【收藏】Python教程基础篇，超详细超长！

机器学习算法与Python学习 1周前

来源：雷课

安装Python

前往 官网下载 对应平台对应工具。另外Python2.7版本和3.3版本并不兼容，所以开发时请注意使用Python的版本。

作为Mac OS X使用者，其实更推荐 PyCharm IDE 。安装之后直接使用即可。

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：

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。

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，1.23×10^9 和 12.3×10^8 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，1.23×10^9 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。

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。

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。

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。

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 jump
```

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：

print也可以打印整数，或者计算结果：

```
>>> print 300300    #运行结果>>> print 100 + 200300    #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 + 200100 + 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.'
```

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 = 'Chars'    # 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 = "Chars"; // 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 = 10x = 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’。

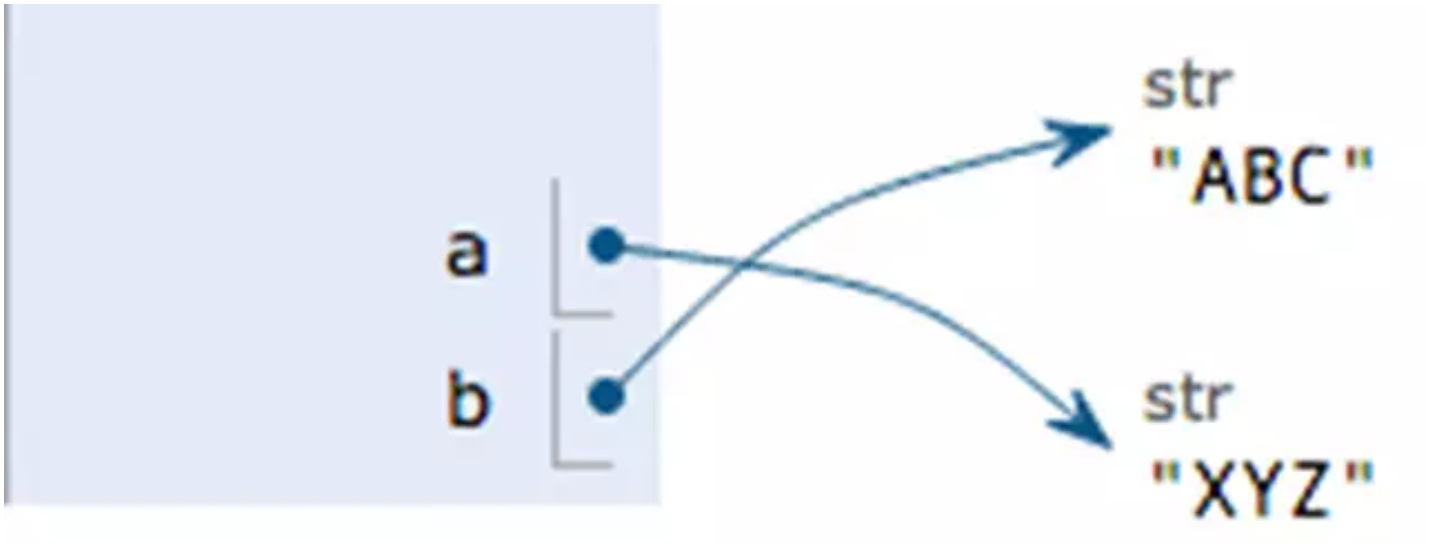
也可以把一个变量a赋值给另一个变量b，这个操作实际上是把变量b指向变量a所指向的数据，例如下面的代码：

```
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'` ：

，创建了字符串 `'XYZ'` ，并把a的指向改为 `'XYZ'` ，但b并没有更改：



所以，最后打印变量b的结果自然是 `'ABC'` 了。

字符串

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 Chars's Blog'
```

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
```

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 Chars's Blog!'''
```

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。

List

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['
```

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 = []
```

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** 。

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** 。

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。

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']
```

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'
```

Tuple

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
```

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 t1
```

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,)
```

