



《从零开始学 Python》(第二版)

极客学院出版

前言

Python 是一种面向对象、解释型计算机程序设计语言，由 Guido van Rossum 于 1989 年底发明，第一个公开发行版发行于 1991 年。Python 语法简洁而清晰，具有丰富和强大的类库。它常被昵称为胶水语言，能够把用其他语言制作的各种模块（尤其是 C/C++）很轻松地联结在一起。

Python 在设计上坚持了清晰划一的风格，这使得 Python 成为一门易读、易维护，并且被大量用户所欢迎的、用途广泛的语言。Python 的设计哲学是“优雅”、“明确”、“简单”。

本课程并不是一本教程，而是一本完整的关于 Python 语言学习的书。该书用风趣幽默的语言、丰富的图片、清晰的代码以及完整的实战案例，系统地介绍了 Python 这门当下非常流行的编程语言，是一本非常难得的 Python 学习书籍。该书为同名原书第二版。

适用人群

本书既适用于 Python 的初学者，也适用于已经有一定基础的 Python 开发人员。初学者可以系统的学习 Python，中高级开发人员也能从本书中补充相关知识，加深对 Python 的理解。

学习前提

虽然本书名称为“零基础”学习 Python，但我们还是希望你能有基本的编程思想和简单的数学知识，这对本书的学习非常有帮助。

版本信息

书中演示代码基于以下版本：

语言/框架	版本信息
Python	2.7.6

鸣谢：该书由原作者[老齐](#)授权转载

最后更新时间：2015 年 7 月 30 日

目录

前言	1
第 1 章 预备	6
关于 Python 的故事	7
从小工到专家	12
Python 安装	14
集成开发环境(IDE)	18
第 2 章 基本数据类型	24
数和四则运算	25
除法	31
常用数学函数和运算优先级	37
写一个简单的程序	42
字符串(1)	49
字符串(2)	56
字符串(3)	61
.....	68
字符串(4)	69
字符编码	77
列表(1)	84
列表(2)	91
列表(3)	98
回顾 list 和 str	104
元组	110
字典(1)	113
字典(2)	120

集合(1)	131
集合(2)	138
第 3 章 语句和文件	142
运算符	143
语句(1)	149
语句(2)	156
语句(3)	161
语句(4)	170
语句(5)	178
文件(1)	184
文件(2)	190
迭代	197
练习	203
自省	209
第 4 章 函数	220
函数(1)	221
函数(2)	231
函数(3)	239
函数(4)	247
函数练习	258
第 5 章 类	264
类(1)	265
类(2)	271
类(3)	281
类(4)	290
类(5)	298
多态和封装	308

特殊方法 (1)	314
特殊方法 (2)	321
迭代器	327
生成器	332
第 6 章 错误和异常	339
错误和异常 (1)	340
错误和异常 (2)	346
错误和异常 (3)	352
第 7 章 模块	355
编写模块	356
标准库 (1)	362
标准库 (2)	368
标准库 (3)	374
标准库 (4)	382
标准库 (5)	391
标准库 (6)	401
.....	68
标准库 (8)	422
第三方库	427
第 8 章 保存数据	431
将数据存入文件	432
mysql 数据库 (1)	437
MySQL 数据库 (2)	445
mongodb 数据库 (1)	454
SQLite 数据库	463
电子表格	467
第 9 章 实战	475

实战	475
第 10 章 用Tornado做网站.....	477
为做网站而准备	478
分析 Hello	482
用 tornado 做网站 (1)	490
用 tornado 做网站 (2)	495
用 tornado 做网站 (3)	503
用 tornado 做网站 (4)	510
用 tornado 做网站 (5)	519
用 tornado 做网站 (6)	529
用 tornado 做网站 (7)	533
第 11 章 科学计算	540
为计算做准备	541
Pandas 使用 (1)	547
Pandas 使用 (2)	556
处理股票数据	561
.....	68
第 12 章 结尾	565
如何成为 Python 高手	566

HTML



TP



1

预备



unity



HTML



关于 Python 的故事

我已经在[《零基础学 Python（第一版）》](https://github.com/qiwsir/ITArticles/blob/master/BasicPython/index.md)中写了一个专门讲述 Python 故事的——[唠叨一些关于 Python 的事情](#)——章节，今天再写类似的标题，不打算完全重复原来的，只是把部分认为重要的或者不可或缺的东西复制过来。

越来越火的 Python

在前几年(before 2011)，我跟一些朋友介绍 Python 的时候，看到的常常是一种很诧异的眼神，通常会听到：

“那时什么东西？”
“解释性语言会不会很慢？”
“没听说谁用呀？”
“能像 php, java, c# 那样用来做网站吗？”
“什么？你说的是 pascal？你还在用这个老古董？”
“哦，我听说过，有一些老外在用，不过我们这还没有人用呢。”

时过境迁，现在已经有了很大变化。

2014 年初，我开始写《零基础学 Python》系列，就得到了很多朋友的支持，而且吸引了不少学习 Python 的朋友，特别是在我的那个 QQ 群里面，集中了不少学习者和爱好者，当然也有高手深藏不露。

获得我发布的有关 Python 信息途径：

1. 加入 QQ 群，里面可以跟很多人交流。QQ 群：Code Craft：26913719
2. 关注我的新浪微博，名称是：老齐 Py。地址：<http://weibo.com/qiwsir>
3. 到 [github.com 上直接 follow 我](https://github.com/qiwsir)，名称是：qiwsir。地址：<https://github.com/qiwsir>
4. 经常关注我的网站：www.itdiffer.com

特别是今年（2015 年）一开始，在 QQ 群（26913719）里面，就有朋友说，他在上海找工作，看到好多公司都要有 Python 开发经验的。也有朋友委托我推荐 Python 程序员的。

从我自己的经历中也感受到，不仅仅是国外，国内也如此，用 Python 的领域越来越多，找 Pythoner 的公司和机构也越来越多了。

所以，学习 Python，挺好。

需要什么基础吗

这是很多初学者都会问的一个问题。诚然，在计算机方面的基础越好，对学习任何一门新的编程语言，都是更有利的。如果，你在编程语言的学习上，属于零基础，也不用担心，不管用哪门语言作为学习编程的入门语言，总要有一个开始吧。

就我个人来看，Python 是比较适合作为学习编程的入门语言的。换言之，就是不用担心自己的所谓基础问题。

看我这个课程的标题，就是强调“零基础”的。

不仅我这么认为，美国有不少高校也这么认为，纷纷用 Python 作为编程专业甚至是非编程专业的大学生入门语言。

我跟很多计算机专业的大学生朋友聊过，他们比较痛苦的就是大学用 C 语言作为编程入门语言，学了这个，才知道自己不适合学习编程，因为直到课程完毕，甚至考试通过了（一般是师生一块糊里糊涂地通过），对编程这件事也还是雾里看花的那种感觉。当然，或许你不在此列，一来你有天分，二来你下了功夫。

总而言之，学习 Python，你不用担心基础问题。**特别是在这里学习，我的后续内容，就是从零基础开始的。**

优雅的 Python

Python 号称是优雅的。但是这种说法仁者见仁智者见智。比如经常听到大师们说“数学美”，是不是谁都能体验到呢？不见得吧。

所以，是不是优雅，是不是简单，是不是明确，只有“谁用谁知道”。

不过，我特别喜欢下面这句话：人生苦短，我用 Python。意思就是说，Python 能够提高开发效率，让你短暂的人生能够除了工作之外，还有更多的时间休息、娱乐或者别的什么。

或许有的人不相信，那么也只有“谁用谁知道了”。

跟别的语言比较

“如果你遇到的问题无法用 Python 解决，这个问题也不能用别的语言解决。”——这是我向一些徘徊在 Python 之外的人常说的，是不是有点夸张了呢？

最近看到了一篇文章，[《如果编程语言是女人》](#)，我转载如下(考虑到篇幅所限，所了适当删改，非删减请通过连接查看原文)：



PHP 是你的豆蔻年华的心上人，她是情窦初开的你今年夏天傻乎乎的追求的目标。玩一玩可以，但千万不要投入过深，因为这个女孩有严重的问题。

Ruby 是脚本家族中一个非常漂亮的孩子。第一眼看她，你的心魄就会被她的美丽摄走。她还很有有趣。起初她看起来有点慢，不怎么稳定，但近些年来她已经成熟了很多。

Python 是 Ruby 的一个更懂事的姐姐。她优雅，新潮，成熟。她也许太过优秀。很多小伙都会说“嘿，兄弟，你怎么可能不爱 Python 呢！？”。没错，你喜欢 Python。你把她当成了一个脾气和浪漫都退烧了的 Ruby。

Java 是一个事业成功的女人。很多在她手下干过的人都感觉她的能力跟她的地位并不般配，她更多的是通过技巧打动了中层管理人员。你也许会认为她很有智慧的人，你愿意跟随她。但你要准备好在数年里不断的听到“你用错了接口，你遗漏了一个分号”这样的责备。

C++ 是 Java 的表姐。她在很多地方跟 Java 类似，不同的是她成长于一个天真的年代，不认为需要使用“保护措施”。当然，“保护措施”是指自动内存管理。你以为我指的是什么？

C 是 C++ 的妈妈。对一些头发花白的老程序员说起这个名称，会让他们眼睛一亮，产生无限回忆。

Objective C C 语言家族的另外一个成员。她加入了一个奇怪的教会，不愿意和任何教会之外的人约会。

虽然是娱乐，或许有争议，权当参考吧。

The Zen of Python

这就是著名的《Python 之禅》。

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

感谢 Guido van Rossum

Guido van Rossum 是值得所有 Pythoner 感谢和尊重的，因为他发明了这个优雅的编程语言。他发明 Python 的过程是那么让人称赞和惊叹，显示出牛人的风采。

1989 年的圣诞节期间，吉多·范罗苏姆为了在阿姆斯特丹打发时间，决心开发一个新的脚本解释程序，作为 ABC 语言的一种继承。之所以选中 Python 作为程序的名字，是因为他是一个蒙提·派森的飞行马戏团的爱好者。ABC 是由吉多参加设计的一种教学语言。就吉多本人看来，ABC 这种语言非常优美和强大，是专门为非专业程序员设计的。但是 ABC 语言并没有成功，究其原因，吉多认为是非开放造成的。吉多决心在 Python 中避免这一错误，并取得了非常好的效果，完美结合了 C 和其他一些语言。

这段故事的英文刊载在：<https://www.python.org/doc/essays/foreword/>

新版的设想

我写《零基础学 Python（第二版）》，是承接第一版的，并在第一版基础上，最出比较大量的改进，比如每个章节的标题，现在改为更为直接的描述，而不是用那种文艺范写了，因为这样不仅更明确，而且还能用于以后备查。

此外，我会继续原有的大家认可的风格，兼顾零基础和后续的发展。特别是要在里面穿插如更多的项目例子。

[总目录](#) | [下节：从小工到专家](#)

从小工到专家

这个标题，我借用了一本书的名字——《程序员修炼之道：从小工到专家》——这本书特别推荐阅读。

“从小工到专家”，也是很多开始学习编程的朋友的愿望。如何能实现呢？上面所提到的那本书中，给出了非常好的建议，值得借鉴。

我在这里倒是想到了另外一个问题，也是学习 Python 的朋友给我提出来的：

“书已经看了，书上的代码也运行过了，习题也能解答了，但是还不知如何开发一个真正的应用程序，不知从何处下手。”

此外，我在工作中，也遇到过一些刚刚毕业来求职的大学生，从简历上看，相关专业的考试分数是不错的（我一般是相信那些成绩是真的），但是，一讨论到专业问题，常常出乎我的预料。特别是当他面对真是的工作对象时，表现出来的比成绩单差太多了。

我一般会武断地下一个结论：练的少。

从小工到专家，必经之路就是要多阅读代码，多调试程序。

阅读代码

有句话说的好：“读书破万卷，下笔如有神”。这也适用于编程。阅读别人的代码，是必须的。通过阅读别人的代码，“站在巨人的肩膀上”，让自己眼界开阔，思维充实。

阅读代码的最好地方就是：www.github.com

如果你还没有帐号，请尽快注册，他将是你作为一个优秀程序员的起点。当然了，不要忘记来 follow 我，我的帐号是: qiwsir。

阅读代码最好的一个方法是一边阅读，一边进行必要的注释，这是在梳理自己对别人代码的认识。然后，可以 run 一下，看看效果。当然，还可以按照自己的设想进行必要修改，再 run。这样你就将别人的代码消化吸收了。

调试程序

首先就是要自己动手写程序。“一万小时定律”在编程领域也是成立的，除非你是天才，否则，只有通过“一万小时定律”才能成为天才。

“拳不离手，曲不离口”，小工只有通过勤奋地敲代码才能成为专家。

另外，在调试程序的时候，要善于应用网络，看看类似的问题别人如何解决，不要仅仅局限于自己的思维范围。利用网络就少不了搜索引擎。我特别向那些要想成为专家的小工们说：只有 google 能够帮助你成为专家，其它的搜索引擎，特别是某国内常用的，充其量成为“砖家”，更多的是“砖工”。所以，请用：google.com。

我在本教程中，会陆续想有意成为专家的朋友提供更多有用的网站或者工具。

除了以上两条基本方法之外，成为专家之路还要注意很多呢，不过都是旁枝末节的问题了。以上两条做好，至少在编程上不迷茫了。

[总目录 \(页 0\)](#)

Python 安装

任何高级语言都是需要一个自己的编程环境的，这就好比写字一样，需要有纸和笔，在计算机上写东西，也需要有文字处理软件，比如各种名称的 OFFICE。笔和纸以及 office 软件，就是写东西的硬件或软件，总之，那些文字只能写在那个上边，才能最后成为一篇文章。那么编程也是，要有个什么程序之类的东西，要把程序写到那个上面，才能形成最后类似文章那样的东西。

注：推荐一种非常重要的学习方法

在我这里看文章的零基础朋友，乃至于非零基础的朋友，不要希望在这里学到很多高深的 Python 语言技巧。

“靠，那看你胡扯吗？”

非也。重要的是学会一些方法。比如刚才给大家推荐的“上网 google 一下”，就是非常好的学习方法。互联网的伟大之处，不仅仅在于打打游戏、看看养眼的照片或者各种视频之类的，当然，在某国很长时间互联网等于娱乐网，我衷心希望从读本文的朋友开始，互联网不仅仅是娱乐网，还是知识网和创造网。扯远了，拉回来。在学习过程中，如果遇到一点点疑问，都不要放过，思考一下、尝试一下之后，不管有没有结果，还都要 google 一下。

列位看好了，我上面写的很清楚，是 google 一下，不是让大家去用那个什么度来搜索，那个搜索是专用搜索八卦、假药、以及各种穿着很简单衣服的女孩子照片的。如果你真的要提高自己的技术视野并且专心研究技术问题，请用 google。当然，我知道你在用的时候会遇到困难，做一个要在技术上有点成就的人，一定要学点上网的技术的，你懂得。

什么？你不懂？你的确是我的读者：零基础。那就具体来问我吧，不管是加入 QQ 群还是微博，都可以。

所需要安装的东西，都在这个页面里面：www.Python.org/downloads/

www.python.org 是 python 的官方网站，如果你的英语足够使用，那么自己在这里阅读，可以获得非常多的收获。

在 Python 的下载页面里面，显示出 Python 目前有两大类，一类是 Python3.x.x，另外一类是 Python 2.7.x。可以说，Python3 是未来，它比 Python2.7 有进步。但是，现在，还有很多东西没有完全兼容 Python 3。更何况，如果学了 Python2.7，对于 Python3，也只是某些地方的小变化了。

所以，我这里是用 Python2.7 为例子来讲授的。

Linux 系统的安装

你的计算机是什么操作系统的？自己先弄懂。如果是 Linux 某个发行版，就跟我同道了。并且我恭喜你，因为以后会安装更多的一些 Python 库（模块），在这种操作系统下，操作非常简单，当然，如果是 iOS，也一样，因为都是 UNIX 下的蛋。只是 windows 有点另类了。

不过，没关系，Python 就是跨平台的。

但是，我还是推荐列位，至少在某种意义上讲，用 Linux 操作系统是很好的事情。

我用 Ubuntu。

以 ubutu14.04 为例，一般只要装好了这个操作系统，里面就已经把 Python 安装好了。可能是 Python2.7.6 版本，不过，在我来看，不需要升级，虽然目前最高版本是 Python2.7.9（在 64 位的上面，默认也安装了 Python 3，供使用者选择）。

接下来就在 shell 中输入 Python，如果看到了 >>>，并且显示出 Python 的版本信息，恭喜你，这就进入到了 Python 的交互模式下。

如果非要自己安装。参考下面的操作：

- 到官方网站下载源码。比如：

```
wget http://www.python.org/ftp/python/2.7.8/Python-2.7.8.tgz
```

- 解压源码包

```
tar -zxvf Python-2.7.8.tgz
```

- 编译

```
cd Python-2.7.8 ./configure --prefix=/usr/local # 指定了目录，如果不制定，可以使用默认的，直接运行 ./configure 即可。make&&sudo make install
```

安装好之后，进入 shell，输入 Python，会看到如下：

```
qw@qw-Latitude-E4300:~$ python
Python 2.7.6 (default, Nov 13 2013, 19:24:16) # 后来我升级到 2.7.8 了，就是用后面讲到的源码安装方法
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

我用的是 Python2.7.6，或许你的版本号更高。这些差别就不用纠结了。

windows 系统的安装

到[下载页面里面](#)找到 windows 安装包，下载之，比如下载了这个文件：Python-2.7.8.msi。然后就是不断的“下一步”，即可完成安装。

特别注意，安装完之后，需要检查一下，在环境变量是否有 Python。

如果还不知道什么是 windows 环境变量，以及如何设置。不用担心，请 google 一下，搜索：“windows 环境变量”就能找到如何设置了。

以上搞定，在 cmd 中，输入 Python，得到跟上面类似的结果，就说明已经安装好了。

Mac OS X 系统的安装

其实根本就不用再写怎么安装了，因为用 Mac OS X 的朋友，肯定是高手中的高高手了，至少我一直很敬佩那些用 Mac OS X 并坚持没有更换为 windows 的。麻烦用 Mac OS X 的朋友自己网上搜吧，跟前面 unbutu 差不多。

如果按照以上方法，顺利安装成功，只能说明幸运，无它。如果没有安装成功，这是提高自己的绝佳机会，因为只有遇到问题才能解决问题，才能知道更深刻的道理，不要怕，有 google，它能帮助列为看官解决所有问题。当然，加入 QQ 群或者通过微博，问我也可以。

就一般情况而言，Linux 和 Mac OS x 系统都已经安装了某种 Python 的版本，打开就可以使用。但是 windows 是肯定不安装的。除了可以用上面所说的方法安装，还有一个更省事的方法，就是安装：ActivePython

简单记录一下我的安装方法（我是在 linux 系统中做的）：

1. 获得 root 权限
2. 到上述地址下载某种版本的 Python: wget <https://www.Python.org/ftp/Python/2.7.8/Python-2.7.8.tgz>
3. 解压缩: tar xfz Python-2.7.8.tgz
4. 进入该目录: cd Python-2.7.8
5. 配置: ./configure
6. 在上述文件夹内运行: make，然后运行: make install
7. 祝你幸运
8. 安装完毕

OK!已经安装好之后，马上就可以开始编程了。

最后喊一句在一个编程视频课程广告里面看到的口号，很有启发：“我们程序员，不求通过，但求报错”。

还需要补充说明，本教程使用的是 Python2，虽然跟 Python3 有区别，但是，你不用纠结是 2 还是 3，因为两者区别不是很大，再者，目前工程上应用最多的，还是 Python2，虽然 Python3 是趋势，毕竟需要时间过渡的。很多初学者特别是大学生喜欢纠缠这个问题，实在有点浪费脑细胞了。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

集成开发环境(IDE)

当安装好 Python 之后，其实就已经可以进行开发了。按照惯例，第一行代码总是：Hello World

值得纪念的时刻：Hello world

不管你使用的是什么操作系统，总之肯定能够找到一个地方，运行 Python，进入到交互模式。

像下面一样：

```
Python 2.7.6 (default, Nov 13 2013, 19:24:16)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

在 >>> 后面输入 print "Hello, World"，并按回车。这就是见证奇迹的时刻。

```
>>> print "Hello, World"
Hello, World
```

如果你从来不懂编程，从这一刻起，就跨入了程序员行列；如果已经是程序员，那么就温习一下当初的惊喜吧！

Hello, World 是你用代码向这个世界打招呼了。

每个程序员，都曾经经历过这个伟大时刻，不经历这个伟大时刻的程序员不是伟大的程序员。为了纪念这个伟大时刻，理解其伟大之所在，下面执行分解动作：

说明：在下面的分解动作中，用到了一个符号：#，就是键盘上数字 3 上面的那个井号。这个符号，在 Python 编程中，表示注释。所谓注释，就是在计算机不执行那句话，只是为了说明某行语句表达什么意思，是给计算机前面的人看的。特别提醒，在编程实践中，注释是必须的。请牢记：程序在大多数情况下是给人看的，只是偶尔让计算机执行一下。

```
# 看到“>>>”符号，表示 Python 做好了准备，等待你向她发出指令，让她做什么事情
```

```
>>>
```

```
# print，意思是打印。在这里也是这个意思，是要求 Python 打印什么东西
```

```
>>> print
```

```
#"Hello,World"是打印的内容，注意，变量的双引号，都是英文状态下的。引号不是打印内容，它相当于一个包裹，把打印的内容包起
```

```
>>> print "Hello, World"
```

上面命令执行的结果。Python 接收到你要求她做的事情：打印 Hello,World，于是她就老老实实地执行这个命令，丝毫不走样。

```
Hello, World
```

在 Python 中，如果进入了上面的样式，我们称之为“交互模式”。这是非常有用而且简单的模式，她是我们进行各种学习和有关探索的好方式，随着学习的深入，你将更加觉得她魅力四射。

笑一笑：有一个程序员，自己感觉书法太烂了，于是立志继承光荣文化传统，购买了笔墨纸砚。在某天，开始练习。将纸铺好，拿起笔蘸足墨水，挥毫在纸上写下了两个大字：Hello World

虽然进入了程序员序列，但是，如果程序员用的这个工具，也仅仅是打印 Hello,World，怎能用“伟大”来形容呢？

况且，这个工具也太简陋了？你看美工妹妹用的 Photoshop，行政妹妹用的 word，出纳妹妹用的 Excel，就连坐在老板桌后面的那个家伙还用一个 PPT 播放自己都不相信的新理念呢，难道我们伟大的程序员，就用这么简陋的工具写出旷世代码吗？

当然不是。软件是谁开发的？程序员。程序员肯定会先为自己打造好用的工具，这也叫做“近水楼台先得月”。

IDE 就是程序员的工具。

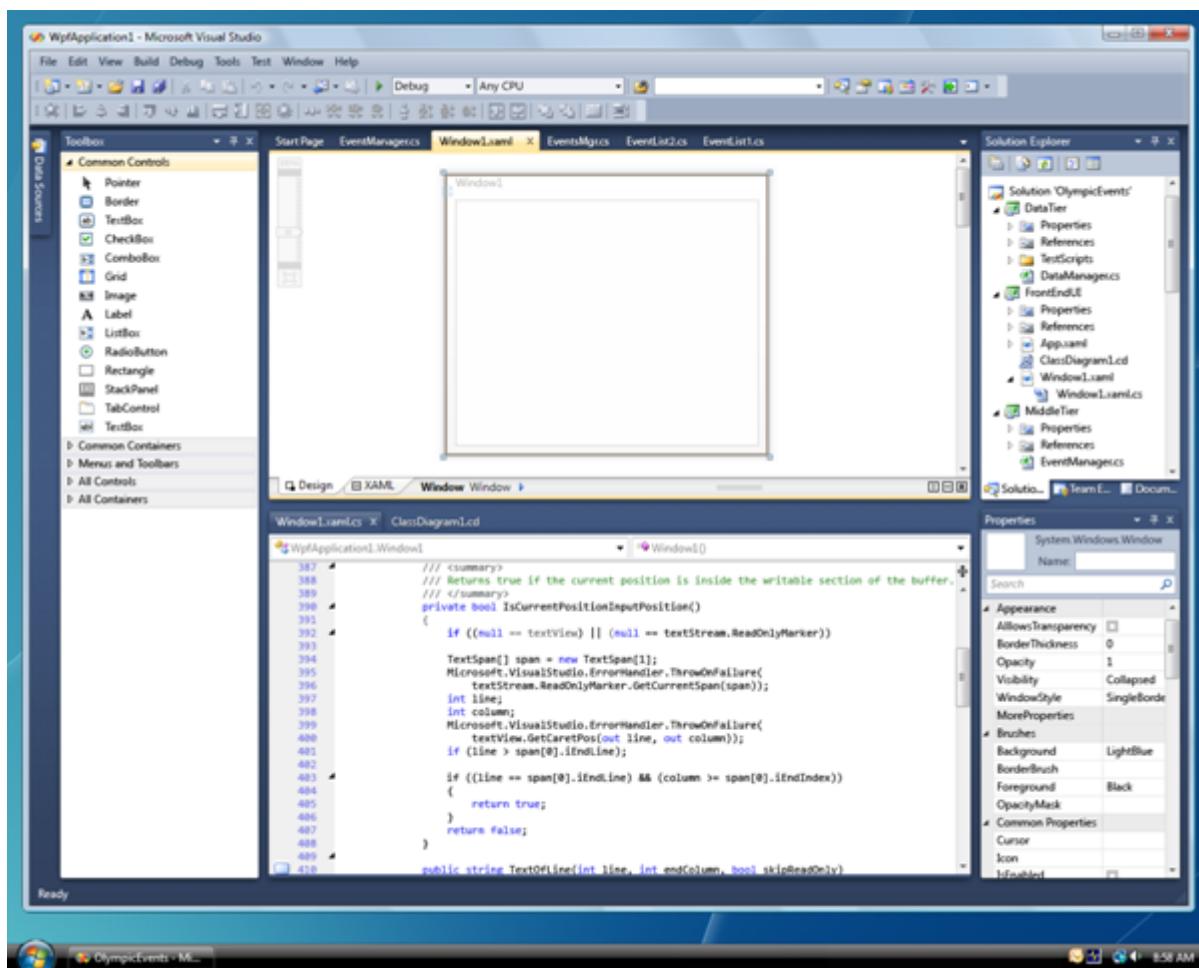
集成开发环境

IDE 的全称是：Integrated Development Environment，简称 IDE，也称为 Integration Design Environment、Integration Debugging Environment，翻译成中文叫做“集成开发环境”，在台湾那边叫做“整合開發環境”。它是一种辅助程序员开发用的应用软件。

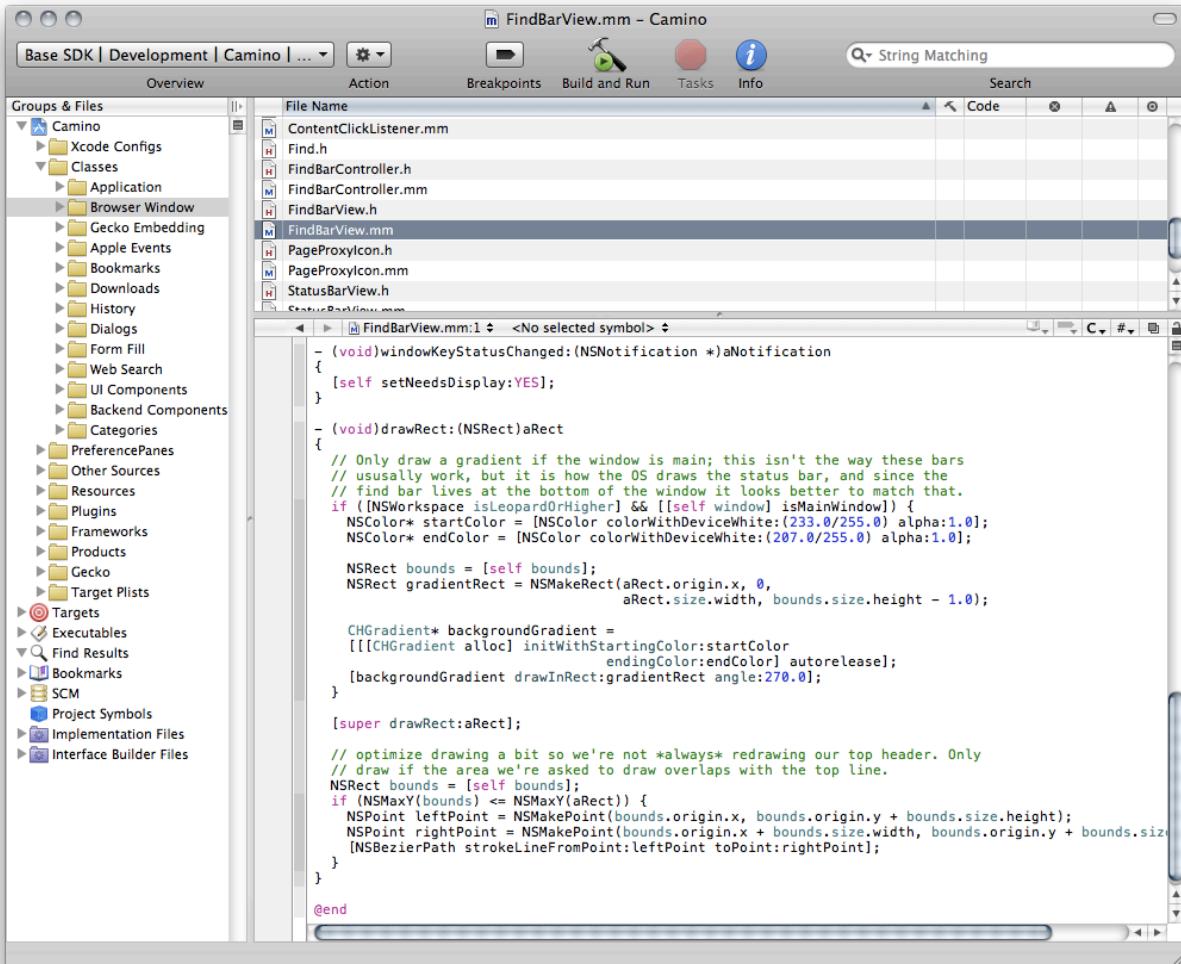
[维基百科](#)这样对 IDE 定义：

IDE 通常包括程式语言编辑器、自动建立工具、通常还包括除错器。有些 IDE 包含编译器 / 直译器，如微软的 Microsoft Visual Studio，有些则不包含，如 Eclipse、SharpDevelop 等，这些 IDE 是通过调用第三方编译器来实现代码的编译工作的。有时 IDE 还会包含版本控制系统和一些可以设计图形用户界面的工具。许多支援物件导向的现代化 IDE 还包括了类别浏览器、物件检视器、物件结构图。虽然目前有一些 IDE 支援多种程式语言（例如 Eclipse、NetBeans、Microsoft Visual Studio），但是一般而言，IDE 主要还是针对特定的程式语言而量身打造（例如 Visual Basic）。

看不懂，没关系，看图，认识一下，混个脸熟就好了。所谓有图有真相。



上面的图显示的是微软的提供的名字叫做 Microsoft Visual Studio 的 IDE。用 C# 进行编程的程序员都用它。



上图是在苹果电脑中出现的名叫 XCode 的 IDE。

要想了解更多 IDE 的信息，推荐阅读维基百科中的词条

- 英文词条：[Integrated development environment](#)
- 中文词条：[集成开发环境](#)

Python 的 IDE

google 一下：Python IDE，会发现，能够进行 Python 编程的 IDE 还真的不少。东西一多，就开始无所适从了。所有，有不少人都问用哪个 IDE 好。可以看看[这个提问](#)，还列出了众多 IDE 的比较。

顺便向列位看客推荐一个非常好的开发相关网站：stackoverflow.com

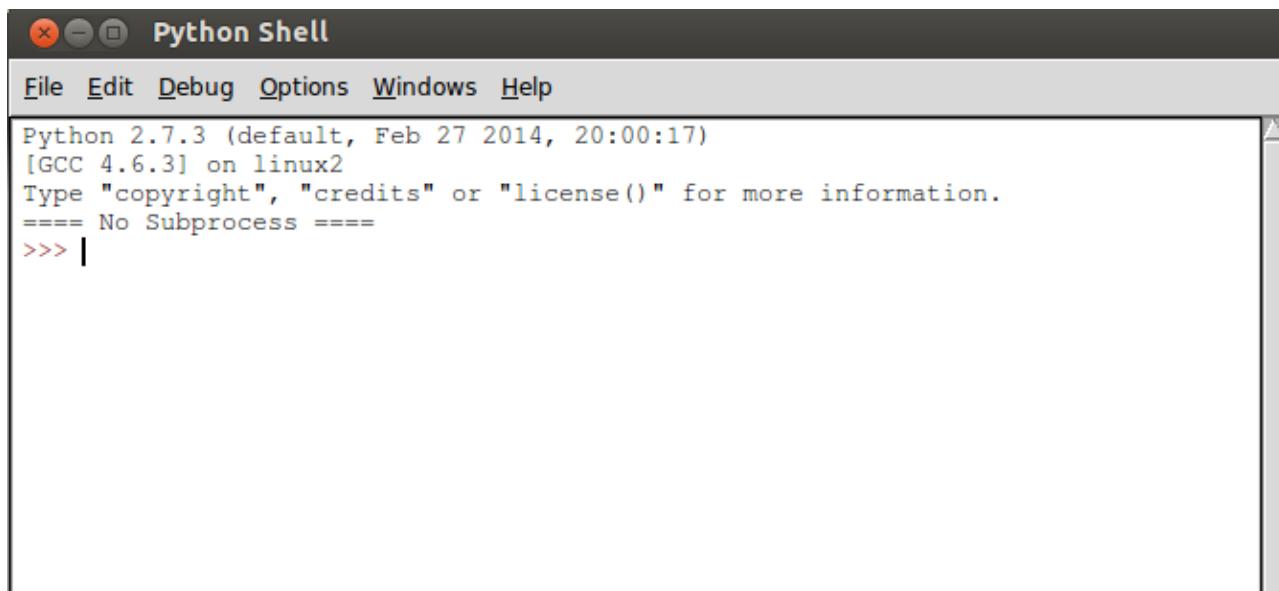
在这里可以提问，可以查看答案。一般如果有问题，先在这里查找，多能找到非常满意的结果，至少有很大启发。

在某国有时候有些地方可能不能访问，需要科学上网。好东西，一定不会让你轻易得到，也不会让任何人都得到。

那么做为零基础的学习者，用什么好呢？

既然是零基础，就别瞎折腾了，就用 Python 自带的 IDLE。原因就是：简单。

Windows 的朋友操作：“开始”菜单→“所有程序”→“Python 2.x”→“IDLE (Python GUI)”来启动 IDLE。启动之后，大概看到这样一个图



注意：看官所看到的界面中显示版本跟这个图不同，因为安装的版本区别。大致模样差不多。

其它操作系统的用户，也都能在找到 idle 这个程序，启动之后，跟上面一样的图。

后面我们所有的编程，就在这里完成了。这就是伟大程序员用的第一个 IDE。

除了这个自带的 IDE，还有很多其它的 IDE，列出来，供喜欢折腾的朋友参考

- PythonWin: 是 Python Win32 Extensions(半官方性质的 Python for win32 增强包)的一部分，也包含在 ActivePython 的 windows 发行版中。如其名字所言，只针对 win32 平台。
- MacPython IDE: MacPythonIDE 是 Python 的 Mac OS 发行版内置的 IDE，可以看作是 PythonWin 的 Mac 对应版本，由 Guido 的哥哥 Just van Rossum 编写。(哥俩都很牛)
- Emacs 和 Vim: Emacs 和 Vim 号称是这个星球上最强大(以及第二强大)的文本编辑器，对于许多程序员来说是万能 IDE 的不二(三?)选择。

- Eclipse + PyDev: Eclipse 是新一代的优秀泛用型 IDE，虽然是基于 Java 技术开发的，但出色的架构使其具有不逊于 Emacs 和 Vim 的可扩展性，现在已经成为了许多程序员最爱的瑞士军刀。

简单列几个，供参考，要找别的 IDE，网上搜一下，五花八门，不少呢。

磨刀不误砍柴工。IDE 已经有了，伟大程序员就要开始从事伟大的编程工作了。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

HTML



TP



基本数据类型



unity



HTML



数和四则运算

一提到计算机，当然现在更多人把她叫做电脑，这两个词都是指 computer。不管什么，只要提到她，普遍都会想到她能够比较快地做加减乘除，甚至乘方开方等。以至于，有的人在口语中区分不开计算机和计算器。

有一篇名为《[计算机前世](#)》的文章，这样讲到：

还是先来看看计算机（computer）这个词是怎么来的。英文学得好的小伙伴看到这货，computer 第一反应好像是：“compute-er”是吧，应该是个什么样的人就对了，就是啊，“做计算的人”。叮咚！恭喜你答对了。最先被命名为 computer 的确实是人。也就是说，电子计算机（与早期的机械计算机）被给予这个名字是因为他们执行的是此前被分配到人的工作。“计算机”原来是工作岗位，它被用来定义一个工种，其任务是执行计算诸如导航表，潮汐图表，天文历书和行星的位置要求的重复计算。从事这个工作的人就是 computer，而且大多是女神！

原文还附有如下图片：



所以，以后要用第三人称来称呼 computer，请用 she（她）。现在你明白为什么程序员中那么多“他”了吧，因为 computer 是“她”。

数

在 Python 中，对数的规定比较简单，基本在小学数学水平即可理解。

那么，做为零基础学习这，也就从计算小学数学题目开始吧。因为从这里开始，数学的基础知识列位肯定过关了。

上面显示的是在交互模式下，如果输入 3，就显示了 3，这样的数称为整数，这个称呼和小学数学一样。

如果输入一个比较大的数，第二个，那么多个 3 组成的一个整数，在 Python 中称之为长整数。为了表示某个数是长整数，Python 会在其末尾显示一个L。其实，现在的 Python 已经能够自动将输入的很大的整数视为长整数了。你不必在这方面进行区别。

第三个，在数学里面称为小数，这里你依然可以这么称呼，不过就像很多编程语言一样，习惯称之为“浮点数”。至于这个名称的由来，也是有点说道的，有兴趣可以 [google](#).

上述举例中，可以说都是无符号（或者说是非负数），如果要表示负数，跟数学中的表示方法一样，前面填上负号即可。

值得注意的是，我们这里说的都是十进制的数。

除了十进制，还有二进制、八进制、十六进制都是在编程中可能用到的，当然用六十进制的时候就比较少了（其实时间记录方式就是典型的六十进制）。

具体每个数字，在 Python 中都是一个对象，比如前面输入的 3，就是一个对象。每个对象，在内存中都有自己的一个地址，这个就是它的身份。

```
>>> id(3)
140574872
>>> id(3.2222222)
140612356
>>> id(3.0)
140612356
>>>
```

用内建函数 `id()` 可以查看每个对象的内存地址，即身份。

内建函数，英文为 built-in Function，读者根据名字也能猜个八九不离十了。不错，就是 Python 中已经定义好的内部函数。

以上三个不同的数字，是三个不同的对象，具有三个不同的内存地址。特别要注意，在数学上，3 和 3.0 是相等的，但是在这里，它们是不同的对象。

用 `id()` 得到的内存地址，是只读的，不能修改。

了解了“身份”，再来看“类型”，也有一个内建函数供使用 `type()`。

```
>>> type(3)
<type 'int'>
>>> type(3.0)
<type 'float'>
>>> type(3.222222)
<type 'float'>
```

用内建函数能够查看对象的类型。, 说明 3 是整数类型 (Integer) ; 则告诉我们那个对象是浮点型 (Floating point real number) 。与 `id()` 的结果类似，`type()` 得到的结果也是只读的。

至于对象的值，在这里就是对象本身了。

看来对象也不难理解。请保持自信，继续。

变量

仅仅写出 3、4、5 是远远不够的，在编程语言中，经常要用到“变量”和“数”（在 Python 中严格来讲是对象）建立一个对应关系。例如：

```
>>> x = 5
>>> x
5
>>> x = 6
>>> x
6
```

在这个例子中，`x = 5` 就是在变量(x)和数(5)之间建立了对应关系，接着又建立了 x 与 6 之间的对应关系。我们可以看到，x 先“是”5，后来“是”6。

在 Python 中，有这样一句话是非常重要的：对象有类型，变量无类型。怎么理解呢？

首先，5、6 都是整数，Python 中为它们取了一个名字，叫做“整数”类型的数据，或者说数据类型是整数，用 int 表示。

当我们在 Python 中写入了 5、6，computer 姑娘就自动在她的内存中某个地方给我们建立这两个对象（对象的定义后面会讲，这里你先用着，逐渐就明晰含义了），就好比建造了两个雕塑，一个是形状似 5，一个形状似 6，这就两个对象，这两个对象的类型就是 int.

那个 x 呢？就好比是一个标签，当 `x = 5` 时，就是将 x 这个标签拴在了 5 上了，通过这个 x，就顺延看到了 5，于是在交互模式中，`>>> x` 输出的结果就是 5，给人的感觉似乎是 x 就是 5，事实是 x 这个标签贴在 5 上面。同样的道理，当 `x = 6` 时，标签就换位置了，贴到 6 上面。

所以，这个标签 x 没有类型之说，它不仅可以贴在整数类型的对象上，还能贴在其它类型的对象上，比如后面会介绍到的 str（字符串）类型的对象等等。

这是 Python 区别于一些语言非常重要的地方。

四则运算

按照下面要求，在交互模式中运行，看看得到的结果和用小学数学知识运算之后得到的结果是否一致

```
>>> 2+5
7
>>> 5-2
3
>>> 10/2
5
>>> 5*2
10
>>> 10/5+1
3
>>> 2*3-4
2
```

上面的运算中，分别涉及到了四个运算符号：加(+)、减(-)、乘(*)、除(/)

另外，我相信看官已经发现了一个重要的公理：

在计算机中，四则运算和小学数学中学习过的四则运算规则是一样的

要不说人是高等动物呢，自己发明的东西，一定要继承自己已经掌握的知识，别跟自己的历史过不去。伟大的科学家们，在当初设计计算机的时候就想到列位现在学习的需要了，一定不能让后世子孙再学新的运算规则，就用小学数学里面的好了。感谢那些科学家先驱者，泽被后世。

下面计算三个算术题，看看结果是什么

- $4 + 2$
- $4.0 + 2$
- $4.0 + 2.0$

看官可能愤怒了，这么简单的题目，就不要劳驾计算机了，太浪费了。

别着急，还是要运算一下，然后看看结果，有没有不一样？要仔细观察哦。

```
>>> 4+2
6
>>> 4.0+2
6.0
>>> 4.0+2.0
6.0
```

不一样的地方是：第一个式子结果是 6，这是一个整数；后面两个是 6.0，这是浮点数。

定义 1：类似 4、-2、129486655、-988654、0 这样形式的数，称之为整数

定义 2：类似 4.0、-2.0、2344.123、3.1415926 这样形式的数，称之为浮点数

对这两个的定义，不用死记硬背，google 一下。记住爱因斯坦说的那句话：书上有的我都不记忆（是这么说的说？好像是，大概意思，反正我也不记忆）。后半句他没说，我补充一下：忘了就 google。

似乎计算机做一些四则运算是不在话下的，但是，有一个问题请你务必注意：在数学中，整数是可以无限大的，但是在计算机中，整数不能无限大。为什么呢？（我推荐你去 google，其实计算机的基本知识中肯定学习过了。）因此，就会有某种情况出现，就是参与运算的数或者运算结果超过了计算机中最大的数了，这种问题称之为“整数溢出问题”。

整数溢出问题

这里有一篇专门讨论这个问题的文章，推荐阅读：[整数溢出](#)

对于其它语言，整数溢出是必须正视的，但是，在 Python 里面，看官就无忧愁了，原因就是 Python 为我们解决了这个问题，请阅读拙文：[\[大整数相乘\]\(https://github.com/qiwsir/algorithm/blob/master/big_int.md"\)](https://github.com/qiwsir/algorithm/blob/master/big_int.md)

ok!看官可以在 IDE 中实验一下大整数相乘。

```
>>> 123456789870987654321122343445567678890098876*1233455667789990099876543332387665443345566  
152278477193527562870044352587576277277562328362032444339019158937017801601677976183816L
```

看官是幸运的，Python 解忧愁，所以，选择学习 Python 就是珍惜光阴了。

上面计算结果的数字最后有一个 L，就表示这个数是一个长整数，不过，看官不用管这点，反正是 Python 为我们搞定了。

在结束本节之前，有两个符号需要看官牢记（不记住也没关系，可以随时 google，只不过记住后使用更方便）

- 整数，用 int 表示，来自单词：integer
- 浮点数，用 float 表示，就是单词：float

可以用一个命令：type(object)来检测一个数是什么类型。

```
>>> type(4)  
<type 'int'> #4 是 int, 整数  
>>> type(5.0)  
<type 'float'> #5.0 是 float, 浮点数  
type(988776544222112233445566778899887766554433221133344455566677788998776543222344556678)  
<type 'long'> # 是长整数，也是一个整数
```

总目录 (页 0)

如果你认为有必要打赏我，请通过支付宝：qjwsir@126.com,不胜感激。

除法

除法啰嗦，不仅是 Python。

整数除以整数

进入 Python 交互模式之后（以后在本教程中，可能不再重复这类的叙述，只要看到>>>，就说明是在交互模式下），练习下面的运算：

```
>>> 2 / 5  
0  
>>> 2.0 / 5  
0.4  
>>> 2 / 5.0  
0.4  
>>> 2.0 / 5.0  
0.4
```

看到没有？麻烦出来了（这是在 Python2.x 中），按照数学运算，以上四个运算结果都应该是 0.4。但我们看到的后三个符合，第一个居然结果是 0。why？

因为，在 Python（严格说是 Python2.x 中，Python3 会有所变化）里面有一个规定，像 2/5 中的除法这样，是要取整（就是去掉小数，但不是四舍五入）。2 除以 5，商是 0（整数），余数是 2（整数）。那么如果用这种形式：2/5，计算结果就是商那个整数。或者可以理解为：整数除以整数，结果是整数（商）。

比如：

```
>>> 5 / 2  
2  
>>> 7 / 2  
3  
>>> 8 / 2  
4
```

注意：得到是商（整数），而不是得到含有小数位的结果再通过“四舍五入”取整。例如：5/2，得到的是商 2，余数 1，最终 $5 / 2 = 2$ 。并不是对 2.5 进行四舍五入。

浮点数与整数相除

这个标题和上面的标题格式不一样，上面的标题是“整数除以整数”，如果按照风格一贯制的要求，本节标题应该是“浮点数除以整数”，但没有，现在是“浮点数与整数相除”，其含义是：

假设： x 除以 y 。其中 x 可能是整数，也可能是浮点数； y 可能是整数，也可能是浮点数。

出结论之前，还是先做实验：

```
>>> 9.0 / 2
4.5
>>> 9 / 2.0
4.5
>>> 9.0 / 2.0
4.5

>>> 8.0 / 2
4.0
>>> 8 / 2.0
4.0
>>> 8.0 / 2.0
4.0
```

归纳，得到规律：不管是被除数还是除数，只要有一个数是浮点数，结果就是浮点数。所以，如果相除的结果有余数，也不会像前面一样了，而是要返回一个浮点数，这就跟在数学上学习的结果一样了。

```
>>> 10.0 / 3
3.3333333333333335
```

这个是不是就有点搞怪了，按照数学知识，应该是 $3.33333\dots$ ，后面是 3 的循环了。那么你的计算机就停不下来了，满屏都是 3。为了避免这个，Python 武断终结了循环，但是，可悲的是没有按照“四舍五入”的原则终止。当然，还会有更奇葩的出现：

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.1 - 0.2
0.0
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17
>>> 0.1 + 0.1 + 0.1 - 0.2
0.10000000000000003
```

越来越糊涂了，为什么 computer 姑娘在计算这么简单的问题上，如此糊涂了呢？不是 computer 姑娘糊涂，她依然冰雪聪明。原因在于十进制和二进制的转换上，computer 姑娘用的是二进制进行计算，上面的例子中，我们输入的是十进制，她就要把十进制的数转化为二进制，然后再计算。但是，在转化中，浮点数转化为二进制，就出问题了。

例如十进制的 0.1，转化为二进制是：0.000110011001100110011001100110011001100110011001100110011...

也就是说，转化为二进制后，不会精确等于十进制的 0.1。同时，计算机存储的位数是有限制的，所以，就出现上述现象了。

这种问题不仅仅是 Python 中有，所有支持浮点数运算的编程语言都会遇到，它不是 Python 的 bug。

明白了问题原因，怎么解决呢？就 Python 的浮点数运算而言，大多数机器上每次计算误差不超过 $2^{**}53$ 分之一。对于大多数任务这已经足够了，但是要在心中记住这不是十进制算法，每个浮点数计算可能会带来一个新的舍入错误。

一般情况下，只要简单地将最终显示的结果用“四舍五入”到所期望的十进制位数，就会得到期望的最终结果。

对于需要非常精确的情况，可以使用 decimal 模块，它实现的十进制运算适合会计方面的应用和高精度要求的应用。另外 fractions 模块支持另外一种形式的运算，它实现的运算基于有理数（因此像 $1/3$ 这样的数字可以精确地表示）。最高要求则可是使用由 SciPy 提供的 Numerical Python 包和其它用于数学和统计学的包。列出这些东西，仅仅是让看官能明白，解决问题的方式很多，后面会用这些中的某些方式解决上述问题。

关于无限循环小数问题，我有一个链接推荐给诸位，它不是想象的那么简单呀。请阅读：[维基百科的词条：0.999...](#)，会不会有深入体会呢？

补充一个资料，供有兴趣的朋友阅读：[浮点数算法：争议和限制](#)

Python 总会要提供多种解决问题的方案的，这是她的风格。

引用模块解决除法——启用轮子

Python 之所以受人欢迎，一个很重要的原因，就是轮子多。这是比喻啦。就好比你要跑的快，怎么办？光天天练习跑步是不行滴，要用轮子。找辆自行车，就快了很多。还嫌不够快，再换电瓶车，再换汽车，再换高铁...反正你可以选择的很多。但是，这些让你跑的快的东西，多数不是你自己造的，是别人造好了，你来用。甚至两条腿也是感谢父母恩赐。正是因为轮子多，可以选择的多，就可以以各种不同速度享受了。

轮子是人类伟大的发明。

Python 就是这样，有各种轮子，我们只需要用。只不过那些轮子在 Python 里面的名字不叫自行车、汽车，叫做“模块”，有人承接别的语言的名称，叫做“类库”、“类”。不管叫什么名字吧。就是别人造好的东西我们拿过来使用。

怎么用？可以通过两种形式用：

- 形式 1：import module-name。import 后面跟空格，然后是模块名称，例如：import os
- 形式 2：from module1 import module11。module1 是一个大模块，里面还有子模块 module11，只想用 module11，就这么写了。

不啰嗦了，实验一个：

```
>>> from __future__ import division
>>> 5 / 2
2.5
>>> 9 / 2
4.5
>>> 9.0 / 2
4.5
>>> 9 / 2.0
4.5
```

注意了，引用了一个模块之后，再做除法，就不管什么情况，都是得到浮点数的结果了。

这就是轮子的力量。

余数

前面计算 $5/2$ 的时候，商是 2，余数是 1

余数怎么得到？在 Python 中（其实大多数语言也都是），用 % 符号来取得两个数相除的余数。

实验下面的操作：

```
>>> 5 % 2
1
>>> 6%4
2
>>> 5.0%2
1.0
```

符号：%，就是要得到两个数（可以是整数，也可以是浮点数）相除的余数。

前面说 Python 有很多人见人爱的轮子（模块），她还有丰富的内建函数，也会帮我们做不少事情。例如函数 `divmod()`

```
>>> divmod(5,2) # 表示 5 除以 2, 返回了商和余数
(2, 1)
>>> divmod(9,2)
(4, 1)
>>> divmod(5.0,2)
(2.0, 1.0)
```

四舍五入

最后一个了，一定要坚持，今天的确有点啰嗦了。要实现四舍五入，很简单，就是内建函数：`round()`

动手试试：

```
>>> round(1.234567,2)
1.23
>>> round(1.234567,3)
1.235
>>> round(10.0/3,4)
3.3333
```

简单吧。越简单的时候，越要小心，当你遇到下面的情况，就有点怀疑了：

```
>>> round(1.2345,3)
1.234      # 应该是: 1.235
>>> round(2.235,2)
2.23      # 应该是: 2.24
```

哈哈，我发现了 Python 的一个 bug，太激动了。

别那么激动，如果真的是 bug，这么明显，是轮不到我的。为什么？具体解释看这里，下面摘录官方文档中的一段话：

Note: The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives `2.67` instead of the expected `2.68`. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See [Floating Point Arithmetic: Issues and Limitations](#) for more information.

原来真的轮不到我。归根到底还是浮点数中的十进制转化为二进制惹的祸。

似乎除法的问题到此要结束了，其实远远没有，不过，做为初学者，至此即可。还留下了很多话题，比如如何处理循环小数问题，我肯定不会让有探索精神的朋友失望的，在我的 github 中有这样一个轮子，如果要深入研究，[可以来这里尝试。](#)

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

常用数学函数和运算优先级

在数学之中，除了加减乘除四则运算之外——这是小学数学——还有其它更多的运算，比如乘方、开方、对数运算等等，要实现这些运算，需要用到 Python 中的一个模块：Math

模块(module)是 Python 中非常重要的东西，你可以把它理解为 Python 的扩展工具。换言之，Python 默认情况下提供了一些可用的东西，但是这些默认情况下提供的还远远不能满足编程实践的需要，于是就有人专门制作了另外一些工具。这些工具被称之为“模块”

任何一个 Pythoner 都可以编写模块，并且把这些模块放到网上供他人来使用。

当安装好 Python 之后，就有一些模块默认安装了，这个称之为“标准库”，“标准库”中的模块不需要安装，就可以直接使用。

如果没有纳入标准库的模块，需要安装之后才能使用。模块的安装方法，我特别推荐使用 pip 来安装。这里仅仅提一下，后面会专门进行讲述，性急的看官可以自己 google。

使用 math 模块

math 模块是标准库中的，所以不用安装，可以直接使用。使用方法是：

```
>>> import math
```

用 import 就将 math 模块引用过来了，下面就可以使用这个模块提供的工具了。比如，要得到圆周率：

```
>>> math.pi
3.141592653589793
```

这个模块都能做哪些事情呢？可以用下面的方法看到：

```
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'd
```

`dir(module)` 是一个非常有用的指令，可以通过它查看任何模块中所包含的工具。从上面的列表中就可以看出，在 math 模块中，可以计算正 $\sin(a), \cos(a), \sqrt{a}$

这些我们称之为函数，也就是在模块 math 中提供了各类计算的函数，比如计算乘方，可以使用 pow 函数。但是，怎么用呢？

Python 是一个非常周到的姑娘，她早就提供了一个命令，让我们来查看每个函数的使用方法。

```
>>> help(math.pow)
```

在交互模式下输入上面的指令，然后回车，看到下面的信息：

```
Help on built-in function pow in module math:
```

```
pow(...)  
pow(x, y)
```

```
Return x**y (x to the power of y).
```

这里展示了 math 模块中的 pow 函数的使用方法和相关说明。

1. 第一行意思是说这里是 math 模块的内建函数 pow 帮助信息（所谓 built-in，称之为内建函数，是说这个函数是 Python 默认就有的）
2. 第三行，表示这个函数的参数，有两个，也是函数的调用方式
3. 第四行，是对函数的说明，返回 `x**y` 的结果，并且在后面解释了 `x**y` 的含义。
4. 最后，按 q 键返回到 Python 交互模式

从上面看到了一个额外的信息，就是 pow 函数和 `x**y` 是等效的，都是计算 x 的 y 次方。

```
>>> 4**2  
16  
>>> math.pow(4,2)  
16.0  
>>> 4*2  
8
```

特别注意，`4**2` 和 `4*2` 是有很大区别的。

用类似的方法，可以查看 math 模块中的任何一个函数的使用方法。

关于“函数”的问题，在这里不做深入阐述，看管姑且按照自己在数学中所学到去理解。后面会有专门研究函数的章节。

下面是几个常用的 math 模块中函数举例，看官可以结合自己调试的进行比照。

```
>>> math.sqrt(9)  
3.0  
>>> math.floor(3.14)  
3.0  
>>> math.floor(3.92)  
3.0  
>>> math.fabs(-2) # 等价于 abs(-2)  
2.0
```

```
>>> abs(-2)
2
>>> math.fmod(5,3) # 等价于 5%3
2.0
>>> 5%3
2
```

几个常见函数

有几个常用的函数，列一下，如果记不住也不要紧，知道有这些就好了，用的时候就 google。

求绝对值

```
>>> abs(10)
10
>>> abs(-10)
10
>>> abs(-1.2)
1.2
```

四舍五入

```
>>> round(1.234)
1.0
>>> round(1.234,2)
1.23

>>> # 如果不清楚这个函数的用法，可以使用下面方法看帮助信息
>>> help(round)
```

Help on built-in function round in module __builtin__:

```
round(...)
    round(number[, ndigits]) -> floating point number

    Round a number to a given precision in decimal digits (default 0 digits).
    This always returns a floating point number. Precision may be negative.
```

运算优先级

从小学数学开始，就研究运算优先级的问题，比如四则运算中“先乘除，后加减”，说明乘法、除法的优先级要高于加减。

对于同级别的，就按照“从左到右”的顺序进行计算。

下面的表格中列出了 Python 中的各种运算的优先级顺序。不过，就一般情况而言，不需要记忆，完全可以按照数学中的去理解，因为人类既然已经发明了数学，在计算机中进行的运算就不需要从新编写一套新规范了，只需要符合数学中的即可。

运算符	描述
lambda	Lambda 表达式
or	布尔“或”
and	布尔“与”
not x	布尔“非”
in, not in	成员测试
is, is not	同一性测试
<, <=, >, >=, !=, ==	比较
	按位或
^	按位异或
&	按位与
<<, >>	移位
+, -	加法与减法
*, /, %	乘法、除法与取余
+x, -x	正负号
~x	按位翻转
**	指数
x.attribute	属性参考
x[index]	下标
x[index:index]	寻址段
f(arguments...)	函数调用
(expression,...)	绑定或元组显示
[expression,...]	列表显示
{key:datum,...}	字典显示
'expression,...'	字符串转换

上面的表格将 Python 中用到的与运算符有关的都列出来了，是按照从低到高的顺序列出的。虽然有很多还不知道是怎么回事，不过先列出来，等以后用到了，还可以回来查看。

最后，要提及的是运算中的绝杀：括号。只要有括号，就先计算括号里面的。这是数学中的共识，无需解释。

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

写一个简单的程序

通过对四则运算的学习，已经初步接触了 Python 中内容，如果看官是零基础的学习者，可能有点迷惑了。难道敲几个命令，然后看到结果，就算编程了？这也不是那些能够自动运行的程序呀？

的确。到目前为止，还不能算编程，只能算是会用一些指令（或者叫做命令）来做点简单的工作。

稍安勿躁，下面就开始编写一个真正的但是简单程序。

程序

下面一段，关于程序的概念，内容来自维基百科：

- 先阅读一段英文的：[computer program and source code](#)，看不懂不要紧，可以跳过去，直接看下一条。

A computer program, or just a program, is a sequence of instructions, written to perform a specified task with a computer.[1] A computer requires programs to function, typically executing the program's instructions in a central processor.[2] The program has an executable form that the computer can use directly to execute the instructions. The same program in its human-readable source code form, from which executable programs are derived (e.g., compiled), enables a programmer to study and develop its algorithms. A collection of computer programs and related data is referred to as the software.

Computer source code is typically written by computer programmers.[3] Source code is written in a programming language that usually follows one of two main paradigms: imperative or declarative programming. Source code may be converted into an executable file (sometimes called an executable program or a binary) by a compiler and later executed by a central processing unit. Alternatively, computer programs may be executed with the aid of an interpreter, or may be embedded directly into hardware.

Computer programs may be ranked along functional lines: system software and application software. Two or more computer programs may run simultaneously on one computer from the perspective of the user, this process being known as multitasking.

- [计算机程序](#)

计算机程序（Computer Program）是指一组指示计算机或其他具有信息处理能力装置每一步动作的指令，通常用某种程序设计语言编写，运行于某种目标体系结构上。打个比方，一个程序就像一个用汉语（程序设计语言）写下的红烧肉菜谱（程序），用于指导懂汉语和烹饪手法的人（体系结构）来做这个菜。

通常，计算机程序要经过编译和链接而成为一种人们不易看清而计算机可解读的格式，然后运行。未经编译就可运行的程序，通常称之为脚本程序（script）。

程序，简而言之，就是指令的集合。但是，有的程序需要编译，有的不需要。Python 编写的程序就不需要，因此她也被称之为解释性语言，编程出来的层序被叫做脚本程序。在有的程序员头脑中，有一种认为“编译型语言比解释性语言高价”的认识。这是错误的。不要认为编译的就好，不编译的就不好；也不要认为编译的就“高端”，不编译的就属于“低端”。有一些做了很多年程序的程序员或者其它什么人，可能会有这样的想法，这是毫无根据的。

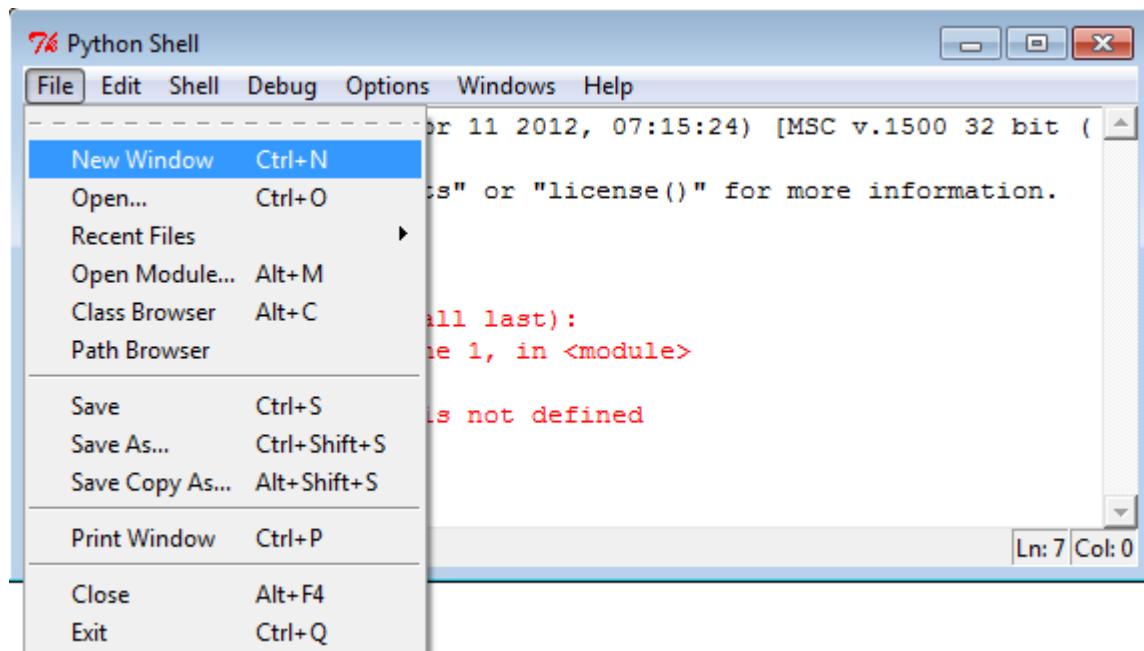
不争论。用得妙就是好。

用 IDLE 的编程环境

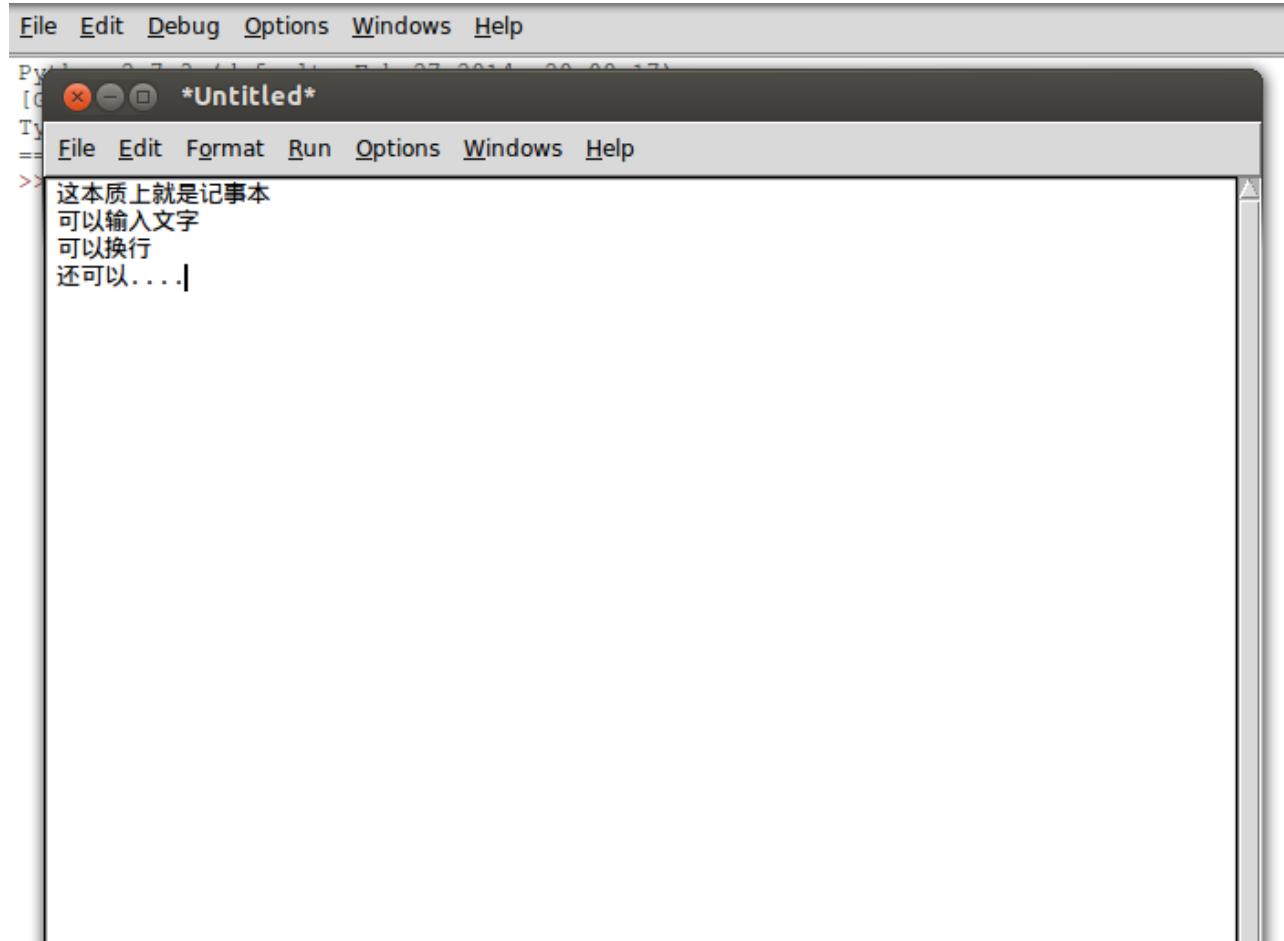
能够写 Python 程序的工具很多，比如记事本就可以。当然，很多人总希望能用一个专门的编程工具，Python 里面自带了一个，作为简单应用是足够了。另外，可以根据自己的喜好用其它的工具，比如我用的是 vim，有不少人也用 eclipse，还有 notepad++，等等。软件领域为编程提供了丰富多彩的工具。

以 Python 默认的 IDE 为例，如下所示：

操作: File->New window



这样，就出现了一个新的操作界面，在这个界面里面，看不到用于输入指令的提示符：>>>，这个界面有点像记事本。说对了，本质上就是一个记事本，只能输入文本，不能直接在里面贴图片。



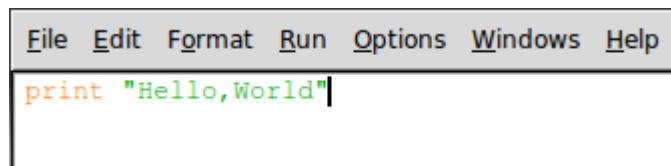
写两个大字：Hello,World

Hello,World.是面向世界的标志，所以，写任何程序，第一句一定要写这个，因为程序员是面向世界的，绝对不畏缩在某个局域网内，所以，所以看官要会科学上网，才能真正与世界 Hello。

直接上代码，就这么一行即可。

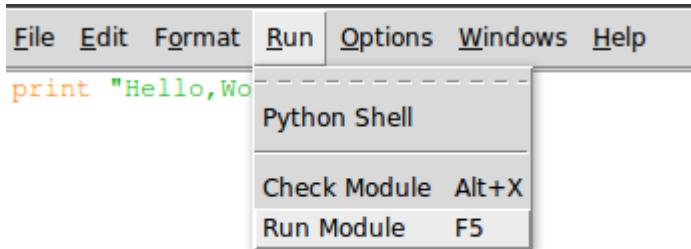
```
print "Hello,World"
```

如下图的样式



前面说过了，程序就是指令的集合，现在，这个程序里面，就一条指令。一条指令也可以成为集合。

注意观察，菜单上有一个 RUN，点击这个菜单，在下拉列表里面选择 Run Module。



会弹出对话框，要求把这个文件保存，这就比较简单了，保存到一个位置，看官一定要记住这个位置，并且取个文件名，文件名是以.py 为扩展名的。

都做好之后，点击确定按钮，就会发现在另外一个带有 >>> 的界面中，就自动出来了 Hello,World 两个大字。

成功了吗？成功了也别兴奋，因为还没有到庆祝的时候。

在这种情况下，我们依然是在 IDLE 的环境中实现了刚才那段程序的自动执行，如果脱离这个环境呢？

下面就关闭 IDLE，打开 shell(如果看官在使用苹果的 Mac OS 操作系统或者某种 linux 发行版的操作系统，比如我使用的是 ubuntu)，或者打开 cmd(windows 操作系统的用户，特别提醒用 windows 的用户，使用 windows 不是你的错，错就错在你只会使用鼠标点来点去，而不想也不会使用命令，更不想也不会使用 linux 的命令，还梦想成为优秀程序员。)，通过命令的方式，进入到你保存刚才的文件目录。

下图是我保存那个文件的地址，我把那个文件命名为 105.py，并保存在一个文件夹中。

```
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ ls
105.py
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$
```

然后在这个 shell 里面，输入：Python 105.py

上面这句话的含义就是告诉计算机，给我运行一个 Python 语言编写的程序，那个程序文件的名称是 105.py

我的计算机我做主。于是它给我乖乖地执行了这条命令。如下图：

```
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ ls
105.py
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ python 105.py
Hello,World
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ |
```

还在沉默？可以欢呼了，德国队 7:1 胜巴西队，列看官中，不管是德国队还是巴西队的粉丝，都可以欢呼，因为你在程序员道路上迈出了伟大的第二步（什么迈出的第一步？）。顺便预测一下，本届世界杯最终冠军应该是：中国队。（还有这么扯的吗？）

解一道题目

请计算： $19+2^4-8/2$

代码如下：

```
#!/usr/bin/env python
#coding:utf-8

#####
请计算：
19+2^4-8/2
#####

a = 19+2^4-8/2
print a
```

提醒初学者，别复制这段代码，而是要一个字一个字的敲进去。然后保存(我保存的文件名是:105-1.py)。

在 shell 或者 cmd 中，执行：Python (文件名.py)

执行结果如下图：

```
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ python 105-1.py
23
```

好像还是比较简单。

下面对这个简单程序进行一一解释。

```
#!/usr/bin/env python
```

这一行是必须写的，它能够引导程序找到 Python 的解析器，也就是说，不管你这个文件保存在什么地方，这个程序都能执行，而不用制定 Python 的安装路径。

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

这一行是告诉 Python，本程序采用的编码格式是 utf-8，什么是编码？什么是 utf-8？这是一个比较复杂且有历史的问题，此处暂不讨论。只有有了上面这句话，后面的程序中才能写汉字，否则就会报错了。看官可以把你的程序中的这行删掉，看看什么结果？

```
#####
请计算:  
19+2*4-8/2  
#####
```

这一行是让人看的，计算机看不懂。在 Python 程序中（别的编程语言也是如此），要写所谓的注释，就是对程序或者某段语句的说明文字，这些文字在计算机执行程序的时候，被计算机姑娘忽略，但是，注释又是必不可少的，正如前面说的那样，程序在大多数情况下是给人看的。注释就是帮助人理解程序的。

写注释的方式有两种，一种是单行注释，用 `#` 开头，另外一种是多行注释，用一对 `" "` 包裹起来。比如：

```
#####
请计算:  
19+2*4-8/2  
#####
```

用 `#` 开头的注释，可以像下面这样来写：

```
# 请计算: 19+2*4-8/2
```

这种注释通常写在程序中的某个位置，比如某个语句的前面或者后面。计算机也会忽略这种注释的内容，只是给人看的。以 `#` 开头的注释，会在后面的编程中大量使用。

一般在程序的开头部分，都要写点东西，主要是告诉别人这个程序是用来做什么的。

```
a = 19+2*4-8/2
```

所谓语句，就是告诉程序要做什么事情。程序就是有各种各样的语句组成的。这条语句，又有一个名字，叫做复制语句。`19+2*4-8/2` 是一个表达式，最后要计算出一个结果，这个结果就是一个对象（又遇到了对象这个术语。在某些地方的方言中，把配偶、男女朋友也称之为对象，“对象”是一个应用很广泛的术语）。`=` 不要理解为数学中的等号，它的作用不是等于，而是完成赋值语句中“赋值”的功能。`a` 就是变量。这样就完成了一个赋值过程。

语句和表达式的区别：“表达式就是某件事”，“语句是做某件事”。

```
print a
```

这还是一个语句，称之为 `print` 语句，就是要打印出 `a` 的值（这种说法不是非常非常严格，但是通常总这么说。按照严格的说法，是打印变量 `a` 做对应的对象的值。嫌这种说法啰嗦，就直接说打印 `a` 的值）。

是不是在为看到自己写的第一个程序而欣慰呢？

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

字符串(1)

如果对自然语言分类，有很多种分法，比如英语、法语、汉语等，这种分法是最常见的。在语言学里面，也有对语言的分类方法，比如什么什么语系之类的。我这里提出一种分法，这种分法尚未得到广大人民群众和研究者的广泛认同，但是，我相信那句“真理是掌握在少数人的手里”，至少在这里可以用来给自己壮壮胆。

我的分法：一种是语言中的两个元素（比如两个字）拼接在一起，出来一个新的元素（比如新的字）；另外一种是两个元素拼接在一起，只是得到这两个元素的并列显示。比如“好”和“人”，两个元素拼接在一起是“好人”，而3和5拼接（就是整数求和）在一起是8，如果你认为是35，那就属于第二类了。

把我的这种分法抽象一下：

- 一种是： $\Delta + \square = \circ$
- 另外一种是： $\Delta + \square = \Delta \square$

我们的语言中，离不开以上两类，不是第一类就是第二类。

太天才了。请鼓掌。

字符串

在我洋洋自得的时候，我google了一下，才发现，自己没那么高明，看[维基百科的字符串词条](#)是这么说的：

字符串（String），是由零个或多个字符组成的有限串行。一般记为 $s=a[1]a[2]\dots a[n]$ 。

看到维基百科的伟大了吧，它已经把我所设想的一种情况取了一个形象的名称，叫做字符串，本质上就是一串字符。

根据这个定义，在前面两次让一个程序员感到伟大的"Hello,World"，就是一个字符串。或者说不管用英文还是中文还是别的某种文，写出来的文字都可以做为字符串对待，当然，里面的特殊符号，也是可以做为字符串的，比如空格等。

严格地说，在Python中的字符串是一种对象类型，这种类型用str表示，通常单引号"或者双引号""包裹起来。

字符串和前面讲过的数字一样，都是对象的类型，或者说都是值。当然，表示方式还是有区别的。

```
>>> "I love Python."
'I love Python.'
```

```
>>> 'I LOVE PYTHON.'
'I LOVE PYTHON.'
```

从这两个例子中可以看出来，不论使用单引号还是双引号，结果都是一样的。

```
>>> 250
250
>>> type(250)
<type 'int'>

>>> "250"
'250'
>>> type("250")
<type 'str'>
```

仔细观察上面的区别，同样是 250，一个没有放在引号里面，一个放在了引号里面，用 `type()` 函数来检验一下，发现它们居然是两种不同的对象类型，前者是 `int` 类型，后者则是 `str` 类型，即字符串类型。所以，请大家务必注意，不是所有数字都是 `int` (or `float`)，必须要看看，它在什么地方，如果在引号里面，就是字符串了。如果搞不清楚是什么类型，就让 `type()` 来帮忙搞定。

操练一下字符串吧。

```
>>> print "good good study, day day up"
good good study, day day up
>>> print "----good---study---day----up"
----good---study---day----up
```

在 `print` 后面，打印的都是字符串。注意，是双引号里面的，引号不是字符串的组成部分。它是在告诉计算机，它里面包裹着的是一个字符串。

爱思考的看官肯定发现上面这句话有问题了。如果我要把下面这句话看做一个字符串，应该怎么做？

```
What's your name?
```

这个问题非常好，因为在这句话中有一个单引号，如果直接在交互模式中像上面那样输入，就会这样：

```
>>> 'What's your name?
File "<stdin>", line 1
  'What's your name?'
 ^
SyntaxError: invalid syntax
```

出现了 `SyntaxError` (语法错误) 引导的提示，这是在告诉我们这里存在错误，错误的类型就是 `SyntaxError`，后面是对这种错误的解释 “`invalid syntax`” (无效的语法)。特别注意，错误提示的上面，有一个 ^ 符号，直接只着一个单引号，不用多说，你也能猜测出，大概在告诉我们，可能是这里出现错误了。

在 python 中，这一点是非常友好的，如果语句存在错误，就会将错误输出来，供程序员改正参考。当然，错误来源有时候比较复杂，需要根据经验和知识进行修改。还有一种修改错误的好办法，就是讲错误提示放到 google 中搜索。

上面那个值的错误原因是什么呢？仔细观察，发现那句话中事实上有三个单引号，本来一对单引号之间包裹的是一个字符串，现在出现了三个（一对半）单引号，computer 姑娘迷茫了，她不知道单引号包裹的到底是谁。于是报错。

解决方法一：双引号包裹单引号

```
>>> "What's your name?"  
"What's your name?"
```

用双引号来包裹，双引号里面允许出现单引号。其实，反过来，单引号里面也可以包裹双引号。这个可以笼统地成为二者的嵌套。

解决方法二：使用转义符

所谓转义，就是让某个符号不在表示某个含义，而是表示另外一个含义。转义符的作用就是它能够转变符号的含义。在 Python 中，用 \ 作为转义符（其实很多语言，只要有转义符的，都是用这个符号）。

```
>>> 'What\'s your name?'  
"What's your name?"
```

是不是看到转义符 \ 的作用了。

本来单引号表示包括字符串，它不是字符串一部分，但是如果前面有转义符，那么它就失去了原来的含义，转化为字符串的一部分，相当于一个特殊字符了。

变量和字符串

前面讲过变量无类型，对象有类型了，比如在数字中：

```
>>> a = 5  
>>> a  
5
```

其本质含义是变量 a 相当于一个标签，贴在了对象 5 上面。并且我们把这个语句叫做赋值语句。

同样，在对字符串类型的对象，也是这样，能够通过赋值语句，将对象与某个标签（变量）关联起来。

```
>>> b = "hello,world"  
>>> b
```

```
'hello,world'
>>> print b
hello,world
```

还记得我们曾经用过一个 type 命令吗？现在它还有用，就是检验一个变量，到底跟什么类型联系着，是字符串还是数字？

```
>>> type(a)
<type 'int'>
>>> type(b)
<type 'str'>
```

有时候，你会听到一种说法：把a称之为数字型变量，把b叫做字符（串）型变量。这种说法，在某些语言中是成立的。某些语言，需要提前声明变量，然后变量就成为了一个筐，将值装到这个筐里面。但是，Python 不是这样的。要注意区别。

拼接字符串

还记得我在本节开篇提出的那个伟大发现吗？就是将两个东西拼接起来。

对数字，如果拼接，就是对两个数字求和。如：3+5，就计算出为8。那么对字符串都能进行什么样的操作呢？试试吧：

```
>>> "Py" + "thon"
'Python'
```

跟我那个不为大多数人认可的发现是一样的，你还不认可吗？两个字符串相加，就相当于把两个字符串连接起来。（别的运算就别尝试了，没什么意义，肯定报错，不信就试试）

```
>>> "Py" - "thon" # 这么做的人，是脑袋进水泥了吧？
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

用 `+` 号实现连接，的确比较简单，不过，有时候你会遇到这样的问题：

```
>>> a = 1989
>>> b = "free"
>>> print b+a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

这里引入了一个指令：`print`，意思就是打印后面的字符串（或者指向字符串的变量），上面是 Python2 中的使用方式，在 Python3 中，它变成了一个函数。应该用 `print(b+a)` 的样式了。

报错了，其错误原因已经打印出来了（一定要注意看打印出来的信息）：`cannot concatenate 'str' and 'int' objects`。原来 `a` 对应的对象是一个 `int` 类型的，不能将它和 `str` 对象连接起来。怎么办？

原来，用 `+` 拼接起来的两个对象，必须是同一种类型的。如果两个都是数字，毫无疑问是正确的，就是求和；如果都是字符串，那么就得到一个新的字符串。

修改上面的错误，可以通过以下方法：

```
>>> print b + `a`  
free1989
```

注意，`\` 是反引号，不是单引号，就是键盘中通常在数字1左边的那个，在英文半角状态下输入的符号。这种方法，在编程实践中比较少应用，特别是在 Python3 中，已经把这种方式弃绝了。我想原因就是这个符号太容易和单引号混淆了。在编程中，也不容易看出来，可读性太差。

常言道：“困难只有一个，解决困难的方法不止一种”，既然反引号可读性不好，在编程实践中就尽量不要使用。于是乎就有了下面的方法，这是被广泛采用的。不但简单，更主要是直白，一看就懂什么意思了。

```
>>> print b + str(a)  
free1989
```

用 `str(a)` 实现将整数对象转换为字符串对象。虽然 `str` 是一种对象类型，但是它也能够实现对象类型的转换，这就起到了一个函数的作用。其实前面已经讲过的 `int` 也有类似的作用。比如：

```
>>> a = "250"  
>>> type(a)  
<type 'str'>  
>>> b = int(a)  
>>> b  
250  
>>> type(b)  
<type 'int'>
```

提醒列位，如果你对 `int` 和 `str` 比较好奇，可以在交互模式中，使用 `help(int)`, `help(str)` 查阅相关的更多资料。

还有第三种：

```
>>> print b + repr(a) #repr(a)与上面的类似  
free1989
```

这里 `repr()` 是一个函数，其实就是反引号的替代品，它能够把结果字符串转化为合法的 python 表达式。

可能看官看到这个，就要问它们三者之间的区别了。首先明确，`repr()`和`\`是一致的，就不用区别了。接下来需要区别的就是`repr()`和`str`，一个最简单的区别，`repr`是函数，`str`是跟`int`一样，一种对象类型。不过这么说不能完全解惑的。幸亏有那么好的google让我辈使用，你会找到不少人对这两者进行区分的内容，我推荐这个：

1. When should i use str() and when should i use repr() ?

Almost always use str when creating output for end users.

`repr` is mainly useful for debugging and exploring. For example, if you suspect a string has non printing characters in it, or a float has a small rounding error, `repr` will show you; `str` may not.

`repr` can also be useful for generating literals to paste into your source code. It can also be used for persistence (with `ast.literal_eval` or `eval`), but this is rarely a good idea--if you want editable persisted values, something like JSON or YAML is much better, and if you don't plan to edit them, use pickle.

2.In which cases i can use either of them ?

Well, you can use them almost anywhere. You shouldn't generally use them except as described above.

3.What can str() do which repr() can't ?

Give you output fit for end-user consumption--not always (e.g., `str(['spam', 'eggs'])` isn't likely to be anything you want to put in a GUI), but more often than `repr`.

4.What can repr() do which str() can't

Give you output that's useful for debugging--again, not always (the default for instances of user-created classes is rarely helpful), but whenever possible.

And sometimes give you output that's a valid Python literal or other expression--but you rarely want to rely on that except for interactive exploration.

以上英文内容来源：<http://stackoverflow.com/questions/19331404/str-vs-repr-functions-in-python-2-7-5>

Python 转义字符

在字符串中，有时需要输入一些特殊的符号，但是，某些符号不能直接输出，就需要用转义符。所谓转义，就是不采用符号本来的含义，而采用另外一含义了。下面表格中列出常用的转义符：

转义字符	描述
\	(在行尾时) 续行符
\	反斜杠符号
'	单引号
"	双引号
\a	响铃
\b	退格(Backspace)
\e	转义
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页
\oyy	八进制数, yy 代表的字符, 例如: \o12 代表换行
\xyy	十六进制数, yy 代表的字符, 例如: \x0a 代表换行
\other	其它的字符以普通格式输出

以上所有转义符, 都可以通过交互模式下 print 来测试一下, 感受实际上是什么样子的。例如:

```
>>> print "hello.I am qiwsir.\n      # 这里换行, 下一行接续
... My website is 'http://qiwsir.github.io'." 
hello.I am qiwsir.My website is 'http://qiwsir.github.io'.

