# Python

## Python入门

简介：Python教程基础分《Python入门》和《Python进阶》两门课程，本教程是Python第一门课程，是Python开发的入门教程，将介绍Python语言的特点和适用范围，Python基本的数据类型，条件判断和循环，函数，以及Python特有的切片和列表生成式。希望本python教程能够让您快速入门并编写简单的Python程序。

### Python变量和数据类型

讲解Python基本的数据类型，包括整数、浮点数、字符串和布尔类型，以及变量的概念和基本的数据运算。

#### Python中数据类型

计算机顾名思义就是可以做数学计算的机器，因此，计算机程序理所当然地可以处理各种数值。但是，计算机能处理的远不止数值，还可以处理文本、图形、音频、视频、网页等各种各样的数据，不同的数据，需要定义不同的数据类型。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

#### Python之print语句

**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'

**注意：**

1.当我们在Python交互式环境下编写代码时，>>>是Python解释器的提示符，不是代码的一部分。

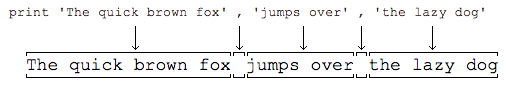
2.当我们在文本编辑器中编写代码时，千万不要自己添加 >>>。

print语句也可以跟上多个字符串，用逗号“,”隔开，就可以连成一串输出：

>>> print 'The quick brown fox', 'jumps over', 'the lazy dog'

The quick brown fox jumps over the lazy dog #运行结果

print会依次打印每个字符串，遇到逗号“,”会输出一个空格，因此，输出的字符串是这样拼起来的：

[](http://img.mukewang.com/54055502000179c205060086.jpg)

print也可以打印整数，或者计算结果：

>>> print 300

300 #运行结果

>>> print 100 + 200

300 #运行结果

因此，我们可以把计算100 + 200的结果打印得更漂亮一点：

>>> print '100 + 200 =', 100 + 200

100 + 200 = 300 #运行结果

**注意:** 对于100 + 200，Python解释器自动计算出结果300，但是，'100 + 200 ='是字符串而非数学公式，Python把它视为字符串，请自行解释上述打印结果。

#### Python的注释

任何时候，我们都可以给程序加上注释。注释是用来说明代码的，给自己或别人看，而程序运行的时候，Python解释器会直接忽略掉注释，所以，有没有注释不影响程序的执行结果，但是影响到别人能不能看懂你的代码。

Python的注释以 # 开头，后面的文字直到行尾都算注释

# 这一行全部都是注释...

print 'hello' # 这也是注释

注释还有一个巧妙的用途，就是一些代码我们不想运行，但又不想删除，就可以用注释暂时屏蔽掉：

# 暂时不想运行下面一行代码:

# print 'hello, python.'

#### Python中什么是变量

在Python中，变量的概念基本上和初中代数的方程变量是一致的。

例如，对于方程式 y=x\*x ，x就是变量。当x=2时，计算结果是4，当x=5时，计算结果是25。

只是在计算机程序中，变量不仅可以是数字，还可以是任意数据类型。

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'

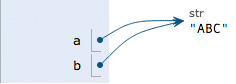
print b

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：

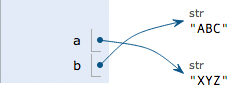
执行a = 'ABC'，解释器创建了字符串  'ABC'和变量 a，并把a指向 'ABC'：

[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。

#### Python中定义字符串

前面我们讲解了什么是字符串。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" in imooc'

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

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。

### 第2章 List和Tuple类型

本章讲解Python最常用的两种集合类型：list和tuple，我们可以用list和tuple来表示有序集合。

#### 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']

['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。

构造list非常简单，按照上面的代码，直接用 [ ] 把list的所有元素都括起来，就是一个list对象。通常，我们会把list赋值给一个变量，这样，就可以通过变量来引用list：

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

>>> classmates # 打印classmates变量的内容

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

由于Python是动态语言，所以list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中包含各种数据：

>>> L = ['Michael', 100, True]

一个元素也没有的list，就是空list：

>>> empty\_list = []

#### 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？方法是通过索引来获取list中的指定元素。

**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

要打印第三名同学的名字，用 L[2]:

>>> print L[2]

Bart

要打印第四名同学的名字，用 L[3]:

>>> print L[3]

Traceback (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

我们还是用一个list按分数从高到低表示出班里的3个同学：

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

这时，老师说，请分数最低的同学站出来。

要写代码完成这个任务，我们可以先数一数这个 list，发现它包含3个元素，因此，最后一个元素的索引是2：

>>> print L[2]

Bart

有没有更简单的方法？

有！

Bart同学是最后一名，俗称倒数第一，所以，我们可以用 -1 这个索引来表示最后一个元素：

>>> print L[-1]

Bart

Bart同学表示躺枪。

类似的，倒数第二用 -2 表示，倒数第三用 -3 表示，倒数第四用 -4 表示：

>>> print L[-2]

Lisa

>>> print L[-3]

Adam

>>> print L[-4]

Traceback (most recent call last):

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

IndexError: list index out of range

L[-4] 报错了，因为倒数第四不存在，一共只有3个元素。

使用倒序索引时，也要注意**不要越界**。

#### Python之添加新元素

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

Paul同学刚来几天又要转走了，那么我们怎么把Paul 从现有的list中删除呢？

如果Paul同学排在最后一个，我们可以用list的pop()方法删除：

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

>>> L.pop()

'Paul'

>>> print L

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

**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() 后，会打印出 'Paul'。

如果Paul同学不是排在最后一个怎么办？比如Paul同学排在第三：

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

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 pop(2)把Paul删掉：

>>> L.pop(2)

'Paul'

>>> print L

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

#### Python中替换元素

假设现在班里仍然是3名同学：

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

现在，Bart同学要转学走了，碰巧来了一个Paul同学，要更新班级成员名单，我们可以先把Bart删掉，再把Paul添加进来。

另一个办法是直接用Paul把Bart给替换掉：

>>> L[2] = 'Paul'

>>> print L

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

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包含的元素个数保持不变。

由于Bart还可以用 -1 做索引，因此，下面的代码也可以完成同样的替换工作：

>>> L[-1] = 'Paul'

#### 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[ ]。

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，新同学没法直接往 tuple 中添加，老同学想退出 tuple 也不行。

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，但是不能赋值成别的元素，不信可以试试：

>>> t[0] = 'Paul'

Traceback (most recent call last):

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

TypeError: 'tuple' object does not support item assignment

#### 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：

>>> t = ()

>>> print t

()

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,)

>>> 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'])

不是说tuple一旦定义后就不可变了吗？怎么现在又变了？

别急，我们先看看定义的时候tuple包含的3个元素：

[](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)

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。

### 第3章 条件判断和循环

讲解Python的条件判断和循环语句，利用条件判断和循环来构造程序的执行逻辑。

#### Python之if语句

计算机之所以能做很多自动化的任务，因为它可以自己做条件判断。

比如，输入用户年龄，根据年龄打印不同的内容，在Python程序中，可以用if语句实现：

age = 20

if age >= 18:

print 'your age is', age

print 'adult'

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：

>>> age = 20

>>> if age >= 18:

... print 'your age is', age

... print 'adult'

...

your age is 20

adult

#### Python之 if-else

当 if 语句判断表达式的结果为 True 时，就会执行 if 包含的代码块：

if age >= 18:

print 'adult'

如果我们想判断年龄在18岁以下时，打印出 'teenager'，怎么办？

方法是再写一个 if:

if age < 18:

print 'teenager'

或者用 not 运算：

if not age >= 18:

print 'teenager'

细心的同学可以发现，这两种条件判断是“非此即彼”的，要么符合条件1，要么符合条件2，因此，完全可以用一个 if ... else ... 语句把它们统一起来：

if age >= 18:

print 'adult'

else:

print 'teenager'

利用 if ... else ... 语句，我们可以根据条件表达式的值为 True 或者 False ，分别执行 if 代码块或者 else 代码块。

**注意:** else 后面有个“:”。

#### Python之 if-elif-else

有的时候，一个 if ... else ... 还不够用。比如，根据年龄的划分：

条件1：18岁或以上：adult

条件2：6岁或以上：teenager

条件3：6岁以下：kid

我们可以用一个 if age >= 18 判断是否符合条件1，如果不符合，再通过一个 if 判断 age >= 6 来判断是否符合条件2，否则，执行条件3：

if age >= 18:

print 'adult'

else:

if age >= 6:

print 'teenager'

else:

print 'kid'

