# Python教程

这是小白的Python新手教程。

Python是一种计算机程序设计语言。你可能已经听说过很多种流行的编程语言，比如非常难学的C语言，非常流行的Java语言，适合初学者的Basic语言，适合网页编程的JavaScript语言，等等。

那Python是一种什么语言？

首选，我们普及一下编程语言的基础知识。用任何编程语言来开发程序，都是为了让计算机干活，比如下载一个MP3，编写一个文档等等，而计算机干活的 CPU只认识机器指令，所以，尽管不同的编程语言差异极大，最后都得“翻译”成CPU可以执行的机器指令。而不同的编程语言，干同一个活，编写的代码量， 差距也很大。

比如，完成同一个任务，C语言要写1000行代码，Java只需要写100行，而Python可能只要20行。

所以Python是一种相当高级的语言。

你也许会问，代码少还不好？代码少的代价是运行速度慢，C程序运行1秒钟，Java程序可能需要2秒，而Python程序可能就需要10秒。

那是不是越低级的程序越难学，越高级的程序越简单？表面上来说，是的，但是，在非常高的抽象计算中，高级的Python程序设计也是非常难学的，所以，高级程序语言不等于简单。

但是，对于初学者和完成普通任务，Python语言是非常简单易用的。连Google都在大规模使用Python，你就不用担心学了会没用。

用Python可以做什么？可以做日常任务，比如自动备份你的MP3；可以做网站，很多著名的网站包括YouTube就是Python写的；可以做网络游戏的后台，很多在线游戏的后台都是Python开发的。总之就是能干很多很多事啦。

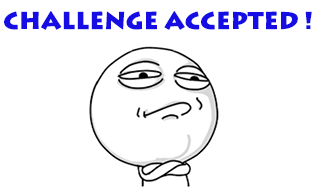
Python当然也有不能干的事情，比如写操作系统，这个只能用C语言写；写手机应用，只能用Objective-C（针对iPhone）和Java（针对Android）；写3D游戏，最好用C或C++。

如果你是小白用户，满足以下条件：

* 会使用电脑，但从来没写过程序；
* 还记得初中数学学的方程式和一点点代数知识；
* 想从编程小白变成专业的软件架构师；
* 每天能抽出半个小时学习。

不要再犹豫了，这个教程就是为你准备的！

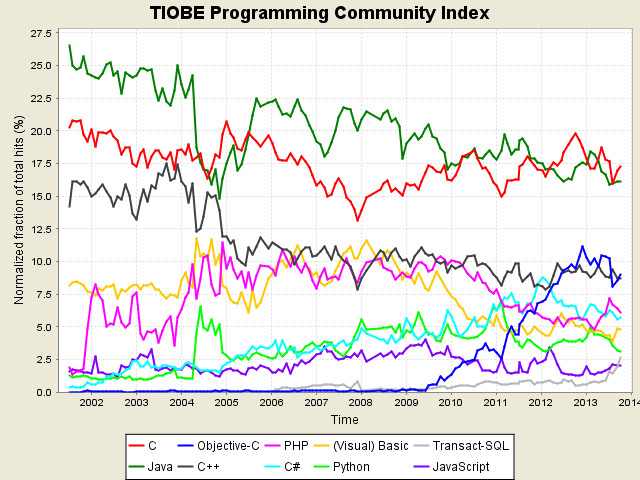
准备好了吗？



## Python简介

Python是著名的“龟叔”Guido van Rossum在1989年圣诞节期间，为了打发无聊的圣诞节而编写的一个编程语言。

现在，全世界差不多有600多种编程语言，但流行的编程语言也就那么20来种。如果你听说过TIOBE排行榜，你就能知道编程语言的大致流行程度。这是最近10年最常用的10种编程语言的变化图：



总的来说，这几种编程语言各有千秋。C语言是可以用来编写操作系统的贴近硬件的语言，所以，C语言适合开发那些追求运行速度、充分发挥硬件性能的程序。而Python是用来编写应用程序的高级编程语言。

当你用一种语言开始作真正的软件开发时，你除了编写代码外，还需要很多基本的已经写好的现成的东西，来帮助你加快开发进度。比如说，要编写一个电子 邮件客户端，如果先从最底层开始编写网络协议相关的代码，那估计一年半载也开发不出来。高级编程语言通常都会提供一个比较完善的基础代码库，让你能直接调 用，比如，针对电子邮件协议的SMTP库，针对桌面环境的GUI库，在这些已有的代码库的基础上开发，一个电子邮件客户端几天就能开发出来。

Python就为我们提供了非常完善的基础代码库，覆盖了网络、文件、GUI、数据库、文本等大量内容，被形象地称作“内置电池（batteries included）”。用Python开发，许多功能不必从零编写，直接使用现成的即可。

除了内置的库外，Python还有大量的第三方库，也就是别人开发的，供你直接使用的东西。当然，如果你开发的代码通过很好的封装，也可以作为第三方库给别人使用。

许多大型网站就是用Python开发的，例如YouTube、[Instagram](http://instagram.com/)，还有国内的[豆瓣](http://www.douban.com/)。很多大公司，包括Google、Yahoo等，甚至[NASA](http://www.nasa.gov/)（美国航空航天局）都大量地使用Python。

龟叔给Python的定位是“优雅”、“明确”、“简单”，所以Python程序看上去总是简单易懂，初学者学Python，不但入门容易，而且将来深入下去，可以编写那些非常非常复杂的程序。

总的来说，Python的哲学就是简单优雅，尽量写容易看明白的代码，尽量写少的代码。如果一个资深程序员向你炫耀他写的晦涩难懂、动不动就几万行的代码，你可以尽情地嘲笑他。

那Python适合开发哪些类型的应用呢？

首选是网络应用，包括网站、后台服务等等；

其次是许多日常需要的小工具，包括系统管理员需要的脚本任务等等；

另外就是把其他语言开发的程序再包装起来，方便使用。

最后说说Python的缺点。

任何编程语言都有缺点，Python也不例外。优点说过了，那Python有哪些缺点呢？

第一个缺点就是运行速度慢，和C程序相比非常慢，因为Python是解释型语言，你的代码在执行时会一行一行地翻译成CPU能理解的机器码，这个翻译过程非常耗时，所以很慢。而C程序是运行前直接编译成CPU能执行的机器码，所以非常快。

但是大量的应用程序不需要这么快的运行速度，因为用户根本感觉不出来。例如开发一个下载MP3的网络应用程序，C程序的运行时间需要0.001秒， 而Python程序的运行时间需要0.1秒，慢了100倍，但由于网络更慢，需要等待1秒，你想，用户能感觉到1.001秒和1.1秒的区别吗？这就好比 F1赛车和普通的出租车在北京三环路上行驶的道理一样，虽然F1赛车理论时速高达400公里，但由于三环路堵车的时速只有20公里，因此，作为乘客，你感 觉的时速永远是20公里。



第二个缺点就是代码不能加密。如果要发布你的Python程序，实际上就是发布源代码，这一点跟C语言不同，C语言不用发布源代码，只需要把编译后 的机器码（也就是你在Windows上常见的xxx.exe文件）发布出去。要从机器码反推出C代码是不可能的，所以，凡是编译型的语言，都没有这个问 题，而解释型的语言，则必须把源码发布出去。

这个缺点仅限于你要编写的软件需要卖给别人挣钱的时候。好消息是目前的互联网时代，靠卖软件授权的商业模式越来越少了，靠网站和移动应用卖服务的模式越来越多了，后一种模式不需要把源码给别人。

再说了，现在如火如荼的开源运动和互联网自由开放的精神是一致的，互联网上有无数非常优秀的像Linux一样的开源代码，我们千万不要高估自己写的 代码真的有非常大的“商业价值”。那些大公司的代码不愿意开放的更重要的原因是代码写得太烂了，一旦开源，就没人敢用他们的产品了。



当然，Python还有其他若干小缺点，请自行忽略，就不一一列举了。

## 一、安装Python

因为Python是跨平台的，它可以运行在Windows、Mac和各种Linux/Unix系统上。在Windows上写Python程序，放到Linux上也是能够运行的。

要开始学习Python编程，首先就得把Python安装到你的电脑里。安装后，你会得到Python解释器（就是负责运行Python程序的），一个命令行交互环境，还有一个简单的集成开发环境。

### 2.x还是3.x

目前，Python有两个版本，一个是2.x版，一个是3.x版，这两个版本是不兼容的，因为现在Python正在朝着3.x版本进化，在进化过程中，大量的针对2.x版本的代码要修改后才能运行，所以，目前有许多第三方库还暂时无法在3.x上使用。

为了保证你的程序能用到大量的第三方库，我们的教程仍以2.x版本为基础，确切地说，是2.7版本。请确保你的电脑上安装的Python版本是2.7.x，这样，你才能无痛学习这个教程。

### 在Mac上安装Python

如果你正在使用Mac，系统是OS X 10.8或者最新的10.9 Mavericks，恭喜你，系统自带了Python 2.7。如果你的系统版本低于10.8，请自行备份系统并免费升级到最新的10.9，就可以获得Python 2.7。

查看系统版本的办法是点击左上角的苹果图标，选择“关于本机”：



### 在Linux上安装Python

如果你正在使用Linux，那我可以假定你有Linux系统管理经验，自行安装Python 2.7应该没有问题，否则，请换回Windows系统。

对于大量的目前仍在使用Windows的同学，如果短期内没有打算换Mac，就可以继续阅读以下内容。

### 在Windows上安装Python

首先，从Python的官方网站[www.python.org](http://www.python.org/)下载最新的2.7.6版本，地址是这个：

<http://www.python.org/ftp/python/2.7.6/python-2.7.6.msi>

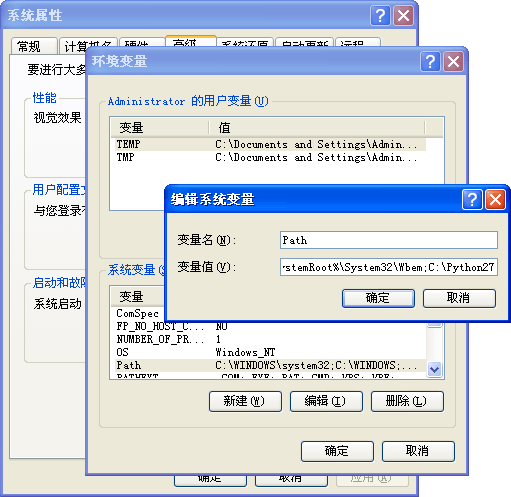
然后，运行下载的MSI安装包，不需要更改任何默认设置，直接一路点“Next”即可完成安装：

默认会安装到C:\Python27目录下，但是当你兴致勃勃地打开命令提示符窗口，敲入python后，会得到：

‘python’不是内部或外部命令，也不是可运行的程序或批处理文件。

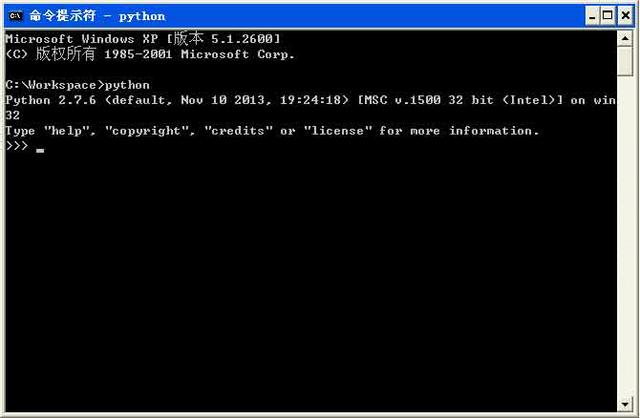
这是因为Windows会根据一个Path的环境变量设定的路径去查找python.exe，如果没找到，就会报错。解决办法是把python.exe所在的路径C:\Python27添加到Path中。

在控制面板中打开“系统属性”，点击“高级”，“环境变量”，打开“环境变量”窗口，在系统变量中，找到“Path”变量，然后点击“编辑”：



在“编辑系统变量”的窗口中，可以看到，变量名是Path，在变量值的最后面，先添加一个分号“;”（注意用英文输入法，千万不要输入中文分号），再写上C:\Python27（如果安装的时候没有更改过安装目录），然后连续点“确定”，“确定”，“确定”把所有窗口都关掉。

现在，再打开一个新的命令行窗口（一定要关掉原来的命令行窗口，再新开一个），输入python：



看到上面的画面，就说明Python安装成功！

你看到提示符>>>就表示我们已经在Python交互式环境中了，可以输入任何Python代码，回车后会立刻得到执行结果。现在，输入exit()并回车，就可以退出Python交互式环境（直接关掉命令行窗口也可以！）。

### 小结

学会如何把Python安装到计算机中，并且熟练打开和退出Python交互式环境。

## 二、Python解释器

当我们编写Python代码时，我们得到的是一个包含Python代码的以.py为扩展名的文本文件。要运行代码，就需要Python解释器去执行.py文件。

由于整个Python语言从规范到解释器都是开源的，所以理论上，只要水平够高，任何人都可以编写Python解释器来执行Python代码（当然难度很大）。事实上，确实存在多种Python解释器。

### CPython

当我们从[Python官方网站](https://www.python.org/)下载并安装好Python 2.7后，我们就直接获得了一个官方版本的解释器：CPython。这个解释器是用C语言开发的，所以叫CPython。在命令行下运行python就是启动CPython解释器。

CPython是使用最广的Python解释器。教程的所有代码也都在CPython下执行。

### IPython

IPython是基于CPython之上的一个交互式解释器，也就是说，IPython只是在交互方式上有所增强，但是执行Python代码的功能和CPython是完全一样的。好比很多国产浏览器虽然外观不同，但内核其实都是调用了IE。

CPython用>>>作为提示符，而IPython用In [序号]:作为提示符。

### PyPy

PyPy是另一个Python解释器，它的目标是执行速度。PyPy采用[JIT技术](http://en.wikipedia.org/wiki/Just-in-time_compilation)，对Python代码进行动态编译（注意不是解释），所以可以显著提高Python代码的执行速度。

绝大部分Python代码都可以在PyPy下运行，但是PyPy和CPython有一些是不同的，这就导致相同的Python代码在两种解释器下执行可能会有不同的结果。如果你的代码要放到PyPy下执行，就需要了解[PyPy和CPython的不同点](http://pypy.readthedocs.org/en/latest/cpython_differences.html)。

### Jython

Jython是运行在Java平台上的Python解释器，可以直接把Python代码编译成Java字节码执行。

### IronPython

IronPython和Jython类似，只不过IronPython是运行在微软.Net平台上的Python解释器，可以直接把Python代码编译成.Net的字节码。

### 小结

Python的解释器很多，但使用最广泛的还是CPython。如果要和Java或.Net平台交互，最好的办法不是用Jython或IronPython，而是通过网络调用来交互，确保各程序之间的独立性。

本教程的所有代码只确保在CPython 2.7版本下运行。请务必在本地安装CPython（也就是从Python官方网站下载的安装程序）。

此外，教程还内嵌一个IPython的Web版本，用来在浏览器内练习执行一些Python代码。要注意两者功能一样，输入的代码一样，但是提示符有所不同。另外，不是所有代码都能在Web版本的IPython中执行，出于安全原因，很多操作（比如文件操作）是受限的，所以有些代码必须在本地环境执行代码。

## 三、第一个Python程序

现在，了解了如何启动和退出Python的交互式环境，我们就可以正式开始编写Python代码了。

在写代码之前，请千万不要用“复制”-“粘贴”把代码从页面粘贴到你自己的电脑上。写程序也讲究一个感觉，你需要一个字母一个字母地把代码自己敲进去，在敲代码的过程中，初学者经常会敲错代码，所以，你需要仔细地检查、对照，才能以最快的速度掌握如何写程序。

在交互式环境的提示符>>>下，直接输入代码，按回车，就可以立刻得到代码执行结果。现在，试试输入100+200，看看计算结果是不是300：

>>> 100+200

300

很简单吧，任何有效的数学计算都可以算出来。

如果要让Python打印出指定的文字，可以用print语句，然后把希望打印的文字用单引号或者双引号括起来，但不能混用单引号和双引号：

>>> print 'hello, world'

hello, world

这种用单引号或者双引号括起来的文本在程序中叫字符串，今后我们还会经常遇到。

最后，用exit()退出Python，我们的第一个Python程序完成！唯一的缺憾是没有保存下来，下次运行时还要再输入一遍代码。

#### 小结

在Python交互式命令行下，可以直接输入代码，然后执行，并立刻得到结果。



### 1.使用文本编辑器

在Python的交互式命令行写程序，好处是一下就能得到结果，坏处是没法保存，下次还想运行的时候，还得再敲一遍。

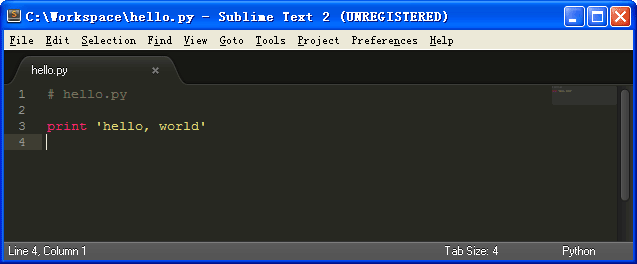
所以，实际开发的时候，我们总是使用一个文本编辑器来写代码，写完了，保存为一个文件，这样，程序就可以反复运行了。

现在，我们就把上次的'hello, world'程序用文本编辑器写出来，保存下来。

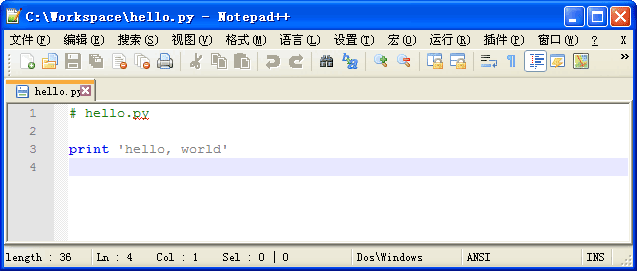
那么问题来了：文本编辑器到底哪家强？

推荐两款文本编辑器：

一个是[Sublime Text](http://www.sublimetext.com/)，免费使用，但是不付费会弹出提示框：



一个是[Notepad++](http://notepad-plus-plus.org/)，免费使用，有中文界面：



请注意，用哪个都行，但是绝对不能用Word和Windows自带的记事本。Word保存的不是纯文本文件，而记事本会自作聪明地在文件开始的地方加上几个特殊字符（UTF-8 BOM），结果会导致程序运行出现莫名其妙的错误。

安装好文本编辑器后，输入以下代码：

print 'hello, world'

注意print前面不要有任何空格。然后，选择一个目录，例如C:\Workspace，把文件保存为hello.py，就可以打开命令行窗口，把当前目录切换到hello.py所在目录，就可以运行这个程序了：

C:\Workspace>python hello.py

hello, world

也可以保存为别的名字，比如abc.py，但是必须要以.py结尾，其他的都不行。此外，文件名只能是英文字母、数字和下划线的组合。

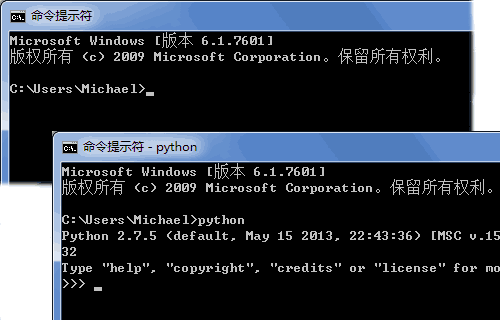
如果当前目录下没有hello.py这个文件，运行python hello.py就会报错：

python hello.py

python: can't open file 'hello.py': [Errno 2] No such file or directory

报错的意思就是，无法打开hello.py这个文件，因为文件不存在。这个时候，就要检查一下当前目录下是否有这个文件了。

请注意区分命令行模式和Python交互模式：



看到类似C:\>是在Windows提供的命令行模式，看到>>>是在Python交互式环境下。

在命令行模式下，可以执行python进入Python交互式环境，也可以执行python hello.py运行一个.py文件，但是在Python交互式环境下，只能输入Python代码执行。

#### 直接运行py文件

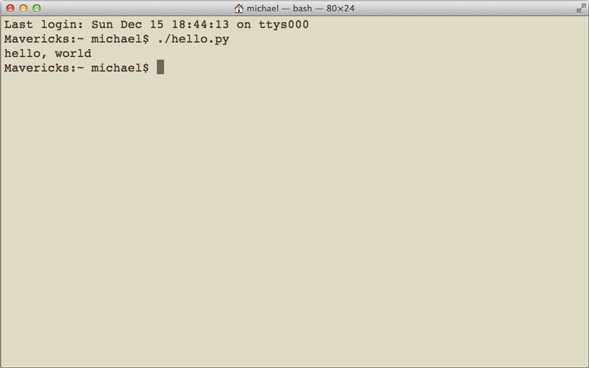
还有同学问，能不能像.exe文件那样直接运行.py文件呢？在Windows上是不行的，但是，在Mac和Linux上是可以的，方法是在.py文件的第一行加上：

#!/usr/bin/env python

然后，通过命令：

$ chmod a+x hello.py

就可以直接运行hello.py了，比如在Mac下运行：



#### 小结

用文本编辑器写Python程序，然后保存为后缀为.py的文件，就可以用Python直接运行这个程序了。

Python的交互模式和直接运行.py文件有什么区别呢？

直接输入python进入交互模式，相当于启动了Python解释器，但是等待你一行一行地输入源代码，每输入一行就执行一行。

直接运行.py文件相当于启动了Python解释器，然后一次性把.py文件的源代码给执行了，你是没有机会输入源代码的。

用Python开发程序，完全可以一边在文本编辑器里写代码，一边开一个交互式命令窗口，在写代码的过程中，把部分代码粘到命令行去验证，事半功倍！前提是得有个27'的超大显示器！

### 2.输入和输出

#### 输出

用print加上字符串，就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：

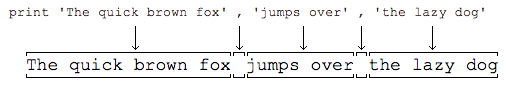
>>> print 'hello, world'

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

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 300

300

>>> print 100 + 200

300

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

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。

#### 输入

现在，你已经可以用print输出你想要的结果了。但是，如果要让用户从电脑输入一些字符怎么办？Python提供了一个raw\_input，可以让用户输入字符串，并存放到一个变量里。比如输入用户的名字：

>>> name = raw\_input()

Michael

当你输入name = raw\_input()并按下回车后，Python交互式命令行就在等待你的输入了。这时，你可以输入任意字符，然后按回车后完成输入。

输入完成后，不会有任何提示，Python交互式命令行又回到>>>状态了。那我们刚才输入的内容到哪去了？答案是存放到name变量里了。可以直接输入name查看变量内容：

>>> name

'Michael'

**什么是变量？**请回忆初中数学所学的代数基础知识：

设正方形的边长为a，则正方形的面积为a x a。把边长a看做一个变量，我们就可以根据a的值计算正方形的面积，比如：

若a=2，则面积为a x a = 2 x 2 = 4；

若a=3.5，则面积为a x a = 3.5 x 3.5 = 12.25。

在计算机程序中，变量不仅可以为整数或浮点数，还可以是字符串，因此，name作为一个变量就是一个字符串。

要打印出name变量的内容，除了直接写name然后按回车外，还可以用print语句：

>>> print name

Michael

有了输入和输出，我们就可以把上次打印'hello, world'的程序改成有点意义的程序了：

name = raw\_input()

print 'hello,', name

运行上面的程序，第一行代码会让用户输入任意字符作为自己的名字，然后存入name变量中；第二行代码会根据用户的名字向用户说hello，比如输入Michael：

C:\Workspace> python hello.py

Michael

hello, Michael

但是程序运行的时候，没有任何提示信息告诉用户：“嘿，赶紧输入你的名字”，这样显得很不友好。幸好，raw\_input可以让你显示一个字符串来提示用户，于是我们把代码改成：

name = raw\_input('please enter your name: ')

print 'hello,', name

再次运行这个程序，你会发现，程序一运行，会首先打印出please enter your name:，这样，用户就可以根据提示，输入名字后，得到hello, xxx的输出：

C:\Workspace> python hello.py

please enter your name: Michael

hello, Michael

每次运行该程序，根据用户输入的不同，输出结果也会不同。

在命令行下，输入和输出就是这么简单。

#### 小结

任何计算机程序都是为了执行一个特定的任务，有了输入，用户才能告诉计算机程序所需的信息，有了输出，程序运行后才能告诉用户任务的结果。

输入是Input，输出是Output，因此，我们把输入输出统称为Input/Output，或者简写为IO。

raw\_input和print是在命令行下面最基本的输入和输出，但是，用户也可以通过其他更高级的图形界面完成输入和输出，比如，在网页上的一个文本框输入自己的名字，点击“确定”后在网页上看到输出信息。

## 四、Python基础

Python是一种计算机编程语言。计算机编程语言和我们日常使用 的自然语言有所不同，最大的区别就是，自然语言在不同的语境下有不同的理解，而计算机要根据编程语言执行任务，就必须保证编程语言写出的程序决不能有歧 义，所以，任何一种编程语言都有自己的一套语法，编译器或者解释器就是负责把符合语法的程序代码转换成CPU能够执行的机器码，然后执行。Python也 不例外。

Python的语法比较简单，采用缩进方式，写出来的代码就像下面的样子：

# print absolute value of an integer:

a = 100

if a >= 0:

print a

else:

print -a

以#开头的语句是注释，注释是给人看的，可以是任意内容，解释器会忽略掉注释。其他每一行都是一个语句，当语句以冒号“:”结尾时，缩进的语句视为代码块。

缩进有利有弊。好处是强迫你写出格式化的代码，但没有规定缩进是几个空格还是Tab。按照约定俗成的管理，应该始终坚持使用4个空格的缩进。

缩进的另一个好处是强迫你写出缩进较少的代码，你会倾向于把一段很长的代码拆分成若干函数，从而得到缩进较少的代码。

缩进的坏处就是“复制－粘贴”功能失效了，这是最坑爹的地方。当你重构代码时，粘贴过去的代码必须重新检查缩进是否正确。此外，IDE很难像格式化Java代码那样格式化Python代码。

最后，请务必注意，Python程序是大小写敏感的，如果写错了大小写，程序会报错。

### 1.数据类型和变量

#### 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，在程序中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，1.23x109和12.3x108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，1.23x109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。如果'本身也是一个字符，那就可以用""括起来，比如"I'm OK"包含的字符是I，'，m，空格，O，K这6个字符。

如果字符串内部既包含'又包含"怎么办？可以用转义字符\来标识，比如：

'I\'m \"OK\"!'

表示的字符串内容是：

I'm "OK"!

转义字符\可以转义很多字符，比如\n表示换行，\t表示制表符，字符\本身也要转义，所以\\表示的字符就是\，可以在Python的交互式命令行用print打印字符串看看：

>>> print 'I\'m ok.'

I'm ok.

>>> print 'I\'m learning\nPython.'

I'm learning

Python.

