要从一个点移动到另一个点，给出坐标、位移和角度，就可以计算出新的坐标：

**# math**包提供了**sin()**和 **cos()**函数，我们先用import引用它：

import math

def move(x, y, step, angle):

nx = x + step \* math.cos(angle)

ny = y - step \* math.sin(angle)

return nx, ny

这样我们就可以同时获得返回值：

>>> x, y = move(100, 100, 60, math.pi / 6)

>>> print x, y

151.961524227 70.0

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，按位置赋给对应的值，所以，**Python的函数**返回多值其实就是**返回一个tuple**，但写起来更方便。

## 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 \* ... \* n，用函数 fact(n)表示，可以看出：

fact(n) = n! = 1 \* 2 \* 3 \* ... \* (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)

>>> fact(5)

120

>>> fact(100)

93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000L

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。可以试试计算 fact(10000)。

## 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)

83

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，又可以传入额外的参数来覆盖默认参数值。

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

## 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):

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')

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，在函数内部，直接把变量 args 看成一个 tuple 就好了。

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

# Python—切片

## 对list进行切片

取一个list的部分元素是非常常见的操作。比如，一个list如下：

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

取前3个元素，应该怎么做？

笨办法：

>>> [L[0], L[1], L[2]]

['Adam', 'Lisa', 'Bart']

之所以是笨办法是因为扩展一下，取前N个元素就没辙了。

取前N个元素，也就是索引为0-(N-1)的元素，可以用循环：

>>> r = []

>>> n = 3

>>> for i in range(n):

... r.append(L[i])

...

>>> r

['Adam', 'Lisa', 'Bart']

对这种经常取指定索引范围的操作，用循环十分繁琐，因此，Python提供了切片（Slice）操作符，能大大简化这种操作。

对应上面的问题，取前3个元素，用一行代码就可以完成切片：

>>> L[0:3]

['Adam', 'Lisa', 'Bart']

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。

## 倒序切片

对于list，Python支持L[-1]取倒数第一个元素，它同样支持倒数切片

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

>>> L[-2:]

['Bart', 'Paul']

>>> L[:-2]

['Adam', 'Lisa']

>>> L[-3:-1]

['Lisa', 'Bart']

>>> L[-4:-1:2]

['Adam', 'Bart']

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，字符串也可以用切片操作，只是操作结果仍是字符串：

>>> 'ABCDEFG'[:3]

'ABC'

>>> 'ABCDEFG'[-3:]

'EFG'

>>> 'ABCDEFG'[::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，其实目的就是对字符串切片。Python没有针对字符串的截取函数，只需要切片一个操作就可以完成，非常简单。

# Python—迭代

## 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（Iteration）。

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，比如Java代码：

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的每一个元素。

**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，根本不关心迭代内部是如何实现的。

请用for循环迭代数列 1-100 并打印出7的倍数。

用range(1, 101)可以创建数组。

**参考代码:**

for i in range(1, 101):

if i % 7 == 0:

print i

## 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() 的特殊语法。实际上，enumerate() 函数把：

**['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) 这样的tuple，再迭代，就同时获得了索引和元素本身。

## 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，我们迭代的就是 dict的每一个 value：

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 **itervalues()** 方法替代 **values()** 方法，迭代效果完全一样：

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() 方法比 values() 方法节省了生成 list 所需的内存。

3. 打印 itervalues() 发现它返回一个 <dictionary-valueiterator> 对象，这说明在Python中，**for 循环可作用的迭代对象远不止 list，tuple，str，unicode，dict等**，任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，它不对迭代对象内部的数据有任何要求。**

## 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？答案是肯定的。

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key和value：

>>> 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转换成list，而是在迭代过程中不断给出 tuple，所以， iteritems() 不占用额外的内存。

# Python--列表

## 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)

...

>>> 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，十分有用，多写几次，很快就可以熟悉这种语法。

## 复杂表达式

使用**for循环**的迭代不仅可以迭代普通的list，还可以迭代dict。

假设有如下的dict：

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

完全可以通过一个复杂的列表生成式把它变成一个 HTML 表格：

tds = ['<tr><td>%s</td><td>%s</td></tr>' % (name, score) for name, score in d.iteritems()]

print '<table>'

print '<tr><th>Name</th><th>Score</th><tr>'

print '\n'.join(tds)

print '</table>'

**注：**字符串可以通过 % 进行格式化，用指定的参数替代%s。字符串的join()方法可以把一个 list 拼接成一个字符串。

把打印出来的结果保存为一个html文件，就可以在浏览器中看到效果了：

<table border="1">

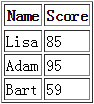
<tr><th>Name</th><th>Score</th><tr>

<tr><td>Lisa</td><td>85</td></tr>

<tr><td>Adam</td><td>95</td></tr>

<tr><td>Bart</td><td>59</td></tr>

</table>

[](http://img.mukewang.com/540fcd2a0001ff4600940104.jpg)

## 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]

## 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)