这样写出来，我们就得到了一个两层嵌套的 if ... else ... 语句。这个逻辑没有问题，但是，如果继续增加条件，比如3岁以下是 baby：

if age >= 18:

print 'adult'

else:

if age >= 6:

print 'teenager'

else:

if age >= 3:

print 'kid'

else:

print 'baby'

这种缩进只会越来越多，代码也会越来越难看。

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。

请思考下面的代码：

age = 8

if age >= 6:

print 'teenager'

elif age >= 18:

print 'adult'

else:

print 'kid'

当 age = 8 时，结果正确，但 age = 20 时，为什么没有打印出 adult？

如果要修复，应该如何修复？

#### 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：

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

print L[0]

print L[1]

print L[2]

如果list只包含几个元素，这样写还行，如果list包含1万个元素，我们就不可能写1万行print。

这时，循环就派上用场了。

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）。

这样一来，遍历一个list或tuple就非常容易了。

#### 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

咋一看， 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。

假设我们已经写好了利用for循环计算平均分的代码：

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

sum = 0.0

n = 0

for x in L:

sum = sum + x

n = n + 1

print sum / n

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，这样，我们可以打印出一个全排列：

### 第4章 Dict和Set类型

讲解Python的另外两种重要的数据类型Dict和Set，可以快速按照关键字检索信息。

#### Python之什么是dict

我们已经知道，list 和 tuple 可以用来表示顺序集合，例如，班里同学的名字：

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

或者考试的成绩列表：

[95, 85, 59]

但是，要根据名字找到对应的成绩，用两个 list 表示就不方便。

如果把名字和分数关联起来，组成类似的查找表：

'Adam' ==> 95

'Lisa' ==> 85

'Bart' ==> 59

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。

#### Python之访问dict

我们已经能创建一个dict，用于表示名字和成绩的对应关系：

d = {

'Adam': 95,

'Lisa': 85,

'Bart': 59

}

那么，如何根据名字来查找对应的成绩呢？

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 必须使用索引返回对应的元素，而dict使用key：**

>>> print d['Adam']

95

>>> print d['Paul']

Traceback (most recent call last):

File "index.py", line 11, in <module>

print d['Paul']

KeyError: 'Paul'

**注意:** 通过 key 访问 dict 的value，只要 key 存在，dict就返回对应的value。如果key不存在，会直接报错：KeyError。

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的查找速度随着元素增加而逐渐下降。

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，占用内存小，但是查找速度慢。

由于dict是按 key 查找，所以，在一个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，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。

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 boolean

}

最常用的key还是字符串，因为用起来最方便。

#### 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

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：

>>> 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，这时，set就派上用场了。

**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'])

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 的顺序有可能是不同的，因为set内部存储的元素是**无序**的。

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？

>>> '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果也可能不同。

#### 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。

### 第5章 函数

介绍Python的代码最基本的抽象类型：函数，以及如何定义和调用函数。

#### Python之什么是函数

我们知道圆的面积计算公式为：

S = πr²

当我们知道半径r的值时，就可以根据公式计算出面积。假设我们需要计算3个不同大小的圆的面积：

r1 = 12.34 s1 = 3.14 \* r1 \* r1

r2 = 9.08 s2 = 3.14 \* r2 \* r2

r3 = 73.1 s3 = 3.14 \* r3 \* r3

当代码出现有规律的重复的时候，你就需要当心了，每次写3.14 \* x \* x不仅很麻烦，而且，如果要把3.14改成3.14159265359的时候，得全部替换。

有了函数，我们就不再每次写s = 3.14 \* x \* x，而是写成更有意义的函数调用 s = area\_of\_circle(x)，而函数 area\_of\_circle 本身只需要写一次，就可以多次调用。

**抽象**是数学中非常常见的概念。举个例子：

计算数列的和，比如：**1 + 2 + 3 + ... + 100**，写起来十分不方便，于是数学家发明了求和符号∑，可以把1 + 2 + 3 + ... + 100记作：

100

∑n

n=1

这种抽象记法非常强大，因为我们看到∑就可以理解成求和，而不是还原成低级的加法运算。

而且，这种抽象记法是可扩展的，比如：

100

∑(n²+1)

n=1

还原成加法运算就变成了：

(1 x 1 + 1) + (2 x 2 + 1) + (3 x 3 + 1) + ... + (100 x 100 + 1)

可见，借助抽象，我们才能不关心底层的具体计算过程，而直接在更高的层次上思考问题。

写计算机程序也是一样，函数就是最基本的一种代码抽象的方式。

Python不但能非常灵活地定义函数，而且本身内置了很多有用的函数，可以直接调用。

#### Python之调用函数

Python内置了很多有用的函数，我们可以直接调用。

要调用一个函数，需要知道**函数**的**名称**和**参数**，比如求绝对值的函数 abs，它接收一个参数。

可以直接从Python的官方网站查看文档：

[http://docs.python.org/2/library/functions.html#abs](https://docs.python.org/2/library/functions.html" \l "abs)

也可以在交互式命令行通过 help(abs) 查看abs函数的帮助信息。

调用 **abs**函数：

>>> abs(100)

100

>>> abs(-20)

20

>>> abs(12.34)

12.34

调用函数的时候，如果传入的参数数量不对，会报**TypeError**的错误，并且Python会明确地告诉你：abs()有且仅有1个参数，但给出了两个：

>>> abs(1, 2)

Traceback (most recent call last):

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

TypeError: abs() takes exactly one argument (2 given)

如果传入的参数数量是对的，但参数类型不能被函数所接受，也会报**TypeError**的错误，并且给出错误信息：str是错误的参数类型：

>>> abs('a')

Traceback (most recent call last):

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

TypeError: bad operand type for abs(): 'str'

而比较函数 cmp(x, y) 就需要两个参数，如果 **x<y**，返回 **-1**，如果 **x==y**，返回 **0**，如果 **x>y**，返回 **1**：

>>> cmp(1, 2)

-1

>>> cmp(2, 1)

1

>>> cmp(3, 3)

0

Python内置的常用函数还包括数据类型转换函数，比如   int()函数可以把其他数据类型转换为整数：

>>> int('123')

123

>>> int(12.34)

12

str()函数把其他类型转换成 str：

>>> str(123)

'123'

>>> str(1.23)

'1.23'

#### 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。

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。

return None可以简写为return。

#### Python函数之返回多值

函数可以返回多个值吗？答案是肯定的。

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)

上面就是一个递归函数。可以试试：

>>> fact(1)

1

>>> fact(5)

120

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，但循环的逻辑不如递归清晰。

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

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

2.4

### 第6章 切片

介绍Python程序特有的一种“切片”操作，可以以极其简洁的方式快速对列表进行操作。

#### 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，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：

>>> 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。

### 第7章 迭代

介绍Python程序中“迭代”的概念，以及各种迭代方式。

#### 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。

#### 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

#### 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。**

的tuple，再迭代，就同时获得了索引和元素本身。

#### 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。

### 第8章 列表生成式

介绍Python特有的列表生成式，利用列表生成式，可以通过某种规则快速创建一个列表。

#### 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">

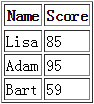
<tr><th>Name</th><th>Score</th><tr>

<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]

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)

## python进阶

简介：Python基础分《Python入门》和《Python进阶》两门课程，《Python进阶》是第二门课程，学习该课程前，请先学习《Python入门》,效果会更好。《Python进阶》课程详细介绍Python强大的函数式编程和面向对象编程，掌握Python高级程序设计的方法。

### 第1章 函数式编程

本章讲解Python函数式编程概念，高阶函数的概念和实际用法，以及装饰器函数的原理和实现方式。

#### python把函数作为参数

高阶函数：

def add(x, y, f):

return f(x) + f(y)

如果传入abs作为参数f的值：

add(-5, 9, abs)

根据函数的定义，函数执行的代码实际上是：

abs(-5) + abs(9)

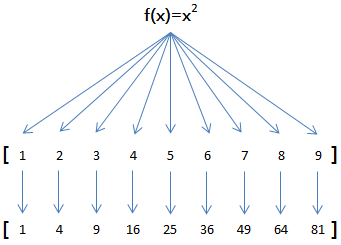
由于参数 x, y 和 f 都可以任意传入，如果 f 传入其他函数，就可以得到不同的返回值。