>>> print '\\\n\\'

\

\

如果字符串里面有很多字符都需要转义，就需要加很多\，为了简化，Python还允许用r''表示''内部的字符串默认不转义，可以自己试试：

>>> print '\\\t\\'

\ \

>>> print r'\\\t\\'

\\\t\\

如果字符串内部有很多换行，用\n写在一行里不好阅读，为了简化，Python允许用'''...'''的格式表示多行内容，可以自己试试：

>>> print '''line1

... line2

... line3'''

line1

line2

line3

上面是在交互式命令行内输入，如果写成程序，就是：

print '''line1

line2

line3'''

多行字符串'''...'''还可以在前面加上r使用，请自行测试。

#### 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：

>>> True

True

>>> False

False

>>> 3 > 2

True

>>> 3 > 5

False

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：

>>> True and True

True

>>> True and False

False

>>> False and False

False

or运算是或运算，只要其中有一个为True，or运算结果就是True：

>>> True or True

True

>>> True or False

True

>>> False or False

False

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：

>>> not True

False

>>> not False

True

布尔值经常用在条件判断中，比如：

if age >= 18:

print 'adult'

else:

print 'teenager'

#### 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。

#### 变量

变量的概念基本上和初中代数的方程变量是一致的，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。

变量在程序中就是用一个变量名表示了，变量名必须是大小写英文、数字和\_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。

Answer = True

变量Answer是一个布尔值True。

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 = 'ABC' # 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 = "ABC"; // 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'

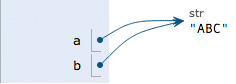
print b

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：

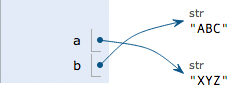
执行a = 'ABC'，解释器创建了字符串'ABC'和变量a，并把a指向'ABC'：

py-var-code-1

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'：



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：



所以，最后打印变量b的结果自然是'ABC'了。

#### 常量

所谓常量就是不能变的变量，比如常用的数学常数π就是一个常量。在Python中，通常用全部大写的变量名表示常量：

PI = 3.14159265359

但事实上PI仍然是一个变量，Python根本没有任何机制保证PI不会被改变，所以，用全部大写的变量名表示常量只是一个习惯上的用法，如果你一定要改变变量PI的值，也没人能拦住你。

最后解释一下整数的除法为什么也是精确的，可以试试：

>>> 10 / 3

3

你没有看错，整数除法永远是整数，即使除不尽。要做精确的除法，只需把其中一个整数换成浮点数做除法就可以：

>>> 10.0 / 3

3.3333333333333335

因为整数除法只取结果的整数部分，所以Python还提供一个余数运算，可以得到两个整数相除的余数：

>>> 10 % 3

1

无论整数做除法还是取余数，结果永远是整数，所以，整数运算结果永远是精确的。

#### 小结

Python支持多种数据类型，在计算机内部，可以把任何数据都看成一个“对象”，而变量就是在程序中用来指向这些数据对象的，对变量赋值就是把数据和变量给关联起来。

### 2.字符串和编码

#### 字符编码

我们已经讲过了，字符串也是一种数据类型，但是，字符串比较特殊的是还有一个编码问题。

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），如果要表示更大的整数，就必须用更多的字节。比如 两个字节可以表示的最大整数是65535，4个字节可以表示的最大整数是4294967295。

由于计算机是美国人发明的，因此，最早只有127个字母被编码到计算机里，也就是大小写英文字母、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。

但是要处理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。

你可以想得到的是，全世界有上百种语言，日本把日文编到Shift\_JIS里，韩国把韩文编到Euc-kr里，各国有各国的标准，就会不可避免地出现冲突，结果就是，在多语言混合的文本中，显示出来会有乱码。



因此，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标准也在不断发展，但最常用的是用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要4个字节）。现代操作系统和大多数编程语言都直接支持Unicode。

现在，捋一捋ASCII编码和Unicode编码的区别：ASCII编码是1个字节，而Unicode编码通常是2个字节。

字母A用ASCII编码是十进制的65，二进制的01000001；

字符0用ASCII编码是十进制的48，二进制的00110000，注意字符'0'和整数0是不同的；

汉字中已经超出了ASCII编码的范围，用Unicode编码是十进制的20013，二进制的01001110 00101101。

你可以猜测，如果把ASCII编码的A用Unicode编码，只需要在前面补0就可以，因此，A的Unicode编码是00000000 01000001。

新的问题又出现了：如果统一成Unicode编码，乱码问题从此消失了。但是，如果你写的文本基本上全部是英文的话，用Unicode编码比ASCII编码需要多一倍的存储空间，在存储和传输上就十分不划算。

所以，本着节约的精神，又出现了把Unicode编码转化为“可变长编码”的UTF-8编码。UTF-8编码把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节。如果你要传输的文本包含大量英文字符，用UTF-8编码就能节省空间：

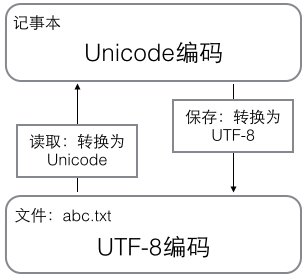
|  |  |  |  |
| --- | --- | --- | --- |
| **字符** | **ASCII** | **Unicode** | **UTF-8** |
| A | 01000001 | 00000000 01000001 | 01000001 |
| 中 | x | 01001110 00101101 | 11100100 10111000 10101101 |

从上面的表格还可以发现，UTF-8编码有一个额外的好处，就是ASCII编码实际上可以被看成是UTF-8编码的一部分，所以，大量只支持ASCII编码的历史遗留软件可以在UTF-8编码下继续工作。

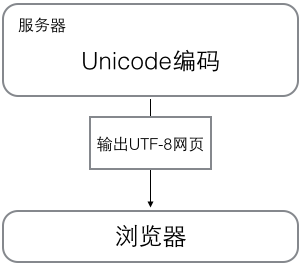
搞清楚了ASCII、Unicode和UTF-8的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

在计算机内存中，统一使用Unicode编码，当需要保存到硬盘或者需要传输的时候，就转换为UTF-8编码。

用记事本编辑的时候，从文件读取的UTF-8字符被转换为Unicode字符到内存里，编辑完成后，保存的时候再把Unicode转换为UTF-8保存到文件：



浏览网页的时候，服务器会把动态生成的Unicode内容转换为UTF-8再传输到浏览器：



所以你看到很多网页的源码上会有类似<meta charset="UTF-8" />的信息，表示该网页正是用的UTF-8编码。

#### Python的字符串

搞清楚了令人头疼的字符编码问题后，我们再来研究Python对Unicode的支持。

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提供了ord()和chr()函数，可以把字母和对应的数字相互转换：

>>> ord('A')

65

>>> chr(65)

'A'

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

>>> u'中'

u'\u4e2d'

写u'中'和u'\u4e2d'是一样的，\u后面是十六进制的Unicode码。因此，u'A'和u'\u0041'也是一样的。

两种字符串如何相互转换？字符串'xxx'虽然是ASCII编码，但也可以看成是UTF-8编码，而u'xxx'则只能是Unicode编码。

把u'xxx'转换为UTF-8编码的'xxx'用encode('utf-8')方法：

>>> u'ABC'.encode('utf-8')

'ABC'

>>> u'中文'.encode('utf-8')

'\xe4\xb8\xad\xe6\x96\x87'

英文字符转换后表示的UTF-8的值和Unicode值相等（但占用的存储空间不同），而中文字符转换后1个Unicode字符将变为3个UTF-8字符，你看到的\xe4就是其中一个字节，因为它的值是228，没有对应的字母可以显示，所以以十六进制显示字节的数值。len()函数可以返回字符串的长度：

>>> len(u'ABC')

3

>>> len('ABC')

3

>>> len(u'中文')

2

>>> len('\xe4\xb8\xad\xe6\x96\x87')

6

反过来，把UTF-8编码表示的字符串'xxx'转换为Unicode字符串u'xxx'用decode('utf-8')方法：

>>> 'abc'.decode('utf-8')

u'abc'

>>> '\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')

u'\u4e2d\u6587'

>>> print '\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')

中文

由于Python源代码也是一个文本文件，所以，当你的源代码中包含中文的时候，在保存源代码时，就需要务必指定保存为UTF-8编码。当Python解释器读取源代码时，为了让它按UTF-8编码读取，我们通常在文件开头写上这两行：

#!/usr/bin/env python

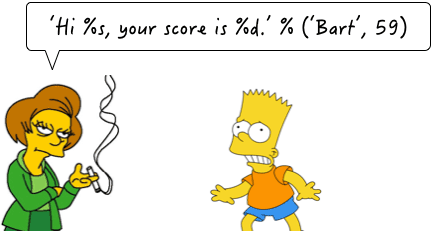
# -\*- coding: utf-8 -\*-

第一行注释是为了告诉Linux/OS X系统，这是一个Python可执行程序，Windows系统会忽略这个注释；

第二行注释是为了告诉Python解释器，按照UTF-8编码读取源代码，否则，你在源代码中写的中文输出可能会有乱码。

#### 格式化

最后一个常见的问题是如何输出格式化的字符串。我们经常会输出类似'亲爱的xxx你好！你xx月的话费是xx，余额是xx'之类的字符串，而xxx的内容都是根据变量变化的，所以，需要一种简便的格式化字符串的方式。



在Python中，采用的格式化方式和C语言是一致的，用%实现，举例如下：

>>> 'Hello, %s' % 'world'

'Hello, world'

>>> 'Hi, %s, you have $%d.' % ('Michael', 1000000)

'Hi, Michael, you have $1000000.'

你可能猜到了，%运算符就是用来格式化字符串的。在字符串内部，%s表示用字符串替换，%d表示用整数替换，有几个%?占位符，后面就跟几个变量或者值，顺序要对应好。如果只有一个%?，括号可以省略。

常见的占位符有：

|  |  |
| --- | --- |
| %d | 整数 |
| %f | 浮点数 |
| %s | 字符串 |
| %x | 十六进制整数 |

其中，格式化整数和浮点数还可以指定是否补0和整数与小数的位数：

>>> '%2d-%02d' % (3, 1)

' 3-01'

>>> '%.2f' % 3.1415926

'3.14'

如果你不太确定应该用什么，%s永远起作用，它会把任何数据类型转换为字符串：

>>> 'Age: %s. Gender: %s' % (25, True)

'Age: 25. Gender: True'

对于Unicode字符串，用法完全一样，但最好确保替换的字符串也是Unicode字符串：

>>> u'Hi, %s' % u'Michael'

u'Hi, Michael'

有些时候，字符串里面的%是一个普通字符怎么办？这个时候就需要转义，用%%来表示一个%：

>>> 'growth rate: %d %%' % 7

'growth rate: 7 %'

#### 小结

由于历史遗留问题，Python 2.x版本虽然支持Unicode，但在语法上需要'xxx'和u'xxx'两种字符串表示方式。

Python当然也支持其他编码方式，比如把Unicode编码成GB2312：

>>> u'中文'.encode('gb2312')

'\xd6\xd0\xce\xc4'

但这种方式纯属自找麻烦，如果没有特殊业务要求，请牢记仅使用Unicode和UTF-8这两种编码方式。

在Python 3.x版本中，把'xxx'和u'xxx'统一成Unicode编码，即写不写前缀u都是一样的，而以字节形式表示的字符串则必须加上b前缀：b'xxx'。

格式化字符串的时候，可以用Python的交互式命令行测试，方便快捷。

### 3.使用list和tuple

#### 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：

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

>>> classmates

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

变量classmates就是一个list。用len()函数可以获得list元素的个数：

>>> len(classmates)

3

用索引来访问list中每一个位置的元素，记得索引是从0开始的：

>>> classmates[0]

'Michael'

>>> classmates[1]

'Bob'

>>> classmates[2]

'Tracy'

>>> classmates[3]

Traceback (most recent call last):

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

IndexError: list index out of range

当索引超出了范围时，Python会报一个IndexError错误，所以，要确保索引不要越界，记得最后一个元素的索引是len(classmates) - 1。

如果要取最后一个元素，除了计算索引位置外，还可以用-1做索引，直接获取最后一个元素：

>>> classmates[-1]

'Tracy'

以此类推，可以获取倒数第2个、倒数第3个：

>>> classmates[-2]

'Bob'

>>> classmates[-3]

'Michael'

>>> classmates[-4]

Traceback (most recent call last):

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

IndexError: list index out of range

当然，倒数第4个就越界了。

list是一个可变的有序表，所以，可以往list中追加元素到末尾：

>>> classmates.append('Adam')

>>> classmates

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

也可以把元素插入到指定的位置，比如索引号为1的位置：

>>> classmates.insert(1, 'Jack')

>>> classmates

['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']

要删除list末尾的元素，用pop()方法：

>>> classmates.pop()

'Adam'

>>> classmates

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

要删除指定位置的元素，用pop(i)方法，其中i是索引位置：

>>> classmates.pop(1)

'Jack'

>>> classmates

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

要把某个元素替换成别的元素，可以直接赋值给对应的索引位置：

>>> classmates[1] = 'Sarah'

>>> classmates

['Michael', 'Sarah', 'Tracy']

list里面的元素的数据类型也可以不同，比如：

>>> L = ['Apple', 123, True]

list元素也可以是另一个list，比如：

>>> s = ['python', 'java', ['asp', 'php'], 'scheme']

>>> len(s)

4

要注意s只有4个元素，其中s[2]又是一个list，如果拆开写就更容易理解了：

>>> p = ['asp', 'php']

>>> s = ['python', 'java', p, 'scheme']

要拿到'php'可以写p[1]或者s[2][1]，因此s可以看成是一个二维数组，类似的还有三维、四维……数组，不过很少用到。

如果一个list中一个元素也没有，就是一个空的list，它的长度为0：

>>> L = []

>>> len(L)

0

#### tuple

另一种有序列表叫元组：tuple。tuple和list非常类似，但是tuple一旦初始化就不能修改，比如同样是列出同学的名字：

>>> classmates = ('Michael', 'Bob', 'Tracy')

现在，classmates这个tuple不能变了，它也没有append()，insert()这样的方法。其他获取元素的方法和list是一样的，你可以正常地使用classmates[0]，classmates[-1]，但不能赋值成另外的元素。

不可变的tuple有什么意义？因为tuple不可变，所以代码更安全。如果可能，能用tuple代替list就尽量用tuple。

tuple的陷阱：当你定义一个tuple时，在定义的时候，tuple的元素就必须被确定下来，比如：

>>> t = (1, 2)

>>> t

(1, 2)

如果要定义一个空的tuple，可以写成()：

>>> t = ()

>>> t

()

但是，要定义一个只有1个元素的tuple，如果你这么定义：

>>> t = (1)

>>> t

1

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

所以，只有1个元素的tuple定义时必须加一个逗号,，来消除歧义：

>>> t = (1,)

>>> t

(1,)

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

最后来看一个“可变的”tuple：

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

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

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

>>> t

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

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

别急，我们先看看定义的时候tuple包含的3个元素：



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



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

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

#### 小结

list和tuple是Python内置的有序集合，一个可变，一个不可变。根据需要来选择使用它们。

### 4.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。

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

age = 20

if age >= 18:

print 'your age is', age

print 'adult'

根据Python的缩进规则，如果if语句判断是True，就把缩进的两行print语句执行了，否则，什么也不做。

也可以给if添加一个else语句，意思是，如果if判断是False，不要执行if的内容，去把else执行了：

age = 3

if age >= 18:

print 'your age is', age

print 'adult'

else:

print 'your age is', age

print 'teenager'

注意不要少写了冒号:。

当然上面的判断是很粗略的，完全可以用elif做更细致的判断：

age = 3

if age >= 18:

print 'adult'

elif age >= 6:

print 'teenager'

else:

print 'kid'

elif是else if的缩写，完全可以有多个elif，所以if语句的完整形式就是：

if <条件判断1>:

<执行1>

elif <条件判断2>:

<执行2>

elif <条件判断3>:

<执行3>

else:

<执行4>

if语句执行有个特点，它是从上往下判断，如果在某个判断上是True，把该判断对应的语句执行后，就忽略掉剩下的elif和else，所以，请测试并解释为什么下面的程序打印的是teenager：

age = 20

if age >= 6:

print 'teenager'

elif age >= 18:

print 'adult'

else:

print 'kid'

if判断条件还可以简写，比如写：

if x:

print 'True'

只要x是非零数值、非空字符串、非空list等，就判断为True，否则为False。

#### 循环

Python的循环有两种，一种是for...in循环，依次把list或tuple中的每个元素迭代出来，看例子：

names = ['Michael', 'Bob', 'Tracy']

for name in names:

print name

执行这段代码，会依次打印names的每一个元素：

Michael

Bob

Tracy

所以for x in ...循环就是把每个元素代入变量x，然后执行缩进块的语句。

再比如我们想计算1-10的整数之和，可以用一个sum变量做累加：

sum = 0

for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:

sum = sum + x

print sum

如果要计算1-100的整数之和，从1写到100有点困难，幸好Python提供一个range()函数，可以生成一个整数序列，比如range(5)生成的序列是从0开始小于5的整数：

>>> range(5)

[0, 1, 2, 3, 4]

range(101)就可以生成0-100的整数序列，计算如下：

sum = 0

for x in range(101):

sum = sum + x

print sum

请自行运行上述代码，看看结果是不是当年高斯同学心算出的5050。

第二种循环是while循环，只要条件满足，就不断循环，条件不满足时退出循环。比如我们要计算100以内所有奇数之和，可以用while循环实现：

sum = 0

n = 99

while n > 0:

sum = sum + n

n = n - 2

print sum

在循环内部变量n不断自减，直到变为-1时，不再满足while条件，循环退出。

#### 再议raw\_input

最后看一个有问题的条件判断。很多同学会用raw\_input()读取用户的输入，这样可以自己输入，程序运行得更有意思：

birth = raw\_input('birth: ')

if birth < 2000:

print '00前'

else:

print '00后'

输入1982，结果却显示00后，这么简单的判断Python也能搞错？

当然不是Python的问题，在Python的交互式命令行下打印birth看看：

>>> birth

'1982'

>>> '1982' < 2000

False

>>> 1982 < 2000

True

原因找到了！原来从raw\_input()读取的内容永远以字符串的形式返回，把字符串和整数比较就不会得到期待的结果，必须先用int()把字符串转换为我们想要的整型：

birth = int(raw\_input('birth: '))

再次运行，就可以得到正确地结果。但是，如果输入abc呢？又会得到一个错误信息：

Traceback (most recent call last):

...

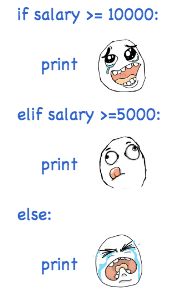
ValueError: invalid literal for int() with base 10: 'abc'

原来int()发现一个字符串并不是合法的数字时就会报错，程序就退出了。

如何检查并捕获程序运行期的错误呢？后面的[错误和调试](http://www.liaoxuefeng.com/wiki/001374738125095c955c1e6d8bb493182103fac9270762a000/001386832254674584228cd773d4db5a02ab85d2db4f83e000)会讲到。

#### 小结

条件判断可以让计算机自己做选择，Python的if...elif...else很灵活。



循环是让计算机做重复任务的有效的方法，有些时候，如果代码写得有问题，会让程序陷入“死循环”，也就是永远循环下去。这时可以用Ctrl+C退出程序，或者强制结束Python进程。

请试写一个死循环程序。

### 5.使用dict和set

Python内置了字典：dict的支持，dict全称dictionary，在其他语言中也称为map，使用键-值（key-value）存储，具有极快的查找速度。

举个例子，假设要根据同学的名字查找对应的成绩，如果用list实现，需要两个list：

names = ['Michael', 'Bob', 'Tracy']

scores = [95, 75, 85]

给定一个名字，要查找对应的成绩，就先要在names中找到对应的位置，再从scores取出对应的成绩，list越长，耗时越长。

如果用dict实现，只需要一个“名字”-“成绩”的对照表，直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。用Python写一个dict如下：

>>> d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}

>>> d['Michael']

95

为什么dict查找速度这么快？因为dict的实现原理和查字典是一样的。假设字典包含了1万个汉字，我们要查某一个字，一个办法是把字典从第一页往后翻，直到找到我们想要的字为止，这种方法就是在list中查找元素的方法，list越大，查找越慢。

第二种方法是先在字典的索引表里（比如部首表）查这个字对应的页码，然后直接翻到该页，找到这个字，无论找哪个字，这种查找速度都非常快，不会随着字典大小的增加而变慢。

dict就是第二种实现方式，给定一个名字，比如'Michael'，dict在内部就可以直接计算出Michael对应的存放成绩的“页码”，也就是95这个数字存放的内存地址，直接取出来，所以速度非常快。

你可以猜到，这种key-value存储方式，在放进去的时候，必须根据key算出value的存放位置，这样，取的时候才能根据key直接拿到value。

把数据放入dict的方法，除了初始化时指定外，还可以通过key放入：

>>> d['Adam'] = 67

>>> d['Adam']

67

由于一个key只能对应一个value，所以，多次对一个key放入value，后面的值会把前面的值冲掉：

>>> d['Jack'] = 90

>>> d['Jack']

90

>>> d['Jack'] = 88

>>> d['Jack']

88

如果key不存在，dict就会报错：

>>> d['Thomas']

Traceback (most recent call last):

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

KeyError: 'Thomas'

要避免key不存在的错误，有两种办法，一是通过in判断key是否存在：

>>> 'Thomas' in d

False

二是通过dict提供的get方法，如果key不存在，可以返回None，或者自己指定的value：

>>> d.get('Thomas')

>>> d.get('Thomas', -1)

-1

注意：返回None的时候Python的交互式命令行不显示结果。

要删除一个key，用pop(key)方法，对应的value也会从dict中删除：

>>> d.pop('Bob')

75

>>> d

{'Michael': 95, 'Tracy': 85}

请务必注意，dict内部存放的顺序和key放入的顺序是没有关系的。

和list比较，dict有以下几个特点：

1. 查找和插入的速度极快，不会随着key的增加而增加；
2. 需要占用大量的内存，内存浪费多。

而list相反：

1. 查找和插入的时间随着元素的增加而增加；
2. 占用空间小，浪费内存很少。

所以，dict是用空间来换取时间的一种方法。

dict可以用在需要高速查找的很多地方，在Python代码中几乎无处不在，正确使用dict非常重要，需要牢记的第一条就是dict的key必须是**不可变对象**。

这是因为dict根据key来计算value的存储位置，如果每次计算相同的key得出的结果不同，那dict内部就完全混乱了。这个通过key计算位置的算法称为哈希算法（Hash）。

要保证hash的正确性，作为key的对象就不能变。在Python中，字符串、整数等都是不可变的，因此，可以放心地作为key。而list是可变的，就不能作为key：

>>> key = [1, 2, 3]

>>> d[key] = 'a list'

Traceback (most recent call last):

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

TypeError: unhashable type: 'list'

#### set

set和dict类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在set中，没有重复的key。

要创建一个set，需要提供一个list作为输入集合：

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

>>> s

set([1, 2, 3])

注意，传入的参数[1, 2, 3]是一个list，而显示的set([1, 2, 3])只是告诉你这个set内部有1，2，3这3个元素，显示的[]不表示这是一个list。

重复元素在set中自动被过滤：

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

>>> s

set([1, 2, 3])

通过add(key)方法可以添加元素到set中，可以重复添加，但不会有效果：

>>> s.add(4)

>>> s

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

>>> s.add(4)

>>> s

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

通过remove(key)方法可以删除元素：

>>> s.remove(4)

>>> s

set([1, 2, 3])

set可以看成数学意义上的无序和无重复元素的集合，因此，两个set可以做数学意义上的交集、并集等操作：

>>> s1 = set([1, 2, 3])

>>> s2 = set([2, 3, 4])

>>> s1 & s2

set([2, 3])

>>> s1 | s2

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

set和dict的唯一区别仅在于没有存储对应的value，但是，set的原理和dict一样，所以，同样不可以放入可变对象，因为无法判断两个可变对象是否相等，也就无法保证set内部“不会有重复元素”。试试把list放入set，看看是否会报错。

#### 再议不可变对象

上面我们讲了，str是不变对象，而list是可变对象。

对于可变对象，比如list，对list进行操作，list内部的内容是会变化的，比如：

>>> a = ['c', 'b', 'a']

>>> a.sort()

>>> a

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

而对于不可变对象，比如str，对str进行操作呢：

>>> a = 'abc'

>>> a.replace('a', 'A')

'Abc'

>>> a

'abc'

虽然字符串有个replace()方法，也确实变出了'Abc'，但变量a最后仍是'abc'，应该怎么理解呢？

我们先把代码改成下面这样：

>>> a = 'abc'

>>> b = a.replace('a', 'A')

>>> b

'Abc'

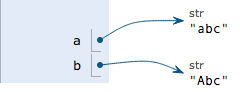
>>> a

'abc'

要始终牢记的是，a是变量，而'abc'才是字符串对象！有些时候，我们经常说，对象a的内容是'abc'，但其实是指，a本身是一个变量，它指向的对象的内容才是'abc'：

a-to-str

当我们调用a.replace('a', 'A')时，实际上调用方法replace是作用在字符串对象'abc'上的，而这个方法虽然名字叫replace，但却没有改变字符串'abc'的内容。相反，replace方法创建了一个新字符串'Abc'并返回，如果我们用变量b指向该新字符串，就容易理解了，变量a仍指向原有的字符串'abc'，但变量b却指向新字符串'Abc'了：



所以，对于不变对象来说，调用对象自身的任意方法，也不会改变该对象自身的内容。相反，这些方法会创建新的对象并返回，这样，就保证了不可变对象本身永远是不可变的。

#### 小结

使用key-value存储结构的dict在Python中非常有用，选择不可变对象作为key很重要，最常用的key是字符串。

tuple虽然是不变对象，但试试把(1, 2, 3)和(1, [2, 3])放入dict或set中，并解释结果。

## 五、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

r2 = 9.08

r3 = 73.1

s1 = 3.14 \* r1 \* r1

s2 = 3.14 \* r2 \* r2

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。

基本上所有的高级语言都支持函数，Python也不例外。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。

#### 抽象

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。

### 1.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，只有一个参数。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 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

#### 数据类型转换

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

>>> float('12.34')

12.34

>>> str(1.23)

'1.23'

>>> unicode(100)

u'100'

>>> bool(1)

True

>>> bool('')

False

函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”：

>>> a = abs # 变量a指向abs函数

>>> a(-1) # 所以也可以通过a调用abs函数

1

#### 小结

调用Python的函数，需要根据函数定义，传入正确的参数。如果函数调用出错，一定要学会看错误信息，所以英文很重要！

### 2.定义函数

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

请自行测试并调用my\_abs看看返回结果是否正确。

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。

#### 空函数

如果想定义一个什么事也不做的空函数，可以用pass语句：

def nop():

pass

pass语句什么都不做，那有什么用？实际上pass可以用来作为占位符，比如现在还没想好怎么写函数的代码，就可以先放一个pass，让代码能运行起来。

pass还可以用在其他语句里，比如：

if age >= 18:

pass

缺少了pass，代码运行就会有语法错误。

#### 参数检查

调用函数时，如果参数个数不对，Python解释器会自动检查出来，并抛出TypeError：

>>> my\_abs(1, 2)

Traceback (most recent call last):

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

TypeError: my\_abs() takes exactly 1 argument (2 given)

但是如果参数类型不对，Python解释器就无法帮我们检查。试试my\_abs和内置函数abs的差别：

>>> my\_abs('A')

'A'

>>> abs('A')

Traceback (most recent call last):

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

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

当传入了不恰当的参数时，内置函数abs会检查出参数错误，而我们定义的my\_abs没有参数检查，所以，这个函数定义不够完善。

让我们修改一下my\_abs的定义，对参数类型做检查，只允许整数和浮点数类型的参数。数据类型检查可以用内置函数isinstance实现：

def my\_abs(x):

if not isinstance(x, (int, float)):

raise TypeError('bad operand type')

if x >= 0:

return x

else:

return -x

添加了参数检查后，如果传入错误的参数类型，函数就可以抛出一个错误：

>>> my\_abs('A')