>>> print "you can connect me by qq\weibo\gmail" #\ 是为了要后面那个 \
you can connect me by qq\weibo\gmail
```

看官自己试试吧。如果有问题, 可以联系我解答。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我, 请通过支付宝: qiwsir@126.com, 不胜感激。

字符串(2)

raw_input 和 print

自从本课程开始以来，我们还没有感受到 computer 姑娘的智能。最简单的智能应该体现在哪里呢？想想小孩子刚刚回说话的时候情景吧。

小孩学说话，是一个模仿的过程，孩子周围的人怎么说，她（他）往往就是重复。看官可以忘记自己当初是怎么学说话了吧？就找个小孩子观察一下吧。最好是自己的孩子。如果没有，就要抓紧了。

通过 Python 能不能实现这个简单的功能呢？当然能，要不然 Python 如何横行天下呀。

不过在写这个功能前，要了解两个函数：raw_input 和 print

这两个都是 Python 的内建函数（built-in function）。关于 Python 的内建函数，下面这个表格都列出来了。所谓内建函数，就是能够在 Python 中直接调用，不需要做其它的操作。

Built-in Functions

<code>abs()</code>	<code>divmod()</code>	<code>input()</code>	<code>open()</code>	<code>staticmethod()</code>
<code>all()</code>	<code>enumerate()</code>	<code>int()</code>	<code>ord()</code>	<code>str()</code>
<code>any()</code>	<code>eval()</code>	<code>isinstance()</code>	<code>pow()</code>	<code>sum()</code>
<code>basestring()</code>	<code>execfile()</code>	<code>issubclass()</code>	<code>print()</code>	<code>super()</code>
<code>bin()</code>	<code>file()</code>	<code>iter()</code>	<code>property()</code>	<code>tuple()</code>
<code>bool()</code>	<code>filter()</code>	<code>len()</code>	<code>range()</code>	<code>type()</code>
<code>bytearray()</code>	<code>float()</code>	<code>list()</code>	<code>raw_input()</code>	<code>unichr()</code>
<code>callable()</code>	<code>format()</code>	<code>locals()</code>	<code>reduce()</code>	<code>unicode()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>long()</code>	<code>reload()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>map()</code>	<code>repr()</code>	<code>xrange()</code>
<code>cmp()</code>	<code>globals()</code>	<code>max()</code>	<code>reversed()</code>	<code>zip()</code>
<code>compile()</code>	<code>hasattr()</code>	<code>memoryview()</code>	<code>round()</code>	<code>import()</code>
<code>complex()</code>	<code>hash()</code>	<code>min()</code>	<code>set()</code>	<code>apply()</code>
<code>delattr()</code>	<code>help()</code>	<code>next()</code>	<code>setattr()</code>	<code>buffer()</code>
<code>dict()</code>	<code>hex()</code>	<code>object()</code>	<code>slice()</code>	<code>coerce()</code>
<code>dir()</code>	<code>id()</code>	<code>oct()</code>	<code>sorted()</code>	<code>intern()</code>

这些内建函数，怎么才能知道哪个函数怎么用，是干什么用的呢？

不知道你是否还记得我在前面使用过的方法，这里再进行演示，这种方法是学习 Python 的法宝。

```
>>> help(raw_input)
```

然后就出现：

```
Help on built-in function raw_input in module __builtin__:
```

```
raw_input(...)  
    raw_input([prompt]) -> string
```

Read a string from standard input. The trailing newline is stripped.

If the user hits EOF (Unix: Ctl-D, Windows: Ctl-Z+Return), raise EOFError.

On Unix, GNU readline is used if enabled. The prompt string, if given, is printed without a trailing newline before reading.

从中是不是已经清晰地看到了 `raw_input()` 的使用方法了。

还有第二种方法，那就是到 Python 的官方网站，查看内建函数的说明。<https://docs.Python.org/2/library/functions.html>

其实，我上面那个表格，就是在这个网页中抄过来的。

例如，对 `print()` 说明如下：

```
print(*objects, sep=' ', end='\n', file=sys.stdout)
```

Print objects to the stream file, separated by sep and followed by end. sep, end and file, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like str() does and written to the stream, separated by sep and followed by end.

The file argument must be an object with a write(string) method; if it is not present or None, sys.stdout will be used. Out-

分别在交互模式下，将这个两个函数操练一下。

```
>>> raw_input("input your name:")  
input your name:python  
'python'
```

输入名字之后，就返回了输入的内容。用一个变量可以获得这个返回值。

```
>>> name = raw_input("input your name:")  
input your name:python  
>>> name  
'python'  
>>> type(name)  
<type 'str'>
```

而且，返回的结果是 str 类型。如果输入的是数字呢？

```
>>> age = raw_input("How old are you?")
How old are you?10
>>> age
'10'
>>> type(age)
<type 'str'>
```

返回的结果，仍然是 str 类型。

再试试 `print()`，看前面对它的说明，是比较复杂的。没关系，我们从简单的开始。在交互模式下操作：

```
>>> print("hello, world")
hello, world
>>> a = "python"
>>> b = "good"
>>> print a
python
>>> print a,b
python good
```

比较简单吧。当然，这是没有搞太复杂了。

特别要提醒的是，`print()` 默认是以 `\n` 结尾的，所以，会看到每个输出语句之后，输出内容后面自动带上了 `\n`，于是就换行了。

有了以上两个准备，接下来就可以写一个能够“对话”的小程序了。

```
#!/usr/bin/env python
# coding=utf-8

name = raw_input("What is your name?")
age = raw_input("How old are you?")

print "Your name is:", name
print "You are " + age + " years old."

after_ten = int(age) + 10
print "You will be " + str(after_ten) + " years old after ten years."
```

对这段小程序中，有几点说明

前面演示了 `print()` 的使用，除了打印一个字符串之外，还可以打印字符串拼接结果。

```
print "You are " + age + " years old."
```

注意，那个变量 `age` 必须是字符串，如最后的那个语句中：

```
print "You will be " + str(after_ten) + " years old after ten years."
```

这句话里面，有一个类型转化，将原本是整数型 `after_ten` 转化为了 `str` 类型。否则，就包括，不信，你可以试试。

同样注意，在 `after_ten = int(age) + 10` 中，因为通过 `raw_input` 得到的是 `str` 类型，当 `age` 和 `10` 求和的时候，需要先用 `int()` 函数进行类型转化，才能和后面的整数 `10` 相加。

这个小程序，是有点综合的，基本上把已经学到的东西综合运用了一次。请看官调试一下，如果没有通过，仔细看报错信息，你能够从中获得修改方向的信息。

原始字符串

所谓原始字符串，就是指字符串里面的每个字符都是原始含义，比如反斜杠，不会被看做转义符。如果在一般字符串中，比如

```
>>> print "I like \npython"
I like
python
```

这里的反斜杠就不是“反斜杠”的原始符号含义，而是和后面的 `n` 一起表示换行（转义了）。当然，这似乎没有什么太大影响，但有的时候，可能会出现问题，比如打印 DOS 路径（DOS，有没有搞错，现在还有人用吗？）

```
>>> dos = "c:\news"
>>> dos
'c:\news'      # 这里貌似没有什么问题
>>> print dos  # 当用 print 来打印这个字符串的时候，就出问题了。
c:
ews
```

如何避免？用前面讲过的转义符可以解决：

```
>>> dos = "c:\\news"
>>> print dos
c:\\news
```

此外，还有一种方法，如：

```
>>> dos = r"c:\\news"
>>> print dos
c:\\news
```

```
>>> print r"c:\news\python"
c:\news\python
```

状如 `r"c:\news"`，由 `r` 开头引起的字符串，就是原始字符串，在里面放任何字符都表示该字符的原始含义。

这种方法在做网站设置网站目录结构的时候非常有用。使用了原始字符串，就不需要转义了。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

字符串(3)

关于字符串的内容，已经有两节进行介绍了。不过，它是一个话题中心，还要再继续。

例如这样一个字符串 `Python`，还记得前面对字符串的定义吗？它就是几个字符：`P,y,t,h,o,n`，排列起来。这种排列是非常严格的，不仅仅是字符本身，而且还有顺序，换言之，如果某个字符换了，就编程一个新字符串了；如果这些字符顺序发生变化了，也成为一个新字符串。

在 Python 中，把像字符串这样的对象类型（后面还会冒出来类似的其它有这种特点的对象类型，比如列表），统称为序列。顾名思义，序列就是“有序排列”。

比如水泊梁山的 108 个好汉（里面分明也有女的，难道女汉子是从这里来的吗？），就是一个“有序排列”的序列。从老大宋江一直排到第 108 位金毛犬段景住。在这个序列中，每个人有编号，编号和每个人一一对应。1 号是宋江，2 号是卢俊义。反过来，通过每个人的姓名，也能找出他对应的编号。武松是多少号？14 号。李逵呢？22 号。

在 Python 中，给这些编号取了一个文雅的名字，叫做索引（别的编程语言也这么称呼，不是 Python 独有的。）。

索引和切片

前面用梁山好汉的为例说明了索引。再看 Python 中的例子：

```
>>> lang = "study Python"
>>> lang[0]
's'
>>> lang[1]
't'
```

有一个字符串，通过赋值语句赋给了变量 `lang`。如果要得到这个字符串的第一个单词 `s`，可以用 `lang[0]`。当然，如果你不愿意通过赋值语句，让变量 `lang` 来指向那个字符串，也可以这样做：

```
>>> "study Python"[0]
's'
```

效果是一样的。因为 `lang` 是标签，就指向了 `"study Python"` 字符串。当让 Python 执行 `lang[0]` 的时候，就是要转到那个字符串对象，如同上面的操作一样。只不过，如果不用 `lang` 这么一个变量，后面如果再写，就费笔墨了，要每次都把那个字符串写全了。为了省事，还是复制给一个变量吧。变量就是字符串的代表了。

字符串这个序列的排序方法跟梁山好汉有点不同，第一个不是用数字1表示，而是用数字0表示。不仅仅 Python，其它很多语言都是从0开始排序的。为什么这样做呢？这就是规定。当然，这个规定是有一定优势的。此处不展开，有兴趣的网上去 google 一下，有专门对此进行解释的文章。

0	1	2	3	4	5	6	7	8	9	10	11
s	t	u	d	y	l	p	y	t	h	o	n

上面的表格中，将这个字符串从第一个到最后一个进行了排序，特别注意，两个单词中间的那个空格，也占用了一个位置。

通过索引能够找到该索引所对应的字符，那么反过来，能不能通过字符，找到其在字符串中的索引值呢？怎么找？

```
>>> lang.index("p")
6
```

就这样，是不是已经能够和梁山好汉的例子对上号了？只不过区别在于第一个的索引值是0。

如果某一天，宋大哥站在大石头上，向着各位弟兄大喊：“兄弟们，都排好队。”等兄弟们排好之后，宋江说：“现在给各位没有老婆的兄弟分配女朋友，我这里已经有了名单，我念叨的兄弟站出来。不过我是按照序号来念的。第29号到第34号先出列，到旁边房子等候分配女朋友。”

在前面的例子中 `lang[1]` 能够得到原来字符串的第二个字符 t，就相当于从原来字符串中把这个“切”出来了。不过，我们这么“切”却不影响原来字符串的整体性，当然可以理解为将那个字符 t 赋值一份拿出来了。

那么宋江大哥没有一个一个“切”，而是一下将几个兄弟叫出来。在 Python 中也能做类似事情。

```
>>> lang
'study Python' #在前面“切”了若干的字符之后，再看一下该字符串，还是完整的。
>>> lang[2:9]
'udy pyt'
```

通过 `lang[2:9]` 要得到部分（不是一个）字符，从返回的结果中可以看出，我们得到的是序号分别对应着 2,3,4,5,6,7,8（跟上面的表格对应一下）字符（包括那个空格）。也就是，这种获得部分字符的方法中，能够得到开始需要的以及最后一个序号之前的所对应的字符。有点拗口，自己对照上面的表格数一数就知道了。简单说就是包括开头，不包括结尾。

上述，不管是得到一个还是多个，通过索引得到字符的过程，称之为切片。

切片是一个很有意思的东西。可以“切”出不少花样呢？

```
>>> lang
'study Python'
```

```
>>> b = lang[1:] # 得到从 1 号到最末尾的字符，这时最后那个需要不用写
>>> b
'tudy Python'
>>> c = lang[:] # 得到所有字符
>>> c
'study Python'
>>> d = lang[:10] # 得到从第一个到 10 号之前的字符
>>> d
'study pyth'
```

在获取切片的时候，如果分号的前面或者后面的序号不写，就表示是到最末（后面的不写）或第一个（前面的不写）

`lang[:10]` 的效果和 `lang[0:10]` 是一样的。

```
>>> e = lang[0:10]
>>> e
'study pyth'
```

那么，`lang[1:]` 和 `lang[1:11]` 效果一样吗？请思考后作答。

```
>>> lang[1:11]
'tudy pytho'
>>> lang[1:]
'tudy python'
```

果然不一样，你思考对了吗？原因就是前述所说的，如果分号后面有数字，所得到的切片，不包含该数字所对应的序号（前包括，后不包括）。那么，是不是可以这样呢？`lang[1:12]`，不包括 12 号（事实没有 12 号），是不是可以得到 1 到 11 号对应的字符呢？

```
>>> lang[1:12]
'tudy python'
>>> lang[1:13]
'tudy python'
```

果然是。并且不仅仅后面写 12，写 13，也能得到同样的结果。但是，我这个特别要提醒，这种获得切片的做法在编程实践中是不提倡的。特别是如果后面要用到循环的时候，这样做或许在什么时候遇到麻烦。

如果在切片的时候，冒号左右都不写数字，就是前面所操作的 `c = lang[:]`，其结果是变量 `c` 的值与原字符串一样，也就是“复制”了一份。注意，这里的“复制”我打上了引号，意思是如同复制，是不是真的复制呢？可以用下面的方式检验一下

```
>>> id(c)
3071934536L
```

```
>>> id(lang)
3071934536L
```

`id()` 的作用就是查看该对象在内存地址（就是在内存中的位置编号）。从上面可以看出，两个的内存地址一样，说明 `c` 和 `lang` 两个变量指向的是同一个对象。用 `c=lang[:]` 的方式，并没有生成一个新的字符串，而是将变量 `c` 这个标签也贴在了原来那个字符串上了。

```
>>> lang = "study python"
>>> c = lang
```

如果这样操作，变量 `c` 和 `lang` 是不是指向同一个对象呢？或者两者所指向的对象内存地址如何呢？看官可以自行查看。

字符串基本操作

字符串是一种序列，所有序列都有如下基本操作：

1. `len()`: 求序列长度
2. • : 连接 2 个序列
3. • : 重复序列元素
4. `in` : 判断元素是否存在于序列中
5. `max()` : 返回最大值
6. `min()` : 返回最小值
7. `cmp(str1,str2)` : 比较 2 个序列值是否相同

通过下面的例子，将这几个基本操作在字符串上的使用演示一下：

“+”连接字符串

```
>>> str1 + str2
'abcdabcde'
>>> str1 + "-->" + str2
'abcd-->abcde'
```

这其实就是拼接，不过在这里，看官应该有一个更大的观念，我们现在只是学了字符串这一种序列，后面还会遇到列表、元组两种序列，都能够如此实现拼接。

in

```
>>> "a" in str1
True
>>> "de" in str1
False
>>> "de" in str2
True
```

in 用来判断某个字符串是不是在另外一个字符串内，或者说判断某个字符串内是否包含某个字符串，如果包含，就返回 True，否则返回 False。

最值

```
>>> max(str1)
'd'
>>> max(str2)
'e'
>>> min(str1)
'a'
```

一个字符串中，每个字符在计算机内都是有编码的，也就是对应着一个数字，min() 和 max() 就是根据这个数字里获得最小值和最大值，然后对应出相应的字符。关于这种编号是多少，看官可以 google 有关字符编码，或者 ASCII 编码什么的，很容易查到。

比较

```
>>> cmp(str1, str2)
-1
```

将两个字符串进行比较，也是首先将字符串中的符号转化为对一个的数字，然后比较。如果返回的数值小于零，说明第一个小于第二个，等于 0，则两个相等，大于 0，第一个大于第二个。为了能够明白其所以然，进入下面的分析。

```
>>> ord('a')
97
>>> ord('b')
98
>>> ord(' ')
32
```

`ord()` 是一个内建函数，能够返回某个字符（注意，是一个字符，不是多个字符组成的串）所对一个的 ASCII 值（是十进制的），字符 a 在 ASCII 中的值是 97，空格在 ASCII 中也有值，是 32。顺便说明，反过来，根据整数值得到相应字符，可以使用 `chr()`：

```
>>> chr(97)
'a'
>>> chr(98)
'b'
```

于是，就得到如下比较结果了：

```
>>> cmp("a","b") #a-->97, b-->98, 97 小于 98, 所以 a 小于 b
-1
>>> cmp("abc","aaa")
1
>>> cmp("a","a")
0
```

看看下面的比较，是怎么进行的呢？

```
>>> cmp("ad","c")
-1
```

在字符串的比较中，是两个字符串的第一个字符先比较，如果相等，就比较下一个，如果不相等，就返回结果。直到最后，如果还相等，就返回 0。位数不够时，按照没有处理（注意，没有不是 0，0 在 ASCII 中对应的是 NUL），位数多的那个天然大了。ad 中的 a 先和后面的 c 进行比较，显然 a 小于 c，于是就返回结果 -1。如果进行下面的比较，是最容易让人迷茫的。看官能不能根据刚才阐述的比较远离理解呢？

```
>>> cmp("123","23")
-1
>>> cmp(123,23) # 也可以比较整数，这时候就是整数的直接比较了。
1
```

“*”

字符串中的“乘法”，这个乘法，就是重复那个字符串的含义。在某些时候很好用的。比如我要打印一个华丽的分割线：

```
>>> str1*3
'abcdabcdabcd'
>>> print "-"*20 # 不用输入很多个`-`
-----
```

len()

要知道一个字符串有多少个字符，一种方法是从头开始，盯着屏幕数一数。哦，这不是计算机在干活，是键客在干活。

键客，不是剑客。剑客是以剑为武器的侠客；而键客是以键盘为武器的侠客。当然，还有贱客，那是贱人的最高境界，贱到大侠的程度，比如岳不群之流。

键客这样来数字符串长度：

```
>>> a="hello"  
>>> len(a)  
5
```

使用的是一个函数 `len(object)`。得到的结果就是该字符串长度。

```
>>> m = len(a) # 把结果返回后赋值给一个变量  
>>> m  
5  
>>> type(m) # 这个返回值（变量）是一个整数型  
<type 'int'>
```

[总目录 \(页 0\)](#)

如果你认为有必要打赏我, 请通过支付宝: qiwsir@126.com, 不胜感激。

字符串(4)

字符串的内容的确不少，甚至都有点啰嗦了。但是，本节依然还要继续，就是在编程实践中，经常会遇到有关字符串的问题，而且也是很多初学者容易迷茫的。

字符串格式化输出

什么是格式化？在维基百科中有专门的词条，这么说的：

格式化是指对磁盘或磁盘中的分区（partition）进行初始化的一种操作，这种操作通常会导致现有的磁盘或分区中所有的文件被清除。

不知道你是否知道这种“格式化”。显然，此格式化非我们这里所说的，我们说的是字符串的格式化，或者说成“格式化字符串”，都可以，表示的意思就是：

格式化字符串，是 C、C++ 等程序设计语言 printf 类函数中用于指定输出参数的格式与相对位置的字符串参数。其中的转换说明（conversion specification）用于把随后对应的 0 个或多个函数参数转换为相应的格式输出；格式化字符串中转换说明以外的其它字符原样输出。

这也是来自维基百科的定义。在这个定义中，是用 C 语言作为例子，并且用了其输出函数来说明。在 Python 中，也有同样的操作和类似的函数 `print`，此前我们已经了解一二了。

如果将那个定义说的通俗一些，字符串格式化化，就是要先制定一个模板，在这个模板中某个或者某几个地方留出空位来，然后在那些空位填上字符串。那么，那些空位，需要用一个符号来表示，这个符号通常被叫做占位符（仅仅是占据着那个位置，并不是输出的内容）。

```
>>> "I like %s"
'I like %s'
```

在这个字符串中，有一个符号：`%s`，就是一个占位符，这个占位符可以被其它的字符串代替。比如：

```
>>> "I like %s" % "python"
'I like python'
>>> "I like %s" % "Pascal"
'I like Pascal'
```

这是较为常用的一种字符串输出方式。

另外，不同的占位符，会表示那个位置应该被不同类型的对象填充。下面列出许多，供参考。不过，不用记忆，常用的只有 `%s` 和 `%d`，或者再加上 `%f`，其它的如果需要了，到这里来查即可。

占位符	说明
%s	字符串(采用 str() 的显示)
%r	字符串(采用 repr() 的显示)
%c	单个字符
%b	二进制整数
%d	十进制整数
%i	十进制整数
%o	八进制整数
%x	十六进制整数
%e	指数 (基底写为 e)
%E	指数 (基底写为 E)
%f	浮点数
%F	浮点数, 与上相同
%g	指数(e)? 或浮点数 (根据显示长度)
%G	指数(E)或浮点数 (根据显示长度)

看例子：

```
>>> a = "%d years" % 15
>>> print a
15 years
```

当然，还可以在一个字符串中设置多个占位符，就像下面一样

```
>>> print "Suzhou is more than %d years. %s lives in here." % (2500, "qiwsir")
Suzhou is more than 2500 years. qiwsir lives in here.
```

对于浮点数字的打印输出，还可以限定输出的小数位数和其它样式。

```
>>> print "Today's temperature is %.2f" % 12.235
Today's temperature is 12.23
>>> print "Today's temperature is %+.2f" % 12.235
Today's temperature is +12.23
```

注意，上面的例子中，没有实现四舍五入的操作。只是截取。

关于类似的操作，还有很多变化，比如输出格式要宽度是多少等等。如果看官在编程中遇到了，可以到网上查找。我这里给一个参考图示，也是从网上抄来的。

数字	格式	输出	描述
3.1415926	{:.2f}	3.14	保留小数点后两位
3.1415926	{:+.2f}	+3.14	带符号保留小数点后两位
-1	{:+.2f}	-1.00	带符号保留小数点后两位
2.71828	{:.0f}	3	不带小数
5	{:0>2d}	05	数字补零 (填充左边, 宽度为 2)
5	{:x<4d}	5xxx	数字补 x (填充右边, 宽度为 4)
10	{:x<4d}	10xx	数字补 x (填充右边, 宽度为 4)
1000000	{:,}	1,000,000	以逗号分隔的数字格式
0.25	{:.2%}	25.00%	百分比格式
1000000000	{:.2e}	1.00e+09	指数记法
13	{:10d}	13	右对齐 (默认, 宽度为 10)
13	{:<10d}	13	左对齐 (宽度为 10)
13	{:^10d}	13	中间对齐 (宽度为 10)

其实，上面这种格式化方法，常常被认为太“古老”了。因为在 Python 中还有新的格式化方法。

```
>>> s1 = "I like {}".format("python")
>>> s1
'I like python'
>>> s2 = "Suzhou is more than {} years. {} lives in here.".format(2500, "qiwsir")
>>> s2
'Suzhou is more than 2500 years. qiwsir lives in here.'
```

这就是 Python 非常提倡的 `string.format()` 的格式化方法，其中 `{}` 作为占位符。

这种方法真的是非常好，而且非常简单，只需要将对应的东西，按照顺序在 `format` 后面的括号中排列好，分别对应占位符 `{}` 即可。我喜欢的方法。

如果你觉得还不明确，还可以这样做。

```
>>> print "Suzhou is more than {year} years. {name} lives in here.".format(year=2500, name="qiwsir")
Suzhou is more than 2500 years. qiwsir lives in here.
```

真的很简洁，看成优雅。

其实，还有一种格式化的方法，被称为“字典格式化”，这里仅仅列一个例子，如果看官要了解字典的含义，本教程后续会有的。

```
>>> lang = "Python"
>>> print "I love %(program)s"%{"program":lang}
I love Python
```

列举了三种基本格式化的方法，你喜欢那种？我推荐： `string.format()`

常用的字符串方法

字符串的方法很多。可以通过 `dir` 来查看：

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__hash__', '__init__', '__iter__', '__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__str__', '__subclasshook__']
```

这么多，不会一一介绍，要了解某个具体的含义和使用方法，最好是使用 `help` 查看。举例：

```
>>> help(str.isalpha)

Help on method_descriptor:

isalpha(...)
    S.isalpha() -> bool

    Return True if all characters in S are alphabetic
    and there is at least one character in S, False otherwise.
```

按照这里的说明，就可以在交互模式下进行实验。

```
>>> "python".isalpha()  # 字符串全是字母，应该返回 True
True
>>> "2python".isalpha()  # 字符串含非字母，返回 False
False
```

split

这个函数的作用是将字符串根据某个分割符进行分割。

```
>>> a = "I LOVE PYTHON"
>>> a.split(" ")
['I', 'LOVE', 'PYTHON']
```

这是用空格作为分割，得到了一个名字叫做列表（list）的返回值，关于列表的内容，后续会介绍。还能用别的分隔吗？

```
>>> b = "www.itdiffer.com"
>>> b.split(".")
['www', 'itdiffer', 'com']
```

去掉字符串两头的空格

这个功能，在让用户输入一些信息的时候非常有用。有的朋友喜欢输入结束的时候敲击空格，比如让他输入自己的名字，输完了，他来个空格。有的则喜欢先加一个空格，总做的输入的第一个字前面应该空两个格。

这些空格是没用的。Python 考虑到有不少人可能有这个习惯，因此就帮助程序员把这些空格去掉。

方法是：

- S.strip() 去掉字符串的左右空格
- S.lstrip() 去掉字符串的左边空格
- S.rstrip() 去掉字符串的右边空格

例如：

```
>>> b=" hello " # 两边有空格
>>> b.strip()
'hello'
>>> b
' hello '
```

特别注意，原来的值没有变化，而是新返回了一个结果。

```
>>> b.lstrip() # 去掉左边的空格
'hello '
>>> b.rstrip() # 去掉右边的空格
' hello'
```

字符大小写的转换

对于英文，有时候要用到大小写转换。最有名驼峰命名，里面就有一些大写和小写的参合。如果有兴趣，可以来这里看[自动将字符串转化为驼峰命名形式的方法[href="https://github.com/qiwsir/algorithm/blob/master/string_to_hump.md"](https://github.com/qiwsir/algorithm/blob/master/string_to_hump.md)]。

在 Python 中有下面一堆内建函数，用来实现各种类型的大小写转化

- S.upper() #S 中的字母大写

- S.lower() #S 中的字母小写
- S.capitalize() # 首字母大写
- S.isupper() #S 中的字母是否全是大写
- S.islower() #S 中的字母是否全是小写
- S.istitle()

看例子：

```
>>> a = "qiwsir,Python"
>>> a.upper()      # 将小写字母完全变成大写字母
'QIWSIR,PYTHON'
>>> a          # 原数据对象并没有改变
'qiwsir,Python'
>>> b = a.upper()
>>> b
'QIWSIR,PYTHON'
>>> c = b.lower()  # 将所有的小写字母变成大写字母
>>> c
'qiwsir,Python'

>>> a
'qiwsir,Python'
>>> a.capitalize() # 把字符串的第一个字母变成大写
'Qiwsir,Python'
>>> a          # 原数据对象没有改变
'qiwsir,Python'
>>> b = a.capitalize() # 新建立了一个
>>> b
'Qiwsir,Python'

>>> a = "qiwsir,github"  # 这里的问题就是网友白羽毛指出的，非常感谢他。
>>> a.istitle()
False
>>> a = "QIWSIR"      # 当全是大写的时候，返回 False
>>> a.istitle()
False
>>> a = "qIWSIR"
>>> a.istitle()
False
>>> a = "Qiwsir,github" # 如果这样，也返回 False
>>> a.istitle()
False
>>> a = "Qiwsir"      # 这样是 True
```

```
>>> a.istitle()
True
>>> a = 'Qiwsir,Github' # 这样也是 True
>>> a.istitle()
True

>>> a = "Qiwsir"
>>> a.isupper()
False
>>> a.upper().isupper()
True
>>> a.islower()
False
>>> a.lower().islower()
True
```

顺着白羽毛网友指出的，再探究一下，可以这么做：

```
>>> a = "This is a Book"
>>> a.istitle()
False
>>> b = a.title() # 这样就把所有单词的第一个字母转化为大写
>>> b
'This Is A Book'
>>> b.istitle() # 判断每个单词的第一个字母是否为大写
True
```

join 拼接字符串

用“+”能够拼接字符串，但不是什么情况下都能够如愿的。比如，将列表（关于列表，后续详细说，它是另外一种类型）中的每个字符（串）元素拼接成一个字符串，并且用某个符号连接，如果用“+”，就比较麻烦了（是能够实现的，麻烦）。

用字符串的 join 就比较容易实现。

```
>>> b
'www.itdiffer.com'
>>> c = b.split(".")
>>> c
['www', 'itdiffer', 'com']
>>> ".".join(c)
'www.itdiffer.com'
>>> "*".join(c)
'www*itdiffer*com'
```

这种拼接，是不是简单呢？

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

字符编码

我在第一版的《零基础学 Python》中，这个标题前面加了“坑爹”两个字。在后来的实践中，很多朋友都在网上问我关于编码的事情。说明这的确是一个“坑”。

首先说明，在 Python2 中，编码问题的确有点麻烦。但是，Python3 就不用纠结于此了。但是，正如前面所说的原因，至少本教程还是用 Python2，所以，必须要搞清楚编码。当然了，搞清楚，也不是坏事。

字符编码，在编程中，是一个让学习者比较郁闷的东西，比如一个 str，如果都是英文，好说多了。但恰恰不是如此，中文是我们不得不使用的。所以，哪怕是初学者，都要了解并能够解决字符编码问题。

```
>>> name = '老齐'  
>>> name  
'\xe8\x80\x81\xe9\xbd\x90'
```

在你的编程中，你遇到过上面的情形吗？认识最下面一行打印出来的东西吗？看人家英文，就好多了

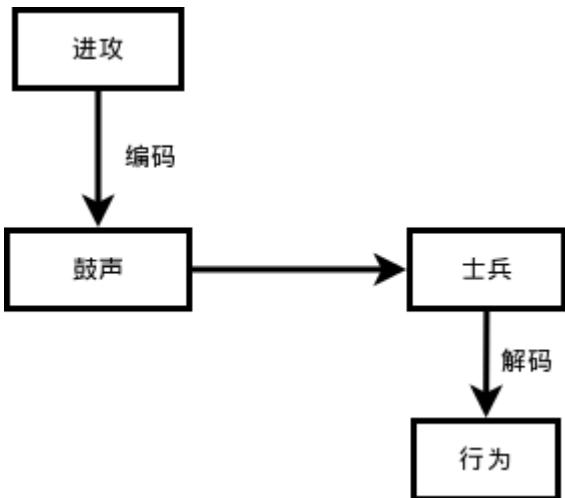
```
>>> name = "qiwsir"  
>>> name  
'qiwsir'
```

难道这是中文的错吗？看来投胎真的是一个技术活。是的，投胎是技术活，但上面的问题不是中文的错。

编码

什么是编码？这是一个比较玄乎的问题。也不好下一个普通定义。我看到有的教材中有定义，不敢说他的定义不对，至少可以说不容易理解。

古代打仗，击鼓进攻、鸣金收兵，这就是编码。把要传达给士兵的命令对应为一定的其它形式，比如命令“进攻”，经过如此的信息传递：



1. 长官下达进攻命令，传令员将这个命令编码为鼓声（如果复杂点，是不是有几声鼓响，如何进攻呢？）。
2. 鼓声在空气中传播，比传令员的嗓子吼出来的声音传播的更远，士兵听到后也不会引起歧义，一般不会有士兵把鼓声当做打呼噜的声音。这就是“进攻”命令被编码成鼓声之后的优势所在。
3. 士兵听到鼓声，就是接收到信息之后，如果接受过训练或者有人告诉过他们，他们就知道这是让我进攻。这个过程就是解码。所以，编码方案要有两套。一套在信息发出者那里，另外一套在信息接受者这里。经过解码之后，士兵明白了，才行动。

以上过程比较简单。其实，真实的编码和解码过程，要复杂了。不过，原理都差不多的。

举一个似乎遥远，其实不久前人们都在使用的东西做例子：[电报](#)

电报是通信业务的一种，在19世纪初发明，是最早使用电进行通信的方法。电报大为加快了消息的流通，是工业社会的其中一项重要发明。早期的电报只能在陆地上通讯，后来使用了海底电缆，开展了越洋服务。到了20世纪初，开始使用无线电拨发电报，电报业务基本上已能抵达地球上大部份地区。电报主要是用作传递文字讯息，使用电报技术用作传送图片称为传真。

中国首条出现电报线路是1871年，由英国、俄国及丹麦敷设，从香港经上海至日本长崎的海底电缆。由于清政府的反对，电缆被禁止在上海登陆。后来丹麦公司不理清政府的禁令，将线路引至上海公共租界，并在6月3日起开始收发电报。至于首条自主敷设的线路，是由福建巡抚丁日昌在台湾所建，1877年10月完工，连接台南及高雄。1879年，北洋大臣李鸿章在天津、大沽及北塘之间架设电报线路，用作军事通讯。1880年，李鸿章奏准开办电报总局，由盛宣怀任总办。并在1881年12月开通天津至上海的电报服务。李鸿章说：“五年来，我国创设沿江沿海各省电线，总计一万多里，国家所费无多，巨款来自民间。当时正值法人挑衅，将帅报告军情，朝廷传达指示，均相机而动，无丝毫阻碍。中国自古用兵，从未如此神速。出使大臣往来问答，朝发夕至，相隔万里好似同居庭院。举设电报一举三得，既防止外敌侵略，又加强国防，亦有利于商务。”天津官电局于庚子遭乱全毁。1887年，台湾巡抚刘铭传敷设了福州至台湾的海底电缆，是中国首条海底电缆。1884年，北京电报开始建设，采用“安设双线，由通州展至京城，以一端引入署中，专递官信，以一端择地安置用便”。

商民”，同年 8 月 5 日，电报线路开始建设，所有电线杆一律漆成红色。8 月 22 日，位于北京崇文门外大街西的喜鹊胡同的外城商用电报局开业。同年 8 月 30 日，位于崇文门内泡子和以西的吕公堂开局，专门收发官方电报。

为了传达汉字，电报部门准备由 4 位数字或 3 位罗马字构成的代码，即中文电码，采用发送前将汉字改写成电码发出，收电报后再将电码改写成汉字的方法。

列位看官注意了，这里出现了电报中用的“[中文电码](#)”，这就是一种编码，将汉字对应成阿拉伯数字，从而能够用电报发送汉字。

1873 年，法国驻华人员威基杰参照《康熙字典》的部首排列方法，挑选了常用汉字 6800 多个，编成了第一部汉字电码本《电报新书》。

电报中的编码被称为[摩尔斯电码](#)，英文是 Morse Code

摩尔斯电码（英语：Morse Code）是一种时通时断的信号代码，通过不同的排列顺序来表达不同的英文字母、数字和标点符号。是由美国人萨缪尔·摩尔斯在 1836 年发明。

摩尔斯电码是一种早期的数字化通信形式，但是它不同于现代只使用 0 和 1 两种状态的二进制代码，它的代码包括五种：点（.）、划（-）、每个字符间短的停顿（在点和划之间的停顿）、每个词之间中等的停顿、以及句子之间长的停顿

看来电报员是一个技术活，不同长短的停顿都代表了不同意思。哦，对了，有一个老片子《永不消逝的电波》，看完之后保证你才知道，里面根本就没有讲电报是怎么编码的。

摩尔斯电码在海事通讯中被作为国际标准一直使用到 1999 年。1997 年，当法国海军停止使用摩尔斯电码时，发送的最后一条消息是：“所有人注意，这是我们在永远沉寂之前最后的一声呐喊！”

```
*****-/*----/-----*/*****-/*----/-----*/-----*/*****-/*----/-*****/***-
```

```
*****-/*----/-----*/*****-/*----/-----*/-----*/*****-/*----/-*****/***-
```

我瞪着眼看了老长时间，这两行不是一样的吗？

不管这个了，总之，这就是编码。

计算机中的字符编码

先抄一段[维基百科对字符编码](#)的解释：

字符编码（英语：Character encoding）、字集码是把字符集中的字符编码为指定集合中某一对象（例如：比特模式、自然数串行、8 位组或者电脉冲），以便文本在计算机中存储和通过通信网络的传递。常见的例子包括将拉丁字母表编码成摩斯电码和 ASCII。其中，ASCII 将字母、数字和其它符号编号，并用 7 比特的二进制来表示这个整数。通常会额外使用一个扩充的比特，以便于以 1 个字节的方式存储。

在计算机技术发展的早期，如 ASCII（1963 年）和 EBCDIC（1964 年）这样的字符集逐渐成为标准。但这些字符集的局限很快就变得明显，于是人们开发了许多方法来扩展它们。对于支持包括东亚 CJK 字符家族在内的写作系统的要求能支持更大量的字符，并且需要一种系统而不是临时的方法实现这些字符的编码。

在这个世界上，有好多不同的字符编码。但是，它们不是自己随便搞搞的。而是要有一定的基础，往往是以名叫 [ASCII](#) 的编码为基础，这里边也应该包括北朝鲜吧（不知道他们用什么字符编码，瞎想的，别当真，不代表本教材立场，只代表瞎想）。

ASCII (pronunciation: 英语发音: /'æski/ ASS-kee[1], American Standard Code for Information Interchange, 美国信息交换标准代码) 是基于拉丁字母的一套电脑编码系统。它主要用于显示现代英语，而其扩展版本 EASCII 则可以部分支持其他西欧语言，并等同于国际标准 ISO/IEC 646。由于万维网使得 ASCII 广为通用，直到 2007 年 12 月，逐渐被 Unicode 取代。

上面的引文中已经说了，现在我们用的编码标准已经变成 Unicode 了，那么什么是 Unicode 呢？还是抄一段来自[维基百科的说明](#)

Unicode (中文：万国码、国际码、统一码、单一码) 是计算机科学领域里的一项业界标准。它对世界上大部分的文字系统进行了整理、编码，使得电脑可以用更为简单的方式来呈现和处理文字。

Unicode 伴随着通用字符集的标准而发展，同时也以书本的形式对外发表。Unicode 至今仍在不断增修，每个新版本都加入更多新的字符。目前最新的版本为 7.0.0，已收入超过十万个字符（第十万个字符在 2005 年获采纳）。Unicode 涵盖的数据除了视觉上的字形、编码方法、标准的字符编码外，还包含了字符特性，如大小写字母。

听这名字：万国码，那就一定包含了中文喽。的确是。但是，光有一个 Unicode 还不行，因为....（此处省略若干字，看官可以到上面给出的维基百科链接中看），还要有其它的一些编码实现方式，Unicode 的实现方式称为 Unicode 转换格式（Unicode Transformation Format，简称为 UTF），于是乎有了一个我们在很多时候都会看到的 utf-8。

什么是 utf-8，还是看[维基百科](#)上怎么说的吧

UTF-8 (8-bit Unicode Transformation Format) 是一种针对 Unicode 的可变长度字符编码，也是一种前缀码。它可以用来表示 Unicode 标准中的任何字符，且其编码中的第一个字节仍与 ASCII 兼容，这使得原来处理 ASCII 字符的软件无须或只须做少部份修改，即可继续使用。因此，它逐渐成为电子邮件、网页及其他存储或发送文字的应用中，优先采用的编码。

不再多引用了，如果要看更多，请到原文。

看官现在是不是就理解了，前面写程序的时候，曾经出现过：coding:utf-8 的字样。就是在告诉 python 我们要用什么字符编码呢。

encode 和 decode

历史部分说完了，接下怎么讲？比较麻烦了。因为不管怎么讲，都不是三言两语说清楚的。姑且从 encode() 和 decode() 两个内置函数起吧。

`codecs.encode(obj[, encoding[, errors]])`:Encodes obj using the codec registered for encoding. codecs.decode(obj[, encoding[, errors]]):Decodes obj using the codec registered for encoding.

Python2 默认的编码是 ascii，通过 encode 可以将对象的编码转换为指定编码格式（称作“编码”），而 decode 是这个过程的逆过程（称作“解码”）。

做一个实验，才能理解：

```
>>> a = "中"
>>> type(a)
<type 'str'>
>>> a
'\xe4\xb8\xad'
>>> len(a)
3

>>> b = a.decode()
>>> b
u'\u4e2d'
>>> type(b)
<type 'unicode'>
>>> len(b)
1
```

这个实验不做之前，或许看官还不是很迷茫（因为不知道，知道的越多越迷茫），实验做完了，自己也迷茫了。别急躁，对编码问题的理解，要慢慢来，如果一时理解不了，也肯定理解不了，就先注意按照要求做，做着做着就豁然开朗了。

上面试验中，变量 a 引用了一个字符串，所谓字符串(str)，严格地将是字节串，它是经过编码后的字节组成的序列。也就是你在上面的实验中，看到的是“中”这个字在计算机中编码之后的字节表示。（关于字节，看官可以 google 一下）。用 len(a) 来度量它的长度，它是由三个字节组成的。

然后通过 decode 函数，将字节串转变为字符串，并且这个字符串是按照 unicode 编码的。在 unicode 编码中，一个汉字对应一个字符，这时候度量它的长度就是 1。

反过来，一个 unicode 编码的字符串，也可以转换为字节串。

```
>>> c = b.encode('utf-8')
>>> c
'\xe4\xb8\xad'
>>> type(c)
<type 'str'>
>>> c == a
True
```

关于编码问题，先到到这里，点到为止吧。因为再扯，还会扯出问题来。看官肯定感到不满意，因为还没有知其所以然。没关系，请尽情 google，即可解决。

Python 中如何避免中文是乱码

这个问题是一个具有很强操作性的问题。我这里有一个经验总结，分享一下，供参考：

首先，提倡使用 utf-8 编码方案，因为它跨平台不错。

经验一：在开头声明：

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

有朋友问我-*-有什么作用，那个就是为了好看，爱美之心人皆有，更何况程序员？当然，也可以写成：

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

经验二：遇到字符（节）串，立刻转化为 unicode，不要用 str()，直接使用 unicode()

```
unicode_str = unicode('中文', encoding='utf-8')
print unicode_str.encode('utf-8')
```

经验三：如果对文件操作，打开文件的时候，最好用 codecs.open，替代 open(这个后面会讲到，先放在这里)

```
import codecs
codecs.open('filename', encoding='utf8')
```

我还收集了网上的一片文章，也挺好的，推荐给看官：[\[Python2.x的中文显示方法](https://github.com/qiwsir/ITArticles/blob/master/Python/Python%E7%9A%84%E4%B8%AD%E6%96%87%E6%98%BE%E7%A4%BA%E6%96%B9%E6%B3%95.md)

最后告诉给我，如果用 Python3，坑爹的编码问题就不烦恼了。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

列表(1)

前面的学习中，我们已经知道了两种 Python 的数据类型：int 和 str。再强调一下对数据类型的理解，这个世界是由数据组成的，数据可能是数字（注意，别搞混了，数字和数据是有区别的），也可能是文字、或者是声音、视频等。在 Python 中（其它高级语言也类似）把状如 2,3 这样的数字划分为一个类型，把状如“你好”这样的文字划分一个类型，前者是 int 类型，后者是 str 类型（这里就不说翻译的名字了，请看官熟悉用英文的名称，对日后编程大有好处，什么好处呢？谁用谁知道！）。

前面还学习了变量，如果某个变量指向一个对象（某种类型的数据）行话是：赋值），通常这个变量我们就把它叫做 int 类型的变量（注意，这种说法是不严格的，或者是受到别的语言影响的，在 Python 中，特别要注意：变量没有类型，对象有类型。在 Python 里，变量不用提前声明（在某些语言，如 JAVA 中需要声明变量之后才能使用。这个如果看官没有了解，不用担心，因为我们是学习 Python，以后学习的语言多了，自然就能体会到这点区别了），随用随命名。

这一讲中的 list 类型，也是 Python 的一种数据类型。翻译为：列表。下面的黑字，请看官注意了：

LIST 在 Python 中具有非常强大的功能。

定义

在 Python 中，用方括号表示一个 list，[]

在方括号里面，可以是 int，也可以是 str 类型的数据，甚至也能够是 True/False 这种布尔值。看下面的例子，特别注意阅读注释。

```
>>> a=[]      #定义了一个变量 a，它是 list 类型，并且是空的。
>>> type(a)
<type 'list'> #用内置函数 type()查看变量 a 的类型，为 list
>>> bool(a)   #用内置函数 bool()看看 list 类型的变量 a 的布尔值，因为是空的，所以为 False
False
>>> print a  #打印 list 类型的变量 a
[]
```

bool() 是一个布尔函数，这个东西后面会详述。它的作用就是来判断一个对象是“真”还是“空”（假）。如果想上面例子那样，list 中什么也没有，就是空的，用 bool() 函数来判断，得到 False，从而显示它是空的。

不能总玩空的，来点实的吧。

```
>>> a=['2',3,'qiwsir.github.io']
>>> a
['2', 3, 'qiwsir.github.io']
>>> type(a)
<type 'list'>
>>> bool(a)
True
>>> print a
['2', 3, 'qiwsir.github.io']
```

用上述方法，定义一个 list 类型的变量和数据。

本讲的标题是“有容乃大的 list”，就指明了 list 的一大特点：可以无限大，就是说 list 里面所能容纳的元素数量无限，当然这是在硬件设备理想的情况下。

如果看官以后或者已经了解了别的语言，比如比较常见的 Java，里面有一个跟 list 相似的数据类型——数组——但是两者还是有区别的。在 Java 中，数组中的元素必须是基本数据类型中某一个，也就是要么都是 int 类型，要么都是 char 类型等，不能一个数组中既有 int 类型又有 char 类型。这是因为 java 中的数组，需要提前声明，声明的时候就确定了里面元素的类型。但是 python 中的 list，尽管跟 java 中的数组有类似的地方——都是 [] 包裹的——list 中的元素是任意类型的，可以是 int,str，甚至还可以是 list，乃至于是以后要学的 dict 等。所以，有一句话说：List 是 python 中的苦力，什么都可以干。

索引和切片

尚记得在[《字符串\(3\)》](#)中，曾经给“索引”(index)和“切片”。

```
>>> url = "qiwsir.github.io"
>>> url[2]
'w'
>>> url[:4]
'qiws'
>>> url[3:9]
'sir.gi'
```

在 list 中，也有类似的操作。只不过是以元素为单位，不是以字符为单位进行索引了。看例子就明白了。

```
>>> a
['2', 3, 'qiwsir.github.io']
>>> a[0] #索引序号也是从 0 开始
'2'
>>> a[1]
3
>>> [2]
```

```
[2]
>>> a[:2] #跟str中的类似，切片的范围是：包含开始位置，到结束位置之前
['2', 3] #不包含结束位置
>>> a[1:]
[3, 'qiwsir.github.io']
```

list 和 str 两种类型的数据，有共同的地方，它们都属于序列（都是一些对象按照某个次序排列起来，这就是序列的最大特征），因此，就有很多类似的地方。如刚才演示的索引和切片，是非常一致的。

```
>>> lang = "python"
>>> lang.index("y")
1
>>> lst = ['python','java','c++']
>>> lst.index('java')
1
```

在前面讲述字符串索引和切片的时候，以及前面的演示，所有的索引都是从左边开始编号，第一个是 0，然后依次增加 1。此外，还有一种编号方式，就是从右边开始，右边第一个可以编号为 -1，然后向左依次是：-2,-3,...，依次类推下来。这对字符串、列表等各种序列类型都是用。

```
>>> lang
'python'
>>> lang[-1]
'n'
>>> lst
['python', 'java', 'c++']
>>> lst[-1]
'c++'
```

从右边开始编号，第 -1 号是右边第一个。但是，如果要切片的话，应该注意了。

```
>>> lang[-1:-3]
"
>>> lang[-3:-1]
'ho'
>>> lst[-3:-1]
['python', 'java']
```

序列的切片，一定要左边的数字小有右边的数字， lang[-1:-3] 就没有遵守这个规则，返回的是一个空。

反转

这个功能作为一个独立的项目提出来，是因为在编程中常常会用到。通过举例来说明反转的方法：

```
>>> alst = [1,2,3,4,5,6]
>>> alst[::-1] #反转
[6, 5, 4, 3, 2, 1]
>>> alst
[1, 2, 3, 4, 5, 6]
```

当然，对于字符串也可以

```
>>> lang
'python'
>>> lang[::-1]
'nohtyp'
>>> lang
'python'
```

看官是否注意到，上述不管是 str 还是 lst 反转之后，再看原来的值，没有改变。这就说明，这里的反转，不是在“原地”把原来的值倒过来，而是新生成了一个值，那个值跟原来的值相比，是倒过来了。

这是一种非常简单的方法，虽然我在写程序的时候常常使用，但是，我不是十分推荐，因为有时候让人感觉迷茫。Python 还有另外一种方法让list反转，是比较容易理解和阅读的，特别推荐之：

```
>>> list(reversed(alst))
[6, 5, 4, 3, 2, 1]
```

比较简单，而且很容易看懂。不是吗？

顺便给出 reversed 函数的详细说明：

```
>>> help(reversed)
Help on class reversed in module __builtin__:

class reversed(object)
| reversed(sequence) -> reverse iterator over values of the sequence
|
| Return a reverse iterator
```

它返回一个可以迭代的对象（关于迭代的问题，后续会详述之），不过是已经将原来的序列对象反转了。比如：

```
>>> list(reversed("abcd"))
['d', 'c', 'b', 'a']
```

很好，很强大，特别推荐使用。

对 list 的操作

任何一个行业都有自己的行话，如同古代的强盗，把撤退称之为“扯乎”一样，纵然是一个含义，但是强盗们愿意用他们自己的行业用语，俗称“黑话”。各行各业都如此。这样做的目的我理解有两个，一个是某种保密；另外一个是行外人士显示本行业的门槛，让别人感觉这个行业很高深，从业者有一定水平。

不管怎么，在 Python 和很多高级语言中，都给本来数学角度就是函数的东西，又在不同情况下有不同的称呼，如方法、类等。当然，这种称呼，其实也是为了区分函数的不同功能。

前面在对 str 进行操作的时候，有一些内置函数，比如 s.strip()，这是去掉左右空格的内置函数，也是 str 的方法。按照一惯制的对称法则，对 list 也会有一些操作方法。

在讲述字符串的时候，提到过，所有的序列，都有几种基本操作。list 当然如此。

基本操作

- len()

在交互模式中操作：

```
>>> lst
['python', 'java', 'c++']
>>> len(lst)
3
```

- +，连接两个序列

交互模式中：

```
>>> lst
['python', 'java', 'c++']
>>> alst
[1, 2, 3, 4, 5, 6]
>>> lst + alst
['python', 'java', 'c++', 1, 2, 3, 4, 5, 6]
```

- *，重复元素

交互模式中操作

```
>>> lst
['python', 'java', 'c++']
```

```
>>> lst * 3
['python', 'java', 'c++', 'python', 'java', 'c++', 'python', 'java', 'c++']
```

- `in`

列表 `lst` 还是前面的值

```
>>> "python" in lst
True
>>> "c#" in lst
False
```

- `max()` 和 `min()`

以 `int` 类型元素为例。如果不是，都是按照字符在 `ascii` 编码中所对应的数字进行比较的。

```
>>> alst
[1, 2, 3, 4, 5, 6]
>>> max(alst)
6
>>> min(alst)
1
>>> max(lst)
'python'
>>> min(lst)
'c++'
```

- `cmp()`

采用上面的方法，进行比较

```
>>> lsta = [2,3]
>>> lstb = [2,4]
>>> cmp(lsta,lstb)
-1
>>> lstc = [2]
>>> cmp(lsta,lstc)
1
>>> lstd = ['2','3']
>>> cmp(lsta,lstd)
-1
```

追加元素

```
>>> a = ["good", "python", "I"]
>>> a
['good', 'python', 'I']
>>> a.append("like")      #向 list 中添加 str 类型 "like"
>>> a
['good', 'python', 'I', 'like']
>>> a.append(100)        #向 list 中添加 int 类型 100
>>> a
['good', 'python', 'I', 'like', 100]
```

[官方文档](#)这样描述 `list.append()`方法

`list.append(x)`

Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.

从以上描述中，以及本部分的标题“追加元素”，是不是能够理解 `list.append(x)`的含义呢？即将新的元素 `x` 追加到 `list` 的尾部。

列位看官，如果您注意看上面官方文档中的那句话，应该注意到，还有后面半句： `equivalent to a[len(a):] = [x]`，意思是说 `list.append(x)`等效于：`a[len(a):]=[x]`。这也相当于告诉我们要了另外一种追加元素的方法，并且两种方法等效。

```
>>> a
['good', 'python', 'I', 'like', 100]
>>> a[len(a):]=[3]    #len(a),即得到 list 的长度，这个长度是指 list 中的元素个数。
>>> a
['good', 'python', 'I', 'like', 100, 3]
>>> len(a)
6
>>> a[6:]=[xxoo']
>>> a
['good', 'python', 'I', 'like', 100, 3, 'xxoo']
```

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qjwsir@126.com,不胜感激。

列表(2)

上一节中已经谈到，list是Python的苦力，那么它都有哪些函数呢？或者它或者对它能做什么呢？在交互模式下这么操作，就看到有关它的函数了。

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__setitem__', '__str__', '__subclasshook__']
```

上面的结果中，以双下划线开始和结尾的暂时不管，如`__add__`（以后会管的）。就剩下以下几个了：

'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort'

下面注意对这些函数进行说明和演示。这都是在编程实践中常常要用到的。

list函数

append 和 extend

[《列表\(1\)》](#)中，对list的基本操作提到了`list.append(x)`，也就是将某个元素x追加到已知的一个list后边。

除了将元素追加到list中，还能够将两个list合并，或者说将一个list追加到另外一个list中。按照前文的惯例，还是首先看[官方文档](#)中的描述：

`list.extend(L)`

Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.

向所有正在学习本内容的朋友提供一个成为优秀程序员的必备：看官方文档，是必须的。

官方文档的这句话翻译过来：

通过将所有元素追加到已知list来扩充它，相当于`a[len(a):]=L`

英语太烂，翻译太差。直接看例子，更明白

```
>>> la
[1, 2, 3]
>>> lb
['qiwsir', 'python']
>>> la.extend(lb)
>>> la
```

```
[1, 2, 3, 'qiwsir', 'python']
>>> lb
['qiwsir', 'python']
```

上面的例子，显示了如何将两个 list，一个是 la，另外一个 lb，将 lb 追加到 la 的后面，也就是把 lb 中的所有元素加入到 la 中，即让 la 扩容。

学程序一定要有好奇心，我在交互环境中，经常实验一下自己的想法，有时候是比较愚蠢的想法。

```
>>> la = [1,2,3]
>>> b = "abc"
>>> la.extend(b)
>>> la
[1, 2, 3, 'a', 'b', 'c']
>>> c = 5
>>> la.extend(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

从上面的实验中，看官能够有什么心得？原来，如果 extend(str)的时候，str 被以字符为单位拆开，然后追加到 la 里面。

如果 extend 的对象是数值型，则报错。

所以，extend 的对象是一个 list，如果是 str，则 Python 会先把它按照字符为单位转化为 list 再追加到已知 list。

不过，别忘记了前面官方文档的后半句话，它的意思是：

```
>>> la
[1, 2, 3, 'a', 'b', 'c']
>>> lb
['qiwsir', 'python']
>>> la[len(la):]=lb
>>> la
[1, 2, 3, 'a', 'b', 'c', 'qiwsir', 'python']
```

list.extend(L) 等效于 list[len(list):] = L，L 是待并入的 list

联想到到 [上一讲](#)中的一个 list 函数 list.append(), 有类似之处。

extend(...) L.extend(iterable) -- extend list by appending elements from the iterable

上面是在交互模式中输入 help(list.extend) 后得到的说明。这是非常重要而且简单的获得文档帮助的方法。

从上面内容可知，`extend` 函数也是将另外的元素增加到一个已知列表中，其元素必须是 `iterable`，什么是 `iterable`？这个从现在开始，后面会经常遇到，所以是要搞清楚的。

`iterable`，中文含义是“可迭代的”。在 Python 中，还有一个词，就是 `iterator`，这个叫做“迭代器”。这两者有着区别和联系。不过，这里暂且不说那么多，说多了就容易糊涂，我也糊涂了。

为了解释 `iterable`(可迭代的)，又引入了一个词“迭代”，什么是迭代呢？

尽管我们很多文档是用英文写的，但是，如果你能充分利用汉语来理解某些名词，是非常有帮助的。因为在汉语中，不仅仅表音，而且能从词语组合中体会到该术语的含义。比如“激光”，这是汉语。英语是从“light amplification by stimulated emission of radiation”化出来的“laser”，它是一个造出来的词。因为此前人们不知道那种条件下发出来的是什么。但是汉语不然，反正用一个“光”就可以概括了，只不过这个“光”不是传统概念中的“光”，而是由于“受激”辐射得到的光，故名“激光”。是不是汉语很牛叉？

“迭”在汉语中的意思是“屡次,反复”。如:高潮迭起。那么跟“代”组合，就可以理解为“反复‘代’”，是不是有点“子子孙孙”的意思了？“结婚-生子-子成长-结婚-生子-子成长-...”，你是不是也在这个“迭代”的过程中呢？

给个稍微严格的定义，来自维基百科。“迭代是重复反馈过程的活动，其目的通常是为了接近并到达所需的目标或结果。”

某些类型的对象是“可迭代”(`iterable`)的，这类数据类型有共同的特点。如何判断一个对象是不是可迭代的？下面演示一种方法。事实上还有别的方式。

```
>>> astr = "Python"
>>> hasattr(astr,'__iter__')
False
```

这里用内建函数 `hasattr()` 判断一个字符串是否是可迭代的，返回了 `False`。用同样的方式可以判断：

```
>>> alst = [1,2]
>>> hasattr(alst,'__iter__')
True
>>> hasattr(3, '__iter__')
False
```

`hasattr()` 的判断本质就是看那个类型中是否有 `__iter__` 函数。看官可以用 `dir()` 找一找，在数字、字符串、列表中，谁有 `__iter__`。同样还可找一找 `dict,tuple` 两种类型对象是否含有这个方法。

以上穿插了一个新的概念“`iterable`”（可迭代的），现在回到 `extend` 上。这个函数需要的参数就是 `iterable` 类型的对象。

```
>>> new = [1,2,3]
>>> lst = ['Python','qiwsir']
>>> lst.extend(new)
>>> lst
['Python', 'qiwsir', 1, 2, 3]
>>> new
[1, 2, 3]
```

通过 `extend` 函数，将`[1,2,3]`中的每个元素都拿出来，然后塞到 `lst` 里面，从而得到了一个跟原来的对象元素不一样的列表，后面的比原来的多了三个元素。上面说的有点啰嗦，只不过是为了把过程完整表达出来。

还要关注一下，从上面的演示中可以看出，`lst` 经过 `extend` 函数操作之后，变成了一个貌似“新”的列表。这句话好像有点别扭，“貌似新”的，之所以这么说，是因为对“新的”可能有不同的理解。不妨深挖一下。

```
>>> new = [1,2,3]
>>> id(new)
3072383244L

>>> lst = ['python', 'qiwsir']
>>> id(lst)
3069501420L
```

用 `id()` 能够看到两个列表分别在内存中的“窝”的编号。

```
>>> lst.extend(new)
>>> lst
['python', 'qiwsir', 1, 2, 3]
>>> id(lst)
3069501420L
```

看官注意到没有，虽然 `lst` 经过 `extend()` 方法之后，比原来扩容了，但是，并没有离开原来的“窝”，也就是在内存中，还是“旧”的，只不过里面的内容增多了。相当于两口之家，经过一番云雨之后，又增加了一个小宝宝，那么这个家是“新”的还是“旧”的呢？角度不同或许说法不一了。

这就是列表的一个重要特征：列表是可以修改的。这种修改，不是复制一个新的，而是在原地进行修改。

其实，`append()` 对列表的操作也是如此，不妨用同样的方式看看。

说明：虽然这里的 `lst` 内容和上面的一样，但是，我从新在 shell 中输入，所以 `id` 会变化。也就是内存分配的“窝”的编号变了。

```
>>> lst = ['Python','qiwsir']
>>> id(lst)
3069501388L
>>> lst.append(new)
```

```
>>> lst
['Python', 'qiwsir', [1, 2, 3]]
>>> id(lst)
3069501388L
```

显然，`append()` 也是原地修改列表。

如果，对于 `extend()`，提供的不是 iterable 类型对象，会如何呢？

```
>>> lst.extend("itdiffer")
>>> lst
['python', 'qiwsir', 'i', 't', 'd', 'i', 'f', 'e', 'r']
```

它把一个字符串 "itdiffer" 转化为 ['i', 't', 'd', 'i', 'f', 'e', 'r']，然后将这个列表作为参数，提供给 `extend`，并将列表中的元素塞入原来的列表中。

```
>>> num_lst = [1,2,3]
>>> num_lst.extend(8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

这就报错了。错误提示中告诉我们，那个数字 8，是 int 类型的对象，不是 iterable 的。

这里讲述的两个让列表扩容的函数 `append()` 和 `extend()`。从上面的演示中，可以看到他们有相同的地方：

- 都是原地修改列表
- 既然是原地修改，就不返回值

原地修改没有返回值，就不能赋值给某个变量。

```
>>> one = ["good", "good", "study"]
>>> another = one.extend(["day", "day", "up"]) #对于没有提供返回值的函数，如果要这样，结果是：
>>> another
#这样的，什么也没有得到。
>>> one
['good', 'good', 'study', 'day', 'day', 'up']
```

那么两者有什么不一样呢？看下面例子：

```
>>> lst = [1,2,3]
>>> lst.append(["qiwsir", "github"])
>>> lst
[1, 2, 3, ['qiwsir', 'github']] #append 的结果
>>> len(lst)
4
```

```
>>> lst2 = [1,2,3]
>>> lst2.extend(["qiwsir","github"])
>>> lst2
[1, 2, 3, 'qiwsir', 'github'] #extend 的结果
>>> len(lst2)
5
```

append 是整建制地追加， extend 是个体化扩编。

count

上面的 len(L)，可得到 list 的长度，也就是 list 中有多少个元素。python 的 list 还有一个函数，就是数一数某个元素在该 list 中出现多少次，也就是某个元素有多少个。官方文档是这么说的：

```
list.count(x)

Return the number of times x appears in the list.
```

一定要不断实验，才能理解文档中精炼的表达。

```
>>> la = [1,2,1,1,3]
>>> la.count(1)
3
>>> la.append('a')
>>> la.append('a')
>>> la
[1, 2, 1, 1, 3, 'a', 'a']
>>> la.count('a')
2
>>> la.count(2)
1
>>> la.count(5) #NOTE:la 中没有 5,但是如果用这种方法找，不报错，返回的是数字 0
0
```

index

[《列表\(1\)》](#) 中已经提到，这里不赘述，但是为了完整，也占个位置吧。

```
>>> la
[1, 2, 3, 'a', 'b', 'c', 'qiwsir', 'python']
>>> la.index(3)
2
>>> la.index('qi') #如果不存在，就报错
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
ValueError: 'qi' is not in list
>>> la.index('qiwsir')
6
```

list.index(x)，x 是 list 中的一个元素，这样就能够检索到该元素在 list 中的位置了。这才是真正的索引，注意那个英文单词 index。

依然是上一条官方解释：

```
list.index(x)
```

Return the index in the list of the first item whose value is x. It is an error if there is no such item.

是不是说的非常清楚明白了？

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qjwsir@126.com,不胜感激。

列表(3)

接着上节内容。下面是上节中说好要介绍的列表方法：

```
'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort'
```

已经在上节讲解了前四个。

继续。

list 函数

insert

前面有向 list 中追加元素的方法，那个追加是且只能是将新元素添加在 list 的最后一个。如：

```
>>> all_users = ["qiwsir", "github"]
>>> all_users.append("io")
>>> all_users
['qiwsir', 'github', 'io']
```

与 `list.append(x)` 类似，`list.insert(i,x)` 也是对 list 元素的增加。只不过是可以在任何位置增加一个元素。

还是先看[官方文档来理解](#)：

`list.insert(i, x)`

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

这次就不翻译了。如果看不懂英语，怎么了解贵国呢？一定要硬着头皮看英语，不仅能够学好程序，更能...（此处省略两千字）

根据官方文档的说明，我们做下面的实验，请看官从实验中理解：

```
>>> all_users
['qiwsir', 'github', 'io']
>>> all_users.insert("python")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: insert() takes exactly 2 arguments (1 given)
```

请注意看报错的提示信息，`insert()` 应该供给两个参数，但是这里只给了一个。所以报错没商量啦。

```
>>> all_users.insert(0,"python")
>>> all_users
['python', 'qiwsir', 'github', 'io']

>>> all_users.insert(1,"http://")
>>> all_users
['python', 'http://', 'qiwsir', 'github', 'io']
```

`list.insert(i, x)` 中的 `i` 是将元素 `x` 插入到 `list` 中的位置，即将 `x` 插入到索引值是 `i` 的元素前面。注意，索引是从 0 开始的。

有一种操作，挺有意思的，如下：

```
>>> length = len(all_users)
>>> length
5
>>> all_users.insert(length,"algorithm")
>>> all_users
['python', 'http://', 'qiwsir', 'github', 'io', 'algorithm']
```

在 `all_users` 中，没有索引最大到 4，如果要 `all_users.insert(5,"algorithm")`，则表示将 `"algorithm"` 插入到索引值是 5 的前面，但是没有。换个说法，5 前面就是 4 的后面。所以，就是追加了。

其实，还可以这样：

```
>>> a = [1,2,3]
>>> a.insert(9,777)
>>> a
[1, 2, 3, 777]
```

也就是说，如果遇到那个 `i` 已经超过了最大索引值，会自动将所要插入的元素放到列表的尾部，即追加。

pop 和 remove

`list` 中的元素，不仅能增加，还能被删除。删除 `list` 元素的方法有两个，它们分别是：

`list.remove(x)`

Remove the first item from the list whose value is `x`. It is an error if there is no such item.

`list.pop([i])`

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the `i` in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

我这里讲授 Python，有一个习惯，就是用学习物理的方法。如果看官当初物理没有学好，那么一定是没有用这种方法，或者你的老师没有用这种教学法。这种方法就是：自己先实验，然后总结规律。

先实验 `list.remove(x)`，注意看上面的描述。这是一个能够删除 list 元素的方法，同时上面说明告诉我们，如果 `x` 没有在 list 中，会报错。

```
>>> all_users
['Python', 'http://', 'qiwsir', 'github', 'io', 'algorithm']
>>> all_users.remove("http://")
>>> all_users      #的确是把"http://"删除了
['Python', 'qiwsir', 'github', 'io', 'algorithm']

>>> all_users.remove("tianchao")    #原 list 中没有“tianchao”，要删除，就报错。
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list

>>> lst = ["python", "java", "python", "c"]
>>> lst.remove("python")
>>> lst
['java', 'python', 'c']
```

重点解释一下第三个操作。哦，忘记一个提醒，我在前面的很多操作中，也都给列表的变量命名为 `lst`，但是不是 `list`，为什么呢？因为 `list` 是 Python 的保留字。

还是继续第三段操作，列表中有两个'Python'字符串，当删除后，发现结果只删除了第一个'Python'字符串，第二个还在。请仔细看前面的文档说明：`remove the first item ...`

注意两点：

- 如果正确删除，不会有任何反馈。没有消息就是好消息。并且是对列表进行原地修改。
- 如果所删除的内容不在 list 中，就报错。注意阅读报错信息：`x not in list`

什么是保留字？在 Python 中，当然别的语言中也是如此啦。某些词语或者拼写是不能被用户拿来做变量 / 函数 / 类等命名，因为它们已经被语言本身先占用了。这些就是所谓保留字。在 Python 中，以下是保留字，不能用于你自己变成中的任何命名。

and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield

这些保留字，都是我们在编程中要用到的。有的你已经在前面遇到了。

看官是不是想到一个问题？如果能够在删除之前，先判断一下这个元素是不是在 list 中，如果在就删，不在就不删，不是更智能吗？

如果看官想到这里，就是在编程的旅程上一进步。Python 的确让我们这么做。

```
>>> all_users
['python', 'qiwsir', 'github', 'io', 'algorithm']
>>> "Python" in all_users      #这里用 in 来判断一个元素是否在 list 中，在则返回 True，否则返回 False
True

>>> if "Python" in all_users:
...     all_users.remove("python")
...     print all_users
... else:
...     print "'Python' is not in all_users"
...
['qiwsir', 'github', 'io', 'algorithm']  #删除了"Python"元素

>>> if "Python" in all_users:
...     all_users.remove("Python")
...     print all_users
... else:
...     print "'Python' is not in all_users"
...
'Python' is not in all_users      #因为已经删除了，所以就没有了。
```

上述代码，就是两段小程序，我是在交互模式中运行的，相当于小实验。这里其实用了一个后面才会讲到的东西：if-else语句。不过，我觉得即使没有学习，你也能看懂，因为它非常接近自然语言了。

另外一个删除 list.pop([i])会怎么样呢？看看文档，做做实验。

```
>>> all_users
['qiwsir', 'github', 'io', 'algorithm']
>>> all_users.pop()    #list.pop([i]),圆括号里面是[i]，表示这个序号是可选的
'algorithm'          #如果不写，就如同这个操作，默认删除最后一个，并且将该结果返回

>>> all_users
['qiwsir', 'github', 'io']

>>> all_users.pop(1)    #指定删除编号为 1 的元素"github"
```

```
'github'

>>> all_users
['qiwsir', 'io']
>>> all_users.pop()
'io'

>>> all_users      #只有一个元素了，该元素编号是 0
['qiwsir']
>>> all_users.pop(1)  #但是非要删除编号为 1 的元素，结果报错。注意看报错信息
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop index out of range    #删除索引超出范围，就是 1 不在 list 的编号范围之内
```

简单总结一下，`list.remove(x)` 中的参数是列表中元素，即删除某个元素；`list.pop([i])` 中的 `i` 是列表中元素的索引值，这个 `i` 用方括号包裹起来，意味着还可以不写任何索引值，如上面操作结果，就是删除列表的最后一个。

给看官留下一个思考题，如果要像前面那样，能不能事先判断一下要删除的编号是不是在 `list` 的长度范围（用 `len(list)` 获取长度）以内？然后进行删除或者不删除操作。

reverse

`reverse` 比较简单，就是把列表的元素顺序反过来。

```
>>> a = [3,5,1,6]
>>> a.reverse()
>>> a
[6, 1, 5, 3]
```

注意，是原地反过来，不是另外生成一个新的列表。所以，它没有返回值。跟这个类似的有一个内建函数 `reversed`，建议读者了解一下这个函数的使用方法。

因为 `list.reverse()` 不返回值，所以不能实现对列表的反向迭代，如果要这么做，可以使用 `reversed` 函数。

sort

`sort` 就是对列表进行排序。帮助文档中这么写的：

`sort(...)`

`L.sort(cmp=None, key=None, reverse=False) -- stable sort IN PLACE; cmp(x, y) -> -1, 0, 1`

```
>>> a = [6, 1, 5, 3]
>>> a.sort()
>>> a
[1, 3, 5, 6]
```

list.sort() 也是让列表进行原地修改，没有返回值。默认情况，如上面操作，实现的是从小到大的排序。

```
>>> a.sort(reverse=True)
>>> a
[6, 5, 3, 1]
```

这样做，就实现了从大到小的排序。

在前面的函数说明中，还有一个参数 key，这个怎么用呢？不知道看官是否用过电子表格，里面就是能够设置按照哪个关键字进行排序。这里也是如此。

```
>>> lst = ["Python", "java", "c", "pascal", "basic"]
>>> lst.sort(key=len)
>>> lst
['c', 'java', 'basic', 'Python', 'pascal']
```

这是以字符串的长度为关键词进行排序。

对于排序，也有一个更为常用的内建函数 sorted。

顺便指出，排序是一个非常有研究价值的话题。不仅仅是现在这么一个函数。有兴趣的读者可以去网上搜一下排序相关知识。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qjwsir@126.com,不胜感激。

回顾 list 和 str

list 和 str 两种类型数据，有不少相似的地方，也有很大的区别。本讲对她们做个简要比较，同时也是对前面有关两者知识复习一下，所谓“温故而知新”。

相同点

都属于序列类型的数据

所谓序列类型的数据，就是说它的每一个元素都可以通过指定一个编号，行话叫做“偏移量”的方式得到，而要想一次得到多个元素，可以使用切片。偏移量从 0 开始，总元素数减 1 结束。

例如：

```
>>> welcome_str = "Welcome you"
>>> welcome_str[0]
'W'
>>> welcome_str[1]
'e'
>>> welcome_str[len(welcome_str)-1]
'u'
>>> welcome_str[:4]
'Welc'
>>> a = "python"
>>> a*3
'pythonpythonpython'

>>> git_list = ["qiwsir", "github", "io"]
>>> git_list[0]
'qiwsir'
>>> git_list[len(git_list)-1]
'io'
>>> git_list[0:2]
['qiwsir', 'github']
>>> b = ['qiwsir']
>>> b*7
['qiwsir', 'qiwsir', 'qiwsir', 'qiwsir', 'qiwsir', 'qiwsir', 'qiwsir']
```

对于此类数据，下面一些操作是类似的：

```

>>> first = "hello,world"
>>> welcome_str
'Welcome you'
>>> first+", "+welcome_str #用 + 号连接 str
'hello,world,Welcome you'
>>> welcome_str      #原来的 str 没有受到影响，即上面的+号连接后重新生成了一个字符串
'Welcome you'
>>> first
'hello,world'

>>> language = ['python']
>>> git_list
['qiwsir', 'github', 'io']
>>> language + git_list #用 + 号连接 list，得到一个新的 list
['python', 'qiwsir', 'github', 'io']
>>> git_list
['qiwsir', 'github', 'io']
>>> language
['Python']

>>> len(welcome_str) # 得到字符数
11
>>> len(git_list)    # 得到元素数
3

```

另外，前面的讲述中已经说明了关于序列的基本操作，此处不再重复。

区别

list 和 str 的最大区别是：list 是可以改变的，str 不可变。这个怎么理解呢？

首先看对 list 的这些操作，其特点是在原处将 list 进行了修改：

```

>>> git_list
['qiwsir', 'github', 'io']

>>> git_list.append("python")
>>> git_list
['qiwsir', 'github', 'io', 'python']

>>> git_list[1]
'github'
>>> git_list[1] = 'github.com'
>>> git_list

```

```
['qiwsir', 'github.com', 'io', 'python']

>>> git_list.insert(1,"algorithm")
>>> git_list
['qiwsir', 'algorithm', 'github.com', 'io', 'python']

>>> git_list.pop()
'python'

>>> del git_list[1]
>>> git_list
['qiwsir', 'github.com', 'io']
```

以上这些操作，如果用在 str 上，都会报错，比如：

```
>>> welcome_str
'Welcome you'

>>> welcome_str[1]='E'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

>>> del welcome_str[1]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object doesn't support item deletion

>>> welcome_str.append("E")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'append'
```

如果要修改一个 str，不得不这样。

```
>>> welcome_str
'Welcome you'
>>> welcome_str[0]+"E"+welcome_str[2:] #从新生成一个 str
'WEcome you'
>>> welcome_str          #对原来的没有任何影响
'Welcome you'
```

其实，在这种做法中，相当于重新生成了一个 str。

多维 list

这个也应该算是两者的区别了，虽然有点牵强。在 str 中，里面的每个元素只能是字符，在 list 中，元素可以是任何类型的数据。前面见的多是数字或者字符，其实还可以这样：

```
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]]
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]]
>>> matrix[0][1]
2
>>> mult = [[1,2,3],['a','b','c'],'d','e']
>>> mult
[[1, 2, 3], ['a', 'b', 'c'], 'd', 'e']
>>> mult[1][1]
'b'
>>> mult[2]
'd'
```

以上显示了多维 list 以及访问方式。在多维的情况下，里面的 list 被当成一个元素对待。

list 和 str 转化

以下涉及到的 `split()` 和 `join()` 在前面字符串部分已经见过。一回生，二回熟，这次再见面，特别是在已经学习了列表的基础上，应该有更深刻的理解。

`str.split()`

这个内置函数实现的是将 str 转化为 list。其中 `str=""` 是分隔符。

在看例子之前，请看官在交互模式下做如下操作：

```
>>> help(str.split)
```

得到了对这个内置函数的完整说明。特别强调：这是一种非常好的学习方法

`split(...)` `S.split([sep [,maxsplit]]) -> list of strings`

Return a list of the words in the string S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

不管是否看懂上面这段话，都可以看例子。还是希望看官能够理解上面的内容。

```
>>> line = "Hello.I am qiwsi.Welcome you."
>>> line.split(".") #以英文的句点为分隔符，得到 list
['Hello', 'I am qiwsi', 'Welcome you', '']
>>> line.split(".",1) #这个 1,就是表达了上文中的： If maxsplit is given, at most maxsplit splits are done.
['Hello', 'I am qiwsi.Welcome you.']
>>> name = "Albert Ainstain" #也有可能用空格来做为分隔符
>>> name.split(" ")
['Albert', 'Ainstain']
```

下面的例子，让你更有点惊奇了。

```
>>> s = "I am, writing\npython\tbook on line" #这个字符串中有空格，逗号，换行\n, tab 缩进\t 符号
>>> print s      #输出之后的样式
I am, writing
python book on line
>>> s.split()    #用 split(),但是括号中不输入任何参数
['I', 'am,', 'writing', 'Python', 'book', 'on', 'line']
```

如果 split()不输入任何参数，显示就是见到任何分割符号，就用其分割了。

"[sep]".join(list)

join 可以说是 split 的逆运算，举例：

```
>>> name
['Albert', 'Ainstain']
>>> "".join(name)    #将 list 中的元素连接起来，但是没有连接符，表示一个一个紧邻着
'AlbertAinstain'
>>> ".join(name)    #以英文的句点做为连接分隔符
'Albert.Ainstain'
>>> " ".join(name)   #以空格做为连接的分隔符
'Albert Ainstain'
```

回到上面那个神奇的例子中，可以这么使用 join.