#### python中map()函数

**map()**是 Python 内置的高阶函数，它接收一个**函数 f** 和一个 **list**，并通过把函数 f 依次作用在 list 的每个元素上，得到一个新的 list 并返回。

例如，对于list [1, 2, 3, 4, 5, 6, 7, 8, 9]

如果希望把list的每个元素都作平方，就可以用map()函数：

[](http://img.mukewang.com/54c8a7e40001327303410245.png)

因此，我们只需要传入函数f(x)=x\*x，就可以利用map()函数完成这个计算：

def f(x):

return x\*x

print map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])

**输出结果：**

[1, 4, 9, 10, 25, 36, 49, 64, 81]

**注意：**map()函数不改变原有的 list，而是返回一个新的 list。

利用map()函数，可以把一个 list 转换为另一个 list，只需要传入转换函数。

由于list包含的元素可以是任何类型，因此，map() 不仅仅可以处理只包含数值的 list，事实上它可以处理包含任意类型的 list，只要传入的函数f可以处理这种数据类型。

#### python中reduce()函数

**reduce()**函数也是Python内置的一个高阶函数。reduce()函数接收的参数和 map()类似，**一个函数 f，一个list**，但行为和 map()不同，reduce()传入的函数 f 必须接收两个参数，reduce()对list的每个元素反复调用函数f，并返回最终结果值。

例如，编写一个f函数，接收x和y，返回x和y的和：

def f(x, y):

return x + y

调用 **reduce(f, [1, 3, 5, 7, 9])**时，reduce函数将做如下计算：

先计算头两个元素：f(1, 3)，结果为4；

再把结果和第3个元素计算：f(4, 5)，结果为9；

再把结果和第4个元素计算：f(9, 7)，结果为16；

再把结果和第5个元素计算：f(16, 9)，结果为25；

由于没有更多的元素了，计算结束，返回结果25。

上述计算实际上是对 list 的所有元素求和。虽然Python内置了求和函数sum()，但是，利用reduce()求和也很简单。

**reduce()还可以接收第3个可选参数，作为计算的初始值。**如果把初始值设为100，计算：

reduce(f, [1, 3, 5, 7, 9], 100)

结果将变为125，因为第一轮计算是：

计算初始值和第一个元素：**f(100, 1)**，结果为**101**。

#### python中filter()函数

**filter()**函数是 Python 内置的另一个有用的高阶函数，filter()函数接收一个**函数 f**和一个**list**，这个函数 f 的作用是对每个元素进行判断，返回 True或 False，**filter()根据判断结果自动过滤掉不符合条件的元素，返回由符合条件元素组成的新list。**

例如，要从一个list [1, 4, 6, 7, 9, 12, 17]中删除偶数，保留奇数，首先，要编写一个判断奇数的函数：

def is\_odd(x):

return x % 2 == 1

然后，利用filter()过滤掉偶数：

filter(is\_odd, [1, 4, 6, 7, 9, 12, 17])

**结果：**[1, 7, 9, 17]

利用filter()，可以完成很多有用的功能，例如，删除 None 或者空字符串：

def is\_not\_empty(s):

return s and len(s.strip()) > 0

filter(is\_not\_empty, ['test', None, '', 'str', ' ', 'END'])

**结果：**['test', 'str', 'END']

**注意:** s.strip(rm) 删除 s 字符串中开头、结尾处的 rm 序列的字符。

当rm为空时，默认删除空白符（包括'\n', '\r', '\t', ' ')，如下：

a = ' 123'

a.strip()

**结果：** '123'

a='\t\t123\r\n'

a.strip()

**结果：**'123'

#### python中自定义排序函数

Python内置的 **sorted()**函数可对list进行排序：

>>>sorted([36, 5, 12, 9, 21])  
[5, 9, 12, 21, 36]

但 **sorted()**也是一个高阶函数，它可以接收一个比较函数来实现自定义排序，比较函数的定义是，传入两个待比较的元素 x, y，**如果 x 应该排在 y 的前面，返回 -1，如果 x 应该排在 y 的后面，返回 1。如果 x 和 y 相等，返回 0。**

因此，如果我们要实现倒序排序，只需要编写一个reversed\_cmp函数：

def reversed\_cmp(x, y):

if x > y:

return -1

if x < y:

return 1

return 0

这样，调用 sorted() 并传入 reversed\_cmp 就可以实现倒序排序：

>>> sorted([36, 5, 12, 9, 21], reversed\_cmp)

[36, 21, 12, 9, 5]

sorted()也可以对字符串进行排序，字符串默认按照ASCII大小来比较：

>>> sorted(['bob', 'about', 'Zoo', 'Credit'])

['Credit', 'Zoo', 'about', 'bob']

'Zoo'排在'about'之前是因为'Z'的ASCII码比'a'小。

#### python中返回函数

Python的函数不但可以返回int、str、list、dict等数据类型，还可以返回函数！

例如，定义一个函数 f()，我们让它返回一个函数 g，可以这样写：

def f():

print 'call f()...'

# 定义函数g:

def g():

print 'call g()...'

# 返回函数g:

return g

仔细观察上面的函数定义，我们在函数 f 内部又定义了一个函数 g。由于函数 g 也是一个对象，函数名 g 就是指向函数 g 的变量，所以，最外层函数 f 可以返回变量 g，也就是函数 g 本身。

调用函数 f，我们会得到 f 返回的一个函数：

>>> x = f() # 调用f()

call f()...

>>> x # 变量x是f()返回的函数：

<function g at 0x1037bf320>

>>> x() # x指向函数，因此可以调用

call g()... # 调用x()就是执行g()函数定义的代码

请注意区分返回函数和返回值：

def myabs():

return abs # 返回函数

def myabs2(x):

return abs(x) # 返回函数调用的结果，返回值是一个数值

返回函数可以把一些计算延迟执行。例如，如果定义一个普通的求和函数：

def calc\_sum(lst):

return sum(lst)

调用calc\_sum()函数时，将立刻计算并得到结果：

>>> calc\_sum([1, 2, 3, 4])

10

但是，如果返回一个函数，就可以“延迟计算”：

def calc\_sum(lst):

def lazy\_sum():

return sum(lst)

return lazy\_sum

# 调用calc\_sum()并没有计算出结果，而是返回函数:

>>> f = calc\_sum([1, 2, 3, 4])

>>> f

<function lazy\_sum at 0x1037bfaa0>

# 对返回的函数进行调用时，才计算出结果:

>>> f()

10

由于可以返回函数，我们在后续代码里就可以决定到底要不要调用该函数。

#### python中闭包

在函数内部定义的函数和外部定义的函数是一样的，只是他们无法被外部访问：

def g():

print 'g()...'

def f():

print 'f()...'

return g

将 **g** 的定义移入函数 **f** 内部，防止其他代码调用 **g**：

def f():

print 'f()...'

def g():

print 'g()...'

return g

但是，考察上一小节定义的 **calc\_sum** 函数：

def calc\_sum(lst):

def lazy\_sum():

return sum(lst)

return lazy\_sum

**注意:** 发现没法把 **lazy\_sum** 移到 **calc\_sum** 的外部，因为它引用了 **calc\_sum** 的参数 **lst**。

像这种内层函数引用了外层函数的变量（参数也算变量），然后返回内层函数的情况，称为**闭包（Closure）**。

**闭包的特点**是返回的函数还引用了外层函数的局部变量，所以，要正确使用闭包，就要确保引用的局部变量在函数返回后不能变。举例如下：

# 希望一次返回3个函数，分别计算1x1,2x2,3x3:

def count():

fs = []

for i in range(1, 4):

def f():

return i\*i

fs.append(f)

return fs

f1, f2, f3 = count()

你可能认为调用f1()，f2()和f3()结果应该是1，4，9，但实际结果全部都是 9（请自己动手验证）。

原因就是当count()函数返回了3个函数时，这3个函数所引用的变量 i 的值已经变成了3。由于f1、f2、f3并没有被调用，所以，此时他们并未计算 i\*i，当 f1 被调用时：

>>> f1()

9 # 因为f1现在才计算i\*i，但现在i的值已经变为3

因此，返回函数不要引用任何循环变量，或者后续会发生变化的变量。

#### python中匿名函数

高阶函数可以接收函数做参数，有些时候，我们不需要显式地定义函数，直接传入匿名函数更方便。

在Python中，对匿名函数提供了有限支持。还是以map()函数为例，计算 f(x)=x2时，除了定义一个f(x)的函数外，还可以直接传入匿名函数：

>>> map(lambda x: x \* x, [1, 2, 3, 4, 5, 6, 7, 8, 9])