Traceback (most recent call last):

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

File "<stdin>", line 3, in my\_abs

TypeError: bad operand type

错误和异常处理将在后续讲到。

#### 返回多个值

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，就可以计算出新的新的坐标：

import math

def move(x, y, step, angle=0):

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)

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。

### 3.函数的参数

定义函数的时候，我们把参数的名字和位置确定下来，函数的接口定义就完成了。对于函数的调用者来说，只需要知道如何传递正确的参数，以及函数将返回什么样的值就够了，函数内部的复杂逻辑被封装起来，调用者无需了解。

Python的函数定义非常简单，但灵活度却非常大。除了正常定义的必选参数外，还可以使用默认参数、可变参数和关键字参数，使得函数定义出来的接口，不但能处理复杂的参数，还可以简化调用者的代码。

#### 默认参数

我们仍以具体的例子来说明如何定义函数的默认参数。先写一个计算x2的函数：

def power(x):

return x \* x

当我们调用power函数时，必须传入有且仅有的一个参数x：

>>> power(5)

25

>>> power(15)

225

现在，如果我们要计算x3怎么办？可以再定义一个power3函数，但是如果要计算x4、x5……怎么办？我们不可能定义无限多个函数。

你也许想到了，可以把power(x)修改为power(x, n)，用来计算xn，说干就干：

def power(x, n):

s = 1

while n > 0:

n = n - 1

s = s \* x

return s

对于这个修改后的power函数，可以计算任意n次方：

>>> power(5, 2)

25

>>> power(5, 3)

125

但是，旧的调用代码失败了，原因是我们增加了一个参数，导致旧的代码无法正常调用：

>>> power(5)

Traceback (most recent call last):

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

TypeError: power() takes exactly 2 arguments (1 given)

这个时候，默认参数就排上用场了。由于我们经常计算x2，所以，完全可以把第二个参数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，当我们调用power(5)时，相当于调用power(5, 2)：

>>> power(5)

25

>>> power(5, 2)

25

而对于n > 2的其他情况，就必须明确地传入n，比如power(5, 3)。

从上面的例子可以看出，默认参数可以简化函数的调用。设置默认参数时，有几点要注意：

一是必选参数在前，默认参数在后，否则Python的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；

二是如何设置默认参数。

当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

使用默认参数有什么好处？最大的好处是能降低调用函数的难度。

举个例子，我们写个一年级小学生注册的函数，需要传入name和gender两个参数：

def enroll(name, gender):

print 'name:', name

print 'gender:', gender

这样，调用enroll()函数只需要传入两个参数：

>>> enroll('Sarah', 'F')

name: Sarah

gender: F

如果要继续传入年龄、城市等信息怎么办？这样会使得调用函数的复杂度大大增加。

我们可以把年龄和城市设为默认参数：

def enroll(name, gender, age=6, city='Beijing'):

print 'name:', name

print 'gender:', gender

print 'age:', age

print 'city:', city

这样，大多数学生注册时不需要提供年龄和城市，只提供必须的两个参数：

>>> enroll('Sarah', 'F')

Student:

name: Sarah

gender: F

age: 6

city: Beijing

只有与默认参数不符的学生才需要提供额外的信息：

enroll('Bob', 'M', 7)

enroll('Adam', 'M', city='Tianjin')

可见，默认参数降低了函数调用的难度，而一旦需要更复杂的调用时，又可以传递更多的参数来实现。无论是简单调用还是复杂调用，函数只需要定义一个。

有多个默认参数时，调用的时候，既可以按顺序提供默认参数，比如调用enroll('Bob', 'M', 7)，意思是，除了name，gender这两个参数外，最后1个参数应用在参数age上，city参数由于没有提供，仍然使用默认值。

也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。比如调用enroll('Adam', 'M', city='Tianjin')，意思是，city参数用传进去的值，其他默认参数继续使用默认值。

默认参数很有用，但使用不当，也会掉坑里。默认参数有个最大的坑，演示如下：

先定义一个函数，传入一个list，添加一个END再返回：

def add\_end(L=[]):

L.append('END')

return L

当你正常调用时，结果似乎不错：

>>> add\_end([1, 2, 3])

[1, 2, 3, 'END']

>>> add\_end(['x', 'y', 'z'])

['x', 'y', 'z', 'END']

当你使用默认参数调用时，一开始结果也是对的：

>>> add\_end()

['END']

但是，再次调用add\_end()时，结果就不对了：

>>> add\_end()

['END', 'END']

>>> add\_end()

['END', 'END', 'END']

很多初学者很疑惑，默认参数是[]，但是函数似乎每次都“记住了”上次添加了'END'后的list。

原因解释如下：

Python函数在定义的时候，默认参数L的值就被计算出来了，即[]，因为默认参数L也是一个变量，它指向对象[]，每次调用该函数，如果改变了L的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的[]了。

所以，定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，我们可以用None这个不变对象来实现：

def add\_end(L=None):

if L is None:

L = []

L.append('END')

return L

现在，无论调用多少次，都不会有问题：

>>> add\_end()

['END']

>>> add\_end()

['END']

为什么要设计str、None这样的不变对象呢？因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修 改数据导致的错误。此外，由于对象不变，多任务环境下同时读取对象不需要加锁，同时读一点问题都没有。我们在编写程序时，如果可以设计一个不变对象，那就 尽量设计成不变对象。

#### 可变参数

在Python函数中，还可以定义可变参数。顾名思义，可变参数就是传入的参数个数是可变的，可以是1个、2个到任意个，还可以是0个。

我们以数学题为例子，给定一组数字a，b，c……，请计算a2 + b2 + c2 + ……。

要定义出这个函数，我们必须确定输入的参数。由于参数个数不确定，我们首先想到可以把a，b，c……作为一个list或tuple传进来，这样，函数可以定义如下：

def calc(numbers):

sum = 0

for n in numbers:

sum = sum + n \* n

return sum

但是调用的时候，需要先组装出一个list或tuple：

>>> calc([1, 2, 3])

14

>>> calc((1, 3, 5, 7))

84

如果利用可变参数，调用函数的方式可以简化成这样：

>>> calc(1, 2, 3)

14

>>> calc(1, 3, 5, 7)

84

所以，我们把函数的参数改为可变参数：

def calc(\*numbers):

sum = 0

for n in numbers:

sum = sum + n \* n

return sum

定义可变参数和定义list或tuple参数相比，仅仅在参数前面加了一个\*号。在函数内部，参数numbers接收到的是一个tuple，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括0个参数：

>>> calc(1, 2)

5

>>> calc()

0

如果已经有一个list或者tuple，要调用一个可变参数怎么办？可以这样做：

>>> nums = [1, 2, 3]

>>> calc(nums[0], nums[1], nums[2])

14

这种写法当然是可行的，问题是太繁琐，所以Python允许你在list或tuple前面加一个\*号，把list或tuple的元素变成可变参数传进去：

>>> nums = [1, 2, 3]

>>> calc(\*nums)

14

这种写法相当有用，而且很常见。

#### 关键字参数

可变参数允许你传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple。而关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。请看示例：

def person(name, age, \*\*kw):

print 'name:', name, 'age:', age, 'other:', kw

函数person除了必选参数name和age外，还接受关键字参数kw。在调用该函数时，可以只传入必选参数：

>>> person('Michael', 30)

name: Michael age: 30 other: {}

也可以传入任意个数的关键字参数：

>>> person('Bob', 35, city='Beijing')

name: Bob age: 35 other: {'city': 'Beijing'}

>>> person('Adam', 45, gender='M', job='Engineer')

name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}

关键字参数有什么用？它可以扩展函数的功能。比如，在person函数里，我们保证能接收到name和age这两个参数，但是，如果调用者愿意提供更多的参数，我们也能收到。试想你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。

和可变参数类似，也可以先组装出一个dict，然后，把该dict转换为关键字参数传进去：

>>> kw = {'city': 'Beijing', 'job': 'Engineer'}

>>> person('Jack', 24, city=kw['city'], job=kw['job'])

name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}

当然，上面复杂的调用可以用简化的写法：

>>> kw = {'city': 'Beijing', 'job': 'Engineer'}

>>> person('Jack', 24, \*\*kw)

name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}

#### 参数组合

在Python中定义函数，可以用必选参数、默认参数、可变参数和关键字参数，这4种参数都可以一起使用，或者只用其中某些，但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数和关键字参数。

比如定义一个函数，包含上述4种参数：

def func(a, b, c=0, \*args, \*\*kw):

print 'a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw

在函数调用的时候，Python解释器自动按照参数位置和参数名把对应的参数传进去。

>>> func(1, 2)

a = 1 b = 2 c = 0 args = () kw = {}

>>> func(1, 2, c=3)

a = 1 b = 2 c = 3 args = () kw = {}

>>> func(1, 2, 3, 'a', 'b')

a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}

>>> func(1, 2, 3, 'a', 'b', x=99)

a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}

最神奇的是通过一个tuple和dict，你也可以调用该函数：

>>> args = (1, 2, 3, 4)

>>> kw = {'x': 99}

>>> func(\*args, \*\*kw)

a = 1 b = 2 c = 3 args = (4,) kw = {'x': 99}

所以，对于任意函数，都可以通过类似func(\*args, \*\*kw)的形式调用它，无论它的参数是如何定义的。

#### 小结

Python的函数具有非常灵活的参数形态，既可以实现简单的调用，又可以传入非常复杂的参数。

默认参数一定要用不可变对象，如果是可变对象，运行会有逻辑错误！

要注意定义可变参数和关键字参数的语法：

\*args是可变参数，args接收的是一个tuple；

\*\*kw是关键字参数，kw接收的是一个dict。

以及调用函数时如何传入可变参数和关键字参数的语法：

可变参数既可以直接传入：func(1, 2, 3)，又可以先组装list或tuple，再通过\*args传入：func(\*(1, 2, 3))；

关键字参数既可以直接传入：func(a=1, b=2)，又可以先组装dict，再通过\*\*kw传入：func(\*\*{'a': 1, 'b': 2})。

使用\*args和\*\*kw是Python的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。

### 4.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 x 2 x 3 x ... x n，用函数fact(n)表示，可以看出：

fact(n) = n! = 1 x 2 x 3 x ... x (n-1) x n = (n-1)! x n = fact(n-1) x n

所以，fact(n)可以表示为n x 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)

93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000L

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fact(1000)：

>>> fact(1000)

Traceback (most recent call last):

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

File "<stdin>", line 4, in fact

...

File "<stdin>", line 4, in fact

RuntimeError: maximum recursion depth exceeded

解决递归调用栈溢出的方法是通过**尾递归**优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。

尾递归是指，在函数返回的时候，调用自身本身，并且，return语句不能包含表达式。这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。

上面的fact(n)函数由于return n \* fact(n - 1)引入了乘法表达式，所以就不是尾递归了。要改成尾递归方式，需要多一点代码，主要是要把每一步的乘积传入到递归函数中：

def fact(n):

return fact\_iter(1, 1, n)

def fact\_iter(product, count, max):

if count > max:

return product

return fact\_iter(product \* count, count + 1, max)

可以看到，return fact\_iter(product \* count, count + 1, max)仅返回递归函数本身，product \* count和count + 1在函数调用前就会被计算，不影响函数调用。

fact(5)对应的fact\_iter(1, 1, 5)的调用如下：

===> fact\_iter(1, 1, 5)

===> fact\_iter(1, 2, 5)

===> fact\_iter(2, 3, 5)

===> fact\_iter(6, 4, 5)

===> fact\_iter(24, 5, 5)

===> fact\_iter(120, 6, 5)

===> 120

尾递归调用时，如果做了优化，栈不会增长，因此，无论多少次调用也不会导致栈溢出。

遗憾的是，大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化，所以，即使把上面的fact(n)函数改成尾递归方式，也会导致栈溢出。

有一个针对尾递归优化的decorator，可以参考源码：

<http://code.activestate.com/recipes/474088-tail-call-optimization-decorator/>

我们后面会讲到如何编写decorator。现在，只需要使用这个@tail\_call\_optimized，就可以顺利计算出fact(1000)：

>>> fact(1000)

402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208486969404800479988610197196058631666872994808558901323829669944590997424504087073759918823627727188732519779505950995276120874975462497043601418278094646496291056393887437886487337119181045825783647849977012476632889835955735432513185323958463075557409114262417474349347553428646576611667797396668820291207379143853719588249808126867838374559731746136085379534524221586593201928090878297308431392844403281231558611036976801357304216168747609675871348312025478589320767169132448426236131412508780208000261683151027341827977704784635868170164365024153691398281264810213092761244896359928705114964975419909342221566832572080821333186116811553615836546984046708975602900950537616475847728421889679646244945160765353408198901385442487984959953319101723355556602139450399736280750137837615307127761926849034352625200015888535147331611702103968175921510907788019393178114194545257223865541461062892187960223838971476088506276862967146674697562911234082439208160153780889893964518263243671616762179168909779911903754031274622289988005195444414282012187361745992642956581746628302955570299024324153181617210465832036786906117260158783520751516284225540265170483304226143974286933061690897968482590125458327168226458066526769958652682272807075781391858178889652208164348344825993266043367660176999612831860788386150279465955131156552036093988180612138558600301435694527224206344631797460594682573103790084024432438465657245014402821885252470935190620929023136493273497565513958720559654228749774011413346962715422845862377387538230483865688976461927383814900140767310446640259899490222221765904339901886018566526485061799702356193897017860040811889729918311021171229845901641921068884387121855646124960798722908519296819372388642614839657382291123125024186649353143970137428531926649875337218940694281434118520158014123344828015051399694290153483077644569099073152433278288269864602789864321139083506217095002597389863554277196742822248757586765752344220207573630569498825087968928162753848863396909959826280956121450994871701244516461260379029309120889086942028510640182154399457156805941872748998094254742173582401063677404595741785160829230135358081840096996372524230560855903700624271243416909004153690105933983835777939410970027753472000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

#### 小结

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。

针对尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的，没有循环语句的编程语言只能通过尾递归实现循环。

Python标准的解释器没有针对尾递归做优化，任何递归函数都存在栈溢出的问题。

## 六、高级特性

掌握了Python的数据类型、语句和函数，基本上就可以编写出很多有用的程序了。

比如构造一个1, 3, 5, 7, ..., 99的列表，可以通过循环实现：

L = []

n = 1

while n <= 99:

L.append(n)

n = n + 2

[Try](http://www.liaoxuefeng.com/wiki/001374738125095c955c1e6d8bb493182103fac9270762a000/0013868196169906eb9ca5864384546bf3405ae6a172b3e000#0)

取list的前一半的元素，也可以通过循环实现。

但是在Python中，代码不是越多越好，而是越少越好。代码不是越复杂越好，而是越简单越好。

基于这一思想，我们来介绍Python中非常有用的高级特性，一行代码能实现的功能，决不写5行代码。

### 1.切片

取一个list或tuple的部分元素是非常常见的操作。比如，一个list如下：

>>> L = ['Michael', 'Sarah', 'Tracy', 'Bob', 'Jack']

[Try](http://www.liaoxuefeng.com/wiki/001374738125095c955c1e6d8bb493182103fac9270762a000/0013868196352269f28f1f00aee485ea27e3c4e47f12bc7000#0)

取前3个元素，应该怎么做？

笨办法：

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

['Michael', 'Sarah', 'Tracy']

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

['Michael', 'Sarah', 'Tracy']

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]

['Michael', 'Sarah', 'Tracy']

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]

['Michael', 'Sarah', 'Tracy']

也可以从索引1开始，取出2个元素出来：

>>> L[1:3]

['Sarah', 'Tracy']

类似的，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[-2:]

['Bob', 'Jack']

>>> L[-2:-1]

['Bob']

记住倒数第一个元素的索引是-1。

切片操作十分有用。我们先创建一个0-99的数列：

>>> L = range(100)

>>> L

[0, 1, 2, 3, ..., 99]

可以通过切片轻松取出某一段数列。比如前10个数：

>>> L[:10]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

后10个数：

>>> L[-10:]

[90, 91, 92, 93, 94, 95, 96, 97, 98, 99]

前11-20个数：

>>> L[10:20]

[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

前10个数，每两个取一个：

>>> L[:10:2]

[0, 2, 4, 6, 8]

所有数，每5个取一个：

>>> L[::5]

[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]

甚至什么都不写，只写[:]就可以原样复制一个list：

>>> L[:]

[0, 1, 2, 3, ..., 99]

tuple也是一种list，唯一区别是tuple不可变。因此，tuple也可以用切片操作，只是操作的结果仍是tuple：

>>> (0, 1, 2, 3, 4, 5)[:3]

(0, 1, 2)

字符串'xxx'或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'[::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。

#### 小结

有了切片操作，很多地方循环就不再需要了。Python的切片非常灵活，一行代码就可以实现很多行循环才能完成的操作。

### 2.迭代

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，还可以作用在其他可迭代对象上。

list这种数据类型虽然有下标，但很多其他数据类型是没有下标的，但是，只要是可迭代对象，无论有无下标，都可以迭代，比如dict就可以迭代：

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

>>> for key in d:

... print key

...

a

c

b

因为dict的存储不是按照list的方式顺序排列，所以，迭代出的结果顺序很可能不一样。

默认情况下，dict迭代的是key。如果要迭代value，可以用for value in d.itervalues()，如果要同时迭代key和value，可以用for k, v in d.iteritems()。

由于字符串也是可迭代对象，因此，也可以作用于for循环：

>>> for ch in 'ABC':

... print ch

...

A

B

C

所以，当我们使用for循环时，只要作用于一个可迭代对象，for循环就可以正常运行，而我们不太关心该对象究竟是list还是其他数据类型。

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

>>> from collections import Iterable

>>> isinstance('abc', Iterable) # str是否可迭代

True

>>> isinstance([1,2,3], Iterable) # list是否可迭代

True

>>> isinstance(123, Iterable) # 整数是否可迭代

False

最后一个小问题，如果要对list实现类似Java那样的下标循环怎么办？Python内置的enumerate函数可以把一个list变成索引-元素对，这样就可以在for循环中同时迭代索引和元素本身：

>>> for i, value in enumerate(['A', 'B', 'C']):

... print i, value

...

0 A

1 B

2 C

上面的for循环里，同时引用了两个变量，在Python里是很常见的，比如下面的代码：

>>> for x, y in [(1, 1), (2, 4), (3, 9)]:

... print x, y

...

1 1

2 4

3 9

#### 小结

任何可迭代对象都可以作用于for循环，包括我们自定义的数据类型，只要符合迭代条件，就可以使用for循环。

### 3.列表生成式

列表生成式即List Comprehensions，是Python内置的非常简单却强大的可以用来创建list的生成式。

举个例子，要生成list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]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]

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。

for循环后面还可以加上if判断，这样我们就可以筛选出仅偶数的平方：

>>> [x \* x for x in range(1, 11) if x % 2 == 0]

[4, 16, 36, 64, 100]

还可以使用两层循环，可以生成全排列：

>>> [m + n for m in 'ABC' for n in 'XYZ']

['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']

三层和三层以上的循环就很少用到了。

运用列表生成式，可以写出非常简洁的代码。例如，列出当前目录下的所有文件和目录名，可以通过一行代码实现：

>>> import os # 导入os模块，模块的概念后面讲到

>>> [d for d in os.listdir('.')] # os.listdir可以列出文件和目录

['.emacs.d', '.ssh', '.Trash', 'Adlm', 'Applications', 'Desktop', 'Documents', 'Downloads', 'Library', 'Movies', 'Music', 'Pictures', 'Public', 'VirtualBox VMs', 'Workspace', 'XCode']

for循环其实可以同时使用两个甚至多个变量，比如dict的iteritems()可以同时迭代key和value：

>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }

>>> for k, v in d.iteritems():

... print k, '=', v

...

y = B

x = A

z = C

因此，列表生成式也可以使用两个变量来生成list：

>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }

>>> [k + '=' + v for k, v in d.iteritems()]

['y=B', 'x=A', 'z=C']

最后把一个list中所有的字符串变成小写：

>>> L = ['Hello', 'World', 'IBM', 'Apple']

>>> [s.lower() for s in L]

['hello', 'world', 'ibm', 'apple']

#### 小结

运用列表生成式，可以快速生成list，可以通过一个list推导出另一个list，而代码却十分简洁。

思考：如果list中既包含字符串，又包含整数，由于非字符串类型没有lower()方法，所以列表生成式会报错：

>>> L = ['Hello', 'World', 18, 'Apple', None]

>>> [s.lower() for s in L]

Traceback (most recent call last):

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

AttributeError: 'int' object has no attribute 'lower'

使用内建的isinstance函数可以判断一个变量是不是字符串：

>>> x = 'abc'

>>> y = 123

>>> isinstance(x, str)

True

>>> isinstance(y, str)

False

请修改列表生成式，通过添加if语句保证列表生成式能正确地执行。

### 4.生成器

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

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

要创建一个generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的[]改成()，就创建了一个generator：

>>> L = [x \* x for x in range(10)]

>>> L

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> g = (x \* x for x in range(10))

>>> g

<generator object <genexpr> at 0x104feab40>

创建L和g的区别仅在于最外层的[]和()，L是一个list，而g是一个generator。

我们可以直接打印出list的每一个元素，但我们怎么打印出generator的每一个元素呢？

如果要一个一个打印出来，可以通过generator的next()方法：

>>> g.next()

0

>>> g.next()

1

>>> g.next()

4

>>> g.next()

9

>>> g.next()

16

>>> g.next()

25

>>> g.next()

36

>>> g.next()

49

>>> g.next()

64

>>> g.next()

81

>>> g.next()

Traceback (most recent call last):

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

StopIteration

我们讲过，generator保存的是算法，每次调用next()，就计算出下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration的错误。

当然，上面这种不断调用next()方法实在是太变态了，正确的方法是使用for循环，因为generator也是可迭代对象：

>>> g = (x \* x for x in range(10))

>>> for n in g:

... print n

...

0

1

4

9

16

25

36

49

64

81

所以，我们创建了一个generator后，基本上永远不会调用next()方法，而是通过for循环来迭代它。

generator非常强大。如果推算的算法比较复杂，用类似列表生成式的for循环无法实现的时候，还可以用函数来实现。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

def fib(max):

n, a, b = 0, 0, 1

while n < max:

print b

a, b = b, a + b

n = n + 1

上面的函数可以输出斐波那契数列的前N个数：

>>> fib(6)

1

1

2

3

5

8

仔细观察，可以看出，fib函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

也就是说，上面的函数和generator仅一步之遥。要把fib函数变成generator，只需要把print b改为yield b就可以了：

def fib(max):

n, a, b = 0, 0, 1

while n < max:

yield b

a, b = b, a + b

n = n + 1

这就是定义generator的另一种方法。如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator：

>>> fib(6)

<generator object fib at 0x104feaaa0>

这里，最难理解的就是generator和函数的执行流程不一样。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。

举个简单的例子，定义一个generator，依次返回数字1，3，5：

>>> def odd():

... print 'step 1'

... yield 1

... print 'step 2'

... yield 3

... print 'step 3'

... yield 5

...

>>> o = odd()

>>> o.next()

step 1

1

>>> o.next()

step 2

3

>>> o.next()

step 3

5

>>> o.next()

Traceback (most recent call last):

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

StopIteration

可以看到，odd不是普通函数，而是generator，在执行过程中，遇到yield就中断，下次又继续执行。执行3次yield后，已经没有yield可以执行了，所以，第4次调用next()就报错。

回到fib的例子，我们在循环过程中不断调用yield，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator后，我们基本上从来不会用next()来调用它，而是直接使用for循环来迭代：

>>> for n in fib(6):

... print n

...

1

1

2

3

5

8

#### 小结

generator是非常强大的工具，在Python中，可以简单地把列表生成式改成generator，也可以通过函数实现复杂逻辑的generator。

要理解generator的工作原理，它是在for循环的过程中不断计算出下一个元素，并在适当的条件结束for循环。对于函数改成的generator来说，遇到return语句或者执行到函数体最后一行语句，就是结束generator的指令，for循环随之结束。

## 七、函数式编程

函数是Python内建支持的一种封装，我们通过把大段代码拆成函数，通过一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称之为面向过程的程序设计。函数就是面向过程的程序设计的基本单元。

而函数式编程（请注意多了一个“式”字）——Functional Programming，虽然也可以归结到面向过程的程序设计，但其思想更接近数学计算。

我们首先要搞明白计算机（Computer）和计算（Compute）的概念。

在计算机的层次上，CPU执行的是加减乘除的指令代码，以及各种条件判断和跳转指令，所以，汇编语言是最贴近计算机的语言。

而计算则指数学意义上的计算，越是抽象的计算，离计算机硬件越远。

对应到编程语言，就是越低级的语言，越贴近计算机，抽象程度低，执行效率高，比如C语言；越高级的语言，越贴近计算，抽象程度高，执行效率低，比如Lisp语言。

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这 种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作 用的。

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

Python对函数式编程提供部分支持。由于Python允许使用变量，因此，Python不是纯函数式编程语言。

### 1.高阶函数

高阶函数英文叫Higher-order function。什么是高阶函数？我们以实际代码为例子，一步一步深入概念。

##### 变量可以指向函数

以Python内置的求绝对值的函数abs()为例，调用该函数用以下代码：

>>> abs(-10)

10

但是，如果只写abs呢？

>>> abs

<built-in function abs>

可见，abs(-10)是函数调用，而abs是函数本身。

要获得函数调用结果，我们可以把结果赋值给变量：

>>> x = abs(-10)

>>> x

10

但是，如果把函数本身赋值给变量呢？

>>> f = abs

>>> f

<built-in function abs>

结论：函数本身也可以赋值给变量，即：变量可以指向函数。

如果一个变量指向了一个函数，那么，可否通过该变量来调用这个函数？用代码验证一下：

>>> f = abs

>>> f(-10)

10

成功！说明变量f现在已经指向了abs函数本身。

##### 函数名也是变量

那么函数名是什么呢？函数名其实就是指向函数的变量！对于abs()这个函数，完全可以把函数名abs看成变量，它指向一个可以计算绝对值的函数！

如果把abs指向其他对象，会有什么情况发生？

>>> abs = 10

>>> abs(-10)

Traceback (most recent call last):

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

TypeError: 'int' object is not callable

把abs指向10后，就无法通过abs(-10)调用该函数了！因为abs这个变量已经不指向求绝对值函数了！

当然实际代码绝对不能这么写，这里是为了说明函数名也是变量。要恢复abs函数，请重启Python交互环境。

注：由于abs函数实际上是定义在\_\_builtin\_\_模块中的，所以要让修改abs变量的指向在其它模块也生效，要用\_\_builtin\_\_.abs = 10。

##### 传入函数

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

def add(x, y, f):

return f(x) + f(y)

当我们调用add(-5, 6, abs)时，参数x，y和f分别接收-5，6和abs，根据函数定义，我们可以推导计算过程为：

x ==> -5

y ==> 6

f ==> abs

f(x) + f(y) ==> abs(-5) + abs(6) ==> 11

用代码验证一下：

>>> add(-5, 6, abs)

11

编写高阶函数，就是让函数的参数能够接收别的函数。

##### 小结

把函数作为参数传入，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。

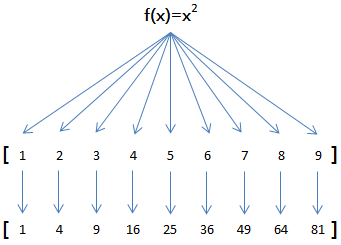
#### ①。map/reduce

Python内建了map()和reduce()函数。

如果你读过Google的那篇大名鼎鼎的论文“[MapReduce: Simplified Data Processing on Large Clusters](http://research.google.com/archive/mapreduce.html)”，你就能大概明白map/reduce的概念。

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

举例说明，比如我们有一个函数f(x)=x2，要把这个函数作用在一个list [1, 2, 3, 4, 5, 6, 7, 8, 9]上，就可以用map()实现如下：



现在，我们用Python代码实现：

>>> def f(x):

... return x \* x

...