```
>>> s = "I am, writing\npython\tbook on line"
>>> print s
I am, writing
python book on line
>>> s.split()
```

```
[", 'am,', 'writing', 'Python', 'book', 'on', 'line']  
>>> " ".join(s.split())      #重新连接，不过有一点遗憾， am 后面逗号还是有的。怎么去掉?  
'I am, writing python book on line'
```

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

元组

定义

先看一个例子：

```
>>> # 变量引用 str
>>> s = "abc"
>>> s
'abc'

>>> #如果这样写，就会是...
>>> t = 123,'abc',['come','here']
>>> t
(123, 'abc', ['come', 'here'])
```

上面例子中看到的变量 t，并没有报错，也没有“最后一个有效”，而是将对象做一个新的数据类型：tuple（元组），赋值给了变量 t。

元组是用圆括号括起来的，其中的元素之间用逗号隔开。（都是英文半角）

元组中的元素类型是任意的 Python 数据。

这种类型，可以歪着想，所谓“元”组，就是用“圆”括号啦。

其实，你不应该对元组陌生，还记得前面讲述字符串的格式化输出时，有这样一种方式：

```
>>> print "I love %s, and I am a %s" % ('python', 'programmer')
I love Python, and I am a programmer
```

这里的圆括号，就是一个元组。

显然，tuple 是一种序列类型的数据，这点上跟 list/str 类似。它的特点就是其中的元素不能更改，这点上跟 list 不同，倒是跟 str 类似；它的元素又可以是任何类型的数据，这点上跟 list 相同，但不同于 str。

```
>>> t = 1,"23",[123,"abc"],("python","learn") #元素多样性，近 list
>>> t
(1, '23', [123, 'abc'], ('python', 'learn'))

>>> t[0] = 8                      #不能原地修改，近 str
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
```

```
>>> t.append("no")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>>
```

从上面的简单比较似乎可以认为，tuple 就是一个融合了部分 list 和部分 str 属性的杂交产物。此言有理。

索引和切片

因为前面有了关于列表和字符串的知识，它们都是序列类型，元组也是。因此，元组的基本操作就和它们是一样的。

例如：

```
>>> t
(1, '23', [123, 'abc'], ('python', 'learn'))
>>> t[2]
[123, 'abc']
>>> t[1:]
('23', [123, 'abc'], ('python', 'learn'))

>>> t[2][0]  #还能这样呀，哦对了，list 中也能这样
123
>>> t[3][1]
'learn'
```

关于序列的基本操作在 tuple 上的表现，就不一一展示了。看官可以去试试。

但是这里要特别提醒，如果一个元组中只有一个元素的时候，应该在该元素后面加一个半角的英文逗号。

```
>>> a = (3)
>>> type(a)
<type 'int'>

>>> b = (3,)
>>> type(b)
<type 'tuple'>
```

以上面的例子说明，如果不加那个逗号，就不是元组，加了才是。这也是为了避免让 Python 误解你要表达的内容。

顺便补充：如果要想看一个对象是什么类型，可以使用 `type()` 函数，然后就返回该对象的类型。

所有在 `list` 中可以修改 `list` 的方法，在 `tuple` 中，都失效。

分别用 `list()` 和 `tuple()` 能够实现两者的转化：

```
>>> t
(1, '23', [123, 'abc'], ('python', 'learn'))
>>> tls = list(t)           #tuple-->list
>>> tls
[1, '23', [123, 'abc'], ('python', 'learn')]

>>> t_tuple = tuple(tls)    #list-->tuple
>>> t_tuple
(1, '23', [123, 'abc'], ('python', 'learn'))
```

tuple 用在哪里？

既然它是 `list` 和 `str` 的杂合，它有什么用途呢？不是用 `list` 和 `str` 都可以了吗？

在很多时候，的确是用 `list` 和 `str` 都可以了。但是，看官不要忘记，我们用计算机语言解决的问题不都是简单问题，就如同我们的自然语言一样，虽然有的词汇看似可有可无，用别的也能替换之，但是我们依然需要在某些情况下使用它们。

一般认为，`tuple` 有这类特点，并且也是它使用的情景：

- Tuple 比 `list` 操作速度快。如果您定义了一个值的常量集，并且唯一要用它做的是不断地遍历它，请使用 `tuple` 代替 `list`。
- 如果对不需要修改的数据进行“写保护”，可以使代码更安全。使用 `tuple` 而不是 `list` 如同拥有一个隐含的 `assert` 语句，说明这一数据是常量。如果必须要改变这些值，则需要执行 `tuple` 到 `list` 的转换（需要使用一个特殊的函数）。
- Tuples 可以在 `dictionary`（字典，后面要讲述）中被用做 `key`，但是 `list` 不行。`Dictionary key` 必须是不可变的。Tuple 本身是不可改变的，但是如果您有一个 `list` 的 `tuple`，那就认为是可变的了，用做 `dictionary key` 就是不安全的。只有字符串、整数或其它对 `dictionary` 安全的 `tuple` 才可以用作 `dictionary key`。
- Tuples 可以用在字符串格式化中。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qjwsir@126.com, 不胜感激。

字典(1)

字典，这个东西你现在还用吗？随着网络的发展，用的人越来越少了。不少人习惯于在网上搜索，不仅有 web 版，以至于已经有手机版的各种字典了。我在上小学的时候曾经用过一本小小的《新华字典》，记得是拾了不少废品，然后换钱，最终花费了 1.01 元人民币买的。

《新华字典》是中国第一部现代汉语字典。最早的名字叫《伍记小字典》，但未能编纂完成。自 1953 年，开始重编，其凡例完全采用《伍记小字典》。从 1953 年开始出版，经过反复修订，但是以 1957 年商务印书馆出版的《新华字典》作为第一版。原由新华辞书社编写，1956 年并入中科院语言研究所（现中国社科院语言研究所）词典编辑室。新华字典由商务印书馆出版。历经几代上百名专家学者 10 余次大规模的修订，重印 200 多次。成为迄今为止世界出版史上最高发行量的字典。

这里讲到字典，不是为了回忆青葱岁月。而是提醒看官想想我们如何使用字典：先查索引（不管是拼音还是偏旁查字），然后通过索引找到相应内容。不用从头开始一页一页地找。

这种方法能够快捷的找到目标。

正是基于这种需要，Python 中有了一种叫做 dictionary 的数据类型，翻译过来就是“字典”，用 dict 表示。

假设一种需要，要存储城市和电话区号，苏州的区号是 0512，唐山的是 0315，北京的是 011，上海的是 012。用前面已经学习过的知识，可以这么做：

```
>>> citys = ["suzhou", "tangshan", "beijing", "shanghai"]
>>> city_codes = ["0512", "0315", "011", "012"]
```

用一个列表来存储城市名称，然后用另外一个列表，一一对应地保存区号。假如要输出苏州的区号，可以这么做：

```
>>> print "{} : {}".format(citys[0], city_codes[0])
suzhou : 0512
```

请特别注意，我在 city_codes 中，表示区号的元素没有用整数型，而是使用了字符串类型，你知道为什么吗？如果用整数，就是这样的。

```
>>> suzhou_code = 0512
>>> print suzhou_code
330
```

怎么会这样？原来在 Python 中，如果按照上面那样做，0512 并没有被认为是一个八进制的数，用 print 打印的时候，将它转换为了十进制输出。关于进制转换问题，看官可以网上搜索一下有关资料。此处不详述。一般时用几个内建函数实现：int()，bin()，oct()，hex()

这样来看，用两个列表分别来存储城市和区号，似乎能够解决问题。但是，这不是最好的选择，至少在 Python 里面。因为 Python 还提供了另外一种方案，那就是字典(dict)。

创建 dict

方法 1：

创建一个空的 dict，这个空 dict，可以在以后向里面加东西用。

```
>>> mydict = {}
>>> mydict
{}
```

不要小看“空”，“空即是色，色即是空”，在编程中，“空”是很重要。一般带“空”字的人都很有名，比如孙悟空，哦。好像他应该是猴、或者是神。举一个人的名字，带“空”字，你懂得。

创建有内容的 dict。

```
>>> person = {"name": "qiwsir", "site": "qiwsir.github.io", "language": "python"}
>>> person
{'name': 'qiwsir', 'language': 'python', 'site': 'qiwsir.github.io'}
```

"name": "qiwsir"，有一个优雅的名字：键值对。前面的 name 叫做键（key），后面的 qiwsir 是前面的键所对应的值(value)。在一个 dict 中，键是唯一的，不能重复。值则是对应于键，值可以重复。键值之间用(:)英文的分号，每一对键值之间用英文的逗号(,)隔开。

```
>>> person['name2'] = "qiwsir" #这是一种向 dict 中增加键值对的方法
>>> person
{'name2': 'qiwsir', 'name': 'qiwsir', 'language': 'Python', 'site': 'qiwsir.github.io'}
```

用这样的方法可以向一个 dict 类型的数据中增加“键值对”，也可以说是增加数值。那么，增加了值之后，那个字典还是原来的吗？也就是也要同样探讨一下，字典是否能原地修改？（列表可以，所以列表是可变的；字符串和元组都不行，所以它们是不可变的。）

```
>>> ad = {}
>>> id(ad)
3072770636L
>>> ad["name"] = "qiwsir"
```

```
>>> ad
{'name': 'qiwsir'}
>>> id(ad)
3072770636L
```

实验表明，字典可以原地修改，即它是可变的。

方法 2：

利用元组在建构字典，方法如下：

```
>>> name = (["first", "Google"], ["second", "Yahoo"])
>>> website = dict(name)
>>> website
{'second': 'Yahoo', 'first': 'Google'}
```

或者用这样的方法：

```
>>> ad = dict(name="qiwsir", age=42)
>>> ad
{'age': 42, 'name': 'qiwsir'}
```

方法 3：

这个方法，跟上面的不同在于使用 fromkeys

```
>>> website = {}.fromkeys(("third", "forth"), "facebook")
>>> website
{'forth': 'facebook', 'third': 'facebook'}
```

需要提醒的是，这种方法是重新建立一个 dict。

需要提醒注意的是，在字典中的“键”，必须是不可变的数据类型；“值”可以是任意数据类型。

```
>>> dd = {(1,2):1}
>>> dd
{(1, 2): 1}
>>> dd = {[1,2]:1}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

访问 dict 的值

dict 数据类型是以键值对的形式存储数据的，所以，只要知道键，就能得到值。这本质上就是一种映射关系。

映射，就好比“物体”和“影子”的关系，“形影相吊”，两者之间是映射关系。此外，映射也是一个严格数学概念：A 是非空集合，A 到 B 的映射是指：A 中每个元素都对应到 B 中的某个元素。

既然是映射，就可以通过字典的“键”找到相应的“值”。

```
>>> person
{'name2': 'qiwsir', 'name': 'qiwsir', 'language': 'python', 'site': 'qiwsir.github.io'}
>>> person['name']
'qiwsir'
>>> person['language']
'python'
```

如同前面所讲，通过“键”能够增加 dict 中的“值”，通过“键”能够改变 dict 中的“值”，通过“键”也能够访问 dict 中的“值”。

本节开头那个城市和区号的关系，也可以用字典来存储和读取。

```
>>> city_code = {"suzhou": "0512", "tangshan": "0315", "beijing": "011", "shanghai": "012"}
>>> print city_code["suzhou"]
0512
```

既然 dict 是键值对的映射，就不用考虑所谓“排序”问题了，只要通过键就能找到值，至于这个键值对位置在哪里就不用考虑了。比如，刚才建立的 city_code

```
>>> city_code
{'suzhou': '0512', 'beijing': '011', 'shanghai': '012', 'tangshan': '0315'}
```

虽然这里显示的和刚刚赋值的时候顺序有别，但是不影响读取其中的值。

在 list 中，得到值是用索引的方法。那么在字典中有索引吗？当然没有，因为它没有顺序，哪里来的索引呢？所以，在字典中就不要什么索引和切片了。

dict 中的这类以键值对的映射方式存储数据，是一种非常高效的方法，比如要读取值得时候，如果用列表，Python 需要从头开始读，直到找到指定的那个索引值。但是，在 dict 中是通过“键”来得到值。要高效得多。正是这个特点，键值对这样的形式可以用来存储大规模的数据，因为检索快捷。规模越大越明显。所以，mongodb 这种非关系型数据库在大数据方面比较流行了。

基本操作

字典虽然跟列表有很大的区别，但是依然有不少类似的地方。它的基本操作：

- len(d)，返回字典(d)中的键值对的数量

- `d[key]`, 返回字典(d)中的键(key)的值
- `d[key]=value`, 将值(value)赋给字典(d)中的键(key)
- `del d[key]`, 删除字典(d)的键(key)项 (将该键值对删除)
- `key in d`, 检查字典(d)中是否含有键为 key 的项

下面依次进行演示。

```
>>> city_code
{'suzhou': '0512', 'beijing': '011', 'shanghai': '012', 'tangshan': '0315'}
>>> len(city_code)
4
```

以 `city_code` 为操作对象, `len(city_code)` 的值是 4, 表明有四组键值对, 也可以说是四项。

```
>>> city_code["nanjing"] = "025"
>>> city_code
{'suzhou': '0512', 'beijing': '011', 'shanghai': '012', 'tangshan': '0315', 'nanjing': '025'}
```

向其中增加一项

```
>>> city_code["beijing"] = "010"
>>> city_code
{'suzhou': '0512', 'beijing': '010', 'shanghai': '012', 'tangshan': '0315', 'nanjing': '025'}
```

突然发现北京的区号写错了。可以这样修改。这进一步说明字典是可变的。

```
>>> city_code["shanghai"]
'012'
>>> del city_code["shanghai"]
```

通过 `city_code["shanghai"]` 能够查看到该键(key)所对应的值(value), 结果发现也错了。干脆删除, 用 `del`, 将那一项都删掉。

```
>>> city_code["shanghai"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'shanghai'
>>> "shanghai" in city_code
False
```

因为键是"shanghai"的那个键值对项已经删除了, 随意不能找到, 用 `in` 来看看, 返回的是 `False`。

```
>>> city_code
{'suzhou': '0512', 'beijing': '010', 'tangshan': '0315', 'nanjing': '025'}
```

真的删除了哦。没有了。

字符串格式化输出

这是一个前面已经探讨过的话题，请参看《字符串(4)》，这里再次提到，就是因为用字典也可以实现格式化字符串的目的。虽然在《字符串(4)》那节中已经有了简单演示，但是我还是愿意重复一下。

```
>>> city_code = {"suzhou":"0512", "tangshan":"0315", "hangzhou":"0571"}
>>> " Suzhou is a beautiful city, its area code is %(suzhou)s" % city_code
' Suzhou is a beautiful city, its area code is 0512'
```

这种写法是非常简洁，而且很有意思的。有人说它简直是酷。

其实，更酷还是下面的——模板

在做网页开发的时候，通常要用到模板，也就是你只需要写好 HTML 代码，然后将某些部位空出来，等着 Python 后台提供相应的数据即可。当然，下面所演示的是玩具代码，基本没有什么使用价值，因为在真实的网站开发中，这样的姿势很少用上。但是，它绝非花拳绣腿，而是你能够明了其本质，至少了解到一种格式化方法的应用。

```
>>> temp = "<html><head><title>%(<lang>s</lang><title><body><p>My name is %(<name>s.</p></body></head></html>""
>>> my = {"name":"qiwsir", "lang":"python"}
>>> temp % my
'<html><head><title>python</title><body><p>My name is qiwsir.</p></body></head></html>'
```

temp 就是所谓的模板，在双引号所包裹的实质上是一段 HTML 代码。然后在 dict 中写好一些数据，按照模板的要求在相应位置显示对应的数据。

是不是一个很有意思的屠龙之技？

什么是 HTML？下面是在《维基百科》上抄录的：

超文本标记语言（英文：HyperText Markup Language，HTML）是为「网页创建和其它可在网页浏览器中看到的信息」设计的一种标记语言。HTML 被用来结构化信息——例如标题、段落和列表等等，也可用来在一定程度上描述文档的外观和语义。1982 年由蒂姆·伯纳斯-李创建，由 IETF 用简化的 SGML（标准通用标记语言）语法进行进一步发展的 HTML，后来成为国际标准，由万维网联盟（W3C）维护。

HTML 经过发展，现在已经到 HTML5 了。现在的 HTML 设计，更强调“响应式”设计，就是能够兼顾 PC、手机和各种 PAD 的不同尺寸的显示器浏览。如果要开发一个网站，一定要做到“响应式”设计，否则就只能在 PC 上看，在手机上看就不得不左右移动。

知识

什么是关联数组？以下解释来自[维基百科](#)

在计算机科学中，关联数组（英语：Associative Array），又称映射（Map）、字典（Dictionary）是一个抽象的数据结构，它包含着类似于（键，值）的有序对。一个关联数组中的有序对可以重复（如 C++ 中的 multi map）也可以不重复（如 C++ 中的 map）。

这种数据结构包含以下几种常见的操作：

1. 向关联数组添加配对
2. 从关联数组内删除配对
3. 修改关联数组内的配对
4. 根据已知的键寻找配对

字典问题是设计一种能够具备关联数组特性的数据结构。解决字典问题的常用方法，是利用散列表，但有些情况下，也可以直接使用有地址的数组，或二叉树，和其他结构。

许多程序设计语言内置基本的数据类型，提供对关联数组的支持。而 Content-addressable memory 则是硬件层面上实现对关联数组的支持。

什么是哈希表？关于哈希表的叙述比较多，这里仅仅截取了概念描述，更多的可以到[维基百科上阅读](#)。

散列表（Hash table，也叫哈希表），是根据关键字（Key value）而直接访问在内存存储位置的数据结构。也就是说，它通过把键值通过一个函数的计算，映射到表中一个位置来访问记录，这加快了查找速度。这个映射函数称做散列函数，存放记录的数组称做散列表。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

字典(2)

字典方法

跟前面所讲述的其它数据类型类似，字典也有一些方法。通过这些方法，能够实现对字典类型数据的操作。这回可不是屠龙之技的。这些方法在编程实践中经常会用到。

copy

拷贝，这个汉语是 copy 的音译，标准的汉语翻译是“复制”。我还记得当初在学 DOS 的时候，那个老师说“拷贝”，搞得我晕头转向，他没有说英文的“copy”发音，而是用标准汉语说“kao(三声)bei(四声)”，对于一直学习过英语、标准汉语和我家乡方言的人来说，理解“拷贝”是有点困难的。谁知道在编程界用的是音译呢。

在一般的理解中，copy 就是将原来的东西再搞一份。但是，在 Python 里面（乃至于很多编程语言中），copy 可不是那么简单的。

```
>>> a = 5
>>> b = a
>>> b
5
```

这样做，是不是就得到了两个 5 了呢？表面上看似乎是，但是，不要忘记我在前面反复提到的：**对象有类型，变量无类型**，正是因着这句话，变量其实是一个标签。不妨请出法宝：`id()`，专门查看内存中对象编号

```
>>> id(a)
139774080
>>> id(b)
139774080
```

果然，并没有两个 5，就一个，只不过是贴了两张标签而已。这种现象普遍存在于 Python 的多种数据类型中。其它的就不演示了，就仅看看 dict 类型。

```
>>> ad = {"name":"qiwsir", "lang":"Python"}
>>> bd = ad
>>> bd
{'lang': 'Python', 'name': 'qiwsir'}
>>> id(ad)
3072239652L
```

```
>>> id(bd)
3072239652L
```

是的，验证了。的确是一个对象贴了两个标签。这是用赋值的方式，实现的所谓“假装拷贝”。那么如果用 copy 方法呢？

```
>>> cd = ad.copy()
>>> cd
{'lang': 'Python', 'name': 'qiwsir'}
>>> id(cd)
3072239788L
```

果然不同，这次得到的 cd 是跟原来的 ad 不同的，它在内存中另辟了一个空间。如果我尝试修改 cd，就应该对原来的 ad 不会造成任何影响。

```
>>> cd["name"] = "itdiffer.com"
>>> cd
{'lang': 'Python', 'name': 'itdiffer.com'}
>>> ad
{'lang': 'Python', 'name': 'qiwsir'}
```

真的是那样，跟推理一模一样。所以，要理解了“变量”是对象的标签，对象有类型而变量无类型，就能正确推断出 Python 能够提供的结果。

```
>>> bd
{'lang': 'Python', 'name': 'qiwsir'}
>>> bd["name"] = "laoqi"
>>> ad
{'lang': 'Python', 'name': 'laoqi'}
>>> bd
{'lang': 'Python', 'name': 'laoqi'}
```

这是又修改了 bd 所对应的“对象”，结果发现 ad 的“对象”也变了。

然而，事情没有那么简单，看下面的，要仔细点，否则就迷茫了。

```
>>> x = {"name": "qiwsir", "lang": ["Python", "java", "c"]}
>>> y = x.copy()
>>> y
{'lang': ['Python', 'java', 'c'], 'name': 'qiwsir'}
>>> id(x)
3072241012L
>>> id(y)
3072241284L
```

y 是从 x 拷贝过来的，两个在内存中是不同的对象。

```
>>> y["lang"].remove("c")
```

为了便于理解，尽量使用短句子，避免用很长很长的复合句。在 y 所对应的 dict 对象中，键"lang"的值是一个列表，为['Python', 'java', 'c']，这里用 `remove()` 这个列表方法删除其中的一个元素"c"。删除之后，这个列表变为：['Python', 'java']

```
>>> y
{'lang': ['Python', 'java'], 'name': 'qiwsir'}
```

果然不出所料。那么，那个x所对应的字典中，这个列表变化了吗？应该没有变化。因为按照前面所讲的，它是另外一个对象，两个互不干扰。

```
>>> x
{'lang': ['Python', 'java'], 'name': 'qiwsir'}
```

是不是有点出乎意料呢？我没有作弊哦。你如果不信，就按照操作自己在交互模式中试试，是不是能够得到这个结果呢？这是为什么？

但是，如果要操作另外一个键值对：

```
>>> y["name"] = "laoqi"
>>> y
{'lang': ['python', 'java'], 'name': 'laoqi'}
>>> x
{'lang': ['python', 'java'], 'name': 'qiwsir'}
```

前面所说的原理是有效的，为什么到值是列表的时候就不奏效了呢？

要破解这个迷局还得用 `id()`

```
>>> id(x)
3072241012L
>>> id(y)
3072241284L
```

x,y 对应着两个不同对象，的确如此。但这个对象（字典）是由两个键值对组成的。其中一个键的值是列表。

```
>>> id(x["lang"])
3072243276L
>>> id(y["lang"])
3072243276L
```

发现了这样一个事实，列表是同一个对象。

但是，作为字符串为值得那个键值对，是分属不同对象。

```
>>> id(x["name"])
3072245184L
>>> id(y["name"])
3072245408L
```

这个事实，就说明了为什么修改一个列表，另外一个也跟着修改；而修改一个的字符串，另外一个不跟随的原因了。

但是，似乎还没有解开深层的原因。深层的原因，这跟 Python 存储的数据类型特点有关，Python 只存储基本类型的数据，比如 int,str，对于不是基础类型的，比如刚才字典的值是列表，Python 不会在被复制的那个对象中重新存储，而是用引用的方式，指向原来的值。如果读者没有明白这句话的意思，我就只能说点通俗的了（我本来不想说通俗的，装着自己有学问），Python 在所执行的复制动作中，如果是基本类型的数据，就在内存中重新建个窝，如果不是基本类型的，就不新建窝了，而是用标签引用原来的窝。这也好理解，如果比较简单，随便建立新窝简单；但是，如果对象太复杂了，就别费劲了，还是引用一下原来的省事。（这么讲有点忽悠了）。

所以，在编程语言中，把实现上面那种拷贝的方式称之为“浅拷贝”。顾名思义，没有解决深层次问题。言外之意，还有能够解决深层次问题的方法喽。

的确是，在 Python 中，有一个“深拷贝”(deep copy)。不过，要用下一 `import` 来导入一个模块。这个东西后面会讲述，前面也遇到过了。

```
>>> import copy
>>> z = copy.deepcopy(x)
>>> z
{'lang': ['python', 'java'], 'name': 'qiwsir'}
```

用 `copy.deepcopy()` 深拷贝了一个新的副本，看这个函数的名字就知道是深拷贝(deepcopy)。用上面用过的武器 `id()` 来勘察一番：

```
>>> id(x["lang"])
3072243276L
>>> id(z["lang"])
3072245068L
```

果然是另外一个“窝”，不是引用了。如果按照这个结果，修改其中一个列表中的元素，应该不影响另外一个人了。

```
>>> x
{'lang': ['Python', 'java'], 'name': 'qiwsir'}
>>> x["lang"].remove("java")
>>> x
{'lang': ['Python'], 'name': 'qiwsir'}
>>> z
{'lang': ['Python', 'java'], 'name': 'qiwsir'}
```

果然如此。再试试，才过瘾呀。

```
>>> x["lang"].append("c++")
>>> x
{'lang': ['Python', 'c++'], 'name': 'qiwsir'}
```

这就是所谓浅拷贝和深拷贝。

clear

在交互模式中，用 help 是一个很好的习惯

```
>>> help(dict.clear)

clear(...)
D.clear() -> None. Remove all items from D.
```

这是一个清空字典中所有元素的操作。

```
>>> a = {"name": "qiwsir"}
>>> a.clear()
>>> a
{}
```

这就是 `clear` 的含义，将字典清空，得到的是“空”字典。这个上节说的 `del` 有着很大的区别。`del` 是将字典删除，内存中就没有它了，不是为“空”。

```
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

果然删除了。

另外，如果要清空一个字典，还能够使用 `a = {}` 这种方法，但这种方法本质是将变量 `a` 转向了 `{} 这个对象`，那么原来的呢？原来的成为了断线的风筝。这样的东西在 Python 中称之为垃圾，而且 Python 能够自动的将这样的垃圾回收。编程者就不用关心它了，反正 Python 会处理了。

get, setdefault

`get` 的含义是：

```
get(...)
D.get(k,d) -> D[k] if k in D, else d. d defaults to None.
```

注意这个说明中，“if k in D ”，就返回其值，否则...(等会再说)。

```
>>> d
{'lang': 'python'}
>>> d.get("lang")
'python'
```

`dict.get()` 就是要得到字典中某个键的值，不过，它不是那么“严厉”罢了。因为类似获得字典中键的值得方法，上节已经有过，如 `d['lang']` 就能得到对应的值 "Python"，可是，如果要获取的键不存在，如：

```
>>> print d.get("name")
None

>>> d["name"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'name'
```

这就是 `dict.get()` 和 `dict['key']` 的区别。

前面有一个半句话，如果键不在字典中，会返回 `None`，这是一种情况。还可以这样：

```
>>> d = {"lang":"Python"}
>>> newd = d.get("name",'qiwsir')
>>> newd
'qiwsir'
>>> d
{'lang': 'Python'}
```

以 `d.get("name",'qiwsir')` 的方式，如果不能得到键"name"的值，就返回后面指定的值"qiwsir"。这就是文档中那句话：`D[k]` if k in D , else d . 的含义。这样做，并没有影响原来的字典。

另外一个跟 `get` 在功能上有相似地方的 `D.setdefault(k)`，其含义是：

```
setdefault(...)
D.setdefault(k,d) -> D.get(k,d), also set D[k]=d if k not in D
```

首先，它要执行 `D.get(k,d)`，就跟前面一样了，然后，进一步执行另外一个操作，如果键 k 不在字典中，就在字典中增加这个键值对。当然，如果有就没有必要执行这一步了。

```
>>> d
{'lang': 'Python'}
```

```
>>> d.setdefault("lang")
'Python'
```

在字典中，有"lang"这个键，那么就返回它的值。

```
>>> d.setdefault("name", "qiwsir")
'qiwsir'
>>> d
{'lang': 'Python', 'name': 'qiwsir'}
```

没有"name"这个键，于是返回 `d.setdefault("name", "qiwsir")` 指定的值"qiwsir"，并且将键值对 `'name': "qiwsir"` 添加到原来的字典中。

如果这样操作：

```
>>> d.setdefault("web")
```

什么也没有返回吗？不是，返回了，只不过没有显示出来，如果你用 `print` 就能看到。因为这里返回的是一个 `None`。不妨查看一下那个字典。

```
>>> d
{'lang': 'Python', 'web': None, 'name': 'qiwsir'}
```

是不是键"web"的值成为了 `None`

items/iteritems, keys/iterkeys, values/itervalues

这个标题中列出的是三组 `dict` 的函数，并且这三组有相似的地方。在这里详细讲述第一组，其余两组，我想凭借读者的聪明智慧是不在话下的。

```
>>> help(dict.items)
items(...)
D.items() -> list of D's (key, value) pairs, as 2-tuples
```

这种方法是惯用的伎俩了，只要在交互模式中鼓捣一下，就能得到帮助信息。从中就知道 `D.items()` 能够得到一个关于字典的列表，列表中的元素是由字典中的键和值组成的元组。例如：

```
>>> dd = {"name": "qiwsir", "lang": "python", "web": "www.itdiffer.com"}
>>> dd_kv = dd.items()
>>> dd_kv
[('lang', 'Python'), ('web', 'www.itdiffer.com'), ('name', 'qiwsir')]
```

显然，是有返回值的。这个操作，在后面要讲到的循环中，将有很大的作用。

跟 `items` 类似的就是 `iteritems`，看这个词的特点，是由 `iter` 和 `items` 拼接而成的，后部分 `items` 就不用说了，肯定是在告诉我们，得到的结果跟 `D.items()` 的结果类似。是的，但是，还有一个 `iter` 是什么意思？在《[列表\(2\)](#)》中，我提到了一个词“`iterable`”，它的含义是“可迭代的”，这里的 `iter` 是指的名词 `iterator` 的前部分，意思是“迭代器”。合起来，“`iteritems`”的含义就是：

```
iteritems(...)
D.iteritems() -> an iterator over the (key, value) items of D
```

你看，学习 Python 不是什么难事，只要充分使用帮助文档就好了。这里告诉我们，得到的是一个“迭代器”（关于什么是迭代器，以及相关的内容，后续会详细讲述），这个迭代器是关于“`D.items()`”的。看个例子就明白了。

```
>>> dd
{'lang': 'Python', 'web': 'www.itdiffer.com', 'name': 'qiwsir'}
>>> dd_iter = dd.iteritems()
>>> type(dd_iter)
<type 'dictionary-itemiterator'>
>>> dd_iter
<dictionary-itemiterator object at 0xb72b9a2c>
>>> list(dd_iter)
[('lang', 'Python'), ('web', 'www.itdiffer.com'), ('name', 'qiwsir')]
```

得到的 `dd_iter` 的类型，是一个‘`dictionary-itemiterator`’类型，不过这种迭代器类型的数据不能直接输出，必须用 `list()` 转换一下，才能看到里面的真面目。

另外两组，含义跟这个相似，只不过是得到 `key` 或者 `value`。下面仅列举一下例子，具体内容，读者可以自行在交互模式中看文档。

```
>>> dd
{'lang': 'Python', 'web': 'www.itdiffer.com', 'name': 'qiwsir'}
>>> dd.keys()
['lang', 'web', 'name']
>>> dd.values()
['Python', 'www.itdiffer.com', 'qiwsir']
```

这里先交代一句，如果要实现对键值对或者键或者值的循环，用迭代器的效率会高一些。对这句话的理解，在后面会给大家进行详细分析。

pop, popitem

在《[列表\(3\)](#)》中，有关于删除列表中元素的函数 `pop` 和 `remove`，这两个的区别在于 `list.remove(x)` 用来删除指定的元素，而 `list.pop([i])` 用于删除指定索引的元素，如果不提供索引值，就默认删除最后一个。

在字典中，也有删除键值对的函数。

```
pop(...)

D.pop(k[,d]) -> v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised
```

`D.pop(k[,d])` 是以字典的键为参数，删除指定键的键值对，当然，如果输入对应的值也可以，那个是可选的。

```
>>> dd
{'lang': 'Python', 'web': 'www.itdiffer.com', 'name': 'qiwsir'}
>>> dd.pop("name")
'qiwsir'
```

要删除指定键"name"，返回了其值"qiwsir"。这样，在原字典中，“'name':'qiwsir'”这个键值对就被删除了。

```
>>> dd
{'lang': 'Python', 'web': 'www.itdiffer.com'}
```

值得注意的是，`pop` 函数中的参数是不能省略的，这跟列表中的那个 `pop` 有所不同。

```
>>> dd.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pop expected at least 1 arguments, got 0
```

如果要删除字典中没有的键值对，也会报错。

```
>>> dd.pop("name")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'name'
```

有意思的是 `D.popitem()` 倒是跟 `list.pop()` 有相似之处，不用写参数（`list.pop` 是可以不写参数），但是，`D.popitem()` 不是删除最后一个，前面已经交代过了，`dict` 没有顺序，也就没有最后和最先了，它是随机删除一个，并将所删除的返回。

```
popitem(...)

D.popitem() -> (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.
```

如果字典是空的，就要报错了

```
>>> dd
{'lang': 'Python', 'web': 'www.itdiffer.com'}
>>> dd.popitem()
('lang', 'Python')
```

```
>>> dd
{'web': 'www.itdiffer.com'}
```

成功地删除了一对，注意是随机的，不是删除前面显示的最后一个。并且返回了删除的内容，返回的数据格式是 tuple

```
>>> dd.popitems()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'popitems'
```

错了？！注意看提示信息，没有那个...，哦，果然错了。注意是 popitem，不要多了 s，前面的 D.items() 中包含 s，是复数形式，说明它能够返回多个结果（多个元组组成的列表），而在 D.popitem() 中，一次只能随机删除一对键值对，并以一个元组的形式返回，所以，要单数形式，不能用复数形式了。

```
>>> dd.popitem()
('web', 'www.itdiffer.com')
>>> dd
{}
```

都删了，现在那个字典成空的了。如果再删，会怎么样？

```
>>> dd.popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

报错信息中明确告知，字典已经是空的了，没有再供删的东西了。

update

update() ,看名字就猜测到一二了，是不是更新字典内容呢？的确是。

```
update(...)
D.update([E, ]**F) -> None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does:  for k in E: D[k] = E[k]
If E present and lacks .keys() method, does:  for (k, v) in E: D[k] = v
In either case, this is followed by: for k in F: D[k] = F[k]
```

不过，看样子这个函数有点复杂。不要着急，通过实验可以一点一点鼓捣明白的。

首先，这个函数没有返回值，或者说返回值是 None,它的作用就是更新字典。其参数可以是字典或者某种可迭代的数据类型。

```
>>> d1 = {"lang": "python"}
>>> d2 = {"song": "I dreamed a dream"}
>>> d1.update(d2)
>>> d1
{'lang': 'Python', 'song': 'I dreamed a dream'}
>>> d2
{'song': 'I dreamed a dream'}
```

这样就把字典 d2 更新入了 d1 那个字典，于是 d1 中就多了一些内容，把 d2 的内容包含进来了。d2 当然还存在，并没有受到影响。

还可以用下面的方法更新：

```
>>> d2
{'song': 'I dreamed a dream'}
>>> d2.update([("name", "qiwsir"), ("web", "itdiffer.com")])
>>> d2
{'web': 'itdiffer.com', 'name': 'qiwsir', 'song': 'I dreamed a dream'}
```

列表的元组是键值对。

has_key

这个函数的功能是判断字典中是否存在某个键

```
has_key(...)
D.has_key(k) -> True if D has a key k, else False
```

跟前一节中遇到的 `k in D` 类似。

```
>>> d2
{'web': 'itdiffer.com', 'name': 'qiwsir', 'song': 'I dreamed a dream'}
>>> d2.has_key("web")
True
>>> "web" in d2
True
```

关于 dict 的函数，似乎不少。但是，不用着急，也不用担心记不住，因为根本不需要记忆。只要会用搜索即可。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

集合(1)

回顾一下已经学过的数据类型:int/str/bool/list/dict/tuple

还真的不少了.

不过,Python 是一个发展的语言,没准以后还出别的呢.看官可能有疑问了,出了这么多的数据类型,我也记不住呀,特别是里面还有不少方法.

不要担心记不住,你只要记住爱因斯坦说的就好了.

爱因斯坦在美国演讲,有人问:“你可记得声音的速度是多少?你如何记下许多东西?”

爱因斯坦轻松答道:“声音的速度是多少,我必须查辞典才能回答.因为我从来不记在辞典上已经印着的东西,我的记忆力是用来记忆书本上没有的东西.”

多么霸气的回答,这回答不仅仅霸气,更告诉我们一种方法:只要能够通过某种方法查找到的,就不需要记忆.

那么,上面那么多数据类型及其各种方法,都不需要记忆了,因为它们都可以通过下述方法但不限于这些方法查到(这句话的逻辑还是比较严密的,包括但不限于...)

- 交互模式下用 dir()或者 help()
- google(不推荐 Xdu,原因自己体会啦)

在已经学过的数据类型中:

- 能够索引的,如 list/str,其中的元素可以重复
- 可变的,如 list/dict,即其中的元素/键值对可以原地修改
- 不可变的,如 str/int,即不能进行原地修改
- 无索引序列的,如 dict,即其中的元素(键值对)没有排列顺序

现在要介绍另外一种类型的数据,英文是 set,翻译过来叫做“集合”.它的特点是:有的可变,有的不可变;元素无次序,不可重复.

创建 set

tuple 算是 list 和 str 的杂合(杂交的都有自己的优势,上一节的末后已经显示了),那么 set 则可以堪称是 list 和 dict 的杂合.

set 拥有类似 dict 的特点:可以用{}花括号来定义; 其中的元素没有序列,也就是是非序列类型的数据;而且, set 中的元素不可重复,这就类似 dict 的键.

set 也有一点 list 的特点:有一种集合可以原处修改.

下面通过实验,进一步理解创建 set 的方法:

```
>>> s1 = set("qiwsir")
>>> s1
set(['q', 'i', 's', 'r', 'w'])
```

把 str 中的字符拆解开,形成 set.特别注意观察:qiwsir 中有两个 i,但是在 s1 中,只有一个 i,也就是集合中元素不能重复。

```
>>> s2 = set([123,"google","face","book","facebook","book"])
>>> s2
set(['facebook', 123, 'google', 'book', 'face'])
```

在创建集合的时候,如果发现了重复的元素,就会过滤一下,剩下不重复的。而且,从 s2 的创建可以看出,查看结果是显示的元素顺序排列与开始建立是不同,完全是随意显示的,这说明集合中的元素没有序列。

```
>>> s3 = {"facebook",123}      #通过{}直接创建
>>> s3
set([123, 'facebook'])
```

除了用 `set()` 来创建集合。还可以使用 {} 的方式,但是这种方式不提倡使用,因为在某些情况下,Python 搞不清楚是字典还是集合。看看下面的探讨就发现问题了。

```
>>> s3 = {"facebook",[1,2,'a'],"name":"Python","lang":"english"},123}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'

>>> s3 = {"facebook",[1,2],123}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

从上述实验中,可以看出,通过{}无法创建含有 list/dict 元素的 set.

认真阅读报错信息,有这样的词汇:“unhashable”,在理解这个词之前,先看它的反义词“hashable”,很多时候翻译为“可哈希”,其实它有一个不是音译的名词“散列”,这个在[《字典\(1\)》](#)中有说明。网上搜一下,有不少文章对这个进行诠释。如果我们简单点理解,某数据“不可哈希”(unhashable)就是其可变,如 list/

dict，都能原地修改，就是 unhashable。否则，不可变的，类似 str 那样不能原地修改，就是 hashable（可哈希）的。

对于前面已经提到的字典，其键必须是 hashable 数据，即不可变的。

现在遇到的集合，其元素也要是“可哈希”的。上面例子中，试图将字典、列表作为元素的元素，就报错了。而且报错信息中明确告知 list/dict 是不可哈希类型，言外之意，里面的元素都应该是可哈希类型。

继续探索一个情况：

```
>>> s1
set(['q', 'i', 's', 'r', 'w'])
>>> s1[1] = "l"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support item assignment
```

这里报错，进一步说明集合没有序列，不能用索引方式对其进行修改。

```
>>> s1
set(['q', 'i', 's', 'r', 'w'])
>>> lst = list(s1)
>>> lst
['q', 'i', 's', 'r', 'w']
>>> lst[1] = "l"
>>> lst
['q', 'l', 's', 'r', 'w']
```

分别用 `list()` 和 `set()` 能够实现两种数据类型之间的转化。

特别说明，利用 `set()` 建立起来的集合是可变集合，可变集合都是 unhashable 类型的。

set 的方法

还是用前面已经介绍过多次的自学方法，把 `set` 的有关内置函数找出来，看看都可以对 `set` 做什么操作。

```
>>> dir(set)
['__and__', '__class__', '__cmp__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__hash__', '__iter__', '__len__', '__new__', '__ne__', '__reduce__', '__reduce_ex__', '__repr__', '__ror__', '__str__', '__subclasshook__']
```

为了看的清楚，我把双划线`__`开始的先删除掉(后面我们会有专题讲述这些)：

```
'add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection', 'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference', 'symmetric_difference_update', 'union', 'update'
```

然后用 help() 可以找到每个函数的具体使用方法,下面列几个例子:

add, update

```
>>> help(set.add)

Help on method_descriptor:

add(...)
Add an element to a set.

This has no effect if the element is already present.
```

下面在交互模式这个最好的实验室里面做实验:

```
>>> a_set = {}      #我想当然地认为这样也可以建立一个 set
>>> a_set.add("qiwsir")  #报错.看看错误信息,居然告诉我 dict 没有 add.我分明建立的是 set 呀.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'add'
>>> type(a_set)      #type 之后发现,计算机认为我建立的是一个 dict
<type 'dict'>
```

特别说明一下,{}这个东西,在 dict 和 set 中都用.但是,如上面的方法建立的是 dict,不是 set.这是 Python 规定的.要建立 set,只能用前面介绍的方法了.

```
>>> a_set = {'a', 'i'}    #这回就是 set 了吧
>>> type(a_set)
<type 'set'>      #果然

>>> a_set.add("qiwsir")  #增加一个元素
>>> a_set          #原处修改,即原来的 a_set 引用对象已经改变
set(['i', 'a', 'qiwsir'])

>>> b_set = set("python")
>>> type(b_set)
<type 'set'>
>>> b_set
set(['h', 'o', 'n', 'p', 't', 'y'])
>>> b_set.add("qiwsir")
```

```
>>> b_set
set(['h', 'o', 'n', 'p', 't', 'qiwsir', 'y'])

>>> b_set.add([1,2,3])    #报错.list 是不可哈希的，集合中的元素应该是 hashable 类型。
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'

>>> b_set.add('[1,2,3]')  #可以这样!
>>> b_set
set(['[1,2,3]', 'h', 'o', 'n', 'p', 't', 'qiwsir', 'y'])
```

除了上面的增加元素方法之外,还能够从另外一个 set 中合并过来元素,方法是 set.update(s2)

```
>>> help(set.update)
update(...)
    Update a set with the union of itself and others.

>>> s1
set(['a', 'b'])
>>> s2
set(['github', 'qiwsir'])
>>> s1.update(s2)    #把 s2 的元素并入到 s1 中.
>>> s1              #s1 的引用对象修改
set(['a', 'qiwsir', 'b', 'github'])
>>> s2              #s2 的未变
set(['github', 'qiwsir'])
```

`pop, remove, discard, clear`

```
>>> help(set.pop)
pop(...)
    Remove and return an arbitrary set element.
    Raises KeyError if the set is empty.

>>> b_set
set(['[1,2,3]', 'h', 'o', 'n', 'p', 't', 'qiwsir', 'y'])
>>> b_set.pop()    #从 set 中任意选一个删除,并返回该值
'[1,2,3]'
>>> b_set.pop()
'h'
>>> b_set.pop()
'o'
>>> b_set
set(['n', 'p', 't', 'qiwsir', 'y'])
```

```
>>> b_set.pop("n") #如果要指定删除某个元素,报错了.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pop() takes no arguments (1 given)
```

set.pop()是从 set 中任意选一个元素,删除并将这个值返回.但是,不能指定删除某个元素.报错信息中就告诉我们了,`pop()`不能有参数.此外,如果 set 是空的了,也报错.这条是帮助信息告诉我们的,看官可以试试.

要删除指定的元素,怎么办?

```
>>> help(set.remove)

remove(...)

Remove an element from a set; it must be a member.

If the element is not a member, raise a KeyError.
```

set.remove(obj) 中的 obj,必须是 set 中的元素,否则就报错.试一试:

```
>>> a_set
set(['i', 'a', 'qiwsir'])
>>> a_set.remove("i")
>>> a_set
set(['a', 'qiwsir'])
>>> a_set.remove("w")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'w'
```

跟 remove(obj)类似的还有一个 discard(obj):

```
>>> help(set.discard)

discard(...)

Remove an element from a set if it is a member.

If the element is not a member, do nothing.
```

与 `help(set.remove)` 的信息对比,看看有什么不同.discard(obj)中的 obj 如果是 set 中的元素,就删除,如果不是,就什么也不做,do nothing.新闻就要对比着看才有意思呢.这里也一样.

```
>>> a_set.discard('a')
>>> a_set
set(['qiwsir'])
```

```
>>> a_set.discard('b')
>>>
```

在删除上还有一个绝杀,就是 set.clear(),它的功能是:Remove all elements from this set.(看官自己在交互模式下 help(set.clear))

```
>>> a_set
set(['qiwsir'])
>>> a_set.clear()
>>> a_set
set([])
>>> bool(a_set)  #空了,bool一下返回 False.
False
```

知识

集合,也是一个数学概念(以下定义来自[维基百科](#))

集合 (或简称集) 是基本的数学概念, 它是集合论的研究对象。最简单的说法, 即是在最原始的集合论—朴素集合论—中的定义, 集合就是“一堆东西”。集合里的“东西”, 叫作元素。若然 x 是集合 A 的元素, 记作 $x \in A$ 。

集合是现代数学中一个重要的基本概念。集合论的基本理论直到十九世纪末才被创立, 现在已经是数学教育中一个普遍存在的部分, 在小学时就开始学习了。这里对被数学家们称为“直观的”或“朴素的”集合论进行一个简短而基本的介绍; 更详细的分析可见朴素集合论。对集合进行严格的公理推导可见公理化集合论。

在计算机中,集合是什么呢?同样来自[维基百科](#),这么说的:

在计算机科学中, 集合是一组可变数量的数据项 (也可能是 0 个) 的组合, 这些数据项可能共享某些特征, 需要以某种操作方式一起进行操作。一般来讲, 这些数据项的类型是相同的, 或基类相同 (若使用的语言支持继承)。列表 (或数组) 通常不被认为是集合, 因为其大小固定, 但事实上它常常在实现中作为某些形式的集合使用。

集合的种类包括列表, 集, 多重集, 树和图。枚举类型可以是列表或集。

不管是否明白,貌似很厉害呀.

是的,所以本讲仅仅是对集合有一个入门.关于集合的更多操作如运算/比较等,还没有涉及呢.

[总目录 \(页 0\)](#)

如果你认为有必要打赏我, 请通过支付宝: qiwsir@126.com,不胜感激。

集合(2)

不变的集合

《[集合\(1\)](#)》中以 `set()` 来建立集合，这种方式所创立的集合都是可原处修改的集合，或者说是可变的，也可以说是 unhashable

还有一种集合，不能在原处修改。这种集合的创建方法是用 `frozenset()`，顾名思义，这是一个被冻结的集合，当然是不能修改了，那么这种集合就是 hashable 类型——可哈希。

```
>>> f_set = frozenset("qiwsir")
>>> f_set
frozenset(['q', 'i', 's', 'r', 'w'])
>>> f_set.add("python")      #报错，不能修改，则无此方法
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'

>>> a_set = set("github")    #对比看一看，这是一个可以原处修改的 set
>>> a_set
set(['b', 'g', 'i', 'h', 'u', 't'])
>>> a_set.add("python")
>>> a_set
set(['b', 'g', 'i', 'h', 'python', 'u', 't'])
```

集合运算

唤醒一下中学数学（准确说是高中数学中的一点知识）中关于集合的一点知识，当然，你如果是某个理工科的专业大学毕业，更应该熟悉集合之间的关系。

元素与集合的关系

就一种关系，要么术语某个集合，要么不属于。

```
>>> aset
set(['h', 'o', 'n', 'p', 't', 'y'])
>>> "a" in aset
False
```

```
>>> "h" in a
True
```

集合与集合的关系

假设两个集合 A、B

- A 是否等于 B，即两个集合的元素完全一样

在交互模式下实验

```
>>> a
set(['q', 'i', 's', 'r', 'w'])
>>> b
set(['a', 'q', 'i', 'l', 'o'])
>>> a == b
False
>>> a != b
True
```

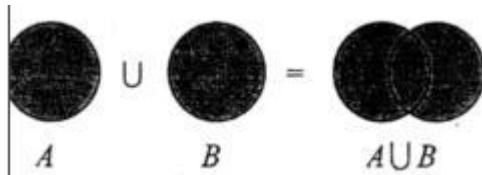
- A 是否是 B 的子集，或者反过来，B 是否是 A 的超集。即 A 的元素也都是 B 的元素，但是 B 的元素比 A 的元素数量多。

判断集合 A 是否是集合 B 的子集，可以使用 `A < B`，返回 `true` 则是子集，否则不是。另外，还可以使用函数 `A.issubset(B)` 判断。

```
>>> a
set(['q', 'i', 's', 'r', 'w'])
>>> c
set(['q', 'i'])
>>> c < a  #c 是 a 的子集
True
>>> c.issubset(a)  #或者用这种方法，判断 c 是否是 a 的子集
True
>>> a.issuperset(c) #判断 a 是否是 c 的超集
True

>>> b
set(['a', 'q', 'i', 'l', 'o'])
>>> a < b  #a 不是 b 的子集
False
>>> a.issubset(b)  #或者这样做
False
```

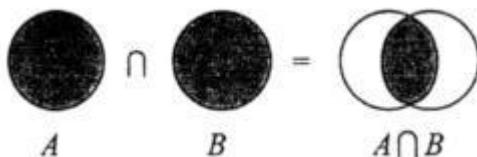
- A、B 的并集，即 A、B 所有元素，如下图所示



可以使用的符号是“|”，是一个半角状态写的竖线，输入方法是在英文状态下，按下“shift”加上右方括号右边的那个键。找找吧。表达式是 `A | B` .也可使用函数 `A.union(B)`，得到的结果就是两个集合并集，注意，这个结果是新生成的一个对象，不是将结合 A 扩充。

```
>>> a
set(['q', 'i', 's', 'r', 'w'])
>>> b
set(['a', 'q', 'i', 'l', 'o'])
>>> a | b          #可以有两种方式，结果一样
set(['a', 'i', 'l', 'o', 'q', 's', 'r', 'w'])
>>> a.union(b)
set(['a', 'i', 'l', 'o', 'q', 's', 'r', 'w'])
```

- A、B 的交集，即 A、B 所公有的元素，如下图所示

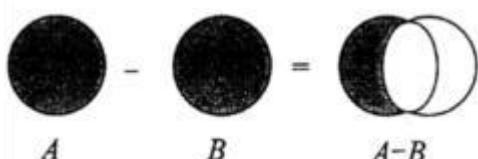


```
>>> a
set(['q', 'i', 's', 'r', 'w'])
>>> b
set(['a', 'q', 'i', 'l', 'o'])
>>> a & b      #两种方式，等价
set(['q', 'i'])
>>> a.intersection(b)
set(['q', 'i'])
```

我在实验的时候，顺手敲了下面的代码，出现的结果如下，看官能解释一下吗？（思考题）

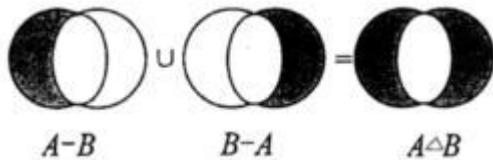
```
>>> a and b
set(['a', 'q', 'i', 'l', 'o'])
```

- A 相对 B 的差（补），即 A 相对 B 不同的部分元素，如下图所示



```
>>> a
set(['q', 'i', 's', 'r', 'w'])
>>> b
set(['a', 'q', 'i', 'l', 'o'])
>>> a - b
set(['s', 'r', 'w'])
>>> a.difference(b)
set(['s', 'r', 'w'])
```

-A、B 的对称差集，如下图所示



```
>>> a
set(['q', 'i', 's', 'r', 'w'])
>>> b
set(['a', 'q', 'i', 'l', 'o'])
>>> a.symmetric_difference(b)
set(['a', 'l', 'o', 's', 'r', 'w'])
```

以上是集合的基本运算。在编程中，如果用到，可以用前面说的方法查找。不用死记硬背。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。



IT
程序设计



3

语句和文件



运算符

在编程语言，运算符是比较多样化的，虽然在《常用数学函数和运算优先级》中给出了一个各种运算符及其优先级的表格，但是，那时对 Python 理解还比较肤浅。建议诸位先回头看看那个表格，然后继续下面的内容。

这里将各种运算符总结一下，有复习，也有拓展。

算术运算符

前面已经讲过了四则运算，其中涉及到一些运算符：加减乘除，对应的符号分别是：`+``-``*``/`，此外，还有求余数的：`%`。这些都是算术运算符。其实，算术运算符不止这些。根据中学数学的知识，看官也应该想到，还应该有乘方、开方之类的。

下面列出一个表格，将所有的运算符表现出来。不用记，但是要认真地看一看，知道有那些，如果以后用到，但是不自信能够记住，可以来查。

运算符	描述	实例
<code>+</code>	加 – 两个对象相加	<code>10+20</code> 输出结果 30
<code>-</code>	减 – 得到负数或是一个数减去另一个数	<code>10-20</code> 输出结果 -10
<code>*</code>	乘 – 两个数相乘或是返回一个被重复若干次的字符串	<code>10 * 20</code> 输出结果 200
<code>/</code>	除 – x除以y	<code>20/10</code> 输出结果 2
<code>%</code>	取余 – 返回除法的余数	<code>20%10</code> 输出结果 0
<code>**</code>	幂 – 返回x的y次幂	<code>10**2</code> 输出结果 100
<code>//</code>	取整除 – 返回商的整数部分	<code>9//2</code> 输出结果 4, <code>9.0//2.0</code> 输出结果 4.0

列为看官可以根据中学数学的知识，想想上面的运算符在混合运算中，应该按照什么顺序计算。并且亲自试试，是否与中学数学中的规律一致。（应该是一致的，计算机科学家不会另外搞一套让我们和他们一块受罪。）

比较运算符

所谓比较，就是比一比两个东西。这在某国是最常见的了，做家长的经常把自己的孩子跟别的孩子比较，唯恐自己孩子在某方面差了；官员经常把自己的收入和银行比较，总觉得少了。

在计算机高级语言编程中，任何两个同一类型的量的都可以比较，比如两个数字可以比较，两个字符串可以比较。注意，是两个同一类型的。不同类型的量可以比较吗？首先这种比较没有意义。就好比二两肉和三尺布进行比较，它们谁大呢？这种比较无意义。所以，在真正的编程中，我们要谨慎对待这种不同类型量的比较。

但是，在某些语言中，允许这种无意思的比较。因为它在比较的时候，都是将非数值的转化为了数值类型比较。

对于比较运算符，在小学数学中就学习了一些：大于、小于、等于、不等于。没有陌生的东西，Python 里面也是如此。且看下表：

以下假设变量 a 为 10，变量 b 为 20：

运算符	描述	实例
==	等于 – 比较对象是否相等	(a == b) 返回 False。
!=	不等于 – 比较两个对象是否不相等	(a != b) 返回 true.
>	大于 – 返回x是否大于y	(a > b) 返回 False。
<	小于 – 返回x是否小于y	(a < b) 返回 true。
>=	大于等于 – 返回x是否大于等于y。	(a >= b) 返回 False。
<=	小于等于 – 返回x是否小于等于y。	(a <= b) 返回 true。

上面的表格实例中，显示比较的结果就是返回一个 true 或者 false，这是什么意思呢。就是在告诉你，这个比较如果成立，就是为真，返回 True，否则返回 False，说明比较不成立。

请按照下面方式进行比较操作，然后再根据自己的想象，把比较操作熟练熟练。

```
>>> a=10
>>> b=20
>>> a>b
False
>>> a<b
True
>>> a==b
False
>>> a!=b
True
>>> a>=b
False
>>> a<=b
True
```

除了数字之外，还可以对字符串进行比较。字符串中的比较是按照“字典顺序”进行比较的。当然，这里说的是英文的字典，不是前面说的字典数据类型。

```
>>> a = "qiwsir"
>>> b = "Python"
>>> a > b
True
```

先看第一个字符，按照字典顺序，q 大于 p（在字典中，q 排在 p 的后面），那么就返回结果 True.

在 Python 中，如果是两种不同类型的对象，虽然可以比较。但我是不赞成这样进行比较的。

```
>>> a = 5
>>> b = "5"
>>> a > b
False
```

逻辑运算符

首先谈谈什么是逻辑，韩寒先生对逻辑有一个分类：

逻辑分两种，一种是逻辑，另一种是中国人的逻辑。——韩寒

这种分类的确非常精准。在很多情况下，中国人是有很奇葩的逻辑的。但是，在 Python 中，讲的是逻辑，不是中国人的逻辑。

逻辑（logic），又称理则、论理、推理、推论，是有效推论的哲学研究。逻辑被使用在大部份的智能活动中，但主要在哲学、数学、语义学和计算机科学等领域内被视为一门学科。在数学里，逻辑是指研究某个形式语言的有效推论。

关于逻辑问题，看官如有兴趣，可以听一听 [《国立台湾大学公开课：逻辑》](#)

布尔类型的变量

在所有的高级语言中，都有这么一类变量，被称之为布尔型。从这个名称，看官就知道了，这是用一个人的名字来命名的。

乔治·布尔（George Boole，1815年11月–1864年），英格兰数学家、哲学家。

乔治·布尔是一个皮匠的儿子，生于英格兰的林肯。由于家境贫寒，布尔不得不在协助养家的同时为自己能受教育而奋斗，不管怎么说，他成了19世纪最重要的数学家之一。尽管他考虑过以牧师为业，但最终还是决定从教，而且不久就开办了自己的学校。

在备课的时候，布尔不满意当时的数学课本，便决定阅读伟大数学家的论文。在阅读伟大的法国数学家拉格朗日的论文时，布尔有了变分法方面的新发现。变分法是数学分析的分支，它处理的是寻求优化某些参数的曲线和曲面。

1848年，布尔出版了《The Mathematical Analysis of Logic》，这是他对符号逻辑诸多贡献中的第一次。

1849年，他被任命位于爱尔兰科克的皇后学院（今科克大学或UCC）的数学教授。1854年，他出版了《The Laws of Thought》，这是他最著名的著作。在这本书中布尔介绍了现在以他的名字命名的布尔代数。布尔撰写了微分方程和差分方程的课本，这些课本在英国一直使用到19世纪末。

由于其在符号逻辑运算中的特殊贡献，很多计算机语言中将逻辑运算称为布尔运算，将其结果称为布尔值。

请看官认真阅读布尔的生平，励志呀。

布尔所创立的这套逻辑被称之为“布尔代数”。其中规定只有两种值，True 和 False，正好对应这计算机上二进制数的1和0。所以，布尔代数和计算机是天然吻合的。

所谓布尔类型，就是返回结果为1(True)、0(False)的数据变量。

在Python中（其它高级语言也类似，其实就是布尔代数的运算法则），有三种运算符，可以实现布尔类型的变量间的运算。

布尔运算

看下面的表格，对这种逻辑运算符比较容易理解：

（假设变量a为10，变量b为20）

运算符	描述	实例
and	布尔"与" – 如果x为False，x and y返回False，否则它返回y的计算值。	(a and b) 返回true。
or	布尔"或" – 如果x是True，它返回True，否则它返回y的计算值。	(a or b) 返回true。
not	布尔"非" – 如果x为True，返回False。如果x为False，它返回True。	not(a and b) 返回false。

- and

and，翻译为“与”运算，但事实上，这种翻译容易引起望文生义的理解。先说一下正确的理解。A and B，含义是：首先运算A，如果A的值是true，就计算B，并将B的结果返回做为最终结果，如果B是False，那么A and B的最终结果就是False，如果B的结果是True，那么A and B的结果就是True；如果A的值是False，就不计算B了，直接返回A and B的结果为False。

比如：

4>3 and 4<9，首先看4>3的值，这个值是True，再看4<9的值，是True，那么最终这个表达式的结果为True。

```
>>> 4>3 and 4<9
True
```

`4>3 and 4<2` , 先看 `4>3` , 返回 `True` , 再看 `4<2` , 返回的是 `False` , 那么最终结果是 `False` .

```
>>> 4>3 and 4<2
False
```

`4<3 and 4<9` , 先看 `4<3` , 返回为 `False` ,就不看后面的了, 直接返回这个结果做为最终结果 (对这种现象, 有一个形象的说法, 叫做“短路”。这个说法形象吗? 不熟悉物理的是不是感觉形象?)。

```
>>> 4<3 and 4<2
False
```

前面说容易引起望文生义的理解, 就是有相当不少的人认为无论什么时候都看 `and` 两边的值, 都是 `true` 返回 `true`, 有一个是 `false` 就返回 `false`。根据这种理解得到的结果, 与前述理解得到的结果一样, 但是, 运算量不一样哦。

- `or`

`or`, 翻译为“或”运算。在 `A or B` 中, 它是这么运算的:

```
if A==True:
    return True
else:
    if B==True:
        return True
    else if B==False:
        return False
```

上面这段算是伪代码啦。所谓伪代码, 就是不是真正的代码, 无法运行。但是, 伪代码也有用途, 就是能够以类似代码的方式表达一种计算过程。

看官是不是能够看懂上面的伪代码呢? 下面再增加上每行的注释。这个伪代码跟自然的英语差不多呀。

```
if A==True:      #如果 A 的值是 True
    return True  #返回 True, 表达式最终结果是 True
else:           #否则, 也就是 A 的值不是 True
    if B==True:  #看 B 的值, 然后就返回B的值做为最终结果。
        return True
    else if B==False:
        return False
```

举例, 根据上面的运算过程, 分析一下下面的例子, 是不是与运算结果一致?

```
>>> 4<3 or 4<9
```

```
True
```

```
>>> 4<3 or 4>9
```

```
False
```

```
>>> 4>3 or 4>9
```

```
True
```

- not

not，翻译成“非”，窃以为非常好，不论面对什么，就是要否定它。

```
>>> not(4>3)
```

```
False
```

```
>>> not(4<3)
```

```
True
```

关于运算符问题，其实不止上面这些，还有呢，比如成员运算符 in，在后面的学习中会逐渐遇到。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

语句(1)

数据类型已经学的差不多了，但是，到现在为止我们还不能真正的写程序，这就好比小学生学习写作一样，到目前为止仅仅学会了一些词语，还不知道如何造句子。从现在开始就学习如何造句子了。

在编程语言中，句子被称之为“语句”，

什么是语句

事实上，前面已经用过语句了，最典型的那句：`print "Hello, World"` 就是语句。

为了能够严谨地阐述这个概念，抄一段[维基百科中的词条：命令式编程](#)

命令式编程（英语：Imperative programming），是一种描述电脑所需作出的行为的编程范型。几乎所有电脑的硬件工作都是指令式的；几乎所有电脑的硬件都是设计来运行机器码，使用指令式的风格来写的。较高级的指令式编程语言使用变量和更复杂的语句，但仍依从相同的范型。

运算语句一般来说都表现了在存储器内的数据进行运算的行为，然后将结果存入存储器中以便日后使用。高级命令式编程语言更能处理复杂的表达式，可能会产生四则运算和函数计算的结合。

一般所有高级语言，都包含如下语句，Python 也不例外：

- 循环语句：容许一些语句反复运行数次。循环可依据一个默认的数目来决定运行这些语句的次数；或反复运行它们，直至某些条件改变。
- 条件语句：容许仅当某些条件成立时才运行某个区块。否则，这个区块中的语句会略去，然后按区块后的语句继续运行。
- 无条件分支语句：容许运行顺序转移到程序的其他部分之中。包括跳跃（在很多语言中称为 Goto）、副程序和 Procedure 等。

循环、条件分支和无条件分支都是控制流程。

当然，Python 中的语句还是有 Python 特别之处的（别的语言中，也会有自己的特色）。下面就开始娓娓道来。

print

在 Python2.x 中，print 是一个语句，但是在 Python3.x 中它是一个函数了。这点请注意。不过，这里所使用的还是 Python2.x。

为什么不用 Python3.x？这个问题在开始就回答过。但是还有朋友问。重复回答：因为现在很多工程项目都是 Python2.x，Python3.x 相对 Python2.x 有不完全兼容的地方。学 Python 的目的就是要在真实的工程项目中使用，理所应当要学 Python2.x。此外，学会了 Python2.x，将来过渡到 Python3.x，只需要注意一些细节即可。

print 发起的语句，在程序中主要是将某些东西打印出来，还记得在讲解字符串的时候，专门讲述了字符串的格式化输出吗？那就是用来 print 的。

```
>>> print "hello, world"
hello, world
>>> print "hello", "world"
hello world
```

请仔细观察，上面两个 print 语句的差别。第一个打印的是"hello, world"，包括其中的逗号和空格，是一个完整的字符串。第二个打印的是两个字符串，一个是"hello"，另外一个是"world"，两个字符串之间用逗号分隔。

本来，在 print 语句中，字符串后面会接一个 `\n` 符号。即换行。但是，如果要在字符串后面跟着逗号，那么换行就取消了，意味着两个字符串"hello"，"world"打印在同一行。

或许现在体现的还不时很明显，如果换一个方法，就显示出来了。（下面用到了一个被称之为循环的语句，下一节会重点介绍。）

```
>>> for i in [1,2,3,4,5]:
...     print i
...
1
2
3
4
5
```

这个循环的意思就是要从列表中依次取出每个元素，然后赋值给变量 i，并用 print 语句打印打出来。在变量 i 后面没有任何符号，每打印一个，就换行，再打印另外一个。

下面的方式就跟上面的有点区别了。

```
>>> for i in [1,2,3,4,5]:
...     print i,
...
1 2 3 4 5
```

就是在 `print` 语句的最后加了一个逗号，打印出来的就在一行了。

`print` 语句经常用在调试程序的过程，让我们能够知道程序在执行过程中产生的结果。

import

在《常用数学函数和运算优先级》中，曾经用到过一个 `math` 模块，它能提供很多数学函数，但是这些函数不是 Python 的内建函数，是 `math` 模块的，所以，要用 `import` 引用这个模块。

这种用 `import` 引入模块的方法，是 Python 编程经常用到的。引用方法有如下几种：

```
>>> import math
>>> math.pow(3,2)
9.0
```

这是常用的一种方式，而且非常明确，`math.pow(3,2)` 就明确显示了，`pow()` 函数是 `math` 模块里的。可以说这是一种可读性非常好的引用方式，并且不同模块的同名函数不会产生冲突。

```
>>> from math import pow
>>> pow(3,2)
9.0
```

这种方法就有点偷懒了，不过也不难理解，从字面意思就知道 `pow()` 函数来自于 `math` 模块。在后续使用的时候，只需要直接使用 `pow()` 即可，不需要在前面写上模块名称了。这种引用方法，比较适合于引入模块较少的时候。如果引入模块多了，可读性就下降了，会不知道那个函数来自那个模块。

```
>>> from math import pow as pingfang
>>> pingfang(3,2)
9.0
```

这是在前面那种方式基础上的发展，将从某个模块引入的函数重命名，比如讲 `pow` 充命名为 `pingfang`，然后使用 `pingfang()` 就相当于在使用 `pow()` 了。

如果要引入多个函数，可以这样做：

```
>>> from math import pow, e, pi
>>> pow(e,pi)
23.140692632779263
```

引入了 math 模块里面的 pow,e,pi, pow() 是一个乘方函数, e, 就是那个欧拉数; pi 就是 π .

e, 作为数学常数, 是自然对数函数的底数。有时称他为欧拉函数 (Euler's number), 以瑞士数学家欧拉命名; 也有个较鲜见的名字纳皮尔常数, 以纪念苏格兰数学家约翰·纳皮尔引进对数。它是一个无限不循环小数。e = 2.71828182845904523536(《维基百科》)

e 的 π 次方, 是一个数学常数。与 e 和 π 一样, 它是一个超越数。这个常数在希尔伯特第七问题中曾提到过。(《维基百科》)

```
>>> from math import *
>>> pow(3,2)
9.0
>>> sqrt(9)
3.0
```

这种引入方式是最贪图省事的了, 一下将 math 中的所有函数都引过来了。不过, 这种方式的结果是让可读性更降低了。仅适用于模块中的函数比较少的时候, 并且在程序中应用比较频繁。

在这里, 我们用 math 模块为例, 引入其中的函数。事实上, 不仅函数可以引入, 模块中还可以包括常数等, 都可以引入。在编程中, 模块中可以包括各样的对象, 都可以引入。

赋值语句

对于赋值语句, 应该不陌生, 在前面已经频繁使用了, 如 `a = 3` 这样的, 就是将一个整数赋给了变量。

编程中的“=”和数学中的“=”是完全不同的。在编程语言中, “=”表示赋值过程。

除了那种最简单的赋值之外, 还可以这么干:

```
>>> x, y, z = 1, "python", ["hello", "world"]
>>> x
1
>>> y
'python'
>>> z
['hello', 'world']
```

这里就一一对应赋值了。如果把几个值赋给一个, 可以吗?

```
>>> a = "itdiffer.com", "python"
>>> a
('itdiffer.com', 'python')
```

原来是将右边的两个值装入了一个元组，然后将元组赋给了变量 a。这个 Python 太聪明了。

在 Python 的赋值语句中，还有一个更聪明的，它一出场，简直是让一些已经学习过某种其它语言的人亮瞎眼。

有两个变量，其中 `a = 2, b = 9`。现在想让这两个变量的值对调，即最终是 `a = 9, b = 2`。

这是一个简单而经典的题目。在很多编程语言中，是这么处理的：

```
temp = a;
a = b;
b = temp;
```

这么做的那些编程语言，变量就如同一个盒子，值就如同放到盒子里面的东西。如果要实现对调，必须在找一个盒子，将 a 盒子里面的东西（数字 2）拿到那个临时盒子(temp)中，这样 a 盒子就空了，然后将 b 盒子中的东西拿(数字 9)拿到 a 盒子中(a = b)，完成这步之后，b 盒子是空的了，最后将临时盒子里面的那个数字 2 拿到 b 盒子中。这就实现了两个变量值得对调。

太啰嗦了。

Python 只要一行就完成了。

```
>>> a = 2
>>> b = 9

>>> a, b = b, a

>>> a
9
>>> b
2
```

`a, b = b, a` 就实现了数值对调，多么神奇。之所以神奇，就是因为我前面已经数次提到的 Python 中变量和数据对象的关系。变量相当于贴在对象上的标签。这个操作只不过是将标签换个位置，就分别指向了不同的数据对象。

还有一种赋值方式，被称为“链式赋值”

```
>>> m = n = "I use python"
>>> print m,n
I use python I use python
```

用这种方式，实现了一次性对两个变量赋值，并且值相同。

```
>>> id(m)
3072659528L
```

```
>>> id(n)
3072659528L
```

用 `id()` 来检查一下，发现两个变量所指向的是同一个对象。

另外，还有一种判断方法，来检查两个变量所指向的值是否是同一个（注意，同一个和相等是有差别的。在编程中，同一个就是 `id()` 的结果一样。）

```
>>> m is n
True
```

这是在检查 `m` 和 `n` 分别指向的对象是否是同一个，`True` 说明是同一个。

```
>>> a = "I use python"
>>> b = a
>>> a is b
True
```

这是跟上面链式赋值等效的。

但是：

```
>>> b = "I use python"
>>> a is b
False
>>> id(a)
3072659608L
>>> id(b)
3072659568L

>>> a == b
True
```

看出其中的端倪了吗？这次 `a`、`b` 两个变量虽然相等，但不是指向同一个对象。

还有一种赋值形式，如果从数学的角度看，是不可思议的，如：`x = x + 1`，在数学中，这个等式是不成立的。因为数学中的“=”是等于的含义，但是在编程语言中，它成立，因为“=”是赋值的含义，即将变量 `x` 增加 1 之后，再把得到的结果赋值变量 `x`。

这种变量自己变化之后将结果再赋值给自己的形式，称之为“增量赋值”。`+`、`-`、`*`、`/`、`%` 都可以实现这种操作。

为了让这个操作写起来省点事（要写两遍同样一个变量），可以写成：`x += 1`

```
>>> x = 9
>>> x += 1
```

```
>>> x  
10
```

除了数字，字符串进行增量赋值，在实际中也很有价值。

```
>>> m = "py"  
>>> m += "th"  
>>> m  
'pyth'  
>>> m += "on"  
>>> m  
'python'
```

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

语句(2)

所谓条件语句，顾名思义，就是依据某个条件，满足这个条件后执行下面的内容。

if

if，其含义就是：conj.（表条件）如果。if 发起的就是一个条件，它是构成条件语句的关键词。

```
>>> a = 8
>>> if a==8:
...     print a
...
8
```

在交互模式下，简单书写一下if发起的条件语句。特别说明，我上面这样写，只是简单演示一下。如果你要写大段的代码，千万不要在交互模式下写。

`if a==8:`，这句话里面如果条件 `a==8` 返回的是 True，那么就执行下面的语句。特别注意，冒号是必须的。下面一行语句 `print a` 要有四个空格的缩进。这是 Python 的特点，称之为语句块。

唯恐说的不严谨，我还是引用维基百科中的叙述：

Python 开发者有意让违反了缩排规则的程序不能通过编译，以此来强迫程序员养成良好的编程习惯。并且 Python 语言利用缩排表示语句块的开始和结束（Off-side 规则），而非使用花括号或者某种关键字。增加缩表示语句块的开始，而减少缩则表示语句块的结束。缩排成为了语法的一部分。例如 if 语句。

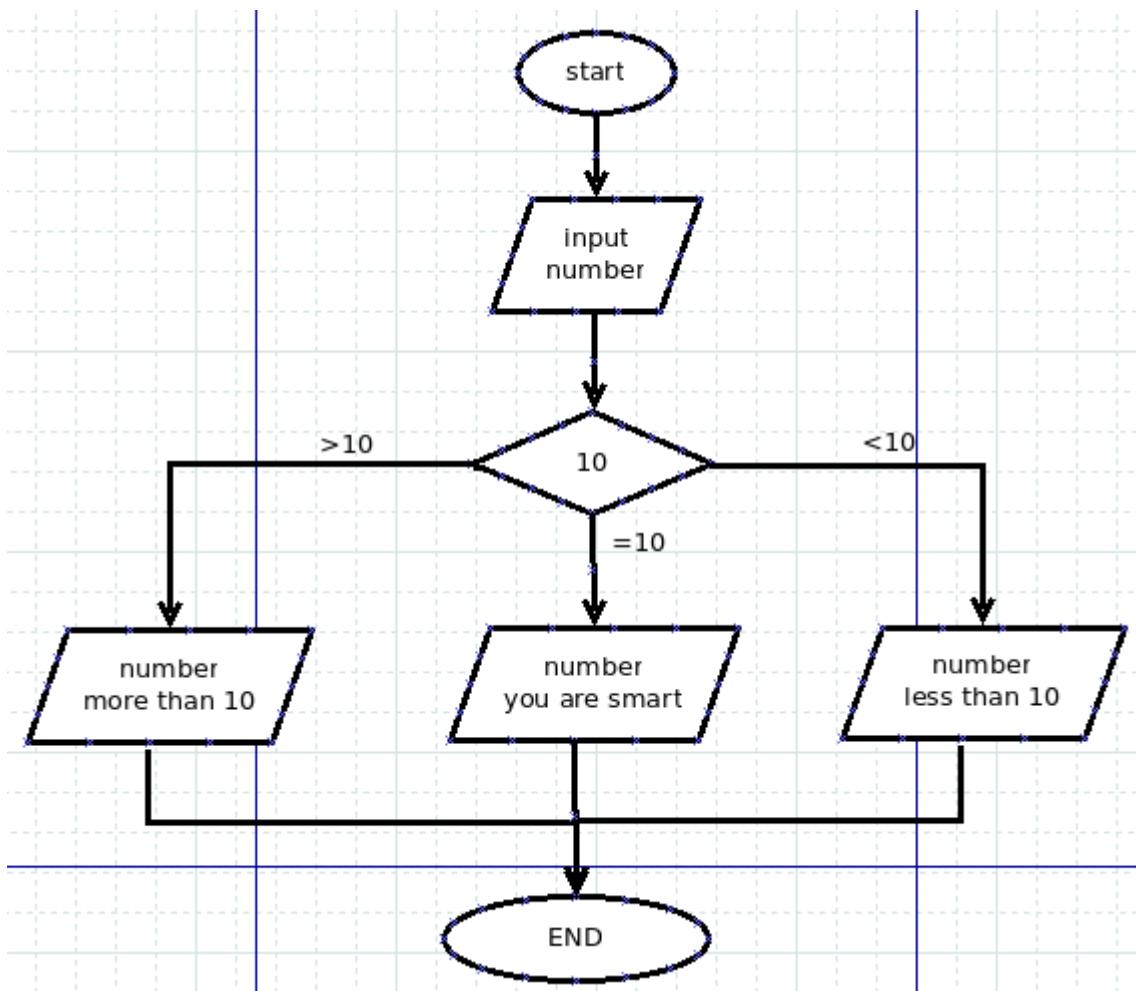
根据 PEP 的规定，必须使用 4 个空格来表示每级缩排。使用 Tab 字符和其它数目的空格虽然都可以编译通过，但不符合编码规范。支持 Tab 字符和其它数目的空格仅仅是为兼容很旧的 Python 程式和某些有问题的编辑程式。

从上面的这段话中，提炼出几个关键点：

- 必须要通过缩进方式来表示语句块的开始和结束
- 缩进用四个空格（也是必须的，别的方式或许可以，但不提倡）

if/else/elif

在进行条件判断的时候，只有 if，往往是不够的。比如下图所示的流程



这张图反应的是这样一个问题：

输入一个数字，并输出输入的结果，如果这个数字大于 10，那么同时输出大于 10，如果小于 10，同时输出提示小于 10，如果等于 10，就输出表扬的一句话。

从图中就已经显示出来了，仅仅用 if 来判断，是不足的，还需要其它分支。这就需要引入别的条件判断了。所以，有了 if...elif...else 语句。

基本样式结构：

```

if 条件 1:
    执行的内容 1
elif 条件 2:
    执行的内容 2
elif 条件 3:
    执行的内容 3
else:
    执行的内容 4
  
```

elif 用于多个条件时使用，可以没有。另外，也可以只有 if，而没有 else。

下面我们就不再交互模式中写代码了。打开文本编辑界面，你的编辑器也能提供这个功能，如果找不到，请回到[《写一个简单的程序》](#)查看。

代码实例如下：

```
#! /usr/bin/env python
#coding:utf-8

print "请输入任意一个整数数字："

number = int(raw_input()) #通过 raw_input()输入的数字是字符串
                         #用 int()将该字符串转化为整数

if number == 10:
    print "您输入的数字是: %d"%number
    print "You are SMART."
elif number > 10:
    print "您输入的数字是: %d"%number
    print "This number is more than 10."
elif number < 10:
    print "您输入的数字是: %d"%number
    print "This number is less than 10."
else:
    print "Are you a human?"
```

特别提醒看官注意，前面我们已经用过 `raw_input()` 函数了，这个是获得用户在界面上输入的信息，而通过它得到的是字符串类型的数据。

上述程序，依据条件进行判断，不同条件下做不同的事情了。需要提醒的是在条件中：`number == 10`，为了阅读方便，在 `number` 和 `==` 之间有一个空格最好了，同理，后面也有一个。这里的 `10` 是 `int` 类型，`number` 也是 `int` 类型。

把这段程序保存成一个扩展名是`.py` 的文件，比如保存为 `num.py`，进入到存储这个文件的目录，运行 `Python num.py`，就能看到程序执行结果了。下面是我执行的结果，供参考。

```
$ Python num.py
请输入任意一个整数数字:
12
您输入的数字是: 12
This number is more than 10.
```

```
$ Python num.py
请输入任意一个整数数字:
10
```

```

您输入的数字是: 10
You are SMART.