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

通过对比可以看出，匿名函数 lambda x: x \* x 实际上就是：

def f(x):

return x \* x

关键字lambda 表示匿名函数，冒号前面的 x 表示函数参数。

匿名函数有个限制，就是**只能有一个表达式**，**不写return**，返回值就是该表达式的结果。

使用匿名函数，可以不必定义函数名，直接创建一个函数对象，很多时候可以简化代码：

>>> sorted([1, 3, 9, 5, 0], lambda x,y: -cmp(x,y))

[9, 5, 3, 1, 0]

返回函数的时候，也可以返回匿名函数：

>>> myabs = lambda x: -x if x < 0 else x

>>> myabs(-1)

1

>>> myabs(1)

1

#### python中编写无参数decorator

Python的 decorator 本质上就是一个高阶函数，它接收一个函数作为参数，然后，返回一个新函数。

使用 decorator 用Python提供的 @ 语法，这样可以避免手动编写 f = decorate(f) 这样的代码。

**考察一个@log的定义：**

def log(f):

def fn(x):

print 'call ' + f.\_\_name\_\_ + '()...'

return f(x)

return fn

对于阶乘函数，@log工作得很好：

@log

def factorial(n):

return reduce(lambda x,y: x\*y, range(1, n+1))

print factorial(10)

**结果：**

call factorial()...

3628800

但是，对于参数不是一个的函数，调用将报错：

@log

def add(x, y):

return x + y

print add(1, 2)

**结果：**

Traceback (most recent call last):

File "test.py", line 15, in <module>

print add(1,2)

TypeError: fn() takes exactly 1 argument (2 given)

因为 add() 函数需要传入两个参数，但是 @log 写死了只含一个参数的返回函数。

要让 @log 自适应任何参数定义的函数，可以利用Python的 \*args 和 \*\*kw，保证任意个数的参数总是能正常调用：

def log(f):

def fn(\*args, \*\*kw):

print 'call ' + f.\_\_name\_\_ + '()...'

return f(\*args, \*\*kw)

return fn

现在，对于任意函数，@log 都能正常工作。

#### python中编写带参数decorator

考察上一节的 @log 装饰器：

def log(f):

def fn(x):

print 'call ' + f.\_\_name\_\_ + '()...'

return f(x)

return fn

发现对于被装饰的函数，log打印的语句是不能变的（除了函数名）。

如果有的函数非常重要，希望打印出'[INFO] call xxx()...'，有的函数不太重要，希望打印出'[DEBUG] call xxx()...'，这时，log函数本身就需要传入'INFO'或'DEBUG'这样的参数，类似这样：

@log('DEBUG')

def my\_func():

pass

把上面的定义翻译成高阶函数的调用，就是：

my\_func = log('DEBUG')(my\_func)

上面的语句看上去还是比较绕，再展开一下：

log\_decorator = log('DEBUG')

my\_func = log\_decorator(my\_func)

上面的语句又相当于：

log\_decorator = log('DEBUG')

@log\_decorator

def my\_func():

pass

所以，带参数的log函数首先返回一个decorator函数，再让这个decorator函数接收my\_func并返回新函数：

def log(prefix):

def log\_decorator(f):

def wrapper(\*args, \*\*kw):

print '[%s] %s()...' % (prefix, f.\_\_name\_\_)

return f(\*args, \*\*kw)

return wrapper

return log\_decorator

@log('DEBUG')

def test():

pass

print test()

**执行结果：**

[DEBUG] test()...

None

对于这种3层嵌套的decorator定义，你可以先把它拆开：

# 标准decorator:

def log\_decorator(f):

def wrapper(\*args, \*\*kw):

print '[%s] %s()...' % (prefix, f.\_\_name\_\_)

return f(\*args, \*\*kw)

return wrapper

return log\_decorator

# 返回decorator:

def log(prefix):

return log\_decorator(f)

拆开以后会发现，调用会失败，因为在3层嵌套的decorator定义中，最内层的wrapper引用了最外层的参数prefix，所以，把一个闭包拆成普通的函数调用会比较困难。不支持闭包的编程语言要实现同样的功能就需要更多的代码。

#### python中完善decorator

@decorator可以动态实现函数功能的增加，但是，经过@decorator“改造”后的函数，和原函数相比，除了功能多一点外，有没有其它不同的地方？

在没有decorator的情况下，打印函数名：

def f1(x):

pass

print f1.\_\_name\_\_

**输出：**f1

有decorator的情况下，再打印函数名：

def log(f):

def wrapper(\*args, \*\*kw):

print 'call...'

return f(\*args, \*\*kw)

return wrapper

@log

def f2(x):

pass

print f2.\_\_name\_\_

**输出：** wrapper

可见，由于decorator返回的新函数函数名已经不是'f2'，而是@log内部定义的'wrapper'。这对于那些依赖函数名的代码就会失效。decorator还改变了函数的\_\_doc\_\_等其它属性。如果要让调用者看不出一个函数经过了@decorator的“改造”，就需要把原函数的一些属性复制到新函数中：

def log(f):

def wrapper(\*args, \*\*kw):

print 'call...'

return f(\*args, \*\*kw)

wrapper.\_\_name\_\_ = f.\_\_name\_\_

wrapper.\_\_doc\_\_ = f.\_\_doc\_\_

return wrapper

这样写decorator很不方便，因为我们也很难把原函数的所有必要属性都一个一个复制到新函数上，所以Python内置的functools可以用来自动化完成这个“复制”的任务：

import functools

def log(f):

@functools.wraps(f)

def wrapper(\*args, \*\*kw):

print 'call...'

return f(\*args, \*\*kw)

return wrapper

最后需要指出，由于我们把原函数签名改成了(\*args, \*\*kw)，因此，无法获得原函数的原始参数信息。即便我们采用固定参数来装饰只有一个参数的函数：

def log(f):

@functools.wraps(f)

def wrapper(x):

print 'call...'

return f(x)

return wrapper

也可能改变原函数的参数名，因为新函数的参数名始终是 'x'，原函数定义的参数名不一定叫 'x'。

#### python中偏函数

当一个函数有很多参数时，调用者就需要提供多个参数。如果减少参数个数，就可以简化调用者的负担。

比如，int()函数可以把字符串转换为整数，当仅传入字符串时，int()函数默认按十进制转换：

>>> int('12345')

12345

但int()函数还提供额外的base参数，默认值为10。如果传入base参数，就可以做 N 进制的转换：

>>> int('12345', base=8)

5349

>>> int('12345', 16)

74565

假设要转换大量的二进制字符串，每次都传入int(x, base=2)非常麻烦，于是，我们想到，可以定义一个int2()的函数，默认把base=2传进去：

def int2(x, base=2):

return int(x, base)

这样，我们转换二进制就非常方便了：

>>> int2('1000000')

64

>>> int2('1010101')

85

functools.partial就是帮助我们创建一个偏函数的，不需要我们自己定义int2()，可以直接使用下面的代码创建一个新的函数int2：

>>> import functools

>>> int2 = functools.partial(int, base=2)

>>> int2('1000000')

64

>>> int2('1010101')

85

所以，functools.partial可以把一个参数多的函数变成一个参数少的新函数，少的参数需要在创建时指定默认值，这样，新函数调用的难度就降低了。

### 第2章 模块

本章讲解如何使用Python的模块，如何编写和导入模块，以及如何安装并使用第三方模块。

#### python之导入模块

要使用一个模块，我们必须首先导入该模块。Python使用import语句导入一个模块。例如，导入系统自带的模块 math：

import math

你可以认为math就是一个指向已导入模块的变量，通过该变量，我们可以访问math模块中所定义的所有公开的函数、变量和类：

>>> math.pow(2, 0.5) # pow是函数

1.4142135623730951

>>> math.pi # pi是变量

3.141592653589793

如果我们只希望导入用到的math模块的某几个函数，而不是所有函数，可以用下面的语句：

from math import pow, sin, log

这样，可以直接引用 pow, sin, log 这3个函数，但math的其他函数没有导入进来：

>>> pow(2, 10)

1024.0

>>> sin(3.14)