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'
```

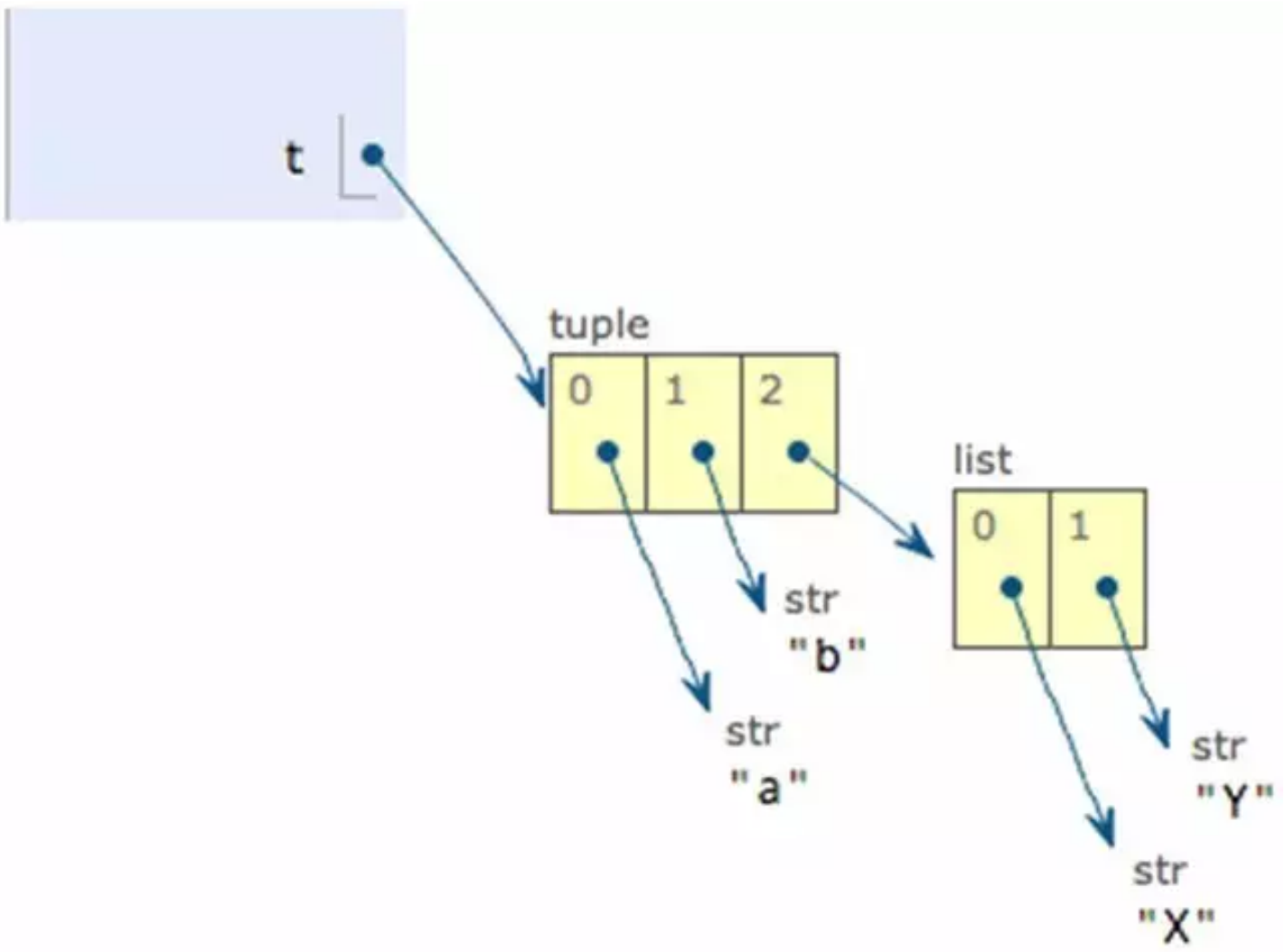
再看看tuple的内容：

```
>>> 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：

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



表面上看，tuple的元素确实变了，但其实变的不是 tuple 的元素，而是list的元素。

tuple一开始指向的list并没有改成别的list，所以，tuple所谓的“不变”是说，tuple的每个元素，指向永远不变。即指向’a’，就不能改成指向’b’，指向一个list，就不能改成指向其他对象，但指向的这个list本身是可变的！

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

条件判断和循环

条件判断

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。

if语句

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 = 20if 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 20adult
```

if-else语句

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'
```

细心的读者可以发现，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“:”。

if-elif-else语句

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:    pri
```

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。

循环

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。

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 = 10x = 0while 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。

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 = 0x = 1while True:
    sum = sum + x
    x = x + 1
    if x > 100:        breakprint 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。

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 = 0.0n = 0for x in L:
    sum = sum + x
```

```
    n = n + 1print 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
```

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。

Dict类型

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。

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
```

Dict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。

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}
```

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... L
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。

Set类型

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。

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 sTrue
```

Bill是该班的同学吗？

```
>>> 'Bill' in sFalse
```

bart是该班的同学吗？

```
>>> 'bart' in sFalse
```

看来大小写很重要，’ Bart’ 和 ‘bart’被认为是两个不同的元素。

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':
```

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
```

这样一来，代码就简单多了。

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。

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

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 = 12.34r2 = 9.08r3 = 73.1s1 = 3.14 * r1 * r1s2 = 3.14 * r2 * r2s3 = 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。

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

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 g
```

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：

```
>>> 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'
```

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。

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 mathdef 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, y151.961524227 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。

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>>> fact(100)933262154439441526816992388562667004907159682643
```

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)。

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
```

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')
```

```
>>> 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
```

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，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。

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):...     p
```

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.va
```

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-valueiter
```

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 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 : 85Adam : 95Ba
```

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。

列表

列表生成

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[1×1, 2×2, 3×3, ..., 10×10]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.iterite
```

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
```

Name
Score

Lisa	85
Adam	95
Bart	59

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)
```


机器学习算法与Python学习
ID: guodongwei1991
引领AI，指向优秀的你。与10W+AIer同行！