$ Python num.py
请输入任意一个整数数字:
9
您输入的数字是: 9
This number is less than 10.

```

不知道各位是否注意到，上面的那段代码，开始有一行：

```
#! /usr/bin/env python
```

这是什么意思呢？

这句话以 # 开头，表示本来不在程序中运行。这句话的用途是告诉机器寻找到该设备上的 Python 解释器，操作系统使用它找到的解释器来运行文件中的程序代码。有的程序里写的是 /usr/bin Python，表示 Python 解释器在 /usr/bin 里面。但是，如果写成 /usr/bin/env，则表示要通过系统搜索路径寻找 Python 解释器。不同系统，可能解释器的位置不同，所以这种方式能够让代码更将拥有可移植性。对了，以上是对 Unix 系列操作系统而言。对与 windows 系统，这句话就当不存在。

在“条件”中，就是上节提到的各种条件运算表达式，如果是 True，就执行该条件下的语句。

三元操作符

三元操作，是条件语句中比较简练的一种赋值方式，它的模样是这样的：

```

>>> name = "qiwsir" if "laoqi" else "github"
>>> name
'qiwsir'
>>> name = 'qiwsir' if "" else "python"
>>> name
'Python'
>>> name = "qiwsir" if "github" else ""
>>> name
'qiwsir'

```

总结一下：A = Y if X else Z

什么意思，结合前面的例子，可以看出：

- 如果 X 为真，那么就执行 A=Y
- 如果 X 为假，就执行 A=Z

如此例

```
>>> x = 2
>>> y = 8
>>> a = "python" if x>y else "qiwsir"
>>> a
'qiwsir'
>>> b = "python" if x<y else "qiwsir"
>>> b
'python'
```

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

语句(3)

循环，也是现实生活中常见的现象，我们常说日复一日，就是典型的循环。又如：日月更迭，斗转星移，无不是循环；王朝更迭；子子孙孙，繁衍不息，从某个角度看也都是循环。

编程语言就是要解决现实问题的，因此也少不了要循环。

在 Python 中，循环有一个语句：for 语句。

其基本结构是：

```
for 循环规则:  
    操作语句
```

从这个基本结构看，有着同 if 条件语句类似的地方：都有冒号；语句块都要缩进。是的，这是不可或缺的。

简单的 for 循环例子

前面介绍 print 语句的时候，出现了一个简单例子。重复一个类似的：

```
>>> hello = "world"  
>>> for i in hello:  
...     print i  
  
w  
o  
r  
l  
d
```

这个 for 循环是怎么工作的呢？

1. hello 这个变量引用的是"world"这个 str 类型的数据
2. 变量 i 通过 hello 找到它所引用的对象"world",因为 str 类型的数据属于序列类型，能够进行索引，于是就按照索引顺序，从第一字符开始，依次获得该字符的引用。
3. 当 i="w"的时候，执行 print i，打印出了字母 w，结束之后循环第二次，让 i="e"，然后执行 print i,打印出字母 e，如此循环下去，一直到最后一个字符被打印出来，循环自动结束。注意，每次打印之后，要换行。如果不换行，怎么办？参见《语句(1)》中关于 print 语句。

因为可以也通过使用索引（偏移量），得到序列对象的某个元素。所以，还可以通过下面的循环方式实现同样效果：

```
>>> for i in range(len(hello)):
...     print hello[i]
...
w
o
r
l
d
```

其工作方式是：

1. `len(hello)`得到 `hello` 引用的字符串的长度，为 5
2. `range(len(hello))`,就是 `range(5)`,也就是`[0, 1, 2, 3, 4]`,对应这"world"每个字母索引，也可以称之为偏移量。这里应用了一个新的函数 `range()`，关于它的用法，继续阅读，就能看到了。
3. `for i in range(len(hello))`,就相当于 `for i in [0,1,2,3,4]`,让`i`依次等于 `list` 中的各个值。当 `i=0` 时，打印 `hell`
`o[0]`，也就是第一个字符。然后顺序循环下去，直到最后一个 `i=4` 为止。

以上的循环举例中，显示了对 `str` 的字符依次获取，也涉及了 `list`，感觉不过瘾呀。那好，看下面对 `list` 的循环：

```
>>> ls_line
['Hello', 'I am qiwsi', 'Welcome you', '']
>>> for word in ls_line:
...     print word
...
Hello
I am qiwsi
Welcome you

>>> for i in range(len(ls_line)):
...     print ls_line[i]
...
Hello
I am qiwsi
Welcome you
```

range(start,stop[, step])

这个内建函数，非常有必要给予说明，因为它会经常被使用。一般形式是 `range(start, stop[, step])`

要研究清楚一些函数特别是内置函数的功能，建议看官首先要明白内置函数名称的含义。因为在 Python 中，名称不是随便取的，是代表一定意义的。所谓：名不正言不顺。

range

n. 范围；幅度；排；山脉 vi. (在...内) 变动；平行，列为一行；延伸；漫游；射程达到 vt. 漫游；放牧；使并列；归类于；来回走动

在具体实验之前，还是按照管理，摘抄一段[官方文档的原话](#)，让我们能够深刻理解之：

This is a versatile function to create lists containing arithmetic progressions. It is most often used in for loops. The arguments must be plain integers. If the step argument is omitted, it defaults to 1. If the start argument is omitted, it defaults to 0. The full form returns a list of plain integers [start, start + step, start + 2 * step, ...]. If step is positive, the last element is the largest start + i * step less than stop; if step is negative, the last element is the smallest start + i * step greater than stop. step must not be zero (or else ValueError is raised).

从这段话，我们可以得出关于 `range()` 函数的以下几点：

- 这个函数可以创建一个数字元素组成的列表。
- 这个函数最常用于 for 循环（关于 for 循环，马上就要涉及到了）
- 函数的参数必须是整数，默认从 0 开始。返回值是类似[start, start + step, start + 2*step, ...]的列表。
- step 默认值是 1。如果不写，就是按照此值。
- 如果 step 是正数，返回 list 的最最后的值不包含 stop 值，即 start+istep 这个值小于 stop；如果 step 是负数，start+istep 的值大于 stop。
- step 不能等于零，如果等于零，就报错。

在实验开始之前，再解释 `range(start,stop[,step])` 的含义：

- start：开始数值，默认为 0，也就是如果不写这项，就是认为 start=0
- stop：结束的数值，必须要写的。
- step：变化的步长，默认是 1，也就是不写，就是认为步长为 1。坚决不能为 0

实验开始，请以各项对照前面的讲述：

```
>>> range(9)      #stop=9, 别的都没有写, 含义就是 range(0,9,1)
[0, 1, 2, 3, 4, 5, 6, 7, 8] #从 0 开始, 步长为 1,增加, 直到小于 9 的那个数
>>> range(0,9)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

```
>>> range(0,9,1)
[0, 1, 2, 3, 4, 5, 6, 7, 8]

>>> range(1,9)      #start=1
[1, 2, 3, 4, 5, 6, 7, 8]

>>> range(0,9,2)    #step=2,每个元素等于 start+i*step,
[0, 2, 4, 6, 8]
```

仅仅解释一下 `range(0,9,2)`

- 如果是从 0 开始，步长为 1,可以写成 `range(9)` 的样子，但是，如果步长为 2，写成 `range(9,2)` 的样子，计算机就有点糊涂了，它会认为 `start=9,stop=2`。所以，在步长不为 1 的时候，切忌，要把 `start` 的值也写上。
- `start=0,step=2,stop=9`.`list` 中的第一个值是 `start=0`,第二个值是 `start+1step=2` (注意，这里是 1，不是 2，不要忘记，前面已经讲过，不论是 `list` 还是 `str`，对元素进行编号的时候，都是从 0 开始的)，第 n 个值就是 `start+(n-1)step`。直到小于 `stop` 前的那个值。

熟悉了上面的计算过程，看看下面的输入谁是什么结果？

```
>>> range(-9)
```

我本来期望给我返回 `[0,-1,-2,-3,-4,-5,-6,-7,-8]`,我的期望能实现吗？

分析一下，这里 `start=0,step=1,stop=-9`.

第一个值是 0；第二个是 `start+1*step`，将上面的数代入，应该是 1,但是最后一个还是 -9，显然出现问题了。但是，Python 在这里不报错，它返回的结果是：

```
>>> range(-9)
[]
>>> range(0,-9)
[]
>>> range(0)
[]
```

报错和返回结果，是两个含义，虽然返回的不是我们要的。应该如何修改呢？

```
>>> range(0,-9,-1)
[0, -1, -2, -3, -4, -5, -6, -7, -8]
>>> range(0,-9,-2)
[0, -2, -4, -6, -8]
```

有了这个内置函数，很多事情就简单了。比如：

```
>>> range(0,100,2)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68]
```

100 以内的自然数中的偶数组成的 list，就非常简单地搞定了。

思考一个问题，现在有一个列表，比如是["I","am","a","Pythoner","I","am","learning","it","with","qiwsir"],要得到这个 list 的所有序号组成的 list，但是不能一个一个用手指头来数。怎么办？

请沉思两分钟之后，自己实验一下，然后看下面。

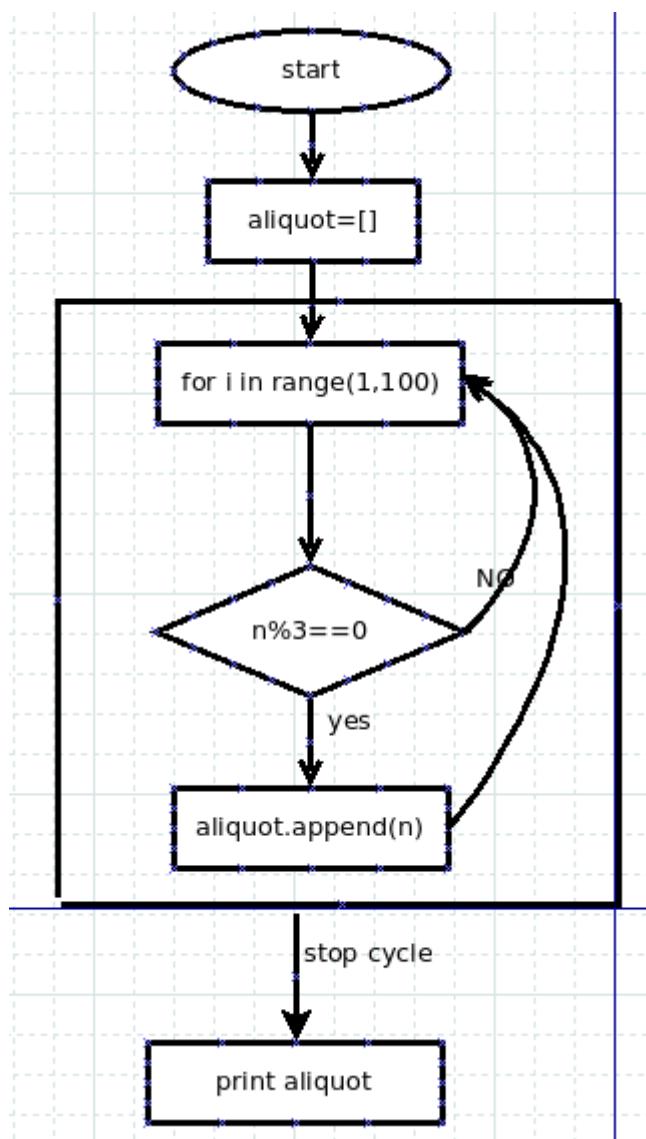
```
>>> pythoner
['I', 'am', 'a', 'pythoner', 'I', 'am', 'learning', 'it', 'with', 'qiwsir']
>>> py_index = range(len(pythoner))  #以 len(pythoner)为 stop 的值
>>> py_index
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

再用手指头指着 Pythoner 里面的元素，数一数，是不是跟结果一样。

例：找出 100 以内的能够被 3 整除的正整数。

分析：这个问题有两个限制条件，第一是 100 以内的正整数，根据前面所学，可以用 range(1,100)来实现；第二个是要解决被 3 整除的问题，假设某个正整数 n，这个数如果能够被 3 整除，也就是 $n \% 3$ (% 是取余数)为 0.那么如何得到 n 呢，就是要用 for 循环。

以上做了简单分析，要实现流程，还需要细化一下。按照前面曾经讲授过的一种方法，要画出问题解决的流程图。



下面写代码就是按图索骥了。

代码：

```

#!/usr/bin/env python
#coding:utf-8

aliquot = []

for n in range(1,100):
    if n%3 == 0:
        aliquot.append(n)

print aliquot
  
```

代码运行结果：

```
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99]
```

上面的代码中，将 for 循环和 if 条件判断都用上了。

不过，感觉有点麻烦，其实这么做就可以了：

```
>>> range(3,100,3)
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99]
```

能够用来 for 的对象

所有的序列类型对象，都能够用 for 来循环。比如：

```
>>> name_str = "qiwsir"
>>> for i in name_str: #可以对 str 使用 for 循环
...   print i,
...
q i w s i r

>>> name_list = list(name_str)
>>> name_list
['q', 'i', 'w', 's', 'i', 'r']
>>> for i in name_list: #对 list 也能用
...   print i,
...
q i w s i r

>>> name_set = set(name_str) #set 还可以用
>>> name_set
set(['q', 'i', 's', 'r', 'w'])
>>> for i in name_set:
...   print i,
...
q i s r w

>>> name_tuple = tuple(name_str)
>>> name_tuple
('q', 'i', 'w', 's', 'i', 'r')
>>> for i in name_tuple: #tuple 也能呀
...   print i,
...
q i w s i r

>>> name_dict={"name":"qiwsir","lang":"python","website":"qiwsir.github.io"}
```

```
>>> for i in name_dict:      #dict 也不例外，这里本质上是将字典的键拿出来，成为序列后进行循环
...   print i,"-->",name_dict[i]
...
lang --> Python
website --> qiwsir.github.io
name --> qiwsir
```

在用 for 来循环读取字典键值对上，需要多说几句。

有这样一个字典：

```
>>> a_dict = {"name":"qiwsir", "lang":"python", "email":"qiwsir@gmail.com", "website":"www.itdiffer.com"}
```

曾记否？在《[字典\(2\)](#)》中有获得字典键、值的函数：items/iteritems/keys/iterkeys/values/itervalues，通过这些函数得到的是键或者值的列表。

```
>>> for k in a_dict.keys():
...   print k, a_dict[k]
...
lang python
website www.itdiffer.com
name qiwsir
email qiwsir@gmail.com
```

这是最常用的一种获得字典键/值对的方法，而且效率也不错。

```
>>> for k,v in a_dict.items():
...   print k,v
...
lang python
website www.itdiffer.com
name qiwsir
email qiwsir@gmail.com

>>> for k,v in a_dict.iteritems():
...   print k,v
...
lang python
website www.itdiffer.com
name qiwsir
email qiwsir@gmail.com
```

这两种方法也能够实现同样的效果，但是因为有了上面的方法，一般就少用了。但是，用也无妨，特别是第二个 iteritems()，效率也是挺高的。

但是，要注意下面的方法：

```
>>> for k in a_dict.keys():
...     print k, a_dict[k]
...
lang python
website www.itdiffer.com
name qiwsir
email qiwsir@gmail.com
```

这种方法其实是不提倡的，虽然实现了同样的效果，但是效率常常是比较低的。切记。

```
>>> for v in a_dict.values():
...     print v
...
python
www.itdiffer.com
qiwsir
qiwsir@gmail.com

>>> for v in a_dict.itervalues():
...     print v
...
python
www.itdiffer.com
qiwsir
qiwsir@gmail.com
```

单独取 values，推荐第二种方法。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

语句(4)

for 循环在 Python 中应用广泛，所以，要用更多的篇幅来介绍。

并行迭代

关于迭代，在《[列表\(2\)](#)》中曾经提到过“可迭代的(iterator)”这个词，并给予了适当解释，这里再次提到“迭代”，说明它在 Python 中占有重要的位置。

迭代，在 Python 中表现就是用 for 循环，从序列对象中获得一定数量的元素。

在前面一节中，用 for 循环来获得列表、字符串、元组，乃至于字典的键值对，都是迭代。

现实中迭代不都是那么简单的，比如这个问题：

问题：有两个列表，分别是：a = [1,2,3,4,5], b = [9,8,7,6,5]，要计算这两个列表中对应元素的和。

解析：

太简单了，一看就知道结果了。

很好，这是你的方法，如果是 computer 姑娘来做，应该怎么做呢？

观察发现两个列表的长度一样，都是 5。那么对应元素求和，就是相同的索引值对应的元素求和，即 a[i]+b[i],(i=0,1,2,3,4)，这样一个个地就把相应元素和求出来了。当然，要用 for 来做这个事情了。

```
>>> a = [1,2,3,4,5]
>>> b = [9,8,7,6,5]
>>> c = []
>>> for i in range(len(a)):
...     c.append(a[i]+b[i])
...
>>> c
[10, 10, 10, 10, 10]
```

看来 for 的表现还不错。不过，这种方法虽然解决问题了，但 Python 总不会局限于一个解决之道。于是又有一个内建函数 `zip()`，可以让同样的问题有不一样的解决途径。

`zip` 是什么东西？在交互模式下用 `help(zip)`，得到官方文档是：

```
| zip(... zip(seq1 [, seq2 [...]]) -> [(seq1[0], seq2[0] ...), (...)]
```

Return a list of tuples, where each tuple contains the i-th element from each of the argument sequences. The returned list is truncated in length to the length of the shortest argument sequence.

seq1, seq2 分别代表了序列类型的数据。通过实验来理解上面的文档：

```
>>> a = "qiwsir"
>>> b = "github"
>>> zip(a,b)
[('q', 'g'), ('i', 't'), ('w', 'h'), ('s', 'u'), ('r', 'b')]
```

如果序列长度不同，那么就以"the length of the shortest argument sequence"为准。

```
>>> c = [1,2,3]
>>> d = [9,8,7,6]
>>> zip(c,d)
[(1, 9), (2, 8), (3, 7)]

>>> m = {"name": "lang"}
>>> n = {"qiwsir": "python"}
>>> zip(m,n)
[("lang", 'python'), ('name', 'qiwsir')]
```

m, n 是字典吗？当然不是。下面的才是字典呢。

```
>>> s = {"name": "qiwsir"}
>>> t = {"lang": "python"}
>>> zip(s,t)
[("name", 'lang')]
```

zip 是一个内置函数，它的参数必须是某种序列数据类型，如果是字典，那么键视为序列。然后将序列对应的元素依次组成元组，做一个 list 的元素。

下面是比較特殊的情况，参数是一个序列数据的时候，生成的结果样子：

```
>>> a
'qiwsir'
>>> c
[1, 2, 3]
>>> zip(c)
[(1,), (2,), (3,)]
>>> zip(a)
[('q',), ('i',), ('w',), ('s',), ('r',)]
```

很好的 `zip()`！那么就用它来解决前面那个两个列表中值对应相加吧。

```
>>> d = []
>>> for x,y in zip(a,b):
...     d.append(x+y)
...
>>> d
[10, 10, 10, 10, 10]
```

多么优雅的解决！

比较这个问题的两种解法，似乎第一种解法适用面较窄，比如，如果已知给定的两个列表长度不同，第一种解法就出问题了。而第二种解法还可以继续适用。的确如此，不过，第一种解法也不是不能修订的。

```
>>> a = [1,2,3,4,5]
>>> b = ["python","www.itdiffer.com","qiwsir"]
```

如果已知是这样两个列表，要讲对应的元素“加起来”。

```
>>> length = len(a) if len(a)<len(b) else len(b)
>>> length
3
```

首先用这种方法获得两个列表中最短的那个列表的长度。看那句三元操作，这是非常 Pythonic 的写法啦。写出这句，就可以冒充高手了。哈哈。

```
>>> for i in range(length):
...     c.append(str(a[i]) + ":" + b[i])
...
>>> c
['1:python', '2:www.itdiffer.com', '3:qiwsir']
```

我还是用第一个思路做的，经过这么修正一下，也还能用。要注意一个细节，在“加”的时候，不能直接用 `a[i]`，因为它引用的对象是一个 `int` 类型，不能跟后面的 `str` 类型相加，必须转化一下。

当然，`zip()` 也是能解决这个问题的。

```
>>> d = []
>>> for x,y in zip(a,b):
...     d.append(x + y)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

报错！看错误信息，我刚刚提醒的那个问题就冒出来了。所以，应该这么做：

```
>>> for x,y in zip(a,b):
...     d.append(str(x) + ":" + y)
...
>>> d
['1:python', '2:www.itdiffer.com', '3:qiwsir']
```

这才得到了正确结果。

切记：computer 是一个姑娘，她非常秀气，需要敲代码的小伙子们耐心地、细心地跟她相处。

以上两种写法那个更好呢？前者？后者？哈哈。我看差不多了。

```
>>> result
[(2, 11), (4, 13), (6, 15), (8, 17)]
>>> zip(*result)
[(2, 4, 6, 8), (11, 13, 15, 17)]
```

zip() 还能这么干，是不是有点意思？

下面延伸一个问题：

问题：有一个 dictionary，`myinfor = {"name":"qiwsir","site":"qiwsir.github.io","lang":"python"}`, 将这个字典变换成：`infor = {"qiwsir":"name","qiwsir.github.io":"site","python":"lang"}`

解析：

解法有几个，如果用 for 循环，可以这样做（当然，看官如果有方法，欢迎贴出来）。

```
>>> infor = {}
>>> for k,v in myinfor.items():
...     infor[v]=k
...
>>> infor
{'python': 'lang', 'qiwsir.github.io': 'site', 'qiwsir': 'name'}
```

下面用 zip() 来试试：

```
>>> dict(zip(myinfor.values(),myinfor.keys()))
{'python': 'lang', 'qiwsir.github.io': 'site', 'qiwsir': 'name'}
```

呜呼，这是什么情况？原来这个 zip() 还能这样用。是的，本质上是这么回事情。如果将上面这一行分解开来，看官就明白其中的奥妙了。

```
>>> myinfor.values() #得到两个 list
['Python', 'qiwsir', 'qiwsir.github.io']
>>> myinfor.keys()
```

```
[lang', 'name', 'site']
>>> temp = zip(myinfor.values(), myinfor.keys()) #压缩成一个 list, 每个元素是一个 tuple
>>> temp
[('python', 'lang'), ('qiwsir', 'name'), ('qiwsir.github.io', 'site')]

>>> dict(temp) #这是函数 dict() 的功能, 将上述列表转化为 dictionary
{'Python': 'lang', 'qiwsir.github.io': 'site', 'qiwsir': 'name'}
```

至此，是不是明白 zip() 和循环的关系了呢？有了它可以让某些循环简化。

enumerate

这是一个有意思的内置函数，本来我们可以通过 `for i in range(len(list))` 的方式得到一个 list 的每个元素索引，然后在用 `list[i]` 的方式得到该元素。如果要同时得到元素索引和元素怎么办？就是这样了：

```
>>> for i in range(len(week)):
...     print week[i] + ' is ' + str(i) #注意, i 是 int 类型, 如果和前面的用 + 连接, 必须是 str 类型
...
monday is 0
sunday is 1
friday is 2
```

Python 中提供了一个内置函数 `enumerate`，能够实现类似的功能

```
>>> for (i, day) in enumerate(week):
...     print day + ' is ' + str(i)
...
monday is 0
sunday is 1
friday is 2
```

官方文档是这么说的：

Return an enumerate object. sequence must be a sequence, an iterator, or some other object which supports iteration. The `next()` method of the iterator returned by `enumerate()` returns a tuple containing a count (from start which defaults to 0) and the values obtained from iterating over sequence:

顺便抄录几个例子，供看官欣赏，最好实验一下。

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

对于这样一个列表：

```
>>> mylist = ["qiwsir", 703, "python"]
>>> enumerate(mylist)
<enumerate object at 0xb74a63c4>
```

出现这个结果，用 list 就能实现转换，显示内容。意味着可迭代。

```
>>> list(enumerate(mylist))
[(0, 'qiwsir'), (1, 703), (2, 'python')]
```

再设计一个小问题，练习一下这个函数。

问题：将字符串中的某些字符替换为其它的字符串。原始字符串"Do you love Canglaoshi? Canglaoshi is a good teacher."，请将"Canglaoshi"替换为"PHP"。

解析：

```
>>> raw = "Do you love Canglaoshi? Canglaoshi is a good teacher."
```

这是所要求的那个字符串，当时，不能直接对这个字符串使用 `enumerate()`，因为它会变成这样：

```
>>> list(enumerate(raw))
[(0, 'D'), (1, 'o'), (2, ' '), (3, 'y'), (4, 'o'), (5, 'u'), (6, ' '), (7, 'l'), (8, 'o'), (9, 'v'), (10, 'e'), (11, ' '), (12, 'C'), (13, 'a'), (14, 'n'), (15, 'g'), (16,
```

这不是所需要的。所以，先把 raw 转化为列表：

```
>>> raw_lst = raw.split(" ")
```

然后用 `enumerate()`

```
>>> for i, string in enumerate(raw_lst):
...     if string == "Canglaoshi":
...         raw_lst[i] = "PHP"
...
```

没有什么异常现象，查看一下那个 raw_lst 列表，看看是不是把"Canglaoshi"替换为"PHP"了。

```
>>> raw_lst
['Do', 'you', 'love', 'Canglaoshi?', 'PHP', 'is', 'a', 'good', 'teacher.']}
```

只替换了一个，还有一个没有替换。为什么？仔细观察发现，没有替换的那个是'Canglaoshi?'，跟条件判断中的"Canglaoshi"不一样。

修改一下，把条件放宽：

```
>>> for i, string in enumerate(raw_lst):
...     if "Canglaoshi" in string:
...         raw_lst[i] = "PHP"
...
>>> raw_lst
['Do', 'you', 'love', 'PHP', 'PHP', 'is', 'a', 'good', 'teacher.']}
```

好的。然后呢？再转化为字符串？留给读者试试。

list 解析

先看下面的例子，这个例子是想得到 1 到 9 的每个整数的平方，并且将结果放在 list 中打印出来

```
>>> power2 = []
>>> for i in range(1,10):
...     power2.append(i*i)
...
>>> power2
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Python 有一个非常有意思的功能，就是 list 解析，就是这样的：

```
>>> squares = [x**2 for x in range(1,10)]
>>> squares
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

看到这个结果，看官还不惊叹吗？这就是 Python，追求简洁优雅的 Python！

其官方文档中有这样一段描述，道出了 list 解析的真谛：

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

这就是 Python 有意思的地方，也是计算机高级语言编程有意思的地方，你只要动脑筋，总能找到惊喜的东西。

其实，不仅仅对数字组成的 list，所有的都可以如此操作。请在平复了激动的心之后，默默地看下面的代码，感悟一下 list 解析的魅力。

```
>>> mybag = [' glass', ' apple', 'green leaf '] #有的前面有空格，有的后面有空格
>>> [one.strip() for one in mybag]           #去掉元素前后的空格
['glass', 'apple', 'green leaf']
```

上面的问题，都能用 list 解析来重写。读者不妨试试。

在很多情况下，list 解析的执行效率高，代码简洁明了。是实际写程序中经常被用到的。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

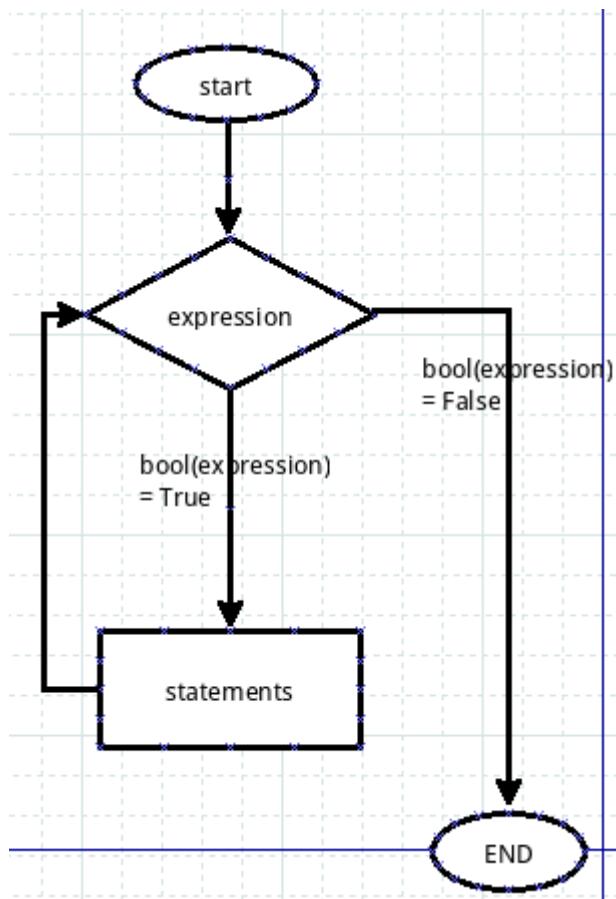
语句(5)

while，翻译成中文是“当...的时候”，这个单词在英语中，常常用来做为时间状语，while ... someone do something，这种类型的说法是有的。在 Python 中，它也有这个含义，不过有点区别的是，“当...时候”这个条件成立在一段范围或者时间间隔内，从而在这段时间间隔内让 Python 做好多事情。就好比这样一段情景：

```
while 年龄大于 60 岁: ----->当年龄大于 60 岁的时候
    退休      ----->凡是符合上述条件就执行的动作
```

展开想象，如果制作一道门，这道门就是用上述的条件调控开关的，假设有很多人经过这个门，报上年龄，只要年龄大于 60，就退休（门打开，人可以出去），一个接一个地这样循环下去，突然有一个人年龄是 50，那么这个循环在他这里就停止，也就是这时候他不满足条件了。

这就是 while 循环。写一个严肃点的流程，可以看下图：



做猜数字游戏

前不久，有一个在校的大学生朋友（他叫李航），给我发邮件，让我看了他做的游戏，能够实现多次猜数，直到猜中为止。这是一个多么喜欢学习的大学生呀。

我在这里将他写的程序恭录于此，如果李航同学认为此举侵犯了自己的知识产权，可以告知我，我马上撤下此代码。

```
#! /usr/bin/env Python
#coding:UTF-8

import random

i=0
while i < 4:
    print"*****"
    num = input('请您输入 0 到 9 任一个数：')      #李同学用的是 Python3

    xnum = random.randint(0,9)

    x = 3 - i

    if num == xnum:
        print'运气真好，您猜对了！'
        break
    elif num > xnum:
        print'"您猜大了!\n 哈哈,正确答案是:%s\n 您还有 %s 次机会！"'%(xnum,x)
    elif num < xnum:
        print'"您猜小了!\n 哈哈,正确答案是:%s\n 您还有 %s 次机会！"'%(xnum,x)
    print"*****"

    i += 1
```

我们就用这段程序来分析一下，首先看 `while i<4`，这是程序中为猜测限制了次数，最大是三次，请看官注意，在 `while` 的循环体中的最后一句：`i +=1`，这就是说每次循环到最后，就给 `i` 增加 1，当 `bool(i<4)` 为 `False` 的时候，就不再循环了。

当 `bool(i<4)` 为 `True` 的时候，就执行循环体内的语句。在循环体内，让用户输入一个整数，然后程序随机选择一个整数，最后判断随机生成的数和用户输入的数是否相等，并且用 `if` 语句判断三种不同情况。

根据上述代码，看官看看是否可以修改？

为了让用户的体验更爽，不妨把输入的整数范围扩大，在1到100之间吧。

```
num_input = raw_input("please input one integer that is in 1 to 100:") #我用的是Python2.7，在输入指令上区别于李同学
```

程序用 num_input 变量接收了输入的内容。但是，请列位看官一定要注意，看到这里想睡觉的要打起精神了，我要分享一个多年编程经验：

请牢记：任何用户输入的内容都是不可靠的。

这句话含义深刻，但是，这里不做过多的解释，需要各位在随后的编程生涯中体验了。为此，我们要检验用户输入的是否符合我们的要求，我们要求用户输入的是1到100之间的整数，那么就要做如下检验：

1. 输入的是否是整数
2. 如果是整数，是否在1到100之间。

为此，要做：

```
if not num_input.isdigit(): #str.isdigit()是用来判断字符串是否纯粹由数字组成
    print "Please input interger."
elif int(num_input)<0 and int(num_input)>=100:
    print "The number should be in 1 to 100."
else:
    pass #这里用 pass，意思是暂时省略，如果满足了前面提出的要求，就该执行此处语句
```

再看看李航同学的程序，在循环体内产生一个随机的数字，这样用户每次输入，面对的都是一个新的随机数字。这样的猜数字游戏难度太大了。我希望是程序产生一个数字，直到猜中，都是这个数字。所以，要把产生随机数字这个指令移动到循环之前。

```
import random

number = random.randint(1,100)

while True:      #不限制用户的次数了
    ...
```

观察李同学的程序，还有一点需要向列位说明的，那就是在条件表达式中，两边最好是同种类型数据，上面的程序中有：num>xnum 样式的条件表达式，而一边是程序生成的 int 类型数据，一边是通过输入函数得到的 str 类型数据。在某些情况下可以运行，为什么？看官能理解吗？都是数字的时候，是可以的。但是，这样不好。

那么，按照这种思路，把这个猜数字程序重写一下：

```
#!/usr/bin/env python
#coding:utf-8
```

```

import random

number = random.randint(1,101)

guess = 0

while True:

    num_input = raw_input("please input one integer that is in 1 to 100:")
    guess += 1

    if not num_input.isdigit():
        print "Please input interger."
    elif int(num_input) < 0 or int(num_input) >= 100:
        print "The number should be in 1 to 100."
    else:
        if number == int(num_input):
            print "OK, you are good.It is only %d, then you successed." % guess
            break
        elif number > int(num_input):
            print "your number is more less."
        elif number < int(num_input):
            print "your number is bigger."
        else:
            print "There is something bad, I will not work"

```

以上供参考，看官还可改进。

break 和 continue

break,在上面的例子中已经出现了，其含义就是要在这个地方中断循环，跳出循环体。下面这个简要的例子更明显：

```

#!/usr/bin/env python
#coding:utf-8

a = 8
while a:
    if a%2 == 0:
        break
    else:
        print "%d is odd number"%a
        a = 0
print "%d is even number"%a

```

a=8 的时候，执行循环体中的 break，跳出循环，执行最后的打印语句，得到结果：

```
8 is even number
```

如果 a=9，则要执行 else 里面的的 print，然后 a=0，循环就在执行一次，又 break 了，得到结果：

```
9 is odd number
0 is even number
```

而 continue 则是要从当前位置（即 continue 所在的位置）跳到循环体的最后一行的后面（不执行最后一行），对一个循环体来讲，就如同首尾衔接一样，最后一行的后面是哪里呢？当然是开始了。

```
#!/usr/bin/env python
#coding:utf-8

a = 9
while a:
    if a%2==0:
        a -=1
        continue #如果是偶数，就返回循环的开始
    else:
        print "%d is odd number"%a #如果是奇数，就打印出来
        a -=1
```

其实，对于这两东西，我个人在编程中很少用到。我有一个固执的观念，尽量将条件在循环之前做足，不要在循环中跳来跳去，不仅可读性下降，有时候自己也糊涂了。

while...else

这两个的配合有点类似 if ... else，只需要一个例子列为就理解了，当然，一遇到 else 了，就意味着已经不在 while 循环内了。

```
#!/usr/bin/env Python

count = 0
while count < 5:
    print count, " is less than 5"
    count = count + 1
else:
    print count, " is not less than 5"
```

执行结果：

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

for...else

除了有 while...else 外，还可以有 for...else。这个循环也通常用在当跳出循环之后要做的事情。

```
#!/usr/bin/env python
# coding=utf-8

from math import sqrt

for n in range(99, 1, -1):
    root = sqrt(n)
    if root == int(root):
        print n
        break

else:
    print "Nothing."
```

读 读者是否能够读懂这段代码的含义？

阅读代码是一个提升自己编程水平的好方法。如何阅读代码？像看网上新闻那样吗？一目只看自己喜欢的文字，甚至标题看不完就开始喷。

绝对不是这样，如果这样，不是阅读代码呢。阅读代码的最好方法是给代码做注释。对，如果可能就给每行代码做注释。这样就能理解代码的含义了。

上面的代码，读者不妨做注释，看看它到底在干什么。如果把 `for n in range(99, 1, -1)` 修改为 `for n in range(9, 81, -1)` 看看是什么结果？

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

文件(1)

文件，是 computer 姑娘中非常重要的东西，在 Python 里，它也是一种类型的对象，类似前面已经学习过的其它数据类型，包括文本的、图片的、音频的、视频的等等，还有不少没见过的扩展名的。事实上，在 linux 操作系统中，所有的东西都被保存到文件中。

先在交互模式下查看一下文件都有哪些属性：

```
>>> dir(file)
```

```
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__', '__format__', '__getattribute__', '__hash__', '__init__', '__iter__']
```

然后对部分属性进行详细说明，就是看官学习了。

特别注意观察，在上面有 `__iter__` 这个东西。曾经在讲述列表的时候，是不是也出现这个东西了呢？是的。它意味着这种类型的数据是可迭代的(iterable)。在下面的讲解中，你就会看到了，能够用 `for` 来读取其中的内容。

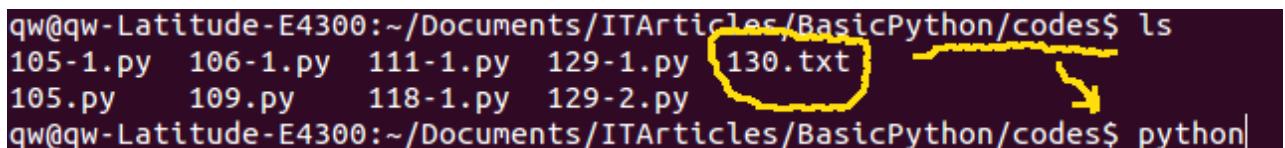
打开文件

在某个文件夹下面建立了一个文件，名曰：130.txt，并且在里面输入了如下内容：

```
learn python
http://qiwsir.github.io
qiwsir@gmail.com
```

此文件一共三行。

下图显示了这个文件的存储位置：



```
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ ls
105-1.py  106-1.py  111-1.py  129-1.py  130.txt
105.py    109.py    118-1.py  129-2.py
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ python
```

在上面截图中，我在当前位置输入了 Python（我已经设置了环境变量，如果你没有，需要写全启动 Python 命令路径），进入到交互模式。在这个交互模式下，这样操作：

```
>>> f = open("130.txt")  # 打开已经存在的文件
>>> for line in f:
...     print line
...
learn Python
```

```
http://qiwsir.github.io
```

```
qiwsir@gmail.com
```

提醒初学者注意，在那个文件夹输入了启动 Python 交互模式的命令，那么，如果按照上面的方法 `open("130.txt")` 打开文件，就意味着这个文件 `130.txt` 是在当前文件夹内的。如果要打开其它文件夹内的文件，请用相对路径或者绝对路径来表示，从而让 python 能够找到那个文件。

将打开的文件，赋值给变量 `f`，这样也就是变量 `f` 跟对象文件 `130.txt` 用线连起来了（对象引用），本质上跟前面所讲述的其它类型数据进行赋值是一样的。

接下来，用 `for` 来读取文件中的内容，就如同读取一个前面已经学过的序列对象一样，如 `list`、`str`、`tuple`，把读到的文件中的每行，赋值给变量 `line`。也可以理解为，`for` 循环是一行一行地读取文件内容。每次扫描一行，遇到行结束符号 `\n` 表示本行结束，然后是下一行。

从打印的结果看出，每一行跟前面看到的文件内容中的每一行是一样的。只是行与行之间多了一空行，前面显示文章内容的时候，没有这个空行。或许这无关紧要，但是，还要深究一下，才能豁然。

在原文中，每行结束有本行结束符号 `\n`，表示换行。在 `for` 语句汇总，`print line` 表示每次打印完 `line` 的对象之后，就换行，也就是打印完 `line` 的对象之后会增加一个 `\n`。这样看来，在每行末尾就有两个 `\n`，即：`\n\n`，于是在打印中就出现了一个空行。

```
>>> f = open('130.txt')
>>> for line in f:
...     print line,    #后面加一个逗号，就去掉了原来默认增加的\n了，看看，少了空行。
...
learn Python
http://qiwsir.github.io
qiwsir@gmail.com
```

在进行上述操作的时候，有没有遇到这样的情况呢？

```
>>> f = open('130.txt')
>>> for line in f:
...     print line,
...
learn Python
http://qiwsir.github.io
qiwsir@gmail.com

>>> for line2 in f:    #在前面通过 for 循环读取了文件内容之后，再次读取，
...     print line2    #然后打印，结果就什么也显示，这是什么问题？
...
>>>
```

如果看官没有遇到上面问题，可以试试。这不是什么错误，是因为前一次已经读取了文件内容，并且到了文件的末尾了。再重复操作，就是从末尾开始继续读了。当然显示不了什么东西，但是Python并不认为这是错误，因为后面就会讲到，或许在这次读取之前，已经又向文件中追加内容了。那么，如果要再次读取怎么办？就从新来一边好了。这就好比有一个指针在指着文件中的每一行，每读完一行，指针向移动一行。直到指针指向了文件的最末尾。当然，也有办法把指针移动到任何位置。

特别提醒看官，因为当前的交互模式是在该文件所在目录启动的，所以，就相当于这个实验室和文件130.txt是同一个目录，这时候我们打开文件130.txt，就认为是在本目录中打开，如果文件不是在本目录中，需要写清楚路径。

比如：在上一级目录中（~/Documents/ITArticles/BasicPython），假如我进入到那个目录中，运行交互模式，然后试图打开130.txt文件。

```
~/Documents/ITArticles/BasicPython/codes$ ls
11-1.py 129-1.py 130.txt
18-1.py 129-2.py
~/Documents/ITArticles/BasicPython/codes$ cd ..
~/Documents/ITArticles/BasicPython$ python
```

```
>>> f = open("130.txt")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: '130.txt'
```

```
>>> f = open("./codes/130.txt") #必须得写上路径了（注意，windows的路径是\隔开，需要转义。对转义符，看官看以前讲座）
>>> for line in f:
...     print line
...
learn Python
```

<http://qiwsir.github.io>

qiwsir@gmail.com

>>>

创建文件

上面的实验中，打开的是一个已经存在的文件。如何创建文件呢？

```
>>> nf = open("131.txt","w")
>>> nf.write("This is a file")
```

就这样创建了一个文件？并写入了文件内容呢？看看再说：

```
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ ls
105-1.py 106-1.py 111-1.py 129-1.py 130.txt
105.py    109.py    118-1.py 129-2.py 131.txt
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ cat 131.txt
This is a file
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$
```

真的就这样创建了新文件，并且里面有那句话呢。

看官注意了没有，这次我们同样是用 open() 这个函数，但是多了个"w"，这是在告诉 Python 用什么样的模式打开文件。也就是说，用 open() 打开文件，可以有不同的模式打开。看下表：

模式	描述
r	以读方式打开文件，可读取文件信息。
w	以写方式打开文件，可向文件写入信息。如文件存在，则清空该文件，再写入新内容
a	以追加模式打开文件（即一打开文件，文件指针自动移到文件末尾），如果文件不存在则创建
r+	以读写方式打开文件，可对文件进行读和写操作。
w+	消除文件内容，然后以读写方式打开文件。
a+	以读写方式打开文件，并把文件指针移到文件尾。
b	以二进制模式打开文件，而不是以文本模式。该模式只对 Windows 或 Dos 有效，类 Unix 的文件是用二进制模式进行操作的。

从表中不难看出，不同模式下打开文件，可以进行相关的读写。那么，如果什么模式都不写，像前面那样呢？那样就是默认为 r 模式，只读的方式打开文件。

```
>>> f = open("130.txt")
>>> f
<open file '130.txt', mode 'r' at 0xb7530230>
>>> f = open("130.txt","r")
>>> f
<open file '130.txt', mode 'r' at 0xb750a700>
```

可以用这种方式查看当前打开的文件是采用什么模式的，上面显示，两种模式是一样的效果，如果不写那个"r"，就默认为是只读模式了。下面逐个对各种模式进行解释

"w":以写方式打开文件，可向文件写入信息。如文件存在，则清空该文件，再写入新内容

131.txt 这个文件是存在的，前面建立的，并且在里面写了一句话：This is a file

```
>>> fp = open("131.txt")
>>> for line in fp:      #原来这个文件里面的内容
...   print line
...
```

```
This is a file
>>> fp = open("131.txt","w") #这时候再看看这个文件，里面还有什么呢？是不是空了呢？
>>> fp.write("My name is qiwsir.\nMy website is qiwsir.github.io") #再查看内容
>>> fp.close()
```

查看文件内容：

```
$ cat 131.txt #cat 是 linux 下显示文件内容的命令，这里就是要显示 131.txt 内容
My name is qiwsir.
My website is qiwsir.github.io
```

"a":以追加模式打开文件（即一打开文件，文件指针自动移到文件末尾），如果文件不存在则创建

```
>>> fp = open("131.txt","a")
>>> fp.write("\nAha,I like program\n") #向文件中追加
>>> fp.close() #这是关闭文件，一定要养成一个习惯，写完内容之后就关闭
```

查看文件内容：

```
$ cat 131.txt
My name is qiwsir.
My website is qiwsir.github.io
Aha,I like program
```

其它项目就不一一讲述了。看官可以自己实验。

使用 with

在对文件进行写入操作之后，一定要牢记一个事情： `file.close()`，这个操作千万不要忘记，忘记了怎么办，那就补上吧，也没有什么天塌地陷的后果。

有另外一种方法，能够不用这么让人揪心，实现安全地关闭文件。

```
>>> with open("130.txt","a") as f:
...     f.write("\nThis is about 'with...as...'")

...
>>> with open("130.txt","r") as f:
...     print f.read()

...
learn python
http://qiwsir.github.io
qiwsir@gmail.com
hello
```

```
This is about 'with...as...'>>>
```

这里就不用 close() 了。而且这种方法更有 Python 味道，或者说是更符合 Pythonic 的一个要求。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

文件(2)

上一节，对文件有了初步认识。读者要牢记，文件无非也是一种类型的数据。

文件的状态

很多时候，我们需要获取一个文件的有关状态（也称为属性），比如创建日期，访问日期，修改日期，大小，等等。在 os 模块中，有这样一个方法，专门让我们查看文件的这些状态参数的。

```
>>> import os
>>> file_stat = os.stat("131.txt")    #查看这个文件的状态
>>> file_stat                      #文件状态是这样的。从下面的内容，有不少从英文单词中可以猜测出来。
posix.stat_result(st_mode=33204, st_ino=5772566L, st_dev=2049L, st_nlink=1, st_uid=1000, st_gid=1000, st_size=69L,
                  st_atime=1407734600.0882277, st_mtime=1407734600.0882277, st_ctime=1407734600.0882277)
```

这是什么时间？看不懂！别着急，换一种方式。在 Python 中，有一个模块 time，是专门针对时间设计的。

```
>>> import time
>>> time.localtime(file_stat.st_ctime) #这回看清楚了。
time.struct_time(tm_year=2014, tm_mon=8, tm_mday=11, tm_hour=13, tm_min=23, tm_sec=20, tm_wday=0, tm_yday=223)
```

read/readline/readlines

上节中，简单演示了如何读取文件内容，但是，在用 dir(file) 的时候，会看到三个函数：read/readline/readlines，它们各自有什么特点，为什么要三个？一个不行吗？

在读者向下看下面内容之前，请想一想，如果要回答这个问题，你要用什么方法？注意，我问的是用什么方法能够找到答案，不是问答案内容是什么。因为内容，肯定是在某个地方存放着呢，关键是用什么方法找到。

搜索？是一个不错的方法。

还有一种，就是在交互模式下使用的，你肯定也想到了。

```
>>> help(file.read)
```

用这样的方法，可以分别得到三个函数的说明：

`read(...)`
`read([size])` -> read at most size bytes, returned as a string.

If the size argument is negative or omitted, read until EOF is reached.
Notice that when in non-blocking mode, less data than what was requested may be returned, even if no size parameter was given.

`readline(...)`
`readline([size])` -> next line from the file, as a string.

Retain newline. A non-negative size argument limits the maximum number of bytes to return (an incomplete line may be returned then).
Return an empty string at EOF.

`readlines(...)`
`readlines([size])` -> list of strings, each a line from the file.

Call readline() repeatedly and return a list of the lines so read.
The optional size argument, if given, is an approximate bound on the total number of bytes in the lines returned.

对照一下上面的说明，三个的异同就显现了。

EOF 什么意思？End-of-file。在[维基百科](#)中居然有对它的解释：

In computing, End Of File (commonly abbreviated EOF[1]) is a condition in a computer operating system where no more data can be read from a file.

明白 EOF 之后，就对比一下：

- `read`: 如果指定了参数 `size`, 就按照该指定长度从文件中读取内容, 否则, 就读取全文。被读出来的内容, 全部塞到一个字符串里面。这样有好处, 就是东西都到内存里面了, 随时取用, 比较快捷; “成也萧何败萧何”, 也是因为这点, 如果文件内容太多了, 内存会吃不消的。文档中已经提醒注意在“non-blocking”模式下的问题, 关于这个问题, 不是本节的重点, 暂时不讨论。
- `readline`: 那个可选参数 `size` 的含义同上。它则是以行为单位返回字符串, 也就是每次读一行, 依次循环, 如果不限定 `size`, 直到最后一个返回的是空字符串, 意味着到文件末尾了(EOF)。
- `readlines`: `size` 同上。它返回的是以行为单位的列表, 即相当于先执行 `readline()`, 得到每一行, 然后把这一行的字符串作为列表中的元素塞到一个列表中, 最后将此列表返回。

依次演示操作, 即可明了。有这样一个文档, 名曰: `you.md`, 其内容和基本格式如下:

You Raise Me Up When I am down and, oh my soul, so weary; When troubles come and my heart burdened be; Then, I am still and wait here in the silence, Until you come and sit awhile with me. You r

aise me up, so I can stand on mountains; You raise me up, to walk on stormy seas; I am strong, when I am on your shoulders; You raise me up: To more than I can be.

分别用上述三种函数读取这个文件。

```
>>> f = open("you.md")
>>> content = f.read()
>>> content
'You Raise Me Up\nWhen I am down and, oh my soul, so weary;\nWhen troubles come and my heart burdened be;\nThen, I am still and wait here in the silence,\nUntil you come and sit awhile with me.\nYou raise me up, so I can stand on mountains;\nYou raise me up, to walk on stormy seas;\nI am strong, when I am on your shoulders;\nYou raise me up: To more than I can be.'
```

提示：养成一个好习惯，只要打开文件，不用该文件了，就一定要随手关闭它。如果不关闭它，它还驻留在内存中，后面又没有对它的操作，是不是浪费内存空间了呢？同时也增加了文件安全的风险。

注意：在 Python 中，'\n'表示换行，这也是 UNIX 系统中的规范。但是，在奇葩的 windows 中，用'\r\n'表示换行。Python 在处理这个的时候，会自动将'\r\n'转换为'\n'。

请仔细观察，得到的就是一个大大的字符串，但是这个字符串里面包含着一些符号 '\n'，因为原文中有换行符。如果用 print 输出这个字符串，就是这样的了，其中的 '\n' 起作用了。

```
>>> print content
You Raise Me Up
When I am down and, oh my soul, so weary;
When troubles come and my heart burdened be;
Then, I am still and wait here in the silence,
Until you come and sit awhile with me.
You raise me up, so I can stand on mountains;
You raise me up, to walk on stormy seas;
I am strong, when I am on your shoulders;
You raise me up: To more than I can be.
```

用 readline() 读取，则是这样的：

```
>>> f = open("you.md")
>>> f.readline()
'You Raise Me Up\n'
>>> f.readline()
'When I am down and, oh my soul, so weary;\n'
>>> f.readline()
'When troubles come and my heart burdened be;\n'
>>> f.close()
```

显示出一行一行读取了，每操作一次 f.readline()，就读取一行，并且将指针向下移动一行，如此循环。显然，这是一种循环，或者说可迭代的。因此，就可以用循环语句来完成对全文的读取。

```

#!/usr/bin/env Python
# coding=utf-8

f = open("you.md")

while True:
    line = f.readline()
    if not line:      #到 EOF, 返回空字符串, 则终止循环
        break
    print line ,     #注意后面的逗号, 去掉 print 语句后面的 '\n', 保留原文件中的换行

f.close()          #别忘记关闭文件

```

将其和文件"you.md"保存在同一个目录中, 我这里命名的文件名是 12701.py, 然后在该目录中运行 `Python 12701.py`, 就看到下面的效果了:

```

~/Documents$ python 12701.py
You Raise Me Up
When I am down and, oh my soul, so weary;
When troubles come and my heart burdened be;
Then, I am still and wait here in the silence,
Until you come and sit awhile with me.
You raise me up, so I can stand on mountains;
You raise me up, to walk on stormy seas;
I am strong, when I am on your shoulders;
You raise me up: To more than I can be.

```

也用 `readlines()` 来读取此文件:

```

>>> f = open("you.md")
>>> content = f.readlines()
>>> content
['You Raise Me Up\n', 'When I am down and, oh my soul, so weary;\n', 'When troubles come and my heart burdened be;\n']

```

返回的是一个列表, 列表中每个元素都是一个字符串, 每个字符串中的内容就是文件的一行文字, 含行末的符号。显而易见, 它是可以用 `for` 来循环的。

```

>>> for line in content:
...     print line ,
...
You Raise Me Up
When I am down and, oh my soul, so weary;
When troubles come and my heart burdened be;
Then, I am still and wait here in the silence,
Until you come and sit awhile with me.
You raise me up, so I can stand on mountains;

```

```
You raise me up, to walk on stormy seas;
I am strong, when I am on your shoulders;
You raise me up: To more than I can be.
>>> f.close()
```

读很大的文件

前面已经说明了，如果文件太大，就不能用 `read()` 或者 `readlines()` 一次性将全部内容读入内存，可以使用 `while` 循环和 `readline()` 来完成这个任务。

此外，还有一个方法：`fileinput` 模块

```
>>> import fileinput
>>> for line in fileinput.input("you.md"):
...     print line,
...
You Raise Me Up
When I am down and, oh my soul, so weary;
When troubles come and my heart burdened be;
Then, I am still and wait here in the silence,
Until you come and sit awhile with me.
You raise me up, so I can stand on mountains;
You raise me up, to walk on stormy seas;
I am strong, when I am on your shoulders;
You raise me up: To more than I can be.
```

我比较喜欢这个，用起来是那么得心应手，简洁明快，还用 `for`。

对于这个模块的更多内容，读者可以自己在交互模式下利用 `dir()`，`help()` 去查看明白。

还有一种方法，更为常用：

```
>>> for line in f:
...     print line ,
...
You Raise Me Up
When I am down and, oh my soul, so weary;
When troubles come and my heart burdened be;
Then, I am still and wait here in the silence,
Until you come and sit awhile with me.
You raise me up, so I can stand on mountains;
You raise me up, to walk on stormy seas;
I am strong, when I am on your shoulders;
You raise me up: To more than I can be.
```

之所以能够如此，是因为 file 是可迭代的数据类型，直接用 for 来迭代即可。

seek

这个函数的功能就是让指针移动。特别注意，它是以字节为单位进行移动的。比如：

```
>>> f = open("you.md")
>>> f.readline()
'You Raise Me Up\n'
>>> f.readline()
'When I am down and, oh my soul, so weary;\n'
```

现在已经移动到第四行末尾了，看 seek() 的能力：

```
>>> f.seek(0)
```

意图是要回到文件的最开头，那么如果用 f.readline() 应该读取第一行。

```
>>> f.readline()
'You Raise Me Up\n'
```

果然如此。此时指针所在的位置，还可以用 tell() 来显示，如

```
>>> f.tell()
17L
>>> f.seek(4)
```

f.seek(4) 就将位置定位到从开头算起的第四个字符后面，也就是"You "之后，字母"R"之前的位置。

```
>>> f.tell()
4L
```

tell() 也是这么说的。这时候如果使用 readline()，得到就是从当前位置开始到行末。

```
>>> f.readline()
'Raise Me Up\n'
>>> f.close()
```

seek() 还有别的参数，具体如下：

seek(...) seek(offset[, whence]) -> None. Move to new file position.

Argument offset is a byte count. Optional argument whence defaults to 0 (offset from start of file, offset should be ≥ 0); other values are 1 (move relative to current position, positive or negative), and 2 (move relative to end of file, usually negative, although many platforms allow seeking beyond the en

d of a file). If the file is opened in text mode, only offsets returned by tell() are legal. Use of other offsets causes undefined behavior. Note that not all file objects are seekable.

whence 的值：

- 默认值是 0，表示从文件开头开始计算指针偏移的量（简称偏移量）。这是 offset 必须是大于等于 0 的整数。
 - 是 1 时，表示从当前位置开始计算偏移量。offset 如果是负数，表示从当前位置向前移动，整数表示向后移动。
 - 是 2 时，表示相对文件末尾移动。
-

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

迭代

跟一些比较牛 X 的程序员交流，经常听到他们嘴里冒出一个不标准的英文单词，而 loop、iterate、traversal 和 recursion 如果不在其内，总觉得他还不够牛 X。当然，真正牛 X 的绝对不会这么说的，他们只是说“循环、迭代、遍历、递归”，然后再问“这个你懂吗？”哦，这就是真正牛 X 的程序员。不过，他也仅仅是牛 X 罢了，还不是大神。大神程序员是什么样儿呢？他是扫地僧，大隐隐于市。

先搞清楚这些名词再说别的：

- 循环（loop），指的是在满足条件的情况下，重复执行同一段代码。比如，while 语句。
- 迭代（iterate），指的是按照某种顺序逐个访问列表中的每一项。比如，for 语句。
- 递归（recursion），指的是一个函数不断调用自身的行为。比如，以编程方式输出著名的斐波纳契数列。
- 遍历（traversal），指的是按照一定的规则访问树形结构中的每个节点，而且每个节点都只访问一次。

对于这四个听起来高深莫测的词汇，其实前面，已经涉及到了一个——循环（loop），本节主要介绍一下迭代（iterate），看官在网上 google，就会发现，对于迭代和循环、递归之间的比较的文章不少，分别从不同角度将它们进行了对比。这里暂不比较，先搞明白 python 中的迭代。

当然，迭代的话题如果说起来，会很长，本着循序渐进的原则，这里介绍比较初级的。

逐个访问

在 python 中，访问对象中每个元素，可以这么做：（例如一个 list）

```
>>> lst
['q', 'i', 'w', 's', 'i', 'r']
>>> for i in lst:
...     print i,
...
q i w s i r
```

除了这种方法，还可以这样：

```
>>> lst_iter = iter(lst) #对原来的 list 实施了一个 iter()
>>> lst_iter.next()    #要不厌其烦地一个一个手动访问
'q'
>>> lst_iter.next()
'i'
>>> lst_iter.next()
```

```
'w'
>>> lst_iter.next()
's'
>>> lst_iter.next()
'i'
>>> lst_iter.next()
'r'
>>> lst_iter.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

`iter()` 是一个内建函数，其含义是：

上面的 `next()` 就是要获得下一个元素，但是做为一名优秀的程序员，最佳品质就是“懒惰”，当然不能这样一个个地敲啦，于是就：

```
>>> while True:
...     print lst_iter.next()
...
Traceback (most recent call last):      #居然报错，而且错误跟前面一样？什么原因
  File "<stdin>", line 2, in <module>
StopIteration
```

先不管错误，再来一遍。

```
>>> lst_iter = iter(lst)          #上面的错误暂且搁置，回头在研究
>>> while True:
...     print lst_iter.next()
...
q                      #果然自动化地读取了
i
w
s
i
r
Traceback (most recent call last):      #读取到最后一个之后，报错，停止循环
  File "<stdin>", line 2, in <module>
StopIteration
```

首先了解一下上面用到的那个内置函数：`iter()`，官方文档中有这样一段话描述之：

`iter(o[, sentinel])`

Return an iterator object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, o must be a collection object which supports

s the iteration protocol (the `iter()` method), or it must support the sequence protocol (the `getitem()` method with integer arguments starting at 0). If it does not support either of those protocols, `TypeError` is raised. If the second argument, `sentinel`, is given, then `o` must be a callable object. The iterator created in this case will call `o` with no arguments for each call to its `next()` method; if the value returned is equal to `sentinel`, `StopIteration` will be raised, otherwise the value will be returned.

大意是说...(此处故意省略若干字, 因为我相信看此文章的看官英语水平是达到看文档的水平了, 如果没有, 也不用着急, 找个词典什么的帮助一下。)

尽管不翻译了, 但是还要提炼一下主要的东西:

- 返回值是一个迭代器对象
- 参数需要是一个符合迭代协议的对象或者是一个序列对象
- `next()` 配合与之使用

什么是“可迭代的对象”呢? 在前面学习的时候, 曾经提到过, 如果忘记了请往前翻阅。

一般, 我们常常将哪些能够用诸如循环语句之类的方法来一个一个读取元素的对象, 就称之为可迭代的对象。那么用来循环的如 `for` 就被称之为迭代工具。

用严格点的语言说: 所谓迭代工具, 就是能够按照一定顺序扫描迭代对象的每个元素(按照从左到右的顺序)。

显然, 除了 `for` 之外, 还有别的可以称作迭代工具。

那么, 刚才介绍的 `iter()` 的功能呢? 它与 `next()` 配合使用, 也是实现上述迭代工具的作用。

在 Python 中, 甚至在其它的语言中, 迭代这块的说法比较乱, 主要是名词乱, 刚才我们说, 那些能够实现迭代的东西, 称之为迭代工具, 就是这些迭代工具, 不少程序员都喜欢叫做迭代器。当然, 这都是汉语翻译, 英语就是 `iterator`。

看官看上面的所有例子会发现, 如果用 `for` 来迭代, 当到末尾的时候, 就自动结束了, 不会报错。如果用 `iter()...next()` 迭代, 当最后一个完成之后, 它不会自动结束, 还要向下继续, 但是后面没有元素了, 于是就报一个称之为 `StopIteration` 的错误(这个错误的名字叫做: 停止迭代, 这哪里是报错, 分明是警告)。

看官还要关注 `iter()...next()` 迭代的一个特点。当迭代对象 `lst_iter` 被迭代结束, 即每个元素都读取了一遍之后, 指针就移动到了最后一个元素的后面。如果再访问, 指针并没有自动返回到首位置, 而是仍然停留在末位置, 所以报 `StopIteration`, 想要再开始, 需要重新载入迭代对象。所以, 当我在上面重新进行迭代对象赋值之后, 又可以继续了。这在 `for` 等类型的迭代工具中是没有的。

文件迭代器

现在有一个文件，名称：208.txt，其内容如下：

```
Learn python with qiwsir.  
There is free python course.  
The website is:  
http://qiwsir.github.io  
Its language is Chinese.
```

用迭代器来操作这个文件，我们在前面讲述文件有关知识的时候已经做过了，无非就是：

```
>>> f = open("208.txt")  
>>> f.readline()      #读第一行  
'Learn python with qiwsir.\n'  
>>> f.readline()      #读第二行  
'There is free python course.\n'  
>>> f.readline()      #读第三行  
'The website is:\n'  
>>> f.readline()      #读第四行  
'http://qiwsir.github.io\n'  
>>> f.readline()      #读第五行，也就是这真在读完最后一行之后，到了此行的后面  
'Its language is Chinese.\n'  
>>> f.readline()      #无内容了，但是不报错，返回空。  
"
```

以上演示的是用 `readline()` 一行一行地读。当然，在实际操作中，我们是绝对不能这样做的，一定要让它自动进行，比较常用的方法是：

```
>>> for line in f:    #这个操作是紧接着上面的操作进行的，请看官主要观察  
...   print line,    #没有打印出任何东西  
...
```

这段代码之所没有打印出东西来，是因为经过前面的迭代，指针已经移到了最后了。这就是迭代的一个特点，要小心指针的位置。

```
>>> f = open("208.txt")  #从头再来  
>>> for line in f:  
...   print line,  
...  
Learn python with qiwsir.  
There is free python course.  
The website is:
```

```
http://qiwsir.github.io
Its language is Chinese.
```

这种方法是读取文件常用的。另外一个 `readlines()` 也可以。但是，需要有一些小心的地方，看官如果想不起来小心什么，可以在将关于文件的课程复习一边。

上面过程用 `next()` 也能够读取。

```
>>> f = open("208.txt")
>>> f.next()
'Learn python with qiwsir.\n'
>>> f.next()
'There is free python course.\n'
>>> f.next()
'The website is:\n'
>>> f.next()
'http://qiwsir.github.io\n'
>>> f.next()
'It's language is Chinese.\n'
>>> f.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

如果用 `next()`，就可以直接读取每行的内容。这说明文件是天然的可迭代对象，不需要用 `iter()` 转换了。

再有，我们用 `for` 来实现迭代，在本质上，就是自动调用 `next()`，只不过这个工作，已经让 `for` 偷偷地替我们干了，到这里，列位是不是应该给 `for` 取另外一个名字：它叫雷锋。

还有，列表解析也能够做为迭代工具，在研究列表的时候，看官想必已经清楚了。那么对文件，是否可以用？试一试：

```
>>> [ line for line in open('208.txt') ]
['Learn python with qiwsir.\n', 'There is free python course.\n', 'The website is:\n', 'http://qiwsir.github.io\n', "It's language is Chinese.\n"]
```

至此，看官难道还不为列表解析所折服吗？真的很强大，又强又大呀。

其实，迭代器远远不止上述这么简单，下面我们随便列举一些，在 Python 中还可以这样得到迭代对象中的元素。

```
>>> list(open('208.txt'))
['Learn python with qiwsir.\n', 'There is free python course.\n', 'The website is:\n', 'http://qiwsir.github.io\n', "It's language is Chinese.\n"]

>>> tuple(open('208.txt'))
('Learn python with qiwsir.\n', 'There is free python course.\n', 'The website is:\n', 'http://qiwsir.github.io\n', "It's language is Chinese.\n")
```

```
>>> "$$".join(open('208.txt'))
'Learn python with qiwsir.\n$$$There is free python course.\n$$$The website is:\n$$http://qiwsir.github.io\n$$$Its language is Chinese.\n'

>>> a,b,c,d,e = open("208.txt")
>>> a
'Learn python with qiwsir.\n'
>>> b
'There is free python course.\n'
>>> c
'The website is:\n'
>>> d
'http://qiwsir.github.io\n'
>>> e
'Its language is Chinese.\n'
```

上述方式，在编程实践中不一定用得上，只是向看官展示一下，并且看官要明白，可以这么做，不是非要这么做。

补充一下，字典也可以迭代，看官自己不妨摸索一下（其实前面已经用 for 迭代过了，这次请摸索一下用 iter()...next() 手动一步一步迭代）。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

练习

已经将 Python 的基础知识学习完毕，包含基本的数据类型（或者说对象类型）和语句。利用这些，加上个人的聪明才智，就能解决一些问题了。

练习 1

问题描述

有一个列表，其中包括 10 个元素，例如这个列表是[1,2,3,4,5,6,7,8,9,0]，要求将列表中的每个元素一次向前移动一个位置，第一个元素到列表的最后，然后输出这个列表。最终样式是[2,3,4,5,6,7,8,9,0,1]

解析

或许刚看题目的读者，立刻想到把列表中的第一个元素拿出来，然后追加到最后，不就可以了？是的。就是这么简单。主要是联系一下已经学习过的列表操作。

看下面代码之前，不妨自己写一写试试。然后再跟我写的对照。

注意，我在这里所写的代码不能算标准答案。只能是参考。很可能你写的比我写的还要好。在代码界，没有标准答案。

参考代码如下，这个我保存为 12901.py 文件

```
#!/usr/bin/env python
# coding=utf-8

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

b = raw.pop(0)
raw.append(b)
print raw
```

执行这个文件：

```
$ python 12901.py
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
[2, 3, 4, 5, 6, 7, 8, 9, 0, 1]
```

第一行所打印的是原来的列表，第二行是需要的列表。这里用到的主要时列表的两个函数 `pop()` 和 `append()`。如果读者感觉不是很熟悉，或者对这个问题，在我提供的参考之前只有一个模糊认识，但是没有明晰地写出代码，说明对前面的函数还没有烂熟于胸。唯一的方法就是多练习。

练习 2

问题描述

按照下面的要求实现对列表的操作：

1. 产生一个列表，其中有 40 个元素，每个元素是 0 到 100 的一个随机整数
2. 如果这个列表中的数据代表着某个班级 40 人的分数，请计算成绩低于平均分的学生人数，并输出
3. 对上面的列表元素从大到小排序

解析

这个问题中，需要几个知识点：

第一个是随机产生整数。一种方法是你做 100 个纸片，分别写上 1 到 100 的数字（每张上一个整数），然后放到一个盒子里面。抓出一个，看是几，就将这个数字写到列表中，直到抓出第 40 个。这样得到的列表是随机了。但是，好像没有 Python 什么事情。那么久要用另外一种方法，让 Python 来做。Python 中有一个模块：`random`，专门提供随机事件的。

```
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom', 'TWOPI', 'Wichman'
```

在这个问题中，只需要 `random.randint()`，专门获取某个范围内的随机整数。

第二个是求平均数，方法是将所有数字求和，然后除以总人数（40）。求和方法就是 `sum()` 函数。在计算平均数的时候，要注意，一般平均数不能仅仅是整数，最好保留一位小数吧。这是除法中的知识了。

第三个是列表排序。

下面就依次展开。不忙，在我开始之前，你先试试吧。

```
#!/usr/bin/env Python
# coding=utf-8

from __future__ import division
import random

score = [random.randint(0,100) for i in range(40)] #0 到 100 之间，随机得到 40 个整数，组成列表
```

```

print score

num = len(score)
sum_score = sum(score)          #对列表中的整数求和
ave_num = sum_score/num         #计算平均数
less_ave = len([i for i in score if i<ave_num])  #将小于平均数的找出来，组成新的列表，并度量该列表的长度
print "the average score is:%.1f" % ave_num
print "There are %d students less than average." % less_ave

sorted_score = sorted(score, reverse=True)  #对原列表排序
print sorted_score

```

练习 3

问题描述

如果将一句话作为一个字符串，那么这个字符串中必然会有空格（这里仅讨论英文），比如"How are you."，但有的时候，会在两个单词之间多大一个空格。现在的任务是，如果一个字符串中有连续的两个空格，请把它删除。

解析

对于一个字符串中有空格，可以使用[《字符串\(4\)》](#)中提到的 `strip()` 等。但是，它不是仅仅去掉一个空格，而是把字符串两遍的空格都去掉。都去掉似乎也没有什么关系，再用空格把单词拼起来就好了。

按照这个思路，我这样写代码，供你参考（更建议你先写出一段来，然后我们两个对照）。

```

#!/usr/bin/env Python
# coding=utf-8

string = "I love code."  #在 code 前面有两个空格，应该删除一个
print string            #为了能够清楚看到每步的结果，把过程中的量打印出来

str_lst = string.split(" ")  #以空格为分割，得到词汇的列表
print str_lst

words = [s.strip() for s in str_lst]  #去除单词两边的空格
print words

new_string = " ".join(words)  #以空格为连接符，将单词链接起来
print new_string

```

保存之后，运行这个代码，结果是：

```
I love code.  
['I', 'love', ' ', 'code.'][  
'I', 'love', ' ', 'code.'][  
I love code.
```

结果是令人失望的。经过一番折腾，空格根本就没有被消除。最后的输出和一开始的字符串完全一样。泪奔！

查找原因。

从输出中已经清楚表示了。当执行 `string.split(" ")` 的时候，是以空格为分割符，将字符串分割，并返回列表。列表中元素是由单词组成。原来字符串中单词之间的空格已经被作为分隔符，那么列表中单词两遍就没有空格了。所以，前面代码中就无需在用 `strip()` 去删除空格。另外，特别要注意的是，有两个空格连着呢，其中一个空格作为分隔符，另外一个空格就作为列表元素被返回了。这样一来，分割之后的操作都无作用了。

看官是否明白错误原因了？

如何修改？显然是分割之后，不能用 `strip()`，而是要想办法把那个返回列表中的空格去掉，得到只含有单词的列表。再用空格连接之，就应该对了。所以，我这样修正它。

```
#!/usr/bin/env Python  
# coding=utf-8  
  
string = "I love code."  
print string  
  
str_lst = string.split(" ")  
print str_lst  
  
words = [s for s in str_lst if s!=""] #利用列表解析，将空格检出  
print words  
  
new_string = " ".join(words)  
print new_string
```

将文件保存，名为 12903.py，运行之得到下面结果：

```
I love code.  
['I', 'love', ' ', 'code.'][  
'I', 'love', 'code.'][  
I love code.
```

OK！完美地解决了问题，去除了 code 前面的一个空格。

练习 4

问题描述

根剧高德纳 (Donald Ervin Knuth) 的《计算机程序设计艺术》(The Art of Computer Programming) , 150 年印度数学家 Gopala 和金月在研究箱子包装物件长宽刚好为 1 和 2 的可行方法数目时, 首先描述这个数列。在西方, 最先研究这个数列的人是比萨的李奥纳多 (意大利人斐波那契 Leonardo Fibonacci) , 他描述兔子生長的数目時用上了这数列。

第一个月初有一对刚诞生的兔子;第二个月之后 (第三个月初) 他们可以生育,每月每对可生育的兔子会诞生下一对新兔子;兔子永不死去

假设计 n 月有可生育的兔子总共 a 对, $n+1$ 月就总共有 b 对。在 $n+2$ 月必定总共有 $a+b$ 对: 因为在 $n+2$ 月的时候, 前一月 ($n+1$ 月) 的 b 对兔子可以存留至第 $n+2$ 月 (在当月属于新诞生的兔子尚不能生育) 。而新生出的兔子對數等于所有在 n 月就已存在的 a 对

上面故事是一个著名的数列——斐波那契数列——的起源。斐波那契数列用数学方式表示就是:

$$\begin{aligned} a_0 &= 0 & (n=0) \\ a_1 &= 1 & (n=1) \\ a[n] &= a[n-1] + a[n-2] & (n \geq 2) \end{aligned}$$

我们要做的事情是用程序计算出 $n=100$ 是的值。

在解决这个问题之前, 你可以先观看一个[关于斐波那契数列数列的视频](#), 注意, 请在墙内欣赏。

解析

斐波那契数列是各种编程语言中都要秀一下的东西, 通常用在阐述“递归”中。什么是递归? 后面的 Python 中也会讲到。不过, 在这里不准备讲。

其实, 如果用递归来写, 会更容易明白。但是, 这里我给出一个用 for 循环写的, 看看是否能够理解之。

```
#!/usr/bin/env Python
# coding=utf-8

a, b = 0, 1

for i in range(4):  #改变这里的数, 就能得到相应项的结果
    a, b = b, a+b

print a
```

保存运行之，看看结果和你推算的是否一致。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

自省

特别说明，这一讲的内容不是我写的，是我从《Python自省指南》抄录过来的，当然，为了适合本教程，我在某些地方做了修改或者重写。

什么是自省？

在日常生活中，自省（introspection）是一种自我检查行为。自省是指对某人自身思想、情绪、动机和行为的检查。伟大的哲学家苏格拉底将生命中的大部分时间用于自我检查，并鼓励他的雅典朋友们也这样做。他甚至对自己作出了这样的要求：“未经自省的生命不值得存在。”无独有偶，在中国《论语》中，也有这样的名言：“吾日三省吾身”。显然，自省对个人成长多么重要呀。

在计算机编程中，自省是指这种能力：检查某些事物以确定它是什么、它知道什么以及它能做什么。自省向程序员提供了极大的灵活性和控制力。一旦您使用了支持自省的编程语言，就会产生类似这样的感觉：“未经检查的对象不值得实例化。”

整个 Python 语言对自省提供了深入而广泛的支持。实际上，很难想象假如 Python 语言没有其自省特性是什么样子。

学完这节，你就能够轻松洞察到 Python 对象的“灵魂”。

在深入研究更高级的技术之前，我们尽可能用最普通的方式来研究 Python 自省。有些读者甚至可能会争论说：我们开始时所讨论的特性不应称之为“自省”。我们必须承认，它们是否属于自省的范畴还有待讨论。但从本节的意图出发，我们所关心的只是找出有趣问题的答案。

现在让我们以交互方式使用 Python 来开始研究。这是前面已经在使用的一种方式。

联机帮助

在交互模式下，用 help 向 Python 请求帮助。

```
>>> help()
```

```
Welcome to Python 2.7! This is the online help utility.
```

```
If this is your first time using Python, you should definitely check out  
the tutorial on the Internet at http://docs.python.org/2.7/tutorial/.
```

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules", "keywords", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose summaries contain a given word such as "spam", type "modules spam".

```
help>
```

这时候就进入了联机帮助状态，根据提示输入 `keywords`

```
help> keywords
```

Here is a list of the Python keywords. Enter any keyword to get more help.

and	elif	if	print
as	else	import	raise
assert	except	in	return
break	exec	is	try
class	finally	lambda	while
continue	for	not	with
def	from	or	yield
del	global	pass	

现在显示出了 Python 关键词的列表。依照说明亦步亦趋，输入每个关键词，就能看到那个关键词的相关文档。这里就不展示输入的结果了。读者可以自行尝试。要记住，如果从文档说明界面返回到帮助界面，需要按 `q` 键。

这样，我们能够得到联机帮助。从联机帮助状态退回到 Python 的交互模式，使用 `quit` 命令。

```
help> quit
```

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)". Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>>

联机帮助实用程序会显示关于各种主题或特定对象的信息。

帮助实用程序很有用，并确实利用了 Python 的自省能力。但仅仅使用帮助不会揭示帮助是如何获得其信息的。而且，因为我们的目的是揭示 Python 自省的所有秘密，所以我们必须迅速地跳出对帮助实用程序的讨论。

在结束关于帮助的讨论之前，让我们用它来获得一个可用模块的列表。

模块只是包含 Python 代码的文本文件，其名称后缀是 .py，关于模块，本教程会在后面有专门的讲解。如果在 Python 提示符下输入 `help('modules')`，或在 help 提示符下输入 `modules`，则会看到一长列可用模块，类似于下面所示的部分列表。自己尝试它以观察您的系统中有哪些可用模块，并了解为什么会认为 Python 是“自带电池”的（自带电池，这是一个比喻，就是说 Python 在被安装时，就带了很多模块，这些模块是你以后开发中会用到的，比喻成电池，好比开发的助力工具），或者说是 Python 一被安装，就已经包含有的模块，不用我们费力再安装了。

```
>>> help("modules")

Please wait a moment while I gather a list of all available modules...
ANSI          _threading_local  gnomekeyring    repr
BaseHTTPServer _warnings      gobject        requests
MySQLdb        chardet       lsb_release    sre_parse
.....(此处省略一些)
PyQt4         codeop        markupbase    stringprep
Queue          collections   marshal       strop
ScrolledText   colorama     math         struct
.....(省略其它的模块)
Enter any module name to get more help. Or, type "modules spam" to search
for modules whose descriptions contain the word "spam".
```

因为太多，无法全部显示。你可以仔细观察一下，是不是有我们前面已经用过的那个 `math`、`random` 模块呢？

如果是在 Python 交互模式 `>>>` 下，比如要得到有关 `math` 模块的更多帮助，可以输入 `>>> help("math")`，如果是在帮助模式 `help>` 下，直接输入 `>math` 就能得到关于 `math` 模块的详细信息。简直太贴心了。

dir()

尽管查找和导入模块相对容易，但要记住每个模块包含什么却不是这么简单。你或许并不希望总是必须查看源代码来找出答案。幸运的是，Python 提供了一种方法，可以使用内置的 `dir()` 函数来检查模块（以及其它对象）的内容。

其实，这个东西我们已经一直在使用。

`dir()` 函数可能是 Python 自省机制中最著名的部分了。它返回传递给它的任何对象的属性名称经过排序的列表。如果不指定对象，则 `dir()` 返回当前作用域中（这里冒出来一个新名词：“作用域”，暂且不用管它，后面会详解，你就姑且理解为某个范围吧）的名称。让我们将 `dir()` 函数应用于 `keyword` 模块，并观察它揭示了什么：

```
>>> import keyword
>>> dir(keyword)
['__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__', 'iskeyword', 'kwlist', 'main']
```

如果不带任何参数，则 `dir()` 返回当前作用域中的名称。请注意，因为我们先前导入了 `keyword`，所以它们出现在列表中。导入模块将把该模块的名称添加到当前作用域：

```
>>> dir()
['GFileDescriptorBased', 'GInitiallyUnowned', 'GPollableInputStream', 'GPollableOutputStream', '__builtins__', '__doc__',
>>> import math
>>> dir()
['GFileDescriptorBased', 'GInitiallyUnowned', 'GPollableInputStream', 'GPollableOutputStream', '__builtins__', '__doc__', ...]
```

`dir()` 函数是内置函数，这意味着我们不必为了使用该函数而导入模块。不必做任何操作，Python 就可识别内置函数。

再观察，看到调用 `dir()` 后返回了这个名称 `__builtins__`。也许此处有连接。让我们在 Python 提示符下输入名称 `__builtins__`，并观察 Python 是否会告诉我们关于它的任何有趣的事情：

```
>>> __builtins__
<module '__builtin__' (built-in)>
```

因此 `__builtins__` 看起来象是当前作用域中绑定到名为 `__builtin__` 的模块对象的名称。（因为模块不是只有多个单一值的简单对象，所以 Python 改在尖括号中显示关于模块的信息。）

注：如果您在磁盘上寻找 `__builtin__.py` 文件，将空手而归。这个特殊的模块对象是 Python 解释器凭空创建的，因为它包含着解释器始终可用的项。尽管看不到物理文件，但我们仍可以将 `dir()` 函数应用于这个对象，以观察所有内置函数、错误对象以及它所包含的几个杂项属性。

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BufferError', 'BytesWarning', 'DeprecationWarning', 'E...']
```

`dir()` 函数适用于所有对象类型，包括字符串、整数、列表、元组、字典、函数、定制类、类实例和类方法（不理解的对象类型，会在随后的教程中讲解）。例如将 `dir()` 应用于字符串对象，如您所见，即使简单的 Python 字符串也有许多属性（这是前面已经知道的了，权当复习）

```
>>> dir("You raise me up")
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__...']
```

读者可以尝试一下其它的对象类型，观察返回结果，如：`dir(42)`，`dir([])`，`dir(()`，`dir({})`，`dir(dir)``。

文档字符串

在许多 dir() 示例中，您可能会注意到的一个属性是 `__doc__` 属性。这个属性是一个字符串，它包含了描述对象的注释。Python 称之为文档字符串或 docstring（这个内容，会在下一部分中讲解如何自定义设置）。

如果模块、类、方法或函数定义的第一条语句是字符串，那么该字符串会作为对象的 `__doc__` 属性与该对象关联起来。例如，看一下 str 类型对象的文档字符串。因为文档字符串通常包含嵌入的换行 \n，我们将使用 Python 的 print 语句，以便输出更易于阅读：

```
>>> print str.__doc__
str(object='') -> string

Return a nice string representation of the object.
If the argument is a string, the return value is the same object.
```

检查 Python 对象

前面已经好几次提到了“对象（object）”这个词，但一直没有真正定义它。编程环境中的对象很象现实世界中的对象。实际的对象有一定的形状、大小、重量和其它特征。实际的对象还能够对其环境进行响应、与其它对象交互或执行任务。计算机中的对象试图模拟我们身边现实世界中的对象，包括象文档、日程表和业务过程这样的抽象对象。

其实，我总觉得把 object 翻译成对象，让人感觉很没有具象的感觉，因为在汉语里面，对象是一个很笼统的词汇。另外一种翻译，流行于台湾，把它称为“物件”，倒是挺不错的理解。当然，名词就不纠缠了，关键是理解内涵。关于面向对象编程，可以阅读维基百科的介绍——[面向对象程序设计](#)——先了解大概。

类似于实际的对象，几个计算机对象可能共享共同的特征，同时保持它们自己相对较小的变异特征。想一想您在书店中看到的书籍。书籍的每个物理副本都可能有污迹、几张破损的书页或唯一的标识号。尽管每本书都是唯一的对象，但都拥有相同标题的每本书都只是原始模板的实例，并保留了原始模板的大多数特征。

对于面向对象的类和类实例也是如此。例如，可以看到每个 Python 符串都被赋予了一些属性，dir() 函数揭示了这些属性。

于是在计算机术语中，对象是拥有标识和值的事物，属于特定类型、具有特定特征和以特定方式执行操作。并且，对象从一个或多个父类继承了它们的许多属性。除了关键字和特殊符号（象运算符，如 +、-、*、**、/、%、<、> 等）外，Python 中的所有东西都是对象。Python 具有一组丰富的对象类型：字符串、整数、浮点、列表、元组、字典、函数、类、类实例、模块、文件等。

当您有一个任意的对象（也许是一个作为参数传递给函数的对象）时，可能希望知道一些关于该对象的情况。如希望 Python 告诉我们：

- 对象的名称是什么？
- 这是哪种类型的对象？
- 对象知道些什么？
- 对象能做些什么？
- 对象的父对象是谁？

名称

并非所有对象都有名称，但那些有名称的对象都将名称存储在其 `__name__` 属性中。注：名称是从对象而不是引用该对象的变量中派生的。

```
>>> dir() #dir() 函数
['GFileDescriptorBased', 'GInitiallyUnowned', 'GPollableInputStream', 'GPollableOutputStream', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
>>> directory = dir #新变量
>>> directory() #跟 dir() 一样的结果
['GFileDescriptorBased', 'GInitiallyUnowned', 'GPollableInputStream', 'GPollableOutputStream', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
>>> dir.__name__ #dir() 的名字
'dir'
>>> directory.__name__
'dir'

>>> __name__ #这是不一样的
'__main__'
```

模块拥有名称，Python 解释器本身被认为是顶级模块或主模块。当以交互的方式运行 Python 时，局部 `__name__` 变量被赋予值 `'__main__'`。同样地，当从命令行执行 Python 模块，而不是将其导入另一个模块时，其 `__name__` 属性被赋予值 `'__main__'`，而不是该模块的实际名称。这样，模块可以查看其自身的 `__name__` 值来自行确定它们自己正被如何使用，是作为另一个程序的支持，还是作为从命令行执行的主应用程序。因此，下面这条惯用的语句在 Python 模块中是很常见的：

```
if __name__ == '__main__':
    # Do something appropriate here, like calling a
    # main() function defined elsewhere in this module.
    main()
else:
    # Do nothing. This module has been imported by another
```

```
# module that wants to make use of the functions,
# classes and other useful bits it has defined.
```

类型

`type()` 函数有助于我们确定对象是字符串还是整数，或是其它类型的对象。它通过返回类型对象来做到这一点，可以将这个类型对象与 `types` 模块中定义的类型相比较：

```
>>> import types
>>> print types.__doc__
Define names for all type symbols known in the standard interpreter.