0.0015926529164868282

如果遇到名字冲突怎么办？比如math模块有一个log函数，logging模块也有一个log函数，如果同时使用，如何解决名字冲突？

如果使用import导入模块名，由于必须通过模块名引用函数名，因此不存在冲突：

import math, logging

print math.log(10) # 调用的是math的log函数

logging.log(10, 'something') # 调用的是logging的log函数

如果使用 from...import 导入 log 函数，势必引起冲突。这时，可以给函数起个“别名”来避免冲突：

from math import log

from logging import log as logger # logging的log现在变成了logger

print log(10) # 调用的是math的log

logger(10, 'import from logging') # 调用的是logging的log

#### python中动态导入模块

如果导入的模块不存在，Python解释器会报 ImportError 错误：

>>> import something

Traceback (most recent call last):

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

ImportError: No module named something

有的时候，两个不同的模块提供了相同的功能，比如 **StringIO**和 **cStringIO**都提供了StringIO这个功能。

这是因为Python是动态语言，解释执行，因此Python代码运行速度慢。

如果要提高Python代码的运行速度，最简单的方法是把某些关键函数用 C 语言重写，这样就能大大提高执行速度。

同样的功能，StringIO 是纯Python代码编写的，而 cStringIO 部分函数是 C 写的，因此 cStringIO 运行速度更快。

利用ImportError错误，我们经常在Python中动态导入模块：

try:

from cStringIO import StringIO

except ImportError:

from StringIO import StringIO

上述代码先尝试从cStringIO导入，如果失败了（比如cStringIO没有被安装），再尝试从StringIO导入。这样，如果cStringIO模块存在，则我们将获得更快的运行速度，如果cStringIO不存在，则顶多代码运行速度会变慢，但不会影响代码的正常执行。

**try**的作用是捕获错误，并在捕获到指定错误时执行 **except**语句。

#### python之使用\_\_future\_\_

Python的新版本会引入新的功能，但是，实际上这些功能在上一个老版本中就已经存在了。要“试用”某一新的特性，就可以通过导入\_\_future\_\_模块的某些功能来实现。

例如，Python 2.7的整数除法运算结果仍是整数：

>>> 10 / 3

3

但是，Python 3.x已经改进了整数的除法运算，“**/**”除将得到浮点数，“**//**”除才仍是整数：

>>> 10 / 3

3.3333333333333335

>>> 10 // 3

3

要在Python 2.7中引入3.x的除法规则，导入**\_\_future\_\_**的**division**：

>>> from \_\_future\_\_ import division

>>> print 10 / 3

3.3333333333333335

当新版本的一个特性与旧版本不兼容时，该特性将会在旧版本中添加到**\_\_future\_\_**中，以便旧的代码能在旧版本中测试新特性。

### 第3章 面向对象编程基础

本章讲解Python面向对象编程的概念，如何创建类和实例，如何定义类的属性和方法。

#### python之定义类并创建实例

在Python中，类通过 **class**关键字定义。以 **Person** 为例，定义一个**Person类**如下：

class Person(object):

pass

按照 **Python** 的编程习惯，类名以大写字母开头，紧接着是(object)，表示该类是从哪个类继承下来的。类的继承将在后面的章节讲解，现在我们只需要简单地从**object**类继承。

有了Person类的定义，就可以创建出具体的**xiaoming、xiaohong**等实例。创建实例使用 **类名+()**，类似函数调用的形式创建：

xiaoming = Person()

xiaohong = Person()

#### python中创建实例属性

虽然可以通过**Person**类创建出**xiaoming、xiaohong**等实例，但是这些实例看上除了地址不同外，没有什么其他不同。在现实世界中，区分**xiaoming、xiaohong**要依靠他们各自的名字、性别、生日等属性。

如何让每个实例拥有各自不同的属性？由于Python是动态语言，对每一个实例，都可以直接给他们的属性赋值，例如，给**xiaoming**这个实例加上**name、gender**和**birth**属性：

xiaoming = Person()

xiaoming.name = 'Xiao Ming'

xiaoming.gender = 'Male'

xiaoming.birth = '1990-1-1'

给**xiaohong**加上的属性不一定要和**xiaoming**相同：

xiaohong = Person()

xiaohong.name = 'Xiao Hong'

xiaohong.school = 'No. 1 High School'

xiaohong.grade = 2

实例的属性可以像普通变量一样进行操作：

xiaohong.grade = xiaohong.grade + 1

#### python中初始化实例属性

虽然我们可以自由地给一个实例绑定各种属性，但是，现实世界中，一种类型的实例应该拥有相同名字的属性。例如，**Person类**应该在创建的时候就拥有 **name、gender**和 **birth**属性，怎么办？

在定义 Person 类时，可以为Person类添加一个特殊的**\_\_init\_\_()**方法，当创建实例时，**\_\_init\_\_()**方法被自动调用，我们就能在此为每个实例都统一加上以下属性：

class Person(object):

def \_\_init\_\_(self, name, gender, birth):

self.name = name

self.gender = gender

self.birth = birth

**\_\_init\_\_()**方法的第一个参数必须是 **self**（也可以用别的名字，但建议使用习惯用法），后续参数则可以自由指定，和定义函数没有任何区别。

相应地，创建实例时，就必须要提供除 **self**以外的参数：

xiaoming = Person('Xiao Ming', 'Male', '1991-1-1')

xiaohong = Person('Xiao Hong', 'Female', '1992-2-2')

有了**\_\_init\_\_()**方法，每个Person实例在创建时，都会有 **name、gender**和 **birth**这3个属性，并且，被赋予不同的属性值，访问属性使用.操作符：

print xiaoming.name

# 输出 'Xiao Ming'

print xiaohong.birth

# 输出 '1992-2-2'

要特别注意的是，初学者定义**\_\_init\_\_()**方法常常忘记了 self 参数：

>>> class Person(object):

... def \_\_init\_\_(name, gender, birth):

... pass

...

>>> xiaoming = Person('Xiao Ming', 'Male', '1990-1-1')

Traceback (most recent call last):

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

TypeError: \_\_init\_\_() takes exactly 3 arguments (4 given)

这会导致创建失败或运行不正常，因为第一个参数name被Python解释器传入了实例的引用，从而导致整个方法的调用参数位置全部没有对上。

#### python中访问限制

我们可以给一个实例绑定很多属性，如果有些属性不希望被外部访问到怎么办？

Python对属性权限的控制是通过属性名来实现的，如果一个属性由双下划线开头(\_\_)，该属性就无法被外部访问。看例子：

class Person(object):

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

self.name = name

self.\_title = 'Mr'

self.\_\_job = 'Student'

p = Person('Bob')

print p.name

# => Bob

print p.\_title

# => Mr

print p.\_\_job

# => Error

Traceback (most recent call last):

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

AttributeError: 'Person' object has no attribute '\_\_job'

可见，只有以双下划线开头的"**\_\_job**"不能直接被外部访问。

但是，如果一个属性以"\_\_xxx\_\_"的形式定义，那它又可以被外部访问了，以"\_\_xxx\_\_"定义的属性在Python的类中被称为特殊属性，有很多预定义的特殊属性可以使用，通常我们不要把普通属性用"**\_\_xxx\_\_**"定义。

以单下划线开头的属性"\_xxx"虽然也可以被外部访问，但是，按照习惯，他们不应该被外部访问。

#### python中创建类属性

**类是模板，而实例则是根据类创建的对象。**

绑定在一个实例上的属性不会影响其他实例，但是，类本身也是一个对象，如果在类上绑定一个属性，则所有实例都可以访问类的属性，并且，所有实例访问的类属性都是同一个！也就是说，实例属性每个实例各自拥有，互相独立，而类属性有且只有一份。

定义类属性可以直接在 **class**中定义：

class Person(object):

address = 'Earth'

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

self.name = name

因为类属性是直接绑定在类上的，所以，访问类属性不需要创建实例，就可以直接访问：

print Person.address

# => Earth

对一个实例调用类的属性也是可以访问的，所有实例都可以访问到它所属的类的属性：

p1 = Person('Bob')

p2 = Person('Alice')

print p1.address

# => Earth

print p2.address

# => Earth

由于Python是动态语言，类属性也是可以动态添加和修改的：

Person.address = 'China'

print p1.address

# => 'China'

print p2.address

# => 'China'

因为类属性只有一份，所以，当**Person**类的**address**改变时，所有实例访问到的类属性都改变了。

#### python中类属性和实例属性名字冲突怎么办

修改类属性会导致所有实例访问到的类属性全部都受影响，但是，如果在实例变量上修改类属性会发生什么问题呢？

class Person(object):

address = 'Earth'