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

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

map()传入的第一个参数是f，即函数对象本身。

你可能会想，不需要map()函数，写一个循环，也可以计算出结果：

L = []

for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:

L.append(f(n))

print L

的确可以，但是，从上面的循环代码，能一眼看明白“把f(x)作用在list的每一个元素并把结果生成一个新的list”吗？

所以，map()作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的f(x)=x2，还可以计算任意复杂的函数，比如，把这个list所有数字转为字符串：

>>> map(str, [1, 2, 3, 4, 5, 6, 7, 8, 9])

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

只需要一行代码。

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

reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)

比方说对一个序列求和，就可以用reduce实现：

>>> def add(x, y):

... return x + y

...

>>> reduce(add, [1, 3, 5, 7, 9])

25

当然求和运算可以直接用Python内建函数sum()，没必要动用reduce。

但是如果要把序列[1, 3, 5, 7, 9]变换成整数13579，reduce就可以派上用场：

>>> def fn(x, y):

... return x \* 10 + y

...

>>> reduce(fn, [1, 3, 5, 7, 9])

13579

这个例子本身没多大用处，但是，如果考虑到字符串str也是一个序列，对上面的例子稍加改动，配合map()，我们就可以写出把str转换为int的函数：

>>> def fn(x, y):

... return x \* 10 + y

...

>>> def char2num(s):

... return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}[s]

...

>>> reduce(fn, map(char2num, '13579'))

13579

整理成一个str2int的函数就是：

def str2int(s):

def fn(x, y):

return x \* 10 + y

def char2num(s):

return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}[s]

return reduce(fn, map(char2num, s))

还可以用lambda函数进一步简化成：

def char2num(s):

return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}[s]

def str2int(s):

return reduce(lambda x,y: x\*10+y, map(char2num, s))

也就是说，假设Python没有提供int()函数，你完全可以自己写一个把字符串转化为整数的函数，而且只需要几行代码！

lambda函数的用法在后面介绍。

##### 练习

利用map()函数，把用户输入的不规范的英文名字，变为首字母大写，其他小写的规范名字。输入：['adam', 'LISA', 'barT']，输出：['Adam', 'Lisa', 'Bart']。

Python提供的sum()函数可以接受一个list并求和，请编写一个prod()函数，可以接受一个list并利用reduce()求积。

#### ②。filter

Python内建的filter()函数用于过滤序列。

和map()类似，filter()也接收一个函数和一个序列。和map()不同的时，filter()把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。

例如，在一个list中，删掉偶数，只保留奇数，可以这么写：

def is\_odd(n):

return n % 2 == 1

filter(is\_odd, [1, 2, 4, 5, 6, 9, 10, 15])

# 结果: [1, 5, 9, 15]

把一个序列中的空字符串删掉，可以这么写：

def not\_empty(s):

return s and s.strip()

filter(not\_empty, ['A', '', 'B', None, 'C', ' '])

# 结果: ['A', 'B', 'C']

可见用filter()这个高阶函数，关键在于正确实现一个“筛选”函数。

##### 练习

请尝试用filter()删除1~100的素数。

#### ③。sorted

##### 排序算法

排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，我们可以直接比较，但如果是字符串或者两个dict呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。通常规定，对于两个元素x和y，如果认为x < y，则返回-1，如果认为x == y，则返回0，如果认为x > y，则返回1，这样，排序算法就不用关心具体的比较过程，而是根据比较结果直接排序。

Python内置的sorted()函数就可以对list进行排序：

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

[5, 9, 12, 21, 36]

此外，sorted()函数也是一个高阶函数，它还可以接收一个比较函数来实现自定义的排序。比如，如果要倒序排序，我们就可以自定义一个reversed\_cmp函数：

def reversed\_cmp(x, y):

if x > y:

return -1

if x < y:

return 1

return 0

传入自定义的比较函数reversed\_cmp，就可以实现倒序排序：

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

[36, 21, 12, 9, 5]

我们再看一个字符串排序的例子：

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

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

默认情况下，对字符串排序，是按照ASCII的大小比较的，由于'Z' < 'a'，结果，大写字母Z会排在小写字母a的前面。

现在，我们提出排序应该忽略大小写，按照字母序排序。要实现这个算法，不必对现有代码大加改动，只要我们能定义出忽略大小写的比较算法就可以：

def cmp\_ignore\_case(s1, s2):

u1 = s1.upper()

u2 = s2.upper()

if u1 < u2:

return -1

if u1 > u2:

return 1

return 0

忽略大小写来比较两个字符串，实际上就是先把字符串都变成大写（或者都变成小写），再比较。

这样，我们给sorted传入上述比较函数，即可实现忽略大小写的排序：

>>> sorted(['bob', 'about', 'Zoo', 'Credit'], cmp\_ignore\_case)

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

从上述例子可以看出，高阶函数的抽象能力是非常强大的，而且，核心代码可以保持得非常简洁。

### 2.返回函数

#### 函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

def calc\_sum(\*args):

ax = 0

for n in args:

ax = ax + n

return ax

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数！

def lazy\_sum(\*args):

def sum():

ax = 0

for n in args:

ax = ax + n

return ax

return sum

当我们调用lazy\_sum()时，返回的并不是求和结果，而是求和函数：

>>> f = lazy\_sum(1, 3, 5, 7, 9)

>>> f

<function sum at 0x10452f668>

调用函数f时，才真正计算求和的结果：

>>> f()

25

在这个例子中，我们在函数lazy\_sum中又定义了函数sum，并且，内部函数sum可以引用外部函数lazy\_sum的参数和局部变量，当lazy\_sum返回函数sum时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。

请再注意一点，当我们调用lazy\_sum()时，每次调用都会返回一个新的函数，即使传入相同的参数：

>>> f1 = lazy\_sum(1, 3, 5, 7, 9)

>>> f2 = lazy\_sum(1, 3, 5, 7, 9)

>>> f1==f2

False

f1()和f2()的调用结果互不影响。

#### 闭包

注意到返回的函数在其定义内部引用了局部变量args，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。

另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了f()才执行。我们来看一个例子：

def count():

fs = []

for i in range(1, 4):

def f():

return i\*i

fs.append(f)

return fs

f1, f2, f3 = count()

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都返回了。

你可能认为调用f1()，f2()和f3()结果应该是1，4，9，但实际结果是：

>>> f1()

9

>>> f2()

9

>>> f3()

9

全部都是9！原因就在于返回的函数引用了变量i，但它并非立刻执行。等到3个函数都返回时，它们所引用的变量i已经变成了3，因此最终结果为9。

返回闭包时牢记的一点就是：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。

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

>>> def count():

... fs = []

... for i in range(1, 4):

... def f(j):

... def g():

... return j\*j

... return g

... fs.append(f(i))

... return fs

...

>>> f1, f2, f3 = count()

>>> f1()

1

>>> f2()

4

>>> f3()

9

缺点是代码较长，可利用lambda函数缩短代码。

### 3.匿名函数

当我们在传入函数时，有些时候，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，不用写return，返回值就是该表达式的结果。

用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数：

>>> f = lambda x: x \* x

>>> f

<function <lambda> at 0x10453d7d0>

>>> f(5)

25

同样，也可以把匿名函数作为返回值返回，比如：

def build(x, y):

return lambda: x \* x + y \* y

#### 小结

Python对匿名函数的支持有限，只有一些简单的情况下可以使用匿名函数。

### 4.装饰器

由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。

>>> def now():

... print '2013-12-25'

...

>>> f = now

>>> f()

2013-12-25

[Try](http://www.liaoxuefeng.com/wiki/001374738125095c955c1e6d8bb493182103fac9270762a000/001386819879946007bbf6ad052463ab18034f0254bf355000#0)

函数对象有一个\_\_name\_\_属性，可以拿到函数的名字：

>>> now.\_\_name\_\_

'now'

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

'now'

现在，假设我们要增强now()函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改now()函数的定义，这种在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。

本质上，decorator就是一个返回函数的高阶函数。所以，我们要定义一个能打印日志的decorator，可以定义如下：

def log(func):

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

print 'call %s():' % func.\_\_name\_\_

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

return wrapper

观察上面的log，因为它是一个decorator，所以接受一个函数作为参数，并返回一个函数。我们要借助Python的@语法，把decorator置于函数的定义处：

@log

def now():

print '2013-12-25'

调用now()函数，不仅会运行now()函数本身，还会在运行now()函数前打印一行日志：

>>> now()

call now():

2013-12-25

把@log放到now()函数的定义处，相当于执行了语句：

now = log(now)

由于log()是一个decorator，返回一个函数，所以，原来的now()函数仍然存在，只是现在同名的now变量指向了新的函数，于是调用now()将执行新函数，即在log()函数中返回的wrapper()函数。

wrapper()函数的参数定义是(\*args, \*\*kw)，因此，wrapper()函数可以接受任意参数的调用。在wrapper()函数内，首先打印日志，再紧接着调用原始函数。

如果decorator本身需要传入参数，那就需要编写一个返回decorator的高阶函数，写出来会更复杂。比如，要自定义log的文本：

def log(text):

def decorator(func):

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

print '%s %s():' % (text, func.\_\_name\_\_)

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

return wrapper

return decorator

这个3层嵌套的decorator用法如下：

@log('execute')

def now():

print '2013-12-25'

执行结果如下：

>>> now()

execute now():

2013-12-25

和两层嵌套的decorator相比，3层嵌套的效果是这样的：

>>> now = log('execute')(now)

我们来剖析上面的语句，首先执行log('execute')，返回的是decorator函数，再调用返回的函数，参数是now函数，返回值最终是wrapper函数。

以上两种decorator的定义都没有问题，但还差最后一步。因为我们讲了函数也是对象，它有\_\_name\_\_等属性，但你去看经过decorator装饰之后的函数，它们的\_\_name\_\_已经从原来的'now'变成了'wrapper'：

>>> now.\_\_name\_\_

'wrapper'

因为返回的那个wrapper()函数名字就是'wrapper'，所以，需要把原始函数的\_\_name\_\_等属性复制到wrapper()函数中，否则，有些依赖函数签名的代码执行就会出错。

不需要编写wrapper.\_\_name\_\_ = func.\_\_name\_\_这样的代码，Python内置的functools.wraps就是干这个事的，所以，一个完整的decorator的写法如下：

import functools

def log(func):

@functools.wraps(func)

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

print 'call %s():' % func.\_\_name\_\_

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

return wrapper

或者针对带参数的decorator：

import functools

def log(text):

def decorator(func):

@functools.wraps(func)

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

print '%s %s():' % (text, func.\_\_name\_\_)

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

return wrapper

return decorator

import functools是导入functools模块。模块的概念稍候讲解。现在，只需记住在定义wrapper()的前面加上@functools.wraps(func)即可。

#### 小结

在面向对象（OOP）的设计模式中，decorator被称为装饰模式。OOP的装饰模式需要通过继承和组合来实现，而Python除了能支持 OOP的decorator外，直接从语法层次支持decorator。Python的decorator可以用函数实现，也可以用类实现。

decorator可以增强函数的功能，定义起来虽然有点复杂，但使用起来非常灵活和方便。

请编写一个decorator，能在函数调用的前后打印出'begin call'和'end call'的日志。

再思考一下能否写出一个@log的decorator，使它既支持：

@log

def f():

pass

又支持：

@log('execute')

def f():

pass

### 5.偏函数

Python的functools模块提供了很多有用的功能，其中一个就是偏函数（Partial function）。要注意，这里的偏函数和数学意义上的偏函数不一样。

在介绍函数参数的时候，我们讲到，通过设定参数的默认值，可以降低函数调用的难度。而偏函数也可以做到这一点。举例如下：

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，简单总结functools.partial的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

注意到上面的新的int2函数，仅仅是把base参数重新设定默认值为2，但也可以在函数调用时传入其他值：

>>> int2('1000000', base=10)

1000000

最后，创建偏函数时，实际上可以接收函数对象、\*args和\*\*kw这3个参数，当传入：

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

实际上固定了int()函数的关键字参数base，也就是：

int2('10010')

相当于：

kw = { base: 2 }

int('10010', \*\*kw)

当传入：

max2 = functools.partial(max, 10)

实际上会把10作为\*args的一部分自动加到左边，也就是：

max2(5, 6, 7)

相当于：

args = (10, 5, 6, 7)

max(\*args)

结果为10。

#### 小结

当函数的参数个数太多，需要简化时，使用functools.partial可以创建一个新的函数，这个新函数可以固定住原函数的部分参数，从而在调用时更简单。

## 八、模块

在计算机程序的开发过程中，随着程序代码越写越多，在一个文件里代码就会越来越长，越来越不容易维护。

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。在Python中，一个.py文件就称之为一个模块（Module）。

使用模块有什么好处？

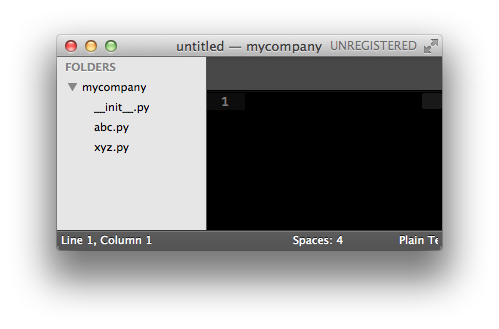
最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括Python内置的模块和来自第三方的模块。

使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突。点[这里](http://docs.python.org/2/library/functions.html)查看Python的所有内置函数。

你也许还想到，如果不同的人编写的模块名相同怎么办？为了避免模块名冲突，Python又引入了按目录来组织模块的方法，称为包（Package）。

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

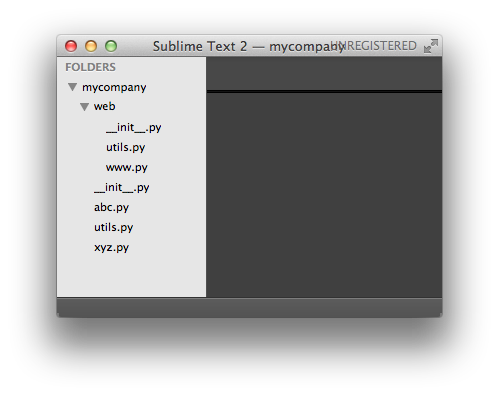
现在，假设我们的abc和xyz这两个模块名字与其他模块冲突了，于是我们可以通过包来组织模块，避免冲突。方法是选择一个顶层包名，比如mycompany，按照如下目录存放：



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

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

类似的，可以有多级目录，组成多级层次的包结构。比如如下的目录结构：



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

mycompany.web也是一个模块，请指出该模块对应的.py文件。

### 1.使用模块

Python本身就内置了很多非常有用的模块，只要安装完毕，这些模块就可以立刻使用。

我们以内建的sys模块为例，编写一个hello的模块：

#!/usr/bin/env python

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

' a test module '

\_\_author\_\_ = 'Michael Liao'

import sys

def test():

args = sys.argv

if len(args)==1:

print 'Hello, world!'

elif len(args)==2:

print 'Hello, %s!' % args[1]

else:

print 'Too many arguments!'

if \_\_name\_\_=='\_\_main\_\_':

test()

第1行和第2行是标准注释，第1行注释可以让这个hello.py文件直接在Unix/Linux/Mac上运行，第2行注释表示.py文件本身使用标准UTF-8编码；

第4行是一个字符串，表示模块的文档注释，任何模块代码的第一个字符串都被视为模块的文档注释；

第6行使用\_\_author\_\_变量把作者写进去，这样当你公开源代码后别人就可以瞻仰你的大名；

以上就是Python模块的标准文件模板，当然也可以全部删掉不写，但是，按标准办事肯定没错。

后面开始就是真正的代码部分。

你可能注意到了，使用sys模块的第一步，就是导入该模块：

import sys

导入sys模块后，我们就有了变量sys指向该模块，利用sys这个变量，就可以访问sys模块的所有功能。

sys模块有一个argv变量，用list存储了命令行的所有参数。argv至少有一个元素，因为第一个参数永远是该.py文件的名称，例如：

运行python hello.py获得的sys.argv就是['hello.py']；

运行python hello.py Michael获得的sys.argv就是['hello.py', 'Michael]。

最后，注意到这两行代码：

if \_\_name\_\_=='\_\_main\_\_':

test()

当我们在命令行运行hello模块文件时，Python解释器把一个特殊变量\_\_name\_\_置为\_\_main\_\_，而如果在其他地方导入该hello模块时，if判断将失败，因此，这种if测试可以让一个模块通过命令行运行时执行一些额外的代码，最常见的就是运行测试。

我们可以用命令行运行hello.py看看效果：

$ python hello.py

Hello, world!

$ python hello.py Michael

Hello, Michael!

如果启动Python交互环境，再导入hello模块：

$ python

Python 2.7.5 (default, Aug 25 2013, 00:04:04)

[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

>>> import hello

>>>

导入时，没有打印Hello, word!，因为没有执行test()函数。

调用hello.test()时，才能打印出Hello, word!：

>>> hello.test()

Hello, world!

#### 别名

导入模块时，还可以使用别名，这样，可以在运行时根据当前环境选择最合适的模块。比如Python标准库一般会提供StringIO和cStringIO两个库，这两个库的接口和功能是一样的，但是cStringIO是C写的，速度更快，所以，你会经常看到这样的写法：

try:

import cStringIO as StringIO

except ImportError: # 导入失败会捕获到ImportError

import StringIO

这样就可以优先导入cStringIO。如果有些平台不提供cStringIO，还可以降级使用StringIO。导入cStringIO时，用import ... as ...指定了别名StringIO，因此，后续代码引用StringIO即可正常工作。

还有类似simplejson这样的库，在Python 2.6之前是独立的第三方库，从2.6开始内置，所以，会有这样的写法：

try:

import json # python >= 2.6

except ImportError:

import simplejson as json # python <= 2.5

由于Python是动态语言，函数签名一致接口就一样，因此，无论导入哪个模块后续代码都能正常工作。

#### 作用域

在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在Python中，是通过\_前缀来实现的。

正常的函数和变量名是公开的（public），可以被直接引用，比如：abc，x123，PI等；

类似\_\_xxx\_\_这样的变量是特殊变量，可以被直接引用，但是有特殊用途，比如上面的\_\_author\_\_，\_\_name\_\_就是特殊变量，hello模块定义的文档注释也可以用特殊变量\_\_doc\_\_访问，我们自己的变量一般不要用这种变量名；

类似\_xxx和\_\_xxx这样的函数或变量就是非公开的（private），不应该被直接引用，比如\_abc，\_\_abc等；

之所以我们说，private函数和变量“不应该”被直接引用，而不是“不能”被直接引用，是因为Python并没有一种方法可以完全限制访问private函数或变量，但是，从编程习惯上不应该引用private函数或变量。

private函数或变量不应该被别人引用，那它们有什么用呢？请看例子：

def \_private\_1(name):

return 'Hello, %s' % name

def \_private\_2(name):

return 'Hi, %s' % name

def greeting(name):

if len(name) > 3:

return \_private\_1(name)

else:

return \_private\_2(name)

我们在模块里公开greeting()函数，而把内部逻辑用private函数隐藏起来了，这样，调用greeting()函数不用关心内部的private函数细节，这也是一种非常有用的代码封装和抽象的方法，即：

外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义为public。

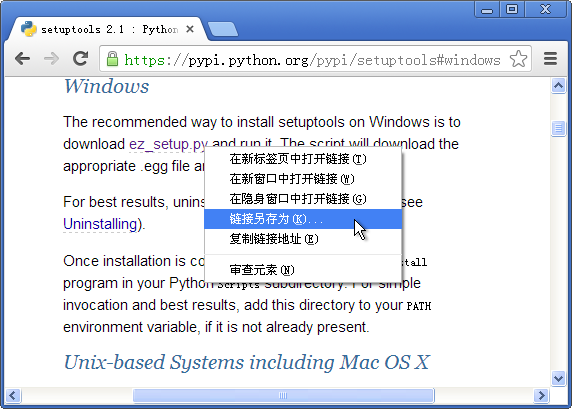
### 2.安装第三方模块

在Python中，安装第三方模块，是通过setuptools这个工具完成的。

如果你正在使用Mac或Linux，安装setuptools本身这个步骤就可以跳过了。

如果你正在使用Windows，请首先从这个地址下载**ez\_setup.py**：

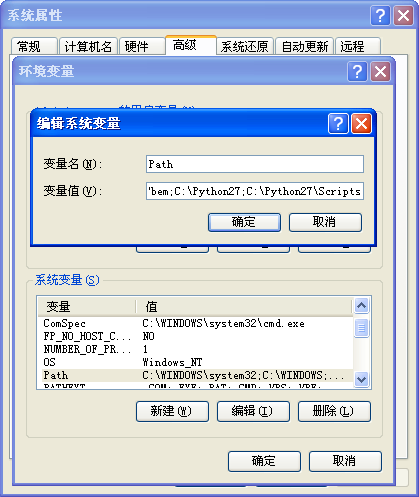
<https://pypi.python.org/pypi/setuptools#windows>



下载后，随便放到一个目录下，然后运行以下命令来安装setuptools：

python ez\_setup.py

在命令提示符窗口下尝试运行easy\_install，Windows会提示未找到命令，原因是easy\_install.exe所在路径还没有被添加到环境变量Path中。请添加C:\Python27\Scripts到环境变量Path：



重新打开命令提示符窗口，就可以运行easy\_install了：

现在，让我们来安装一个第三方库——Python Imaging Library，这是Python下非常强大的处理图像的工具库。一般来说，第三方库都会在Python官方的[pypi.python.org](https://pypi.python.org/)网站注册，要安装一个第三方库，必须先知道该库的名称，可以在官网或者pypi上搜索，比如Python Imaging Library的名称叫PIL，因此，安装Python Imaging Library的命令就是：

easy\_install PIL

耐心等待下载并安装后，就可以使用PIL了。

有了PIL，处理图片易如反掌。随便找个图片生成缩略图：

>>> import Image

>>> im = Image.open('test.png')

>>> print im.format, im.size, im.mode

PNG (400, 300) RGB

>>> im.thumbnail((200, 100))

>>> im.save('thumb.jpg', 'JPEG')

其他常用的第三方库还有MySQL的驱动：MySQL-python，用于科学计算的NumPy库：numpy，用于生成文本的模板工具Jinja2，等等。

#### 模块搜索路径

当我们试图加载一个模块时，Python会在指定的路径下搜索对应的.py文件，如果找不到，就会报错：

>>> import mymodule

Traceback (most recent call last):

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

ImportError: No module named mymodule

默认情况下，Python解释器会搜索当前目录、所有已安装的内置模块和第三方模块，搜索路径存放在sys模块的path变量中：

>>> import sys

>>> sys.path

['', '/Library/Python/2.7/site-packages/pycrypto-2.6.1-py2.7-macosx-10.9-intel.egg', '/Library/Python/2.7/site-packages/PIL-1.1.7-py2.7-macosx-10.9-intel.egg', ...]

如果我们要添加自己的搜索目录，有两种方法：

一是直接修改sys.path，添加要搜索的目录：

>>> import sys

>>> sys.path.append('/Users/michael/my\_py\_scripts')

这种方法是在运行时修改，运行结束后失效。

第二种方法是设置环境变量PYTHONPATH，该环境变量的内容会被自动添加到模块搜索路径中。设置方式与设置Path环境变量类似。注意只需要添加你自己的搜索路径，Python自己本身的搜索路径不受影响。

### 3.使用\_\_future\_\_

Python的每个新版本都会增加一些新的功能，或者对原来的功能作一些改动。有些改动是不兼容旧版本的，也就是在当前版本运行正常的代码，到下一个版本运行就可能不正常了。

从Python 2.7到Python 3.x就有不兼容的一些改动，比如2.x里的字符串用'xxx'表示str，Unicode字符串用u'xxx'表示unicode，而在3.x中，所有字符串都被视为unicode，因此，写u'xxx'和'xxx'是完全一致的，而在2.x中以'xxx'表示的str就必须写成b'xxx'，以此表示“二进制字符串”。

要直接把代码升级到3.x是比较冒进的，因为有大量的改动需要测试。相反，可以在2.7版本中先在一部分代码中测试一些3.x的特性，如果没有问题，再移植到3.x不迟。

Python提供了\_\_future\_\_模块，把下一个新版本的特性导入到当前版本，于是我们就可以在当前版本中测试一些新版本的特性。举例说明如下：

为了适应Python 3.x的新的字符串的表示方法，在2.7版本的代码中，可以通过unicode\_literals来使用Python 3.x的新的语法：

# still running on Python 2.7

from \_\_future\_\_ import unicode\_literals

print '\'xxx\' is unicode?', isinstance('xxx', unicode)

print 'u\'xxx\' is unicode?', isinstance(u'xxx', unicode)

print '\'xxx\' is str?', isinstance('xxx', str)

print 'b\'xxx\' is str?', isinstance(b'xxx', str)

注意到上面的代码仍然在Python 2.7下运行，但结果显示去掉前缀u的'a string'仍是一个unicode，而加上前缀b的b'a string'才变成了str：

$ python task.py

'xxx' is unicode? True

u'xxx' is unicode? True

'xxx' is str? False

b'xxx' is str? True

类似的情况还有除法运算。在Python 2.x中，对于除法有两种情况，如果是整数相除，结果仍是整数，余数会被扔掉，这种除法叫“地板除”：

>>> 10 / 3

3

要做精确除法，必须把其中一个数变成浮点数：

>>> 10.0 / 3

3.3333333333333335

而在Python 3.x中，所有的除法都是精确除法，地板除用//表示：

$ python3

Python 3.3.2 (default, Jan 22 2014, 09:54:40)

[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.2.79)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

>>> 10 / 3

3.3333333333333335

>>> 10 // 3

3

如果你想在Python 2.7的代码中直接使用Python 3.x的除法，可以通过\_\_future\_\_模块的division实现：

from \_\_future\_\_ import division

print '10 / 3 =', 10 / 3

print '10.0 / 3 =', 10.0 / 3

print '10 // 3 =', 10 // 3

结果如下：

10 / 3 = 3.33333333333

10.0 / 3 = 3.33333333333

10 // 3 = 3

#### 小结

由于Python是由社区推动的开源并且免费的开发语言，不受商业公司控制，因此，Python的改进往往比较激进，不兼容的情况时有发生。Python为了确保你能顺利过渡到新版本，特别提供了\_\_future\_\_模块，让你在旧的版本中试验新版本的一些特性。

## 九、面向对象编程

面向对象编程——Object Oriented Programming，简称OOP，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

而面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

在Python中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

我们以一个例子来说明面向过程和面向对象在程序流程上的不同之处。

假设我们要处理学生的成绩表，为了表示一个学生的成绩，面向过程的程序可以用一个dict表示：

std1 = { 'name': 'Michael', 'score': 98 }

std2 = { 'name': 'Bob', 'score': 81 }

而处理学生成绩可以通过函数实现，比如打印学生的成绩：

def print\_score(std):

print '%s: %s' % (std['name'], std['score'])

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是Student这种数据类型应该被视为一个对象，这个对象拥有name和score这两个属性（Property）。如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后，给对象发一个print\_score消息，让对象自己把自己的数据打印出来。

class Student(object):

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

self.name = name

self.score = score

def print\_score(self):

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

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（Method）。面向对象的程序写出来就像这样：

bart = Student('Bart Simpson', 59)

lisa = Student('Lisa Simpson', 87)

bart.print\_score()

lisa.print\_score()

面向对象的设计思想是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。 Class是一种抽象概念，比如我们定义的Class——Student，是指学生这个概念，而实例（Instance）则是一个个具体的 Student，比如，Bart Simpson和Lisa Simpson是两个具体的Student：

所以，面向对象的设计思想是抽象出Class，根据Class创建Instance。

面向对象的抽象程度又比函数要高，因为一个Class既包含数据，又包含操作数据的方法。

#### 小结

数据封装、继承和多态是面向对象的三大特点，我们后面会详细讲解。

### 1.类和实例

面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记类是抽象的模板，比如Student类，而实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。

仍以Student类为例，在Python中，定义类是通过class关键字：

class Student(object):

pass

class后面紧接着是类名，即Student，类名通常是大写开头的单词，紧接着是(object)，表示该类是从哪个类继承下来的，继承的概念我们后面再讲，通常，如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类。

定义好了Student类，就可以根据Student类创建出Student的实例，创建实例是通过类名+()实现的：

>>> bart = Student()

>>> bart

<\_\_main\_\_.Student object at 0x10a67a590>

>>> Student

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

可以看到，变量bart指向的就是一个Student的object，后面的0x10a67a590是内存地址，每个object的地址都不一样，而Student本身则是一个类。

可以自由地给一个实例变量绑定属性，比如，给实例bart绑定一个name属性：

>>> bart.name = 'Bart Simpson'

>>> bart.name

'Bart Simpson'

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的\_\_init\_\_方法，在创建实例的时候，就把name，score等属性绑上去：

class Student(object):

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

self.name = name

self.score = score

注意到\_\_init\_\_方法的第一个参数永远是self，表示创建的实例本身，因此，在\_\_init\_\_方法内部，就可以把各种属性绑定到self，因为self就指向创建的实例本身。

有了\_\_init\_\_方法，在创建实例的时候，就不能传入空的参数了，必须传入与\_\_init\_\_方法匹配的参数，但self不需要传，Python解释器自己会把实例变量传进去：

>>> bart = Student('Bart Simpson', 59)

>>> bart.name

'Bart Simpson'

>>> bart.score

59

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量self，并且，调用时，不用传递该参数。除此之外，类的方法和普通函数没有什么区别，所以，你仍然可以用默认参数、可变参数和关键字参数。

#### 数据封装

面向对象编程的一个重要特点就是数据封装。在上面的Student类中，每个实例就拥有各自的name和score这些数据。我们可以通过函数来访问这些数据，比如打印一个学生的成绩：

>>> def print\_score(std):

... print '%s: %s' % (std.name, std.score)

...