Types that are part of optional modules (e.g. array) are not listed.

>>> dir(types)
['BooleanType', 'BufferType', 'BuiltinFunctionType', 'BuiltinMethodType', 'ClassType', 'CodeType', 'ComplexType', 'DictPro...
>>> p = "I love Python"
>>> type(p)
<type 'str'>
>>> if type(p) is types.StringType:
...     print "p is a string"
...
p is a string
>>> type(42)
<type 'int'>
>>> type([])
<type 'list'>
>>> type({})
<type 'dict'>
>>> type(dir)
<type 'builtin_function_or_method'>
```

标识

先前说过，每个对象都有标识、类型和值。值得注意的是，可能有多个变量引用同一对象，同样地，变量可以引用看起来相似（有相同的类型和值），但拥有截然不同标识的多个对象。当更改对象时（如将某一项添加到列表），这种关于对象标识的概念尤其重要，如在下面的示例中，`blist` 和 `cplist` 变量引用同一个列表对象。正如您在示例中所见，`id()` 函数给任何给定对象返回唯一的标识符。其实，这个东东我们在前面已经使用过了。在这里再次提出，能够让你理解上有提升吧。

```
>>> print id.__doc__
id(object) -> integer
```

Return the identity of an object. This is guaranteed to be unique among simultaneously existing objects. (Hint: it's the object's memory address.)

```
>>> alist = [1,2,3]
>>> blist = [1,2,3]
>>> clist = blist
>>> id(alist)
2979691052L
>>> id(blist)
2993911916L
>>> id(clist)
2993911916L
>>> alist is blist
False
>>> blist is clist
True
>>> clist.append(4)
>>> clist
[1, 2, 3, 4]
>>> blist
[1, 2, 3, 4]
>>> alist
[1, 2, 3]
```

如果对上面的操作还有疑惑，可以回到前面复习有关深拷贝和浅拷贝的知识。

属性

对象拥有属性，并且 `dir()` 函数会返回这些属性的列表。但是，有时我们只想测试一个或多个属性是否存在。如果对象具有我们正在考虑的属性，那么通常希望只检索该属性。这个任务可以由 `hasattr()` 和 `getattr()` 函数来完成。

```
>>> print hasattr.__doc__
hasattr(object, name) -> bool
```

Return whether the object has an attribute with the given name.
(This is done by calling `getattr(object, name)` and catching exceptions.)

```
>>> print getattr.__doc__
getattr(object, name[, default]) -> value
```

Get a named attribute from an object; `getattr(x, 'y')` is equivalent to `x.y`.
When a default argument is given, it is returned when the attribute doesn't exist; without it, an exception is raised in that case.

```
>>>
>>> hasattr(id, '__doc__')
True

>>> print getattr(id, '__doc__')
id(object) -> integer
```

Return the identity of an object. This is guaranteed to be unique among simultaneously existing objects. (Hint: it's the object's memory address.)

可调用

可以调用表示潜在行为（函数和方法）的对象。可以用 callable() 函数测试对象的可调用性：

```
>>> print callable.__doc__
callable(object) -> bool

Return whether the object is callable (i.e., some kind of function).
Note that classes are callable, as are instances with a __call__() method.
>>> callable("a string")
False
>>> callable(dir)
True
```

实例

这个名词还很陌生，没关系，先看看，混个脸熟，以后会经常用到。

在 type() 函数提供对象的类型时，还可以使用 isinstance() 函数测试对象，以确定它是否是某个特定类型或定制类的实例：

```
>>> print isinstance.__doc__
isinstance(object, class-or-type-or-tuple) -> bool

Return whether an object is an instance of a class or of a subclass thereof.
With a type as second argument, return whether that is the object's type.
The form using a tuple, isinstance(x, (A, B, ...)), is a shortcut for
isinstance(x, A) or isinstance(x, B) or ... (etc.).
>>> isinstance(42, str)
False
>>> isinstance("python", str)
True
```

子类

关于类的问题，有一个“继承”概念，有继承就有父子问题，这是在现实生活中很正常的，在编程语言中也是如此。虽然这是后面要说的，但是，为了本讲内容的完整，也姑且把这个内容放在这里。读者可以不看，留着以后看也行。我更建议还是阅读一下，有个印象。

在类这一级别，可以根据一个类来定义另一个类，同样地，这个新类会按照层次化的方式继承属性。Python 甚至支持多重继承，多重继承意味着可以用多个父类来定义一个类，这个新类继承了多个父类。`issubclass()` 函数使我们可以查看一个类是不是继承了另一个类：

```
>>> print issubclass.__doc__
issubclass(C, B) -> Boolean
Return whether class C is a subclass (i.e., a derived class) of class B.
>>> class SuperHero(Person): # SuperHero inherits from Person...
...     def intro(self):    # but with a new SuperHero intro
...         """Return an introduction."""
...         return "Hello, I'm SuperHero %s and I'm %s." % (self.name, self.age)
...
>>> issubclass(SuperHero, Person)
1
>>> issubclass(Person, SuperHero)
0
```

Python 文档

文档，这个词语在经常在程序员的嘴里冒出来，有时候他们还经常以文档有没有或者全不全为标准来衡量一个软件项目是否高大上。那么，软件中的文档是什么呢？有什么要求呢？Python 文档又是什么呢？文档有什么用呢？

文档很重要。独孤九剑的剑诀、易筋经的心法、写着辟邪剑谱的袈裟，这些都是文档。连那些大牛人都要这些文档，更何况我们呢？所以，文档是很重要的。

文档，说白了就是用 word（这个最多了）等（注意这里的等，把不常用的工具都等掉了，包括我编辑文本时用的 vim 工具）文本编写工具写成的包含文本内容但不限于文字的文件。有点啰嗦，啰嗦的目的是为了严谨，呵呵。最好还是来一个更让人信服的定义，当然是来自维基百科。

软件文档或者源代码文档是指与软件系统及其软件工程过程有关联的文本实体。文档的类型包括软件需求文档，设计文档，测试文档，用户手册等。其中的需求文档，设计文档和测试文档一般是在软件开发过程中由开发者写就的，而用户手册等非过程类文档是由专门的非技术类写作人员写就的。

早期的软件文档主要指的是用户手册，根据 Barker 的定义，文档是用来对软件系统界面元素的设计、规划和实现过程的记录，以此来增强系统的可用性。而 Forward 则认为软件文档是被软件工程师之间用作沟通交流的一种方式，沟通的信息主要是有关所开发的软件系统。Parnas 则强调文档的权威性，他认为文档应该提供对软件系统的精确描述。

综上，我们可以将软件文档定义为：

1. 文档是一种对软件系统的书面描述； 2. 文档应当精确地描述软件系统； 3. 软件文档是软件工程师之间用作沟通交流的一种方式； 4. 文档的类型有很多，包括软件需求文档，设计文档，测试文档，用户手册等； 5. 文档的呈现方式有很多，可以是传统的书面文字形式或图表形式，也可是动态的网页形式

那么这里说的 Python 文档指的是什么呢？一个方面就是每个学习者要学习 Python，Python 的开发者们（他们都是大牛）给我们这些小白提供了什么东西没有？能够让我们给他们这些大牛沟通，理解 Python 中每个函数、指令等的含义和用法呢？

有。大牛就是大牛，他们准备了，而且还不止一个。

真诚的敬告所有看本教程的诸位，要想获得编程上的升华，看文档是必须的。文档胜过了所有的教程和所有的老师以及所有的大牛。为什么呢？其中原因，都要等待看官看懂了之后，有了体会感悟之后才能明白。

Python 文档的网址：<https://docs.python.org/2/>，这是 Python2.x，从这里也可以找到 Python3.x 的文档。

当然，除了看官方文档之外，自己写的东西也可以写上文档。这个先不要着急，我们会在后续的学习中看到。

总目录 (页 0)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

4

函数

函数(1)

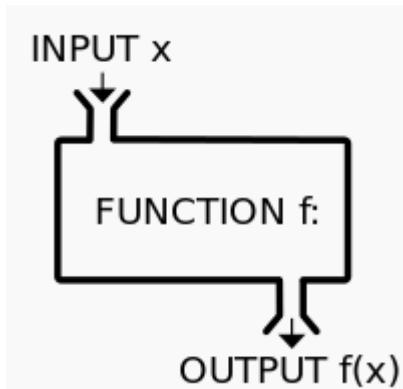
函数，对于人类来讲，能够发展到这个数学思维层次，是一个飞跃。可以说，它的提出，直接加快了现代科技和社会的发展，不论是现代的任何科技门类，乃至于经济学、政治学、社会学等，都已经普遍使用函数。

下面一段来自维基百科（在本教程中，大量的定义来自维基百科，因为它真的很百科）：[函数词条](#)

函数这个数学名词是莱布尼兹在 1694 年开始使用的，以描述曲线的一个相关量，如曲线的斜率或者曲线上的某一点。莱布尼兹所指的函数现在被称作可导函数，数学家之外的普通人一般接触到的函数即属此类。对于可导函数可以讨论它的极限和导数。此两者描述了函数输出值的变化同输入值变化的关系，是微积分学的基础。

中文的“函数”一词由清朝数学家李善兰译出。其《代数学》书中解释：“凡此變數中函（包含）彼變數者，則此為彼之函數”。

函数，从简单到复杂，各式各样。前面提供的维基百科中的函数词条，里面可以做一个概览。但不管什么样子的函数，都可以用下图概括：



有初中数学水平都能理解一个大概了。这里不赘述。

本讲重点说明用 Python 怎么来构造一个函数。

深入理解函数

在中学数学中，可以用这样的方式定义函数： $y=4x+3$ ，这就是一个一次函数，当然，也可以写成： $f(x)=4x+3$ 。其中 x 是变量，它可以代表任何数。

当 $x=2$ 时，代入到上面的函数表达式：

$$f(2) = 4*2+3 = 11$$

$$\text{所以: } f(2) = 11$$

但是，这并不是函数的全部，在函数中，其实变量并没有规定只能是一个数，它可以是馒头、还可是苹果，不知道读者是否对函数有这个层次的理解。请继续阅读即更深刻

变量不仅仅是数

变量 x 只能是任意数吗？其实，一个函数，就是一个对应关系。看官尝试着将上面表达式的 x 理解为馅饼， $4x+3$ ，就是 4 个馅饼在加上 3（一般来讲，单位是统一的，但你非让它不统一，也无妨），这个结果对应着另外一个东西，那个东西比如说是 iphone。或者说可以理解为 4 个馅饼加 3 就对应一个 iphone。这就是所谓映射关系。

所以， x ，不仅仅是数，可以是你认为的任何东西。

变量本质——占位符

函数中为什么变量用 x ？这是一个有趣的问题，自己 google 一下，看能不能找到答案。

我也不清楚原因。不过，我清楚地知道，变量可以用 x ，也可以用别的符号，比如 $y,z,k,i,j\dots$ ，甚至用 $\alpha,\beta,\epsilon,\theta$ 这样的字母组合也可以。

变量在本质上就是一个占位符。这是一针见血的理解。什么是占位符？就是先把那个位置用变量占上，表示这里有一个东西，至于这个位置放什么东西，以后再说，反正先用一个符号占着这个位置（占位符）。

其实在高级语言编程中，变量比我们在初中数学中学习的要复杂。但是，先不管那些，复杂东西放在以后再说。现在，就按照初中数学来研究 Python 中的变量。

通常使小写字母来命名 Python 中的变量，也可以在其中加上下划线什么的，表示区别。

比如： α,x,j,p_beta ，这些都可以做为 Python 的变量。

建立简单函数

```
>>> a = 2
>>> y=3*a+2
>>> y
8
```

这种方式建立的函数，跟在初中数学中学习的没有什么区别。当然，这种方式的函数，在编程实践中的用途不大，一般是在学习阶段理解函数来使用的。

别急躁，你在输入 `a=3`,然后输入 `y`，看看得到什么结果呢？

```
>>> a=2
>>> y=3*a+2
>>> y
8
>>> a=3
>>> y
8
```

是不是很奇怪？为什么后面已经让`a`等于3了，结果 `y` 还是 8。

还记得前面已经学习过的关于“变量赋值”的原理吗？`a=2` 的含义是将 2 这个对象贴上了变量 `a` 标签，经过计算，得到了 8，之后变量 `y` 引用了对象 8。当变量 `a` 引用的对象修改为3的时候，但是`y`引用的对象还没有变，所以，还是 8。再计算一次，`y` 的连接对象就变了：

```
>>> a=3
>>> y
8
>>> y=3*a+2
>>> y
11
```

特别注意，如果没有先 `a=2`，就直接下函数表达式了，像这样，就会报错。

```
>>> y=3*a+2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

注意看错误提示，`a` 是一个变量，提示中告诉我们这个变量没有定义。显然，如果函数中要使用某个变量，不得不提前定义出来。定义方法就是给这个变量赋值。

建立实用的函数

上面用命令方式建立函数，还不够“正规化”，那么就来写一个.py 文件吧。

例如下面的代码：

```
#!/usr/bin/env python
#coding:utf-8

def add_function(a,b):
```

```
c = a + b
print c

if __name__ == "__main__":
    add_function(2,3)
```

然后将文件保存，我把她命名为 20101.py，你根据自己的喜好取个名字。

然后我就进入到那个文件夹，运行这个文件，出现下面的结果，如图：

```
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ ls
105-1.py 105.py 106-1.py
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ python 106-1.py
5
qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ |
```

你运行的结果是什么？如果没有得到上面的结果，你就非常认真地检查代码，是否跟我写的完全一样，注意，包括冒号和空格，都得一样。冒号和空格很重要。

下面开始庖丁解牛：

- `def add_function(a,b)`：这里是函数的开始。在声明要建立一个函数的时候，一定要使用 `def` (`def` 就是英文 `define` 的前三个字母)，意思就是告知计算机，这里要声明一个函数；`add_function` 是这个函数名称，取名字是有讲究的，就好比你的名字一样。在 Python 中取名字的讲究就是要有一定意义，能够从名字中看出这个函数是用来干什么的。从 `add_function` 这个名字中，是不是看出她是用来计算加法的呢（严格地说，是把两个对象“相加”，这里相加的含义是比较宽泛的，包括对字符串等相加）？`(a,b)`这个括号里面的是这个函数的参数，也就是函数变量。冒号，这个冒号非常非常重要，如果少了，就报错了。冒号的意思就是下面好开始真正的函数内容了。
- `c=a+b` 特别注意，这一行比上一行要缩进四个空格。这是 Python 的规定，要牢记，不可丢掉，丢了就报错。然后这句话就是将两个参数(变量)相加，结果赋值与另外一个变量 `c`。
- `print c` 还是提醒看官注意，缩进四个空格。将得到的结果 `c` 的值打印出来。
- `if __name__=="__main__"`：这句话先照抄，不解释，因为在[《自省》](#)有说明，不知道你是不是认真阅读了。注意就是不缩进了。
- `add_function(2,3)` 这才是真正调用前面建立的函数，并且传入两个参数：`a=2,b=3`。仔细观察传入参数的方法，就是把 `2` 放在 `a` 那个位置，`3` 放在 `b` 那个位置（所以说，变量就是占位符）。

解牛完毕，做个总结：

定义函数的格式为：

```
def 函数名(参数 1, 参数 2, ..., 参数 n):
    函数体 (语句块)
```

是不是样式很简单呢？

几点说明：

- 函数名的命名规则要符合 Python 中的命名要求。一般用小写字母和单下划线、数字等组合
- def 是定义函数的关键词，这个简写来自英文单词 define
- 函数名后面是圆括号，括号里面，可以有参数列表，也可以没有参数
- 千万不要忘记了括号后面的冒号
- 函数体（语句块），相对于 def 缩进，按照 Python 习惯，缩进四个空格

看简单例子，深入理解上面的要点：

```
>>> def name():      # 定义一个无参数的函数，只是通过这个函数打印
...   print "qiwsir" # 缩进 4 个空格
...
>>> name()          # 调用函数，打印结果
qiwsir

>>> def add(x,y):    # 定义一个非常简单的函数
...   return x+y     # 缩进 4 个空格
...
>>> add(2,3)        # 通过函数，计算 2+3
5
```

注意上面的 add(x,y) 函数，在这个函数中，没有特别规定参数 x,y 的类型。其实，这句话本身就是错的，还记得在前面已经多次提到，在 Python 中，变量无类型，只有对象才有类型，这句话应该说成：x,y 并没有严格规定其所引用的对象类型。这是 Python 跟某些语言比如 java 很大的区别，在有些语言中，需要在定义函数的时候告诉函数参数的数据类型。Python 不用那样做。

为什么？列位不要忘记了，这里的所谓参数，跟前面说的变量，本质上是一回事。只有当用到该变量的时候，才建立变量与对象的对应关系，否则，关系不建立。而对象才有类型。那么，在 add(x,y) 函数中，x,y 在引用对象之前，是完全飘忽的，没有被贴在任何一个对象上，换句话说它们有可能引用任何对象，只要后面的运算许可，如果后面的运算不许可，则会报错。

```
>>> add("qiw","sir")  # 这里，x="qiw",y="sir"，让函数计算 x+y,也就是"qiw"+ "sir"
'qiwsir'

>>> add("qiwsir",4)
```

```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in add
TypeError: cannot concatenate 'str' and 'int' objects #仔细阅读报错信息，就明白错误之处了

```

从实验结果中发现： $x+y$ 的意义完全取决于对象的类型。在 Python 中，将这种依赖关系，称之为多态。对于 Python 中的多态问题，以后还会遇到，这里仅仅以此例子显示一番。请看官要留心注意的：Python 中为对象编写接口，而不是为数据类型。读者先留心一下这句话，或者记住它，随着学习的深入，会领悟到其真谛的。

此外，也可以将函数通过赋值语句，与某个变量建立引用关系：

```

>>> result = add(3,4)
>>> result
7

```

在这里，其实解释了函数的一个秘密。 $\text{add}(x,y)$ 在被运行之前，计算机内是不存在的，直到代码运行到这里的时候，在计算机中，就建立起来了一个对象，这就如同前面所学习过的字符串、列表等类型的对象一样，运行 $\text{add}(x,y)$ 之后，也建立了一个 $\text{add}(x,y)$ 的对象，这个对象与变量 `result` 可以建立引用关系，并且 $\text{add}(x,y)$ 将运算结果返回。于是，通过 `result` 就可以查看运算结果。

如果看官上面一段，感觉有点吃力或者晕乎，也不要紧，那就再读一边。是在搞不明白，就不要搞了。随着学习的深入，它会被明白的。

关于命名

到现在为止，我们已经接触过变量的命名、函数的命名问题。似乎已经到了将命名问题进行总结的时候了。

在某国，向来重视“名”，所谓“名不正言不顺”，取名字或者给什么东西命名，常常是天大的事情，在很多时候就是为了那个“名”进行争斗。

江湖上还有的大师，会通过某个人的名字来预测他/她的吉凶祸福等。看来名字这玩意太重要了。“名不正，言不顺”，歪解：名字不正规化，就不顺。这是歪解，希望不要影响看官正确理解。不知道大师们是不是能够通过外国人名字预测外国人的吉凶祸福呢？

不管怎样，某国人是很在意名字的，旁边有个国家似乎就不在乎。

Python 也很在乎名字问题，其实，所有高级语言对名字都有要求。为什么呢？因为如果命名乱了，计算机就有点不知所措了。看 Python 对命名的一般要求。

- 文件名：全小写，可使用下划线

- 函数名: 小写, 可以用下划线风格单词以增加可读性。如: myfunction, my_example_function。注意: 混合大小写仅被允许用于这种风格已经占据优势的时候, 以便保持向后兼容。有的人, 喜欢用这样的命名风格: myFunction, 除了第一个单词首字母外, 后面的单词首字母大写。这也是可以的, 因为在某些语言中就习惯如此。
- 函数的参数: 如果一个函数的参数名称和保留的关键字(所谓保留关键字, 就是 Python 语言已经占用的名称, 通常被用来做为已经有的函数等的命名了, 你如果还用, 就不行了。)冲突, 通常使用一个后缀下划线好于使用缩写或奇怪的拼写。
- 变量: 变量名全部小写, 由下划线连接各个单词。如 color = WHITE, this_is_a_variable = 1。

其实, 关于命名的问题, 还有不少争论呢? 最典型的是所谓匈牙利命名法、驼峰命名等。如果你喜欢, 可以 google 一下。以下内容供参考:

- [匈牙利命名法](#)
- [驼峰式大小写](#)
- [帕斯卡命名法](#)
- [Python命名的官方要求](#), 如果看官的英文可以, 一定要阅读。如果英文稍逊, 可以来阅读[中文](#), 不用梯子能行吗? 看你命了。

调用函数

前面的例子中已经有了一些关于调用的问题, 为了深入理解, 把这个问题单独拿出来看看。

为什么要写函数? 从理论上说, 不用函数, 也能够编程, 我们在前面已经写了程序, 就没有写函数, 当然, 用 Python 的内建函数姑且不算了。现在之所以使用函数, 主要是:

1. 降低编程的难度, 通常将一个复杂的大问题分解成一系列更简单的小问题, 然后将小问题继续划分成更小的问题, 当问题细化为足够简单时, 就可以分而治之。为了实现这种分而治之的设想, 就要通过编写函数, 将各个小问题逐个击破, 再集合起来, 解决大的问题。(看官请注意, 分而治之的思想是编程的一个重要思想, 所谓“分治”方法也。)
2. 代码重(chong, 二声音)用。在编程的过程中, 比较忌讳同样一段代码不断的重复, 所以, 可以定义一个函数, 在程序的多个位置使用, 也可以用于多个程序。当然, 后面我们还会讲到“模块”(此前也涉及到了, 就是 import 导入的那个东西), 还可以把函数放到一个模块中供其他程序员使用。也可以使用其他程序员定义的函数(比如 import ..., 前面已经用到了, 就是应用了别人——创造 Python 的人——写好的函数)。这就避免了重复劳动, 提供了工作效率。

这样看来，函数还是很必要的了。废话少说，那就看函数怎么调用吧。以 `add(x,y)` 为例，前面已经演示了基本调用方式，此外，还可以这样：

```
>>> def add(x,y):    #为了能够更明了显示参数赋值特点，重写此函数
...   print "x=",x  #分别打印参数赋值结果
...   print "y=",y
...   return x+y
...
>>> add(10,3)      #x=10,y=3
x= 10
y= 3
13

>>> add(3,10)      #x=3,y=10
x= 3
y= 10
13
```

所谓调用，最关键是要弄清楚如何给函数的参数赋值。这里就是按照参数次序赋值，根据参数的位置，值与之对应。

```
>>> add(x=10,y=3)  #同上
x= 10
y= 3
13
```

还可以直接把赋值语句写到里面，就明确了参数和对象的关系。当然，这时候顺序就不重要了，也可以这样

```
>>> add(y=10,x=3)  #x=3,y=10
x= 3
y= 10
13
```

在定义函数的时候，参数可以像前面那样，等待被赋值，也可以定义的时候就赋给一个默认值。例如：

```
>>> def times(x,y=2):    #y的默认值为 2
...   print "x=",x
...   print "y=",y
...   return x*y
...
>>> times(3)           #x=3,y=2
x= 3
y= 2
6
```

```
>>> times(x=3)      #同上
x= 3
y= 2
6
```

如果不给那个有默认值的参数传递值（赋值的另外一种说法），那么它就是用默认的值。如果给它传一个，它就采用新赋给它的值。如下：

```
>>> times(3,4)      #x=3,y=4,y 的值不再是 2
x= 3
y= 4
12

>>> times("qiwsir")    #再次体现了多态特点
x= qiwsir
y= 2
'qiwsirqiwsir'
```

给列位看官提一个思考题，请在闲暇之余用 Python 完成：写两个数的加、减、乘、除的函数，然后用这些函数，完成简单的计算。

注意事项

下面的若干条，是常见编写代码的注意事项：

1. 别忘了冒号。一定要记住符合语句首行末尾输入“：“（if,while,for 等的第一行）
2. 从第一行开始。要确定顶层（无嵌套）程序代码从第一行开始。
3. 空白行在交互模式提示符下很重要。模块文件中符合语句内的空白行常被忽视。但是，当你在交互模式提示符下输入代码时，空白行则是会结束语句。
4. 缩进要一致。避免在块缩进中混合制表符和空格。
5. 使用简洁的 for 循环，而不是 while or range.相比，for 循环更易写，运行起来也更快
6. 要注意赋值语句中的可变对象。
7. 不要期待在原处修改的函数会返回结果,比如 list.append()，这在可修改的对象中特别注意
8. 调用函数是，函数名后面一定要跟随着括号，有时候括号里面就是空空的，有时候里面放参数。
9. 不要在导入和重载中使用扩展名或路径。

以上各点如果有不理解的，也不要紧，在以后编程中，是不是地回来复习一下，能不断领悟其内涵。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我, 请通过支付宝: qiwsir@126.com,不胜感激。

函数(2)

在上一节中，已经明确了函数的基本结构和初步的调用方法。但是，上一节中写的函数，还有点缺憾，不知道读者是否觉察到了。我把结果是用 `print` 语句打印出来的。这是实际编程中广泛使用的吗？肯定不是。因为函数在编程中，起到一段具有抽象价值的代码作用，一般情况下，用它得到一个结果，这个结果要用在其它的运算中。所以，不能仅仅局限在把某个结果打印出来。所以，函数必须返回一个结果。

结论：函数要有返回值，也必须有返回值。

返回值

为了能够说明清楚，先编写一个函数。还记得斐波那契数列吗？我打算定义一个能够得到斐波那契数列的函数，从而实现可以实现任意的数列。你先想想，要怎么写？

参考代码：

```
#!/usr/bin/env Python
# coding=utf-8

def fibs(n):
    result = [0,1]
    for i in range(n-2):
        result.append(result[-2] + result[-1])
    return result

if __name__ == "__main__":
    lst = fibs(10)
    print lst
```

把含有这些代码的文件保存为名为 `20202.py` 的文件。在这个文件中，首先定义了一个函数，名字叫做 `fibs`，其参数是输入一个整数。在后面，通过 `lst = fibs(10)` 调用这个函数。这里参数给的是 10，就意味着要得到 $n=10$ 的斐波那契数列。

运行后打印数列：

```
$ python 20202.py
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

当然，如果要换 n 的值，只需要在调用函数的时候，修改一下参数即可。这才体现出函数的优势呢。

观察 `fibs` 函数，最后一个语句 `return result`，意思是将变量 `result` 的值返回。返回给谁呢？一般这类函数调用的时候，要通过类似 `lst = fibs(10)` 的语句，那么返回的那个值，就被变量 `lst` 贴上了，通过 `lst` 就能得到该值。如果没有这个赋值语句，虽然函数照样返回值，但是它飘忽在内存中，我们无法得到，并且最终还被当做垃圾被 Python 回收了。

注意：上面的函数之返回了一个返回值（是一个列表），有时候需要返回多个，是以元组形式返回。

```
>>> def my_fun():
...     return 1,2,3
...
>>> a = my_fun()
>>> a
(1, 2, 3)
```

有的函数，没有 `return`，一样执行完毕，就算也干了某些活儿吧。事实上，不是没有返回值，也有，只不过是 `None`。比如这样一个函数：

```
>>> def my_fun():
...     print "I am doing somthin."
...

```

我在交互模式下构造一个很简单的函数，注意，我这是构造了一个简单函数，如果是复杂的，千万不要在交互模式下做。如果你非要做，是能尝到苦头的。

这个函数的作用就是打印出一段话。也就是执行这个函数，就能打印出那段话，但是没有 `return`。

```
>>> a = my_fun()
I am doing somthin.
```

我们再看看那个变量 `a`，到底是什么

```
>>> print a
None
```

这就是这类只干活儿，没有 `return` 的函数，返回给变量的是一个 `None`。这种模样的函数，通常不用上述方式调用，而采用下面的方式，因为他们返回的是 `None`，似乎这个返回值利用价值不高，于是就不用找一个变量来接受返回值了。

```
>>> my_fun()
I am doing somthin.
```

特别注意那个 `return`，它还有一个作用。观察下面的函数和执行结果，看看能不能发现它的另外一个作用。

```
>>> def my_fun():
...     print "I am coding."
...     return
...     print "I finished."
...
>>> my_fun()
I am coding.
```

看出玄机了吗？在函数中，本来有两个 print 语句，但是中间插入了一个 return，仅仅是一个 return。当执行函数的时候，只执行了第一个 print 语句，第二个并没有执行。这是因为第一个之后，遇到了 return，它告诉函数要返回，即中断函数体内的流程，离开这个函数。结果第二个 print 就没有被执行。所以，return 在这里就有一个作用，结束正在执行的函数。有点类似循环中的 break 的作用。

函数中的文档

“程序在大多数情况下是给人看的，只是偶尔被机器执行以下。”所以，写程序必须要写注释。前面已经有过说明，如果用 `#` 开始，Python 就不执行那句（Python 看不到它，但是人能看到），它就作为注释存在。

除了这样的一句之外，一般在每个函数名字的下面，还要写一写文档，以此来说明这个函数的用途。

```
#!/usr/bin/env python
# coding=utf-8

def fibs(n):
    """
    This is a Fibonacci sequence.
    """
    result = [0,1]
    for i in range(n-2):
        result.append(result[-2] + result[-1])
    return result

if __name__ == "__main__":
    lst = fibs(10)
    print lst
```

在这个函数的名称下面，用三个引号的方式，包裹着对这个函数的说明，那个就是函数文档。

还记得在《自省》那节中，提到的 `__doc__` 吗？对于函数，它的内容就来自这里。

```
>>> def my_fun():
...     """
...     This is my function.
```

```

...     """
...
...     print "I am a craft."
...
>>> my_fun.__doc__
\n This is my function.\n  '

```

如果在交互模式中用 `help(my_fun)` 得到的也是三个引号所包裹的文档信息。

```
Help on function my_fun in module __main__:
```

```

my_fun()
This is my function.

```

参数和变量

参数

虽然在一节，已经知道如何通过函数的参数传值，如何调用函数等。但是，这里还有必要进一步讨论参数问题。在别的程序员嘴里，你或许听说过“形参”、“实参”、“参数”等名词，到底指什么呢？

在定义函数的时候（`def` 来定义函数，称为 `def` 语句），函数名后面的括号里如果有变量，它们通常被称为“形参”。调用函数的时候，给函数提供的值叫做“实参”，或者“参数”。

其实，根本不用区分这个，因为没有什么意义，只不过类似孔乙己先生知道茴香豆的茴字有多少种写法罢了。但是，我居然碰到过某公司面试官问这种问题。

在本教程中，把那个所谓实参，就称之为值（或者数据、或者对象），形参就笼统称之为参数（似乎不很合理，但是接近数学概念）。

比较参数和变量

参数问题就算说明白了，糊涂就糊涂吧，也没有什么关系。不过，对于变量和参数，这两个就不能算糊涂账了。因为它容易让人糊涂了。

在数学的函数中 $y = 3x + 2$ ，那个 x 叫做参数，也可以叫做变量。但是，在编程语言的函数中，与此有异。

先参考一段来自[微软网站](#)的比较高度抽象，而且意义涵盖深远的说明。我摘抄过来，看官读一读，是否理解，虽然是针对VB而言的，一样有启发。

参数和变量之间的差异 (Visual Basic)

多数情况下，过程必须包含有关调用环境的一些信息。执行重复或共享任务的过程对每次调用使用不同的信息。此信息包含每次调用过程时传递给它的变量、常量和表达式。

若要将此信息传递给过程，过程先要定义一个形参，然后调用代码将一个实参传递给所定义的形参。您可以将形参当作一个停车位，而将实参当作一辆汽车。就像一个停车位可以在不同时间停放不同的汽车一样，调用代码在每次调用过程时可以将不同的实参传递给同一个形参。

形参表示一个值，过程希望您在调用它时传递该值。

当您定义 Function 或 Sub 过程时，需要在紧跟过程名称的括号内指定形参列表。对于每个形参，您可以指定名称、数据类型和传入机制（ ByVal (Visual Basic) 或 ByRef (Visual Basic) ）。您还可以指示某个形参是可选的。这意味着调用代码不必传递它的值。

每个形参的名称均可作为过程内的局部变量。形参名称的使用方法与其他任何变量的使用方法相同。

实参表示在您调用过程时传递给过程形参的值。调用代码在调用过程时提供参数。

调用 Function 或 Sub 过程时，需要在紧跟过程名称的括号内包括实参列表。每个实参均与此列表中位于相同位置的那个形参相对应。

与形参定义不同，实参没有名称。每个实参就是一个表达式，它包含零或多个变量、常数和文本。求值的表达式的数据类型通常应与为相应形参定义的数据类型相匹配，并且在任何情况下，该表达式值都必须可转换为此形参类型。

看官如果硬着头皮看完这段引文，发现里面有几个关键词：参数、变量、形参、实参。本来想弄清楚参数和变量，结果又冒出另外两个东东，更混乱了。请稍安勿躁，本来这段引文就是有点多余，但是，之所以引用，就是让列位开阔一下眼界，在编程业界，类似的东西有很多名词。下次听到有人说这些，不用害怕啦，反正自己听过了。

在 Python 中，没有这么复杂。

看完上面让人晕头转向的引文之后，再看下面的代码，就会豁然开朗了。

```
>>> def add(x): #x 是参数，准确说是形参
...   a = 10 #a 是变量
...   return a+x #x 就是那个形参作为变量，其本质是要传递赋给这个函数的值
...
>>> x = 3      #x 是变量，只不过在函数之外
>>> add(x)    #这里的 x 是参数，但是它由前面的变量 x 传递对象 3
13
>>> add(3)    #把上面的过程合并了
13
```

至此，看官是否清楚了一点点。当然，我所表述不正确之处或者理解错误之处，也请看官不吝赐教，小可作揖感谢。

其实没有那么复杂。关键要理解函数名括号后面的东东（管它什么参呢）的作用是传递值。

全局变量和局部变量

下面是一段代码，注意这段代码中有一个函数 funcx()，这个函数里面有一个变量 x=9，在函数的前面也有一个变量 x=2

```
x = 2

def funcx():
    x = 9
    print "this x is in the funcx:-->",x

funcx()
print "-----"
print "this x is out of funcx:-->",x
```

那么，这段代码输出的结果是什么呢？看：

```
this x is in the funcx:--> 9
-----
this x is out of funcx:--> 2
```

从输出看出，运行 funcx()，输出了 funcx() 里面的变量 x=9；然后执行代码中的最后一行，print "this x is out of funcx:-->",x

特别要关注的是，前一个 x 输出的是函数内部的变量 x；后一个 x 输出的是函数外面的变量 x。两个变量彼此没有互相影响，虽然都是 x。从这里看出，两个 x 各自在各自的领域内起到作用。

把那个只在函数体内（某个范围内）起作用的变量称之为 **局部变量**。

有局部，就有对应的全部，在汉语中，全部变量，似乎有歧义，幸亏汉语丰富，于是又取了一个名词：**全局变量**

```
x = 2
def funcx():
    global x  #跟上面函数的不同之处
    x = 9
    print "this x is in the funcx:-->",x

funcx()
```

```
print "-----"
print "this x is out of funcx:-->",x
```

以上两段代码的不同之处在于，后者的在函数内多了一个 `global x`，这句话的意思是在声明 `x` 是全局变量，也就是说这个 `x` 跟函数外面的那个 `x` 同一个，接下来通过 `x=9` 将 `x` 的引用对象变成了 9。所以，就出现了下面的结果。

```
this x is in the funcx:--> 9
-----
this x is out of funcx:--> 9
```

好似全局变量能力很强悍，能够统帅函数内外。但是，要注意，这个东西要慎重使用，因为往往容易带来变量的换乱。内外有别，在程序中一定要注意的。

命名空间

这是一个比较不容易理解的概念，特别是对于初学者而言，似乎它很飘渺。我在维基百科中看到对它的定义，不仅定义比较好，连里面的例子都不错。所以，抄录下来，帮助读者理解这个名词。

命名空间（英语：Namespace）表示标识符（identifier）的可见范围。一个标识符可在多个命名空间中定义，它在不同命名空间中的含义是互不相干的。这样，在一个新的命名空间中可定义任何标识符，它们不会与任何已有的标识符发生冲突，因为已有的定义都处于其它命名空间中。

例如，设 Bill 是 X 公司的员工，工号为 123，而 John 是 Y 公司的员工，工号也是 123。由于两人在不同的公司工作，可以使用相同的工号来标识而不会造成混乱，这里每个公司就表示一个独立的命名空间。如果两人在同一家公司工作，其工号就不能相同了，否则在支付工资时便会发生混乱。

这一特点是使用命名空间的主要理由。在大型的计算机程序或文档中，往往会出现数百或数千个标识符。命名空间（或类似的方法，见“命名空间的模拟”一节）提供一隐藏区域标识符的机制。通过将逻辑上相关的标识符组织成相应的命名空间，可使整个系统更加模块化。

在编程语言中，命名空间是对作用域的一种特殊的抽象，它包含了处于该作用域内的标识符，且本身也用一个标识符来表示，这样便将一系列在逻辑上相关的标识符用一个标识符组织了起来。许多现代编程语言都支持命名空间。在一些编程语言（例如 C++ 和 Python）中，命名空间本身的标识符也属于一个外层的命名空间，也即命名空间可以嵌套，构成一个命名空间树，树根则是无名的全局命名空间。

函数和类的作用域可被视作隐式命名空间，它们和可见性、可访问性和对象生命周期不可分割的联系在一起。

显然，用“命名空间”或者“作用域”这样的名词，就是因为有了函数（后面还会有类）之后，在函数内外都可能有外形一样的符号（标识符），在 Python 中（乃至于其它高级语言），通常就是变量，为了区分此变量非彼变量（虽然外形一样），需要用这样的东西来框定每个变量所对应的值（发生作用的范围）。

前面已经讲过，变量和对象（就是所变量所对应的值）之间的关系是：变量类似标签，贴在了对象上。也就是，通过赋值语句实现了一个变量标签对应一个数据对象（值），这种对应关系让你想起了什么？映射！Python 中唯一的映射就是 dict，里面有“键值对”。变量和值得关系就有点像“键”和“值”的关系。有一个内建函数 vars，可以帮助我们研究一下这种对应关系。

```
>>> x = 7
>>> scope = vars()
>>> scope['x']
7
>>> scope['x'] += 1
>>> x
8
>>> scope['x']
8
```

既然如此，诚如前面的全局变量和局部变量，即使是同样一个变量名称。但是它在不同范围（还是用“命名空间”这个词是不是更专业呢？）对应不同的值。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

函数(3)

在设计函数的时候，有时候我们能够确认参数的个数，比如一个用来计算圆面积的函数，它所需要的参数就是半径（ πr^2 ），这个函数的参数是确定的。

| 你能不能写一个能够计算圆面积的函数呢？

然而，这个世界不总是这么简单的，也不总是这么确定的，反而不确定性是这个世界常常存在的。如果看官了解量子力学——好多人听都没有听过的东西——就更理解真正的不确定性了。当然，不用研究量子力学也一样能够体会到，世界充满里了不确定性。不是吗？塞翁失马焉知非福，这不就是不确定性吗？

参数收集

既然有很多不确定性，那么函数的参数的个数，也当然有不确定性，函数怎么解决这个问题呢？Python 用这样的方式解决参数个数的不确定性：

```
def func(x,*arg):
    print x      #输出参数 x 的值
    result = x
    print arg    #输出通过 *arg 方式得到的值
    for i in arg:
        result +=i
    return result

print func(1,2,3,4,5,6,7,8,9)  #赋给函数的参数个数不仅仅是 2个
```

运行此代码后，得到如下结果：

```
1          #这是函数体内的第一个 print，参数x得到的值是 1
(2, 3, 4, 5, 6, 7, 8, 9) #这是函数内的第二个 print，参数 arg 得到的是一个元组
45         #最后的计算结果
```

从上面例子可以看出，如果输入的参数个数不确定，其它参数全部通过 *arg，以元组的形式由 arg 收集起来。对照上面的例子不难发现：

- 值 1 传给了参数 x
- 值 2,3,4,5,6,7,8,9 被塞入一个 tuple 里面，传给了 arg

为了能够更明显地看出 *args（名称可以不一样，但是 * 符号必须要有），可以用下面的一个简单函数来演示：

```
>>> def foo(*args):
...     print args    #打印通过这个参数得到的对象
...
...
```

下面演示分别传入不同的值，通过参数 `*args` 得到的结果：

```
>>> foo(1,2,3)
(1, 2, 3)

>>> foo("qiwsir","qiwsir.github.io","python")
('qiwsir', 'qiwsir.github.io', 'python')

>>> foo("qiwsir",307,[{"qiwsir":2}, {"name":"qiwsir", "lang":"python"}])
('qiwsir', 307, ['qiwsir', 2], {'lang': 'python', 'name': 'qiwsir'})
```

不管是什什么，都一股脑地塞进了 tuple 中。

```
>>> foo("python")
('python',)
```

即使只有一个值，也是用 tuple 收集它。特别注意，在 tuple 中，如果只有一个元素，后面要有一个逗号。

还有一种可能，就是不给那个 `*args` 传值，也是许可的。例如：

```
>>> def foo(x, *args):
...     print "x:",x
...     print "tuple:",args
...
>>> foo(7)
x: 7
tuple: ()
```

这时候 `*args` 收集到的是一个空的 tuple。

在各类编程语言中，常常会遇到以 `foo`, `bar`, `foobar` 等之类的命名，不管是对变量、函数还是后面要讲到的类。这是什么意思呢？下面是来自维基百科的解释。

在计算机程序设计与计算机技术的相关文档中，术语 `foobar` 是一个常见的无名氏化名，常被作为“伪变量”使用。

从技术上讲，“`foobar`”很可能在 1960 年代至 1970 年代初通过迪吉多的系统手册传播开来。另一种说法是，“`foobar`”可能来源于电子学中反转的 `foo` 信号；这是因为如果一个数字信号是低电平有效（即负压或零电压代表“1”），那么在信号标记上方一般会标有一根水平横线，而横线的英文即为“`bar`”。在《新黑客辞典》中，还提到“`foo`”可能早于“`FUBAR`”出现。

单词“foobar”或分离的“foo”与“bar”常出现于程序设计的案例中，如同Hello World程序一样，它们常被用于向学习者介绍某种程序语言。“foo”常被作为函数/方法的名称，而“bar”则常被用作变量名。

除了用`*args`这种形式的参数接收多个值之外，还可以用`**kargs`的形式接收数值，不过这次有点不一样：

```
>>> def foo(**kargs):
...     print kargs
...
>>> foo(a=1,b=2,c=3) #注意观察这次赋值的方式和打印的结果
{'a': 1, 'c': 3, 'b': 2}
```

如果这次还用`foo(1,2,3)`的方式，会有什么结果呢？

```
>>> foo(1,2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes exactly 0 arguments (3 given)
```

如果用`**kargs`的形式收集值，会得到`dict`类型的数据，但是，需要在传值的时候说明“键”和“值”，因为在字典中是以键值对形式出现的。

看官到这里可能想了，不是不确定性吗？我也不知道参数到底会可能用什么样的方式传值呀，这好办，把上面的都综合起来。

```
>>> def foo(x,y,z,*args,**kargs):
...     print x
...     print y
...     print z
...     print args
...     print kargs
...
>>> foo('qiwsir',2,"python")
qiwsir
2
python
()
{}
>>> foo(1,2,3,4,5)
1
2
3
(4, 5)
{}
>>> foo(1,2,3,4,5,name="qiwsir")
1
```

```
2
3
(4, 5)
{'name': 'qiwsir'}
```

很 good 了，这样就能够足以应付各种各样的参数要求了。

另外一种传值方式

```
>>> def add(x,y):
...     return x + y
...
>>> add(2,3)
5
```

这是通常的函数调用方法，在前面已经屡次用到。这种方法简单明快，很容易理解。但是，世界总是多样性的，有时候你秀出下面的方式，甚至在某种情况用下面的方法可能更优雅。

```
>>> bars = (2,3)
>>> add(*bars)
5
```

先把要传的值放到元组中，赋值给一个变量 `bars`，然后用 `add(*bars)` 的方式，把值传到函数内。这有点像前面收集参数的逆过程。注意的是，元组中元素的个数，要跟函数所要求的变量个数一致。如果这样：

```
>>> bars = (2,3,4)
>>> add(*bars)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() takes exactly 2 arguments (3 given)
```

就报错了。

这是使用一个星号 `*`，是以元组形式传值，如果用 `**` 的方式，是不是应该以字典的形式呢？理当如此。

```
>>> def book(author,name):
...     print "%s is writing %s" % (author,name)
...
>>> bars = {"name":"Starter learning Python","author":"Kivi"}
>>> book(**bars)
Kivi is writing Starter learning Python
```

这种调用函数传值的方式，至少在我的编程实践中，用的不多。不过，不代表读者不用。这或许是习惯问题。

复习

python 中函数的参数通过赋值的方式来传递引用对象。下面总结通过总结常见的函数参数定义方式，来理解参数传递的流程。

```
def foo(p1,p2,p3,...)
```

这种方式最常见了，列出有限个数的参数，并且彼此之间用逗号隔开。在调用函数的时候，按照顺序以此对参数进行赋值，特备注意的是，参数的名字不重要，重要的是位置。而且，必须数量一致，一一对应。第一个对象（可能是数值、字符串等等）对应第一个参数，第二个对应第二个参数，如此对应，不得偏左也不得偏右。

```
>>> def foo(p1,p2,p3):
...     print "p1==>",p1
...     print "p2==>",p2
...     print "p3==>",p3
...
>>> foo("python",1,[{"qiwsir","github","io"]])  #一一对应地赋值
p1==> python
p2==> 1
p3==> ['qiwsir', 'github', 'io']

>>> foo("python")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes exactly 3 arguments (1 given)  #注意看报错信息

>>> foo("python",1,2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes exactly 3 arguments (4 given)  #要求 3 个参数，实际上放置了 4 个，报错
```

```
def foo(p1=value1,p2=value2,...)
```

这种方式比前面一种更明确某个参数的赋值，貌似这样就不乱子了，很明确呀。颇有一个萝卜对着一个坑的意味。

还是上面那个函数，用下面的方式赋值，就不用担心顺序问题了。

```
>>> foo(p3=3,p1=10,p2=222)
p1==> 10
```

```
p2==> 222
p3==> 3
```

也可以采用下面的方式定义参数，给某些参数有默认的值

```
>>> def foo(p1,p2=22,p3=33): #设置了两个参数 p2,p3 的默认值
...     print "p1==>",p1
...     print "p2==>",p2
...     print "p3==>",p3
...
>>> foo(11) #p1=11, 其它的参数为默认赋值
p1==> 11
p2==> 22
p3==> 33
>>> foo(11,222) #按照顺序, p2=222,p3 依旧维持原默认值
p1==> 11
p2==> 222
p3==> 33
>>> foo(11,222,333) #按顺序赋值
p1==> 11
p2==> 222
p3==> 333

>>> foo(11,p2=122)
p1==> 11
p2==> 122
p3==> 33

>>> foo(p2=122) #p1 没有默认值, 必须要赋值的, 否则报错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes at least 1 argument (1 given)
```

```
def foo(*args)
```

这种方式适合于不确定参数个数的时候，在参数 args 前面加一个 *，注意，仅一个哟。

```
>>> def foo(*args):    #接收不确定个数的数据对象
...     print args
...
>>> foo("qiwsir.github.io") #以 tuple 形式接收到, 哪怕是一个
('qiwsir.github.io',)
>>> foo("qiwsir.github.io","python")
('qiwsir.github.io', 'python')
```

```
def foo(**args)
```

这种方式跟上面的区别在于，必须接收类似 arg=val 形式的。

```
>>> def foo(**args): #这种方式接收，以 dictionary 的形式接收数据对象
...     print args
...
>>> foo(1,2,3)      #这样就报错了
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes exactly 0 arguments (3 given)

>>> foo(a=1,b=2,c=3) #这样就可以了，因为有了键值对
{'a': 1, 'c': 3, 'b': 2}
```

下面来一个综合的，看看以上四种参数传递方法的执行顺序

```
>>> def foo(x,y=2,*targs,**dargs):
...     print "x==>",x
...     print "y==>",y
...     print "targs_tuple==>",targs
...     print "dargs_dict==>",dargs
...
>>> foo("1x")
x==> 1x
y==> 2
targs_tuple==> ()
dargs_dict==> {}

>>> foo("1x","2y")
x==> 1x
y==> 2y
targs_tuple==> ()
dargs_dict==> {}

>>> foo("1x","2y","3t1","3t2")
x==> 1x
y==> 2y
targs_tuple==> ('3t1', '3t2')
dargs_dict==> {}

>>> foo("1x","2y","3t1","3t2",d1="4d1",d2="4d2")
x==> 1x
y==> 2y
```

```
targs_tuple==> ('3t1', '3t2')
dargs_dict==> {'d2': '4d2', 'd1': '4d1'}
```

[总目录 \(页 0\)](#)

如果你认为有必要打赏我, 请通过支付宝: qiwsir@126.com,不胜感激。

函数(4)

还记得在《[迭代](#)》中提到的那几个说出来就让人感觉牛 X 的名词吗？前面已经学习过“循环”、“遍历”和“迭代”了。现在来看“递归”。

递归

什么是递归？

递归，见递归。

这是对“递归”最精简的定义。还有故事类型的定义。

从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事。故事是什么呢？“从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事。故事是什么呢？”“从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事。故事是什么呢？……”

如果用上面的做递归的定义，总感觉有点调侃，来个严肃的(选自维基百科)：

递归（英语：Recursion），又译为递回，在数学与计算机科学中，是指在函数的定义中使用函数自身的方法。

最典型的递归例子之一是斐波那契数列，虽然前面用迭代的方式实现了它，但是那种方法在理解上不很直接。如果忘记了这个数列的定义，可以回到[《练习》](#)中查看。

根据斐波那契数列的定义，可以直接写成这样的斐波那契数列递归函数。

```
#!/usr/bin/env Python
# coding=utf-8

def fib(n):
    """
    This is Fibonacci by Recursion.
    """

    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

if __name__ == "__main__":
```

```
f = fib(10)
print f
```

把上述代码保存。这个代码的意图是要得到 n=10 的值。运行之：

```
$ python 20401.py
55
```

`fib(n-1) + fib(n-2)` 就是又调用了这个函数自己，实现递归。为了明确递归的过程，下面走一个计算过程（考虑到次数不能太多，就让 n=3）

1. n=3,fib(3)，自然要走 `return fib(3-1) + fib(3-2)` 分支
2. 先看 `fib(3-1)`,即 `fib(2)`，也要走 else 分支，于是计算 `fib(2-1) + fib(2-2)`
3. `fib(2-1)` 即 `fib(1)`，在函数中就要走 elif 分支，返回 1，即 `fib(2-1)=1`。同理，容易得到 `fib(2-2)=0`。将这两个值返回到上面一步。得到 `fib(3-1)=1+0=1`
4. 再计算 `fib(3-2)`,就简单了一些，返回的值是 1，即 `fib(3-2)=1`
5. 最后计算第一步中的结果： `fib(3-1) + fib(3-2) = 1 + 1 = 2`，将计算结果 2 作为返回值

从而得到 `fib(3)` 的结果是 2。

从上面的过程中可以看出，每个递归的过程，都是向着最初的已知条件 `a0=0,a1=1` 方向挺近一步，直到通过这个最底层的条件得到结果，然后再一层一层向上回馈计算机结果。

其实，上面的代码有一个问题。因为 `a0=0,a1=1` 是已知的了，不需要每次都判断一边。所以，还可以优化一下。优化的基本方案就是初始化最初的两个值。

```
#!/usr/bin/env Python
# coding=utf-8

"""

the better Fibonacci

"""

meno = {0:0, 1:1} #初始化

def fib(n):
    if not n in meno: #如果不在初始化范围内
        meno[n] = fib(n-1) + fib(n-2)
    return meno[n]

if __name__ == "__main__":
    f = fib(10)
    print f
```

```
#运行结果
$ python 20402.py
55
```

以上实现了递归，但是，至少在 Python 中，递归要慎重使用。在一般情况下，递归是能够被迭代或者循环替代的，而且后的效率常常比递归要高。所以，我个人的建议是，对使用递归要考虑的周密一下，不小心就永远运行下去了。

几个特殊函数

前面已经知道了如何编写、调用函数。此外，在 Python 中，有几个特别的函数，很有意思，它们常常被看做是 Python 能够进行所谓“函数式编程”的见证。

如果以前没有听过，等你开始进入编程界，也会经常听人说“函数式编程”、“面向对象编程”、“指令式编程”等属于。它们是什么呢？这个话题要从“编程范式”讲起。（以下内容源自维基百科）

编程范型或编程范式（英语：Programming paradigm），（范即模范之意，范式即模式、方法），是一类典型的编程风格，是指从事软件工程的一类典型的风格（可以对照方法学）。如：函数式编程、程序编程、面向对象编程、指令式编程等等为不同的编程范型。

编程范型提供了（同时决定了）程序员对程序执行的看法。例如，在面向对象编程中，程序员认为程序是一系列相互作用的对象，而在函数式编程中一个程序会被看作是一个无状态的函数计算的串行。

正如软件工程中不同的群体会提倡不同的“方法学”一样，不同的编程语言也会提倡不同的“编程范型”。一些语言是专门为某个特定的范型设计的（如 Smalltalk 和 Java 支持面向对象编程，而 Haskell 和 Scheme 则支持函数式编程），同时还有另一些语言支持多种范型（如 Ruby、Common Lisp、Python 和 Oz）。

编程范型和编程语言之间的关系可能十分复杂，由于一个编程语言可以支持多种范型。例如，C++ 设计时，支持过程化编程、面向对象编程以及泛型编程。然而，设计师和程序员们要考虑如何使用这些范型元素来构建一个程序。一个人可以用 C++ 写出一个完全过程化的程序，另一个人也可以用 C++ 写出一个纯粹的面向对象程序，甚至还有人可以写出杂揉了两种范型的程序。

不管读者是初学还是老油条，都建议将上面这段话认真读完，不管理解还是不理解，总能有点感觉的。

正如前面引文中所说的，Python 是支持多种范型的语言，可以进行所谓函数式编程，其突出体现在有这么几个函数：

filter、map、reduce、lambda、yield

有了它们，最大的好处是程序更简洁；没有它们，程序也可以用别的方式实现，只不过麻烦一些罢了。所以，还是能用则用之吧。更何况，恰当地使用这几个函数，能让别人感觉你更牛X。

(注: 本节不对 `yield` 进行介绍, 请阅读《[生成器](#)》)

lambda

`lambda` 函数, 是一个只用一行就能解决问题的函数, 听着是多么诱人呀。看下面的例子:

```
>>> def add(x): # 定义一个函数, 将输入的变量增加 3, 然后返回增加之后的值
...     x += 3
...     return x
...
>>> numbers = range(10)
>>> numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] # 有这样一个 list, 想让每个数字增加 3, 然后输出到一个新的 list 中

>>> new_numbers = []
>>> for i in numbers:
...     new_numbers.append(add(i)) # 调用 add() 函数, 并 append 到 list 中
...
>>> new_numbers
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

在这个例子中, `add()` 只是一个中间操作。当然, 上面的例子完全可以用别的方式实现。比如:

```
>>> new_numbers = [ i+3 for i in numbers ]
>>> new_numbers
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

首先说明, 这种列表解析的方式是非常非常好的。

但是, 我们偏偏要用 `lambda` 这个函数替代 `add(x)`, 如果看官和我一样这么偏执, 就可以:

```
>>> lam = lambda x:x+3
>>> n2 = []
>>> for i in numbers:
...     n2.append(lam(i))
...
>>> n2
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

这里的 `lam` 就相当于 `add(x)`, 请看官对应一下, 这一行 `lambda x:x+3` 就完成 `add(x)` 的三行 (还是两行?), 特别是最后返回值。还可以写这样的例子:

```
>>> g = lambda x,y:x+y # x+y, 并返回结果
>>> g(3,4)
```

```
7
>>> (lambda x:x**2)(4) #返回 4 的平方
16
```

通过上面例子，总结一下 lambda 函数的使用方法：

- 在 lambda 后面直接跟变量
- 变量后面是冒号
- 冒号后面是表达式，表达式计算结果就是本函数的返回值

为了简明扼要，用一个式子表示是必要的：

```
lambda arg1, arg2, ...argN : expression using arguments
```

要特别提醒看官：虽然 lambda 函数可以接收任意多个参数（包括可选参数）并且返回单个表达式的值，但是 lambda 函数不能包含命令，包含的表达式不能超过一个。不要试图向 lambda 函数中塞入太多的东西；如果你需要更复杂的东西，应该定义一个普通函数，然后想让它多长就多长。

就 lambda 而言，它并没有给程序带来性能上的提升，它带来的是代码的简洁。比如，要打印一个 list，里面依次是某个数字的 1 次方，二次方，三次方，四次方。用 lambda 可以这样做：

```
>>> lamb = [ lambda x:x,lambda x:x**2,lambda x:x**3,lambda x:x**4 ]
>>> for i in lamb:
...     print i(3),
...
3 9 27 81
```

lambda 做为一个单行的函数，在编程实践中，可以选择使用。

map

先看一个例子，还是上面讲述 lambda 的时候第一个例子，用 map 也能够实现：

```
>>> numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]    #把列表中每一项都加 3

>>> map(add,numbers)      #add(x) 是上面讲述的那个函数，但是这里只引用函数名称即可
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

>>> map(lambda x: x+3,numbers)  #用 lambda 当然可以啦
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

map() 是 Python 的一个内置函数，它的基本样式是：

```
map(func,seq)
```

func 是一个函数，seq 是一个序列对象。在执行的时候，序列对象中的每个元素，按照从左到右的顺序，依次被取出来，并塞入到 func 那个函数里面，并将 func 的返回值依次存到一个 list 中。

在应用中，map 的所能实现的，也可以用别的方式实现。比如：

```
>>> items = [1,2,3,4,5]
>>> squared = []
>>> for i in items:
...     squared.append(i**2)
...
>>> squared
[1, 4, 9, 16, 25]

>>> def sqr(x): return x**2
...
>>> map(sqr,items)
[1, 4, 9, 16, 25]

>>> map(lambda x: x**2, items)
[1, 4, 9, 16, 25]

>>> [ x**2 for x in items ]  #这个我最喜欢了，一般情况下速度足够快，而且可读性强
[1, 4, 9, 16, 25]
```

条条大路通罗马，以上方法，在编程中，自己根据需要来选用啦。

在以上感性认识的基础上，在来浏览有关 map() 的官方说明，能够更明白一些。

```
map(function, iterable, ...)
```

Apply function to every item of iterable and return a list of the results. If additional iterable arguments are passed, function must take that many arguments and is applied to the items from all iterables in parallel. If one iterable is shorter than another it is assumed to be extended with None items. If function is None, the identity function is assumed; if there are multiple arguments, map() returns a list consisting of tuples containing the corresponding items from all iterables (a kind of transpose operation). The iterable arguments may be a sequence or any iterable object; the result is always a list.

理解要点：

- 对 iterable 中的每个元素，依次应用 function 的方法（函数）（这本质上就是一个 for 循环）。
- 将所有结果返回一个 list。

- 如果参数很多，则对那些参数并行执行 function。

例如：

```
>>> lst1 = [1,2,3,4,5]
>>> lst2 = [6,7,8,9,0]
>>> map(lambda x,y: x+y, lst1,lst2)  #将两个列表中的对应项加起来，并返回一个结果列表
[7, 9, 11, 13, 5]
```

请看官注意了，上面这个例子如果用 for 循环来写，还不是很难，如果扩展一下，下面的例子用 for 来改写，就要小心了：

```
>>> lst1 = [1,2,3,4,5]
>>> lst2 = [6,7,8,9,0]
>>> lst3 = [7,8,9,2,1]
>>> map(lambda x,y,z: x+y+z, lst1,lst2,lst3)
[14, 17, 20, 15, 6]
```

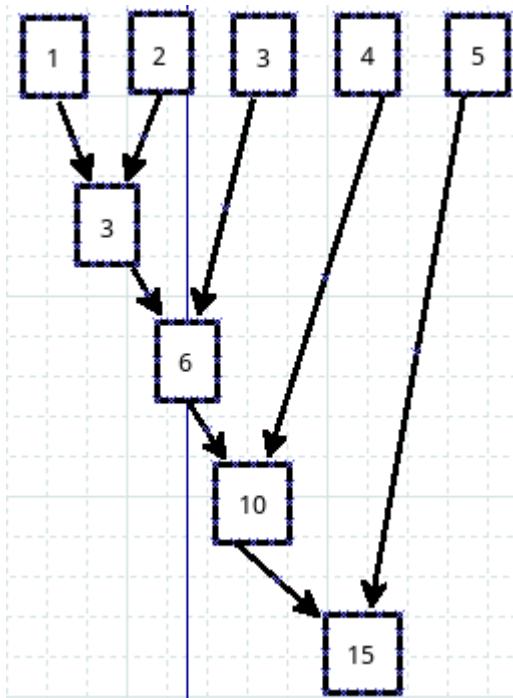
这才显示出 map 的简洁优雅。

reduce

直接看这个：

```
>>> reduce(lambda x,y: x+y,[1,2,3,4,5])
15
```

请看官仔细观察，是否能够看出是如何运算的呢？画一个图：



还记得 map 是怎么运算的吗？忘了？看代码：

```

>>> list1 = [1,2,3,4,5,6,7,8,9]
>>> list2 = [9,8,7,6,5,4,3,2,1]
>>> map(lambda x,y: x+y, list1,list2)
[10, 10, 10, 10, 10, 10, 10, 10, 10]
  
```

看官对比一下，就知道两个的区别了。原来map是上下运算，reduce 是横着逐个元素进行运算。

权威的解释来自官网：

`reduce(function, iterable[, initializer])`

Apply function of two arguments cumulatively to the items of iterable, from left to right, so as to reduce the iterable to a single value. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates $((((1+2)+3)+4)+5)$. The left argument, `x`, is the accumulated value and the right argument, `y`, is the update value from the iterable. If the optional initializer is present, it is placed before the items of the iterable in the calculation, and serves as a default when the iterable is empty. If initializer is not given and iterable contains only one item, the first item is returned. Roughly equivalent to:

```

def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        try:
            initializer = next(it)
        except StopIteration:
  
```

```

raise TypeError('reduce() of empty sequence with no initial value')
accum_value = initializer
for x in it:
    accum_value = function(accum_value, x)
return accum_value

```

如果用我们熟悉的 for 循环来做上面 reduce 的事情，可以这样做：

```

>>> lst = range(1,6)
>>> lst
[1, 2, 3, 4, 5]
>>> r = 0
>>> for i in range(len(lst)):
...     r += lst[i]
...
>>> r
15

```

for 普世的，reduce 是简洁的。

为了锻炼思维，看这么一个问题，有两个 list， $a = [3, 9, 8, 5, 2]$, $b=[1, 4, 9, 2, 6]$,计算： $a[0]b[0]+a[1]b[1]+\dots$ 的结果。

```

>>> a
[3, 9, 8, 5, 2]
>>> b
[1, 4, 9, 2, 6]

>>> zip(a,b)      #复习一下 zip，下面的方法中要用到
[(3, 1), (9, 4), (8, 9), (5, 2), (2, 6)]

>>> sum(x*y for x,y in zip(a,b))  #解析后直接求和
133

>>> new_list = [x*y for x,y in zip(a,b)]  #可以看做是上面方法的分布实施

>>> #这样解析也可以： new_tuple = (x*y for x,y in zip(a,b))
>>> new_list
[3, 36, 72, 10, 12]
>>> sum(new_list)  #或者:sum(new_tuple)
133

>>> reduce(lambda sum,(x,y): sum+x*y,zip(a,b),0)  #这个方法是在耍酷呢吗？
133

>>> from operator import add,mul      #耍酷的方法也不止一个

```

```
>>> reduce(add, map(mul, a, b))
133

>>> reduce(lambda x, y: x+y, map(lambda x, y: x*y, a, b)) #map,reduce,lambda 都齐全了，更酷吗？
133
```

最后，要特别提醒：如果读者使用的是 Python3，跟上面有点不一样，因为在 Python3 中，`reduce()` 已经从全局命名空间中移除，放到了 `functools` 模块中，如果要是用，需要用 `from functools import reduce` 引入之。

filter

`filter` 的中文含义是“过滤器”，在 Python 中，它就是起到了过滤器的作用。首先看官方说明：

`filter(function, iterable)`

Construct a list from those elements of iterable for which function returns true. iterable may be either a sequence, a container which supports iteration, or an iterator. If iterable is a string or a tuple, the result also has that type; otherwise it is always a list. If function is None, the identity function is assumed, that is, all elements of iterable that are false are removed.

Note that `filter(function, iterable)` is equivalent to `[item for item in iterable if function(item)]` if function is not None and `[item for item in iterable if item]` if function is None.

这次真的不翻译了（好像以往也没有怎么翻译呀），而且也不解释要点了。请列位务必自己阅读上面的文字，并且理解其含义。英语，无论怎么强调都是不过分的，哪怕是做乞丐，说两句英语，没准还可以讨到英镑美元呢。

通过下面代码体会：

```
>>> numbers = range(-5,5)
>>> numbers
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> filter(lambda x: x>0, numbers)
[1, 2, 3, 4]

>>> [x for x in numbers if x>0] #与上面那句等效
[1, 2, 3, 4]

>>> filter(lambda c: c!='i', 'qiwsir') #能不能对应上面文档说明那句话呢？
'qwsr'                                # “If iterable is a string or a tuple, the result also has that type;”
```

至此，介绍了几个函数，这些函数在对程序的性能提高上，并没有显著或者稳定预期，但是，在代码的简洁上，是有目共睹的。有时候是可以用来秀一秀，彰显 Python 的优雅和自己耍酷。如何用、怎么用，看你自己的喜好了。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

函数练习

已经学习了函数的基本知识，现在练习练习。完成下面练习的原则：

1. 请读者先根据自己的设想写下代码，然后运行调试，检查得到的结果是否正确
2. 我也给出参考代码，但是，参考代码并不是最终结果
3. 读者可以在上述基础上对代码进行完善
4. 如果读者愿意，可以将代码提交到 github 上，或者到我的 QQ 群(群号:26913719)中跟大家分享讨论

解一元二次方程

解一元二次方程，是初中数学中的基本知识，一般来讲解法有：公式法、因式分解法等。读者可以根据自己的理解，写一段求解一元二次方程的程序。

最简单的思路就是用公式法求解，这是普适法则（普世法则？普适是否等同于普世？）。

古巴比伦留下的陶片显示，在大约公元前 2000 年 (2000 BC) 古巴比伦的数学家就能解一元二次方程了。在大約公元前 480 年，中國人已经使用配方法求得了二次方程的正根，但是并没有提出通用的求解方法。公元前 300 年左右，欧几里得提出了一种更抽象的几何方法求解二次方程。

7 世纪印度的婆罗摩笈多 (Brahmagupta) 是第一位懂得用使用代数方程，它同时容许有正负数的根。

11 世纪阿拉伯的花拉子密 独立地发展了一套公式以求方程的正数解。亚伯拉罕 · 巴希亚 (亦以拉丁文名字萨瓦索达著称) 在他的著作 Liber embadorum 中，首次将完整的一元二次方程解法传入欧洲。(源自《维基百科》)

参考代码：

```
#!/usr/bin/env Python
# coding=utf-8

"""

solving a quadratic equation

"""

from __future__ import division
import math

def quadratic_equation(a,b,c):
    delta = b*b - 4*a*c
```

```

if delta<0:
    return False
elif delta==0:
    return -(b/(2*a))
else:
    sqrt_delta = math.sqrt(delta)
    x1 = (-b + sqrt_delta)/(2*a)
    x2 = (-b - sqrt_delta)/(2*a)
    return x1, x2

if __name__ == "__main__":
    print "a quadratic equation: x^2 + 2x + 1 = 0"
    coefficients = (1, 2, 1)
    roots = quadratic_equation(*coefficients)
    if roots:
        print "the result is:",roots
    else:
        print "this equation has no solution."

```

保存为 20501.py，并运行之：

```

$ python 20501.py
a quadratic equation: x^2 + 2x + 1 = 0
the result is: -1.0

```

能够正常运行，求解方程。

但是，如果再认真思考，发现上述代码是有很大改进空间的。至少我发现：

- 如果不小心将第一个系数(a)的值输入了 0，程序肯定会报错。如何避免之？要记住，任何人的输入都是不可靠的。
- 结果貌似只能是小数，这在某些情况下是近似值，能不能得到以分数形式表示的精确结果呢？
- 复数，Python 是可以表示复数的，如果 $\text{delta}<0$ ，是不是写成复数更好，毕竟我是学过高中数学的。

读者是否还有其它改进呢？你能不能进行改进，然后跟我和其他朋友一起来分享你的成就呢？

至少要完成上述改进，可能需要其它的有关 Python 知识，甚至于前面没有介绍。这都不要紧，掌握了基本知识之后，在编程的过程中，就要不断发挥 google 的优势，让她帮助你找寻完成任务的工具。

Python 是一个开发的语言，很多大牛人都写了一些工具，让别人使用，减轻了后人的劳动负担。这就是所谓的第三方模块。虽然 Python 中已经有一些“自带电池”，即默认安装的，比如上面程序中用到的 math，但是我们还嫌不够。于是又很多第三方的模块来专门解决某个问题。比如这个解方程问题，就可以使用 SymPy(www.sympy.org)来解决，当然 NumPy 也是非常强悍的工具。

统计考试成绩

每次考试之后，教师都要统计考试成绩，一般包括：平均分，对所有人按成绩从高到低排队，谁成绩最好，谁成绩最差。还有其它的统计项，暂且不做了。只统计这几项吧。下面的任务就是读者转动脑筋，思考如何用程序实现上面的统计。为了简化，以字典形式表示考试成绩记录，例如：`{"zhangsan":90, "lisi":78, "wangermazi":39}`，当然，也许不止这三项，可能还有，每个老师所处理的内容稍有不同，因此字典里的键值对也不一样。

怎么做？

有几种可能要考虑到：

- 最高分或者最低分，可能有人并列。
- 要实现不同长度的字典作为输入值。
- 输出结果中，除了平均分，其它的都要有姓名和分数两项，否则都匿名了，怎么刺激学渣，表扬学霸呢？

不管你是学渣还是学霸，都能学好 Python。请思考后敲代码调试你的程序，调试之后再阅读下文。

参考代码：

```
#!/usr/bin/env Python
# coding=utf-8
"""

统计考试成绩
"""

from __future__ import division

def average_score(scores):
    """
    统计平均分.
    """

    score_values = scores.values()
    sum_scores = sum(score_values)
    average = sum_scores/len(score_values)
    return average

def sorted_score(scores):
    """
    对成绩从高到低排队.
    """

    score_lst = [(scores[k],k) for k in scores]
    sort_lst = sorted(score_lst, reverse=True)
    return [(i[1], i[0]) for i in sort_lst]
```

```

def max_score(scores):
    """
    成绩最高的姓名和分数.
    """

    lst = sorted_score(scores)  #引用分数排序的函数 sorted_score
    max_score = lst[0][1]
    return [(i[0],i[1]) for i in lst if i[1]==max_score]

def min_score(scores):
    """
    成绩最低的姓名和分数.
    """

    lst = sorted_score(scores)
    min_score = lst[len(lst)-1][1]
    return [(i[0],i[1]) for i in lst if i[1]==min_score]

if __name__ == "__main__":
    examine_scores = {"google":98, "facebook":99, "baidu":52, "alibaba":80, "yahoo":49, "IBM":70, "android":76, "apple":99, "amazon":99}

    ave = average_score(examine_scores)
    print "the average score is: ",ave  #平均分

    sor = sorted_score(examine_scores)
    print "list of the scores: ",sor  #成绩表

    xueba = max_score(examine_scores)
    print "Xueba is: ",xueba  #学霸们

    xuezha = min_score(examine_scores)
    print "Xuzha is: ",xuezha  #学渣们

```

保存为 20502.py，然后运行：

```

$ python 20502.py
the average score is: 80.2222222222
list of the scores: [('facebook', 99), ('apple', 99), ('amazon', 99), ('google', 98), ('alibaba', 80), ('android', 76), ('IBM', 70), ('ba
Xueba is: [('facebook', 99), ('apple', 99), ('amazon', 99)]
Xuzha is: [('yahoo', 49)]

```

貌似结果还不错。不过，还有改进余地，看看现实，就感觉不怎么友好了。看官能不能优化一下？当然，里面的函数也不一定是最好的方法，你也可以修改优化。期盼能够在我上面公布的途径中交流一二。

找素数

这是一个比较常见的题目。我们姑且将范围缩小一下，找出 100 以内的素数吧。

还是按照前面的管理，读者先做，然后我提供参考代码，然后自行优化。

質數（Prime number），又称素数，指在大於1的自然数中，除了1和此整数自身外，無法被其他自然数整除的数（也可定義為只有1和本身两个因数的数）。

哥德巴赫猜想是数论中存在最久的未解问题之一。这个猜想最早出现在 1742 年普鲁士人克里斯蒂安·哥德巴赫与瑞士数学家莱昂哈德·欧拉的通信中。用现代的数学语言，哥德巴赫猜想可以陈述为：“任一大于 2 的偶数，都可表示成两个质数之和。”。哥德巴赫猜想在提出后的很长一段时间内毫无进展，直到二十世纪二十年代，数学家从组合数学与解析数论两方面分别提出了解决的思路，并在其后的半个世纪里取得了一系列突破。目前最好的结果是陈景润在 1973 年发表的陈氏定理（也被称为“1+2”）。（源自《维基百科》）

对这个练习，我的思路是先做一个函数，用它来判断某个整数是否是素数。然后循环即可。参考代码：

```
#!/usr/bin/env Python
# coding=utf-8

"""
寻找素数
"""

import math

def is_prime(n):
    """
    判断一个数是否是素数
    """

    if n <= 1:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

if __name__ == "__main__":
    primes = [i for i in range(2,100) if is_prime(i)]  #从 2 开始，因为 1 显然不是质数
    print primes
```

代码保存后运行：

```
$ python 20503.py  
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

打印出了 100 以内的质数。

还是前面的观点，这个程序你或许也发现了需要进一步优化的地方，那就太好了。另外，关于判断质数的方法，还有好多种，读者可以自己创造或者网上搜索一些，拓展思路。

编写函数的注意事项

编写函数，在开发实践中是非常必要和常见的，一般情况，你写的函数应该是：

1. 尽量不要使用全局变量。
2. 如果参数是可变类型数据，在函数内，不要修改它。
3. 每个函数的功能和目标要单纯，不要试图一个函数做很多事情。
4. 函数的代码行数尽量少。
5. 函数的独立性越强越好，不要跟其它的外部东西产生关联。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

HTML



TP



类



unity



HTML



类(1)

类，这个词如果是你第一次听到，把它作为一个单独的名词，总感觉怪怪的，因为在汉语体系中，很常见的是说“鸟类”、“人类”等词语，而单独说“类”，总感觉前面缺点修饰成分。其实，它对应的是英文单词 class，“类”是这个 class 翻译过来的，你就把它作为一个翻译术语吧。

除了“类”这个术语，从现在开始，还要经常提到一个 OOP，即面向对象编程（或者“面向对象程序设计”）。

为了理解类和 OOP，需要对一些枯燥的名词有了解。

术语

必须了解这些术语的基本含义，因为后面经常用到。下面的术语定义均来自维基百科。

问题空间

定义：

问题空间是问题解决者对一个问题所达到的全部认识状态，它是由问题解决者利用问题所包含的信息和已贮存的信息主动地构成的。

一个问题一般有下面三个方面来定义：

- 初始状态——开始时的不完全的信息或令人不满意的状况；
- 目标状态——你希望获得的信息或状态；
- 操作——为了从初始状态迈向目标状态，你可能采取的步骤。

这三个部分加在一起定义了问题空间（problem space）。

对象

定义：

对象（object），台湾译作物件，是面向对象（Object Oriented）中的术语，既表示客观世界问题空间（Namespace）中的某个具体的事物，又表示软件系统解空间中的基本元素。

把 object 翻译为“对象”，是比较抽象的。因此，有人认为，不如翻译为“物件”更好。因为“物件”让人感到一种具体的东西。

这种看法在某些语言中是非常适合的。但是，在 Python 中，则无所谓，不管怎样，Python 中的一切都是对象，不管是字符串、函数、模块还是类，都是对象。“万物皆对象”。

都是对象有什么优势吗？太有了。这说明 Python 天生就是 OOP 的。也说明，Python 中的所有东西，都能够进行拼凑组合应用，因为对象就是可以拼凑组合应用的。

对于对象这个东西，OOP 大师 Grandy Booch 的定义，应该是权威的，相关定义的内容包括：

- 对象：一个对象有自己的状态、行为和唯一的标识；所有相同类型的对象所具有的结构和行为在他们共同的类中被定义。
- 状态（state）：包括这个对象已有的属性（通常是类里面已经定义好的）在加上对象具有的当前属性值（这些属性往往是动态的）
- 行为（behavior）：是指一个对象如何影响外界及被外界影响，表现为对象自身状态的改变和信息的传递。
- 标识（identity）：是指一个对象所具有的区别于所有其它对象的属性。（本质上指内存中所创建的对象的地址）

大师的话的确有水平，听起来非常高深。不过，初学者可能理解起来就有点麻烦了。我就把大师的话化简一下，但是化简了之后可能在严谨性上就不足了，我想对于初学者来讲，应该是影响不很大的。随着学习和时间的深入，就更能理解大师的严谨描述了。

简化之，对象应该具有属性（就是上面的状态，因为属性更常用）、方法（就是上面的行为，方法通常被使用）和标识。因为标识是内存中自动完成的，所以，平时不用怎么管理它。主要就是属性和方法。