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

self.name = name

p1 = Person('Bob')

p2 = Person('Alice')

print 'Person.address = ' + Person.address

p1.address = 'China'

print 'p1.address = ' + p1.address

print 'Person.address = ' + Person.address

print 'p2.address = ' + p2.address

结果如下：

Person.address = Earth

p1.address = China

Person.address = Earth

p2.address = Earth

我们发现，在设置了 **p1.address = 'China'**后，p1访问 address 确实变成了 'China'，但是，Person.address和p2.address仍然是'Earch'，怎么回事？

原因是 **p1.address = 'China'并没有改变 Person 的 address**，而是给 **p1这个实例绑定了实例属性address** ，对p1来说，它有一个实例属性address（值是'China'），而它所属的类Person也有一个类属性address，所以:

**访问 p1.address 时**，优先查找实例属性，返回'China'。

**访问 p2.address 时**，p2没有实例属性address，但是有类属性address，因此返回'Earth'。

可见，当实例属性和类属性重名时，实例属性优先级高，它将屏蔽掉对类属性的访问。

当我们把 p1 的 address 实例属性删除后，访问 p1.address 就又返回类属性的值 'Earth'了：

del p1.address

print p1.address

# => Earth

可见，千万不要在实例上修改类属性，它实际上并没有修改类属性，而是给实例绑定了一个实例属性。

#### python中定义实例方法

一个实例的私有属性就是以\_\_开头的属性，无法被外部访问，那这些属性定义有什么用？

虽然私有属性无法从外部访问，但是，从类的内部是可以访问的。除了可以定义实例的属性外，还可以定义实例的方法。

**实例的方法**就是在类中定义的函数，它的第一个参数永远是 self，指向调用该方法的实例本身，其他参数和一个普通函数是完全一样的：

class Person(object):

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

self.\_\_name = name

def get\_name(self):

return self.\_\_name

**get\_name(self)**就是一个实例方法，它的第一个参数是self。\_**\_init\_\_(self, name)**其实也可看做是一个特殊的实例方法。

调用实例方法必须在实例上调用：

p1 = Person('Bob')

print p1.get\_name() # self不需要显式传入

# => Bob

在实例方法内部，可以访问所有实例属性，这样，如果外部需要访问私有属性，可以通过方法调用获得，这种数据封装的形式除了能保护内部数据一致性外，还可以简化外部调用的难度。

#### python中方法也是属性

我们在 **class** 中定义的实例方法其实也是属性，它实际上是一个函数对象：

class Person(object):

def \_\_init\_\_(self, name, score):

self.name = name

self.score = score

def get\_grade(self):

return 'A'

p1 = Person('Bob', 90)

print p1.get\_grade

# => <bound method Person.get\_grade of <\_\_main\_\_.Person object at 0x109e58510>>

print p1.get\_grade()

# => A

也就是说，**p1.get\_grade**返回的是一个函数对象，但这个函数是一个绑定到实例的函数，**p1.get\_grade()**才是方法调用。

因为方法也是一个属性，所以，它也可以动态地添加到实例上，只是需要用 types.MethodType() 把一个函数变为一个方法：

import types

def fn\_get\_grade(self):

if self.score >= 80:

return 'A'

if self.score >= 60:

return 'B'

return 'C'

class Person(object):

def \_\_init\_\_(self, name, score):

self.name = name

self.score = score

p1 = Person('Bob', 90)

p1.get\_grade = types.MethodType(fn\_get\_grade, p1, Person)

print p1.get\_grade()

# => A

p2 = Person('Alice', 65)

print p2.get\_grade()

# ERROR: AttributeError: 'Person' object has no attribute 'get\_grade'

# 因为p2实例并没有绑定get\_grade

给一个实例动态添加方法并不常见，直接在class中定义要更直观。

#### python中定义类方法

和属性类似，方法也分实例方法和类方法。

在**class**中定义的全部是实例方法，实例方法第一个参数 **self**是实例本身。

要在class中定义类方法，需要这么写：

class Person(object):

count = 0

@classmethod

def how\_many(cls):

return cls.count

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

self.name = name

Person.count = Person.count + 1

print Person.how\_many()

p1 = Person('Bob')

print Person.how\_many()

通过标记一个 @classmethod，该方法将绑定到 **Person** 类上，而非类的实例。类方法的第一个参数将传入类本身，通常将参数名命名为 **cls**，上面的 **cls.count**实际上相当于 **Person.count**。

因为是在类上调用，而非实例上调用，因此类方法无法获得任何实例变量，只能获得类的引用。

### 第4章 类的继承

本章讲解Python类的继承，如何判断实例类型，多态以及如何获取对象信息。

#### python中继承一个类

如果已经定义了**Person**类，需要定义新的**Student**和**Teacher**类时，可以直接从Person类继承：

class Person(object):

def \_\_init\_\_(self, name, gender):

self.name = name

self.gender = gender

定义**Student**类时，只需要把额外的属性加上，例如score：

class Student(Person):

def \_\_init\_\_(self, name, gender, score):

super(Student, self).\_\_init\_\_(name, gender)

self.score = score

一定要用 super(Student, self).\_\_init\_\_(name, gender) 去初始化父类，否则，继承自 **Person** 的 **Student** 将没有 **name** 和 **gender**。

函数**super(Student, self)**将返回当前类继承的父类，即**Person**，然后调用**\_\_init\_\_()**方法，注意self参数已在super()中传入，在\_\_init\_\_()中将隐式传递，不需要写出（也不能写）。

#### python中判断类型

函数**isinstance()**可以判断一个变量的类型，既可以用在Python内置的数据类型如**str、list、dict**，也可以用在我们自定义的类，它们本质上都是数据类型。

假设有如下的 **Person、Student** 和 **Teacher**的定义及继承关系如下：

class Person(object):

def \_\_init\_\_(self, name, gender):

self.name = name

self.gender = gender

class Student(Person):

def \_\_init\_\_(self, name, gender, score):

super(Student, self).\_\_init\_\_(name, gender)

self.score = score

class Teacher(Person):

def \_\_init\_\_(self, name, gender, course):

super(Teacher, self).\_\_init\_\_(name, gender)

self.course = course

p = Person('Tim', 'Male')

s = Student('Bob', 'Male', 88)

t = Teacher('Alice', 'Female', 'English')

当我们拿到变量 **p、s、t**时，可以使用 **isinstance** 判断类型：

>>> isinstance(p, Person)

True # p是Person类型

>>> isinstance(p, Student)

False # p不是Student类型

>>> isinstance(p, Teacher)

False # p不是Teacher类型

这说明在继承链上，一个父类的实例不能是子类类型，因为子类比父类多了一些属性和方法。

**我们再考察 s ：**

>>> isinstance(s, Person)

True # s是Person类型

>>> isinstance(s, Student)

True # s是Student类型

>>> isinstance(s, Teacher)

False # s不是Teacher类型

**s** 是Student类型，不是Teacher类型，这很容易理解。但是，**s** 也是Person类型，因为Student继承自Person，虽然它比Person多了一些属性和方法，但是，把 **s** 看成Person的实例也是可以的。

这说明在一条继承链上，一个实例可以看成它本身的类型，也可以看成它父类的类型。

#### python中多态

类具有继承关系，并且子类类型可以向上转型看做父类类型，如果我们从 **Person**派生出 **Student**和**Teacher**，并都写了一个 **whoAmI()**方法：

class Person(object):

def \_\_init\_\_(self, name, gender):

self.name = name

self.gender = gender

def whoAmI(self):

return 'I am a Person, my name is %s' % self.name

class Student(Person):

def \_\_init\_\_(self, name, gender, score):

super(Student, self).\_\_init\_\_(name, gender)

self.score = score

def whoAmI(self):

return 'I am a Student, my name is %s' % self.name

class Teacher(Person):

def \_\_init\_\_(self, name, gender, course):

super(Teacher, self).\_\_init\_\_(name, gender)

self.course = course

def whoAmI(self):

return 'I am a Teacher, my name is %s' % self.name

在一个函数中，如果我们接收一个变量 **x**，则无论该 **x** 是 **Person、Student**还是 **Teacher**，都可以正确打印出结果：

def who\_am\_i(x):

print x.whoAmI()

p = Person('Tim', 'Male')

s = Student('Bob', 'Male', 88)

t = Teacher('Alice', 'Female', 'English')

who\_am\_i(p)

who\_am\_i(s)

who\_am\_i(t)