>>> print\_score(bart)

Bart Simpson: 59

但是，既然Student实例本身就拥有这些数据，要访问这些数据，就没有必要从外面的函数去访问，可以直接在Student类的内部定义访问数据的函数，这样，就把“数据”给封装起来了。这些封装数据的函数是和Student类本身是关联起来的，我们称之为类的方法：

class Student(object):

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

self.name = name

self.score = score

def print\_score(self):

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

要定义一个方法，除了第一个参数是self外，其他和普通函数一样。要调用一个方法，只需要在实例变量上直接调用，除了self不用传递，其他参数正常传入：

>>> bart.print\_score()

Bart Simpson: 59

这样一来，我们从外部看Student类，就只需要知道，创建实例需要给出name和score，而如何打印，都是在Student类的内部定义的，这些数据和逻辑被“封装”起来了，调用很容易，但却不用知道内部实现的细节。

封装的另一个好处是可以给Student类增加新的方法，比如get\_grade：

class Student(object):

...

def get\_grade(self):

if self.score >= 90:

return 'A'

elif self.score >= 60:

return 'B'

else:

return 'C'

同样的，get\_grade方法可以直接在实例变量上调用，不需要知道内部实现细节：

>>> bart.get\_grade()

'C'

#### 小结

类是创建实例的模板，而实例则是一个一个具体的对象，各个实例拥有的数据都互相独立，互不影响；

方法就是与实例绑定的函数，和普通函数不同，方法可以直接访问实例的数据；

通过在实例上调用方法，我们就直接操作了对象内部的数据，但无需知道方法内部的实现细节。

和静态语言不同，Python允许对实例变量绑定任何数据，也就是说，对于两个实例变量，虽然它们都是同一个类的不同实例，但拥有的变量名称都可能不同：

>>> bart = Student('Bart Simpson', 59)

>>> lisa = Student('Lisa Simpson', 87)

>>> bart.age = 8

>>> bart.age

8

>>> lisa.age

Traceback (most recent call last):

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

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

### 2.访问限制

在Class内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，这样，就隐藏了内部的复杂逻辑。

但是，从前面Student类的定义来看，外部代码还是可以自由地修改一个实例的name、score属性：

>>> bart = Student('Bart Simpson', 98)

>>> bart.score

98

>>> bart.score = 59

>>> bart.score

59

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线\_\_，在Python中，实例的变量名如果以\_\_开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问，所以，我们把Student类改一改：

class Student(object):

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

self.\_\_name = name

self.\_\_score = score

def print\_score(self):

print '%s: %s' % (self.\_\_name, self.\_\_score)

改完后，对于外部代码来说，没什么变动，但是已经无法从外部访问实例变量.\_\_name和实例变量.\_\_score了：

>>> bart = Student('Bart Simpson', 98)

>>> bart.\_\_name

Traceback (most recent call last):

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

AttributeError: 'Student' object has no attribute '\_\_name'

这样就确保了外部代码不能随意修改对象内部的状态，这样通过访问限制的保护，代码更加健壮。

但是如果外部代码要获取name和score怎么办？可以给Student类增加get\_name和get\_score这样的方法：

class Student(object):

...

def get\_name(self):

return self.\_\_name

def get\_score(self):

return self.\_\_score

如果又要允许外部代码修改score怎么办？可以给Student类增加set\_score方法：

class Student(object):

...

def set\_score(self, score):

self.\_\_score = score

你也许会问，原先那种直接通过bart.score = 59也可以修改啊，为什么要定义一个方法大费周折？因为在方法中，可以对参数做检查，避免传入无效的参数：

class Student(object):

...

def set\_score(self, score):

if 0 <= score <= 100:

self.\_\_score = score

else:

raise ValueError('bad score')

需要注意的是，在Python中，变量名类似\_\_xxx\_\_的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量，所以，不能用\_\_name\_\_、\_\_score\_\_这样的变量名。

有些时候，你会看到以一个下划线开头的实例变量名，比如\_name，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。

双下划线开头的实例变量是不是一定不能从外部访问呢？其实也不是。不能直接访问\_\_name是因为Python解释器对外把\_\_name变量改成了\_Student\_\_name，所以，仍然可以通过\_Student\_\_name来访问\_\_name变量：

>>> bart.\_Student\_\_name

'Bart Simpson'

但是强烈建议你不要这么干，因为不同版本的Python解释器可能会把\_\_name改成不同的变量名。

总的来说就是，Python本身没有任何机制阻止你干坏事，一切全靠自觉。

### 3.继承和多态

在OOP程序设计中，当我们定义一个class的时候，可以从某个现有的class继承，新的class称为子类（Subclass），而被继承的class称为基类、父类或超类（Base class、Super class）。

比如，我们已经编写了一个名为Animal的class，有一个run()方法可以直接打印：

class Animal(object):

def run(self):

print 'Animal is running...'

当我们需要编写Dog和Cat类时，就可以直接从Animal类继承：

class Dog(Animal):

pass

class Cat(Animal):

pass

对于Dog来说，Animal就是它的父类，对于Animal来说，Dog就是它的子类。Cat和Dog类似。

继承有什么好处？最大的好处是子类获得了父类的全部功能。由于Animial实现了run()方法，因此，Dog和Cat作为它的子类，什么事也没干，就自动拥有了run()方法：

dog = Dog()

dog.run()

cat = Cat()

cat.run()

运行结果如下：

Animal is running...

Animal is running...

当然，也可以对子类增加一些方法，比如Dog类：

class Dog(Animal):

def run(self):

print 'Dog is running...'

def eat(self):

print 'Eating meat...'

继承的第二个好处需要我们对代码做一点改进。你看到了，无论是Dog还是Cat，它们run()的时候，显示的都是Animal is running...，符合逻辑的做法是分别显示Dog is running...和Cat is running...，因此，对Dog和Cat类改进如下：

class Dog(Animal):

def run(self):

print 'Dog is running...'

class Cat(Animal):

def run(self):

print 'Cat is running...'

再次运行，结果如下：

Dog is running...

Cat is running...

当子类和父类都存在相同的run()方法时，我们说，子类的run()覆盖了父类的run()，在代码运行的时候，总是会调用子类的run()。这样，我们就获得了继承的另一个好处：多态。

要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个class的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和Python自带的数据类型，比如str、list、dict没什么两样：

a = list() # a是list类型

b = Animal() # b是Animal类型

c = Dog() # c是Dog类型

判断一个变量是否是某个类型可以用isinstance()判断：

>>> isinstance(a, list)

True

>>> isinstance(b, Animal)

True

>>> isinstance(c, Dog)

True

看来a、b、c确实对应着list、Animal、Dog这3种类型。

但是等等，试试：

>>> isinstance(c, Animal)

True

看来c不仅仅是Dog，c还是Animal！

不过仔细想想，这是有道理的，因为Dog是从Animal继承下来的，当我们创建了一个Dog的实例c时，我们认为c的数据类型是Dog没错，但c同时也是Animal也没错，Dog本来就是Animal的一种！

所以，在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是，反过来就不行：

>>> b = Animal()

>>> isinstance(b, Dog)

False

Dog可以看成Animal，但Animal不可以看成Dog。

要理解多态的好处，我们还需要再编写一个函数，这个函数接受一个Animal类型的变量：

def run\_twice(animal):

animal.run()

animal.run()

当我们传入Animal的实例时，run\_twice()就打印出：

>>> run\_twice(Animal())

Animal is running...

Animal is running...

当我们传入Dog的实例时，run\_twice()就打印出：

>>> run\_twice(Dog())

Dog is running...

Dog is running...

当我们传入Cat的实例时，run\_twice()就打印出：

>>> run\_twice(Cat())

Cat is running...

Cat is running...

看上去没啥意思，但是仔细想想，现在，如果我们再定义一个Tortoise类型，也从Animal派生：

class Tortoise(Animal):

def run(self):

print 'Tortoise is running slowly...'

当我们调用run\_twice()时，传入Tortoise的实例：

>>> run\_twice(Tortoise())

Tortoise is running slowly...

Tortoise is running slowly...

你会发现，新增一个Animal的子类，不必对run\_twice()做任何修改，实际上，任何依赖Animal作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态。

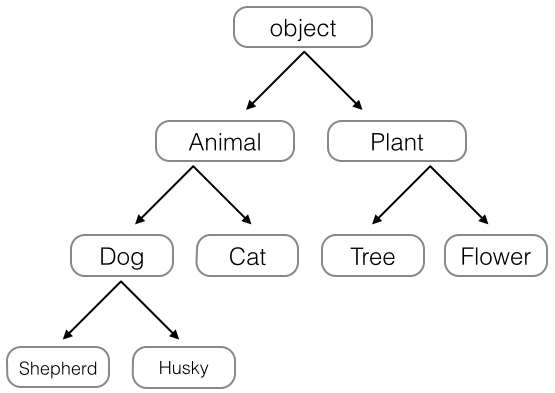
多态的好处就是，当我们需要传入Dog、Cat、Tortoise……时，我们只需要接收Animal类型就可以了，因为Dog、Cat、Tortoise……都是Animal类型，然后，按照Animal类型进行操作即可。由于Animal类型有run()方法，因此，传入的任意类型，只要是Animal类或者子类，就会自动调用实际类型的run()方法，这就是多态的意思：

对于一个变量，我们只需要知道它是Animal类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Animal、Dog、Cat还是Tortoise对象上，由运行时该对象的确切类型决定，这就是多态真正的威力：调用方只管调用，不管细节，而当我们新增一种Animal的子类时，只要确保run()方法编写正确，不用管原来的代码是如何调用的。这就是著名的“开闭”原则：

对扩展开放：允许新增Animal子类；

对修改封闭：不需要修改依赖Animal类型的run\_twice()等函数。

继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类object，这些继承关系看上去就像一颗倒着的树。比如如下的继承树：



#### 小结

继承可以把父类的所有功能都直接拿过来，这样就不必重零做起，子类只需要新增自己特有的方法，也可以把父类不适合的方法覆盖重写；

有了继承，才能有多态。在调用类实例方法的时候，尽量把变量视作父类类型，这样，所有子类类型都可以正常被接收；

旧的方式定义Python类允许不从object类继承，但这种编程方式已经严重不推荐使用。任何时候，如果没有合适的类可以继承，就继承自object类。

### 4.获取对象信息

当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

#### 使用type()

首先，我们来判断对象类型，使用type()函数：

基本类型都可以用type()判断：

>>> type(123)

<type 'int'>

>>> type('str')

<type 'str'>

>>> type(None)

<type 'NoneType'>

如果一个变量指向函数或者类，也可以用type()判断：

>>> type(abs)

<type 'builtin\_function\_or\_method'>

>>> type(a)

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

但是type()函数返回的是什么类型呢？它返回type类型。如果我们要在if语句中判断，就需要比较两个变量的type类型是否相同：

>>> type(123)==type(456)

True

>>> type('abc')==type('123')

True

>>> type('abc')==type(123)

False

但是这种写法太麻烦，Python把每种type类型都定义好了常量，放在types模块里，使用之前，需要先导入：

>>> import types

>>> type('abc')==types.StringType

True

>>> type(u'abc')==types.UnicodeType

True

>>> type([])==types.ListType

True

>>> type(str)==types.TypeType

True

最后注意到有一种类型就叫TypeType，所有类型本身的类型就是TypeType，比如：

>>> type(int)==type(str)==types.TypeType

True

#### 使用isinstance()

对于class的继承关系来说，使用type()就很不方便。我们要判断class的类型，可以使用isinstance()函数。

我们回顾上次的例子，如果继承关系是：

object -> Animal -> Dog -> Husky

那么，isinstance()就可以告诉我们，一个对象是否是某种类型。先创建3种类型的对象：

>>> a = Animal()

>>> d = Dog()

>>> h = Husky()

然后，判断：

>>> isinstance(h, Husky)

True

没有问题，因为h变量指向的就是Husky对象。

再判断：

>>> isinstance(h, Dog)

True

h虽然自身是Husky类型，但由于Husky是从Dog继承下来的，所以，h也还是Dog类型。换句话说，isinstance()判断的是一个对象是否是该类型本身，或者位于该类型的父继承链上。

因此，我们可以确信，h还是Animal类型：

>>> isinstance(h, Animal)

True

同理，实际类型是Dog的d也是Animal类型：

>>> isinstance(d, Dog) and isinstance(d, Animal)

True

但是，d不是Husky类型：

>>> isinstance(d, Husky)

False

能用type()判断的基本类型也可以用isinstance()判断：

>>> isinstance('a', str)

True

>>> isinstance(u'a', unicode)

True

>>> isinstance('a', unicode)

False

并且还可以判断一个变量是否是某些类型中的一种，比如下面的代码就可以判断是否是str或者unicode：

>>> isinstance('a', (str, unicode))

True

>>> isinstance(u'a', (str, unicode))

True

由于str和unicode都是从basestring继承下来的，所以，还可以把上面的代码简化为：

>>> isinstance(u'a', basestring)

True

#### 使用dir()

如果要获得一个对象的所有属性和方法，可以使用dir()函数，它返回一个包含字符串的list，比如，获得一个str对象的所有属性和方法：

>>> dir('ABC')

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_getslice\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_formatter\_field\_name\_split', '\_formatter\_parser', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

类似\_\_xxx\_\_的属性和方法在Python中都是有特殊用途的，比如\_\_len\_\_方法返回长度。在Python中，如果你调用len()函数试图获取一个对象的长度，实际上，在len()函数内部，它自动去调用该对象的\_\_len\_\_()方法，所以，下面的代码是等价的：

>>> len('ABC')

3

>>> 'ABC'.\_\_len\_\_()

3

我们自己写的类，如果也想用len(myObj)的话，就自己写一个\_\_len\_\_()方法：

>>> class MyObject(object):

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

... return 100

...

>>> obj = MyObject()

>>> len(obj)

100

剩下的都是普通属性或方法，比如lower()返回小写的字符串：

>>> 'ABC'.lower()

'abc'

仅仅把属性和方法列出来是不够的，配合getattr()、setattr()以及hasattr()，我们可以直接操作一个对象的状态：

>>> class MyObject(object):

... def \_\_init\_\_(self):

... self.x = 9

... def power(self):

... return self.x \* self.x

...

>>> obj = MyObject()

紧接着，可以测试该对象的属性：

>>> hasattr(obj, 'x') # 有属性'x'吗？

True

>>> obj.x

9

>>> hasattr(obj, 'y') # 有属性'y'吗？

False

>>> setattr(obj, 'y', 19) # 设置一个属性'y'

>>> hasattr(obj, 'y') # 有属性'y'吗？

True

>>> getattr(obj, 'y') # 获取属性'y'

19

>>> obj.y # 获取属性'y'

19

如果试图获取不存在的属性，会抛出AttributeError的错误：

>>> getattr(obj, 'z') # 获取属性'z'

Traceback (most recent call last):

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

AttributeError: 'MyObject' object has no attribute 'z'

可以传入一个default参数，如果属性不存在，就返回默认值：

>>> getattr(obj, 'z', 404) # 获取属性'z'，如果不存在，返回默认值404

404

也可以获得对象的方法：

>>> hasattr(obj, 'power') # 有属性'power'吗？

True

>>> getattr(obj, 'power') # 获取属性'power'

<bound method MyObject.power of <\_\_main\_\_.MyObject object at 0x108ca35d0>>

>>> fn = getattr(obj, 'power') # 获取属性'power'并赋值到变量fn

>>> fn # fn指向obj.power

<bound method MyObject.power of <\_\_main\_\_.MyObject object at 0x108ca35d0>>

>>> fn() # 调用fn()与调用obj.power()是一样的

81

#### 小结

通过内置的一系列函数，我们可以对任意一个Python对象进行剖析，拿到其内部的数据。要注意的是，只有在不知道对象信息的时候，我们才会去获取对象信息。如果可以直接写：

sum = obj.x + obj.y

就不要写：

sum = getattr(obj, 'x') + getattr(obj, 'y')

一个正确的用法的例子如下：

def readImage(fp):

if hasattr(fp, 'read'):

return readData(fp)

return None

假设我们希望从文件流fp中读取图像，我们首先要判断该fp对象是否存在read方法，如果存在，则该对象是一个流，如果不存在，则无法读取。hasattr()就派上了用场。

请注意，在Python这类动态语言中，有read()方法，不代表该fp对象就是一个文件流，它也可能是网络流，也可能是内存中的一个字节流，但只要read()方法返回的是有效的图像数据，就不影响读取图像的功能。

## 十、面向对象高级编程

数据封装、继承和多态只是面向对象程序设计中最基础的3个概念。在Python中，面向对象还有很多高级特性，允许我们写出非常强大的功能。

我们会讨论多重继承、定制类、元类等概念。

### 1.使用\_\_slots\_\_

正常情况下，当我们定义了一个class，创建了一个class的实例后，我们可以给该实例绑定任何属性和方法，这就是动态语言的灵活性。先定义class：

>>> class Student(object):

... pass

...

[Try](http://www.liaoxuefeng.com/wiki/001374738125095c955c1e6d8bb493182103fac9270762a000/0013868200605560b1bd3c660bf494282ede59fee17e781000#0)

然后，尝试给实例绑定一个属性：

>>> s = Student()

>>> s.name = 'Michael' # 动态给实例绑定一个属性

>>> print s.name

Michael

还可以尝试给实例绑定一个方法：

>>> def set\_age(self, age): # 定义一个函数作为实例方法

... self.age = age

...

>>> from types import MethodType

>>> s.set\_age = MethodType(set\_age, s, Student) # 给实例绑定一个方法

>>> s.set\_age(25) # 调用实例方法

>>> s.age # 测试结果

25

但是，给一个实例绑定的方法，对另一个实例是不起作用的：

>>> s2 = Student() # 创建新的实例

>>> s2.set\_age(25) # 尝试调用方法

Traceback (most recent call last):

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

AttributeError: 'Student' object has no attribute 'set\_age'

为了给所有实例都绑定方法，可以给class绑定方法：

>>> def set\_score(self, score):

... self.score = score

...

>>> Student.set\_score = MethodType(set\_score, None, Student)

给class绑定方法后，所有实例均可调用：

>>> s.set\_score(100)

>>> s.score

100

>>> s2.set\_score(99)

>>> s2.score

99

通常情况下，上面的set\_score方法可以直接定义在class中，但动态绑定允许我们在程序运行的过程中动态给class加上功能，这在静态语言中很难实现。

#### 使用\_\_slots\_\_

但是，如果我们想要限制class的属性怎么办？比如，只允许对Student实例添加name和age属性。

为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的\_\_slots\_\_变量，来限制该class能添加的属性：

>>> class Student(object):

... \_\_slots\_\_ = ('name', 'age') # 用tuple定义允许绑定的属性名称

...

然后，我们试试：

>>> s = Student() # 创建新的实例

>>> s.name = 'Michael' # 绑定属性'name'

>>> s.age = 25 # 绑定属性'age'

>>> s.score = 99 # 绑定属性'score'

Traceback (most recent call last):

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

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

由于'score'没有被放到\_\_slots\_\_中，所以不能绑定score属性，试图绑定score将得到AttributeError的错误。

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

>>> class GraduateStudent(Student):

... pass

...

>>> g = GraduateStudent()

>>> g.score = 9999

除非在子类中也定义\_\_slots\_\_，这样，子类允许定义的属性就是自身的\_\_slots\_\_加上父类的\_\_slots\_\_。

### 2.使用@property

在绑定属性时，如果我们直接把属性暴露出去，虽然写起来很简单，但是，没办法检查参数，导致可以把成绩随便改：

s = Student()

s.score = 9999

这显然不合逻辑。为了限制score的范围，可以通过一个set\_score()方法来设置成绩，再通过一个get\_score()来获取成绩，这样，在set\_score()方法里，就可以检查参数：

class Student(object):

def get\_score(self):

return self.\_score

def set\_score(self, value):

if not isinstance(value, int):

raise ValueError('score must be an integer!')

if value < 0 or value > 100:

raise ValueError('score must between 0 ~ 100!')

self.\_score = value

现在，对任意的Student实例进行操作，就不能随心所欲地设置score了：

>>> s = Student()

>>> s.set\_score(60) # ok!

>>> s.get\_score()

60

>>> s.set\_score(9999)

Traceback (most recent call last):

...

ValueError: score must between 0 ~ 100!

但是，上面的调用方法又略显复杂，没有直接用属性这么直接简单。

有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量呢？对于追求完美的Python程序员来说，这是必须要做到的！

还记得装饰器（decorator）可以给函数动态加上功能吗？对于类的方法，装饰器一样起作用。Python内置的@property装饰器就是负责把一个方法变成属性调用的：

class Student(object):

@property

def score(self):

return self.\_score

@score.setter

def score(self, value):

if not isinstance(value, int):

raise ValueError('score must be an integer!')

if value < 0 or value > 100:

raise ValueError('score must between 0 ~ 100!')

self.\_score = value

@property的实现比较复杂，我们先考察如何使用。把一个getter方法变成属性，只需要加上@property就可以了，此时，@property本身又创建了另一个装饰器@score.setter，负责把一个setter方法变成属性赋值，于是，我们就拥有一个可控的属性操作：

>>> s = Student()

>>> s.score = 60 # OK，实际转化为s.set\_score(60)

>>> s.score # OK，实际转化为s.get\_score()

60

>>> s.score = 9999

Traceback (most recent call last):

...

ValueError: score must between 0 ~ 100!

注意到这个神奇的@property，我们在对实例属性操作的时候，就知道该属性很可能不是直接暴露的，而是通过getter和setter方法来实现的。

还可以定义只读属性，只定义getter方法，不定义setter方法就是一个只读属性：

class Student(object):

@property

def birth(self):

return self.\_birth

@birth.setter

def birth(self, value):

self.\_birth = value

@property

def age(self):

return 2014 - self.\_birth

上面的birth是可读写属性，而age就是一个**只读**属性，因为age可以根据birth和当前时间计算出来。

#### 小结

@property广泛应用在类的定义中，可以让调用者写出简短的代码，同时保证对参数进行必要的检查，这样，程序运行时就减少了出错的可能性。

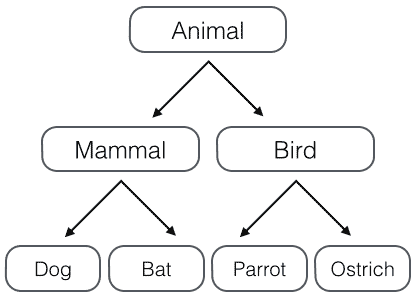
### 3.多重继承

继承是面向对象编程的一个重要的方式，因为通过继承，子类就可以扩展父类的功能。

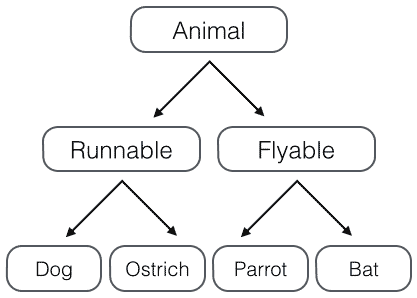
回忆一下Animal类层次的设计，假设我们要实现以下4种动物：

* Dog - 狗狗；
* Bat - 蝙蝠；
* Parrot - 鹦鹉；
* Ostrich - 鸵鸟。

如果按照哺乳动物和鸟类归类，我们可以设计出这样的类的层次：



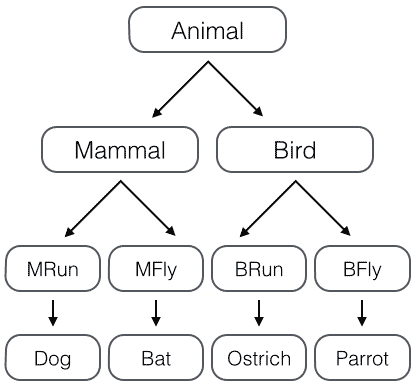
但是如果按照“能跑”和“能飞”来归类，我们就应该设计出这样的类的层次：



如果要把上面的两种分类都包含进来，我们就得设计更多的层次：

* 哺乳类：能跑的哺乳类，能飞的哺乳类；
* 鸟类：能跑的鸟类，能飞的鸟类。

这么一来，类的层次就复杂了：



如果要再增加“宠物类”和“非宠物类”，这么搞下去，类的数量会呈指数增长，很明显这样设计是不行的。

正确的做法是采用多重继承。首先，主要的类层次仍按照哺乳类和鸟类设计：

class Animal(object):

pass

# 大类:

class Mammal(Animal):

pass

class Bird(Animal):

pass

# 各种动物:

class Dog(Mammal):

pass

class Bat(Mammal):

pass

class Parrot(Bird):

pass

class Ostrich(Bird):

pass

现在，我们要给动物再加上Runnable和Flyable的功能，只需要先定义好Runnable和Flyable的类：

class Runnable(object):

def run(self):

print('Running...')

class Flyable(object):

def fly(self):

print('Flying...')