任何一个对象都要包括这两部分：属性（是什么）和方法（能做什么）。

面向对象

定义：

面向对象程序设计（英语：Object-oriented programming，缩写：OOP）是一种程序设计范型，同时也是一种程序开发的方法。对象指的是类的实例。它将对象作为程序的基本单元，将程序和数据封装其中，以提高软件的重用性、灵活性和扩展性。

面向对象程序设计可以看作一种在程序中包含各种独立而又互相调用的对象的思想，这与传统的思想刚好相反：传统的程序设计主张将程序看作一系列函数的集合，或者直接就是一系列对电脑下达的指令。面向对象程序

设计中的每一个对象都应该能够接受数据、处理数据并将数据传达给其它对象，因此它们都可以被看作一个小型的“机器”，即对象。

目前已经被证实的是，面向对象程序设计推广了程序的灵活性和可维护性，并且在大型项目设计中广为应用。此外，支持者声称面向对象程序设计要比以往的做法更加便于学习，因为它能够让人们更简单地设计并维护程序，使得程序更加便于分析、设计、理解。反对者在某些领域对此予以否认。

当我们提到面向对象的时候，它不仅指一种程序设计方法。它更多意义上是一种程序开发方式。在这一方面，我们必须了解更多关于面向对象系统分析和面向对象设计（Object Oriented Design，简称 OOD）方面的知识。

下面再引用一段来自维基百科中关于 OOP 的历史。

面向对象程序设计的雏形，早在 1960 年的 Simula 语言中即可发现，当时的程序设计领域正面临着一种危机：在软硬件环境逐渐复杂的情况下，软件如何得到良好的维护？面向对象程序设计在某种程度上通过强调可重复性解决了这一问题。20 世纪 70 年代的 Smalltalk 语言在面向对象方面堪称经典——以至于 30 年后的今天依然将这一语言视为面向对象语言的基础。

计算机科学中对象和实例概念的最早萌芽可以追溯到麻省理工学院的 PDP-1 系统。这一系统大概是最早的基于容量架构（capability based architecture）的实际系统。另外 1963 年 Ivan Sutherland 的 Sketchpad 应用中也蕴含了同样的思想。对象作为编程实体最早是于 1960 年代由 Simula 67 语言引入思维。Simula 这一语言是奥利-约翰·达尔和克利斯登·奈加特在挪威奥斯陆计算机中心为模拟环境而设计的。（据说，他们是为了模拟船只而设计的这种语言，并且对不同船只间属性的相互影响感兴趣。他们将不同的船只归纳为不同的类，而每一个对象，基于它的类，可以定义它自己的属性和行为。）这种办法是分析式程序的最早概念体现。在分析式程序中，我们将真实世界的对象映射到抽象的对象，这叫做“模拟”。Simula 不仅引入了“类”的概念，还应用了实例这一思想——这可能是这些概念的最早应用。

20 世纪 70 年代施乐 PARC 研究所发明的 Smalltalk 语言将面向对象程序设计的概念定义为，在基础运算中，对对象和消息的广泛应用。Smalltalk 的创建者深受 Simula 67 的主要思想影响，但 Smalltalk 中的对象是完全动态的——它们可以被创建、修改并销毁，这与 Simula 中的静态对象有所区别。此外，Smalltalk 还引入了继承性的思想，它因此一举超越了不可创建实例的程序设计模型和不具备继承性的 Simula。此外，Simula 67 的思想亦被应用在许多不同的语言，如 Lisp、Pascal。

面向对象程序设计在 80 年代成为了一种主导思想，这主要应归功于 C++——C 语言的扩充版。在图形用户界面（GUI）日渐崛起的情况下，面向对象程序设计很好地适应了潮流。GUI 和面向对象程序设计的紧密关联在 Mac OS X 中可见一斑。Mac OS X 是由 Objective-C 语言写成的，这一语言是一个仿 Smalltalk 的 C 语言扩充版。面向对象程序设计的思想也使事件处理式的程序设计更加广泛被应用（虽然这一概念并非仅存在于面向对象程序设计）。一种说法是，GUI 的引入极大地推动了面向对象程序设计的发展。

苏黎世联邦理工学院的尼克劳斯·维尔特和他的同事们对抽象数据和模块化程序设计进行了研究。Modula-2 将这些都包括了进去，而 Oberon 则包括了一种特殊的面向对象方法——不同于 Smalltalk 与 C++。

面向对象的特性也被加入了当时较为流行的语言：Ada、BASIC、Lisp、Fortran、Pascal 以及种种。由于这些语言最初并没有面向对象的设计，故而这种糅合常常会导致兼容性和维护性的问题。与之相反的是，“纯正的”面向对象语言却缺乏一些程序员们赖以生存的特性。在这一大环境下，开发新的语言成为了当务之急。作为先行者，Eiffel 成功地解决了这些问题，并成为了当时较受欢迎的语言。

在过去的几年中，Java 语言成为了广为应用的语言，除了它与 C 和 C++ 语法上的近似性。Java 的可移植性是它的成功中不可磨灭的一步，因为这一特性，已吸引了庞大的程序员群的投入。

在最近的计算机语言发展中，一些既支持面向对象程序设计，又支持面向过程程序设计的语言悄然浮出水面。它们中的佼佼者有 Python、Ruby 等等。

正如面向过程程序设计使得结构化程序设计的技术得以提升，现代的面向对象程序设计方法使得对设计模式的用途、契约式设计和建模语言（如 UML）技术也得到了一定提升。

列位看官，当您阅读到这句话的时候，我就姑且认为您已经对面向对象有了一个模糊的认识了。那么，类和 OOP 有什么关系呢？

类

定义：

在面向对象程式设计，类（class）是一种面向对象计算机编程语言的构造，是创建对象的蓝图，描述了所创建的对象共同的属性和方法。

类的更严格的定义是由某种特定的元数据所组成的内聚的包。它描述了一些对象的行为规则，而这些对象就被称为该类的实例。类有接口和结构。接口描述了如何通过方法与类及其实例互操作，而结构描述了一个实例中数据如何划分为多个属性。类是与某个层的对象的最具体的类型。类还可以有运行时表示形式（元对象），它为操作与类相关的元数据提供了运行时支持。

支持类的编程语言在支持与类相关的各种特性方面都多多少少有一些微妙的差异。大多数都支持不同形式的类继承。许多语言还支持提供封装性的特性，比如访问修饰符。类的出现，为面向对象编程的三个最重要的特性（封装性，继承性，多态性），提供了实现的手段。

看到这里，看官或许有一个认识，要 OOP 编程，就得用到类。可以这么说，虽然不是很严格。但是，反过来就不能说了。不是说用了类就一定是 OOP。

编写类

首先要明确，类是对某一群具有同样属性和方法的对象的抽象。比如这个世界上有很多长翅膀并且会飞的生物，于是聪明的人们就将它们统一称为“鸟”——这就是一个类，虽然它也可以称作“鸟类”。

还是以美女为例子，因为这个例子不仅能阅读本课程不犯困，还能兴趣昂然。

要定义类，就要抽象，找出共同的方面。

```
class 美女:    #用 class 来声明，后面定义的是一个类
    pass
```

好，现在就从这里开始，编写一个类，不过这次我们暂时不用 Python，而是用伪代码，当然，这个代码跟 Python 相去甚远。如下：

```
class 美女:
    胸围 = 90
    腰围 = 58
    臀围 = 83
    皮肤 = white
    唱歌()
    做饭()
```

定义了一个名称为“美女”的类，其中我约定，没有括号的是属性，带有括号的是方法。这个类仅仅是对美女的通常抽象，并不是某个具体美女。

对于一个具体的美女，比如前面提到的苍老师或者王美女，她们都是上面所定义的“美女”那个类的具体化，这在编程中称为“美女类”的实例。

```
王美女 = 美女()
```

我用这样一种表达方式，就是将“美女类”实例化了，对“王美女”这个实例，就可以具体化一些属性，比如胸围；还可以具体实施一些方法，比如做饭。通常可以用这样一种方式表示：

```
a = 王美女.胸围
```

用点号 . 的方式，表示王美女胸围的属性，得到的变量 a 就是 90. 另外，还可以通过这种方式给属性赋值，比如

```
王美女.皮肤 = black
```

这样，这个实例（王美女）的皮肤就是黑色的了。

通过实例，也可以访问某个方法，比如：

王美女.做饭()

这就是在执行一个方法，让王美女这个实例做饭。现在也比较好理解了，只有一个具体的实例才能做饭。

至此，你是否对类和实例，类的属性和方法有初步理解了呢？

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

类(2)

现在开始不用伪代码了，用真正的 Python 代码来理解类。当然，例子还是要用读者感兴趣的例子。

新式类和旧式类

因为 Python 是一个不断发展的高级语言（似乎别的语言是不断发展的，甚至于自然语言也是），导致了在 Python2.x 的版本中，有“新式类”和“旧式类（也叫做经典类）”之分。新式类是 Python2.2 引进的，在此后的版本中，我们一般用的都是新式类。本着知其然还要知其所以然的目的，简单回顾一下两者的差别。

```
>>> class AA:  
...     pass  
...
```

这是定义了一个非常简单的类，而且是旧式类。至于如何定义类，下面会详细说明。读者姑且囫囵吞枣似的的认同我刚才建立的名为 `AA` 的类，为了简单，这个类内部什么也不做，就是用 `pass` 一带而过。但不管怎样，是一个类，而且是一个旧式类（或曰经典类）

然后，将这个类实例化（还记得上节中实例化吗？对，就是那个王美女干的事情）：

```
>>> aa = AA()
```

不要忘记，实例化的时候，类的名称后面有一对括号。接下来做如下操作：

```
>>> type(AA)  
<type 'classobj'>  
>>> aa.__class__  
<class '__main__.AA' at 0xb71f017c>  
>>> type(aa)  
<type 'instance'>
```

解读一下上面含义：

- `type(AA)`：查看类 `AA` 的类型，返回的是 `'classobj'`
- `aa.__class__`：`aa` 是一个实例，也是一个对象，每个对象都有 `__class__` 属性，用于显示它的类型。这里返回的结果是 `<class '__main__.AA' at 0xb71f017c>`，从这个结果中可以读出的信息是，`aa` 是类 `AA` 的实例，并且类 `AA` 在内存中的地址是 `0xb71f017c`。
- `type(aa)`：是要看实例 `aa` 的类型，它显示的结果是 `'instance'`，意思是告诉我们它的类型是一个实例。

在这里是不是有点感觉不和谐呢？`aa.__class__` 和 `type(aa)` 都可以查看对象类型，但是它们居然显示不一样的结果。比如，查看这个对象：

```
>>> a = 7
>>> a.__class__
<type 'int'>
>>> type(a)
<type 'int'>
```

别忘记了，前面提到过的“万物皆对象”，那么一个整数7也是对象，用两种方式查看，返回的结果一样。为什么到类（严格讲是旧式类）这里，居然返回不一样呢？太不和谐了。

于是乎，就有了新式类，从 Python2.2 开始，变成这样了：

```
>>> class BB(object):
...     pass
...
>>> bb = BB()

>>> bb.__class__
<class '__main__.BB'>
>>> type(bb)
<class '__main__.BB'>
```

终于把两者统一起来了，世界和谐了。

这就是新式类和旧式类的不同。

当然，不同点绝非仅仅于此，这里只不过提到一个现在能够理解的不同罢了。另外的不同还在于两者对于多重继承的查找和调用方法不同，旧式类是深度优先，新式类是广度优先。可以先不理解，后面会碰到的。

不管是新式类、还是旧式类，都可以通过这样的方法查看它们在内存中的存储空间信息

```
>>> print aa
<__main__.AA instance at 0xb71efd4c>

>>> print bb
<__main__.BB object at 0xb71efe6c>
```

分别告诉了我们两个实例是基于谁生成的，不过还是稍有区别。

知道了旧式类和新式类，那么下面的所有内容，就都是对新式类而言。“喜新厌旧”不是编程经常干的事情吗？所以，旧式类就不是我们讨论的内容了。

还要注意，如果你用的是 Python3，就不用为新式类和旧式类而担心了，因为在 Python3 中压根儿就没有这个问题存在。

如何定义新式类呢？

第一种定义方法，就是如同前面那样：

```
>>> class BB(object):
...     pass
...
...
```

跟旧式类的区别就在于类的名字后面跟上 `(object)`，这其实是一种名为“继承”的类的操作，当前的类 `BB` 是以类 `object` 为上级的（`object` 被称为父类），即 `BB` 是继承自类 `object` 的新类。在 Python3 中，所有的类自然地都是类 `object` 的子类，就不用彰显出继承关系了。对了，这里说的有点让读者糊涂，因为冒出来了“继承”、“父类”、“子类”，不用着急，继续向下看。下面精彩，并且能解惑。

第二种定义方法，在类的前面写上这么一句：`__metaclass__ == type`，然后定义类的时候，就不需要在名字后面写 `(object)` 了。

```
>>> __metaclass__ = type
>>> class CC:
...     pass
...
...
>>> cc = CC()
>>> cc.__class__
<class '__main__.CC'>
>>> type(cc)
<class '__main__.CC'>
```

两种方法，任你选用，没有优劣之分。

创建类

因为在一般情况下，一个类都不是两三行能搞定的。所以，下面可能很少使用交互模式了，因为那样一旦有一点错误，就前功尽弃。我改用编辑界面。你用什么工具编辑？Python 自带一个 IDE，可以使用。我习惯用 vim。你用你习惯的工具即可。如果你没有别的工具，就用安装 Python 是自带的那个 IDE。

```
#!/usr/bin/env Python
# coding=utf-8

__metaclass__ = type
```

```

class Person:
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

    def color(self, color):
        print "%s is %s" % (self.name, color)

```

上面定义的是一个比较常见的类，一般情况下，都是这样子的。下面对这个“大众脸”的类一一解释。

新式类

`__metaclass__ = type`，意味着下面的类是新式类。

定义类

`class Person`，这是在声明创建一个名为"Person"的类。类的名称一般用大写字母开头，这是惯例。如果名称是两个单词，那么两个单词的首字母都要大写，例如 `class HotPerson`，这种命名方法有一个形象的名字，叫做“驼峰式命名”。当然，如果故意不遵循此惯例，也未尝不可，但是，会给别人阅读乃至自己以后阅读带来麻烦，不要忘记“代码通常是给人看的，只是偶尔让机器执行”。既然大家都是靠右走的，你就别非要在路中间睡觉了。

接下来，分别以缩进表示的，就是这个类的内容了。其实那些东西看起来并不陌生，你一眼就认出它们了——就是已经学习过的函数。没错，它们就是函数。不过，很多程序员喜欢把类里面的函数叫做“方法”。是的，就是上节中说到的对象的“方法”。我也看到有人撰文专门分析了“方法”和“函数”的区别。但是，我倒是认为这不重要，重要的是类的中所谓“方法”和前面的函数，在数学角度看，丝毫没有区别。所以，你尽可以称之为函数。当然，听到有人说方法，也不要诧异和糊涂。它们本质是一样的。

需要再次提醒，函数的命名方法是以 `def` 发起，并且函数名称首字母不要用大写，可以使用 `aa_bb` 的样式，也可以使用 `aaBb` 的样式，一切看你的习惯了。

不过，要注意的是，类中的函数（方法）的参数跟以往的参数样式有区别，那就是每个函数必须包括 `self` 参数，并且作为默认的第一个参数。这是需要注意的地方。至于它的用途，继续学习即可知道。

初始化

`def __init__`，这个函数是一个比较特殊的，并且有一个名字，叫做**初始化函数**（注意，很多教材和资料中，把它叫做**构造函数**，这种说法貌似没有错误，但是一来从字面意义上讲，它对应的含义是**初始化**，二来在 Python 中它的作用和其它语言比如 java 中的**构造函数**还不完全一样，因为还有一个 `__new__` 的函数，是真正地**构造**。所以，在本教程中，我称之为**初始化函数**）。它是以两个下划线开始，然后是 `init`，最后以两个下划线结束。

所谓**初始化**，就是让类有一个基本的面貌，而不是空空如也。做很多事情，都要**初始化**，让事情有一个具体的起点状态。比如你要喝水，必须先**初始化**杯子里里面有水。在 Python 的类中，**初始化**就担负着类似的工作。这个工作是在类被实例化的时候就执行这个函数，从而将初始化的一些属性可以放到这个函数里面。

此例子中的**初始化函数**，就意味着实例化的时候，要给参数 `name` 提供一个值，作为类**初始化**的内容。通俗点说，就是在这个类被实例化的同时，要通过 `name` 参数传一个值，这个值被一开始就写入了类和实例中，成为了类和实例的一个属性。比如：

```
girl = Person('wangguniang')
```

`girl` 是一个实例对象，就如同前面所说的一样，它有属性和方法。这里仅说属性吧。当通过上面的方式实例化后，就自动执行了**初始化函数**，让实例 `girl` 就具有了 `name` 属性。

```
print girl.name
```

执行这句话的结果是打印出 `wangguniang`。

这就是**初始化**的功能。简而言之，通过**初始化函数**，确定了这个实例（类）的“**基本属性**”（实例是什么样子的）。比如上面的实例化之后，就确立了实例 `girl` 的 `name` 是“`wangguniang`”。

初始化函数，就是一个函数，所以，它的参数设置，也符合前面学过的函数参数设置规范。比如

```
def __init__(self,*args):
    pass
```

这种类型的参数：`*args` 和前面讲述函数参数一样，就不多说了。忘了的看官，请去复习。但是，`self` 这个参数是必须的。

很多时候，并不是每次都要从外面传入数据，有时候会把**初始化函数**的某些参数设置默认值，如果没有新的数据传入，就应用这些默认值。比如：

```
class Person:
    def __init__(self, name, lang="golang", website="www.google.com"):
        self.name = name
        self.lang = lang
```

```

self.website = website
self.email = "qiwsir@gmail.com"

laoqi = Person("LaoQi")
info = Person("qiwsir", lang="python", website="qiwsir.github.io")

print "laoqi.name=",laoqi.name
print "info.name=",info.name
print "-----"
print "laoqi.lang=",laoqi.lang
print "info.lang=",info.lang
print "-----"
print "laoqi.website=",laoqi.website
print "info.website=",info.website

#运行结果

laoqi.name= LaoQi
info.name= qiwsir
-----
laoqi.lang= golang
info.lang= python
-----
laoqi.website= www.google.com
info.website= qiwsir.github.io

```

在编程界，有这样一句话，说“类是实例工厂”，什么意思呢？工厂是干什么的？生产物品，比如生产电脑。一个工厂可以生产好多电脑。那么，类，就能“生产”好多实例，所以，它是“工厂”。比如上面例子中，就有两个实例。

函数（方法）

还是回到本节开头的那个类。构造函数下面的两个函数： def getName(self) , def color(self, color) ，这两个函数和前面的初始化函数有共同的地方，即都是以 self 作为第一个参数。

```

def getName(self):
    return self.name

```

这个函数中的作用就是返回在初始化时得到的值。

```

girl = Person('wangguniang')
name = girl.getName()

```

`girl.getName()` 就是调用实例 `girl` 的方法。调用该方法的时候特别注意，方法名后面的括号不可少，并且括号中不要写参数，在类中的 `getName(self)` 函数第一个参数 `self` 是默认的，当类实例化之后，调用此函数的时候，第一个参数不需要赋值。那么，变量 `name` 的最终结果就是 `name = "wangguniang"`。

同样道理，对于方法：

```
def color(self, color):
    print "%s is %s" % (self.name, color)
```

也是在实例化之后调用：

```
girl.color("white")
```

这也是在执行实例化方法，只是由于类中的该方法有两个参数，除了默认的 `self` 之外，还有一个 `color`，所以，在调用这个方法的时候，要为后面那个参数传值了。

至此，已经将这个典型的类和调用方法分解完毕，把全部代码完整贴出，请读者在从头到尾看看，是否理解了每个部分的含义：

```
#!/usr/bin/env Python
# coding=utf-8

__metaclass__ = type      #新式类

class Person:            #创建类
    def __init__(self, name): #构造函数
        self.name = name

    def getName(self):      #类中的方法（函数）
        return self.name

    def color(self, color):
        print "%s is %s" % (self.name, color)

girl = Person('wangguniang')  #实例化
name = girl.getName()        #调用方法（函数）
print "the person's name is: ", name
girl.color("white")          #调用方法（函数）

print "-----"
print girl.name              #实例的属性
```

保存后，运行得到如下结果：

```
$ python 20701.py
the person's name is: wangguniang
wangguniang is white
-----
wangguniang
```

类和实例

有必要总结一下类和实例的关系：

- “类提供默认行为，是实例的工厂”（源自 Learning Python），这句话非常经典，一下道破了类和实例的关系。所谓工厂，就是可以用同一个模子做出很多具体的产品。类就是那个模子，实例就是具体的产品。所以，实例是程序处理的实际对象。
- 类是由一些语句组成，但是实例，是通过调用类生成，每次调用一个类，就得到这个类的新的实例。
- 对于类的：class Person，class 是一个可执行的语句。如果执行，就得到了一个类对象，并且将这个类对象赋值给对象名（比如 Person）。

也许上述比较还不足以让看官理解类和实例，没关系，继续学习，在前进中排除疑惑。

self 的作用

类里面的函数，第一个参数是 self，但是在实例化的时候，似乎没有这个参数什么事儿，那么 self 是干什么的呢？

self 是一个很神奇的参数。

在 Person 实例化的过程中 girl = Person("wangguniang")，字符串"wangguniang"通过初始化函数（__init__()）的参数已经存入到内存中，并且以 Person 类型的面貌存在，组成了一个对象，这个对象和变量 girl 建立引用关系。这个过程也可以说成这些数据附加到一个实例上。这样就能够以 object.attribute 的形式，在程序中任何地方调用某个数据，例如上面的程序中以 girl.name 的方式得到 "wangguniang"。这种调用方式，在类和实例中经常使用，点号“.”后面的称之为类或者实例的属性。

这是在程序中，并且是在类的外面。如果在类的里面，想在某个地方使用实例化所传入的数据 ("wangguniang")，怎么办？

在类内部，就是将所有传入的数据都赋给一个变量，通常这个变量的名字是 self。注意，这是习惯，而且是共识，所以，看官不要另外取别的名字了。

在初始化函数中的第一个参数 `self`，就是起到了这个作用——接收实例化过程中传入的所有数据，这些数据是初始化函数后面的参数导入的。显然，`self` 应该就是一个实例（准确说法是应用实例），因为它所对应的就是具体数据。

如果将上面的类稍加修改，看看效果：

```
#!/usr/bin/env Python
# coding=utf-8

__metaclass__ = type

class Person:
    def __init__(self, name):
        self.name = name
        print self      #新增
        print type(self)  #新增
```

其它部分省略。当初始化的时候，就首先要运行构造函数，同时就打印新增的两条。结果是：

```
<__main__.Person object at 0xb7282cec>
<class '__main__.Person'>
```

证实了推理。`self` 就是一个实例（准确说是实例的引用变量）。

`self` 这个实例跟前面说的那个 `girl` 所引用的实例对象一样，也有属性。那么，接下来就规定其属性和属性对应的数据。上面代码中：

```
self.name = name
```

就是规定了 `self` 实例的一个属性，这个属性的名字也叫做 `name`，这个属性的值等于初始化函数的参数 `name` 所导入的数据。注意，`self.name` 中的 `name` 和初始化函数的参数 `name` 没有任何关系，它们两个一样，只不过是一种起巧合（经常巧合，其实是为了省事和以后识别方便，故意让它们巧合。），或者说是写代码的人懒惰，不想另外取名字而已，无他。当然，如果写成 `self.xxxxoo = name`，也是可以的。

其实，从效果的角度来理解，这么理解更简化：类的实例 `girl` 对应着 `self`，`girl` 通过 `self` 导入实例属性的所有数据。

当然，`self` 的属性数据，也不一定非得是由参数传入的，也可以在构造函数中自己设定。比如：

```
#!/usr/bin/env Python
#coding:utf-8

__metaclass__ = type
```

```
class Person:  
    def __init__(self, name):  
        self.name = name  
        self.email = "qiwsir@gmail.com" #这个属性不是通过参数传入的  
  
info = Person("qiwsir") #换个字符串和实例化变量  
print "info.name=",info.name  
print "info.email=",info.email #info 通过 self 建立实例，并导入实例属性数据
```

运行结果

```
info.name= qiwsir  
info.email= qiwsir@gmail.com #打印结果
```

通过这个例子，其实让我们拓展了对 `self` 的认识，也就是它不仅仅是为了在类内部传递参数导入的数据，还能在初始化函数中，通过 `self.attribute` 的方式，规定 `self` 实例对象的属性，这个属性也是类实例化对象的属性，即做为类通过初始化函数初始化后所具有的属性。所以在实例 `info` 中，通过 `info.email` 同样能够得到该属性的数据。在这里，就可以把 `self` 形象地理解为“内外兼修”了。或者按照前面所提到的，将 `info` 和 `self` 对应起来，`self` 主内，`info` 主外。

怎么样？是不是明白了类的奥妙？

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

类(3)

在上一节中，对类有了基本的或者说是模糊的认识，为了能够对类有更深刻的认识，本节要深入到一些细节。

类属性和实例属性

正如上节的案例中，一个类实例化后，实例是一个对象，有属性。同样，类也是一个对象，它也有属性。

```
>>> class A(object):
...     x = 7
...
...
```

在交互模式下，定义一个很简单的类（注意观察，有 `(object)`，是新式类），类中有一个变量 `x = 7`，当然，如果愿意还可以写别的。因为一下操作中，只用到这个，我就不写别的了。

```
>>> A.x
7
```

在类 A 中，变量 x 所引用的数据，能够直接通过类来调用。或者说 x 是类 A 的属性，这种属性有一个名称，曰“类属性”。类属性仅限于此——类中的变量。它也有其他的名字，如静态数据。

```
>>> foo = A()
>>> foo.x
7
```

实例化，通过实例也可以得到这个属性，这个属性叫做“实例属性”。对于同一属性，可以用类来访问（类属性），在一般情况下，也可以通过实例来访问同样的属性。但是：

```
>>> foo.x += 1
>>> foo.x
8
>>> A.x
7
```

实例属性更新了，类属性没有改变。这至少说明，类属性不会被实例属性左右，也可以进一步说“类属性与实例属性无关”。那么，`foo.x += 1` 的本质是什么呢？其本质是该实例 foo 又建立了一个新的属性，但是这个属性（新的 `foo.x`）居然与原来的属性（旧的 `foo.x`）重名，所以，原来的 `foo.x` 就被“遮盖了”，只能访问到新的 `foo.x`，它的值是 8.

```
>>> foo.x
8
```

```
>>> del foo.x
>>> foo.x
7
```

既然新的 `foo.x` “遮盖” 了旧的 `foo.x`, 如果删除它, 旧的不久显现出来了? 的确是。删除之后, `foo.x` 就还是原来的值。此外, 还可以通过建立一个不与它重名的实例属性:

```
>>> foo.y = foo.x + 1
>>> foo.y
8
>>> foo.x
7
```

`foo.y` 就是新建的一个实例属性, 它没有影响原来的实例属性 `foo.x`。

但是, 类属性能够影响实例属性, 这点应该好理解, 因为实例就是通过实例化调用类的。

```
>>> A.x += 1
>>> A.x
8
>>> foo.x
8
```

这时候实例属性跟着类属性而改变。

以上所言, 是指当类中变量引用的是不可变数据。如果类中变量引用可变数据, 情形会有所不同。因为可变数据能够进行原地修改。

```
>>> class B(object):
...     y = [1,2,3]
... 
```

这次定义的类中, 变量引用的是一个可变对象。

```
>>> B.y      #类属性
[1, 2, 3]
>>> bar = B()
>>> bar.y    #实例属性
[1, 2, 3]

>>> bar.y.append(4)
>>> bar.y
[1, 2, 3, 4]
>>> B.y
[1, 2, 3, 4]
```

```
>>> B.y.append("aa")
>>> B.y
[1, 2, 3, 4, 'aa']
>>> bar.y
[1, 2, 3, 4, 'aa']
```

从上面的比较操作中可以看出，当类中变量引用的是可变对象时，类属性和实例属性都能直接修改这个对象，从而影响另一方的值。

对于类属性和实例属性，除了上述不同之外，在下面的操作中，也会有差异。

```
>>> foo = A()
>>> dir(foo)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__module__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

实例化类 A，可以查看其所具有的属性（看最后一项，`x`），当然，执行 `dir(A)` 也是一样的。

```
>>> A.y = "hello"
>>> foo.y
'hello'
```

增加一个类属性，同时在实例属性中也增加了一样的名称和数据的属性。如果增加通过实例增加属性呢？看下面：

```
>>> foo.z = "python"
>>> foo.z
'python'
>>> A.z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'A' has no attribute 'z'
```

类并没有收纳这个属性。这进一步说明，类属性不受实例属性左右。另外，在类确定或者实例化之后，也可以增加和修改属性，其方法就是通过类或者实例的点号操作来实现，即 `object.attribute`，可以实现对属性的修改和增加。

数据流转

在类的应用中，最广泛的是将类实例化，通过实例来执行各种方法。所以，对此过程中的数据流转一定要弄明白。

回顾上节已经建立的那个类，做适当修改，读者是否能够写上必要的注释呢？如果你把注释写上，就已经理解了类的基本结构。

```
#!/usr/bin/env Python
# coding=utf-8

__metaclass__ = type

class Person:
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

    def breast(self, n):
        self.breast = n

    def color(self, color):
        print "%s is %s" % (self.name, color)

    def how(self):
        print "%s breast is %s" % (self.name, self.breast)

girl = Person('wangguniang')
girl.breast(90)

girl.color("white")
girl.how()
```

运行后结果：

```
$ python 20701.py
wangguniang is white
wangguniang breast is 90
```

一图胜千言，有图有真相。通过图示，我们看一看数据的流转过程。

```

class Person:
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

    def breast(self, n):
        self.breast = n

    def color(self, color):
        print "%s is %s" % (self.name, color)

    def how(self):
        print "%s breast is %s" % (self.name, self.breast)

girl = Person('canglaoshi')
girl.breast(90)

girl.color("white")
girl.how()

```

创建实例 `girl = Person('wangguniang')`，注意观察图上的箭头方向。`girl` 这个实例和 `Person` 类中的 `self` 对应，这正是应了上节所概括的“实例变量与 `self` 对应，实例变量主外，`self` 主内”的概括。`"wangguniang"` 是一个具体的数据，通过初始化函数中的 `name` 参数，传给 `self.name`，前面已经讲过，`self` 也是一个实例，可以为它设置属性，`self.name` 就是一个属性，经过初始化函数，这个属性的值由参数 `name` 传入，现在就是"`wangguniang`"。

在类 `Person` 的其它方法中，都是以 `self` 为第一个或者唯一一个参数。注意，在 Python 中，这个参数要显明写上，在类内部是不能省略的。这就表示所有方法都承接 `self` 实例对象，它的属性也被带到每个方法之中。例如在方法里面使用 `self.name` 即是调用前面已经确定的实例属性数据。当然，在方法中，还可以继续为实例 `self` 增加属性，比如 `self.breast`。这样，通过 `self` 实例，就实现了数据在类内部的流转。

如果要把数据从类里面传到外面，可以通过 `return` 语句实现。如上例子中所示的 `getName` 方法。

因为实例名称(`girl`)和 `self` 是对应关系，实际上，在类里面也可以用 `girl` 代替 `self`。例如，做如下修改：

```

#!/usr/bin/env Python
# coding=utf-8

__metaclass__ = type

class Person:

```

```

def __init__(self, name):
    self.name = name

def getName(self):
    #return self.name
    return girl.name #修改成这个样子，但是在编程实践中不要这么做。

girl = Person('wangguniang')
name = girl.getName()
print name

```

运行之后，打印：

```
wangguniang
```

这个例子说明，在实例化之后，实例变量 girl 和函数里面的那个 self 实例是完全对应的。但是，提醒读者，千万不要用上面的修改了的那个方式。因为那样写使类没有独立性，这是大忌。

命名空间

命名空间，英文名字：namespaces。在研究类或者面向对象编程中，它常常被提到。虽然在《[函数\(2\)](#)》中已经对命名空间进行了解释，那时是在函数的知识范畴中对命名空间的理解。现在，我们在类的知识范畴中理解“类命名空间”——定义类时，所有位于 class 语句中的代码都在某个命名空间中执行，即类命名空间。

在研习命名空间以前，请打开在 Python 的交互模式下，输入：`import this`，可以看到：

```

>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.

```

Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

这里列位看到的就是所谓《Python 之禅》，请看最后一句：Namespaces are one honking great idea -- I et's do more of those!

这是为了向看官说明 Namespaces、命名空间值重要性。

把在[《函数(2)》 href="https://github.com/qiwsir/StarterLearningPython/blob/master/202.md")中已经阐述的命名空间用一句比较学术化的语言概括：

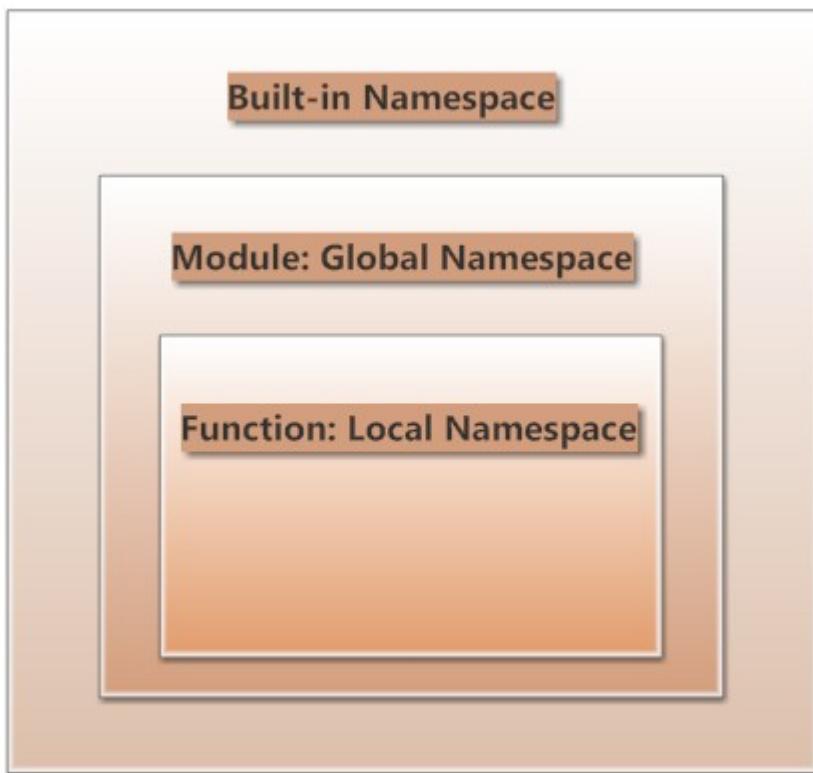
命名空间是从所定义的命名到对象的映射集合。

不同的命名空间，可以同时存在，当彼此相互独立互不干扰。

命名空间因为对象的不同，也有所区别，可以分为如下几种：

- 内置命名空间(Built-in Namespaces)：Python 运行起来，它们就存在了。内置函数的命名空间都属于内置命名空间，所以，我们可以在任何程序中直接运行它们，比如前面的 id(), 不需要做什么操作，拿过来就直接使用了。
- 全局命名空间(Module:Global Namespaces)：每个模块创建它自己所拥有的全局命名空间，不同模块的全局命名空间彼此独立，不同模块中相同名称的命名空间，也会因为模块的不同而不相互干扰。
- 本地命名空间(Function&Class: Local Namespaces)：模块中有函数或者类，每个函数或者类所定义的命名空间就是本地命名空间。如果函数返回了结果或者抛出异常，则本地命名空间也结束了。

从网上盗取了一张图，展示一下上述三种命名空间的关系



那么程序在查询上述三种命名空间的时候，就按照从里到外的顺序，即：Local Namespaces --> Global Namespaces --> Built-in Namespaces

```
>>> def foo(num,str):
...     name = "qiwsir"
...     print locals()
...
>>> foo(221,"qiwsir.github.io")
{'num': 221, 'name': 'qiwsir', 'str': 'qiwsir.github.io'}
>>>
```

这是一个访问本地命名空间的方法，用 `print locals()` 完成，从这个结果中不难看出，所谓的命名空间中的数据存储结构和 `dictionary` 是一样的。

根据习惯，看官估计已经猜测到了，如果访问全局命名空间，可以使用 `print globals()`。

作用域

作用域是指 Python 程序可以直接访问到的命名空间。“直接访问”在这里意味着访问命名空间中的命名时无需加入附加的修饰符。（这句话是从网上抄来的）

程序也是按照搜索命名空间的顺序，搜索相应空间的能够访问到的作用域。

```
def outer_foo():
    b = 20
    def inner_foo():
        c = 30
    a = 10
```

假如我现在位于 `inner_foo()` 函数内，那么 `c` 对我来讲就在本地作用域，而 `b` 和 `a` 就不是。如果我在 `inner_foo()` 内再做：`b=50`，这其实是在本地命名空间内新创建了对象，和上一层中的 `b=20` 毫不相干。可以看下面的例子：

```
#!/usr/bin/env Python
#coding:utf-8

def outer_foo():
    a = 10
    def inner_foo():
        a = 20
        print "inner_foo,a=",a      #a=20

    inner_foo()
    print "outer_foo,a=",a      #a=10

a = 30
outer_foo()
print "a=",a      #a=30

#运行结果

inner_foo,a= 20
outer_foo,a= 10
a= 30
```

如果要将某个变量在任何地方都使用，且能够关联，那么在函数内就使用 `global` 声明，其实就是曾经讲过的全局变量。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qjwsir@126.com,不胜感激。

类(4)

本节介绍类中一个非常重要的东西——继承，其实也没有那么重要，只是听起来似乎有点让初学者晕头转向，然后就感觉它属于很高级的东西，真是情况如何？学了之后你自然有感受。

在现实生活中，“继承”意味着一个人从另外一个人那里得到了一些什么，比如“继承革命先烈的光荣传统”、“某人继承他老爹的万贯家产”等。总之，“继承”之后，自己就在所继承的方面省力气、不用劳神费心，能轻松得到，比如继承了万贯家产，自己就一夜之间变成富豪。如果继承了“革命先烈的光荣传统”，自己是不是一下就变成革命者呢？

当然，生活中的继承或许不那么严格，但是编程语言中的继承是有明确规定和稳定的预期结果的。

继承（Inheritance）是面向对象软件技术当中的一个概念。如果一个类别 A “继承自” 另一个类别 B，就把这个 A 称为“B 的子类别”，而把 B 称为“A 的父类别”，也可以称“B 是 A 的超类”。

继承可以使得子类别具有父类别的各种属性和方法，而不需要再次编写相同的代码。在令子类别继承父类别的同时，可以重新定义某些属性，并重写某些方法，即覆盖父类别的原有属性和方法，使其获得与父类别不同的功能。另外，为子类别追加新的属性和方法也是常见的做法。（源自维基百科）

由上面对继承的表述，可以简单总结出继承的意图或者好处：

- 可以实现代码重用，但不是仅仅实现代码重用，有时候根本就没有重用
- 实现属性和方法继承

诚然，以上也不是全部，随着后续学习，对继承的认识会更深刻。好友令狐虫曾经这样总结继承：

从技术上说，OOP 里，继承最主要的用途是实现多态。对于多态而言，重要的是接口继承性，属性和行为是否存在继承性，这是不一定。事实上，大量工程实践表明，重度的行为继承会导致系统过度复杂和臃肿，反而会降低灵活性。因此现在比较提倡的是基于接口的轻度继承理念。这种模型里因为父类（接口类）完全没有代码，因此根本谈不上什么代码复用了。

在 Python 里，因为存在 Duck Type，接口定义的重要性大大的降低，继承的作用也进一步的被削弱了。

另外，从逻辑上说，继承的目的也不是为了复用代码，而是为了理顺关系。

他是大牛，或许读者感觉比较高深，没关系，随着你的实践经验的积累，你也能对这个问题有自己独到的见解。

或许你也要问我的观点是什么？我的观点就是：走着瞧！怎么理解？继续向下看，只有你先深入这个问题，才能跳到更高层看这个问题。小马过河的故事还记得吧？只有亲自走入河水中，才知道河水的深浅。

对于 Python 中的继承，前面一直在使用，那就是我们写的类都是新式类，所有新式类都是继承自 object 类。不要忘记，新式类的一种写法：

```
class NewStyle(object):
    pass
```

这就是典型的继承。

基本概念

```
#!/usr/bin/env Python
# coding=utf-8

__metaclass__ = type

class Person:
    def speak(self):
        print "I love you."

    def setHeight(self, n):
        self.length = n

    def breast(self, n):
        print "My breast is: ",n

class Girl(Person):
    def setHeight(self):
        print "The height is:1.70m ."

if __name__ == "__main__":
    cang = Girl()
    cang.setHeight()
    cang.speak()
    cang.breast(90)
```

上面这个程序，保存之后运行：

```
$ python 20901.py
The height is:1.70m .
I love you.
My breast is: 90
```

对以上程序进行解释，从中体会继承的概念和方法。

首先定义了一个类 Person，在这个类中定义了三个方法。注意，没有定义初始化函数，初始化函数在类中不是必不可少的。

然后又定义了一个类 Girl，这个类的名字后面的括号中，是上一个类的名字，这就意味着 Girl 继承了 Person，Girl 是 Person 的子类，Person 是 Girl 的父类。

既然是继承了 Person，那么 Girl 就全部拥有了 Person 中的方法和属性（上面的例子虽然没有列出属性）。但是，如果 Girl 里面有一个和 Person 同样名称的方法，那么就把 Person 中的同一个方法遮盖住了，显示的是 Girl 中的方法，这叫做方法的重写。

实例化类 Girl 之后，执行实例方法 `cang.setHeight()`，由于在类 Girl 中重写了 `setHeight` 方法，那么 Person 中的那个方法就不显作用了，在这个实例方法中执行的是类 Girl 中的方法。

虽然在类 Girl 中没有看到 `speak` 方法，但是因为它继承了 Person，所以 `cang.speak()` 就执行类 Person 中的方法。同理 `cang.breast(90)`，它们就好像是在类 Girl 里面已经写了这两个方法一样。既然继承了，就是我的了。

多重继承

所谓多重继承，就是只某一个类的父类，不止一个，而是多个。比如：

```
#!/usr/bin/env Python
# coding=utf-8

__metaclass__ = type

class Person:
    def eye(self):
        print "two eyes"

    def breast(self, n):
        print "The breast is: ",n

class Girl:
    age = 28
    def color(self):
        print "The girl is white"

class HotGirl(Person, Girl):
    pass

if __name__ == "__main__":
```

```
kong = HotGirl()
kong.eye()
kong.breast(90)
kong.color()
print kong.age
```

在这个程序中，前面有两个类：Person 和 Girl，然后第三个类 HotGirl 继承了这两个类，注意观察继承方法，就是在类的名字后面的括号中把所继承的两个类的名字写上。但是第三个类中什么方法也没有。

然后实例化类 HotGirl，既然继承了上面的两个类，那么那两个类的方法就都能够拿过来使用。保存程序，运行一下看看

```
$ python 20902.py
two eyes
The breast is: 90
The girl is white
28
```

值得注意的是，这次在类 Girl 中，有一个 `age = 28`，在对 HotGirl 实例化之后，因为继承的原因，这个类属性也被继承到 HotGirl 中，因此通过实例属性 `kong.age` 一样能够得到该数据。

由上述两个实例，已经清楚看到了继承的特点，即将父类的方法和属性全部承接到子类中；如果子类重写了父类的方法，就使用子类的该方法，父类的被遮盖。

多重继承的顺序

多重继承的顺序很必要了解。比如，如果一个子类继承了两个父类，并且两个父类有同样的方法或者属性，那么在实例化子类后，调用那个方法或属性，是属于哪个父类的呢？造一个没有实际意义，纯粹为了解决这个问题的程序：

```
#!/usr/bin/env Python
# coding=utf-8

class K1(object):
    def foo(self):
        print "K1-foo"

class K2(object):
    def foo(self):
        print "K2-foo"
    def bar(self):
        print "K2-bar"
```

```

class J1(K1, K2):
    pass

class J2(K1, K2):
    def bar(self):
        print "J2-bar"

class C(J1, J2):
    pass

if __name__ == "__main__":
    print C.__mro__
    m = C()
    m.foo()
    m.bar()

```

这段代码，保存后运行：

```

$ python 20904.py
<class '__main__.C'>, <class '__main__.J1'>, <class '__main__.J2'>, <class '__main__.K1'>, <class '__main__.K2'>, <type 'object'>
K1-foo
J2-bar

```

代码中的 `print C.__mro__` 是要打印出类的继承顺序。从上面清晰看出来了。如果要执行 `foo()` 方法，首先看 `J1`，没有，看 `J2`，还没有，看 `J1` 里面的 `K1`，有了，即 `C==>J1==>J2==>K1`；`bar()` 也是按照这个顺序，在 `J2` 中就找到了一个。

这种对继承属性和方法搜索的顺序称之为“广度优先”。

新式类用以及 Python3.x 中都是按照此顺序原则搜寻属性和方法的。

但是，在旧式类中，是按照“深度优先”的顺序的。因为后面读者也基本不用旧式类，所以不举例。如果读者愿意，可以自己模仿上面代码，探索旧式类的“深度优先”含义。

super 函数

对于初始化函数的继承，跟一般方法的继承，还有点不同。可以看下面的例子：

```

#!/usr/bin/env Python
# coding=utf-8

__metaclass__ = type

class Person:

```

```

def __init__(self):
    self.height = 160

def about(self, name):
    print "{} is about {}".format(name, self.height)

class Girl(Person):
    def __init__(self):
        self.breast = 90

    def about(self, name):
        print "{} is a hot girl, she is about {}, and her breast is {}".format(name, self.height, self.breast)

if __name__ == "__main__":
    cang = Girl()
    cang.about("wangguniang")

```

在上面这段程序中，类 Girl 继承了类 Person。在类 Girl 中，初始化设置了 `self.breast = 90`，由于继承了 Person，按照前面的经验，Person 的初始化函数中的 `self.height = 160` 也应该被 Girl 所继承过来。然后在重写的 about 方法中，就是用 `self.height`。

实例化类 Girl，并执行 `cang.about("wangguniang")`，试图打印出一句话 `wangguniang is a hot girl, she is about 160, and her breast is 90`。保存程序，运行之：

```

$ python 20903.py
Traceback (most recent call last):
  File "20903.py", line 22, in <module>
    cang.about("wangguniang")
  File "20903.py", line 18, in about
    print "{} is a hot girl, she is about {}, and her breast is {}".format(name, self.height, self.breast)
AttributeError: 'Girl' object has no attribute 'height'

```

报错！

程序员有一句名言：不求最好，但求报错。报错不是坏事，是我们长经验的时候，是在告诉我们，那么做不对。

重要的是看报错信息。就是我们要打印的那句话出问题了，报错信息显示 `self.height` 是不存在的。也就是说类 Girl 没有从 Person 中继承过来这个属性。

原因是什么？仔细观察类 Girl，会发现，除了刚才强调的 about 方法重写了，`__init__` 方法，也被重写了。不要认为它的名字模样奇怪，就不把它看做类中的方法（函数），它跟类 Person 中的 `__init__` 重名了，也同样是重写了那个初始化函数。

这就提出了一个问题。因为在子类中重写了某个方法之后，父类中同样的方法被遮盖了。那么如何再把父类的该方法调出来使用呢？纵然被遮盖了，应该还是存在的，不要浪费了呀。

Python 中有这样一种方法，这种方式是被提倡的方法：super 函数。

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class Person:
    def __init__(self):
        self.height = 160

    def about(self, name):
        print "{} is about {}".format(name, self.height)

class Girl(Person):
    def __init__(self):
        super(Girl, self).__init__()
        self.breast = 90

    def about(self, name):
        print "{} is a hot girl, she is about {}, and her breast is {}".format(name, self.height, self.breast)
        super(Girl, self).about(name)

if __name__ == "__main__":
    cang = Girl()
    cang.about("wangguniang")
```

在子类中，`__init__` 方法重写了，为了调用父类同方法，使用 `super(Girl, self).__init__()` 的方式。`super` 函数的参数，第一个是当前子类的类名字，第二个是 `self`，然后是点号，点号后面是所要调用的父类的方法。同样在子类重写的 `about` 方法中，也可以调用父类的 `about` 方法。

执行结果：

```
$ python 20903.py
wangguniang is a hot girl, she is about 160, and her breast is 90
wangguniang is about 160
```

最后要提醒注意：`super` 函数仅仅适用于新式类。当然，你一定是使用的新式类。“喜新厌旧”是程序员的嗜好。

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

类(5)

在前面几节讨论类的时候，经常要将类实例化，然后通过实例来调用类的方法（函数）。在此，把前面经常做的这类事情概括一下：

- 方法是类内部定义函数，只不过这个函数的第一个参数是 `self`。（可以认为方法是类属性，但不是实例属性）
- 必须将类实例化之后，才能通过实例调用该类的方法。调用的时候在方法后面要跟括号（括号中默认有 `self` 参数，但是不写出来。）

通过实例调用方法（在前面曾用了一个不严谨的词语：实例方法），我们称这个方法绑定在实例上。

调用绑定方法

前面一直在这样做。比如：

```
class Person(object):
    def foo(self):
        pass
```

如果要调用 `Person.foo()` 方法，必须：

```
pp = Person() #实例化
pp.foo()
```

这样就实现了方法和实例的绑定，于是通过 `pp.foo()` 即可调用该方法。

调用非绑定方法

在《[类\(4\)](#)》中，介绍了一个函数 `super`。为了描述方便，把代码复制过来：

```
#!/usr/bin/env python
# coding=utf-8

__metaclass__ = type

class Person:
    def __init__(self):
        self.height = 160
```

```

def about(self, name):
    print "{} is about {}".format(name, self.height)

class Girl(Person):
    def __init__(self):
        super(Girl, self).__init__()
        self.breast = 90

    def about(self, name):
        print "{} is a hot girl, she is about {}, and her breast is {}".format(name, self.height, self.breast)
        super(Girl, self).about(name)

if __name__ == "__main__":
    cang = Girl()
    cang.about("wangguniang")

```

在子类 Girl 中，因为重写了父类的 `__init__` 方法，如果要调用父类该方法，在上节中不得不使用 `super(Girl, self).__init__()` 调用父类中因为子类方法重写而被遮蔽的同名方法。

其实，在子类中，父类的方法就是非绑定方法，因为在子类中，没有建立父类的实例，却要是用父类的方法。对于这种非绑定方法的调用，还有一种方式。不过这种方式现在已经较少是用了，因为有了 `super` 函数。为了方便读者看其它有关代码，还是要简要说明。

例如在上面代码中，在类 Girl 中想调用父类 Person 的初始化函数，则需要在子类中，写上这么一行：

```
Person.__init__(self)
```

这不是通过实例调用的，而是通过类 Person 实现了对 `__init__(self)` 的调用。这就是调用非绑定方法的用途。但是，这种方法已经被 `super` 函数取代，所以，如果读者在编程中遇到类似情况，推荐使用 `super` 函数。

静态方法和类方法

已知，类的方法第一个参数必须是 `self`，并且如果要调用类的方法，必须将通过类的实例，即方法绑定实例后才能由实例调用。如果不绑定，一般在继承关系的类之间，可以用 `super` 函数等方法调用。

这里再介绍一种方法，这种方法的调用方式跟上述的都不同，这就是：静态方法和类方法。看代码：

```

#!/usr/bin/env Python
# coding=utf-8

__metaclass__ = type

```

```

class StaticMethod:
    @staticmethod
    def foo():
        print "This is static method foo()."

class ClassMethod:
    @classmethod
    def bar(cls):
        print "This is class method bar()."
        print "bar() is part of class:", cls.__name__

if __name__ == "__main__":
    static_foo = StaticMethod() #实例化
    static_foo.foo()          #实例调用静态方法
    StaticMethod.foo()         #通过类来调用静态方法
    print "*****"
    class_bar = ClassMethod()
    class_bar.bar()
    ClassMethod.bar()

```

对于这部分代码，有一处非常特别，那就是包含了“@”符号。在 Python 中：

- `@staticmethod` 表示下面的方法是静态方法
- `@classmethod` 表示下面的方法是类方法

一个一个来看。

先看静态方法，虽然名为静态方法，但也是方法，所以，依然用 `def` 语句来定义。需要注意的是文件名后面的括号内，没有 `self`，这和前面定义的类中的方法是不同的，也正是因着这个不同，才给它另外取了一个名字叫做静态方法，否则不就“泯然众人矣”。如果没有 `self`，那么也就无法访问实例变量、类和实例的属性了，因为它们都是借助 `self` 来传递数据的。

再看类方法，同样也具有一般方法的特点，区别也在参数上。类方法的参数也没有 `self`，但是必须有 `cls` 这个参数。在类方法中，能够方法类属性，但是不能访问实例属性（读者可以自行设计代码检验之）。

简要明确两种方法。下面看调用方法。两种方法都可以通过实例调用，即绑定实例。也可以通过类来调用，即 `StaticMethod.foo()` 这样的形式，这也是区别一般方法的地方，一般方法必须用通过绑定实例调用。

上述代码运行结果：

```

$ python 21001.py
This is static method foo().
This is static method foo().
*****

```

```
This is class method bar().
bar() is part of class: ClassMethod
This is class method bar().
bar() is part of class: ClassMethod
```

这是关于静态方法和类方法的简要介绍。

正当我思考如何讲解的更深入一点的时候，我想起了以往看过的一篇文章，觉得人家讲的非常到位。所以，不敢吝啬，更不敢班门弄斧，所以干醋把那篇文章恭恭敬敬的抄录于此。同时，读者从下面的文章中，也能对前面的知识复习一下。文章标题是：Python 中的 staticmethod 和 classmethod 的差异。原载：www.pythontutorial.net/python-basics/python-static-class-method/。此地址需要你准备梯子才能浏览。后经国人翻译，地址是：<http://www.wkiken.me/posts/2013/12/22/difference-betweenstaticmethod-and-classmethod-in-Python.html>

以下是翻译文章：

Class vs static methods in Python

这篇文章试图解释：什么事 staticmethod/classmethod，并且这两者之间的差异。

staticmethod 和 classmethod 均被作为装饰器，用作定义一个函数为"staticmethod"还是"classmethod"

如果想要了解 Python 装饰器的基础，可以看[这篇文章](#)

Simple, static and class methods

类中最常用到的方法是 实例方法(instance methods)，即，实例对象作为第一个参数传递给函数

例如，下面是一个基本的实例方法

```
class Kls(object):
    def __init__(self, data):
        self.data = data

    def printd(self):
        print(self.data)

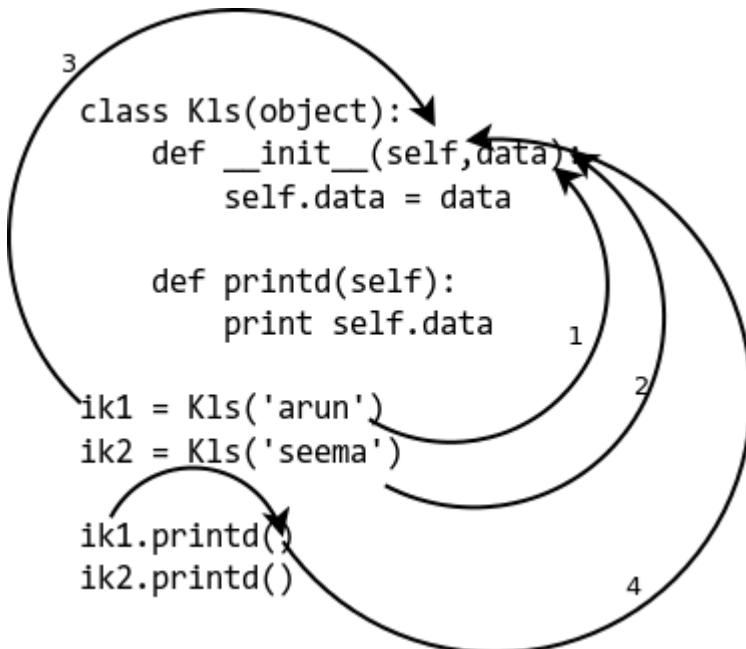
ik1 = Kls('arun')
ik2 = Kls('seema')

ik1.printd()
ik2.printd()
```

得到的输出:

```
arun
seema
```

调用关系图:



查看代码和图解:

1/2 参数传递给函数

3 self 参数指向实例本身

4 我们不需要显式提供实例，解释器本身会处理

假如我们想仅实现类之间交互而不是通过实例？我们可以在类之外建立一个简单的函数来实现这个功能，但是将会使代码扩散到类之外，这个可能对未来代码维护带来问题。

例如:

```
def get_no_of_instances(cls_obj):  
    return cls_obj.no_inst

class Kls(object):  
    no_inst = 0

    def __init__(self):  
        Kls.no_inst = Kls.no_inst + 1
```

```
ik1 = Kls()
ik2 = Kls()

print(get_no_of_instances(Kls))
```

结果：

```
2
```

The Python @classmethod

现在我们要做的是在类里创建一个函数，这个函数参数是类对象而不是实例对象。

在上面那个实现中，如果要实现不获取实例，需要修改如下：

```
def iget_no_of_instance(ins_obj):
    return ins_obj.__class__.no_inst

class Kls(object):
    no_inst = 0

    def __init__(self):
        Kls.no_inst = Kls.no_inst + 1

ik1 = Kls()
ik2 = Kls()
print iget_no_of_instance(ik1)
```

结果

```
2
```

可以使用 Python2.2 引入的新特性，使用 @classmethod 在类代码中创建一个函数

```
class Kls(object):
    no_inst = 0

    def __init__(self):
        Kls.no_inst = Kls.no_inst + 1

    @classmethod
    def get_no_of_instance(cls_obj):
        return cls_obj.no_inst

ik1 = Kls()
```

```
ik2 = Kls()

print ik1.get_no_of_instance()
print Kls.get_no_of_instance()
```

We get the following output:

```
2
2
```

The Python @staticmethod

通常，有很多情况下一些函数与类相关，但不需要任何类或实例变量就可以实现一些功能。

比如设置环境变量，修改另一个类的属性等等。这种情况下，我们也可以使用一个函数，一样会将代码扩散到类之外（难以维护）

下面是一个例子：

```
IND = 'ON'

def checkind():
    return (IND == 'ON')

class Kls(object):
    def __init__(self,data):
        self.data = data

    def do_reset(self):
        if checkind():
            print('Reset done for:', self.data)

    def set_db(self):
        if checkind():
            self.db = 'new db connection'
            print('DB connection made for:',self.data)

ik1 = Kls(12)
ik1.do_reset()
ik1.set_db()
```

结果：

```
Reset done for: 12
DB connection made for: 12
```

现在我们使用 `@staticmethod`, 我们可以将所有代码放到类中

```
IND = 'ON'

class Kls(object):
    def __init__(self, data):
        self.data = data

    @staticmethod
    def checkind():
        return (IND == 'ON')

    def do_reset(self):
        if self.checkind():
            print('Reset done for:', self.data)

    def set_db(self):
        if self.checkind():
            self.db = 'New db connection'
            print('DB connection made for: ', self.data)

ik1 = Kls(12)
ik1.do_reset()
ik1.set_db()
```

得到的结果:

```
Reset done for: 12
DB connection made for: 12
```

How `@staticmethod` and `@classmethod` are different

```
class Kls(object):
    def __init__(self, data):
        self.data = data

    def printd(self):
        print(self.data)

    @staticmethod
    def smethod(*arg):
        print('Static:', arg)

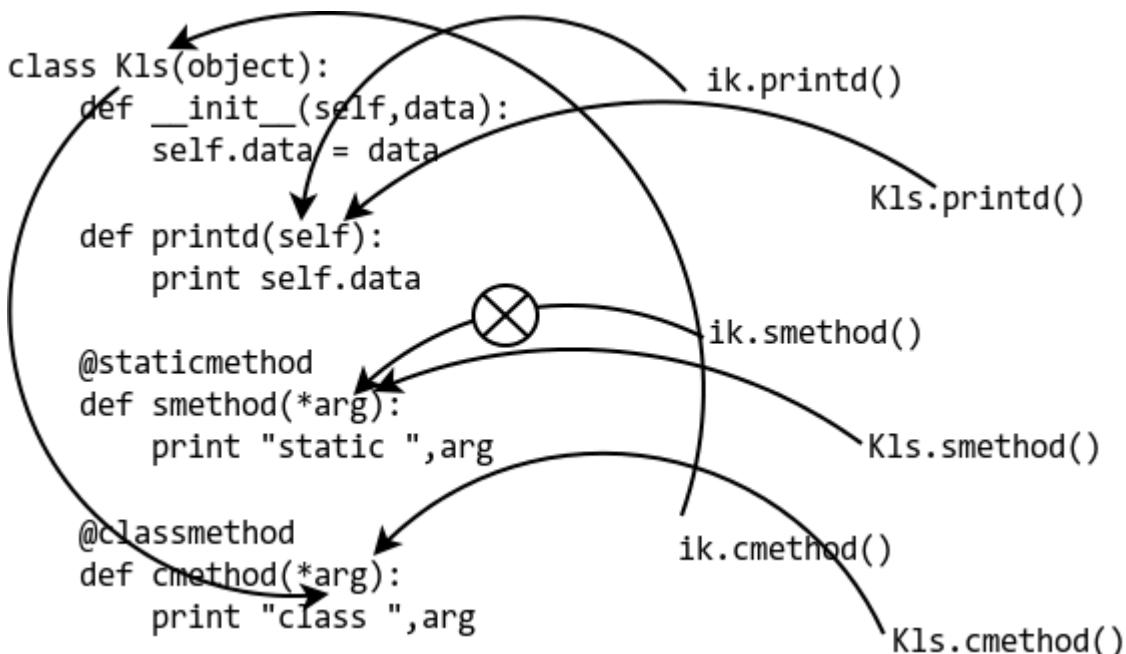
    @classmethod
```

```
def cmethod(*arg):
    print('Class:', arg)
```

调用

```
>>> ik = Kls(23)
>>> ik.printd()
23
>>> ik.smethod()
Static: ()
>>> ik.cmethod()
Class: (<class '__main__.Kls'>,)
>>> Kls.printd()
TypeError: unbound method printd() must be called with Kls instance as first argument (got nothing instead)
>>> Kls.smethod()
Static: ()
>>> Kls.cmethod()
Class: (<class '__main__.Kls'>,)
```

图解



文档字符串

在写程序的时候，必须要写必要的文字说明，没别的原因，除非你的代码写的非常容易理解，特别是各种变量、函数和类等的命名任何人都能够很容易理解，否则，文字说明是不可缺少的。

在函数、类或者文件开头的部分写文档字符串说明，一般采用三重引号。这样写的最大好处是能够用 help() 函数看。

```
"""This is python lesson"""

def start_func(arg):
    """This is a function."""
    pass

class MyClass:
    """This is my class."""
    def my_method(self,arg):
        """This is my method."""
        pass
```

这样的文档是必须的。

当然，在编程中，有不少地方要用“#”符号来做注释。一般用这个来注释局部。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qjwsir@126.com,不胜感激。

多态和封装

前面讲过的“继承”，是类的一个重要特征，在编程中用途很多。这里要说两个在理解和实践上有争议的话题：多态和封装。所谓争议，多来自于对同一个现象不同角度的理解，特别是有不少经验丰富的程序员，还从其它语言的角度来诠释 Python 的多态等。

多态

在网上搜索一下，发现对 Python 的多态问题，的确是仁者见仁智者见智。

作为一个初学者，不一定要也没有必要、或者还没有能力参与这种讨论。但是，应该理解 Python 中关于多态的基本体现，也要对多态有一个基本的理解。

```
>>> "This is a book".count("s")
2
>>> [1,2,4,3,5,3].count(3)
2
```

上面的 `count()` 的作用是数一数某个元素在对象中出现的次数。从例子中可以看出，我们并没有限定 `count` 的参数。类似的例子还有：

```
>>> f = lambda x,y:x+y
```

还记得这个 `lambda` 函数吗？如果忘记了，请复习[\[函数\(4\)\]](https://github.com/qiwsir/StarterLearningPython/blob/master/204.md)中对此的解释。

```
>>> f(2,3)
5
>>> f("qiw","sir")
'qiwsir'
>>> f(["python","java"],["c++","lisp"])
['python', 'java', 'c++', 'lisp']
```

在那个 `lambda` 函数中，我们没有限制参数的类型，也一定不能限制，因为如果限制了，就不是 Pythonic 了。在使用的时候，可以给参数任意类型，都能到的不报错的结果。当然，这样做之所以合法，更多的是来自于 `+` 的功能强悍。

以上，就体现了“多态”。当然，也有人就此提出了反对意见，因为本质上是在参数传入值之前，Python 并没有确定参数的类型，只能让数据进入函数之后再处理，能处理则罢，不能处理就报错。例如：

```
>>> f("qiw", 2)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 1, in <lambda>
TypeError: cannot concatenate 'str' and 'int' objects
```

本教程由于不属于自己这种概念争论范畴，所以不进行这方面的深入探索，仅仅是告诉各位读者相关信息。并且，本教程也是按照“人云亦云”的原则，既然大多数程序员都在讨论多态，那么我们就按照大多数人说的去介绍（尽管有时候真理掌握在少数人手中）。

“多态”，英文是:Polymorphism，在台湾被称作“多型”。维基百科中对此有详细解释说明。

多型（英语：Polymorphism），是指物件导向程式执行时，相同的讯息可能会送给多个不同的类别之物件，而系统可依物件所属类别，引发对应类别的方法，而有不同的行为。简单来说，所谓多型意指相同的讯息给予不同的物件会引发不同的动作称之。

再简化的说法就是“有多种形式”，就算不知道变量（参数）所引用的对象类型，也一样能进行操作，来者不拒。比如上面显示的例子。在 Python 中，更为 Pythonic 的做法是根本就不进行类型检验。

例如著名的 `repr()` 函数，它能够针对输入的任何对象返回一个字符串。这就是多态的代表之一。

```
>>> repr([1,2,3])
'[1, 2, 3]'
>>> repr(1)
'1'
>>> repr({"lang":"python"})
"{'lang': 'Python'}
```

使用它写一个小函数，还是作为多态代表的。

```
>>> def length(x):
...     print "The length of", repr(x), "is", len(x)
...
>>> length("how are you")
The length of 'how are you' is 11
>>> length([1,2,3])
The length of [1, 2, 3] is 3
>>> length({"lang":"python","book":"itdiffer.com"})
The length of {'lang': 'python', 'book': 'itdiffer.com'} is 2
```

不过，多态也不是万能的，如果这样做：

```
>>> length(7)
The length of 7 is
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in length
TypeError: object of type 'int' has no len()
```

报错了。看错误提示，明确告诉了我们 `object of type 'int' has no len()`。

在诸多介绍多态的文章中，都会有这样关于猫和狗的例子。这里也将代码贴出来，读者去体会所谓多态体现。其实，如果你进入了 Python 的语境，有时候是不经意就已经在应用多态特性呢。

```
#!/usr/bin/env Python
# coding=utf-8

"the code is from: http://zetcode.com/lang/python/oop/"

__metaclass__ = type

class Animal:
    def __init__(self, name=""):
        self.name = name

    def talk(self):
        pass

class Cat(Animal):
    def talk(self):
        print "Meow!"

class Dog(Animal):
    def talk(self):
        print "Woof!"

a = Animal()
a.talk()

c = Cat("Missy")
c.talk()

d = Dog("Rocky")
d.talk()
```

保存后运行之：

```
$ python 21101.py
Meow!
Woof!
```

代码中有 Cat 和 Dog 两个类，都继承了类 Animal，它们都有 `talk()` 方法，输入不同的动物名称，会得出相应的结果。

关于多态，有一个被称作“鸭子类型”(duck typeing)的东西，其含义在维基百科中被表述为：

在程序设计中，鸭子类型（英语：duck typing）是动态类型的一种风格。在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定。这个概念的名字来源于由 James Whitcomb Riley 提出的鸭子测试（见下面的“历史”章节），“鸭子测试”可以这样表述：“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”

对于鸭子类型，也是有争议的。这方面的详细信息，读者可以去看有关维基百科的介绍。

对于多态问题，最后还要告诫读者，类型检查是毁掉多态的利器，比如 `type`、`isinstance` 以及 `isubclass` 函数，所以，一定要慎用这些类型检查函数。

封装和私有化

在正式介绍封装之前，先扯个笑话。

某软件公司老板，号称自己懂技术。一次有一个项目要交付给客户，但是他有不想让客户知道实现某些功能的代码，但是交付的时候要给人家代码的。于是该老板就告诉程序员，“你们把那部分核心代码封装一下”。程序员听了之后，迷茫了。

不知道你有没有笑。

“封装”，是不是把代码写到某个东西里面，“人”在编辑器中打开，就看不到了呢？除非是你的显示器坏了。

在程序设计中，封装(Encapsulation)是对 `object` 的一种抽象，即将某些部分隐藏起来，在程序外部看不到，即无法调用（不是人用眼睛看不到那个代码，除非用某种加密或者混淆方法，造成现实上的困难，但这不是封装）。

要了解封装，离不开“私有化”，就是将类或者函数中的某些属性限制在某个区域之内，外部无法调用。

Python 中私有化的方法也比较简单，就是在准备私有化的属性（包括方法、数据）名字前面加双下划线。例如：

```
#!/usr/bin/env Python
# coding=utf-8

__metaclass__ = type

class ProtectMe:
    def __init__(self):
```

```

self.me = "qiwsir"
self.__name = "kivi"

def __python(self):
    print "I love Python."

def code(self):
    print "Which language do you like?"
    self.__python()

if __name__ == "__main__":
    p = ProtectMe()
    print p.me
    print p.__name

```

运行一下，看看效果：

```

$ python 21102.py
qiwsir
Traceback (most recent call last):
  File "21102.py", line 21, in <module>
    print p.__name
AttributeError: 'ProtectMe' object has no attribute '__name'

```

查看报错信息，告诉我们没有 `__name` 那个属性。果然隐藏了，在类的外面无法调用。再试试那个函数，可否？

```

if __name__ == "__main__":
    p = ProtectMe()
    p.code()
    p.__python()

```

修改这部分即可。其中 `p.code()` 的意图是要打印出两句话：“Which language do you like?” 和 “I love Python.”，`code()` 方法和 `__python()` 方法在同一个类中，可以调用之。后面的那个 `p.__Python()` 试图调用那个私有方法。看看效果：

```

$ python 21102.py
Which language do you like?
I love Python.
Traceback (most recent call last):
  File "21102.py", line 23, in <module>
    p.__python()
AttributeError: 'ProtectMe' object has no attribute '__python'

```

如愿以偿。该调用的调用了，该隐藏的隐藏了。

用上面的方法，的确做到了封装。但是，我如果要调用那些私有属性，怎么办？

可以使用 `property` 函数。

```
#!/usr/bin/env Python
# coding=utf-8

__metaclass__ = type

class ProtectMe:
    def __init__(self):
        self.me = "qiwsir"
        self.__name = "kivi"

    @property
    def name(self):
        return self.__name

if __name__ == "__main__":
    p = ProtectMe()
    print p.name
```

运行结果：

```
$ python 21102.py
kivi
```

从上面可以看出，用了 `@property` 之后，在调用那个方法的时候，用的是 `p.name` 的形式，就好像在调用一个属性一样，跟前面 `p.me` 的格式相同。

看来，封装的确不是让“人看不见”。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

特殊方法 (1)

探究更多的类属性，在一些初学者的教程中，一般很少见。我之所以要在这里也将这部分奉献出来，就是因为本教程是“From Beginner to Master”。当然，不是学习了类的更多属性就能达到 Master 水平，但是这是通往 Master 的一步，虽然在初级应用中，本节乃至后面关于类的属性用的不很多，但是，这一步迈出去，你就会在实践中有一个印象，以后需要用到，知道有这一步，会对项目有帮助的。俗话说“艺不压身”。

`__dict__`

前面已经学习过有关类属性和实例属性的内容，并且做了区分，如果忘记了可以回头参阅[《类\(3\)》](#)中的“类属性和实例属性”部分。有一个结论，是一定要熟悉的，那就是可以通过 `object.attribute` 的方式访问对象的属性。

如果接着那部分内容，读者是否思考过一个问题：类或者实例属性，在 Python 中是怎么存储的？或者为什么修改或者增加、删除属性，我们能不能控制这些属性？

```
>>> class A(object):
...     pass
...
>>> a = A()
>>> dir(a)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__module__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>> dir(A)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__module__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

用 `dir()` 来查看一下，发现不管是类还是实例，都有很多属性，这在前面已经反复出现，有点见怪不怪了。不过，这里我们要看一个属性：`__dict__`，因为它是一个保存秘密的东西：对象的属性。

```
>>> class Spring(object):
...     season = "the spring of class"
...
>>> Spring.__dict__
dictProxy({'__dict__': <attribute '__dict__' of 'Spring' objects>,
'season': 'the spring of class',
'__module__': '__main__',
'__weakref__': <attribute '__weakref__' of 'Spring' objects>,
'__doc__': None})
```

为了便于观察，我将上面的显示结果进行了换行，每个键值对一行。

对于类 Spring 的 `__dict__` 属性，可以发现，有一个键 'season'，这就是这个类的属性；其值就是类属性的数据。

```
>>> Spring.__dict__['season']
'the spring of class'
>>> Spring.season
'the spring of class'
```

用这两种方式都能得到类属性的值。或者说 `Spring.__dict__['season']` 就是访问类属性。下面将这个类实例化，再看看它的实例属性：

```
>>> s = Spring()
>>> s.__dict__
{}
```

实例属性的 `__dict__` 是空的。有点奇怪？不奇怪，接着看：

```
>>> s.season
'the spring of class'
```

这个其实是指向了类属性中的 `Spring.season`，至此，我们其实还没有建立任何实例属性呢。下面就建立一个实例属性：

```
>>> s.season = "the spring of instance"
>>> s.__dict__
{'season': 'the spring of instance'}
```

这样，实例属性里面就不空了。这时候建立的实例属性和上面的那个 `s.season` 只不过重名，并且把它“遮盖”了。这句好是不是熟悉？因为在讲述“实例属性”和“类属性”的时候就提到了。现在读者肯定理解更深入了。

```
>>> s.__dict__['season']
'the spring of instance'
>>> s.season
'the spring of instance'
```

此时，那个类属性如何？我们看看：

```
>>> Spring.__dict__['season']
'the spring of class'
>>> Spring.__dict__
dict_proxy({'__dict__': <attribute '__dict__' of 'Spring' objects>, 'season': 'the spring of class', '__module__': '__main__', '__qualname__': 'Spring'}
```

Spring 的类属性没有受到实例属性的影响。

按照前面的讲述类属性和实例熟悉的操作，如果这时候将前面的实例属性删除，会不会回到实例属性 `s.__dict__` 为空呢？

```
>>> del s.season
>>> s.__dict__
{}
>>> s.season
'the spring of class'
```

果然打回原形。

当然，你可以定义其它名称的实例属性，它一样被存储到 `__dict__` 属性里面：

```
>>> s.lang = "python"
>>> s.__dict__
{'lang': 'python'}
>>> s.__dict__['lang']
'python'
```

诚然，这样做仅仅是更改了实例的 `__dict__` 内容，对 `Spring.__dict__` 无任何影响，也就是说通过 `Spring.lang` 或者 `Spring.__dict__['lang']` 是得不到上述结果的。

```
>>> Spring.lang
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Spring' has no attribute 'lang'

>>> Spring.__dict__['lang']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'lang'
```

那么，如果这样操作，会怎样呢？

```
>>> Spring.flower = "peach"
>>> Spring.__dict__
dict_proxy({'__module__': '__main__',
'flower': 'peach',
'season': 'the spring of class',
'__dict__': <attribute '__dict__' of 'Spring' objects>, '__weakref__': <attribute '__weakref__' of 'Spring' objects>, '__doc__':
>>> Spring.__dict__['flower']
'peach'
```

在类的 `__dict__` 被更改了，类属性中增加了一个'flower'属性。但是，实例的 `__dict__` 中如何？

```
>>> s.__dict__
{'lang': 'python'}
```

没有被修改。我也是这么想的，哈哈。你此前这这么觉得吗？然而，还能这样：

```
>>> s.flower
'peach'
```

这个读者是否能解释？其实又回到了前面第一个出现 `s.season` 上面了。

通过上面探讨，是不是基本理解了实例和类的 `__dict__`，并且也看到了属性的变化特点。特别是，这些属性都是可以动态变化的，就是你可以随时修改和增删。

属性如此，方法呢？下面就看看方法（类中的函数）。

```
>>> class Spring(object):
...     def tree(self, x):
...         self.x = x
...         return self.x
...
>>> Spring.__dict__
dict_proxy({'__dict__': <attribute '__dict__' of 'Spring' objects>,
 '__weakref__': <attribute '__weakref__' of 'Spring' objects>,
 '__module__': '__main__',
 'tree': <function tree at 0xb748fdf4>,
 '__doc__': None})
```



```
>>> Spring.__dict__['tree']
<function tree at 0xb748fdf4>
```

结果跟前面讨论属性差不多，方法 `tree` 也在 `__dict__` 里面呢。

```
>>> t = Spring()
>>> t.__dict__
{}
```

又跟前面一样。虽然建立了实例，但是在实例的 `__dict__` 中没有方法。接下来，执行：

```
>>> t.tree("xiangzhangshu")
'xiangzhangshu'
```

在[类\(3\)](#)中有一部分内容阐述“数据流转”，其中有一张图，其中非常明确显示出，当用上面方式执行方法的时候，实例 `t` 与 `self` 建立了对应关系，两者是一个外一个内。在方法中 `self.x = x`，将 `x` 的值给了 `self.x`，也就是实例应该拥有了这么一个属性。

```
>>> t.__dict__
{'x': 'xiangzhangshu'}
```

果然如此。这也印证了实例 `t` 和 `self` 的关系，即实例方法(`t.tree('xiangzhangshu')`)的第一个参数(`self`，但没有写出来)绑定实例 `t`，透过 `self.x` 来设定值，即给 `t.__dict__` 添加属性值。

换一个角度：

```
>>> class Spring(object):
...     def tree(self, x):
...         return x
... 
```

这回方法中没有将 `x` 赋值给 `self` 的属性，而是直接 `return`，结果是：

```
>>> s = Spring()
>>> s.tree("liushu")
'liushu'
>>> s.__dict__
{}
```

是不是理解更深入了？

现在需要对 Python 中一个观点：“一切皆对象”，再深入领悟。以上不管是类还是的实例的属性和方法，都是符合 `object.attribute` 格式，并且属性类似。

当你看到这里的时候，要么明白了类和实例的 `__dict__` 的特点，要么就糊涂了。糊涂也不要紧，再将上面的重复一遍，特别是自己要敲一敲有关代码。（建议一个最好的方法：用两个显示器，一个显示器看本教程，另外一个显示器敲代码。事半功倍的效果。）

需要说明，我们对 `__dict__` 的探讨还留有一个尾巴：属性搜索路径。这个留在后面讲述。

不管是类还是实例，其属性都能随意增加。这点在有时候不是一件好事情，或许在某些时候你不希望别人增加属性。有办法吗？当然有，请继续学习。

__slots__

首先声明，`__slots__` 能够限制属性的定义，但是这不是它存在终极目标，它存在的终极目标更应该是一个在编程中非常重要的方面：优化内存使用。

```
>>> class Spring(object):
...     __slots__ = ("tree", "flower")
...
>>> dir(Spring)
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

仔细看看 `dir()` 的结果，还有 `__dict__` 属性吗？没有了，的确没有了。也就是说 `__slots__` 把 `__dict__` 挤出去了，它进入了类的属性。

```
>>> Spring.__slots__
('tree', 'flower')
```

这里可以看出，类 `Spring` 有且仅有两个属性。

```
>>> t = Spring()
>>> t.__slots__
('tree', 'flower')
```

实例化之后，实例的 `__slots__` 与类的完全一样，这跟前面的 `__dict__` 大不一样了。

```
>>> Spring.tree = "liushu"
```

通过类，先赋予一个属性值。然后，检验一下实例能否修改这个属性：

```
>>> t.tree = "guangyulan"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Spring' object attribute 'tree' is read-only
```

看来，我们的意图不能达成，报错信息中显示，`tree` 这个属性是只读的，不能修改了。

```
>>> t.tree
'liushu'
```

因为前面已经通过类给这个属性赋值了。不能用实例属性来修改。只能：

```
>>> Spring.tree = "guangyulan"
>>> t.tree
'guangyulan'
```

用类属性修改。但是对于没有用类属性赋值的，可以通过实例属性：

```
>>> t.flower = "haitanghua"
>>> t.flower
'haihanghua'
```

但此时：

```
>>> Spring.flower
<member 'flower' of 'Spring' objects>
```

实例属性的值并没有传回到类属性，你也可以理解为新建立了一个同名的实例属性。如果再给类属性赋值，那么就会这样了：

```
>>> Spring.flower = "ziteng"
>>> t.flower
'ziteng'
```

当然，此时在给 `t.flower` 重新赋值，就会爆出跟前面一样的错误了。

```
>>> t.water = "green"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Spring' object has no attribute 'water'
```

这里试图给实例新增一个属性，也失败了。

看来 `__slots__` 已经把实例属性牢牢地管控了起来，但更本质的是优化了内存。诚然，这种优化会在大量的实例时候显出效果。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

特殊方法 (2)

书接上回，不管是实例还是类，都用 `__dict__` 来存储属性和方法，可以笼统地把属性和方法称为成员或者特性，用一句笼统的话说，就是 `__dict__` 存储对象成员。但，有时候访问的对象成员没有存在其中，就是这样：

```
>>> class A(object):
...     pass
...
>>> a = A()
>>> a.x
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'x'
```

x 不是实例的成员，用 `a.x` 访问，就出错了，并且错误提示中报告了原因：“'A' object has no attribute 'x'"

在很多情况下，这种报错是足够的了。但是，在某种我现在还说不出的情况下，你或许不希望这样报错，或许希望能够有某种别的提示、操作等。也就是我们更希望能在成员不存在的时候有所作为，不是等着报错。

要处理类似的问题，就要用到本节中的知识了。

`__getattr__`、`__setattr__` 和其它类似方法

还是用上面的例子，如果访问 `a.x`，它不存在，那么就要转向到某个操作。我们把这种情况称之为“拦截”。就好像“寻隐者不遇”，却被童子“遥指杏花村”，将你“拦截”了。在 Python 中，有一些方法就具有这种“拦截”能力。

- `__setattr__(self, name, value)`：如果要给 `name` 赋值，就调用这个方法。
- `__getattr__(self, name)`：如果 `name` 被访问，同时它不存在的时候，此方法被调用。
- `__getattribute__(self, name)`：当 `name` 被访问时自动被调用（注意：这个仅能用于新式类），无论 `name` 是否存在，都要被调用。
- `__delattr__(self, name)`：如果要删除 `name`，这个方法就被调用。

如果一时没有理解，不要紧，是正常的。需要用例子说明。

```
>>> class A(object):
...     def __getattr__(self, name):
```

```

...     print "You use getattr"
...     def __setattr__(self, name, value):
...         print "You use setattr"
...         self.__dict__[name] = value
...