运行结果：

I am a Person, my name is Tim

I am a Student, my name is Bob

I am a Teacher, my name is Alice

这种行为称为多态。也就是说，方法调用将作用在 **x** 的实际类型上。**s** 是**Student**类型，它实际上拥有自己的 **whoAmI()**方法以及从 Person继承的 whoAmI方法，但调用 **s.whoAmI()**总是先查找它自身的定义，如果没有定义，则顺着继承链向上查找，直到在某个父类中找到为止。

由于Python是动态语言，所以，传递给函数 **who\_am\_i(x)**的参数 **x** 不一定是 Person 或 Person 的子类型。任何数据类型的实例都可以，只要它**有一个whoAmI()**的方法即可：

class Book(object):

def whoAmI(self):

return 'I am a book'

这是动态语言和静态语言（例如Java）最大的差别之一。动态语言调用实例方法，不检查类型，只要方法存在，参数正确，就可以调用。

#### python中多重继承

除了从一个父类继承外，Python允许从多个父类继承，称为多重继承。

多重继承的继承链就不是一棵树了，它像这样：

class A(object):

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

print 'init A...'

self.a = a

class B(A):

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

super(B, self).\_\_init\_\_(a)

print 'init B...'

class C(A):

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

super(C, self).\_\_init\_\_(a)

print 'init C...'

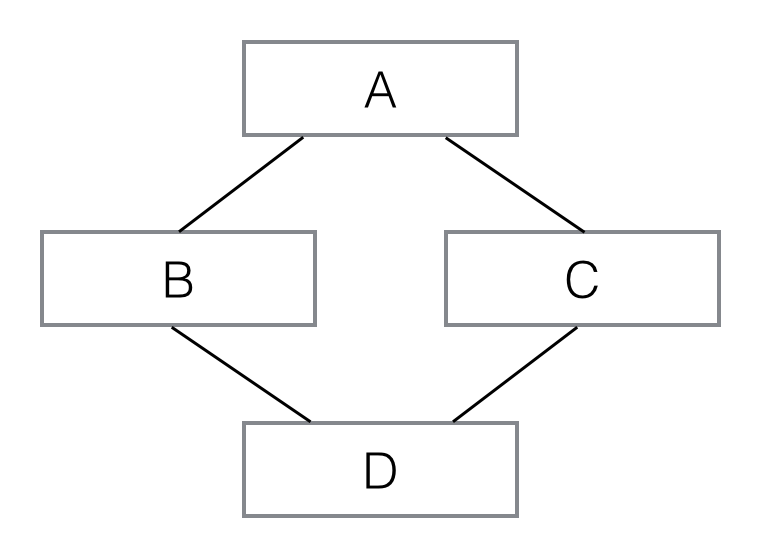
class D(B, C):

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

super(D, self).\_\_init\_\_(a)

print 'init D...'

看下图:

[](http://img.mukewang.com/54daf037000142d207580552.jpg)

像这样，**D**同时继承自 **B** 和 **C**，也就是 **D** 拥有了 **A、B、C** 的全部功能。多重继承通过 **super()**调用**\_\_init\_\_()**方法时，**A** 虽然被继承了两次，但**\_\_init\_\_()**只调用一次：

>>> d = D('d')

init A...

init C...

init B...

init D...

**多重继承的目的**是从两种继承树中分别选择并继承出子类，以便组合功能使用。

举个例子，Python的网络服务器有**TCPServer、UDPServer、UnixStreamServer、UnixDatagramServer**，而服务器运行模式有 **多进程ForkingMixin**和 **多线程ThreadingMixin**两种。

要创建多进程模式的 **TCPServer**：

class MyTCPServer(TCPServer, ForkingMixin)

pass

要创建多线程模式的 **UDPServer**：

class MyUDPServer(UDPServer, ThreadingMixin):

pass

如果没有多重继承，要实现上述所有可能的组合需要 4x2=8 个子类。

#### python中获取对象信息

拿到一个变量，除了用 **isinstance()** 判断它是否是某种类型的实例外，还有没有别的方法获取到更多的信息呢？

例如，已有定义：

class Person(object):

def \_\_init\_\_(self, name, gender):

self.name = name

self.gender = gender

class Student(Person):

def \_\_init\_\_(self, name, gender, score):

super(Student, self).\_\_init\_\_(name, gender)

self.score = score

def whoAmI(self):

return 'I am a Student, my name is %s' % self.name

首先可以用 **type()**函数获取变量的类型，它返回一个 **Type**对象：

>>> type(123)

<type 'int'>

>>> s = Student('Bob', 'Male', 88)

>>> type(s)

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

其次，可以用 **dir()**函数获取变量的所有属性：

>>> dir(123) # 整数也有很多属性...

['\_\_abs\_\_', '\_\_add\_\_', '\_\_and\_\_', '\_\_class\_\_', '\_\_cmp\_\_', ...]

>>> dir(s)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_doc\_\_', '\_\_format\_\_', '\_\_getattribute\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_module\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'gender', 'name', 'score', 'whoAmI']

对于实例变量，**dir()**返回所有实例属性，包括`**\_\_class\_\_**`这类有特殊意义的属性。注意到方法`**whoAmI**`也是 **s**的一个属性。

如何去掉**`\_\_xxx\_\_`**这类的特殊属性，只保留我们自己定义的属性？回顾一下**filter()**函数的用法。

**dir()**返回的属性是字符串列表，如果已知一个属性名称，要获取或者设置对象的属性，就需要用 **getattr()** 和 **setattr( )**函数了：

>>> getattr(s, 'name') # 获取name属性

'Bob'

>>> setattr(s, 'name', 'Adam') # 设置新的name属性

>>> s.name

'Adam'

>>> getattr(s, 'age') # 获取age属性，但是属性不存在，报错：

Traceback (most recent call last):

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

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

>>> getattr(s, 'age', 20) # 获取age属性，如果属性不存在，就返回默认值20：

20

### 第5章 定制类

本章讲解Python的特殊方法，以及如何利用特殊方法定制类，实现各种强大的功能。

#### python中 \_\_str\_\_和\_\_repr\_\_

如果要把一个类的实例变成 **str**，就需要实现特殊方法\_\_str\_\_()：

class Person(object):

def \_\_init\_\_(self, name, gender):

self.name = name

self.gender = gender

def \_\_str\_\_(self):

return '(Person: %s, %s)' % (self.name, self.gender)

现在，在交互式命令行下用 **print**试试：

>>> p = Person('Bob', 'male')

>>> print p

(Person: Bob, male)

但是，如果直接敲变量 **p**：

>>> p

<main.Person object at 0x10c941890>

似乎\_\_str\_\_() 不会被调用。

因为 Python 定义了**\_\_str\_\_()**和**\_\_repr\_\_()**两种方法，\_\_str\_\_()用于显示给用户，而**\_\_repr\_\_()**用于显示给开发人员。

有一个偷懒的定义\_\_repr\_\_的方法：

class Person(object):

def \_\_init\_\_(self, name, gender):

self.name = name

self.gender = gender

def \_\_str\_\_(self):

return '(Person: %s, %s)' % (self.name, self.gender)

\_\_repr\_\_ = \_\_str\_\_

#### python中 \_\_cmp\_\_

对 **int**、**str**等内置数据类型排序时，Python的 **sorted()**按照默认的比较函数 **cmp**排序，但是，如果对一组 **Student**类的实例排序时，就必须提供我们自己的特殊方法 \_\_cmp\_\_()：

class Student(object):

def \_\_init\_\_(self, name, score):

self.name = name

self.score = score

def \_\_str\_\_(self):

return '(%s: %s)' % (self.name, self.score)

\_\_repr\_\_ = \_\_str\_\_

def \_\_cmp\_\_(self, s):

if self.name < s.name:

return -1

elif self.name > s.name:

return 1

else:

return 0

上述 Student 类实现了\_\_cmp\_\_()方法，**\_\_cmp\_\_**用实例自身**self**和传入的实例 **s**进行比较，如果 **self** 应该排在前面，就返回 -1，如果 **s** 应该排在前面，就返回1，如果两者相当，返回 0。

Student类实现了按name进行排序：

>>> L = [Student('Tim', 99), Student('Bob', 88), Student('Alice', 77)]

>>> print sorted(L)