对于需要Runnable功能的动物，就多继承一个Runnable，例如Dog：

class Dog(Mammal, Runnable):

pass

对于需要Flyable功能的动物，就多继承一个Flyable，例如Bat：

class Bat(Mammal, Flyable):

pass

通过多重继承，一个子类就可以同时获得多个父类的所有功能。

#### Mixin

在设计类的继承关系时，通常，主线都是单一继承下来的，例如，Ostrich继承自Bird。但是，如果需要“混入”额外的功能，通过多重继承就可以实现，比如，让Ostrich除了继承自Bird外，再同时继承Runnable。这种设计通常称之为Mixin。

为了更好地看出继承关系，我们把Runnable和Flyable改为RunnableMixin和FlyableMixin。类似的，你还可以定义出肉食动物CarnivorousMixin和植食动物HerbivoresMixin，让某个动物同时拥有好几个Mixin：

class Dog(Mammal, RunnableMixin, CarnivorousMixin):

pass

Mixin的目的就是给一个类增加多个功能，这样，在设计类的时候，我们优先考虑通过多重继承来组合多个Mixin的功能，而不是设计多层次的复杂的继承关系。

Python自带的很多库也使用了Mixin。举个例子，Python自带了TCPServer和UDPServer这两类网络服务，而要同时服务多个用户就必须使用多进程或多线程模型，这两种模型由ForkingMixin和ThreadingMixin提供。通过组合，我们就可以创造出合适的服务来。

比如，编写一个多进程模式的TCP服务，定义如下：

class MyTCPServer(TCPServer, ForkingMixin):

pass

编写一个多线程模式的UDP服务，定义如下：

class MyUDPServer(UDPServer, ThreadingMixin):

pass

如果你打算搞一个更先进的协程模型，可以编写一个CoroutineMixin：

class MyTCPServer(TCPServer, CoroutineMixin):

pass

这样一来，我们不需要复杂而庞大的继承链，只要选择组合不同的类的功能，就可以快速构造出所需的子类。

#### 小结

由于Python允许使用多重继承，因此，Mixin就是一种常见的设计。

只允许单一继承的语言（如Java）不能使用Mixin的设计。

### 4.定制类

看到类似\_\_slots\_\_这种形如\_\_xxx\_\_的变量或者函数名就要注意，这些在Python中是有特殊用途的。

\_\_slots\_\_我们已经知道怎么用了，\_\_len\_\_()方法我们也知道是为了能让class作用于len()函数。

除此之外，Python的class中还有许多这样有特殊用途的函数，可以帮助我们定制类。

#### \_\_str\_\_

我们先定义一个Student类，打印一个实例：

>>> class Student(object):

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

... self.name = name

...

>>> print Student('Michael')

<\_\_main\_\_.Student object at 0x109afb190>

打印出一堆<\_\_main\_\_.Student object at 0x109afb190>，不好看。

怎么才能打印得好看呢？只需要定义好\_\_str\_\_()方法，返回一个好看的字符串就可以了：

>>> class Student(object):

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

... self.name = name

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

... return 'Student object (name: %s)' % self.name

...

>>> print Student('Michael')

Student object (name: Michael)

这样打印出来的实例，不但好看，而且容易看出实例内部重要的数据。

但是细心的朋友会发现直接敲变量不用print，打印出来的实例还是不好看：

>>> s = Student('Michael')

>>> s

<\_\_main\_\_.Student object at 0x109afb310>

这是因为直接显示变量调用的不是\_\_str\_\_()，而是\_\_repr\_\_()，两者的区别是\_\_str\_\_()返回用户看到的字符串，而\_\_repr\_\_()返回程序开发者看到的字符串，也就是说，\_\_repr\_\_()是为调试服务的。

解决办法是再定义一个\_\_repr\_\_()。但是通常\_\_str\_\_()和\_\_repr\_\_()代码都是一样的，所以，有个偷懒的写法：

class Student(object):

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

self.name = name

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

return 'Student object (name=%s)' % self.name

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

#### \_\_iter\_\_

如果一个类想被用于for ... in循环，类似list或tuple那样，就必须实现一个\_\_iter\_\_()方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的next()方法拿到循环的下一个值，直到遇到StopIteration错误时退出循环。

我们以斐波那契数列为例，写一个Fib类，可以作用于for循环：

class Fib(object):

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

self.a, self.b = 0, 1 # 初始化两个计数器a，b

def \_\_iter\_\_(self):

return self # 实例本身就是迭代对象，故返回自己

def next(self):

self.a, self.b = self.b, self.a + self.b # 计算下一个值

if self.a > 100000: # 退出循环的条件

raise StopIteration();

return self.a # 返回下一个值

现在，试试把Fib实例作用于for循环：

>>> for n in Fib():

... print n

...

1

1

2

3

5

...

46368

75025

#### \_\_getitem\_\_

Fib实例虽然能作用于for循环，看起来和list有点像，但是，把它当成list来使用还是不行，比如，取第5个元素：

>>> Fib()[5]

Traceback (most recent call last):

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

TypeError: 'Fib' object does not support indexing

要表现得像list那样按照下标取出元素，需要实现\_\_getitem\_\_()方法：

class Fib(object):

def \_\_getitem\_\_(self, n):

a, b = 1, 1

for x in range(n):

a, b = b, a + b

return a

现在，就可以按下标访问数列的任意一项了：

>>> f = Fib()

>>> f[0]

1

>>> f[1]

1

>>> f[2]

2

>>> f[3]

3

>>> f[10]

89

>>> f[100]

573147844013817084101

但是list有个神奇的切片方法：

>>> range(100)[5:10]

[5, 6, 7, 8, 9]

对于Fib却报错。原因是\_\_getitem\_\_()传入的参数可能是一个int，也可能是一个切片对象slice，所以要做判断：

class Fib(object):

def \_\_getitem\_\_(self, n):

if isinstance(n, int):

a, b = 1, 1

for x in range(n):

a, b = b, a + b

return a

if isinstance(n, slice):

start = n.start

stop = n.stop

a, b = 1, 1

L = []

for x in range(stop):

if x >= start:

L.append(a)

a, b = b, a + b

return L

现在试试Fib的切片：

>>> f = Fib()

>>> f[0:5]

[1, 1, 2, 3, 5]

>>> f[:10]

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

但是没有对step参数作处理：

>>> f[:10:2]

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

也没有对负数作处理，所以，要正确实现一个\_\_getitem\_\_()还是有很多工作要做的。

此外，如果把对象看成dict，\_\_getitem\_\_()的参数也可能是一个可以作key的object，例如str。

与之对应的是\_\_setitem\_\_()方法，把对象视作list或dict来对集合赋值。最后，还有一个\_\_delitem\_\_()方法，用于删除某个元素。

总之，通过上面的方法，我们自己定义的类表现得和Python自带的list、tuple、dict没什么区别，这完全归功于动态语言的“鸭子类型”，不需要强制继承某个接口。

#### \_\_getattr\_\_

正常情况下，当我们调用类的方法或属性时，如果不存在，就会报错。比如定义Student类：

class Student(object):

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

self.name = 'Michael'

调用name属性，没问题，但是，调用不存在的score属性，就有问题了：

>>> s = Student()

>>> print s.name

Michael

>>> print s.score

Traceback (most recent call last):

...

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

错误信息很清楚地告诉我们，没有找到score这个attribute。

要避免这个错误，除了可以加上一个score属性外，Python还有另一个机制，那就是写一个\_\_getattr\_\_()方法，动态返回一个属性。修改如下：

class Student(object):

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

self.name = 'Michael'

def \_\_getattr\_\_(self, attr):

if attr=='score':

return 99

当调用不存在的属性时，比如score，Python解释器会试图调用\_\_getattr\_\_(self, 'score')来尝试获得属性，这样，我们就有机会返回score的值：

>>> s = Student()

>>> s.name

'Michael'

>>> s.score

99

返回函数也是完全可以的：

class Student(object):

def \_\_getattr\_\_(self, attr):

if attr=='age':

return lambda: 25

只是调用方式要变为：

>>> s.age()

25

注意，只有在没有找到属性的情况下，才调用\_\_getattr\_\_，已有的属性，比如name，不会在\_\_getattr\_\_中查找。

此外，注意到任意调用如s.abc都会返回None，这是因为我们定义的\_\_getattr\_\_默认返回就是None。要让class只响应特定的几个属性，我们就要按照约定，抛出AttributeError的错误：

class Student(object):

def \_\_getattr\_\_(self, attr):

if attr=='age':

return lambda: 25

raise AttributeError('\'Student\' object has no attribute \'%s\'' % attr)

这实际上可以把一个类的所有属性和方法调用全部动态化处理了，不需要任何特殊手段。

这种完全动态调用的特性有什么实际作用呢？作用就是，可以针对完全动态的情况作调用。

举个例子：

现在很多网站都搞REST API，比如新浪微博、豆瓣啥的，调用API的URL类似：

http://api.server/user/friends

http://api.server/user/timeline/list

如果要写SDK，给每个URL对应的API都写一个方法，那得累死，而且，API一旦改动，SDK也要改。

利用完全动态的\_\_getattr\_\_，我们可以写出一个链式调用：

class Chain(object):

def \_\_init\_\_(self, path=''):

self.\_path = path

def \_\_getattr\_\_(self, path):

return Chain('%s/%s' % (self.\_path, path))

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

return self.\_path

试试：

>>> Chain().status.user.timeline.list

'/status/user/timeline/list'

这样，无论API怎么变，SDK都可以根据URL实现完全动态的调用，而且，不随API的增加而改变！

还有些REST API会把参数放到URL中，比如GitHub的API：

GET /users/:user/repos

调用时，需要把:user替换为实际用户名。如果我们能写出这样的链式调用：

Chain().users('michael').repos

就可以非常方便地调用API了。有兴趣的童鞋可以试试写出来。

#### \_\_call\_\_

一个对象实例可以有自己的属性和方法，当我们调用实例方法时，我们用instance.method()来调用。能不能直接在实例本身上调用呢？类似instance()？在Python中，答案是肯定的。

任何类，只需要定义一个\_\_call\_\_()方法，就可以直接对实例进行调用。请看示例：

class Student(object):

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

self.name = name

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

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

调用方式如下：

>>> s = Student('Michael')

>>> s()

My name is Michael.

\_\_call\_\_()还可以定义参数。对实例进行直接调用就好比对一个函数进行调用一样，所以你完全可以把对象看成函数，把函数看成对象，因为这两者之间本来就没啥根本的区别。

如果你把对象看成函数，那么函数本身其实也可以在运行期动态创建出来，因为类的实例都是运行期创建出来的，这么一来，我们就模糊了对象和函数的界限。

那么，怎么判断一个变量是对象还是函数呢？其实，更多的时候，我们需要判断一个对象是否能被调用，能被调用的对象就是一个Callable对象，比如函数和我们上面定义的带有\_\_call()\_\_的类实例：

>>> callable(Student())

True

>>> callable(max)

True

>>> callable([1, 2, 3])

False

>>> callable(None)

False

>>> callable('string')

False

通过callable()函数，我们就可以判断一个对象是否是“可调用”对象。

#### 小结

Python的class允许定义许多定制方法，可以让我们非常方便地生成特定的类。

本节介绍的是最常用的几个定制方法，还有很多可定制的方法，请参考[Python的官方文档](http://docs.python.org/2/reference/datamodel.html#special-method-names)。

### 5.使用元类

#### type()

动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时定义的，而是运行时动态创建的。

比方说我们要定义一个Hello的class，就写一个hello.py模块：

class Hello(object):

def hello(self, name='world'):

print('Hello, %s.' % name)

当Python解释器载入hello模块时，就会依次执行该模块的所有语句，执行结果就是动态创建出一个Hello的class对象，测试如下：

>>> from hello import Hello

>>> h = Hello()

>>> h.hello()

Hello, world.

>>> print(type(Hello))

<type 'type'>

>>> print(type(h))

<class 'hello.Hello'>

type()函数可以查看一个类型或变量的类型，Hello是一个class，它的类型就是type，而h是一个实例，它的类型就是class Hello。

我们说class的定义是运行时动态创建的，而创建class的方法就是使用type()函数。

type()函数既可以返回一个对象的类型，又可以创建出新的类型，比如，我们可以通过type()函数创建出Hello类，而无需通过class Hello(object)...的定义：

>>> def fn(self, name='world'): # 先定义函数

... print('Hello, %s.' % name)

...

>>> Hello = type('Hello', (object,), dict(hello=fn)) # 创建Hello class

>>> h = Hello()

>>> h.hello()

Hello, world.

>>> print(type(Hello))

<type 'type'>

>>> print(type(h))

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

要创建一个class对象，type()函数依次传入3个参数：

1. class的名称；
2. 继承的父类集合，注意Python支持多重继承，如果只有一个父类，别忘了tuple的单元素写法；
3. class的方法名称与函数绑定，这里我们把函数fn绑定到方法名hello上。

通过type()函数创建的类和直接写class是完全一样的，因为Python解释器遇到class定义时，仅仅是扫描一下class定义的语法，然后调用type()函数创建出class。

正常情况下，我们都用class Xxx...来定义类，但是，type()函数也允许我们动态创建出类来，也就是说，动态语言本身支持运行期动态创建类，这和静态语言有非常大的不同，要在静态语言运行期创建类，必须构造源代码字符串再调用编译器，或者借助一些工具生成字节码实现，本质上都是动态编译，会非常复杂。

#### metaclass

除了使用type()动态创建类以外，要控制类的创建行为，还可以使用metaclass。

metaclass，直译为元类，简单的解释就是：

当我们定义了类以后，就可以根据这个类创建出实例，所以：先定义类，然后创建实例。

但是如果我们想创建出类呢？那就必须根据metaclass创建出类，所以：先定义metaclass，然后创建类。

连接起来就是：先定义metaclass，就可以创建类，最后创建实例。

所以，metaclass允许你创建类或者修改类。换句话说，你可以把类看成是metaclass创建出来的“实例”。

metaclass是Python面向对象里最难理解，也是最难使用的魔术代码。正常情况下，你不会碰到需要使用metaclass的情况，所以，以下内容看不懂也没关系，因为基本上你不会用到。

我们先看一个简单的例子，这个metaclass可以给我们自定义的MyList增加一个add方法：

定义ListMetaclass，按照默认习惯，metaclass的类名总是以Metaclass结尾，以便清楚地表示这是一个metaclass：

# metaclass是创建类，所以必须从`type`类型派生：

class ListMetaclass(type):

def \_\_new\_\_(cls, name, bases, attrs):

attrs['add'] = lambda self, value: self.append(value)

return type.\_\_new\_\_(cls, name, bases, attrs)

class MyList(list):

\_\_metaclass\_\_ = ListMetaclass # 指示使用ListMetaclass来定制类

当我们写下\_\_metaclass\_\_ = ListMetaclass语句时，魔术就生效了，它指示Python解释器在创建MyList时，要通过ListMetaclass.\_\_new\_\_()来创建，在此，我们可以修改类的定义，比如，加上新的方法，然后，返回修改后的定义。

\_\_new\_\_()方法接收到的参数依次是：

1. 当前准备创建的类的对象；
2. 类的名字；
3. 类继承的父类集合；
4. 类的方法集合。

测试一下MyList是否可以调用add()方法：

>>> L = MyList()

>>> L.add(1)

>>> L

[1]

而普通的list没有add()方法：

>>> l = list()

>>> l.add(1)

Traceback (most recent call last):

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

AttributeError: 'list' object has no attribute 'add'

动态修改有什么意义？直接在MyList定义中写上add()方法不是更简单吗？正常情况下，确实应该直接写，通过metaclass修改纯属变态。

但是，总会遇到需要通过metaclass修改类定义的。ORM就是一个典型的例子。

ORM全称“Object Relational Mapping”，即对象-关系映射，就是把关系数据库的一行映射为一个对象，也就是一个类对应一个表，这样，写代码更简单，不用直接操作SQL语句。

要编写一个ORM框架，所有的类都只能动态定义，因为只有使用者才能根据表的结构定义出对应的类来。

让我们来尝试编写一个ORM框架。

编写底层模块的第一步，就是先把调用接口写出来。比如，使用者如果使用这个ORM框架，想定义一个User类来操作对应的数据库表User，我们期待他写出这样的代码：

class User(Model):

# 定义类的属性到列的映射：

id = IntegerField('id')

name = StringField('username')

email = StringField('email')

password = StringField('password')

# 创建一个实例：

u = User(id=12345, name='Michael', email='test@orm.org', password='my-pwd')

# 保存到数据库：

u.save()

其中，父类Model和属性类型StringField、IntegerField是由ORM框架提供的，剩下的魔术方法比如save()全部由metaclass自动完成。虽然metaclass的编写会比较复杂，但ORM的使用者用起来却异常简单。

现在，我们就按上面的接口来实现该ORM。

首先来定义Field类，它负责保存数据库表的字段名和字段类型：

class Field(object):

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

self.name = name

self.column\_type = column\_type

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

return '<%s:%s>' % (self.\_\_class\_\_.\_\_name\_\_, self.name)

在Field的基础上，进一步定义各种类型的Field，比如StringField，IntegerField等等：

class StringField(Field):

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

super(StringField, self).\_\_init\_\_(name, 'varchar(100)')

class IntegerField(Field):

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

super(IntegerField, self).\_\_init\_\_(name, 'bigint')

下一步，就是编写最复杂的ModelMetaclass了：

class ModelMetaclass(type):

def \_\_new\_\_(cls, name, bases, attrs):

if name=='Model':

return type.\_\_new\_\_(cls, name, bases, attrs)

mappings = dict()

for k, v in attrs.iteritems():

if isinstance(v, Field):

print('Found mapping: %s==>%s' % (k, v))

mappings[k] = v

for k in mappings.iterkeys():

attrs.pop(k)

attrs['\_\_table\_\_'] = name # 假设表名和类名一致

attrs['\_\_mappings\_\_'] = mappings # 保存属性和列的映射关系

return type.\_\_new\_\_(cls, name, bases, attrs)

以及基类Model：

class Model(dict):

\_\_metaclass\_\_ = ModelMetaclass

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

super(Model, self).\_\_init\_\_(\*\*kw)

def \_\_getattr\_\_(self, key):

try:

return self[key]

except KeyError:

raise AttributeError(r"'Model' object has no attribute '%s'" % key)

def \_\_setattr\_\_(self, key, value):

self[key] = value

def save(self):

fields = []

params = []

args = []

for k, v in self.\_\_mappings\_\_.iteritems():

fields.append(v.name)

params.append('?')

args.append(getattr(self, k, None))

sql = 'insert into %s (%s) values (%s)' % (self.\_\_table\_\_, ','.join(fields), ','.join(params))

print('SQL: %s' % sql)

print('ARGS: %s' % str(args))

当用户定义一个class User(Model)时，Python解释器首先在当前类User的定义中查找\_\_metaclass\_\_，如果没有找到，就继续在父类Model中查找\_\_metaclass\_\_，找到了，就使用Model中定义的\_\_metaclass\_\_的ModelMetaclass来创建User类，也就是说，metaclass可以隐式地继承到子类，但子类自己却感觉不到。

在ModelMetaclass中，一共做了几件事情：

1. 排除掉对Model类的修改；
2. 在当前类（比如User）中查找定义的类的所有属性，如果找到一个Field属性，就把它保存到一个\_\_mappings\_\_的dict中，同时从类属性中删除该Field属性，否则，容易造成运行时错误；
3. 把表名保存到\_\_table\_\_中，这里简化为表名默认为类名。

在Model类中，就可以定义各种操作数据库的方法，比如save()，delete()，find()，update等等。

我们实现了save()方法，把一个实例保存到数据库中。因为有表名，属性到字段的映射和属性值的集合，就可以构造出INSERT语句。

编写代码试试：

u = User(id=12345, name='Michael', email='test@orm.org', password='my-pwd')

u.save()

输出如下：

Found model: User

Found mapping: email ==> <StringField:email>

Found mapping: password ==> <StringField:password>

Found mapping: id ==> <IntegerField:uid>

Found mapping: name ==> <StringField:username>

SQL: insert into User (password,email,username,uid) values (?,?,?,?)

ARGS: ['my-pwd', 'test@orm.org', 'Michael', 12345]

可以看到，save()方法已经打印出了可执行的SQL语句，以及参数列表，只需要真正连接到数据库，执行该SQL语句，就可以完成真正的功能。

不到100行代码，我们就通过metaclass实现了一个精简的ORM框架，完整的代码从这里下载：

<https://github.com/michaelliao/learn-python/blob/master/metaclass/simple_orm.py>

最后解释一下类属性和实例属性。直接在class中定义的是类属性：

class Student(object):

name = 'Student'

实例属性必须通过实例来绑定，比如self.name = 'xxx'。来测试一下：

>>> # 创建实例s：

>>> s = Student()

>>> # 打印name属性，因为实例并没有name属性，所以会继续查找class的name属性：

>>> print(s.name)

Student

>>> # 这和调用Student.name是一样的：

>>> print(Student.name)

Student

>>> # 给实例绑定name属性：

>>> s.name = 'Michael'

>>> # 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性：

>>> print(s.name)

Michael

>>> # 但是类属性并未消失，用Student.name仍然可以访问：

>>> print(Student.name)

Student

>>> # 如果删除实例的name属性：

>>> del s.name

>>> # 再次调用s.name，由于实例的name属性没有找到，类的name属性就显示出来了：

>>> print(s.name)

Student

因此，在编写程序的时候，千万不要把实例属性和类属性使用相同的名字。

在我们编写的ORM中，ModelMetaclass会删除掉User类的所有类属性，目的就是避免造成混淆。

## 十三、进程和线程

很多同学都听说过，现代操作系统比如Mac OS X，UNIX，Linux，Windows等，都是支持“多任务”的操作系统。

什么叫“多任务”呢？简单地说，就是操作系统可以同时运行多个任务。打个比方，你一边在用浏览器上网，一边在听MP3，一边在用Word赶作业，这就是多任务，至少同时有3个任务正在运行。还有很多任务悄悄地在后台同时运行着，只是桌面上没有显示而已。

现在，多核CPU已经非常普及了，但是，即使过去的单核CPU，也可以执行多任务。由于CPU执行代码都是顺序执行的，那么，单核CPU是怎么执行多任务的呢？

答案就是操作系统轮流让各个任务交替执行，任务1执行0.01秒，切换到任务2，任务2执行0.01秒，再切换到任务3，执行0.01秒……这样反 复执行下去。表面上看，每个任务都是交替执行的，但是，由于CPU的执行速度实在是太快了，我们感觉就像所有任务都在同时执行一样。

真正的并行执行多任务只能在多核CPU上实现，但是，由于任务数量远远多于CPU的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

对于操作系统来说，一个任务就是一个进程（Process），比如打开一个浏览器就是启动一个浏览器进程，打开一个记事本就启动了一个记事本进程，打开两个记事本就启动了两个记事本进程，打开一个Word就启动了一个Word进程。

有些进程还不止同时干一件事，比如Word，它可以同时进行打字、拼写检查、打印等事情。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（Thread）。

由于每个进程至少要干一件事，所以，一个进程至少有一个线程。当然，像Word这种复杂的进程可以有多个线程，多个线程可以同时执行，多线程的执行 方式和多进程是一样的，也是由操作系统在多个线程之间快速切换，让每个线程都短暂地交替运行，看起来就像同时执行一样。当然，真正地同时执行多线程需要多 核CPU才可能实现。

我们前面编写的所有的Python程序，都是执行单任务的进程，也就是只有一个线程。如果我们要同时执行多个任务怎么办？

有两种解决方案：

一种是启动多个进程，每个进程虽然只有一个线程，但多个进程可以一块执行多个任务。

还有一种方法是启动一个进程，在一个进程内启动多个线程，这样，多个线程也可以一块执行多个任务。

当然还有第三种方法，就是启动多个进程，每个进程再启动多个线程，这样同时执行的任务就更多了，当然这种模型更复杂，实际很少采用。

总结一下就是，多任务的实现有3种方式：

* 多进程模式；
* 多线程模式；
* 多进程+多线程模式。

同时执行多个任务通常各个任务之间并不是没有关联的，而是需要相互通信和协调，有时，任务1必须暂停等待任务2完成后才能继续执行，有时，任务3和任务4又不能同时执行，所以，多进程和多线程的程序的复杂度要远远高于我们前面写的单进程单线程的程序。

因为复杂度高，调试困难，所以，不是迫不得已，我们也不想编写多任务。但是，有很多时候，没有多任务还真不行。想想在电脑上看电影，就必须由一个线 程播放视频，另一个线程播放音频，否则，单线程实现的话就只能先把视频播放完再播放音频，或者先把音频播放完再播放视频，这显然是不行的。

Python既支持多进程，又支持多线程，我们会讨论如何编写这两种多任务程序。

#### 小结

线程是最小的执行单元，而进程由至少一个线程组成。如何调度进程和线程，完全由操作系统决定，程序自己不能决定什么时候执行，执行多长时间。

多进程和多线程的程序涉及到同步、数据共享的问题，编写起来更复杂。

### 1.多进程

要让Python程序实现多进程（multiprocessing），我们先了解操作系统的相关知识。

Unix/Linux操作系统提供了一个fork()系统调用，它非常特殊。普通的函数调用，调用一次，返回一次，但是fork()调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后，分别在父进程和子进程内返回。

子进程永远返回0，而父进程返回子进程的ID。这样做的理由是，一个父进程可以fork出很多子进程，所以，父进程要记下每个子进程的ID，而子进程只需要调用getppid()就可以拿到父进程的ID。

Python的os模块封装了常见的系统调用，其中就包括fork，可以在Python程序中轻松创建子进程：

# multiprocessing.py

import os

print 'Process (%s) start...' % os.getpid()

pid = os.fork()

if pid==0:

print 'I am child process (%s) and my parent is %s.' % (os.getpid(), os.getppid())

else:

print 'I (%s) just created a child process (%s).' % (os.getpid(), pid)

运行结果如下：

Process (876) start...

I (876) just created a child process (877).

I am child process (877) and my parent is 876.

由于Windows没有fork调用，上面的代码在Windows上无法运行。由于Mac系统是基于BSD（Unix的一种）内核，所以，在Mac下运行是没有问题的，推荐大家用Mac学Python！

有了fork调用，一个进程在接到新任务时就可以复制出一个子进程来处理新任务，常见的Apache服务器就是由父进程监听端口，每当有新的http请求时，就fork出子进程来处理新的http请求。

#### multiprocessing

如果你打算编写多进程的服务程序，Unix/Linux无疑是正确的选择。由于Windows没有fork调用，难道在Windows上无法用Python编写多进程的程序？

由于Python是跨平台的，自然也应该提供一个跨平台的多进程支持。multiprocessing模块就是跨平台版本的多进程模块。

multiprocessing模块提供了一个Process类来代表一个进程对象，下面的例子演示了启动一个子进程并等待其结束：

from multiprocessing import Process

import os

# 子进程要执行的代码

def run\_proc(name):

print 'Run child process %s (%s)...' % (name, os.getpid())

if \_\_name\_\_=='\_\_main\_\_':

print 'Parent process %s.' % os.getpid()

p = Process(target=run\_proc, args=('test',))

print 'Process will start.'

p.start()

p.join()

print 'Process end.'

执行结果如下：

Parent process 928.

Process will start.

Run child process test (929)...

Process end.