```

类 A 是新式类，除了两个方法，没有别的属性。

```

>>> a = A()
>>> a.x
You use getattr

```

a.x，按照本节开头的例子，是要报错的。但是，由于在这里使用了 __getattr__(self, name) 方法，当发现 x 不存在于对象的 __dict__ 中的时候，就调用了 __getattribute__，即所谓“拦截成员”。

```

>>> a.x = 7
You use setattr

```

给对象的属性赋值时候，调用了 __setattr__(self, name, value) 方法，这个方法中有一句 self.__dict__[name] = value，通过这个语句，就将属性和数据保存到了对象的 __dict__ 中，如果在调用这个属性：

```

>>> a.x
7

```

它已经存在于对象的 __dict__ 之中。

在上面的类中，当然可以使用 __getattribute__(self, name)，因为它是新式类。并且，只要访问属性就会调用它。例如：

```

>>> class B(object):
...     def __getattribute__(self, name):
...         print "you are useing getattribute"
...         return object.__getattribute__(self, name)
...

```

为了与前面的类区分，新命名一个类名字。需要提醒注意，在这里返回的内容用的是 return object.__getattribute__(self, name)，而没有使用 return self.__dict__[name] 像是。因为如果用这样的方式，就是访问 self.__dict__，只要访问这个属性，就要调用`getattribute`，这样就导致了无线递归下去（死循环）。要避免之。

```

>>> b = B()
>>> b.y
you are useing getattribute
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 4, in __getattribute__

```

```
AttributeError: 'B' object has no attribute 'y'
>>> b.two
you are useing getattr
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __getattribute__
AttributeError: 'B' object has no attribute 'two'
```

访问不存在的成员，可以看到，已经被 `__getattribute__` 拦截了，虽然最后还是要报错的。

```
>>> b.y = 8
>>> b.y
you are useing getattr
8
```

当给其赋值后，意味着已经在 `__dict__` 里面了，再调用，依然被拦截，但是由于已经在 `__dict__` 内，会把结果返回。

当你看到这里，是不是觉得上面的方法有点魔力呢？不错。但是，它有什么具体应用呢？看下面的例子，能给你带来启发。

```
#!/usr/bin/env Python
# coding=utf-8

"""
study __getattr__ and __setattr__
"""

class Rectangle(object):
    """
    the width and length of Rectangle
    """

    def __init__(self):
        self.width = 0
        self.length = 0

    def setSize(self, size):
        self.width, self.length = size
    def getSize(self):
        return self.width, self.length

if __name__ == "__main__":
    r = Rectangle()
    r.width = 3
    r.length = 4
```

```
print r.getSize()
r.setSize( (30, 40) )
print r.width
print r.length
```

上面代码来自《Beginning Python:From Novice to Professional,Second Edition》(by Magnus Lie Hetland)，根据本教程的需要，稍作修改。

```
$ python 21301.py
(3, 4)
30
40
```

这段代码已经可以正确运行了。但是，作为一个精益求精的程序员。总觉得那种调用方式还有可以改进的空间。比如，要给长宽赋值的时候，必须赋予一个元组，里面包含长和宽。这个能不能改进一下呢？

```
#!/usr/bin/env Python
# coding=utf-8

"""
study __getattr__ and __setattr__
"""

class Rectangle(object):
    """
    the width and length of Rectangle
    """

    def __init__(self):
        self.width = 0
        self.length = 0

    def setSize(self, size):
        self.width, self.length = size

    def getSize(self):
        return self.width, self.length

    size = property(getSize, setSize)

if __name__ == "__main__":
    r = Rectangle()
    r.width = 3
    r.length = 4
    print r.size
    r.size = 30, 40
```

```
print r.width
print r.length
```

以上代码的运行结果同上。但是，因为加了一句 `size = property(getSize, setSize)`，使得调用方法是不是更优雅了呢？原来用 `r.getSize()`，现在使用 `r.size`，就好像调用一个属性一样。难道你不觉得眼熟吗？在《[多态和封装](#)》中已经用到过 `property` 函数了，虽然写法略有差别，但是作用一样。

本来，这样就已经足够了。但是，因为本节中出来了特殊方法，所以，一定要用这些特殊方法从新演绎一下这段程序。虽然重新演绎的不一定比原来的好，主要目的是演示本节的特殊方法应用。

```
#!/usr/bin/env Python
# coding=utf-8

class NewRectangle(object):
    def __init__(self):
        self.width = 0
        self.length = 0

    def __setattr__(self, name, value):
        if name == "size":
            self.width, self.length = value
        else:
            self.__dict__[name] = value

    def __getattr__(self, name):
        if name == "size":
            return self.width, self.length
        else:
            raise AttributeError

if __name__ == "__main__":
    r = NewRectangle()
    r.width = 3
    r.length = 4
    print r.size
    r.size = 30, 40
    print r.width
    print r.length
```

除了类的样式变化之外，调用样式没有变。结果是一样的。

这就算了解了一些这些属性了吧。但是，有一篇文章是要必须推荐给读者阅读的：[Python Attributes and Methods](#)，读了这篇文章，对 Python 的对象属性和方法会有更深入的理解。

获得属性顺序

通过实例获取其属性（也有说特性的，名词变化了，但是本质都是属性和方法），如果在 `__dict__` 中有相应的属性，就直接返回其结果；如果没有，会到类属性中找。比如：

```
#!/usr/bin/env Python
# coding=utf-8

class A(object):
    author = "qiwsir"
    def __getattr__(self, name):
        if name != "author":
            return "from starter to master."

if __name__ == "__main__":
    a = A()
    print a.author
    print a.lang
```

运行程序：

```
$ python 21302.py
qiwsir
from starter to master.
```

当 `a = A()` 后，并没有为实例建立任何属性，或者说实例的 `__dict__` 是空的，这在上节中已经探讨过了。但是如果要查看 `a.author`，因为实例的属性中没有，所以就去类属性中找，发现果然有，于是返回其值 `"qiwsir"`。但是，在找 `a.lang` 的时候，不仅实例属性中没有，类属性中也没有，于是就调用了 `__getattr__()` 方法。在上面的类中，有这个方法，如果没有 `__getattr__()` 方法呢？如果没有定义这个方法，就会引发 `AttributeError`，这在前面已经看到了。

这就是通过实例查找特性的顺序。

[总目录\(页0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

迭代器

迭代，对于读者已经不陌生了，曾有专门一节来讲述，如果印象不深，请复习《[迭代](#)》。

正如读者已知，对序列（列表、元组）、字典和文件都可以用 `iter()` 方法生成迭代对象，然后用 `next()` 方法访问。当然，这种访问不是自动的，如果用 `for` 循环，就可以自动完成上述访问了。

如果用 `dir(list)`, `dir(tuple)`, `dir(file)`, `dir(dict)` 来查看不同类型对象的属性，会发现它们都有一个名为 `__iter__` 的东西。这个应该引起读者的关注，因为它和迭代器（`iterator`）内置的函数 `iter()` 在名字上是一样的，除了前后的双下划线。望文生义，我们也能猜出它肯定是跟迭代有关的东西。当然，这种猜测也不是没有根据的，其重要根据就是英文单词，如果它们之间没有一点关系，肯定不会将命名搞得一样。

猜对了。`__iter__` 就是对象的一个特殊方法，它是迭代规则(iterator protocol)的基础。或者说，对象如果没有它，就不能返回迭代器，就没有 `next()` 方法，就不能迭代。

提醒注意，如果读者用的是 Python3.x，迭代器对象实现的是 `__next__()` 方法，不是 `next()`。并且，在 Python3.x 中有一个内建函数 `next()`，可以实现 `next(it)`，访问迭代器，这相当于于 python2.x 中的 `it.next()` (`it` 是迭代对象)。

那些类型是 `list`、`tuple`、`file`、`dict` 对象有 `__iter__()` 方法，标志着他们能够迭代。这些类型都是 Python 中固有的，我们能不能自己写一个对象，让它能够迭代呢？

当然呢！要不然 python 怎么强悍呢。

```
#!/usr/bin/env Python
# coding=utf-8

"""

the interator as range()

"""

class MyRange(object):
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def next(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

```

    self.i += 1
    return i
else:
    raise StopIteration()

if __name__ == "__main__":
    x = MyRange(7)
    print "x.next()=>", x.next()
    print "x.next()=>", x.next()
    print "-----for loop-----"
    for i in x:
        print i

```

将代码保存，并运行，结果是：

```

$ python 21401.py
x.next()=> 0
x.next()=> 1
-----for loop-----
2
3
4
5
6

```

以上代码的含义，是自己仿写了拥有 `range()` 的对象，这个对象是可迭代的。分析如下：

类 `MyRange` 的初始化方法 `__init__()` 就不用赘述了，因为前面已经非常详细分析了这个方法，如果复习，请阅读《[类\(2\)](#)》相关内容。

`__iter__()` 是类中的核心，它返回了迭代器本身。一个实现了 `__iter__()` 方法的对象，即意味着其实可迭代的。

含有 `next()` 的对象，就是迭代器，并且在这个方法中，在没有元素的时候要发起 `StopIteration()` 异常。

如果对以上类的调用换一种方式：

```

if __name__ == "__main__":
    x = MyRange(7)
    print list(x)
    print "x.next()=>", x.next()

```

运行后会出现如下结果：

```

$ python 21401.py
[0, 1, 2, 3, 4, 5, 6]

```

```
x.next()==>
Traceback (most recent call last):
  File "21401.py", line 26, in <module>
    print "x.next()==>", x.next()
  File "21401.py", line 21, in next
    raise StopIteration()
StopIteration
```

说明什么呢？`print list(x)` 将对象返回值都装进了列表中并打印出来，这个正常运行了。此时指针已经移动到了迭代对象的最后一个，正如在《迭代》中描述的那样，`next()` 方法没有检测也不知道是不是要停止了，它还要继续下去，当继续下一个的时候，才发现没有元素了，于是返回了 `StopIteration()`。

为什么要将用这种可迭代的对象呢？就像上面例子一样，列表不是挺好的吗？

列表的确非常好，在很多时候效率很高，并且能够解决相当普遍的问题。但是，不要忘记一点，在某些时候，列表可能会给你带来灾难。因为在你使用列表的时候，需要将列表内容一次性都读入到内存中，这样就增加了内存的负担。如果列表太大太大，就有内存溢出的危险了。这时候需要的是迭代对象。比如斐波那契数列（在本教程多处已经提到这个著名的数列：《练习》的练习 4（页 0））：

```
#!/usr/bin/env Python
# coding=utf-8
"""

compute Fibonacci by iterator
"""

__metaclass__ = type

class Fibs:
    def __init__(self, max):
        self.max = max
        self.a = 0
        self.b = 1

    def __iter__(self):
        return self

    def next(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib

if __name__ == "__main__":
```

```
fib = Fibs(5)
print list(fib)
```

运行结果是：

```
$ python 21402.py
[0, 1, 1, 2, 3, 5]
```

| 给读者一个思考问题：要在斐波那契数列中找出大于 1000 的最小的数，能不能在上述代码基础上改造得出呢？

关于列表和迭代器之间的区别，还有两个非常典型的内建函数：`range()` 和 `xrange()`，研究一下这两个的差异，会有所收获的。

```
range(...)
range(stop) -> list of integers
range(start, stop[, step]) -> list of integers

>>> dir(range)
['__call__', '__class__', '__cmp__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
```

从 `range()` 的帮助文档和方法中可以看出，它的结果是一个列表。但是，如果用 `help(xrange)` 查看：

```
class xrange(object)
| xrange(stop) -> xrange object
| xrange(start, stop[, step]) -> xrange object
|
| Like range(), but instead of returning a list, returns an object that
| generates the numbers in the range on demand. For looping, this is
| slightly faster than range() and more memory efficient.
```

`xrange()` 返回的是对象，并且进一步告诉我们，类似 `range()`，但不是列表。在循环的时候，它跟 `range()` 相比“slightly faster than range() and more memory efficient”，稍快并更高的内存效率（就是省内存呀）。查看它的方法：

```
>>> dir(xrange)
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__getitem__', '__hash__', '__init__', '__iter__', '
```

看到令人兴奋的 `__iter__` 了吗？说明它是可迭代的，它返回的是一个可迭代的对象。

也就是说，通过 `range()` 得到的列表，会一次性被读入内存，而 `xrange()` 返回的对象，则是需要一个数值才从返回一个数值。比如这样一个应用：

还记得 `zip()` 吗？

```
>>> a = ["name", "age"]
>>> b = ["qiwsir", 40]
```

```
>>> zip(a,b)
[('name', 'qiwsir'), ('age', 40)]
```

如果两个列表的个数不一样，就会以短的为准了，比如：

```
>>> zip(range(4), xrange(100000000))
[(0, 0), (1, 1), (2, 2), (3, 3)]
```

第一个 `range(4)` 产生的列表被读入内存；第二个是不是也太长了？但是不用担心，它根本不会产生那么长的列表，因为只需要前 4 个数值，它就提供前四个数值。如果你要修改为 `xrange(100000000)`，就要花费时间了，可以尝试一下哦。

迭代器的确有迷人之处，但是它也不是万能之物。比如迭代器不能回退，只能如过河的卒子，不断向前。另外，迭代器也不适合在多线程环境中对可变集合使用（这句话可能理解有困难，先混个脸熟吧，等你遇到多线程问题再说）。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

生成器

生成器（英文：generator）是一个非常迷人的东西，也常被认为是 Python 的高级编程技能。不过，我依然很乐意在这里跟读者——尽管你可能是一个初学者——探讨这个话题，因为我相信读者看本教程的目的，绝非仅仅将自己限制于初学者水平，一定有一颗不羁的心——要成为 Python 高手。那么，开始了解生成器吧。

还记得上节的“迭代器”吗？生成器和迭代器有着一定的渊源关系。生成器必须是可迭代的，诚然它又不仅仅是迭代器，但除此之外，又没有太多的别的用途，所以，我们可以把它理解为非常方便的自定义迭代器。

最这个关系实在感觉有点糊涂了。稍安勿躁，继续阅读即明了。

简单的生成器

```
>>> my_generator = (x*x for x in range(4))
```

这是不是跟列表解析很类似呢？仔细观察，它不是列表，如果这样的得到的是列表：

```
>>> my_list = [x*x for x in range(4)]
```

以上两的区别在于是 `[]` 还是 `()`，虽然是细小的差别，但是结果完全不一样。

```
>>> dir(my_generator)
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__',
'__iter__',
'__name__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'next',
'send', 'throw']
```

为了容易观察，我将上述结果进行了重新排版。是不是发现了在迭代器中必有的方法 `__iter__()` 和 `next()`，这说明它是迭代器。如果是迭代器，就可以用 `for` 循环来依次读出其值。

```
>>> for i in my_generator:
...     print i
...
0
1
4
9
>>> for i in my_generator:
...     print i
...
```

当第一遍循环的时候，将 my_generator 里面的值依次读出并打印，但是，当再读一次的时候，就发现没有任何结果。这种特性也正是迭代器所具有的。

如果对那个列表，就不一样了：

```
>>> for i in my_list:  
...     print i  
...  
0  
1  
4  
9  
>>> for i in my_list:  
...     print i  
...  
0  
1  
4  
9
```

难道生成器就是把列表解析中的 `[]` 换成 `()` 就行了吗？这仅仅是生成器的一种表现形式和使用方法罢了，仿照列表解析式的命名，可以称之为“生成器解析式”（或者：生成器推导式、生成器表达式）。

生成器解析式是有很多用途的，在不少地方替代列表，是一个不错的选择。特别是针对大量值的时候，如上节所说的，列表占内存较多，迭代器（生成器是迭代器）的优势就在于少占内存，因此无需将生成器（或者说是迭代器）实例化为一个列表，直接对其进行操作，方显示出其迭代的优势。比如：

```
>>> sum(i*i for i in range(10))  
285
```

请读者注意观察上面的 `sum()` 运算，不要以为里面少了一个括号，就是这么写。是不是很迷人？如果列表，你不得不：

```
>>> sum([i*i for i in range(10)])  
285
```

通过生成器解析式得到的生成器，掩盖了生成器的一些细节，并且适用领域也有限。下面就要剖析生成器的内部，深入理解这个魔法工具。

定义和执行过程

`yield`这个词在汉语中有“生产、出产”之意，在Python中，它作为一个关键词（你在变量、函数、类的名称中就不能用这个了），是生成器的标志。

```
>>> def g():
...     yield 0
...     yield 1
...     yield 2
...
>>> g
<function g at 0xb71f3b8c>
```

建立了一个非常简单的函数，跟以往看到的函数唯一不同的地方是用了三个`yield`语句。然后进行下面的操作：

```
>>> ge = g()
>>> ge
<generator object g at 0xb7200edc>
>>> type(ge)
<type 'generator'>
```

上面建立的函数返回值是一个生成器(generator)类型的对象。

```
>>> dir(ge)
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__iter__', '__name__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__str__', '__weakref__']
```

在这里看到了`__iter__()`和`next()`，说明它是迭代器。既然如此，当然可以：

```
>>> ge.next()
0
>>> ge.next()
1
>>> ge.next()
2
>>> ge.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

从这个简单例子中可以看出，那个含有`yield`关键词的函数返回值是一个生成器类型的对象，这个生成器对象就是迭代器。

我们把含有 `yield` 语句的函数称作生成器。生成器是一种用普通函数语法定义的迭代器。通过上面的例子可以看出，这个生成器（也是迭代器），在定义过程中并没有像上节迭代器那样写 `__iter__()` 和 `next()`，而是只要用了 `yield` 语句，那个普通函数就神奇般地成为了生成器，也就具备了迭代器的功能特性。

`yield` 语句的作用，就是在调用的时候返回相应的值。详细剖析一下上面的运行过程：

1. `ge = g()`：除了返回生成器之外，什么也没有操作，任何值也没有被返回。
2. `ge.next()`：直到这时候，生成器才开始执行，遇到了第一个 `yield` 语句，将值返回，并暂停执行（有的称之为挂起）。
3. `ge.next()`：从上次暂停的位置开始，继续向下执行，遇到 `yield` 语句，将值返回，又暂停。
4. `gen.next()`：重复上面的操作。
5. `gene.next()`：从上面的挂起位置开始，但是后面没有可执行的了，于是 `next()` 发出异常。

从上面的执行过程中，发现 `yield` 除了作为生成器的标志之外，还有一个功能就是返回值。那么它跟 `return` 这个返回值有什么区别呢？

yield

为了弄清楚 `yield` 和 `return` 的区别，我们写两个没有什么用途的函数：

```
>>> def r_return(n):
...     print "You taked me."
...     while n > 0:
...         print "before return"
...         return n
...         n -= 1
...         print "after return"
...
>>> rr = r_return(3)
You taked me.
before return
>>> rr
3
```

从函数被调用的过程可以清晰看出，`rr = r_return(3)`，函数体内的语句就开始执行了，遇到 `return`，将值返回，然后就结束函数体内的执行。所以 `return` 后面的语句根本没有执行。这是 `return` 的特点，关于此特点的详细说明请阅读《[函数\(2\)](#)》中的返回值相关内容。

下面将 `return` 改为 `yield`：

```

>>> def y_yield(n):
...     print "You taked me."
...     while n > 0:
...         print "before yield"
...         yield n
...         n -= 1
...         print "after yield"
...
...
>>> yy = y_yield(3)  #没有执行函数体内语句
>>> yy.next()      #开始执行
You taked me.
before yield
3          #遇到 yield，返回值，并暂停
>>> yy.next()      #从上次暂停位置开始继续执行
after yield
before yield
2          #又遇到 yield，返回值，并暂停
>>> yy.next()      #重复上述过程
after yield
before yield
1
>>> yy.next()
after yield      #没有满足条件的值，抛出异常
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

```

结合注释和前面对执行过程的分析，读者一定能理解 `yield` 的特点了，也深知与 `return` 的区别了。

一般的函数，都是止于 `return`。作为生成器的函数，由于有了 `yield`，则会遇到它挂起，如果还有 `return`，遇到它就直接抛出 `StopIteration` 异常而中止迭代。

斐波那契数列已经是老相识了。不论是循环、迭代都用它举例过，现在让我们还用它吧，只不过是要用上 `yield`：

```

#!/usr/bin/env Python
# coding=utf-8

def fibs(max):
    """
    斐波那契数列的生成器
    """
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1

```

```
if __name__ == "__main__":
    f = fibs(10)
    for i in f:
        print i,
```

运行结果如下：

```
$ python 21501.py
1 1 2 3 5 8 13 21 34 55
```

用生成器方式实现的斐波那契数列是不是跟以前的有所不同了呢？读者可以将本教程中已经演示过的斐波那契数列实现方式做一下对比，体会各种方法的差异。

经过上面的各种例子，已经明确，一个函数中，只要包含了 `yield` 语句，它就是生成器，也是迭代器。这种方式显然比前面写迭代器的类要简便多了。但，并不意味着上节的就被抛弃。是生成器还是迭代器，都是根据具体的使用情景而定。

生成器方法

在 python2.5 以后，生成器有了一个新特征，就是在开始运行后能够为生成器提供新的值。这就好似生成器和“外界”之间进行数据交流。

```
>>> def repeater(n):
...     while True:
...         n = (yield n)
...
>>> r = repeater(4)
>>> r.next()
4
>>> r.send("hello")
'hello'
```

当执行到 `r.next()` 的时候，生成器开始执行，在内部遇到了 `yield n` 挂起。注意在生成器函数中，`n = (yield n)` 中的 `yield n` 是一个表达式，并将结果赋值给 `n`，虽然不严格要求它必须用圆括号包裹，但是一般情况都这么做，请读者也追随这个习惯。

当执行 `r.send("hello")` 的时候，原来已经被挂起的生成器（函数）又被唤醒，开始执行 `n = (yield n)`，也就是讲 `send()` 方法发送的值返回。这就是在运行后能够为生成器提供值的含义。

如果接下来再执行 `r.next()` 会怎样？

```
>>> r.next()
```

什么也没有，其实就是返回了 None。按照前面的叙述，读者可以看到，这次执行 `r.next()`，由于没有传入任何值，`yield` 返回的就只能是 `None`。

还要注意，`send()` 方法必须在生成器运行后并挂起才能使用，也就是 `yield` 至少被执行一次。如果不是这样：

```
>>> s = repeater(5)
>>> s.send("how")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't send non-None value to a just-started generator
```

就报错了。但是，可将参数设为 `None`：

```
>>> s.send(None)
5
```

这是返回的是调用函数的时传入的值。

此外，还有两个方法：`close()` 和 `throw()`

- `throw(type, value=None, traceback=None)`: 用于在生成器内部（生成器的当前挂起处，或未启动时在定义处）抛出一个异常（在 `yield` 表达式中）。
- `close()`: 调用时不用参数，用于关闭生成器。

最后一句，你在编程中，不用生成器也可以。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qjwsir@126.com,不胜感激。

HTML



6

错误和异常

unity



HTML



错误和异常 (1)

虽然在前面的学习中，已经遇到了错误和异常问题，但是一直没有很认真的研究它。现在来近距离观察错误和异常。

错误

Python 中的错误之一是语法错误(syntax errors)，比如：

```
>>> for i in range(10)
File "<stdin>", line 1
  for i in range(10)
  ^
SyntaxError: invalid syntax
```

上面那句话因为缺少冒号`:`，导致解释器无法解释，于是报错。这个报错行为是由 Python 的语法分析器完成的，并且检测到了错误所在文件和行号（`File "<stdin>", line 1`），还以向上箭头`^`标识错误位置（后面缺少`:`），最后显示错误类型。

错误之二是在没有语法错误之后，会出现逻辑错误。逻辑错误可能会由于不完整或者不合法的输入导致，也可能是无法生成、计算等，或者是其它逻辑问题。

当 Python 检测到一个错误时，解释器就无法继续执行下去，于是抛出异常。

异常

看一个异常（让 0 做分母了，这是小学生都相信会有异常的）：

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

当 Python 抛出异常的时候，首先有“跟踪记录(Traceback)”，还可以给它取一个更优雅的名字“回溯”。后面显示异常的详细信息。异常所在位置（文件、行、在某个模块）。

最后一行是错误类型以及导致异常的原因。

下表中列出常见的异常

异常	描述
NameError	尝试访问一个没有申明的变量
ZeroDivisionError	除数为 0
SyntaxError	语法错误
IndexError	索引超出序列范围
KeyError	请求一个不存在的字典关键字
IOError	输入输出错误（比如你要读的文件不存在）
AttributeError	尝试访问未知的对象属性

为了能够深入理解，依次举例，展示异常的出现条件和结果。

NameError

```
>>> bar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'bar' is not defined
```

Python 中变量需要初始化，即要赋值。虽然不需要像某些语言那样声明，但是要赋值先。因为变量相当于一个标签，要把它贴到对象上才有意义。

ZeroDivisionError

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

貌似这样简单的错误时不会出现的，但在实际情境中，可能没有这么容易识别，所以，依然要小心为妙。

SyntaxError

```
>>> for i in range(10)
    File "<stdin>", line 1
      for i in range(10)
        ^
SyntaxError: invalid syntax
```

这种错误发生在 Python 代码编译的时候，当编译到这一句时，解释器不能将代码转化为 Python 字节码，就报错。只有改正才能继续。所以，它是在程序运行之前就会出现的（如果有错）。现在有不少编辑器都有语法校验功能，在你写代码的时候就能显示出语法的正误，这多少会对编程者有帮助。

IndexError

```
>>> a = [1,2,3]
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> d = {"python":"itdiffer.com"}
>>> d["java"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'java'
```

这两个都属于“鸡蛋里面挑骨头”类型，一定得报错了。不过在编程实践中，特别是循环的时候，常常由于循环条件设置不合理出现这种类型的错误。

IOError

```
>>> f = open("foo")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'foo'
```

如果你确认有文件，就一定要把路径写正确，因为你并没有告诉 Python 对你的 computer 进行全身搜索，所以，Python 会按照你指定位置去找，找不到就异常。

AttributeError

```
>>> class A(object): pass
...
>>> a = A()
>>> a.foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'foo'
```

属性不存在。这种错误前面多次见到。

其实，Python 内建的异常也仅仅上面几个，上面只是列出常见的异常中的几个。比如还有：

```
>>> range("aaa")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: range() integer end argument expected, got str.
```

总之，如果读者在调试程序的时候遇到了异常，不要慌张，这是好事情，是 Python 在帮助你修改错误。只要认真阅读异常信息，再用 `dir()`，`help()` 或者官方网站文档、google 等来协助，一定能解决问题。

处理异常

在一段程序中，为了能够让程序健壮，必须要处理异常。举例：

```
#!/usr/bin/env Python
# coding=utf-8

while 1:
    print "this is a division program."
    c = raw_input("input 'c' continue, otherwise logout:")
    if c == 'c':
        a = raw_input("first number:")
        b = raw_input("second number:")
        try:
            print float(a)/float(b)
            print "*****"
        except ZeroDivisionError:
            print "The second number can't be zero!"
            print "*****"
    else:
        break
```

运行这段程序，显示如下过程：

```
$ python 21601.py
this is a division program.
input 'c' continue, otherwise logout:c
first number:5
second number:2
2.5
*****
this is a division program.
input 'c' continue, otherwise logout:c
first number:5
second number:0
The second number can't be zero!
```

```
*****
this is a division program.
input 'c' continue, otherwise logout:d
$
```

从运行情况看，当在第二个数，即除数为 0 时，程序并没有因为这个错误而停止，而是给用户一个友好的提示，让用户有机会改正错误。这完全得益于程序中“处理异常”的设置，如果没有“处理异常”，异常出现，就会导致程序终止。

处理异常的方式之一，使用 `try...except...`。

对于上述程序，只看 `try` 和 `except` 部分，如果没有异常发生，`except` 子句在 `try` 语句执行之后被忽略；如果 `try` 语句中有异常可，该部分的其它语句被忽略，直接跳到 `except` 部分，执行其后面指定的异常类型及其子句。

`except` 后面也可以没有任何异常类型，即无异常参数。如果这样，不论 `try` 部分发生什么异常，都会执行 `except`。

在 `except` 子句中，可以根据异常或者别的需要，进行更多的操作。比如：

```
#!/usr/bin/env Python
# coding=utf-8

class Calculator(object):
    is_raise = False
    def calc(self, express):
        try:
            return eval(express)
        except ZeroDivisionError:
            if self.is_raise:
                print "zero can not be division."
            else:
                raise
```

在这里，应用了一个函数 `eval()`，它的含义是：

```
eval(...)
eval(source[, globals[, locals]]) -> value

Evaluate the source in the context of globals and locals.
The source may be a string representing a Python expression
or a code object as returned by compile().
The globals must be a dictionary and locals can be any mapping,
defaulting to the current globals and locals.
If only globals is given, locals defaults to it.
```

例如：

```
>>> eval("3+5")
8
```

另外，在 except 子句中，有一个 `raise`，作为单独一个语句。它的含义是将异常信息抛出。并且，except 子句用了一个判断语句，根据不同的情况确定走不同分支。

```
if __name__ == "__main__":
    c = Calculator()
    print c.calc("8/0")
```

这时候 `is_raise = False`，则会：

```
$ python 21602.py
Traceback (most recent call last):
File "21602.py", line 17, in <module>
    print c.calc("8/0")
File "21602.py", line 8, in calc
    return eval(express)
File "<string>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

如果将 `is_raise` 的值改为 True，就是这样了：

```
if __name__ == "__main__":
    c = Calculator()
    c.is_raise = True #通过实例属性修改
    print c.calc("8/0")
```

运行结果：

```
$ python 21602.py
zero can not be division.
None
```

最后的 None 是 `c.calc("8/0")` 的返回值，因为有 `print c.calc("8/0")`，所以被打印出来。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

错误和异常 (2)

try...except...是处理异常的基本方式。在原来的基础上，还可有扩展。

处理多个异常

处理多个异常，并不是因为同时报出多个异常。程序在运行中，只要遇到一个异常就会有反应，所以，每次捕获到的异常一定是一个。所谓处理多个异常的意思是可以容许捕获不同的异常，有不同的 except 子句处理。

```
#!/usr/bin/env Python
# coding=utf-8

while 1:
    print "this is a division program."
    c = raw_input("input 'c' continue, otherwise logout:")
    if c == 'c':
        a = raw_input("first number:")
        b = raw_input("second number:")
        try:
            print float(a)/float(b)
            print "*****"
        except ZeroDivisionError:
            print "The second number can't be zero!"
            print "*****"
        except ValueError:
            print "please input number."
            print "*****"
    else:
        break
```

将上节的一个程序进行修改，增加了一个 except 子句，目的是如果用户输入的不是数字时，捕获并处理这个异常。测试如下：

```
$ python 21701.py
this is a division program.
input 'c' continue, otherwise logout:c
first number:3
second number:"hello"      #输入了一个不是数字的东西
please input number.      #对照上面的程序，捕获并处理了这个异常
*****
this is a division program.
```

```
input 'c' continue, otherwise logout:c
first number:4
second number:0
The second number can't be zero!
*****
this is a division program.
input 'c' continue, otherwise logout:4
$
```

如果有多个 except，在 try 里面如果有一个异常，就转到相应的 except 子句，其它的忽略。如果 except 没有相应的异常，该异常也会抛出，不过这是程序就要中止了，因为异常“浮出”程序顶部。

除了用多个 except 之外，还可以在一个 except 后面放多个异常参数，比如上面的程序，可以将 except 部分修改为：

```
except (ZeroDivisionError, ValueError):
    print "please input rightly."
    print *****
```

运行的结果就是：

```
$ python 21701.py
this is a division program.
input 'c' continue, otherwise logout:c
first number:2
second number:0      #捕获异常
please input rightly.
*****
this is a division program.
input 'c' continue, otherwise logout:c
first number:3
second number:a      #异常
please input rightly.
*****
this is a division program.
input 'c' continue, otherwise logout:d
$
```

需要注意的是，except 后面如果是多个参数，一定要用圆括号包裹起来。否则，后果自负。

突然有一种想法，在对异常的处理中，前面都是自己写一个提示语，发现自己写的不如内置的异常错误提示更好。希望把它打印出来。但是程序还能不能中断。Python 提供了一种方式，将上面代码修改如下：

```
while 1:
    print "this is a division program."
    c = raw_input("input 'c' continue, otherwise logout:")
```

```

if c == 'c':
    a = raw_input("first number:")
    b = raw_input("second number:")
    try:
        print float(a)/float(b)
        print "*****"
    except (ZeroDivisionError, ValueError), e:
        print e
        print "*****"
else:
    break

```

运行一下，看看提示信息。

```

$ python 21702.py
this is a division program.
input 'c' continue, otherwise logout:c
first number:2
second number:a          #异常
could not convert string to float: a
*****
this is a division program.
input 'c' continue, otherwise logout:c
first number:2
second number:0          #异常
float division by zero
*****
this is a division program.
input 'c' continue, otherwise logout:d
$ 

```

| 在 Python3.x 中，常常这样写： `except (ZeroDivisionError, ValueError) as e:`

以上程序中，之处理了两个异常，还可能有更多的异常呢？如果要处理，怎么办？可以这样：`except:` 或者 `except Exception, e,`，后面什么参数也不写就好了。

else 子句

有了 `try...except...`，在一般情况下是够用的，但总有不一般的时候出现，所以，就增加了一个 `else` 子句。其实，人类的自然语言何尝不是如此呢？总要根据需要添加不少东西。

```

>>> try:
...     print "I am try"
... except:

```

```

...     print "I am except"
... else:
...     print "I am else"
...
I am try
I am else

```

这段演示，能够帮助读者理解 else 的执行特点。如果执行了 try，则 except 被忽略，但是 else 被执行。

```

>>> try:
...     print 1/0
... except:
...     print "I am except"
... else:
...     print "I am else"
...
I am except

```

这时候 else 就不被执行了。

理解了 else 的执行特点，可以写这样一段程序，还是类似于前面的计算，只不过这次要求，如果输入的有误，就不断要求从新输入，知道输入正确，并得到了结果，才不再要求输入内容，程序结束。

在看下面的参考代码之前，读者是否可以先自己写一段呢？并调试一下，看看结果如何。

```

#!/usr/bin/env Python
# coding=utf-8
while 1:
    try:
        x = raw_input("the first number:")
        y = raw_input("the second number:")

        r = float(x)/float(y)
        print r
    except Exception, e:
        print e
        print "try again."
    else:
        break

```

先看运行结果：

```

$ python 21703.py
the first number:2
the second number:0      #异常，执行except
float division by zero

```

```

try again.      #循环
the first number:2
the second number:a      #异常
could not convert string to float: a
try again.
the first number:4
the second number:2      #正常, 执行 try
2.0                      #然后 else: break, 退出程序
$
```

相当满意的执行结果。

需要对程序中的 except 简单说明, 这次没有像前面那样写, 而是 `except Exception, e`, 意思是不管什么异常, 这里都会捕获, 并且传给变量 `e`, 然后用 `print e` 把异常信息打印出来。

finally

`finally` 子句, 一听这个名字, 就感觉它是做善后工作的。的确如此, 如果有了 `finally`, 不管前面执行的是 `try`, 还是 `except`, 它都要执行。因此一种说法是用 `finally` 用来在可能的异常后进行清理。比如:

```

>>> x = 10

>>> try:
...     x = 1/0
... except Exception, e:
...     print e
... finally:
...     print "del x"
...     del x
...
integer division or modulo by zero
del x
```

看一看 `x` 是否被删除?

```

>>> x
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

当然, 在应用中, 可以将上面的各个子句都综合起来使用, 写成如下样式:

```

try:
    do something
```

```
except:  
    do something  
else:  
    do something  
finally  
    do something
```

和条件语句相比

`try...except...` 在某些情况下能够替代 `if...else...` 的条件语句。这里我无意去比较两者的性能，因为看到有人讨论这个问题。我个人觉得这不是主要的，因为它们之间性能的差异不大。主要是你的选择。一切要根据实际情况而定，不是说用一个就能包打天下。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

错误和异常 (3)

按照一般的学习思路，掌握了前两节内容，已经足够编程所需了。但是，我还想再多一步，还是因为本教程的读者是要 from beginner to master。

assert

```
>>> assert 1==1
>>> assert 1==0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

从上面的举例中可以基本了解了 assert 的特点。

assert，翻译过来是“断言”之意。assert 是一句等价于布尔真的判定，发生异常就意味着表达式为假。

assert 的应用情景就有点像汉语的意思一样，当程序运行到某个节点的时候，就断定某个变量的值必然是什么，或者对象必然拥有某个属性等，简单说就是断定什么东西必然是什么，如果不是，就抛出错误。

```
#!/usr/bin/env Python
# coding=utf-8

class Account(object):
    def __init__(self, number):
        self.number = number
        self.balance = 0

    def deposit(self, amount):
        assert amount > 0
        self.balance += balance

    def withdraw(self, amount):
        assert amount > 0
        if amount <= self.balance:
            self.balance -= amount
        else:
            print "balance is not enough."
```

上面的程序中，deposit() 和 withdraw() 方法的参数 amount 值必须是大于零的，这里就用断言，如果不满足条件就会报错。比如这样来运行：

```
if __name__ == "__main__":
    a = Account(1000)
    a.deposit(-10)
```

出现的结果是：

```
$ python 21801.py
Traceback (most recent call last):
  File "21801.py", line 22, in <module>
    a.deposit(-10)
  File "21801.py", line 10, in deposit
    assert amount > 0
AssertionError
```

这就是断言 assert 的引用。什么是使用断言的最佳时机？有文章做了总结：

如果没有特别的目的，断言应该用于如下情况：

- 防御性的编程
- 运行时对程序逻辑的检测
- 合约性检查（比如前置条件，后置条件）
- 程序中的常量
- 检查文档

(上述要点来自：[Python 使用断言的最佳时机](#))

不论是否理解，可以先看看，请牢记，在具体开发过程中，有时间就回来看看本教程，不断加深对这些概念的理解，这也是 master 的成就之法。

最后，引用危机百科中对“异常处理”词条的说明，作为对“错误和异常”部分的总结（有所删改）：

异常处理，是编程语言或计算机硬件里的一种机制，用于处理软件或信息系统中出现的异常状况（即超出程序正常执行流程的某些特殊条件）。

各种编程语言在处理异常方面具有非常显著的不同点（错误检测与异常处理区别在于：错误检测是在正常的程序流中，处理不可预见问题的代码，例如一个调用操作未能成功结束）。某些编程语言有这样的函数：当输入存在非法数据时不能被安全地调用，或者返回值不能与异常进行有效的区别。例如，C 语言中的 atoi 函数（ASCII 串到整数的转换）在输入非法时可以返回 0。在这种情况下编程者需要另外进行错误检测（可能通过某些辅助全局变量如 C 的 errno），或进行输入检验（如通过正则表达式），或者共同使用这两种方法。

通过异常处理，我们可以对用户在程序中的非法输入进行控制和提示，以防程序崩溃。

从进程的视角，硬件中断相当于可恢复异常，虽然中断一般与程序流本身无关。

从子程序编程者的视角，异常是很有用的一种机制，用于通知外界孩子程序不能正常执行。如输入的数据无效（例如除数是 0），或所需资源不可用（例如文件丢失）。如果系统没有异常机制，则编程者需要用返回值来标示发生了哪些错误。

一段代码是异常安全的，如果这段代码运行时的失败不会产生有害后果，如内存泄露、存储数据混淆、或无效的输出。

Python 语言对异常处理机制是非常普遍深入的，所以想写出不含 try, except 的程序非常困难。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

HTML



TP



模块



unity



HTML



编写模块

在本章之前，Python 还没有显示出太突出的优势。本章开始，读者就会越来越感觉到 Python 的强大了。这种强大体现在“模块自信”上，因为 Python 不仅有很强大的自有模块（称之为标准库），还有海量的第三方模块，任何人还都能自己开发模块，正是有了这么强大的“模块自信”，才体现了 Python 的优势所在。并且这种方式也正在不断被更多其它语言所借鉴。

“模块自信”的本质是：开放。

Python 不是一个封闭的体系，是一个开放系统。开放系统的最大好处就是避免了“熵增”。

熵的概念是由德国物理学家克劳修斯于 1865 年（这一年李鸿章建立了江南机械制造总局，美国废除奴隶制，林肯总统遇刺身亡，美国南北战争结束。）所提出。是一种测量在动力学方面不能做功的能量总数，也就是当总体的熵增加，其做功能力也下降，熵的量度正是能量退化的指标。

熵亦被用于计算一个系统中的失序现象，也就是计算该系统混乱的程度。

根据熵的统计学定义，热力学第二定律说明一个孤立系统的倾向于增加混乱程度。换句话说就是对于封闭系统而言，会越来越趋向于无序化。反过来，开放系统则能避免无序化。

回忆过去

在本教程的《语句(1)》中，曾经介绍了 import 语句，有这样一个例子：

```
>>> import math  
>>> math.pow(3,2)  
9.0
```

这里的 math 就是一个模块，用 import 引入这个模块，然后可以使用模块里面的函数，比如这个 pow() 函数。显然，这里我们是不需要自己动手写具体函数的，我们的任务就是拿过来使用。这就是模块的好处：拿过来就用，不用自己重写。

模块是程序

这个标题，一语道破了模块的本质，它就是一个扩展名为 .py 的 Python 程序。我们能够在应该使用它的时候将它引用过来，节省精力，不需要重写雷同的代码。

但是，如果我自己写一个 `.py` 文件，是不是就能作为模块 import 过来呢？还不那么简单。必须得让 Python 解释器能够找到你写的模块。比如：在某个目录中，我写了这样一个文件：

```
#!/usr/bin/env Python
# coding=utf-8

lang = "python"
```

并把它命名为 `pm.py`，那么这个文件就可以作为一个模块被引入。不过由于这个模块是我自己写的，Python 解释器并不知道，我得先告诉它我写了这样一个文件。

```
>>> import sys
>>> sys.path.append("~/Documents/VBS/StartLearningPython/2code/pm.py")
```

用这种方式就是告诉 Python 解释器，我写的那个文件在哪里。在这个告诉方法中，也用了一个模块 `import sys`，不过由于 `sys` 模块是 Python 被安装的时候就有的，所以不用特别告诉，Python 解释器就知道它在哪里了。

上面那个一长串的地址，是 ubuntu 系统的地址格式，如果读者使用的 windows 系统，请写你所保存的文件路径。

```
>>> import pm
>>> pm.lang
'python'
```

本来在 `pm.py` 文件中，有一个变量 `lang = "Python"`，这次它作为模块引入（注意作为模块引入的时候，不带扩展名），就可以通过模块名字来访问变量 `pm.py`，当然，如果不存在的属性这么去访问，肯定是要报错的。

```
>>> pm.xx
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'xx'
```

请读者回到 `pm.py` 文件的存储目录，是不是多了一个扩展名是 `.pyc` 的文件？如果不是，你那个可能是外星人用的 Python。

解释器，英文是：interpreter，港台翻译为：直译器。在 Python 中，它的作用就是将 `.py` 的文件转化为 `.pyc` 文件，而 `.pyc` 文件是由字节码(bytecode)构成的，然后计算机执行 `.pyc` 文件。关于这方面的详细解释，请参阅维基百科的词条：[直译器](#)

不少人喜欢将这个世界简化简化再简化。比如人，就分为好人还坏人，比如编程语言就分为解释型和编译型，不但如此，还将两种类型的语言分别贴上运行效率高低的标签，解释型的运行速度就慢，编译型的就快。一般人都把 Python 看成解释型的，于是就得出它运行速度慢的结论。不少人都因此上当受骗了，认为 Python 不值得

学，或者做不了什么“大事”。这就是将本来复杂的多样化的世界非得划分为“黑白”的结果。这种喜欢用“非此即彼”的思维方式考虑问题的现象可以说在现在很常见，比如一提到“日本人”，都该杀，这基本上是小孩子的思维方法，可惜在某个过度内大行其道。

世界是复杂的，“敌人的敌人就是朋友”是幼稚的，“一分为二”是机械的。

就如同刚才看到的那个 .pyc 文件一样，当 Python 解释器读取了 .py 文件，先将它变成由字节码组成的 .pyc 文件，然后这个 .pyc 文件交给一个叫做 Python 虚拟机的东西去运行（那些号称编译型的语言也是这个流程，不同的是它们先有一个明显的编译过程，编译好了之后再运行）。如果 .py 文件修改了，Python 解释器会重新编译，只是这个编译过程不是完全显示给你看的。

我这里说的比较笼统，要深入了解 Python 程序的执行过程，可以阅读这篇文章：[说说 Python 程序的执行过程](#)

总之，有了 .pyc 文件后，每次运行，就不需要从新让解释器来编译 .py 文件了，除非 .py 文件修改了。这样，Python 运行的就是那个编译好了的 .pyc 文件。

是否还记得，我们在前面写有关程序，然后执行，常常要用到 `if __name__ == "__main__"`。那时我们写的 .py 文件是来执行的，这时我们同样写了 .py 文件，是作为模块引入的。这就得深入探究一下，同样是 .py 文件，它是怎么知道是被当做程序执行还是被当做模块引入？

为了便于比较，将 pm.py 文件进行改造，稍微复杂点。

```
#!/usr/bin/env Python
# coding=utf-8

def lang():
    return "Python"

if __name__ == "__main__":
    print lang()
```

如以前做的那样，可以用这样的方式：

```
$ Python pm.py
python
```

但是，如果将这个程序作为模块，导入，会是这样的：

```
>>> import sys
>>> sys.path.append("~/Documents/VBS/StarterLearningPython/2code/pm.py")
>>> import pm
>>> pm.lang()
'python'
```

因为这时候 pm.py 中的函数 lang() 就是一个属性：

```
>>> dir(pm)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'lang']
```

同样一个 .py 文件，可以把它当做程序来执行，还可以将它作为模块引入。

```
>>> __name__
'__main__'
>>> pm.__name__
'pm'
```

如果要作为程序执行，则 `__name__ == "__main__"`；如果作为模块引入，则 `pm.__name__ == "pm"`，即变量 `__name__` 的值是模块名称。

用这种方式就可以区分是执行程序还是作为模块引入了。

在一般情况下，如果仅仅是用作模块引入，可以不写 `if __name__ == "__main__"`。

模块的位置

为了让我们自己写的模块能够被 Python 解释器知道，需要用 `sys.path.append("~/Documents/VBS/StarterLearningPython/2code/pm.py")`。其实，在 Python 中，所有模块都被加入到了 `sys.path` 里面了。用下面的方法可以看到模块所在位置：

```
>>> import sys
>>> import pprint
>>> pprint.pprint(sys.path)
[",
 '/usr/local/lib/python2.7/dist-packages/autopep8-1.1-py2.7.egg',
 '/usr/local/lib/python2.7/dist-packages/pep8-1.5.7-py2.7.egg',
 '/usr/lib/python2.7',
 '/usr/lib/python2.7/plat-i386-linux-gnu',
 '/usr/lib/python2.7/lib-tk',
 '/usr/lib/python2.7/lib-old',
 '/usr/lib/python2.7/lib-dynload',
 '/usr/local/lib/python2.7/dist-packages',
 '/usr/lib/python2.7/dist-packages',
 '/usr/lib/python2.7/dist-packages/PILcompat',
 '/usr/lib/python2.7/dist-packages/gtk-2.0',
 '/usr/lib/python2.7/dist-packages/ubuntu-sso-client',
 '~/Documents/VBS/StarterLearningPython/2code/pm.py']
```

从中也发现了我们自己写的那个文件。凡在上面列表所包括位置内的 .py 文件都可以作为模块引入。不妨举个例子。把前面自己编写的 pm.py 文件修改为 pmlib.py，然后把它复制到 /usr/lib/Python2.7/dist-packages 中。（这是以 ubuntu 为例说明，如果是其它操作系统，读者用类似方法也能找到。）

```
$ sudo cp pm.py /usr/lib/python2.7/dist-packages/pmlib.py
[sudo] password for qw:

$ ls /usr/lib/python2.7/dist-packages/pm*
/usr/lib/Python2.7/dist-packages/pmlib.py
```

文件放到了指定位置。看下面的：

```
>>> import pmlib
>>> pmlib.lang
<function lang at 0xb744372c>
>>> pmlib.lang()
'python'
```

也就是，要将模块文件放到合适的位置——就是 sys.path 包括位置——就能够直接用 import 引入了。

PYTHONPATH 环境变量

将模块文件放到指定位置是一种不错的方法。当程序员都喜欢自由，能不能放到别处呢？当然能，用 sys.path.append() 就是不管把文件放哪里，都可以把其位置告诉 Python 解释器。但是，这种方法不是很常用。因为它也有麻烦的地方，比如在交互模式下，如果关闭了，然后再开启，还得从新告知。

比较常用的告知方法是设置 PYTHONPATH 环境变量。

环境变量，不同操作系统的设置方法略有差异。读者可以根据自己的操作系统，到网上搜索设置方法。

我以 ubuntu 为例，建立一个 Python 的目录，然后将我自己写的 .py 文件放到这里，并设置环境变量。

```
:~$ mkdir Python
:~$ cd python
:~/Python$ cp ~/Documents/VBS/StarterLearningPython/2code/pm.py mypm.py
:~/Python$ ls
mypm.py
```

然后将这个目录 ~/Python，也就是 /home/qw/Python 设置环境变量。

```
vim /etc/profile
```

提醒要用 root 权限，在打开的文件最后增加 export PATH = /home/qw/python:\$PAT，然后保存退出即可。

注意，我是在 `~/Python` 目录下输入 `Python`，进入到交互模式：

```
:$ cd Python  
:/python$ Python  
  
>>> import mypm  
>>> mypm.lang()  
'Python'
```

如此，就完成了告知过程。

__init__.py 方法

`__init__.py` 是一个空文件，将它放在某个目录中，就可以将该目录中的其它 `.py` 文件作为模块被引用。这个具体应用参见[用 tornado 做网站\(2\)](#)

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

标准库 (1)

“Python 自带 ‘电池’ ”，听说过这种说法吗？

在 Python 被安装的时候，就有不少模块也随着安装到本地的计算机上了。这些东西就如同“能源”、“电力”一样，让 Python 拥有了无限生机，能够非常轻而易举地免费使用很多模块。所以，称之为“自带电池”。

它们被称为“标准库”。

熟悉标准库，是进行编程的必须。

引用的方式

不仅使标准库的模块，所有模块都服从下述引用方式。

最基本的、也是最常用的，还是可读性非常好的：

```
import modulename
```

例如：

```
>>> import pprint
>>> a = {"lang": "Python", "book": "www.itdiffer.com", "teacher": "qiwsir", "goal": "from beginner to master"}
>>> pprint.pprint(a)
{'book': 'www.itdiffer.com',
 'goal': 'from beginner to master',
 'lang': 'python',
 'teacher': 'qiwsir'}
```

在对模块进行说明的过程中，我以标准库 pprint 为例。以 `pprint.pprint()` 的方式应用了一种方法，这种方法能够让 dict 格式化输出。看看结果，是不是比原来更容易阅读了你？

在 `import` 后面，理论上可以跟好多模块名称。但是在实践中，我还是建议大家一次一个名称吧。这样简单明了，容易阅读。

这是用 `import pprint` 样式引入模块，并以 `.` 点号的形式引用其方法。

还可以：

```
>>> from pprint import pprint
```

意思是将 `pprint` 模块中之将 `pprint()` 引入，然后就可以这样来应用它：

```
>>> pprint(a)
{'book': 'www.itdiffer.com',
'goal': 'from beginner to master',
'lang': 'Python',
'teacher': 'qiwsir'}
```

再懒惰一些，可以：

```
>>> from pprint import *
```

这就将 pprint 模块中的一切都引入了，于是可以像上面那样直接使用每个函数。但是，这样造成的结果是可读性不是很好，并且，有用没用的都拿过来，是不是太贪婪了？贪婪的结果是内存就消耗了不少。所以，这种方法，可以用于常用并且模块属性或方法不是很多的情况。

诚然，如果很明确使用那几个，那么使用类似 `from modulename import name1, name2, name3...` 也未尝不可。一再提醒的是不能因为引入了模块东西而降低了可读性，让别人不知道呈现在眼前的方法是从何而来。如果这样，就要慎用这种方法。

有时候引入的模块或者方法名称有点长，可以给它重命名。如：

```
>>> import pprint as pr
>>> pr pprint(a)
{'book': 'www.itdiffer.com',
'goal': 'from beginner to master',
'lang': 'python',
'teacher': 'qiwsir'}
```

当然，还可以这样：

```
>>> from pprint import pprint as pt
>>> pt(a)
{'book': 'www.itdiffer.com',
'goal': 'from beginner to master',
'lang': 'python',
'teacher': 'qiwsir'}
```

但是不管怎么样，一定要让人看懂，过了若干时间，自己也还能看懂。记住：“软件很多时候是给人看的，只是偶尔让机器执行”。

深入探究

继续以 pprint 为例，深入研究：

```
>>> import pprint
>>> dir(pprint)
['PrettyPrinter', '_StringIO', '__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__', '_commajoin', '_id', '_']
```

对 `dir()` 并不陌生。从结果中可以看到 `pprint` 的属性和方法。其中有不少是双划线、电话线开头的。为了不影响我们的视觉，先把它们去掉。

```
>>> [m for m in dir(pprint) if not m.startswith('__')]
['PrettyPrinter', 'isreadable', 'isrecursive', 'pformat', 'pprint', 'saferepr', 'warnings']
```

对这几个，为了能够搞清楚它们的含义，可以使用 `help()`，比如：

```
>>> help(isreadable)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'isreadable' is not defined
```

这样做是错误的。知道错在何处吗？

```
>>> help(pprint.isreadable)
```

别忘记了，我前面是用 `import pprint` 方式引入模块的。

```
Help on function isreadable in module pprint:

isreadable(object)
    Determine if saferepr(object) is readable by eval().
```

通过帮助信息，能够查看到该方法的详细说明。可以用这种方法一个一个地查过来，反正也不多，对每个方法都熟悉一些。

注意的是 `pprint.PrettyPrinter` 是一个类，后面的是函数（方法）。

在回头看看 `dir(pprint)` 的结果，关注一个：

```
>>> pprint.__all__
['pprint', 'pformat', 'isreadable', 'isrecursive', 'saferepr', 'PrettyPrinter']
```

这个结果是不是眼熟？除了“`warnings`”，跟前面通过列表解析式得到的结果一样。

其实，当我们使用 `from pprint import *` 的时候，就是将 `__all__` 里面的方法引入，如果没有这个，就会将其它所有属性、方法等引入，包括那些以双划线或者单划线开头的变量、函数，这些东西事实上很少被在引入模块时使用。

帮助、文档和源码

不知道读者是否能够记住看过的上述内容？反正我记不住。所以，我非常喜欢使用 `dir()` 和 `help()`，这也是本教程从开始到现在，乃至到以后，总在提倡的方式。

```
>>> print pprint.__doc__
Support to pretty-print lists, tuples, & dictionaries recursively.

Very simple, but useful, especially in debugging data structures.

Classes
-----
PrettyPrinter()
Handle pretty-printing operations onto a stream using a configured
set of formatting parameters.

Functions
-----
pformat()
Format a Python object into a pretty-printed representation.

pprint()
Pretty-print a Python object to a stream [default is sys.stdout].

saferepr()
Generate a 'standard' repr()-like value, but protect against recursive
data structures.
```

`pprint.__doc__` 是查看整个类的文档，还知道整个文档是写在什么地方的吗？

关于文档的问题，曾经在[《类\(5\)》\(页 0\)](#)中有介绍。但是，现在出现的是模块文档。

还是使用 `pm.py` 那个文件，增加如下内容：

```
#!/usr/bin/env Python
# coding=utf-8

"""
#增加的
This is a document of the python module. #增加的
#增加的
```

```
def lang():
    ...
        #省略了，后面的也省略了
```

在这个文件的开始部分，所有类和方法、以及 import 之前，写一个用三个引号包括的字符串。那就是文档。

```
>>> import sys
>>> sys.path.append("~/Documents/VBS/StarterLearningPython/2code")
>>> import pm
>>> print pm.__doc__
```

```
This is a document of the python module.
```

这就是撰写模块文档的方法，即在 .py 文件的最开始写相应的内容。这个要求应该成为开发习惯。

Python 的模块，不仅可以看帮助信息和文档，还能够查看源码，因为它是开放的。

还是回头到 `dir(pprint)` 中找一找，有一个 `__file__`，它就告诉我们这个模块的位置：

```
>>> print pprint.__file__
/usr/lib/python2.7/pprint.pyc
```

我是在 ubuntu 中为例，读者要注意观察自己的操作系统结果。

虽然是 .pyc 文件，但是不用担心，根据现实的目录，找到相应的 .py 文件即可。

```
$ ls /usr/lib/python2.7/pp*
/usr/lib/python2.7/pprint.py /usr/lib/python2.7/pprint.pyc
```

果然有一个 pprint.py。打开它，就看到源码了。

```
$ cat /usr/lib/python2.7/pprint.py
...
"""
Support to pretty-print lists, tuples, & dictionaries recursively.

Very simple, but useful, especially in debugging data structures.
```

Classes

PrettyPrinter()

Handle pretty-printing operations onto a stream using a configured set of formatting parameters.

Functions

```
pformat()  
Format a Python object into a pretty-printed representation.
```

```
....  
.....
```

我只查抄了文档中的部分信息，是不是跟前面通过 `__doc__` 查看的结果一样一样的呢？

请读者在闲暇时间，阅读以下源码。事实证明，这种标准库中的源码是质量最好的。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

标准库 (2)

Python 标准库内容非常多，有人专门为此写过一本书。在本教程中，由于我的原因，不会将标准库进行完整的详细介绍，但是，我根据自己的理解和喜好，选几个呈现出来，一来显示标准库之强大功能，二来演示如何理解和使用标准库。

sys

这是一个跟 Python 解释器关系密切的标准库，上一节中我们使用过 `sys.path.append()`。

```
>>> import sys
>>> print sys.__doc__
```

显示了 `sys` 的基本文档，看第一句话，概括了本模块的基本特点。

This module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.

在诸多 `sys` 函数和变量中，选择常用的（应该说是我觉得常用的）来说明。

sys.argv

`sys.argv` 是变量，专门用来向 Python 解释器传递参数，所以名曰“命令行参数”。

先解释什么是命令行参数。

```
$ Python --version
Python 2.7.6
```

这里的 `--version` 就是命令行参数。如果你使用 `Python --help` 可以看到更多：

```
$ Python --help
usage: Python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-B   : don't write .py[co] files on import; also PYTHONDONOTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d   : debug output from parser; also PYTHONDEBUG=x
-E   : ignore PYTHON* environment variables (such as PYTHONPATH)
-h   : print this help message and exit (also --help)
-i   : inspect interactively after running script; forces a prompt even
```

```

if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-m mod : run library module as a script (terminates option list)
-O   : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x
-OO  : remove doc-strings in addition to the -O optimizations
-R   : use a pseudo-random salt to make hash() values of various types be
      unpredictable between separate invocations of the interpreter, as
      a defense against denial-of-service attacks

```

只选择了部分内容摆在这里。所看到的如 `-B, -h` 之流，都是参数，比如 `Python -h`，其功能同上。那么 `-h` 也是命令行参数。

`sys.argv` 在 Python 中的作用就是这样。通过它可以向解释器传递命令行参数。比如：

```

#!/usr/bin/env Python
# coding=utf-8

import sys

print "The file name: ", sys.argv[0]
print "The number of argument", len(sys.argv)
print "The argument is: ", str(sys.argv)

```

将上述代码保存，文件名是 `22101.py`（这名称取的，多么数字化）。然后如此做：

```

$ python 22101.py
The file name: 22101.py
The number of argument 1
The argument is: ['22101.py']

```

将结果和前面的代码做个对照。

- 在 `$ Python 22101.py` 中，“`22101.py`”是要运行的文件名，同时也是命令行参数，是前面的 `Python` 这个指令的参数。其地位与 `Python -h` 中的参数 `-h` 是等同的。
- `sys.argv[0]` 是第一个参数，就是上面提到的 `22101.py`，即文件名。

如果我们这样来试试，看看结果：

```

$ python 22101.py beginner master www.itdiffer.com
The file name: 22101.py
The number of argument 4
The argument is: ['22101.py', 'beginner', 'master', 'www.itdiffer.com']

```

如果在这里，用 `sys.argv[1]` 得到的就是 `beginner`，依次类推。

sys.exit()

这是一个方法，意思是退出当前的程序。

```
Help on built-in function exit in module sys:
```

```
exit(...)  
    exit([status])
```

Exit the interpreter by raising SystemExit(status).

If the status is omitted or None, it defaults to zero (i.e., success).

If the status is an integer, it will be used as the system exit status.

If it is another kind of object, it will be printed and the system
exit status will be one (i.e., failure).

从文档信息中可知，如果用 `sys.exit()` 退出程序，会返回 `SystemExit` 异常。这里先告知读者，还有另外一退出方式，是 `os._exit()`，这两个有所区别。后者会在后面介绍。

```
#!/usr/bin/env Python  
# coding=utf-8  
  
import sys  
  
for i in range(10):  
    if i == 5:  
        sys.exit()  
    else:  
        print i
```

这段程序的运行结果就是：

```
$ python 22102.py  
0  
1  
2  
3  
4
```

需要提醒读者注意的是，在函数中，用到 `return`，这个的含义是终止当前的函数，并返回相应值（如果有，如果没有就是 `None`）。但是 `sys.exit()` 的含义是退出当前程序，并发起 `SystemExit` 异常。这就是两者的区别了。

如果使用 `sys.exit(0)` 表示正常退出。如果读者要测试，需要在某个地方退出的时候有一个有意义的提示，可以用 `sys.exit("I wet out at here.")`，那么字符串信息就被打印出来。

sys.path

sys.path 已经不陌生了，前面用过。它可以查找模块所在的目录，以列表的形式显示出来。如果用 append() 方法，就能够向这个列表增加新的模块目录。如前所演示。不再赘述。不理解的读者可以往前复习。

sys.stdin, sys.stdout, sys.stderr

这三个放到一起，因为他们的变量都是类文件流对象，分别表示标准 UNIX 概念中的标准输入、标准输出和标准错误。与 Python 功能对照，sys.stdin 获得输入（用 raw_input() 输入的通过它获得，Python3.x 中是 input()），sys.stdout 负责输出了。

流是程序输入或输出的一个连续的字节序列，设备(例如鼠标、键盘、磁盘、屏幕、调制解调器和打印机)的输入和输出都是用流来处理的。程序在任何时候都可以使用它们。一般来讲，stdin (输入) 并不一定来自键盘，stdout (输出) 也并不一定显示在屏幕上，它们都可以重定向到磁盘文件或其它设备上。

还记得 print() 吧，在这个学习过程中，用的很多。它的本质就是 sys.stdout.write(object + '\n')。

```
>>> for i in range(3):
...     print i
...
0
1
2

>>> import sys
>>> for i in range(3):
...     sys.stdout.write(str(i))
...
012>>>
```

造成上面输出结果在表象上如此差异，原因就是那个 '\n' 的有无。

```
>>> for i in range(3):
...     sys.stdout.write(str(i) + '\n')
...
0
1
2
```

从这看出，两者是完全等效的。如果仅仅止于此，意义不大。关键是通过 `sys.stdout` 能够做到将输出内容从“控制台”转到“文件”，称之为重定向。这样也许控制台看不到（很多时候这个不重要），但是文件中已经有了输出内容。比如：

```
>>> f = open("stdout.md", "w")
>>> sys.stdout = f
>>> print "Learn Python: From Beginner to Master"
>>> f.close()
```

当 `sys.stdout = f` 之后，就意味着将输出目的地转到了打开（建立）的文件中，如果使用 `print()`，即将内容输出到这个文件中，在控制台并无显现。

打开文件看看便知：

```
$ cat stdout.md
Learn Python: From Beginner to Master
```

这是标准输出。另外两个，输入和错误，也类似。读者可以自行测试。

关于对文件的操作，虽然前面这里都涉及到一些。但是，远远不足，后面我会专门讲授对某些特殊但常用的文件读写操作。

copy

在《字典(2)》中曾经对 `copy` 做了讲授，这里再次提出，即是复习，又是凑数，以显得我考虑到了这个常用模块，还有：

```
>>> import copy
>>> copy.__all__
['Error', 'copy', 'deepcopy']
```

这个模块中常用的就是 `copy` 和 `deepcopy`。

为了具体说明，看这样一个例子：

```
#!/usr/bin/env Python
# coding=utf-8

import copy

class MyCopy(object):
    def __init__(self, value):
        self.value = value
```

```
def __repr__(self):
    return str(self.value)

foo = MyCopy(7)

a = ["foo", foo]
b = a[:]
c = list(a)
d = copy.copy(a)
e = copy.deepcopy(a)

a.append("abc")
foo.value = 17

print "original: %r\n slice: %r\n list(): %r\n copy(): %r\n deepcopy(): %r" % (a,b,c,d,e)
```

保存并运行：

```
$ python 22103.py
original: ['foo', 17, 'abc']
slice: ['foo', 17]
list(): ['foo', 17]
copy(): ['foo', 17]
deepcopy(): ['foo', 7]
```

读者可以对照结果和程序，就能理解各种拷贝的实现方法和含义了。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

标准库 (3)

OS

os 模块提供了访问操作系统服务的功能，它所包含的内容比较多。

```
>>> import os  
>>> dir(os)  
['EX_CANTCREATE', 'EX_CONFIG', 'EX_DATAERR', 'EX_IOERR', 'EX_NOHOST', 'EX_NOINPUT', 'EX_NOPERM', 'EX_N
```

这么多内容不能都介绍，况且不少方法在实践中用的不多，比如 `os.popen()` 在实践中用到了，但是 `os` 模块还有 `popen2`、`popen3`、`popen4`，这三个我在实践中都没有用过，或者有别人用了，也请补充。不过，我下面介绍的都是自认为用的比较多的，至少是我用的比较多或者用过的。如果没有读者要是用，但是我这里没有介绍，读者也完全可以自己用我们常用的 `help()` 来自学明白其应用方法，当然，还有最好的工具——`google`（内事不决问 `google`，外事不明问谷歌，须梯子）。

操作文件：重命名、删除文件

在对文件操作的时候，`open()` 这个内建函数可以建立、打开文件。但是，如果对文件进行改名、删除操作，就一定要用 `os` 模块的方法了。

首先建立一个文件，文件名为 22201.py，文件内容是：

```
#!/usr/bin/env python  
# coding=utf-8  
  
print "This is a tmp file"
```

然后将这个文件名称修改为其它的名称。

```
>>> import os  
>>> os.rename("22201.py", "newtemp.py")
```

注意，我是先进入到了文件 22201.py 的目录，然后进入到 Python 交互模式，所以，可以直接写文件名，如果不是这样，需要将文件名的路径写上。`os.rename("22201.py", "newtemp.py")` 中，第一个文件是原文件名，第二个是打算修改成为的文件名。

```
$ ls new*
```

查看，能够看到这个文件。并且文件内容可以用 `cat newtemp.py` 看看（这是在 ubuntu 系统，如果是 windows 系统，可以用其相应的编辑器打开文件看内容）。

```
Help on built-in function rename in module posix:
```

```
rename(...)  
    rename(old, new)
```

Rename a file or directory.

除了修改文件名称，还可以修改目录名称。请注意阅读帮助信息。

另外一个 `os.remove()`，首先看帮助信息，然后再实验。

```
Help on built-in function remove in module posix:
```

```
remove(...)  
    remove(path)
```

Remove a file (same as `unlink(path)`).

比较简单。那就测试一下。为了测试，先建立一些文件吧。

```
$ pwd  
/home/qw/Documents/VBS/StarterLearningPython/2code/rd
```

这是我建立的临时目录，里面有几个文件：

```
$ ls  
a.py b.py c.py
```

下面删除 `a.py` 文件

```
>>> import os  
>>> os.remove("/home/qw/Documents/VBS/StarterLearningPython/2code/rd/a.py")
```

看看删了吗？

```
$ ls  
b.py c.py
```

果然管用呀。再来一个狠的：

```
>>> os.remove("/home/qw/Documents/VBS/StarterLearningPython/2code/rd")  
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
OSError: [Errno 21] ls a directory: '/home/qw/Documents/VBS/StarterLearningPython/2code/rd'
```

报错了。我打算将这个目录下的所剩文件删光光。这么做不行。注意帮助中一句话 `Remove a file`，`os.remove()` 就是用来删除文件的。并且从报错中也可以看到，告诉我们错误的原因在于那个参数是一个目录。

要删除目录，还得继续向下学习。

操作目录

`os.listdir`: 显示目录中的文件

```
Help on built-in function listdir in module posix:
```

```
listdir(...)
listdir(path) -> list_of_strings
```

Return a list containing the names of the entries in the directory.

`path`: path of directory to list

The list is in arbitrary order. It does not include the special entries '.' and '..' even if they are present in the directory.

看完帮助信息，读者一定觉得这是一个非常简单的方法，不过，特别注意它返回的值是列表，还有就是如果文件夹中有那样的特殊格式命名的文件，不显示。在 linux 中，用 `ls` 命令也看不到这些隐藏的东东。

```
>>> os.listdir("/home/qw/Documents/VBS/StarterLearningPython/2code/rd")
['b.py', 'c.py']
>>> files = os.listdir("/home/qw/Documents/VBS/StarterLearningPython/2code/rd")
>>> for f in files:
...     print f
...
b.py
c.py
```

`os.getcwd`, `os.chdir`: 当前工作目录，改变当前工作目录

这两个函数怎么用？惟有通过 `help()` 看文档啦。请读者自行看看。我就不贴出来了，仅演示一个例子：

```
>>> cwd = os.getcwd()  #当前目录
>>> print cwd
/home/qw/Documents/VBS/StarterLearningPython/2code/rd
>>> os.chdir(os.pardir)  #进入到上一级
```

```
>>> os.getcwd()      #当前
'/home/qw/Documents/VBS/StarterLearningPython/2code'

>>> os.chdir("rd")    #进入下级

>>> os.getcwd()
'/home/qw/Documents/VBS/StarterLearningPython/2code/rd'
```

`os.pardir` 的功能是获得父级目录，相当于 `..`

```
>>> os.pardir
'..'
```

`os.makedirs, os.removedirs`: 创建和删除目录

废话少说，路子还是前面那样，就省略看帮助了，读者可以自己看。直接上例子：

```
>>> dir = os.getcwd()
>>> dir
'/home/qw/Documents/VBS/StarterLearningPython/2code/rd'
>>> os.removedirs(dir)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.7/os.py", line 170, in removedirs
    rmdir(name)
OSError: [Errno 39] Directory not empty: '/home/qw/Documents/VBS/StarterLearningPython/2code/rd'
```

什么时候都不能得意忘形，一定要谦卑。那就是从看文档开始一点一点地理解。不能像上面那样，自以为是、贸然行事。看报错信息，要删除某个目录，那个目录必须是空的。

```
>>> os.getcwd()
'/home/qw/Documents/VBS/StarterLearningPython/2code'
```

这是当前目录，在这个目录下再建一个新的子目录：

```
>>> os.makedirs("newrd")
>>> os.chdir("newrd")
>>> os.getcwd()
'/home/qw/Documents/VBS/StarterLearningPython/2code/newrd'
```

建立了一个。下面把这个删除了。这个是空的。

```
>>> os.listdir(os.getcwd())
[]
>>> newdir = os.getcwd()
>>> os.removedirs(newdir)
```

按照我的理解，这里应该报错。因为我在当前工作目录删除当前工作目录。如果这样能够执行，总觉得有点别扭。但事实上，就行得通了。就算是 python 的规定吧。不过，让我来确定这个功能的话，还是习惯不能在本地删除本地。

按照上面的操作，在看当前工作目录：

```
>>> os.getcwd()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 2] No such file or directory
```

目录被删了，当然没有啦。只能回到父级。

```
>>> os.chdir(os.pardir)
>>> os.getcwd()
'/home/qw/Documents/VBS/StarterLearningPython/2code'
```

有点不可思议。本来没有当前工作目录，怎么会有“父级”的呢？但 Python 就是这样。

补充一点，前面说的如果目录不空，就不能用 `os.removedirs()` 删除。但是，可以用模块 `shutil` 的 `rmtree` 方法。

```
>>> os.getcwd()
'/home/qw/Documents/VBS/StarterLearningPython/2code'
>>> os.chdir("rd")
>>> now = os.getcwd()
>>> now
'/home/qw/Documents/VBS/StarterLearningPython/2code/rd'
>>> os.listdir(now)
['b.py', 'c.py']
>>> import shutil
>>> shutil.rmtree(now)
>>> os.getcwd()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 2] No such file or directory
```

请读者注意的是，对于 `os.makedirs()` 还有这样的特点：

```
>>> os.getcwd()
'/home/qw/Documents/VBS/StarterLearningPython/2code'
>>> d0 = os.getcwd()
>>> d1 = d0 + "/ndir1/ndir2/ndir3" #这是想建立的目录，但是中间的 ndir1,ndir2 也都不存在。
>>> d1
'/home/qw/Documents/VBS/StarterLearningPython/2code/ndir1/ndir2/ndir3'
```

```
>>> os.makedirs(d1)
>>> os.chdir(d1)
>>> os.getcwd()
'/home/qw/Documents/VBS/StarterLearningPython/2code/ndir1/ndir2/ndir3'
```

中间不存在的目录也被建立起来，直到做右边的目录为止。与 `os.makedirs()` 类似的还有 `os.mkdir()`，不过，`os.mkdir()` 没有上面这个功能，它只能一层一层地建目录。

`os.removedirs()` 和 `os.rmdir()` 也类似，区别也类似上面。

文件和目录属性

不管是在什么操作系统，都能看到文件或者目录的有关属性，那么，在 `os` 模块中，也有这样的一个方法：`os.stat()`

```
>>> p = os.getcwd() #当前目录
>>> p
'/home/qw/Documents/VBS/StarterLearningPython'

# 这个目录的有关信息
>>> os.stat(p)
posix.stat_result(st_mode=16895, st_ino=4L, st_dev=26L, st_nlink=1, st_uid=0, st_gid=0, st_size=12288L, st_atime=143

# 指定一个文件
>>> pf = p + "/README.md"
# 此文件的信息
>>> os.stat(pf)
posix.stat_result(st_mode=33279, st_ino=67L, st_dev=26L, st_nlink=1, st_uid=0, st_gid=0, st_size=50L, st_atime=14295
```

从结果中看，可能看不出什么来，先不用着急。这样的结果是对 computer 姑娘友好的，对读者可能不友好。如果用下面的方法，就友好多了：

```
>>> fi = os.stat(pf)
>>> mt = fi[8]
```

`fi[8]` 就是 `st_mtime` 的值，它代表最后 `modified`（修改）文件的时间。看结果：

```
>>> mt
1429580969
```

还是不友好。下面就用 `time` 模块来友好一下：

```
>>> import time
>>> time.ctime(mt)
'Tue Apr 21 09:49:29 2015'
```

现在就对读者友好了。

用 `os.stat()` 能够查看文件或者目录的属性。如果要修改呢？比如在部署网站的时候，常常要修改目录或者文件的权限等。这种操作在 Python 的 `os` 模块能做到吗？

要求越来越多了。在一般情况下，不在 Python 里做这个呀。当然，世界是复杂的。肯定有人会用到的，所以 `os` 模块提供了 `os.chmod()`

操作命令

读者如果使用某种 linux 系统，或者曾经用过 dos（恐怕很少），或者再 windows 里面用过 command，对敲命令都不陌生。通过命令来做事情的确是很酷的。比如，我是在 ubuntu 中，要查看文件和目录，只需要 `ls` 就足够了。我并不是否认图形界面，而是在某些情况下，还是离不开命令的，比如用程序来完成查看文件和目录的操作。所以，`os` 模块中提供了这样的方法，许可程序员在 Python 程序中使用操作系统的命令。（以下是在 ubuntu 系统，如果读者是 windows，可以将命令换成 DOS 命令。）

```
>>> p
'/home/qw/Documents/VBS/StarterLearningPython'
>>> command = "ls " + p
>>> command
'ls /home/qw/Documents/VBS/StarterLearningPython'
```

为了输入方便，我采用了前面例子中已经有的那个目录，并且，用拼接字符串的方式，将要输入的命令（查看某文件夹下的内容）组装成一个字符串，赋值给变量 `command`，然后：

```
>>> os.system(command)
01.md 101.md 105.md 109.md 113.md 117.md 121.md 125.md 129.md 201.md 205.md 209.md 213.md 217.md 22
02.md 102.md 106.md 110.md 114.md 118.md 122.md 126.md 130.md 202.md 206.md 210.md 214.md 218.md 22
03.md 103.md 107.md 111.md 115.md 119.md 123.md 127.md 1code 203.md 207.md 211.md 215.md 219.md 22
0images 104.md 108.md 112.md 116.md 120.md 124.md 128.md images 204.md 208.md 212.md 216.md 220.md ir
0
```

这样就列出来了该目录下的所有内容。

需要注意的是，`os.system()` 是在当前进程中执行命令，直到它执行结束。如果需要一个新的进程，可以使用 `os.exec` 或者 `os.execvp`。对此有兴趣详细了解的读者，可以查看帮助文档了解。另外，`os.system()` 是通过 shell 执行命令，执行结束后将控制权返回到原来的进程，但是 `os.exec()` 及相关的函数，则在执行后不将控制权返回到原继承，从而使 Python 失去控制。

关于 Python 对进程的管理，此处暂不过多介绍。

`os.system()` 是一个用途不少的函数。曾有一个朋友网上询问，用它来启动浏览器。不过，这个操作的确要非常仔细。为什么呢？演示一下就明白了。

```
>>> os.system("/usr/bin/firefox")
(process:4002): GLib-CRITICAL **: g_slice_set_config: assertion 'sys_page_size == 0' failed
(firefox:4002): GLib-GObject-WARNING **: Attempt to add property GnomeProgram::sm-connect after class was initialized
.....
```

我是在 ubuntu 上操作的，浏览器的地址是 `/usr/bin/firefox`，可是，那个朋友是 windows，他就要非常小心了，因为在 windows 里面，表示路径的斜杠是跟上面显示的是反着的，可是在 Python 中 `\` 这种斜杠代表转义。解决这个问题可以参看[《字符串\(1\)》](#)的转义符以及[《字符串\(2\)》](#)的原始字符串讲述。比较简单的一个方法用 `r"c:\user\firfox.exe"` 的样式，因为在 `r""` 中的，都是被认为原始字符了。还没完，因为 windows 系统中，一般情况下那个文件不是安装在我演示的那个简单样式的文件夹中，而是 `C:\Program Files`，这中间还有空格，所以还要注意，空格问题。简直有点晕头转向了。读者按照这些提示，看看能不能完成用 `os.system()` 启动 `firefox` 的操作呢？

凡事感觉麻烦的东西，必然有另外简单的来替代。于是又有了一个 `webbrowser` 模块。可以专门用来打开指定网页。