[(Alice: 77), (Bob: 88), (Tim: 99)]

**注意**: 如果list不仅仅包含 Student 类，则 \_\_cmp\_\_ 可能会报错：

L = [Student('Tim', 99), Student('Bob', 88), 100, 'Hello']

print sorted(L)

请思考如何解决。

#### python中 \_\_len\_\_

如果一个类表现得像一个list，要获取有多少个元素，就得用 **len()**函数。

要让 **len()**函数工作正常，类必须提供一个特殊方法\_\_len\_\_()，它返回元素的个数。

例如，我们写一个 Students 类，把名字传进去：

class Students(object):

def \_\_init\_\_(self, \*args):

self.names = args

def \_\_len\_\_(self):

return len(self.names)

只要正确实现了**\_\_len\_\_()**方法，就可以用**len()**函数返回**Students**实例的“长度”：

>>> ss = Students('Bob', 'Alice', 'Tim')

>>> print len(ss)

3

#### python中数学运算

Python 提供的基本数据类型 **int、float**可以做整数和浮点的四则运算以及乘方等运算。

但是，四则运算不局限于int和float，还可以是有理数、矩阵等。

要表示有理数，可以用一个**Rational类**来表示：

class Rational(object):

def \_\_init\_\_(self, p, q):

self.p = p

self.q = q

p、q 都是整数，表示有理数 p/q。

如果要让**Rational进行+运算**，需要正确实现\_\_add\_\_：

class Rational(object):

def \_\_init\_\_(self, p, q):

self.p = p

self.q = q

def \_\_add\_\_(self, r):

return Rational(self.p \* r.q + self.q \* r.p, self.q \* r.q)

def \_\_str\_\_(self):

return '%s/%s' % (self.p, self.q)

\_\_repr\_\_ = \_\_str\_\_

现在可以试试有理数加法：

>>> r1 = Rational(1, 3)

>>> r2 = Rational(1, 2)

>>> print r1 + r2

5/6

#### python中类型转换

**Rational**类实现了有理数运算，但是，如果要把结果转为 **int**或 **float** 怎么办？

考察整数和浮点数的转换：

>>> int(12.34)

12

>>> float(12)

12.0

如果要把 **Rational** 转为 **int**，应该使用：

r = Rational(12, 5)

n = int(r)

要让**int()**函数正常工作，只需要实现特殊方法\_\_int\_\_():

class Rational(object):

def \_\_init\_\_(self, p, q):

self.p = p

self.q = q

def \_\_int\_\_(self):

return self.p // self.q

**结果如下：**

>>> print int(Rational(7, 2))

3

>>> print int(Rational(1, 3))

0

同理，要让**float()**函数正常工作，只需要实现特殊方法**\_\_float\_\_()**。

#### python中 @property

考察 **Student**类：

class Student(object):

def \_\_init\_\_(self, name, score):

self.name = name

self.score = score

当我们想要修改一个 **Student** 的 **scroe** 属性时，可以这么写：

s = Student('Bob', 59)

s.score = 60

但是也可以这么写：

s.score = 1000

显然，直接给属性赋值无法检查分数的有效性。

如果利用两个方法：

class Student(object):

def \_\_init\_\_(self, name, score):

self.name = name

self.\_\_score = score

def get\_score(self):

return self.\_\_score

def set\_score(self, score):

if score < 0 or score > 100:

raise ValueError('invalid score')

self.\_\_score = score

这样一来，**s.set\_score(1000)** 就会报错。

这种使用 **get/set**方法来封装对一个属性的访问在许多面向对象编程的语言中都很常见。

但是写 **s.get\_score()** 和 **s.set\_score()**没有直接写 **s.score** 来得直接。

有没有两全其美的方法？----有。

因为Python支持高阶函数，在函数式编程中我们介绍了装饰器函数，可以用装饰器函数把 **get/set** 方法“装饰”成属性调用：

class Student(object):

def \_\_init\_\_(self, name, score):

self.name = name

self.\_\_score = score

@property

def score(self):

return self.\_\_score

@score.setter

def score(self, score):

if score < 0 or score > 100:

raise ValueError('invalid score')

self.\_\_score = score

**注意:** 第一个score(self)是get方法，用@property装饰，第二个score(self, score)是set方法，用@score.setter装饰，@score.setter是前一个@property装饰后的副产品。

现在，就可以像使用属性一样设置score了：

>>> s = Student('Bob', 59)

>>> s.score = 60

>>> print s.score

60

>>> s.score = 1000

Traceback (most recent call last):

...

ValueError: invalid score

说明对 **score** 赋值实际调用的是 **set方法**。

#### python中 \_\_slots\_\_

由于Python是动态语言，任何实例在运行期都可以动态地添加属性。

如果要限制添加的属性，例如，**Student**类只允许添加 **name、gender**和**score**这3个属性，就可以利用Python的一个特殊的**\_\_slots\_\_**来实现。

顾名思义，**\_\_slots\_\_**是指一个类允许的属性列表：

class Student(object):

\_\_slots\_\_ = ('name', 'gender', 'score')

def \_\_init\_\_(self, name, gender, score):

self.name = name

self.gender = gender

self.score = score

现在，对实例进行操作：

>>> s = Student('Bob', 'male', 59)

>>> s.name = 'Tim' # OK

>>> s.score = 99 # OK

>>> s.grade = 'A'

Traceback (most recent call last):

...

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

**\_\_slots\_\_**的目的是限制当前类所能拥有的属性，如果不需要添加任意动态的属性，使用**\_\_slots\_\_**也能节省内存。

#### python中 \_\_call\_\_

在Python中，函数其实是一个对象：

>>> f = abs

>>> f.\_\_name\_\_

'abs'

>>> f(-123)

123

由于 **f** 可以被调用，所以，**f** 被称为可调用对象。

所有的函数都是可调用对象。

一个类实例也可以变成一个可调用对象，只需要实现一个特殊方法**\_\_call\_\_()**。

我们把 **Person** 类变成一个可调用对象：

class Person(object):

def \_\_init\_\_(self, name, gender):

self.name = name

self.gender = gender

def \_\_call\_\_(self, friend):

print 'My name is %s...' % self.name

print 'My friend is %s...' % friend

现在可以对 **Person** 实例直接调用：

>>> p = Person('Bob', 'male')

>>> p('Tim')

My name is Bob...

My friend is Tim...

单看 **p('Tim')**你无法确定 **p** 是一个函数还是一个类实例，所以，在Python中，函数也是对象，对象和函数的区别并不显著。

## Python文件处理

简介：任何语言都离不开对文件的操作，Python语言是如何来操作和管理文件的。今天就让我们带您走入文件的世界，一起创建文件，一起读写文件，一起管理我们的文件。掌握文件的基本概念、基本操作，了解文件属性、 linux 系统的文件管理机制及os模块对文件和目录的处理等相关内容。

### Python文件基础操作

本章主要讲解文件打开及文件不同的打开方式、使用read,readlines及迭代器访问文件、文件写入方式及写缓存处理、文件关闭及由不关闭导致的问题、文件指针及文件指针操作

#### Python文件打开及读写方式

##### 文件的打开方法：

open(name[,mode[buf]])

name:文件路径

mode:打开方式

buf: 缓冲buffering大小

|  |  |  |
| --- | --- | --- |
| mode | 说明 | 注意 |
| ‘r’ | 只读方式打开 | 文件必须存在 |
| ‘w’ | 只写方式打开 | 文件不存在就创建文件  文件存在则清空文件内容 |
| ‘a’ | 追加方式打开 | 文件不存在创建文件 |
| ‘r+’/’w+’ | 读写方式打开 |  |
| ‘a+’ | 追加和读写方式打开 |  |
| ‘rb’,’wb’,’ab’,’rb+’,’wb+’,’ab+’ 二进制方式打开 | | |

##### 文件的读取方式：

read([size]) 读取文件(读取size个字节，默认读取全部)

readline([size]) 读取一行

readlines([size]) 读取文件，返回每一行所组成的列表（不建议大文件的读取，会占用大量内存）

iter:使用迭代器读取文件

##### 文件的写入方式：

write(str) 将字符串写入文件

writelines(sequence\_of\_strings) 写多行到文件

### 第2章 文件属性及OS模块使用

本章主要讲解文件属性、标准文件、文件命令行参数、文件编码格式、使用os模块中方法对文件进行操作、使用os模块中的方法对目录进行操作