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个Process实例，用start()方法启动，这样创建进程比fork()还要简单。

join()方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

#### Pool

如果要启动大量的子进程，可以用进程池的方式批量创建子进程：

from multiprocessing import Pool

import os, time, random

def long\_time\_task(name):

print 'Run task %s (%s)...' % (name, os.getpid())

start = time.time()

time.sleep(random.random() \* 3)

end = time.time()

print 'Task %s runs %0.2f seconds.' % (name, (end - start))

if \_\_name\_\_=='\_\_main\_\_':

print 'Parent process %s.' % os.getpid()

p = Pool()

for i in range(5):

p.apply\_async(long\_time\_task, args=(i,))

print 'Waiting for all subprocesses done...'

p.close()

p.join()

print 'All subprocesses done.'

执行结果如下：

Parent process 669.

Waiting for all subprocesses done...

Run task 0 (671)...

Run task 1 (672)...

Run task 2 (673)...

Run task 3 (674)...

Task 2 runs 0.14 seconds.

Run task 4 (673)...

Task 1 runs 0.27 seconds.

Task 3 runs 0.86 seconds.

Task 0 runs 1.41 seconds.

Task 4 runs 1.91 seconds.

All subprocesses done.

代码解读：

对Pool对象调用join()方法会等待所有子进程执行完毕，调用join()之前必须先调用close()，调用close()之后就不能继续添加新的Process了。

请注意输出的结果，task 0，1，2，3是立刻执行的，而task 4要等待前面某个task完成后才执行，这是因为Pool的默认大小在我的电脑上是4，因此，最多同时执行4个进程。这是Pool有意设计的限制，并不是操作系统的限制。如果改成：

p = Pool(5)

就可以同时跑5个进程。

由于Pool的默认大小是CPU的核数，如果你不幸拥有8核CPU，你要提交至少9个子进程才能看到上面的等待效果。

#### 进程间通信

Process之间肯定是需要通信的，操作系统提供了很多机制来实现进程间的通信。Python的multiprocessing模块包装了底层的机制，提供了Queue、Pipes等多种方式来交换数据。

我们以Queue为例，在父进程中创建两个子进程，一个往Queue里写数据，一个从Queue里读数据：

from multiprocessing import Process, Queue

import os, time, random

# 写数据进程执行的代码:

def write(q):

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

print 'Put %s to queue...' % value

q.put(value)

time.sleep(random.random())

# 读数据进程执行的代码:

def read(q):

while True:

value = q.get(True)

print 'Get %s from queue.' % value

if \_\_name\_\_=='\_\_main\_\_':

# 父进程创建Queue，并传给各个子进程：

q = Queue()

pw = Process(target=write, args=(q,))

pr = Process(target=read, args=(q,))

# 启动子进程pw，写入:

pw.start()

# 启动子进程pr，读取:

pr.start()

# 等待pw结束:

pw.join()

# pr进程里是死循环，无法等待其结束，只能强行终止:

pr.terminate()

运行结果如下：

Put A to queue...

Get A from queue.

Put B to queue...

Get B from queue.

Put C to queue...

Get C from queue.

在Unix/Linux下，multiprocessing模块封装了fork()调用，使我们不需要关注fork()的细节。由于Windows没有fork调用，因此，multiprocessing需要“模拟”出fork的效果，父进程所有Python对象都必须通过pickle序列化再传到子进程去，所有，如果multiprocessing在Windows下调用失败了，要先考虑是不是pickle失败了。

#### 小结

在Unix/Linux下，可以使用fork()调用实现多进程。

要实现跨平台的多进程，可以使用multiprocessing模块。

进程间通信是通过Queue、Pipes等实现的。

### 2.多线程

多任务可以由多进程完成，也可以由一个进程内的多线程完成。

我们前面提到了进程是由若干线程组成的，一个进程至少有一个线程。

由于线程是操作系统直接支持的执行单元，因此，高级语言通常都内置多线程的支持，Python也不例外，并且，Python的线程是真正的Posix Thread，而不是模拟出来的线程。

Python的标准库提供了两个模块：thread和threading，thread是低级模块，threading是高级模块，对thread进行了封装。绝大多数情况下，我们只需要使用threading这个高级模块。

启动一个线程就是把一个函数传入并创建Thread实例，然后调用start()开始执行：

import time, threading

# 新线程执行的代码:

def loop():

print 'thread %s is running...' % threading.current\_thread().name

n = 0

while n < 5:

n = n + 1

print 'thread %s >>> %s' % (threading.current\_thread().name, n)

time.sleep(1)

print 'thread %s ended.' % threading.current\_thread().name

print 'thread %s is running...' % threading.current\_thread().name

t = threading.Thread(target=loop, name='LoopThread')

t.start()

t.join()

print 'thread %s ended.' % threading.current\_thread().name

执行结果如下：

thread MainThread is running...

thread LoopThread is running...

thread LoopThread >>> 1

thread LoopThread >>> 2

thread LoopThread >>> 3

thread LoopThread >>> 4

thread LoopThread >>> 5

thread LoopThread ended.

thread MainThread ended.

由于任何进程默认就会启动一个线程，我们把该线程称为主线程，主线程又可以启动新的线程，Python的threading模块有个current\_thread()函数，它永远返回当前线程的实例。主线程实例的名字叫MainThread，子线程的名字在创建时指定，我们用LoopThread命名子线程。名字仅仅在打印时用来显示，完全没有其他意义，如果不起名字Python就自动给线程命名为Thread-1，Thread-2……

#### Lock

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了。

来看看多个线程同时操作一个变量怎么把内容给改乱了：

import time, threading

# 假定这是你的银行存款:

balance = 0

def change\_it(n):

# 先存后取，结果应该为0:

global balance

balance = balance + n

balance = balance - n

def run\_thread(n):

for i in range(100000):

change\_it(n)

t1 = threading.Thread(target=run\_thread, args=(5,))

t2 = threading.Thread(target=run\_thread, args=(8,))

t1.start()

t2.start()

t1.join()

t2.join()

print balance

我们定义了一个共享变量balance，初始值为0，并且启动两个线程，先存后取，理论上结果应该为0，但是，由于线程的调度是由操作系统决定的，当t1、t2交替执行时，只要循环次数足够多，balance的结果就不一定是0了。

原因是因为高级语言的一条语句在CPU执行时是若干条语句，即使一个简单的计算：

balance = balance + n

也分两步：

1. 计算balance + n，存入临时变量中；
2. 将临时变量的值赋给balance。

也就是可以看成：

x = balance + n

balance = x

由于x是局部变量，两个线程各自都有自己的x，当代码正常执行时：

初始值 balance = 0

t1: x1 = balance + 5 # x1 = 0 + 5 = 5

t1: balance = x1 # balance = 5

t1: x1 = balance - 5 # x1 = 5 - 5 = 0

t1: balance = x1 # balance = 0

t2: x2 = balance + 8 # x2 = 0 + 8 = 8

t2: balance = x2 # balance = 8

t2: x2 = balance - 8 # x2 = 8 - 8 = 0

t2: balance = x2 # balance = 0

结果 balance = 0

但是t1和t2是交替运行的，如果操作系统以下面的顺序执行t1、t2：

初始值 balance = 0

t1: x1 = balance + 5 # x1 = 0 + 5 = 5

t2: x2 = balance + 8 # x2 = 0 + 8 = 8

t2: balance = x2 # balance = 8

t1: balance = x1 # balance = 5

t1: x1 = balance - 5 # x1 = 5 - 5 = 0

t1: balance = x1 # balance = 0

t2: x2 = balance - 5 # x2 = 0 - 5 = -5

t2: balance = x2 # balance = -5

结果 balance = -5

究其原因，是因为修改balance需要多条语句，而执行这几条语句时，线程可能中断，从而导致多个线程把同一个对象的内容改乱了。

两个线程同时一存一取，就可能导致余额不对，你肯定不希望你的银行存款莫名其妙地变成了负数，所以，我们必须确保一个线程在修改balance的时候，别的线程一定不能改。

如果我们要确保balance计算正确，就要给change\_it()上一把锁，当某个线程开始执行change\_it()时，我们说，该线程因为获得了锁，因此其他线程不能同时执行change\_it()，只能等待，直到锁被释放后，获得该锁以后才能改。由于锁只有一个，无论多少线程，同一时刻最多只有一个线程持有该锁，所以，不会造成修改的冲突。创建一个锁就是通过threading.Lock()来实现：

balance = 0

lock = threading.Lock()

def run\_thread(n):

for i in range(100000):

# 先要获取锁:

lock.acquire()

try:

# 放心地改吧:

change\_it(n)

finally:

# 改完了一定要释放锁:

lock.release()

当多个线程同时执行lock.acquire()时，只有一个线程能成功地获取锁，然后继续执行代码，其他线程就继续等待直到获得锁为止。

获得锁的线程用完后一定要释放锁，否则那些苦苦等待锁的线程将永远等待下去，成为死线程。所以我们用try...finally来确保锁一定会被释放。

锁的好处就是确保了某段关键代码只能由一个线程从头到尾完整地执行，坏处当然也很多，首先是阻止了多线程并发执行，包含锁的某段代码实际上只能以单 线程模式执行，效率就大大地下降了。其次，由于可以存在多个锁，不同的线程持有不同的锁，并试图获取对方持有的锁时，可能会造成死锁，导致多个线程全部挂 起，既不能执行，也无法结束，只能靠操作系统强制终止。

#### 多核CPU

如果你不幸拥有一个多核CPU，你肯定在想，多核应该可以同时执行多个线程。

如果写一个死循环的话，会出现什么情况呢？

打开Mac OS X的Activity Monitor，或者Windows的Task Manager，都可以监控某个进程的CPU使用率。

我们可以监控到一个死循环线程会100%占用一个CPU。

如果有两个死循环线程，在多核CPU中，可以监控到会占用200%的CPU，也就是占用两个CPU核心。

要想把N核CPU的核心全部跑满，就必须启动N个死循环线程。

试试用Python写个死循环：

import threading, multiprocessing

def loop():

x = 0

while True:

x = x ^ 1

for i in range(multiprocessing.cpu\_count()):

t = threading.Thread(target=loop)

t.start()

启动与CPU核心数量相同的N个线程，在4核CPU上可以监控到CPU占用率仅有160%，也就是使用不到两核。

即使启动100个线程，使用率也就170%左右，仍然不到两核。

但是用C、C++或Java来改写相同的死循环，直接可以把全部核心跑满，4核就跑到400%，8核就跑到800%，为什么Python不行呢？

因为Python的线程虽然是真正的线程，但解释器执行代码时，有一个GIL锁：Global Interpreter Lock，任何Python线程执行前，必须先获得GIL锁，然后，每执行100条字节码，解释器就自动释放GIL锁，让别的线程有机会执行。这个GIL 全局锁实际上把所有线程的执行代码都给上了锁，所以，多线程在Python中只能交替执行，即使100个线程跑在100核CPU上，也只能用到1个核。

GIL是Python解释器设计的历史遗留问题，通常我们用的解释器是官方实现的CPython，要真正利用多核，除非重写一个不带GIL的解释器。

所以，在Python中，可以使用多线程，但不要指望能有效利用多核。如果一定要通过多线程利用多核，那只能通过C扩展来实现，不过这样就失去了Python简单易用的特点。

不过，也不用过于担心，Python虽然不能利用多线程实现多核任务，但可以通过多进程实现多核任务。多个Python进程有各自独立的GIL锁，互不影响。

#### 小结

多线程编程，模型复杂，容易发生冲突，必须用锁加以隔离，同时，又要小心死锁的发生。

Python解释器由于设计时有GIL全局锁，导致了多线程无法利用多核。多线程的并发在Python中就是一个美丽的梦。

### 3.ThreadLocal

在多线程环境下，每个线程都有自己的数据。一个线程使用自己的局部变量比使用全局变量好，因为局部变量只有线程自己能看见，不会影响其他线程，而全局变量的修改必须加锁。

但是局部变量也有问题，就是在函数调用的时候，传递起来很麻烦：

def process\_student(name):

std = Student(name)

# std是局部变量，但是每个函数都要用它，因此必须传进去：

do\_task\_1(std)

do\_task\_2(std)

def do\_task\_1(std):

do\_subtask\_1(std)

do\_subtask\_2(std)

def do\_task\_2(std):

do\_subtask\_2(std)

do\_subtask\_2(std)

每个函数一层一层调用都这么传参数那还得了？用全局变量？也不行，因为每个线程处理不同的Student对象，不能共享。

如果用一个全局dict存放所有的Student对象，然后以thread自身作为key获得线程对应的Student对象如何？

global\_dict = {}

def std\_thread(name):

std = Student(name)

# 把std放到全局变量global\_dict中：

global\_dict[threading.current\_thread()] = std

do\_task\_1()

do\_task\_2()

def do\_task\_1():

# 不传入std，而是根据当前线程查找：

std = global\_dict[threading.current\_thread()]

...

def do\_task\_2():

# 任何函数都可以查找出当前线程的std变量：

std = global\_dict[threading.current\_thread()]

...

这种方式理论上是可行的，它最大的优点是消除了std对象在每层函数中的传递问题，但是，每个函数获取std的代码有点丑。

有没有更简单的方式？

ThreadLocal应运而生，不用查找dict，ThreadLocal帮你自动做这件事：

import threading

# 创建全局ThreadLocal对象:

local\_school = threading.local()

def process\_student():

print 'Hello, %s (in %s)' % (local\_school.student, threading.current\_thread().name)

def process\_thread(name):

# 绑定ThreadLocal的student:

local\_school.student = name

process\_student()

t1 = threading.Thread(target= process\_thread, args=('Alice',), name='Thread-A')

t2 = threading.Thread(target= process\_thread, args=('Bob',), name='Thread-B')

t1.start()

t2.start()

t1.join()

t2.join()

执行结果：

Hello, Alice (in Thread-A)

Hello, Bob (in Thread-B)

全局变量local\_school就是一个ThreadLocal对象，每个Thread对它都可以读写student属性，但互不影响。你可以把local\_school看成全局变量，但每个属性如local\_school.student都是线程的局部变量，可以任意读写而互不干扰，也不用管理锁的问题，ThreadLocal内部会处理。

可以理解为全局变量local\_school是一个dict，不但可以用local\_school.student，还可以绑定其他变量，如local\_school.teacher等等。

ThreadLocal最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求，用户身份信息等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

### 4.进程 vs. 线程

我们介绍了多进程和多线程，这是实现多任务最常用的两种方式。现在，我们来讨论一下这两种方式的优缺点。

首先，要实现多任务，通常我们会设计Master-Worker模式，Master负责分配任务，Worker负责执行任务，因此，多任务环境下，通常是一个Master，多个Worker。

如果用多进程实现Master-Worker，主进程就是Master，其他进程就是Worker。

如果用多线程实现Master-Worker，主线程就是Master，其他线程就是Worker。

多进程模式最大的优点就是稳定性高，因为一个子进程崩溃了，不会影响主进程和其他子进程。（当然主进程挂了所有进程就全挂了，但是Master进程只负责分配任务，挂掉的概率低）著名的Apache最早就是采用多进程模式。

多进程模式的缺点是创建进程的代价大，在Unix/Linux系统下，用fork调用还行，在Windows下创建进程开销巨大。另外，操作系统能同时运行的进程数也是有限的，在内存和CPU的限制下，如果有几千个进程同时运行，操作系统连调度都会成问题。

多线程模式通常比多进程快一点，但是也快不到哪去，而且，多线程模式致命的缺点就是任何一个线程挂掉都可能直接造成整个进程崩溃，因为所有线程共享 进程的内存。在Windows上，如果一个线程执行的代码出了问题，你经常可以看到这样的提示：“该程序执行了非法操作，即将关闭”，其实往往是某个线程 出了问题，但是操作系统会强制结束整个进程。

在Windows下，多线程的效率比多进程要高，所以微软的IIS服务器默认采用多线程模式。由于多线程存在稳定性的问题，IIS的稳定性就不如Apache。为了缓解这个问题，IIS和Apache现在又有多进程+多线程的混合模式，真是把问题越搞越复杂。

#### 线程切换

无论是多进程还是多线程，只要数量一多，效率肯定上不去，为什么呢？

我们打个比方，假设你不幸正在准备中考，每天晚上需要做语文、数学、英语、物理、化学这5科的作业，每项作业耗时1小时。

如果你先花1小时做语文作业，做完了，再花1小时做数学作业，这样，依次全部做完，一共花5小时，这种方式称为单任务模型，或者批处理任务模型。

假设你打算切换到多任务模型，可以先做1分钟语文，再切换到数学作业，做1分钟，再切换到英语，以此类推，只要切换速度足够快，这种方式就和单核CPU执行多任务是一样的了，以幼儿园小朋友的眼光来看，你就正在同时写5科作业。

但是，切换作业是有代价的，比如从语文切到数学，要先收拾桌子上的语文书本、钢笔（这叫保存现场），然后，打开数学课本、找出圆规直尺（这叫准备新 环境），才能开始做数学作业。操作系统在切换进程或者线程时也是一样的，它需要先保存当前执行的现场环境（CPU寄存器状态、内存页等），然后，把新任务 的执行环境准备好（恢复上次的寄存器状态，切换内存页等），才能开始执行。这个切换过程虽然很快，但是也需要耗费时间。如果有几千个任务同时进行，操作系 统可能就主要忙着切换任务，根本没有多少时间去执行任务了，这种情况最常见的就是硬盘狂响，点窗口无反应，系统处于假死状态。

所以，多任务一旦多到一个限度，就会消耗掉系统所有的资源，结果效率急剧下降，所有任务都做不好。

#### 计算密集型 vs. IO密集型

是否采用多任务的第二个考虑是任务的类型。我们可以把任务分为计算密集型和IO密集型。

计算密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力。这种计算密集型任务虽 然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低，所以，要最高效地利用CPU，计算密集型任务同时进行的数 量应当等于CPU的核心数。

计算密集型任务由于主要消耗CPU资源，因此，代码运行效率至关重要。Python这样的脚本语言运行效率很低，完全不适合计算密集型任务。对于计算密集型任务，最好用C语言编写。

第二种任务的类型是IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操 作完成（因为IO的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型 任务，比如Web应用。

IO密集型任务执行期间，99%的时间都花在IO上，花在CPU上的时间很少，因此，用运行速度极快的C语言替换用Python这样运行速度极低的 脚本语言，完全无法提升运行效率。对于IO密集型任务，最合适的语言就是开发效率最高（代码量最少）的语言，脚本语言是首选，C语言最差。

#### 异步IO

考虑到CPU和IO之间巨大的速度差异，一个任务在执行的过程中大部分时间都在等待IO操作，单进程单线程模型会导致别的任务无法并行执行，因此，我们才需要多进程模型或者多线程模型来支持多任务并发执行。

现代操作系统对IO操作已经做了巨大的改进，最大的特点就是支持异步IO。如果充分利用操作系统提供的异步IO支持，就可以用单进程单线程模型来执 行多任务，这种全新的模型称为事件驱动模型，Nginx就是支持异步IO的Web服务器，它在单核CPU上采用单进程模型就可以高效地支持多任务。在多核 CPU上，可以运行多个进程（数量与CPU核心数相同），充分利用多核CPU。由于系统总的进程数量十分有限，因此操作系统调度非常高效。用异步IO编程 模型来实现多任务是一个主要的趋势。

对应到Python语言，单进程的异步编程模型称为协程，有了协程的支持，就可以基于事件驱动编写高效的多任务程序。我们会在后面讨论如何编写协程。

### 5.分布式进程

在Thread和Process中，应当优选Process，因为Process更稳定，而且，Process可以分布到多台机器上，而Thread最多只能分布到同一台机器的多个CPU上。

Python的multiprocessing模块不但支持多进程，其中managers子模块还支持把多进程分布到多台机器上。一个服务进程可以作为调度者，将任务分布到其他多个进程中，依靠网络通信。由于managers模块封装很好，不必了解网络通信的细节，就可以很容易地编写分布式多进程程序。

举个例子：如果我们已经有一个通过Queue通信的多进程程序在同一台机器上运行，现在，由于处理任务的进程任务繁重，希望把发送任务的进程和处理任务的进程分布到两台机器上。怎么用分布式进程实现？

原有的Queue可以继续使用，但是，通过managers模块把Queue通过网络暴露出去，就可以让其他机器的进程访问Queue了。

我们先看服务进程，服务进程负责启动Queue，把Queue注册到网络上，然后往Queue里面写入任务：

# taskmanager.py

import random, time, Queue

from multiprocessing.managers import BaseManager

# 发送任务的队列:

task\_queue = Queue.Queue()

# 接收结果的队列:

result\_queue = Queue.Queue()

# 从BaseManager继承的QueueManager:

class QueueManager(BaseManager):

pass

# 把两个Queue都注册到网络上, callable参数关联了Queue对象:

QueueManager.register('get\_task\_queue', callable=lambda: task\_queue)

QueueManager.register('get\_result\_queue', callable=lambda: result\_queue)

# 绑定端口5000, 设置验证码'abc':

manager = QueueManager(address=('', 5000), authkey='abc')

# 启动Queue:

manager.start()

# 获得通过网络访问的Queue对象:

task = manager.get\_task\_queue()

result = manager.get\_result\_queue()

# 放几个任务进去:

for i in range(10):

n = random.randint(0, 10000)

print('Put task %d...' % n)

task.put(n)

# 从result队列读取结果:

print('Try get results...')

for i in range(10):

r = result.get(timeout=10)

print('Result: %s' % r)

# 关闭:

manager.shutdown()

请注意，当我们在一台机器上写多进程程序时，创建的Queue可以直接拿来用，但是，在分布式多进程环境下，添加任务到Queue不可以直接对原始的task\_queue进行操作，那样就绕过了QueueManager的封装，必须通过manager.get\_task\_queue()获得的Queue接口添加。

然后，在另一台机器上启动任务进程（本机上启动也可以）：

# taskworker.py

import time, sys, Queue

from multiprocessing.managers import BaseManager

# 创建类似的QueueManager:

class QueueManager(BaseManager):

pass

# 由于这个QueueManager只从网络上获取Queue，所以注册时只提供名字:

QueueManager.register('get\_task\_queue')

QueueManager.register('get\_result\_queue')

# 连接到服务器，也就是运行taskmanager.py的机器:

server\_addr = '127.0.0.1'

print('Connect to server %s...' % server\_addr)

# 端口和验证码注意保持与taskmanager.py设置的完全一致:

m = QueueManager(address=(server\_addr, 5000), authkey='abc')

# 从网络连接:

m.connect()

# 获取Queue的对象:

task = m.get\_task\_queue()

result = m.get\_result\_queue()

# 从task队列取任务,并把结果写入result队列:

for i in range(10):

try:

n = task.get(timeout=1)

print('run task %d \* %d...' % (n, n))

r = '%d \* %d = %d' % (n, n, n\*n)

time.sleep(1)

result.put(r)

except Queue.Empty:

print('task queue is empty.')

# 处理结束:

print('worker exit.')

任务进程要通过网络连接到服务进程，所以要指定服务进程的IP。

现在，可以试试分布式进程的工作效果了。先启动taskmanager.py服务进程：

$ python taskmanager.py

Put task 3411...

Put task 1605...

Put task 1398...

Put task 4729...

Put task 5300...

Put task 7471...

Put task 68...

Put task 4219...

Put task 339...

Put task 7866...

Try get results...

taskmanager进程发送完任务后，开始等待result队列的结果。现在启动taskworker.py进程：

$ python taskworker.py 127.0.0.1

Connect to server 127.0.0.1...

run task 3411 \* 3411...

run task 1605 \* 1605...

run task 1398 \* 1398...

run task 4729 \* 4729...

run task 5300 \* 5300...

run task 7471 \* 7471...

run task 68 \* 68...

run task 4219 \* 4219...

run task 339 \* 339...

run task 7866 \* 7866...

worker exit.

taskworker进程结束，在taskmanager进程中会继续打印出结果：

Result: 3411 \* 3411 = 11634921

Result: 1605 \* 1605 = 2576025

Result: 1398 \* 1398 = 1954404

Result: 4729 \* 4729 = 22363441

Result: 5300 \* 5300 = 28090000

Result: 7471 \* 7471 = 55815841

Result: 68 \* 68 = 4624

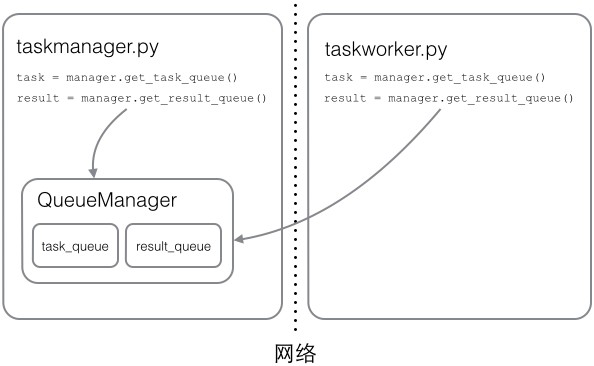
Result: 4219 \* 4219 = 17799961

Result: 339 \* 339 = 114921

Result: 7866 \* 7866 = 61873956

这个简单的Manager/Worker模型有什么用？其实这就是一个简单但真正的分布式计算，把代码稍加改造，启动多个worker，就可以把任务分布到几台甚至几十台机器上，比如把计算n\*n的代码换成发送邮件，就实现了邮件队列的异步发送。

Queue对象存储在哪？注意到taskworker.py中根本没有创建Queue的代码，所以，Queue对象存储在taskmanager.py进程中：



而Queue之所以能通过网络访问，就是通过QueueManager实现的。由于QueueManager管理的不止一个Queue，所以，要给每个Queue的网络调用接口起个名字，比如get\_task\_queue。

authkey有什么用？这是为了保证两台机器正常通信，不被其他机器恶意干扰。如果taskworker.py的authkey和taskmanager.py的authkey不一致，肯定连接不上。

#### 小结

Python的分布式进程接口简单，封装良好，适合需要把繁重任务分布到多台机器的环境下。

注意Queue的作用是用来传递任务和接收结果，每个任务的描述数据量要尽量小。比如发送一个处理日志文件的任务，就不要发送几百兆的日志文件本身，而是发送日志文件存放的完整路径，由Worker进程再去共享的磁盘上读取文件。

## 十四、正则表达式

字符串是编程时涉及到的最多的一种数据结构，对字符串进行操作的需求几乎无处不在。比如判断一个字符串是否是合法的Email地址，虽然可以编程提取@前后的子串，再分别判断是否是单词和域名，但这样做不但麻烦，而且代码难以复用。

正则表达式是一种用来匹配字符串的强有力的武器。它的设计思想是用一种描述性的语言来给字符串定义一个规则，凡是符合规则的字符串，我们就认为它“匹配”了，否则，该字符串就是不合法的。

所以我们判断一个字符串是否是合法的Email的方法是：

1. 创建一个匹配Email的正则表达式；
2. 用该正则表达式去匹配用户的输入来判断是否合法。

因为正则表达式也是用字符串表示的，所以，我们要首先了解如何用字符来描述字符。

在正则表达式中，如果直接给出字符，就是精确匹配。用\d可以匹配一个数字，\w可以匹配一个字母或数字，所以：

* '00\d'可以匹配'007'，但无法匹配'00A'；
* '\d\d\d'可以匹配'010'；
* '\w\w\d'可以匹配'py3'；

.可以匹配任意字符，所以：

* 'py.'可以匹配'pyc'、'pyo'、'py!'等等。

要匹配变长的字符，在正则表达式中，用\*表示任意个字符（包括0个），用+表示至少一个字符，用?表示0个或1个字符，用{n}表示n个字符，用{n,m}表示n-m个字符：

来看一个复杂的例子：\d{3}\s+\d{3,8}。

我们来从左到右解读一下：

1. \d{3}表示匹配3个数字，例如'010'；
2. \s可以匹配一个空格（也包括Tab等空白符），所以\s+表示至少有一个空格，例如匹配' '，' '等；
3. \d{3,8}表示3-8个数字，例如'1234567'。

综合起来，上面的正则表达式可以匹配以任意个空格隔开的带区号的电话号码。

如果要匹配'010-12345'这样的号码呢？由于'-'是特殊字符，在正则表达式中，要用'\'转义，所以，上面的正则是\d{3}\-\d{3,8}。

但是，仍然无法匹配'010 - 12345'，因为带有空格。所以我们需要更复杂的匹配方式。

#### 进阶

要做更精确地匹配，可以用[]表示范围，比如：

* [0-9a-zA-Z\\_]可以匹配一个数字、字母或者下划线；
* [0-9a-zA-Z\\_]+可以匹配至少由一个数字、字母或者下划线组成的字符串，比如'a100'，'0\_Z'，'Py3000'等等；
* [a-zA-Z\\_][0-9a-zA-Z\\_]\*可以匹配由字母或下划线开头，后接任意个由一个数字、字母或者下划线组成的字符串，也就是Python合法的变量；
* [a-zA-Z\\_][0-9a-zA-Z\\_]{0, 19}更精确地限制了变量的长度是1-20个字符（前面1个字符+后面最多19个字符）。

A|B可以匹配A或B，所以[P|p]ython可以匹配'Python'或者'python'。

^表示行的开头，^\d表示必须以数字开头。

$表示行的结束，\d$表示必须以数字结束。

你可能注意到了，py也可以匹配'python'，但是加上^py$就变成了整行匹配，就只能匹配'py'了。

#### re模块

有了准备知识，我们就可以在Python中使用正则表达式了。Python提供re模块，包含所有正则表达式的功能。由于Python的字符串本身也用\转义，所以要特别注意：

s = 'ABC\\-001' # Python的字符串

# 对应的正则表达式字符串变成：

# 'ABC\-001'

因此我们强烈建议使用Python的r前缀，就不用考虑转义的问题了：

s = r'ABC\-001' # Python的字符串

# 对应的正则表达式字符串不变：

# 'ABC\-001'

先看看如何判断正则表达式是否匹配：

>>> import re

>>> re.match(r'^\d{3}\-\d{3,8}$', '010-12345')

<\_sre.SRE\_Match object at 0x1026e18b8>

>>> re.match(r'^\d{3}\-\d{3,8}$', '010 12345')

>>>

match()方法判断是否匹配，如果匹配成功，返回一个Match对象，否则返回None。常见的判断方法就是：

test = '用户输入的字符串'

if re.match(r'正则表达式', test):

print 'ok'

else:

print 'failed'

#### 切分字符串

用正则表达式切分字符串比用固定的字符更灵活，请看正常的切分代码：

>>> 'a b c'.split(' ')

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

嗯，无法识别连续的空格，用正则表达式试试：

>>> re.split(r'\s+', 'a b c')

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

无论多少个空格都可以正常分割。加入,试试：

>>> re.split(r'[\s\,]+', 'a,b, c d')

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

再加入;试试：

>>> re.split(r'[\s\,\;]+', 'a,b;; c d')

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

如果用户输入了一组标签，下次记得用正则表达式来把不规范的输入转化成正确的数组。

#### 分组

除了简单地判断是否匹配之外，正则表达式还有提取子串的强大功能。用()表示的就是要提取的分组（Group）。比如：

^(\d{3})-(\d{3,8})$分别定义了两个组，可以直接从匹配的字符串中提取出区号和本地号码：

>>> m = re.match(r'^(\d{3})-(\d{3,8})$', '010-12345')

>>> m

<\_sre.SRE\_Match object at 0x1026fb3e8>

>>> m.group(0)

'010-12345'

>>> m.group(1)

'010'

>>> m.group(2)

'12345'

如果正则表达式中定义了组，就可以在Match对象上用group()方法提取出子串来。

注意到group(0)永远是原始字符串，group(1)、group(2)……表示第1、2、……个子串。

提取子串非常有用。来看一个更凶残的例子：

>>> t = '19:05:30'

>>> m = re.match(r'^(0[0-9]|1[0-9]|2[0-3]|[0-9])\:(0[0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9]|[0-9])\:(0[0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9]|[0-9])$', t)

>>> m.groups()

('19', '05', '30')

这个正则表达式可以直接识别合法的时间。但是有些时候，用正则表达式也无法做到完全验证，比如识别日期：

'^(0[1-9]|1[0-2]|[0-9])-(0[1-9]|1[0-9]|2[0-9]|3[0-1]|[0-9])$'

对于'2-30'，'4-31'这样的非法日期，用正则还是识别不了，或者说写出来非常困难，这时就需要程序配合识别了。

#### 贪婪匹配

最后需要特别指出的是，正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。举例如下，匹配出数字后面的0：

>>> re.match(r'^(\d+)(0\*)$', '102300').groups()

('102300', '')

由于\d+采用贪婪匹配，直接把后面的0全部匹配了，结果0\*只能匹配空字符串了。

必须让\d+采用非贪婪匹配（也就是尽可能少匹配），才能把后面的0匹配出来，加个?就可以让\d+采用非贪婪匹配：

>>> re.match(r'^(\d+?)(0\*)$', '102300').groups()

('1023', '00')

#### 编译

当我们在Python中使用正则表达式时，re模块内部会干两件事情：

1. 编译正则表达式，如果正则表达式的字符串本身不合法，会报错；
2. 用编译后的正则表达式去匹配字符串。

如果一个正则表达式要重复使用几千次，出于效率的考虑，我们可以预编译该正则表达式，接下来重复使用时就不需要编译这个步骤了，直接匹配：

>>> import re

# 编译:

>>> re\_telephone = re.compile(r'^(\d{3})-(\d{3,8})$')

# 使用：

>>> re\_telephone.match('010-12345').groups()

('010', '12345')

>>> re\_telephone.match('010-8086').groups()

('010', '8086')

编译后生成Regular Expression对象，由于该对象自己包含了正则表达式，所以调用对应的方法时不用给出正则字符串。

#### 小结

正则表达式非常强大，要在短短的一节里讲完是不可能的。要讲清楚正则的所有内容，可以写一本厚厚的书了。如果你经常遇到正则表达式的问题，你可能需要一本正则表达式的参考书。

请尝试写一个验证Email地址的正则表达式。版本一应该可以验证出类似的Email：

someone@gmail.com

bill.gates@microsoft.com

版本二可以验证并提取出带名字的Email地址：

<Tom Paris> tom@voyager.org

## 十五、常用内建模块

Python之所以自称“batteries included”，就是因为内置了许多非常有用的模块，无需额外安装和配置，即可直接使用。

本章将介绍一些常用的内建模块。

### 1.collections

collections是Python内建的一个集合模块，提供了许多有用的集合类。

#### namedtuple

我们知道tuple可以表示不变集合，例如，一个点的二维坐标就可以表示成：

>>> p = (1, 2)

但是，看到(1, 2)，很难看出这个tuple是用来表示一个坐标的。

定义一个class又小题大做了，这时，namedtuple就派上了用场：

>>> from collections import namedtuple

>>> Point = namedtuple('Point', ['x', 'y'])

>>> p = Point(1, 2)

>>> p.x

1

>>> p.y

2

namedtuple是一个函数，它用来创建一个自定义的tuple对象，并且规定了tuple元素的个数，并可以用属性而不是索引来引用tuple的某个元素。

这样一来，我们用namedtuple可以很方便地定义一种数据类型，它具备tuple的不变性，又可以根据属性来引用，使用十分方便。

可以验证创建的Point对象是tuple的一种子类：

>>> isinstance(p, Point)

True

>>> isinstance(p, tuple)

True

类似的，如果要用坐标和半径表示一个圆，也可以用namedtuple定义：

# namedtuple('名称', [属性list]):

Circle = namedtuple('Circle', ['x', 'y', 'r'])

#### deque

使用list存储数据时，按索引访问元素很快，但是插入和删除元素就很慢了，因为list是线性存储，数据量大的时候，插入和删除效率很低。

deque是为了高效实现插入和删除操作的双向列表，适合用于队列和栈：

>>> from collections import deque

>>> q = deque(['a', 'b', 'c'])

>>> q.append('x')

>>> q.appendleft('y')

>>> q

deque(['y', 'a', 'b', 'c', 'x'])

deque除了实现list的append()和pop()外，还支持appendleft()和popleft()，这样就可以非常高效地往头部添加或删除元素。

#### defaultdict

使用dict时，如果引用的Key不存在，就会抛出KeyError。如果希望key不存在时，返回一个默认值，就可以用defaultdict：

>>> from collections import defaultdict

>>> dd = defaultdict(lambda: 'N/A')

>>> dd['key1'] = 'abc'

>>> dd['key1'] # key1存在

'abc'

>>> dd['key2'] # key2不存在，返回默认值

'N/A'

注意默认值是调用函数返回的，而函数在创建defaultdict对象时传入。

除了在Key不存在时返回默认值，defaultdict的其他行为跟dict是完全一样的。

#### OrderedDict

使用dict时，Key是无序的。在对dict做迭代时，我们无法确定Key的顺序。

如果要保持Key的顺序，可以用OrderedDict：

>>> from collections import OrderedDict

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

>>> d # dict的Key是无序的

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

>>> od = OrderedDict([('a', 1), ('b', 2), ('c', 3)])

>>> od # OrderedDict的Key是有序的

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

注意，OrderedDict的Key会按照插入的顺序排列，不是Key本身排序：

>>> od = OrderedDict()

>>> od['z'] = 1

>>> od['y'] = 2

>>> od['x'] = 3

>>> od.keys() # 按照插入的Key的顺序返回

['z', 'y', 'x']

OrderedDict可以实现一个FIFO（先进先出）的dict，当容量超出限制时，先删除最早添加的Key：

from collections import OrderedDict

class LastUpdatedOrderedDict(OrderedDict):

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

super(LastUpdatedOrderedDict, self).\_\_init\_\_()

self.\_capacity = capacity

def \_\_setitem\_\_(self, key, value):

containsKey = 1 if key in self else 0

if len(self) - containsKey >= self.\_capacity:

last = self.popitem(last=False)

print 'remove:', last

if containsKey:

del self[key]

print 'set:', (key, value)

else:

print 'add:', (key, value)

OrderedDict.\_\_setitem\_\_(self, key, value)

#### Counter

Counter是一个简单的计数器，例如，统计字符出现的个数：

>>> from collections import Counter

>>> c = Counter()

>>> for ch in 'programming':

... c[ch] = c[ch] + 1

...

>>> c

Counter({'g': 2, 'm': 2, 'r': 2, 'a': 1, 'i': 1, 'o': 1, 'n': 1, 'p': 1})

Counter实际上也是dict的一个子类，上面的结果可以看出，字符'g'、'm'、'r'各出现了两次，其他字符各出现了一次。

#### 小结

collections模块提供了一些有用的集合类，可以根据需要选用。

### 2.base64

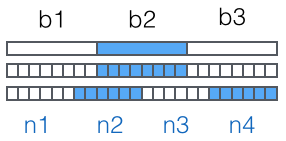
Base64是一种用64个字符来表示任意二进制数据的方法。

用记事本打开exe、jpg、pdf这些文件时，我们都会看到一大堆乱码，因为二进制文件包含很多无法显示和打印的字符，所以，如果要让记事本这样的文本处理软件能处理二进制数据，就需要一个二进制到字符串的转换方法。Base64是一种最常见的二进制编码方法。

Base64的原理很简单，首先，准备一个包含64个字符的数组：

['A', 'B', 'C', ... 'a', 'b', 'c', ... '0', '1', ... '+', '/']

然后，对二进制数据进行处理，每3个字节一组，一共是3x8=24bit，划为4组，每组正好6个bit：



这样我们得到4个数字作为索引，然后查表，获得相应的4个字符，就是编码后的字符串。

所以，Base64编码会把3字节的二进制数据编码为4字节的文本数据，长度增加33%，好处是编码后的文本数据可以在邮件正文、网页等直接显示。

如果要编码的二进制数据不是3的倍数，最后会剩下1个或2个字节怎么办？Base64用\x00字节在末尾补足后，再在编码的末尾加上1个或2个=号，表示补了多少字节，解码的时候，会自动去掉。

Python内置的base64可以直接进行base64的编解码：

>>> import base64

>>> base64.b64encode('binary\x00string')

'YmluYXJ5AHN0cmluZw=='

>>> base64.b64decode('YmluYXJ5AHN0cmluZw==')

'binary\x00string'

由于标准的Base64编码后可能出现字符+和/，在URL中就不能直接作为参数，所以又有一种"url safe"的base64编码，其实就是把字符+和/分别变成-和\_：

>>> base64.b64encode('i\xb7\x1d\xfb\xef\xff')

'abcd++//'

>>> base64.urlsafe\_b64encode('i\xb7\x1d\xfb\xef\xff')

'abcd--\_\_'

>>> base64.urlsafe\_b64decode('abcd--\_\_')

'i\xb7\x1d\xfb\xef\xff'

还可以自己定义64个字符的排列顺序，这样就可以自定义Base64编码，不过，通常情况下完全没有必要。

Base64是一种通过查表的编码方法，不能用于加密，即使使用自定义的编码表也不行。

Base64适用于小段内容的编码，比如数字证书签名、Cookie的内容等。

由于=字符也可能出现在Base64编码中，但=用在URL、Cookie里面会造成歧义，所以，很多Base64编码后会把=去掉：

# 标准Base64:

'abcd' -> 'YWJjZA=='

# 自动去掉=:

'abcd' -> 'YWJjZA'

去掉=后怎么解码呢？因为Base64是把3个字节变为4个字节，所以，Base64编码的长度永远是4的倍数，因此，需要加上=把Base64字符串的长度变为4的倍数，就可以正常解码了。

请写一个能处理去掉=的base64解码函数：

>>> base64.b64decode('YWJjZA==')

'abcd'

>>> base64.b64decode('YWJjZA')

Traceback (most recent call last):

...

TypeError: Incorrect padding

>>> safe\_b64decode('YWJjZA')

'abcd'

#### 小结

Base64是一种任意二进制到文本字符串的编码方法，常用于在URL、Cookie、网页中传输少量二进制数据

### 3.struct

准确地讲，Python没有专门处理字节的数据类型。但由于str既是字符串，又可以表示字节，所以，字节数组＝str。而在C语言中，我们可以很方便地用struct、union来处理字节，以及字节和int，float的转换。

在Python中，比方说要把一个32位无符号整数变成字节，也就是4个长度的str，你得配合位运算符这么写：

>>> n = 10240099

>>> b1 = chr((n & 0xff000000) >> 24)

>>> b2 = chr((n & 0xff0000) >> 16)

>>> b3 = chr((n & 0xff00) >> 8)

>>> b4 = chr(n & 0xff)

>>> s = b1 + b2 + b3 + b4

>>> s

'\x00\x9c@c'

非常麻烦。如果换成浮点数就无能为力了。

好在Python提供了一个struct模块来解决str和其他二进制数据类型的转换。

struct的pack函数把任意数据类型变成字符串：

>>> import struct

>>> struct.pack('>I', 10240099)

'\x00\x9c@c'

pack的第一个参数是处理指令，'>I'的意思是：

>表示字节顺序是big-endian，也就是网络序，I表示4字节无符号整数。

后面的参数个数要和处理指令一致。

unpack把str变成相应的数据类型：

>>> struct.unpack('>IH', '\xf0\xf0\xf0\xf0\x80\x80')

(4042322160, 32896)

根据>IH的说明，后面的str依次变为I：4字节无符号整数和H：2字节无符号整数。

所以，尽管Python不适合编写底层操作字节流的代码，但在对性能要求不高的地方，利用struct就方便多了。

struct模块定义的数据类型可以参考Python官方文档：

<https://docs.python.org/2/library/struct.html#format-characters>

Windows的位图文件（.bmp）是一种非常简单的文件格式，我们来用struct分析一下。

首先找一个bmp文件，没有的话用“画图”画一个。

读入前30个字节来分析：

>>> s = '\x42\x4d\x38\x8c\x0a\x00\x00\x00\x00\x00\x36\x00\x00\x00\x28\x00\x00\x00\x80\x02\x00\x00\x68\x01\x00\x00\x01\x00\x18\x00'

BMP格式采用小端方式存储数据，文件头的结构按顺序如下：

两个字节：'BM'表示Windows位图，'BA'表示OS/2位图； 一个4字节整数：表示位图大小； 一个4字节整数：保留位，始终为0； 一个4字节整数：实际图像的偏移量； 一个4字节整数：Header的字节数； 一个4字节整数：图像宽度； 一个4字节整数：图像高度； 一个2字节整数：始终为1； 一个2字节整数：颜色数。

所以，组合起来用unpack读取：

>>> struct.unpack('<ccIIIIIIHH', s)

('B', 'M', 691256, 0, 54, 40, 640, 360, 1, 24)

结果显示，'B'、'M'说明是Windows位图，位图大小为640x360，颜色数为24。

请编写一个bmpinfo.py，可以检查任意文件是否是位图文件，如果是，打印出图片大小和颜色数。

### 4.hashlib

#### 摘要算法简介

Python的hashlib提供了常见的摘要算法，如MD5，SHA1等等。

什么是摘要算法呢？摘要算法又称哈希算法、散列算法。它通过一个函数，把任意长度的数据转换为一个长度固定的数据串（通常用16进制的字符串表示）。

举个例子，你写了一篇文章，内容是一个字符串'how to use python hashlib - by Michael'，并附上这篇文章的摘要是'2d73d4f15c0db7f5ecb321b6a65e5d6d'。如果有人篡改了你的文章，并发表为'how to use python hashlib - by Bob'，你可以一下子指出Bob篡改了你的文章，因为根据'how to use python hashlib - by Bob'计算出的摘要不同于原始文章的摘要。

可见，摘要算法就是通过摘要函数f()对任意长度的数据data计算出固定长度的摘要digest，目的是为了发现原始数据是否被人篡改过。

摘要算法之所以能指出数据是否被篡改过，就是因为摘要函数是一个单向函数，计算f(data)很容易，但通过digest反推data却非常困难。而且，对原始数据做一个bit的修改，都会导致计算出的摘要完全不同。

我们以常见的摘要算法MD5为例，计算出一个字符串的MD5值：

import hashlib

md5 = hashlib.md5()

md5.update('how to use md5 in python hashlib?')

print md5.hexdigest()

计算结果如下：

d26a53750bc40b38b65a520292f69306

如果数据量很大，可以分块多次调用update()，最后计算的结果是一样的：

md5 = hashlib.md5()

md5.update('how to use md5 in ')

md5.update('python hashlib?')

print md5.hexdigest()

试试改动一个字母，看看计算的结果是否完全不同。

MD5是最常见的摘要算法，速度很快，生成结果是固定的128 bit字节，通常用一个32位的16进制字符串表示。

另一种常见的摘要算法是SHA1，调用SHA1和调用MD5完全类似：

import hashlib

sha1 = hashlib.sha1()

sha1.update('how to use sha1 in ')

sha1.update('python hashlib?')

print sha1.hexdigest()

SHA1的结果是160 bit字节，通常用一个40位的16进制字符串表示。

比SHA1更安全的算法是SHA256和SHA512，不过越安全的算法越慢，而且摘要长度更长。

有没有可能两个不同的数据通过某个摘要算法得到了相同的摘要？完全有可能，因为任何摘要算法都是把无限多的数据集合映射到一个有限的集合中。这种情况称为碰撞，比如Bob试图根据你的摘要反推出一篇文章'how to learn hashlib in python - by Bob'，并且这篇文章的摘要恰好和你的文章完全一致，这种情况也并非不可能出现，但是非常非常困难。

#### 摘要算法应用

摘要算法能应用到什么地方？举个常用例子：

任何允许用户登录的网站都会存储用户登录的用户名和口令。如何存储用户名和口令呢？方法是存到数据库表中：

name | password

--------+----------

michael | 123456

bob | abc999

alice | alice2008

如果以明文保存用户口令，如果数据库泄露，所有用户的口令就落入黑客的手里。此外，网站运维人员是可以访问数据库的，也就是能获取到所有用户的口令。

正确的保存口令的方式是不存储用户的明文口令，而是存储用户口令的摘要，比如MD5：

username | password

---------+---------------------------------

michael | e10adc3949ba59abbe56e057f20f883e

bob | 878ef96e86145580c38c87f0410ad153

alice | 99b1c2188db85afee403b1536010c2c9

当用户登录时，首先计算用户输入的明文口令的MD5，然后和数据库存储的MD5对比，如果一致，说明口令输入正确，如果不一致，口令肯定错误。

练习：根据用户输入的口令，计算出存储在数据库中的MD5口令：

def calc\_md5(password):

pass

存储MD5的好处是即使运维人员能访问数据库，也无法获知用户的明文口令。

练习：设计一个验证用户登录的函数，根据用户输入的口令是否正确，返回True或False：

db = {

'michael': 'e10adc3949ba59abbe56e057f20f883e',

'bob': '878ef96e86145580c38c87f0410ad153',

'alice': '99b1c2188db85afee403b1536010c2c9'

}

def login(user, password):

pass

采用MD5存储口令是否就一定安全呢？也不一定。假设你是一个黑客，已经拿到了存储MD5口令的数据库，如何通过MD5反推用户的明文口令呢？暴力破解费事费力，真正的黑客不会这么干。

考虑这么个情况，很多用户喜欢用123456，888888，password这些简单的口令，于是，黑客可以事先计算出这些常用口令的MD5值，得到一个反推表：

'e10adc3949ba59abbe56e057f20f883e': '123456'

'21218cca77804d2ba1922c33e0151105': '888888'

'5f4dcc3b5aa765d61d8327deb882cf99': 'password'

这样，无需破解，只需要对比数据库的MD5，黑客就获得了使用常用口令的用户账号。

对于用户来讲，当然不要使用过于简单的口令。但是，我们能否在程序设计上对简单口令加强保护呢？

由于常用口令的MD5值很容易被计算出来，所以，要确保存储的用户口令不是那些已经被计算出来的常用口令的MD5，这一方法通过对原始口令加一个复杂字符串来实现，俗称“加盐”：

def calc\_md5(password):

return get\_md5(password + 'the-Salt')

经过Salt处理的MD5口令，只要Salt不被黑客知道，即使用户输入简单口令，也很难通过MD5反推明文口令。

但是如果有两个用户都使用了相同的简单口令比如123456，在数据库中，将存储两条相同的MD5值，这说明这两个用户的口令是一样的。有没有办法让使用相同口令的用户存储不同的MD5呢？

如果假定用户无法修改登录名，就可以通过把登录名作为Salt的一部分来计算MD5，从而实现相同口令的用户也存储不同的MD5。

练习：根据用户输入的登录名和口令模拟用户注册，计算更安全的MD5：

db = {}

def register(username, password):

db[username] = get\_md5(password + username + 'the-Salt')

然后，根据修改后的MD5算法实现用户登录的验证：

def login(username, password):

pass

#### 小结

摘要算法在很多地方都有广泛的应用。要注意摘要算法不是加密算法，不能用于加密（因为无法通过摘要反推明文），只能用于防篡改，但是它的单向计算特性决定了可以在不存储明文口令的情况下验证用户口令。

#### 5.itertools

Python的内建模块itertools提供了非常有用的用于操作迭代对象的函数。

首先，我们看看itertools提供的几个“无限”迭代器：

>>> import itertools

>>> natuals = itertools.count(1)

>>> for n in natuals:

... print n

...

1

2

3

...

因为count()会创建一个无限的迭代器，所以上述代码会打印出自然数序列，根本停不下来，只能按Ctrl+C退出。

cycle()会把传入的一个序列无限重复下去：

>>> import itertools

>>> cs = itertools.cycle('ABC') # 注意字符串也是序列的一种

>>> for c in cs:

... print c

...

'A'

'B'

'C'

'A'

'B'

'C'

...

同样停不下来。

repeat()负责把一个元素无限重复下去，不过如果提供第二个参数就可以限定重复次数：

>>> ns = itertools.repeat('A', 10)

>>> for n in ns:

... print n

...

打印10次'A'

无限序列只有在for迭代时才会无限地迭代下去，如果只是创建了一个迭代对象，它不会事先把无限个元素生成出来，事实上也不可能在内存中创建无限多个元素。

无限序列虽然可以无限迭代下去，但是通常我们会通过takewhile()等函数根据条件判断来截取出一个有限的序列：

>>> natuals = itertools.count(1)

>>> ns = itertools.takewhile(lambda x: x <= 10, natuals)

>>> for n in ns:

... print n

...

打印出1到10

itertools提供的几个迭代器操作函数更加有用：

#### chain()

chain()可以把一组迭代对象串联起来，形成一个更大的迭代器：

for c in chain('ABC', 'XYZ'):

print c

# 迭代效果：'A' 'B' 'C' 'X' 'Y' 'Z'

#### groupby()

groupby()把迭代器中相邻的重复元素挑出来放在一起：

>>> for key, group in itertools.groupby('AAABBBCCAAA'):

... print key, list(group) # 为什么这里要用list()函数呢？

...

A ['A', 'A', 'A']

B ['B', 'B', 'B']

C ['C', 'C']

A ['A', 'A', 'A']

实际上挑选规则是通过函数完成的，只要作用于函数的两个元素返回的值相等，这两个元素就被认为是在一组的，而函数返回值作为组的key。如果我们要忽略大小写分组，就可以让元素'A'和'a'都返回相同的key：

>>> for key, group in itertools.groupby('AaaBBbcCAAa', lambda c: c.upper()):

... print key, list(group)

...

A ['A', 'a', 'a']

B ['B', 'B', 'b']

C ['c', 'C']

A ['A', 'A', 'a']

#### imap()

imap()和map()的区别在于，imap()可以作用于无穷序列，并且，如果两个序列的长度不一致，以短的那个为准。

>>> for x in itertools.imap(lambda x, y: x \* y, [10, 20, 30], itertools.count(1)):

... print x

...

10

40

90

注意imap()返回一个迭代对象，而map()返回list。当你调用map()时，已经计算完毕：

>>> r = map(lambda x: x\*x, [1, 2, 3])

>>> r # r已经计算出来了

[1, 4, 9]

当你调用imap()时，并没有进行任何计算：

>>> r = itertools.imap(lambda x: x\*x, [1, 2, 3])

>>> r

<itertools.imap object at 0x103d3ff90>

# r只是一个迭代对象

必须用for循环对r进行迭代，才会在每次循环过程中计算出下一个元素：

>>> for x in r:

... print x

...

1

4

9

这说明imap()实现了“惰性计算”，也就是在需要获得结果的时候才计算。类似imap()这样能够实现惰性计算的函数就可以处理无限序列：

>>> r = itertools.imap(lambda x: x\*x, itertools.count(1))

>>> for n in itertools.takewhile(lambda x: x<100, r):

... print n

...

结果是什么?

如果把imap()换成map()去处理无限序列会有什么结果？

>>> r = map(lambda x: x\*x, itertools.count(1))

结果是什么?

#### ifilter()

不用多说了，ifilter()就是filter()的惰性实现。

#### 小结

itertools模块提供的全部是处理迭代功能的函数，它们的返回值不是list，而是迭代对象，只有用for循环迭代的时候才真正计算。