```
>>> import webbrowser
>>> webbrowser.open("http://www.itdiffer.com")
True
```

不管是什操作系统，只要如上操作就能打开网页了。

真是神奇的标准库，有如此多的工具，能不加速开发进程吗？能不降低开发成本吗？“人生苦短，我用 Python”！

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com, 不胜感激。

标准库 (4)

heapq

堆 (heap)，是一种数据结构。用维基百科中的说明：

堆（英语：heap），是计算机科学中一类特殊的数据结构的统称。堆通常是一个可以被看做一棵树的数组对象。

对于这个新的概念，读者不要感觉心慌意乱或者恐惧，因为它本质上不是新东西，而是在我们已经熟知的知识基础上的扩展。

堆的实现是通过构造二叉堆，也就是一种二叉树。

基础知识

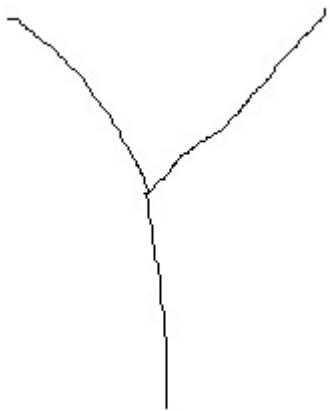
这是一颗在苏州很常见的香樟树，马路两边、公园里随处可见。



但是，在编程中，我们常说的树通常不是上图那样的，而是这样的：

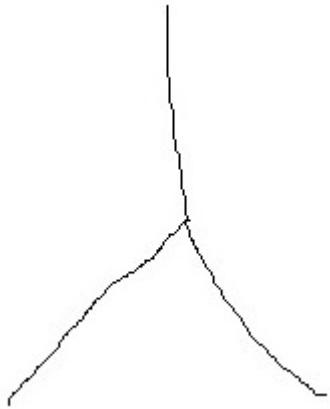


跟真实现实生活中看到的树反过来，也就是“根”在上面。为什么这样呢？我想主要是画着更方便吧。但是，我觉这棵树，是完全写实的作品。我本人做为一名隐姓埋名多年的抽象派画家，不喜欢这样的树，我画出来的是这样的：

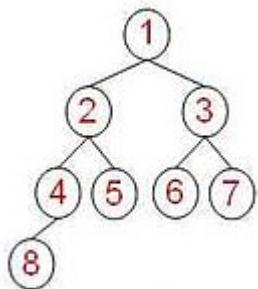


这棵树有两根枝权，可不要小看这两根枝权哦，《道德经》上不是说“一生二，二生三，三生万物”。一就是下面那个干，二就是两个枝权，每个枝权还可以看做下一个一，然后再有两个枝权，如此不断重复（这简直就是递归呀），就成为了一棵大树。

我的确很佩服我自己的后现代抽象派的作品。但是，我更喜欢把这棵树画成这样：



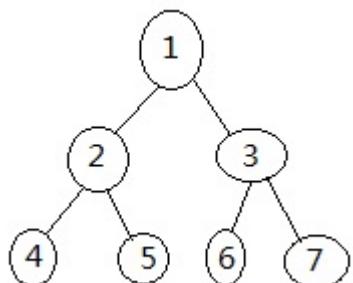
并且给它一个正规的名字：二叉树



这个也是二叉树，完全脱胎于我所画的后现代抽象主义作品。但是略有不同，这幅图在各个枝权上显示的是数字。这种类型的“树”就编程语言中所说的二叉树，维基百科曰：

在计算机科学中，二叉樹（英语：Binary tree）是每個節點最多有兩個子樹的樹結構。通常子樹被稱作「左子樹」（left subtree）和「右子樹」（right subtree）。二叉樹常被用於實現二叉查找樹和二叉堆。

在上图的二叉树中，最顶端的那个数字就相当于树根，也就称作“根”。每个数字所在位置成为一个节点，每个节点向下分散出两个“子节点”。就上图的二叉树，在最后一层，并不是所有节点都有两个子节点，这类二叉树又称为完全二叉树（Complete Binary Tree），也有的二叉树，所有的节点都有两个子节点，这类二叉树称作满二叉树（Full Binarry Tree），如下图：



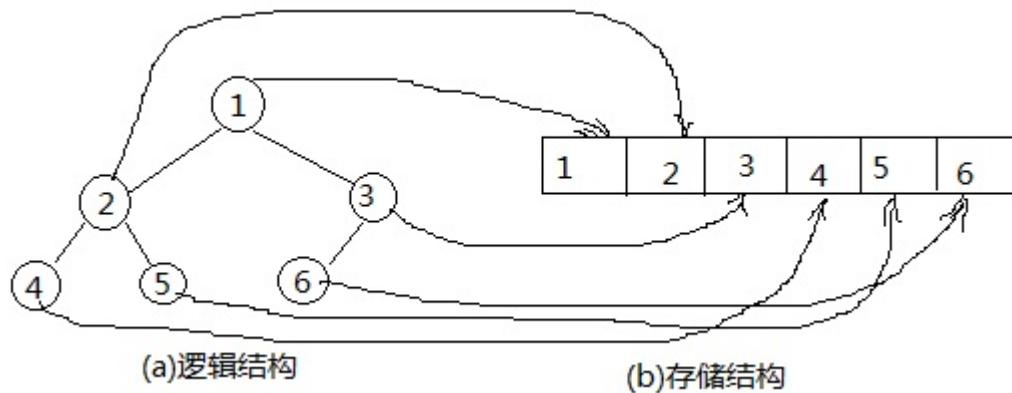
下面讨论的对象是实现二叉堆就是通过二叉树实现的。其应该具有如下特点：

- 节点的值大于等于（或者小于等于）任何子节点的值。
- 节点左子树和右子树是一个二叉堆。如果父节点的值总大于等于任何一个子节点的值，其为最大堆；若父节点值总小于等于子节点值，为最小堆。上面图示中的完全二叉树，就表示一个最小堆。

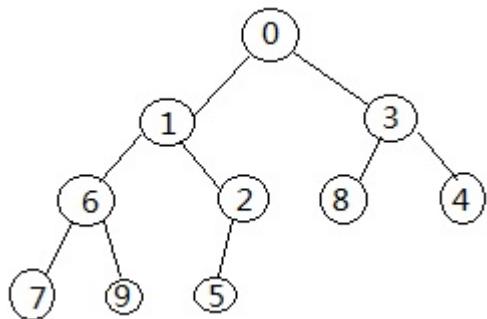
堆的类型还有别的，如斐波那契堆等，但很少用。所以，通常就将二叉堆也说成堆。下面所说的堆，就是二叉堆。而二叉堆又是用二叉树实现的。

堆的存储

堆用列表（有的语言中成为数组）来表示。如下图所示：



从图示中可以看出，将逻辑结构中的树的节点数字依次填入到存储结构中。看这个图，似乎是列表中按照顺序进行排列似的。但是，这仅仅由于那个树的特点造成的，如果是下面的树：



如果将上面的逻辑结构转换为存储结构，读者就能看出来了，不再是按照顺序排列的了。

关于堆的各种，如插入、删除、排序等，本节不会专门讲授编码方法，读者可以参与有关资料。但是，下面要介绍如何用 Python 中的模块 `heapq` 来实现这些操作。

heapq 模块

heapq 中的 heap 是堆，q 就是 queue（队列）的缩写。此模块包括：

```
>>> import heapq
>>> heapq.__all__
['heappush', 'heappop', 'heapify', 'heapreplace', 'merge', 'nlargest', 'nsmallest', 'heappushpop']
```

依次查看这些函数的使用方法。

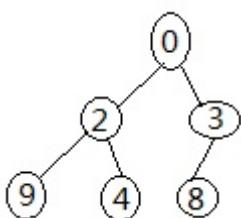
`heappush(heap, x)`: 将 x 压入对 heap (这是一个列表)

```
Help on built-in function heappush in module _heapq:

heappush(...)
    heappush(heap, item) -> None. Push item onto heap, maintaining the heap invariant.
```

```
>>> import heapq
>>> heap = []
>>> heapq.heappush(heap, 3)
>>> heapq.heappush(heap, 9)
>>> heapq.heappush(heap, 2)
>>> heapq.heappush(heap, 4)
>>> heapq.heappush(heap, 0)
>>> heapq.heappush(heap, 8)
>>> heap
[0, 2, 3, 9, 4, 8]
```

请读者注意我上面的操作，在向堆增加数值的时候，我并没有严格按照什么顺序，是随意的。但是，当我查看堆的数据时，显示给我的是一个有一定顺序的数据结构。这种顺序不是按照从小到大，而是按照前面所说的完全二叉树的方式排列。显示的是存储结构，可以把它还原为逻辑结构，看看是不是一颗二叉树。



由此可知，利用 `heappush()` 函数将数据放到堆里面之后，会自动按照二叉树的结构进行存储。

`heappop(heap)`: 删除最小元素

承接上面的操作：

```
>>> heapq.heappop(heap)
0
>>> heap
[2, 4, 3, 9, 8]
```

用 `heappop()` 函数，从 `heap` 堆中删除了一个最小元素，并且返回该值。但是，这时候的 `heap` 显示顺序，并非简单地将 0 去除，而是按照完全二叉树的规范重新进行排列。

`heapify()`: 将列表转换为堆

如果已经建立了一个列表，利用 `heapify()` 可以将列表直接转化为堆。

```
>>> hl = [2, 4, 6, 8, 9, 0, 1, 5, 3]
>>> heapq.heapify(hl)
>>> hl
[0, 3, 1, 4, 9, 6, 2, 5, 8]
```

经过这样的操作，列表 `hl` 就变成了堆（注意观察堆的顺序，和列表不同），可以对 `hl`（堆）使用 `heappop()` 或者 `heappush()` 等函数了。否则，不可。

```
>>> heapq.heappop(hl)
0
>>> heapq.heappop(hl)
1
>>> hl
[2, 3, 5, 4, 9, 6, 8]
>>> heapq.heappush(hl, 9)
>>> hl
[2, 3, 5, 4, 9, 6, 8, 9]
```

不要认为堆里面只能放数字，之所以用数字，是因为对它的逻辑结构比较好理解。

```
>>> heapq.heappush(hl, "q")
>>> hl
[2, 3, 5, 4, 9, 6, 8, 9, 'q']
>>> heapq.heappush(hl, "w")
>>> hl
[2, 3, 5, 4, 9, 6, 8, 9, 'q', 'w']
```

`heapreplace()`

是 `heappop()` 和 `heappush()` 的联合，也就是删除一个，同时加入一个。例如：

```
>>> heap
[2, 4, 3, 9, 8]
>>> heapq.heapreplace(heap, 3.14)
2
>>> heap
[3, 4, 3.14, 9, 8]
```

先简单罗列关于堆的几个常用函数。那么堆在编程实践中的用途在哪方面呢？主要在排序上。一提到排序，读者肯定想到的是 `sorted()` 或者列表中的 `sort()`，不错，这两个都是常用的函数，而且在一般情况下已经足够使用了。如果再使用堆排序，相对上述方法应该有优势。

堆排序的优势不仅更快，更重要的是有效地使用内存，当然，另外一个也不同忽视，就是简单易用。比如前面操作的，删除数列中最小的值，就是在排序基础上进行的操作。

deque 模块

有这样一个问题：一个列表，比如是 `[1,2,3]`，我打算在最右边增加一个数字。

这也太简单了，不就是用 `append()` 这个内建函数，追加一个吗？

这是简单，我要得寸进尺，能不能在最左边增加一个数字呢？

这个嘛，应该有办法。不过得想想了。读者在向下阅读的时候，能不能想出一个方法来？

```
>>> lst = [1, 2, 3]
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
>>> nl = [7]
>>> nl.extend(lst)
>>> nl
[7, 1, 2, 3, 4]
```

你或许还有别的方法。但是，Python 为我们提供了一个更简单的模块，来解决这个问题。

```
>>> from collections import deque
```

这次用这种引用方法，因为 `collections` 模块中东西很多，我们只用到 `deque`。

```
>>> lst
[1, 2, 3, 4]
```

还是这个列表。试试分别从右边和左边增加数

```
>>> qlst = deque(lst)
```

这是必须的，将列表转化为 deque。deque 在汉语中有一个名字，叫做“双端队列”（double-ended queue）。

```
>>> qlst.append(5)      #从右边增加
>>> qlst
deque([1, 2, 3, 4, 5])
>>> qlst.appendleft(7)  #从左边增加
>>> qlst
deque([7, 1, 2, 3, 4, 5])
```

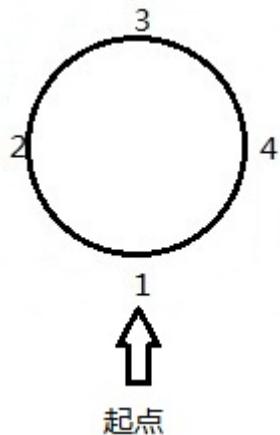
这样操作多么容易呀。继续看删除：

```
>>> qlst.pop()
5
>>> qlst
deque([7, 1, 2, 3, 4])
>>> qlst.popleft()
7
>>> qlst
deque([1, 2, 3, 4])
```

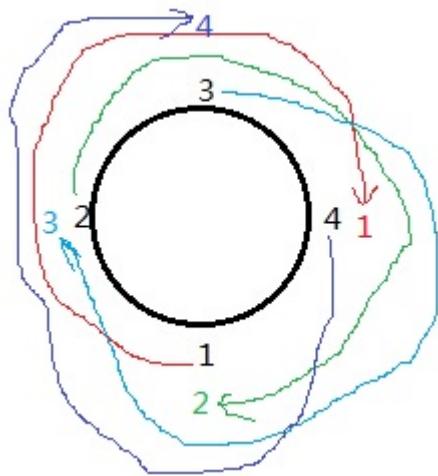
删除也分左右。下面这个，请读者仔细观察，更有点意思。

```
>>> qlst.rotate(3)
>>> qlst
deque([2, 3, 4, 1])
```

rotate() 的功能是将[1, 2, 3, 4]的首位连起来，你就想象一个圆环，在上面有 1,2,3,4 几个数字。如果一开始正对着你的是 1，依顺时针方向排列，就是从 1 开始的数列，如下图所示：



经过 rotate()，这个环就发生旋转了，如果是 rotate(3)，表示每个数字按照顺时针方向前进三个位置，于是变成了：



请原谅我的后现代注意超级抽象派作图方式。从图中可以看出，数列变成了[2, 3, 4, 1]。rotate() 作用就好像在拨转这个圆环。

```
>>> qlst
deque([3, 4, 1, 2])
>>> qlst.rotate(-1)
>>> qlst
deque([4, 1, 2, 3])
```

如果参数是复数，那么就逆时针转。

在 deque 中，还有 extend 和 extendleft 方法。读者可自己调试。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qjwsir@126.com,不胜感激。

标准库 (5)

“一寸光阴一寸金，寸金难买寸光阴”，时间是宝贵的。

在日常生活中，“时间”这个属于是比较笼统和含糊的。在物理学中，“时间”是一个非常明确的概念。在 Python 中，“时间”可以通过相关模块实现。

calendar

```
>>> import calendar
>>> cal = calendar.month(2015, 1)
>>> print cal
January 2015
Mo Tu We Th Fr Sa Su
 1 2 3 4
 5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

轻而易举得到了 2015 年 1 月的日历，并且排列的还那么整齐。这就是 calendar 模块。读者可以用 dir() 去查看这个模块下的所有内容。为了让读者阅读方便，将常用的整理如下：

`calendar(year,w=2,l=1,c=6)`

返回 year 年年历，3 个月一行，间隔距离为 c。每日宽度间隔为 w 字符。每行长度为 $21 * W + 18 + 2 * C$ 。l 是每星期行数。

```
>>> year = calendar.calendar(2015)
>>> print year
2015

January          February         March
Mo Tu We Th Fr Sa Su   Mo Tu We Th Fr Sa Su   Mo Tu We Th Fr Sa Su
 1 2 3 4           1           1
 5 6 7 8 9 10 11   2 3 4 5 6 7 8   2 3 4 5 6 7 8
12 13 14 15 16 17 18   9 10 11 12 13 14 15   9 10 11 12 13 14 15
19 20 21 22 23 24 25   16 17 18 19 20 21 22   16 17 18 19 20 21 22
26 27 28 29 30 31     23 24 25 26 27 28     23 24 25 26 27 28 29
                           30 31
```

April	May	June
Mo Tu We Th Fr Sa Su	Mo Tu We Th Fr Sa Su	Mo Tu We Th Fr Sa Su
1 2 3 4 5	1 2 3 4 5 6 7	
6 7 8 9 10 11 12	4 5 6 7 8 9 10	8 9 10 11 12 13 14
13 14 15 16 17 18 19	11 12 13 14 15 16 17	15 16 17 18 19 20 21
20 21 22 23 24 25 26	18 19 20 21 22 23 24	22 23 24 25 26 27 28
27 28 29 30	25 26 27 28 29 30 31	29 30
July	August	September
Mo Tu We Th Fr Sa Su	Mo Tu We Th Fr Sa Su	Mo Tu We Th Fr Sa Su
1 2 3 4 5	1 2	1 2 3 4 5 6
6 7 8 9 10 11 12	3 4 5 6 7 8 9	7 8 9 10 11 12 13
13 14 15 16 17 18 19	10 11 12 13 14 15 16	14 15 16 17 18 19 20
20 21 22 23 24 25 26	17 18 19 20 21 22 23	21 22 23 24 25 26 27
27 28 29 30 31	24 25 26 27 28 29 30	28 29 30
31		
October	November	December
Mo Tu We Th Fr Sa Su	Mo Tu We Th Fr Sa Su	Mo Tu We Th Fr Sa Su
1 2 3 4	1	1 2 3 4 5 6
5 6 7 8 9 10 11	2 3 4 5 6 7 8	7 8 9 10 11 12 13
12 13 14 15 16 17 18	9 10 11 12 13 14 15	14 15 16 17 18 19 20
19 20 21 22 23 24 25	16 17 18 19 20 21 22	21 22 23 24 25 26 27
26 27 28 29 30 31	23 24 25 26 27 28 29	28 29 30 31
30		

isleap(year)

判断是否为闰年，是则返回 true，否则 false。

```
>>> calendar.isleap(2000)
True
>>> calendar.isleap(2015)
False
```

怎么判断一年是闰年，常常见诸于一些编程语言的练习题，现在用一个方法搞定。

leapdays(y1,y2)

返回在 Y1, Y2 两年之间的闰年总数，包括 y1，但不包括 y2，这有点如同序列的切片一样。

```
>>> calendar.leapdays(2000,2004)
1
>>> calendar.leapdays(2000,2003)
1
```

month(year,month,w=2,l=1)

返回 year 年 month 月日历，两行标题，一周一行。每日宽度间隔为 w 字符。每行的长度为 $7 * w + 6$ 。l 是每星期的行数。

```
>>> print calendar.month(2015, 5)
      May 2015
Mo Tu We Th Fr Sa Su
  1 2 3
 4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

monthcalendar(year,month)

返回一个列表，列表内的元素还是列表，这叫做嵌套列表。每个子列表代表一个星期，都是从星期一到星期日，如果没有本月的日期，则为 0。

```
>>> calendar.monthcalendar(2015, 5)
[[0, 0, 0, 0, 1, 2, 3], [4, 5, 6, 7, 8, 9, 10], [11, 12, 13, 14, 15, 16, 17], [18, 19, 20, 21, 22, 23, 24], [25, 26, 27, 28, 29, 30, 31]]
```

读者可以将这个结果和 `calendar.month(2015, 5)` 去对照理解。

montrange(year,month)

返回一个元组，里面有两个整数。第一个整数代表着该月的第一天从星期几是（从 0 开始，依次为星期一、星期二，直到 6 代表星期日）。第二个整数是该月一共多少天。

```
>>> calendar.montrange(2015, 5)
(4, 31)
```

从返回值可知，2015 年 5 月 1 日是星期五，这个月一共 31 天。这个结果，也可以从日历中看到。

weekday(year,month,day)

输入年月日，知道该日是星期几（注意，返回值依然按照从 0 到 6 依次对应星期一到星期六）。

```
>>> calendar.weekday(2015, 5, 4)  #星期一
0
>>> calendar.weekday(2015, 6, 4)  #星期四
3
```

time

time()

time 模块是常用的。

```
>>> import time
>>> time.time()
1430745298.391026
```

time.time() 获得的是当前时间（严格说是时间戳），只不过这个时间对人不友好，它是以 1970 年 1 月 1 日 0 时 0 分 0 秒为计时起点，到当前的时间长度（不考虑闰秒）

UNIX 时间，或称 POSIX 时间是 UNIX 或类 UNIX 系统使用的时间表示方式：从协调世界时 1970 年 1 月 1 日 0 时 0 分 0 秒起至现在的总秒数，不考虑秒

现时大部分使用 UNIX 的系统都是 32 位元的，即它们会以 32 位二进制数字表示时间。但是它们最多只能表示至协调世界时间 2038 年 1 月 19 日 3 时 14 分 07 秒（二进制：01111111 11111111 11111111 11111111，0x7FFF:FFFF），在下一秒二进制数字会是 10000000 00000000 00000000 00000000，（0x8000:0000），这是负数，因此各系统会把时间误解作 1901 年 12 月 13 日 20 时 45 分 52 秒（亦有说回鬼到 1970 年）。这时可能会令软件发生问题，导致系统瘫痪。

目前的解决方案是把系统由 32 位元转为 64 位元系统。在 64 位系统下，此时间最多可以表示到 292,277,026,596 年 12 月 4 日 15 时 30 分 08 秒。

有没有对人友好一点的时间显示呢？

localtime()

```
>>> time.localtime()
time.struct_time(tm_year=2015, tm_mon=5, tm_mday=4, tm_hour=21, tm_min=33, tm_sec=39, tm_wday=0, tm_yday=123)
```

这个就友好多了。得到的结果可以称之为时间元组（也有括号），其各项的含义是：

索引	属性	含义
0	tm_year	年
1	tm_mon	月
2	tm_mday	日
3	tm_hour	时
4	tm_min	分
5	tm_sec	秒

索引	属性	含义
6	tm_wday	一周中的第几天
7	tm_yday	一年中的第几天
8	tm_isdst	夏令时

```
>>> t = time.localtime()
>>> t[1]
5
```

通过索引，能够得到相应的属性，上面的例子中就得到了当前时间的月份。

其实，`time.localtime()` 不是没有参数，它在默认情况下，以 `time.time()` 的时间戳为参数。言外之意就是说可以自己输入一个时间戳，返回那个时间戳所对应的时间（按照公元和时分秒计时）。例如：

```
>>> time.localtime(100000)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=2, tm_hour=11, tm_min=46, tm_sec=40, tm_wday=4, tm_yday=2,
```

`gmtime()`

`localtime()` 得到的是本地时间，如果要国际化，就最好使用格林威治时间。可以这样：

```
>>> import time
>>> time.gmtime()
time.struct_time(tm_year=2015, tm_mon=5, tm_mday=4, tm_hour=23, tm_min=46, tm_sec=34, tm_wday=0, tm_yday=12,
```

格林威治标准时间（中国大陆翻译：格林尼治平均时间或格林尼治标准时间，台、港、澳翻译：格林威治标准时间；英语：Greenwich Mean Time，GMT）是指位于英国伦敦郊区的皇家格林威治天文台的标准时间，因为本初子午线被定义在通过那裡的经线。

还有更友好的：

`asctime()`

```
>>> time.asctime()
'Mon May 4 21:46:13 2015'
```

`time.asctime()` 的参数为空时，默认是以 `time.localtime()` 的值为参数，所以得到的是当前日期时间和星期。当然，也可以自己设置参数：

```
>>> h = time.localtime(1000000)
>>> h
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=12, tm_hour=21, tm_min=46, tm_sec=40, tm_wday=0, tm_yday=1,
```

```
>>> time.asctime(h)
'Mon Jan 12 21:46:40 1970'
```

注意，`time.asctime()` 的参数必须是时间元组，类似上面那种。不是时间戳，通过 `time.time()` 得到的时间戳，也可以转化为上面形式：

ctime()

```
>>> time.ctime()
'Mon May 4 21:52:22 2015'
```

在没有参数的时候，事实上是以 `time.time()` 的时间戳为参数。也可以自定义一个时间戳。

```
>>> time.ctime(1000000)
'Mon Jan 12 21:46:40 1970'
```

跟前面得到的结果是一样的。只不过是用了时间戳作为参数。

在前述函数中，通过 `localtime()`、`gmtime()` 得到的是时间元组，通过 `time()` 得到的是时间戳。有的函数如 `asctime()` 是以时间元组为参数，有的如 `ctime()` 是以时间戳为参数。这样做的目的是为了满足编程中多样化的需要。

mktime()

`mktime()` 也是以时间元组为参数，但是它返回的不是可读性更好的那种样式，而是：

```
>>> lt = time.localtime()
>>> lt
time.struct_time(tm_year=2015, tm_mon=5, tm_mday=5, tm_hour=7, tm_min=55, tm_sec=29, tm_wday=1, tm_yday=125
>>> time.mktime(lt)
1430783729.0
```

返回了时间戳。就类似于 `localtime()` 的逆过程（`localtime()` 是以时间戳为参数）。

以上基本能够满足编程需要了吗？好像还缺点什么，因为在编程中，用的比较多的是“字符串”，似乎还没有将时间转化为字符串的函数。这个应该有。

strftime()

函数格式稍微复杂一些。

Help on built-in function strftime in module time:

`strftime(...)` `strftime(format[, tuple]) -> string`

Convert a time tuple to a string according to a format specification. See the library reference manual for formatting codes. When the time tuple is not present, current time as returned by `localtime()` is used.

将时间元组按照指定格式要求转化为字符串。如果不指定时间元组，就默认为 `localtime()` 值。我说复杂，是在于其 `format`，需要用到下面的东西。

格式	含义	取值范围（格式）
<code>%y</code>	去掉世纪的年份	00–99，如"15"
<code>%Y</code>	完整的年份	如"2015"
<code>%j</code>	指定日期是一年中的第几天	001–366
<code>%m</code>	返回月份	01–12
<code>%b</code>	本地简化月份的名称	简写英文月份
<code>%B</code>	本地完整月份的名称	完整英文月份
<code>%d</code>	该月的第几日	如 5 月 1 日返回"01"
<code>%H</code>	该日的第几时（24 小时制）	00–23
<code>%I</code>	该日的第几时（12 小时制）	01–12
<code>%M</code>	分钟	00–59
<code>%S</code>	秒	00–59
<code>%U</code>	在该年中的第多少星期（以周日为一周起点）	00–53
<code>%W</code>	同上，只不过是以周一为起点	00–53
<code>%w</code>	一星期中的第几天	0–6
<code>%Z</code>	时区	在中国大陆测试，返回 CST，即 China Standard Time
<code>%x</code>	日期	日/月/年
<code>%X</code>	时间	时:分:秒
<code>%c</code>	详细日期时间	日/月/年 时:分:秒
<code>%%</code>	'%' 字符	'%' 字符
<code>%p</code>	上下午	AM or PM

简要列举如下：

```
>>> time.strftime("%y,%m,%d")
'15,05,05'
>>> time.strftime("%y/%m/%d")
'15/05/05'
```

分隔符可以自由指定。既然已经变成字符串了，就可以“随心所欲不逾矩”了。

strptime()

Help on built-in function strptime in module time:

`strptime(...)` `strptime(string, format) -> struct_time`

Parse a string to a time tuple according to a format specification. See the library reference manual for formatting codes (same as strftime()).

`strptime()` 的作用是将字符串转化为时间元组。请注意的是，其参数要指定两个，一个是时间字符串，另外一个是时间字符串所对应的格式，格式符号用上表中的。例如：

```
>>> today = time.strftime("%y/%m/%d")
>>> today
'15/05/05'
>>> time.strptime(today, "%y/%m/%d")
time.struct_time(tm_year=2015, tm_mon=5, tm_mday=5, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=1, tm_yday=125, t
```

datetime

虽然 `time` 模块已经能够把有关时间方面的东西搞定了，但是，在某些调用的时候，还感觉不是很直接，于是又出来了一个 `datetime` 模块，供程序猿和程序媛们选择使用。

`datetime` 模块中有几个类：

- `datetime.date`: 日期类，常用的属性有 `year/month/day`
- `datetime.time`: 时间类，常用的有 `hour/minute/second/microsecond`
- `datetime.datetime`: 日期时间类
- `datetime.timedelta`: 时间间隔，即两个时间点之间的时间长度
- `datetime.tzinfo`: 时区类

date 类

通过实例了解常用的属性：

```
>>> import datetime
>>> today = datetime.date.today()
>>> today
datetime.date(2015, 5, 5)
```

这里其实生成了一个日期对象，然后操作这个对象的各种属性。用 `print` 语句，可以是视觉更佳：

```
>>> print today
2015-05-05
>>> print today.ctime()
Tue May  5 00:00:00 2015
>>> print today.timetuple()
time.struct_time(tm_year=2015, tm_mon=5, tm_mday=5, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=1, tm_yday=125, t
```

```
>>> print today.toordinal()
735723
```

特别注意，如果你妄图用 `datetime.date.year()`，是会报错的，因为 `year` 不是一个方法，必须这样行：

```
>>> print today.year
2015
>>> print today.month
5
>>> print today.day
5
```

进一步看看时间戳与格式化时间格式的转换

```
>>> to = today.toordinal()
>>> to
735723
>>> print datetime.date.fromordinal(to)
2015-05-05

>>> import time
>>> t = time.time()
>>> t
1430787994.80093
>>> print datetime.date.fromtimestamp(t)
2015-05-05
```

还可以更灵活一些，修改日期。

```
>>> d1 = datetime.date(2015,5,1)
>>> print d1
2015-05-01
>>> d2 = d1.replace(year=2005, day=5)
>>> print d2
2005-05-05
```

time 类

也要生成 `time` 对象

```
>>> t = datetime.time(1,2,3)
>>> print t
01:02:03
```

它的常用属性：

```
>>> print t.hour  
1  
>>> print t.minute  
2  
>>> print t.second  
3  
>>> t.microsecond  
0  
>>> print t.tzinfo  
None
```

timedelta 类

主要用来做时间的运算。比如：

```
>>> now = datetime.datetime.now()  
>>> print now  
2015-05-05 09:22:43.142520
```

没有讲述 datetime 类，因为在有了 date 和 time 类知识之后，这个类比较简单，我最喜欢这个 now 方法了。

对 now 增加 5 个小时

```
>>> b = now + datetime.timedelta(hours=5)  
>>> print b  
2015-05-05 14:22:43.142520
```

增加两周

```
>>> c = now + datetime.timedelta(weeks=2)  
>>> print c  
2015-05-19 09:22:43.142520
```

计算时间差：

```
>>> d = c - b  
>>> print d  
13 days, 19:00:00
```

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qjwsir@126.com,不胜感激。

标准库 (6)

urllib

urllib 模块用于读取来自网上（服务器上）的数据，比如不少人用 Python 做爬虫程序，就可以使用这个模块。先看一个简单例子：

```
>>> import urllib
>>> itdiffer = urllib.urlopen("http://www.itdiffer.com")
```

这样就已经把我的网站[www.itdiffer.com]的内容拿过来了，得到了一个类似文件的对象。接下来的操作跟操作一个文件一样（如果忘记了文件怎么操作，可以参考：[《文件\(1\)》](#)）

```
>>> print itdiffer.read()
<!DOCTYPE HTML>
<html>
<head>
<title>I am Qiwsir</title>
....//因为内容太多，下面就省略了
```

就这么简单，完成了对一个网页的抓取。当然，如果你真的要做爬虫程序，还不是仅仅如此。这里不介绍爬虫程序如何编写，仅说明 urllib 模块的常用属性和方法。

```
>>> dir(urllib)
['ContentTooShortError', 'FancyURLopener', 'MAXFTPCACHE', 'URLopener', '__all__', '__builtins__', '__doc__', '__file__']
```

选几个常用的介绍，其它的如果读者用到，可以通过查看文档了解。

urlopen()

urlopen() 主要用于打开 url 文件，然后就获得指定 url 的数据，接下来就如同在本地操作文件那样来操作。

Help on function urlopen in module urllib:

urlopen(url, data=None, proxies=None) Create a file-like object for the specified URL to read from.

得到的对象被叫做类文件。从名字中也可以理解后面的操作了。先对参数说明一下：

- url：远程数据的路径，常常是网址
- data：如果使用 post 方式，这里就是所提交的数据
- proxies：设置代理

关于参数的详细说明，还可以参考[Python 的官方文档](#)，这里仅演示最常用的，如前面的例子那样。

当得到了类文件对象之后，就可以对它进行操作。变量 `itdiffer` 引用了得到的类文件对象，通过它查看：

```
>>> dir(itdiffer)
['__doc__', '__init__', '__iter__', '__module__', '__repr__', 'close', 'code', 'fileno', 'fp', 'getcode', 'geturl', 'headers', 'info', 'nex
```

读者从这个结果中也可以看出，这个类文件对象也是可迭代的。常用的方法：

- `read()`,`readline()`,`readlines()`,`fileno()`,`close()`: 都与文件操作一样，这里不再赘述。可以参考前面有关文件章节
- `info()`: 返回头信息
- `getcode()`: 返回 http 状态码
- `geturl()`: 返回 url

简单举例：

```
>>> itdiffer.info()
<httplib.HTTPMessage instance at 0xb6eb3f6c>
>>> itdiffer.getcode()
200
>>> itdiffer.geturl()
'http://www.itdiffer.com'
```

更多情况下，已经建立了类文件对象，通过对文件操作方法，获得想要的数据。

对 url 编码、解码

url 对其中的字符有严格要求，不许可某些特殊字符，这就要对 url 进行编码和解码了。这个在进行 web 开发的时候特别要注意。`urllib` 模块提供这种功能。

- `quote(string[, safe])`: 对字符串进行编码。参数 `safe` 指定了不需要编码的字符
- `urllib.unquote(string)` : 对字符串进行解码
- `quote_plus(string [, safe])` : 与 `urllib.quote` 类似，但这个方法用'+'来替换空格 ''，而 `quote` 用'%20'来代替空格
- `unquote_plus(string)` : 对字符串进行解码；
- `urllib.urlencode(query[, doseq])`: 将 `dict` 或者包含两个元素的元组列表转换成 url 参数。例如{'name': 'laoqi', 'age': 40}将被转换为"name=laoqi&age=40"
- `pathname2url(path)`: 将本地路径转换成 url 路径

- url2pathname(path): 将 url 路径转换成本地路径

看例子就更明白了：

```
>>> du = "http://www.itdiffer.com/name=python book"
>>> urllib.quote(du)
'http%3A//www.itdiffer.com/name%3Dpython%20book'
>>> urllib.quote_plus(du)
'http%3A%2F%2Fwww.itdiffer.com%2Fname%3Dpython+book'
```

注意看空格的变化，一个被编码成 `%20`，另外一个是 `+`

再看解码的，假如在 google 中搜索 `零基础 Python`，结果如下图：

About 442,000 results (0.24 seconds)

《零基础学Python》 by qiwsir - GitHub
<https://github.com/qiwsir/.../BasicPython/index.md> ▾ Translate this page
 Jan 7, 2015 - This is for everyone. 零基础学Python. 第零部分独上高楼，望尽天涯路。唠叨一些关于python的事情。开始本栏目的原因 ...
[重回函数 - 222.md](#) - [223.md](#) - [226.md](#)

(零基础学习Python、Python入门) 书籍、视频、资料 - GitHub
<https://github.com/Yixiaohan/codeparkshare> ▾ Translate this page
 Apr 16, 2015 - Python初学者 (零基础学习Python、Python入门) 书籍、视频、资料、社区推荐. Contribute to codeparkshare development by creating an account ...

我的教程可是在这次搜索中排列第一个哦。

这不是重点，重点是看 url，它就是用 `+` 替代空格了。

```
>>> dup = urllib.quote_plus(du)
>>> urllib.unquote_plus(dup)
'http://www.itdiffer.com/name=Python book'
```

从解码效果来看，比较完美地逆过程。

```
>>> urllib.urlencode({"name":"qiwsir","web":"itdiffer.com"})
'web=itdiffer.com&name=qiwsir'
```

这个在编程中，也会用到，特别是开发网站时候。

urlretrieve()

虽然 urlopen() 能够建立类文件对象，但是，那还不等于将远程文件保存在本地存储器中，urlretrieve() 就是满足这个需要的。先看实例：

```
>>> import urllib
>>> urllib.urlretrieve("http://www.itdiffer.com/images/me.jpg","me.jpg")
('me.jpg', <httplib.HTTPMessage instance at 0xb6ecb6cc>)
>>>
```

me.jpg 是一张存在于服务器上的图片，地址是：http://www.itdiffer.com/images/me.jpg，把它保存到本地存储器中，并且仍旧命名为 me.jpg。注意，如果只写这个名字，表示存在启动 Python 交互模式的那个目录中，否则，可以指定存储具体目录和文件名。

在[urllib官方文档](#)中有一大段相关说明，读者可以去认真阅读。这里仅简要介绍一下相关参数。

```
urllib.urlretrieve(url[, filename[, reporthook[, data]]])
```

- url：文件所在的网址
- filename：可选。将文件保存到本地的文件名，如果不指定，urllib 会生成一个临时文件来保存
- reporthook：可选。是回调函数，当链接服务器和相应数据传输完毕时触发本函数
- data：可选。如果用 post 方式所发出的数据

函数执行完毕，返回的结果是一个元组(filename, headers)，filename 是保存到本地的文件名，headers 是服务器响应头信息。

```
#!/usr/bin/env Python
# coding=utf-8

import urllib

def go(a,b,c):
    per = 100.0 * a * b / c
    if per > 100:
        per = 100
    print "%.2f%%" % per

url = "http://youxi.66wz.com/uploads/1046/1321/11410192.90d133701b06f0cc2826c3e5ac34c620.jpg"
```

```
local = "/home/qw/Pictures/g.jpg"
urllib.urlretrieve(url, local, go)
```

这段程序就是要下载指定的图片，并且保存为本地指定位置的文件，同时要显示下载的进度。上述文件保存之后，执行，显示如下效果：

```
$ Python 22501.py
0.00%
8.13%
16.26%
24.40%
32.53%
40.66%
48.79%
56.93%
65.06%
73.19%
81.32%
89.46%
97.59%
100.00%
```

到相应目录中查看，能看到与网上地址一样的文件。我这里就不对结果截图了，唯恐少部分读者鼻子流血。

urllib2

urllib2 是另外一个模块，它跟 urllib 有相似的地方——都是对 url 相关的操作，也有不同的地方。关于这方面，有一篇文章讲的不错：[Python: difference between urllib and urllib2](#)

我选取一段，供大家参考：

urllib2 can accept a Request object to set the headers for a URL request, urllib accepts only a URL.
That means, you cannot masquerade your User Agent string etc.

urllib provides the urlencode method which is used for the generation of GET query strings, urllib2 doesn't have such a function. This is one of the reasons why urllib is often used along with urllib2.

所以，有时候两个要同时使用，urllib 模块和 urllib2 模块有的方法可以相互替代，有的不能。看下面的属性方法列表就知道了。

```
>>> dir(urllib2)
['AbstractBasicAuthHandler', 'AbstractDigestAuthHandler', 'AbstractHTTPHandler', 'BaseHandler', 'CacheFTPHandler', 'F
```

比较常用的比如 `urlopen()` 跟 `urllib.open()` 是完全类似的。

Request 类

正如前面区别 `urllib` 和 `urllib2` 所讲，利用 `urllib2` 模块可以建立一个 `Request` 对象。方法就是：

```
>>> req = urllib2.Request("http://www.itdiffer.com")
```

建立了 `Request` 对象之后，它的最直接应用就是可以作为 `urlopen()` 方法的参数

```
>>> response = urlopen(req)
>>> page = response.read()
>>> print page
```

因为与前面的 `urllib.open("http://www.itdiffer.com")` 结果一样，就不浪费篇幅了。

但是，如果 `Request` 对象仅仅局限于此，似乎还没有什么太大的优势。因为刚才的访问仅仅是满足以 `get` 方式请求页面，并建立类文件对象。如果是通过 `post` 向某地址提交数据，也可以建立 `Request` 对象。

```
import urllib
import urllib2

url = 'http://www.itdiffer.com/register.py'

values = {'name' : 'qiwsir',
          'location' : 'China',
          'language' : 'Python' }

data = urllib.urlencode(values)    # 编码
req = urllib2.Request(url, data)   # 发送请求同时传 data 表单
response = urlopen(req)          # 接受反馈的信息
the_page = response.read()        # 读取反馈的内容
```

注意，读者不能照抄上面的程序，然后运行代码。因为那个 `url` 中没有相应的接受客户端 `post` 上去的 `data` 的程序文件。上面的代码只是以一个例子来显示 `Request` 对象的另外一个用途，还有就是在这个例子中是以 `post` 方式提交数据。

在网站中，有的会通过 `User-Agent` 来判断访问者是浏览器还是别的程序，如果通过别的程序访问，它有可能拒绝。这时候，我们编写程序去访问，就要设置 `headers` 了。设置方法是：

```
user_agent = 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'
headers = { 'User-Agent' : user_agent }
```

然后重新建立 `Request` 对象：

```
req = urllib2.Request(url, data, headers)
```

再用 `urlopen()` 方法访问：

```
response = urllib2.urlopen(req)
```

除了上面演示之外，`urllib2` 模块的东西还很多，比如还可以：

- 设置 HTTP Proxy
- 设置 Timeout 值
- 自动 redirect
- 处理 cookie

等等。这些内容不再一一介绍，当需要用到的时候可以查看文档或者 google。

总目录 (页 0)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

xml

xml 在软件领域用途非常广泛，有名人曰：

“当 XML（扩展标记语言）于 1998 年 2 月被引入软件工业界时，它给整个行业带来了一场风暴。有史以来第一次，这个世界拥有了一种用来结构化文档和数据的通用且适应性强的格式，它不仅仅可以用于 WEB，而且可以被用于任何地方。”

--- 《Designing With Web Standards Second Edition》，Jeffrey Zeldman

对于 xml 如果要做一个定义式的说明，就不得不引用 w3school 里面简洁而明快的说明：

- XML 指可扩展标记语言（EXtensible Markup Language）
- XML 是一种标记语言，很类似 HTML
- XML 的设计宗旨是传输数据，而非显示数据
- XML 标签没有被预定义。您需要自行定义标签。
- XML 被设计为具有自我描述性。
- XML 是 W3C 的推荐标准

如果读者要详细了解和学习有关 xml，可以阅读[w3school的教程](#)

xml 的重要，关键在于它是用来传输数据，因为传输数据，特别是在 web 编程中，经常要用到的。有了这样一种东西，就让数据传输变得简单了。对于这么重要的，Python 当然有支持。

一般来讲，一个引人关注的东西，总会有很多人从不同侧面去关注。在编程语言中也是如此，所以，对 xml 这个明星式的东西，Python 提供了多种模块来处理。

- `xml.dom.*` 模块：Document Object Model。适合用于处理 DOM API。它能够将 xml 数据在内存中解析成一个树，然后通过对树的操作来操作 xml。但是，这种方式由于将 xml 数据映射到内存中的树，导致比较慢，且消耗更多内存。
- `xml.sax.*` 模块：simple API for XML。由于 SAX 以流式读取 xml 文件，从而速度较快，切少占用内存，但是操作上稍复杂，需要用户实现回调函数。
- `xml.parser.expat`：是一个直接的，低级一点的基于 C 的 expat 的语法分析器。expat 接口基于事件反馈，有点像 SAX 但又不太像，因为它的接口并不是完全规范于 expat 库的。
- `xml.etree.ElementTree`（以下简称 ET）：元素树。它提供了轻量级的 Python 式的 API，相对于 DOM，ET 快了很多，而且有很多令人愉悦的 API 可以使用；相对于 SAX，ET 也有 `ET.iterparse` 提供了“在空中”的处理方式，没有必要加载整个文档到内存，节省内存。ET 的性能的平均值和 SAX 差不多，但是 API 的效率更高一点而且使用起来很方便。

所以，我用 `xml.etree.ElementTree`

`ElementTree` 在标准库中有两种实现。一种是纯 Python 实现：`xml.etree.ElementTree`，另外一种是速度快一点：`xml.etree.cElementTree`。

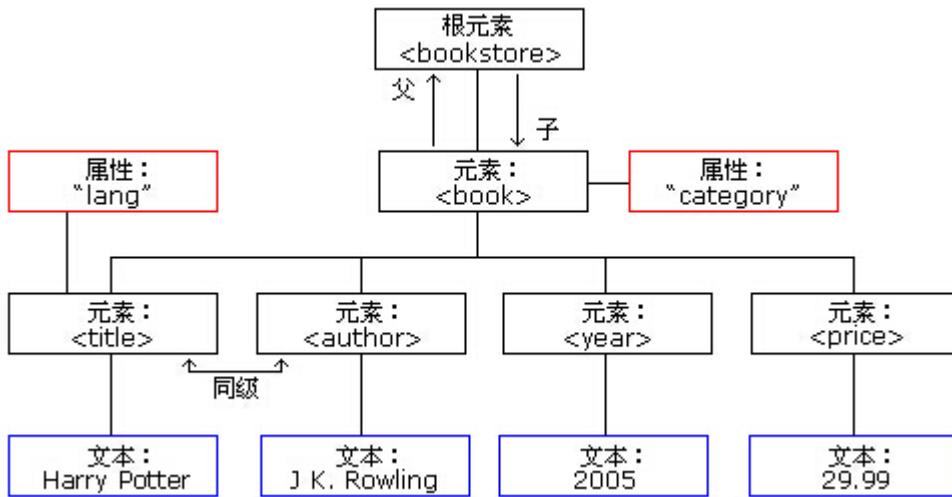
如果读者使用的是 Python2.x，可以像这样引入模块：

```
try:
    import xml.etree.cElementTree as ET
except ImportError:
    import xml.etree.ElementTree as ET
```

如果是 Python3.3 以上，就没有这个必要了，只需要一句话 `import xml.etree.ElementTree as ET` 即可，然后由模块自动来寻找适合的方式。显然 Python3.x 相对 Python2.x 有了很大进步。但是，本教程碍于很多工程项目还没有升级换代，暂且忍受了。

遍历查询

先要搞一个 xml 文档。为了图省事，我就用 w3school 中的一个例子：



这是一个 XML 树，只不过是用图来表示的，还没有用 ET 解析呢。把这棵树写成 XML 文档格式：

```

<bookstore>
    <book category="COOKING">
        <title lang="en">Everyday Italian</title>
        <author>Giada De Laurentiis</author>
        <year>2005</year>
        <price>30.00</price>
    </book>
    <book category="CHILDREN">
        <title lang="en">Harry Potter</title>
        <author>J K. Rowling</author>
        <year>2005</year>
        <price>29.99</price>
    </book>
    <book category="WEB">
        <title lang="en">Learning XML</title>
        <author>Erik T. Ray</author>
        <year>2003</year>
        <price>39.95</price>
    </book>
</bookstore>
    
```

将 XML 保存为名为 22601.xml 的文件，然后对其进行如下操作：

```
>>> import xml.etree.cElementTree as ET
```

为了简化，我用这种方式引入，如果在编程实践中，推荐读者使用 try...except... 方式。

```

>>> tree = ET.ElementTree(file="22601.xml")
>>> tree
<ElementTree object at 0xb724cc2c>
    
```

建立起 xml 解析树。然后可以通过根节点向下开始读取各个元素（element 对象）。

在上述 xml 文档中，根元素是，它没有属性，或者属性为空。

```
>>> root = tree.getroot()      #获得根
>>> root.tag
'bookstore'
>>> root.attrib
{}
```

要想将根下面的元素都读出来，可以：

```
>>> for child in root:
...     print child.tag, child.attrib
...
book {'category': 'COOKING'}
book {'category': 'CHILDREN'}
book {'category': 'WEB'}
```

也可以这样读取指定元素的信息：

```
>>> root[0].tag
'book'
>>> root[0].attrib
{'category': 'COOKING'}
>>> root[0].text      #无内容
'\n'
```

再深点，就有感觉了：

```
>>> root[0][0].tag
'title'
>>> root[0][0].attrib
{'lang': 'en'}
>>> root[0][0].text
'Everyday Italian'
```

对于 ElementTree 对象，有一个 iter 方法可以对指定名称的子节点进行深度优先遍历。例如：

```
>>> for ele in tree.iter(tag="book"):      #遍历名称为 book 的节点
...     print ele.tag, ele.attrib
...
book {'category': 'COOKING'}
book {'category': 'CHILDREN'}
book {'category': 'WEB'}
```

```
>>> for ele in tree.iter(tag="title"):      #遍历名称为 title 的节点
...     print ele.tag, ele.attrib, ele.text
...
title {'lang': 'en'} Everyday Italian
title {'lang': 'en'} Harry Potter
title {'lang': 'en'} Learning XML
```

如果不指定元素名称，就是将所有的元素遍历一边。

```
>>> for ele in tree.iter():
...     print ele.tag, ele.attrib
...
bookstore {}
book {'category': 'COOKING'}
title {'lang': 'en'}
author {}
year {}
price {}
book {'category': 'CHILDREN'}
title {'lang': 'en'}
author {}
year {}
price {}
book {'category': 'WEB'}
title {'lang': 'en'}
author {}
year {}
price {}
```

除了上面的方法，还可以通过路径，搜索到指定的元素，读取其内容。这就是 xpath。此处对 xpath 不详解，如果要了解可以到网上搜索有关信息。

```
>>> for ele in tree.iterfind("book/title"):
...     print ele.text
...
Everyday Italian
Harry Potter
Learning XML
```

利用 findall() 方法，也可以实现查找功能：

```
>>> for ele in tree.findall("book"):
...     title = ele.find('title').text
...     price = ele.find('price').text
...     lang = ele.find('title').attrib
...     print title, price, lang
```

```
...
Everyday Italian 30.00 {'lang': 'en'}
Harry Potter 29.99 {'lang': 'en'}
Learning XML 39.95 {'lang': 'en'}
```

编辑

除了读取有关数据之外，还能对 xml 进行编辑，即增删改查功能。还是以上面的 xml 文档为例：

```
>>> root[1].tag
'book'
>>> del root[1]
>>> for ele in root:
...     print ele.tag
...
book
book
```

如此，成功删除了一个节点。原来有三个 book 节点，现在就还剩两个了。打开源文件再看看，是不是正好少了第二个节点呢？一定很让你失望，源文件居然没有变化。

的确如此，源文件没有变化，这就对了。因为至此的修改动作，还是停留在内存中，还没有将修改结果输出到文件。不要忘记，我们是在内存中建立的 ElementTree 对象。再这样做：

```
>>> import os
>>> outpath = os.getcwd()
>>> file = outpath + "/22601.xml"
```

把当前文件路径拼装好。然后：

```
>>> tree.write(file)
```

再看源文件，已经变成两个节点了。

除了删除，也能够修改：

```
>>> for price in root.iter("price"):      #原来每本书的价格
...     print price.text
...
30.00
39.95
>>> for price in root.iter("price"):      #每本上涨 7 元，并且增加属性标记
...     new_price = float(price.text) + 7
...     price.text = str(new_price)
...     price.set("updated","up")
```

```
...
>>> tree.write(file)
```

查看源文件：

```
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price updated="up">37.0</price>
  </book>
  <book category="WEB">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price updated="up">46.95</price>
  </book>
</bookstore>
```

不仅价格修改了，而且在 price 标签里面增加了属性标记。干得不错。

上面用 `del` 来删除某个元素，其实，在编程中，这个用的不多，更喜欢用 `remove()` 方法。比如我要删除 `price > 40` 的书。可以这么做：

```
>>> for book in root.findall("book"):
...     price = book.find("price").text
...     if float(price) > 40.0:
...         root.remove(book)
...
>>> tree.write(file)
```

于是就这样了：

```
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price updated="up">37.0</price>
  </book>
</bookstore>
```

接下来就要增加元素了。

```
>>> import xml.etree.cElementTree as ET
>>> tree = ET.ElementTree(file="22601.xml")
>>> root = tree.getroot()
>>> ET.SubElement(root, "book")      #在 root 里面添加 book 节点
<Element 'book' at 0xb71c7578>
>>> for ele in root:
...     print ele.tag
...
book
book
>>> b2 = root[1]                  #得到新增的 book 节点
>>> b2.text = "Python"           #添加内容
>>> tree.write("22601.xml")
```

查看源文件：

```
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price updated="up">37.0</price>
  </book>
  <book>python</book>
</bookstore>
```

常用属性和方法总结

ET 里面的属性和方法不少，这里列出常用的，供使用中备查。

Element 对象

常用属性：

- tag: string, 元素数据种类
- text: string, 元素的内容
- attrib: dictionary, 元素的属性字典
- tail: string, 元素的尾形

针对属性的操作

- clear(): 清空元素的后代、属性、text 和 tail 也设置为 None
- get(key, default=None): 获取 key 对应的属性值，如该属性不存在则返回 default 值

- items(): 根据属性字典返回一个列表，列表元素为(key, value)
- keys(): 返回包含所有元素属性键的列表
- set(key, value): 设置新的属性键与值

针对后代的操作

- append(subelement): 添加直系子元素
- extend(subelements): 增加一串元素对象作为子元素
- find(match): 寻找第一个匹配子元素，匹配对象可以为 tag 或 path
- findall(match): 寻找所有匹配子元素，匹配对象可以为 tag 或 path
- findtext(match): 寻找第一个匹配子元素，返回其 text 值。匹配对象可以为 tag 或 path
- insert(index, element): 在指定位置插入子元素
- iter(tag=None): 生成遍历当前元素所有后代或者给定 tag 的后代的迭代器
- iterfind(match): 根据 tag 或 path 查找所有的后代
- itertext(): 遍历所有后代并返回 text 值
- remove(subelement): 删除子元素

ElementTree 对象

- find(match)
- findall(match)
- findtext(match, default=None)
- getroot(): 获取根节点.
- iter(tag=None)
- iterfind(match)
- parse(source, parser=None): 装载 xml 对象，source 可以为文件名或文件类型对象.
- write(file, encoding="us-ascii", xml_declaration=None, default_namespace=None, method="xml")

一个实例

最后，提供一个参考，这是一篇来自网络的文章：[Python xml 属性、节点、文本的增删改](#)，本文的源码我也复制到下面，请读者参考：

实现思想：

使用 ElementTree，先将文件读入，解析成树，之后，根据路径，可以定位到树的每个节点，再对节点进行修改，最后直接将其输出。

```

#!/usr/bin/Python
# -*- coding=utf-8 -*-
# author : wklken@yeah.net
# date: 2012-05-25
# version: 0.1

from xml.etree.ElementTree import ElementTree,Element

def read_xml(in_path):
    """
    读取并解析 xml 文件
    in_path: xml 路径
    return: ElementTree
    """
    tree = ElementTree()
    tree.parse(in_path)
    return tree

def write_xml(tree, out_path):
    """
    将 xml 文件写出
    tree: xml 树
    out_path: 写出路径
    """
    tree.write(out_path, encoding="utf-8",xml_declaration=True)

def if_match(node, kv_map):
    """
    判断某个节点是否包含所有传入参数属性
    node: 节点
    kv_map: 属性及属性值组成的 map
    """
    for key in kv_map:
        if node.get(key) != kv_map.get(key):
            return False
    return True

#-----search -----

def find_nodes(tree, path):

```

```

"""
    查找某个路径匹配的所有节点
    tree: xml 树
    path: 节点路径
"""

return tree.findall(path)

def get_node_by_keyvalue(nodelist, kv_map):
    """
        根据属性及属性值定位符合的节点，返回节点
        nodelist: 节点列表
        kv_map: 匹配属性及属性值 map
    """

    result_nodes = []
    for node in nodelist:
        if if_match(node, kv_map):
            result_nodes.append(node)
    return result_nodes

#-----change -----

def change_node_properties(nodelist, kv_map, is_delete=False):
    """
        修改/增加 /删除 节点的属性及属性值
        nodelist: 节点列表
        kv_map: 属性及属性值 map
    """

    for node in nodelist:
        for key in kv_map:
            if is_delete:
                if key in node.attrib:
                    del node.attrib[key]
            else:
                node.set(key, kv_map.get(key))

def change_node_text(nodelist, text, is_add=False, is_delete=False):
    """
        改变/增加/删除一个节点的文本
        nodelist: 节点列表
        text : 更新后的文本
    """

    for node in nodelist:
        if is_add:
            node.text += text
        elif is_delete:
            node.text = ''

```

```

node.text = ""
else:
    node.text = text

def create_node(tag, property_map, content):
    """
    新造一个节点
    tag:节点标签
    property_map:属性及属性值 map
    content: 节点闭合标签里的文本内容
    return 新节点
    """

    element = Element(tag, property_map)
    element.text = content
    return element

def add_child_node(nodelist, element):
    """
    给一个节点添加子节点
    nodelist: 节点列表
    element: 子节点
    """

    for node in nodelist:
        node.append(element)

def del_node_by_tagkeyvalue(nodelist, tag, kv_map):
    """
    通过属性及属性值定位一个节点，并删除之
    nodelist: 父节点列表
    tag:子节点标签
    kv_map: 属性及属性值列表
    """

    for parent_node in nodelist:
        children = parent_node.getchildren()
        for child in children:
            if child.tag == tag and if_match(child, kv_map):
                parent_node.remove(child)

if __name__ == "__main__":
    #1. 读取 xml 文件
    tree = read_xml("./test.xml")

    #2. 属性修改

```

```

#A. 找到父节点
nodes = find_nodes(tree, "processors/processer")

#B. 通过属性准确定位子节点
result_nodes = get_node_by_keyvalue(nodes, {"name": "BProcessor"})

#C. 修改节点属性
change_node_properties(result_nodes, {"age": "1"})

#D. 删除节点属性
change_node_properties(result_nodes, {"value": ""}, True)

#3. 节点修改
#A.新建节点
a = create_node("person", {"age": "15", "money": "200000"}, "this is the first content")

#B.插入到父节点之下
add_child_node(result_nodes, a)

#4. 删除节点
#定位父节点
del_parent_nodes = find_nodes(tree, "processors/services/service")

#准确定位子节点并删除之
target_del_node = del_node_by_tagkeyvalue(del_parent_nodes, "chain", {"sequency": "chain1"})

#5. 修改节点文本
#定位节点
text_nodes = get_node_by_keyvalue(find_nodes(tree, "processors/services/service/chain"), {"sequency": "chain3"})
change_node_text(text_nodes, "new text")

#6. 输出到结果文件
write_xml(tree, "./out.xml")

```

操作对象（原始 XML 文件）：

```

<?xml version="1.0" encoding="UTF-8"?>
<framework>
  <processors>
    <processor name="AProcessor" file="lib64/A.so"
      path="/tmp">
    </processor>
    <processor name="BProcessor" file="lib64/B.so" value="fordelete">
    </processor>
    <processor name="BProcessor" file="lib64/B.so22222222"/>
  
```

```

<services>
    <service name="search" prefix="/bin/search?">
        output_formatter="OutPutFormatter:service_inc">

        <chain sequency="chain1"/>
        <chain sequency="chain2"></chain>
    </service>
    <service name="update" prefix="/bin/update?">
        <chain sequency="chain3" value="fordelete"/>
    </service>
</services>
</processors>
</framework>

```

执行程序之后，得到的结果文件：

```

<?xml version='1.0' encoding='utf-8'?>
<framework>
    <processors>
        <processer file="lib64/A.so" name="AProcesser" path="/tmp">
        </processer>
        <processer age="1" file="lib64/B.so" name="BProcesser">
            <person age="15" money="200000">this is the firest content</person>
        </processer>
        <processer age="1" file="lib64/B.so2222222" name="BProcesser">
            <person age="15" money="200000">this is the firest content</person>
        </processer>

    <services>
        <service name="search" output_formatter="OutPutFormatter:service_inc"
            prefix="/bin/search?">

            <chain sequency="chain2" />
        </service>
        <service name="update" prefix="/bin/update?">
            <chain sequency="chain3" value="fordelete">new text</chain>
        </service>
    </services>
</processors>
</framework>

```

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

标准库 (8)

json

就传递数据而言，xml 是一种选择，还有另外一种，就是 json，它是一种轻量级的数据交换格式，如果读者要做 web 编程，是会用到它的。根据维基百科的相关内容，对 json 了解一二：

JSON (JavaScript Object Notation) 是一种由道格拉斯 · 克罗克福特构想设计、轻量级的资料交换语言，以文字为基础，且易于让人阅读。尽管 JSON 是 Javascript 的一个子集，但 JSON 是独立于语言的文本格式，并且采用了类似于 C 语言家族的一些习惯。

关于 json 更为详细的内容，可以参考其官方网站：<http://www.json.org>

从官方网站上摘取部分，了解一下 json 的结构：

JSON 建构于两种结构：

- “名称/值” 对的集合（A collection of name/value pairs）。不同的语言中，它被理解为对象（object），纪录（record），结构（struct），字典（dictionary），哈希表（hash table），有键列表（keyed list），或者关联数组（associative array）。
- 值的有序列表（An ordered list of values）。在大部分语言中，它被理解为数组（array）。

python 标准库中有 json 模块，主要是执行序列化和反序列化功能：

- 序列化：encoding，把一个 Python 对象编码转化成 json 字符串
- 反序列化：decoding，把 json 格式字符串解码转换为 Python 数据对象

基本操作

json 模块相对 xml 单纯了很多：

```
>>> import json
>>> json.__all__
['dump', 'dumps', 'load', 'loads', 'JSONDecoder', 'JSONEncoder']
```

encoding: dumps()

```
>>> data = [{"name": "qiwsir", "lang": ("python", "english"), "age": 40}]
>>> print data
```

```
[{"lang": ("python", "english"), "age": 40, "name": "qiwsir"}]
>>> data_json = json.dumps(data)
>>> print data_json
[{"lang": ["python", "english"], "age": 40, "name": "qiwsir"}]
```

encoding 的操作是比较简单的, 请注意观察 data 和 data_json 的不同——lang 的值从元组编程了列表, 还有不同:

```
>>> type(data_json)
<type 'str'>
>>> type(data)
<type 'list'>
```

将 Python 对象转化为 json 类型, 是按照下表所示对照关系转化的:

Python==>	json
dict	object
list, tuple	array
str, unicode	string
int, long, float	number
True	true
False	false
None	null

decoding: loads()

decoding 的过程也像上面一样简单:

```
>>> new_data = json.loads(data_json)
>>> new_data
[{"u'lang': [u'python', u'english'], u'age': 40, u'name': u'qiwsir'}]
```

需要注意的是, 解码之后, 并没有将元组还原。

解码的数据类型对应关系:

json==>	Python
object	dict
array	list
string	unicode
number(int)	int, long
number(real)	float
true	True
false	False

json==>	Python
null	None

对人友好

上面的 data 都不是很长，还能凑合阅读，如果很长了，阅读就有难度了。所以，json 的 dumps() 提供了可选参数，利用它们能在输出上对人更友好（这对机器是无所谓的）。

```
>>> data_j = json.dumps(data, sort_keys=True, indent=2)
>>> print data_j
[
{
    "age": 40,
    "lang": [
        "python",
        "english"
    ],
    "name": "qiwsir"
}
]
```

`sort_keys=True` 意思是按照键的字典顺序排序，`indent=2` 是让每个键值对显示的时候，以缩进两个字符对齐。这样的视觉效果好多了。

大 json 字符串

如果数据不是很大，上面的操作足够了。但是，上面操作是将数据都读入内存，如果太大就不行了。怎么办？json 提供了 `load()` 和 `dump()` 函数解决这个问题，注意，跟上面已经用过的函数相比，是不同的，请仔细观察。

```
>>> import tempfile #临时文件模块
>>> data
[{"lang": ("Python", "english"), "age": 40, "name": "qiwsir"}]
>>> f = tempfile.NamedTemporaryFile(mode='w+')
>>> json.dump(data, f)
>>> f.flush()
>>> print open(f.name, "r").read()
[{"lang": ["Python", "english"], "age": 40, "name": "qiwsir"}]
```

自定义数据类型

一般情况下，用的数据类型都是 Python 默认的。但是，我们学习过类后，就知道，自己可以定义对象类型的。比如：

以下代码参考：[Json 概述以及 Python 对 json 的相关操作](#)

```
#!/usr/bin/env Python
# coding=utf-8

import json

class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return 'Person Object name : %s , age : %d' % (self.name, self.age)

def object2dict(obj):  #convert Person to dict
    d = {}
    d['__class__'] = obj.__class__.__name__
    d['__module__'] = obj.__module__
    d.update(obj.__dict__)
    return d

def dict2object(d):  #convert dict ot Person
    if '__class__' in d:
        class_name = d.pop('__class__')
        module_name = d.pop('__module__')
        module = __import__(module_name)
        class_ = getattr(module, class_name)
        args = dict((key.encode('ascii'), value) for key, value in d.items())  #get args
        inst = class_(**args)  #create new instance
    else:
        inst = d
    return inst

if __name__ == '__main__':
    p = Person('Peter', 40)
    print p
    d = object2dict(p)
    print d
    o = dict2object(d)
    print type(o), o

    dump = json.dumps(p, default=object2dict)
```

```
print dump
load = json.loads(dump, object_hook=dict2object)
print load
```

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

第三方库

标准库的内容已经非常多了，前面仅仅列举几个，但是 Python 给编程者的支持还不仅仅在于标准库，它还有不可胜数的第三方库。因此，如果作为一个 Python 编程者，即使你达到了 master 的水平，最好的还是要在做某个事情之前，在网上搜一下是否有标准库或者第三方库替你完成那件事。因为，伟大的艾萨克·牛顿爵士说过：

如果我比别人看得更远，那是因为我站在巨人的肩上。

编程，就要站在巨人的肩上。标准库和第三方库以及其提供者，就是巨人，我们本应当谦卑地向其学习，并应用其成果。

安装第三方库

要是用第三方库，第一步就是要安装，在本地安装完毕，就能如同标准库一样使用了。其安装方法如下：

方法一：利用源码安装

在 github.com 网站可以下载第三方库的源码（或者其它途径），得到源码之后，在本地安装。

一般情况，得到的码格式大概都是 zip、tar.zip、tar.bz2 格式的压缩包。解压这些包，进入其文件夹，通常会看见一个 setup.py 的文件。如果是 Linux 或者 Mac(我是用 ubuntu，特别推荐哦)，就在这里运行 shell，执行命令：

```
Python setup.py install
```

如果用的是 windows，需要打开命令行模式，执行上述指令即可。

如此，就能把这个第三库安装到系统里。具体位置，要视操作系统和你当初安装 Python 环境时设置的路径而定。默认条件下，windows 是在 C:\Python2.7\Lib\site-packages，Linux 在 /usr/local/lib/python2.7/dist-packages（这个只是参考，不同发行版会有差别，具体请读者根据自己的操作系统，自己找找），Mac 在 /Library/Python/2.7/site-packages。

有安装就要有卸载，卸载所安装的库非常简单，只需要到相应系统的 site-packages 目录，直接删掉库文件即卸载。

方法二：pip

用源码安装，不是我推荐的，我推荐的是用第三方库的管理工具安装。有一个网站，是专门用来存储第三方库的，所有在这个网站上的，都能用 pip 或者 easy_install 这种安装工具来安装。这个网站的地址：<https://pypi.org/pypi>

首先，要安装 pip (Python 官方推荐这个，我当然要顺势了，所以，就只介绍并且后面也只使用这个工具)。如果读者跟我一样，用的是 ubuntu 或者其它某种 Linux，基本不用这个操作，在安装操作系统的时候已经默认把这个东西安装好了（这还不是用 ubuntu 的理由吗？）。如果因为什么原因，没有安装，可以使用如下方法：

Debian and Ubuntu:

```
sudo apt-get install Python-pip
```

Fedora and CentOS:

```
sudo yum install python-pip
```

当然，也可以这里下载文件<get-pip.py>，然后执行 `Python get-pip.py` 来安装。这个方法也适用于 windows。

pip 安装好了。如果要安装第三方库，只需要执行 `pip install XXXXXX` (XXXXXX 代表第三方库的名字) 即可。

当第三方库安装完毕，接下来的使用就如同前面标准库一样。

举例：requests 库

以 requests 模块为例，来说明第三方库的安装和使用。之所以选这个，是因为前面介绍了 urllib 和 urllib2 两个标准库的模块，与之有类似功能的第三方库中 requests 也是一个用于在程序中进行 http 协议下的 get 和 post 请求的模块，并且被网友说成“好用的要哭”。

说明：下面的内容是网友 1world0x00 提供，我仅做了适当编辑。

安装

```
pip install requests
```

安装好之后，在交互模式下：

```
>>> import requests
>>> dir(requests)
['ConnectionError', 'HTTPError', 'NullHandler', 'PreparedRequest', 'Request', 'RequestException', 'Response', 'Session', 'T
```

从上面的列表中可以看出，在 http 中常用到的 get，cookies，post 等都赫然在目。

get 请求

```
>>> r = requests.get("http://www.itdiffer.com")
```

得到一个请求的实例，然后：

```
>>> r.cookies
<<class 'requests.cookies.RequestsCookieJar'>[]>
```

这个网站对客户端没有写任何 cookies 内容。换一个看看：

```
>>> r = requests.get("http://www.1world0x00.com")
```

```
>>> r.cookies
```

```
<<class 'requests.cookies.RequestsCookieJar'>[Cookie(version=0, name='PHPSESSID', value='buqj70k7f9rrg51emsvatveda2; p')
```

原来这样呀。继续，还有别的属性可以看看。

```
>>> r.headers
{'x-powered-by': 'PHP/5.3.3', 'transfer-encoding': 'chunked', 'set-cookie': 'PHPSESSID=buqj70k7f9rrg51emsvatveda2; p'
>>> r.encoding
'UTF-8'
>>> r.status_code
200
```

下面这个比较长，是网页的内容，仅仅截取显示部分：

```
>>> print r.text
<!DOCTYPE html>
<html lang="zh-CN">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>1world0x00sec</title>
    <link rel="stylesheet" href="http://www.1world0x00.com/usr/themes/default/style.min.css">
    <link rel="canonical" href="http://www.1world0x00.com/" />
    <link rel="stylesheet" type="text/css" href="http://www.1world0x00.com/usr/plugins/CodeBox/css/codebox.css" />
    <meta name="description" content="爱生活，爱拉芳。不装逼还能做朋友。" />
    <meta name="keywords" content="php" />
    <link rel="pingback" href="http://www.1world0x00.com/index.php?action/xmlrpc" />
```

.....

请求发出后，requests 会基于 http 头部对相应的编码做出有根据的推测，当你访问 r.text 之时，requests 会使用其推测的文本编码。你可以找出 requests 使用了什么编码，并且能够使用 r.encoding 属性来改变它。

```
>>> r.content
'\xeef\xbb\xbf\xef\xbb\xbf<!DOCTYPE html>\n<html lang="zh-CN">\n  <head>\n    <meta charset="utf-8">\n    <meta name="viewport" content="width=device-width, initial-scale=1.0" />\n  </head>\n  <body>\n    <h1>Hello World</h1>\n  </body>\n</html>'
```

以二进制的方式打开服务器并返回数据。

post 请求

requests 发送 post 请求，通常你会想要发送一些编码为表单的数据——非常像一个 html 表单。要实现这个，只需要简单地传递一个字典给 data 参数。你的数据字典在发出请求时会自动编码为表单形式。

```
>>> import requests
>>> payload = {"key1": "value1", "key2": "value2"}
>>> r = requests.post("http://httpbin.org/post")
>>> r1 = requests.post("http://httpbin.org/post", data=payload)
```

http 头部

```
>>> r.headers['content-type']
'application/json'
```

注意，在引号里面的内容，不区分大小写 'CONTENT-TYPE' 也可以。

还能够自定义头部：

```
>>> r.headers['content-type'] = 'adad'
>>> r.headers['content-type']
'adad'
```

注意，当定制头部的时候，如果需要定制的项目有很多，需要用到数据类型为字典。

网上有一个更为详细叙述有关 requests 模块的网页，可以参考：http://requests-docs-cn.readthedocs.org/zh_CN/latest/index.html

总目录 (页 0)

如果你认为有必要打赏我，请通过支付宝：qjwsir@126.com,不胜感激。

HTML



T

8

保存数据



unity



HTML



将数据存入文件

在《[文件\(1\)](#)》([页 0](#))中，已经学习了如何读写文件。

如果在程序中，有数据要保存到磁盘中，放到某个文件中是一种不错的方法。但是，如果像以前那样存，未免有点凌乱，并且没有什么良好的存储格式，导致数据以后被读出来的时候遇到麻烦，特别是不能让另外的使用者很好地理解。不要忘记了，编程是一个合作的活。还有，存储的数据不一定都是类似字符串、整数那种基础类型的。

总而言之，需要将要存储的对象格式化（或者叫做序列化），才好存好取。这就有点类似集装箱的作用。

所以，要用到本讲中提供的方式。

pickle

pickle 是标准库中的一个模块，还有跟它完全一样的叫做 cpickle，两者的区别就是后者更快。所以，下面操作中，不管是用 `import pickle`，还是用 `import cpickle as pickle`，在功能上都是一样的。

```
>>> import pickle
>>> integers = [1, 2, 3, 4, 5]
>>> f = open("22901.dat", "wb")
>>> pickle.dump(integers, f)
>>> f.close()
```

用 `pickle.dump(obj, file[, protocol])` 将数据 `integers` 保存到了文件 `22901.dat` 中。如果你要打开这个文件，看里面的内容，可能有点失望，但是，它对计算机是友好的。这个步骤，可以称之为将对象序列化。用到的方法是：

```
pickle.dump(obj,file[,protocol])
```

- `obj`: 序列化对象，上面的例子中是一个列表，它是基本类型，也可以序列化自己定义的类型。
- `file`: 一般情况下是要写入的文件。更广泛地可以理解为为拥有 `write()` 方法的对象，并且能接受字符串为参数，所以，它还可以是一个 `StringIO` 对象，或者其它自定义满足条件的对象。
- `protocol`: 可选项。默认为 `False`（或者说 `0`），是以 ASCII 格式保存对象；如果设置为 `1` 或者 `True`，则以压缩的二进制格式保存对象。

下面换一种数据格式，并且做对比：

```
>>> import pickle
>>> d = {}
```

```
>>> integers = range(9999)
>>> d["i"] = integers      #下面将这个 dict 格式的对象存入文件

>>> f = open("22902.dat", "wb")
>>> pickle.dump(d, f)      #文件中以 ascii 格式保存数据
>>> f.close()

>>> f = open("22903.dat", "wb")
>>> pickle.dump(d, f, True)  #文件中以二进制格式保存数据
>>> f.close()

>>> import os
>>> s1 = os.stat("22902.dat").st_size  #得到两个文件的大小
>>> s2 = os.stat("22903.dat").st_size

>>> print "%d, %d, %.2f%%" % (s1, s2, (s2+0.0)/s1*100)
68903, 29774, 43.21%
```

比较结果发现，以二进制方式保存的文件比以 ascii 格式保存的文件小很多，前者约是后者的 43%。

所以，在序列化的时候，特别是面对较大对象时，建议将 dump() 的参数 True 设置上，虽然现在存储设备的价格便宜，但是能省还是省点比较好。

存入文件，仅是一个目标，还有另外一个目标，就是要读出来，也称之为反序列化。

```
>>> integers = pickle.load(open("22901.dat", "rb"))
>>> print integers
[1, 2, 3, 4, 5]
```

就是前面存入的那个列表。再看看被以二进制存入的那个文件：

```
>>> f = open("22903.dat", "rb")
>>> d = pickle.load(f)
>>> print d
{'i': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, .... #省略后面的数字}
>>> f.close()
```

还是有自己定义数据类型的需要，这种类型是否可以用上述方式存入文件并读出来呢？看下面的例子：

```
>>> import cPickle as pickle      #cPickle 更快
>>> import StringIO            #标准库中的一个模块，跟 file 功能类似，只不过是在内存中操作“文件”

>>> class Book(object):        #自定义一种类型
...     def __init__(self, name):
...         self.name = name
...     def my_book(self):
```

```
...     print "my book is: ", self.name
...
>>> pybook = Book("<from beginner to master>")
>>> pybook.my_book()
my book is: <from beginner to master>

>>> file = StringIO.StringIO()
>>> pickle.dump(pybook, file, 1)
>>> print file.getvalue()      #查看“文件”内容，注意下面不是乱码
ccopy_reg
_reconstructor
q?(c__main__
Book
q?c__builtin__
object
q?NtRq?}qU?nameq?U?<from beginner to master>sb.

>>> pickle.dump(pybook, file)  #换一种方式，再看内容，可以比较一下
>>> print file.getvalue()      #视觉上，两者就有很大差异
ccopy_reg
_reconstructor
q?(c__main__
Book
q?c__builtin__
object
q?NtRq?}qU?nameq?U?<from beginner to master>sb.ccopy_reg
_reconstructor
p1
(c__main__
Book
p2
c__builtin__
object
p3
NtRp4
(dp5
S'name'
p6
S'<from beginner to master>'
p7
sb.
```

如果要从文件中读出来：

```
>>> file.seek(0)    #找到对应类型
>>> pybook2 = pickle.load(file)
>>> pybook2.my_book()
my book is: <from beginner to master>
>>> file.close()
```

shelve

pickle 模块已经表现出它足够好的一面了。不过，由于数据的复杂性，pickle 只能完成一部分工作，在另外更复杂的情况下，它就稍显麻烦了。于是，又有了 shelve。

shelve 模块也是标准库中的。先看一下基本操作：写入和读取

```
>>> import shelve
>>> s = shelve.open("22901.db")
>>> s["name"] = "www.itdiffer.com"
>>> s["lang"] = "python"
>>> s["pages"] = 1000
>>> s["contents"] = {"first": "base knowledge", "second": "day day up"}
>>> s.close()
```

以上完成了数据写入的过程。其实，这更接近数据库的样式了。下面是读取。

```
>>> s = shelve.open("22901.db")
>>> name = s["name"]
>>> print name
www.itdiffer.com
>>> contents = s["contents"]
>>> print contents
{'second': 'day day up', 'first': 'base knowledge'}
```

当然，也可以用 for 语句来读：

```
>>> for k in s:
...     print k, s[k]
...
contents {'second': 'day day up', 'first': 'base knowledge'}
lang python
pages 1000
name www.itdiffer.com
```

不管是写，还是读，都似乎要简化了。所建立的对象s，就如同字典一样，可称之为类字典对象。所以，可以如同操作字典那样来操作它。

但是，要小心坑：

```
>>> f = shelve.open("22901.db")
>>> f["author"]
['qiwsir']
>>> f["author"].append("Hetz") #试图增加一个
>>> f["author"]          #坑就在这里
['qiwsir']
>>> f.close()
```

当试图修改一个已有键的值时，没有报错，但是并没有修改成功。要填平这个坑，需要这样做：

```
>>> f = shelve.open("22901.db", writeback=True) #多一个参数 True
>>> f["author"].append("Hetz")
>>> f["author"]          #没有坑了
['qiwsir', 'Hetz']
>>> f.close()
```

还用 for 循环一下：

```
>>> f = shelve.open("22901.db")
>>> for k,v in f.items():
...     print k,":",v
...
contents : {'second': 'day day up', 'first': 'base knowledge'}
lang : python
pages : 1000
author : ['qiwsir', 'Hetz']
name : www.itdiffer.com
```

shelve 更像数据库了。

不过，它还不是真正的数据库。真正的数据库在后面。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

mysql 数据库 (1)

尽管用文件形式将数据保存到磁盘，已经是一种不错的方式。但是，人们还是发明了更具有格式化特点，并且写入和读取更快速便捷的东西——数据库（如果阅读港台的资料，它们称之为“资料库”）。维基百科对数据库有比较详细的说明：

数据库指的是以一定方式储存在一起、能为多个用户共享、具有尽可能小的冗余度、与应用程序彼此独立的数据集合。

到目前为止，地球上三种类型的数据：

- 关系型数据库：MySQL、Microsoft Access、SQL Server、Oracle、...
- 非关系型数据库：MongoDB、BigTable(Google)、...
- 键值数据库：Apache Cassandra(Facebook)、LevelDB(Google) ...

在本教程中，我们主要介绍常用的开源的数据库，其中 MySQL 是典型代表。

概况

MySQL 是一个使用非常广泛的数据库，很多网站都是用它。关于这个数据库有很多传说。例如[维基百科上这么](#)说：

MySQL（官方发音为英语发音：/maɪ̯ .ɛskju:ˈeɪl/ "My S-Q-L", [1]，但也经常读作英语发音：/maɪ̯ 'si:kwəl/ "My Sequel"）原本是一个开放源代码的关系数据库管理系统，原开发者为瑞典的 MySQL AB 公司，该公司于 2008 年被升阳微系统（Sun Microsystems）收购。2009 年，甲骨文公司（Oracle）收购升阳微系统公司，MySQL 成为 Oracle 旗下产品。

MySQL 在过去由于性能高、成本低、可靠性好，已经成为最流行的开源数据库，因此被广泛地应用在 Internet 上的中小型网站中。随着 MySQL 的不断成熟，它也逐渐用于更多大规模网站和应用，比如维基百科、Google 和 Facebook 等网站。非常流行的开源软件组合 LAMP 中的“M”指的就是 MySQL。

但被甲骨文公司收购后，Oracle 大幅调涨 MySQL 商业版的售价，且甲骨文公司不再支持另一个自由软件项目 OpenSolaris 的发展，因此导致自由软件社区们对于 Oracle 是否还会持续支持 MySQL 社区版（MySQL 之中唯一的免费版本）有所隐忧，因此原先一些使用 MySQL 的开源软件逐渐转向其它的数据库。例如维基百科已于 2013 年正式宣布将从 MySQL 迁移到 MariaDB 数据库。

不管怎么着，MySQL 依然一个不错的数据库选择，足够支持读者完成一个相当不小的网站。

安装

你的电脑或许不会天生就有 MySQL（是不是有的操作系统，在安装的时候就内置了呢？的确有，所以特别推荐 Linux 的某发行版），它本质上也是一个程序，若有必要，须安装。

我用 ubuntu 操作系统演示，因为我相信读者将来在真正的工程项目中，多数情况下是要操作 Linux 系统的服务器，并且，我酷爱用 ubuntu。还有，本教程的目标是 from beginner to master，不管是真是的 master，总要装得像，Linux 能够给你撑门面。

第一步，在 shell 端运行如下命令：

```
sudo apt-get install mysql-server
```

运行完毕，就安装好了这个数据库。是不是很简单呢？当然，当然，还要进行配置。

第二步，配置 MySQL

安装之后，运行：

```
service mysqld start
```

启动 mysql 数据库。然后进行下面的操作，对其进行配置。

默认的 MySQL 安装之后根用户是没有密码的，注意，这里有一个名词“根用户”，其用户名是：root。运行：

```
$mysql -u root
```

在这里之所以用 -u root 是因为我现在是一般用户（firehare），如果不加 -u root 的话，mysql 会以为是 firehare 在登录。

进入 mysql 之后，会看到>符号开头，这就是 mysql 的命令操作界面了。

下面设置 Mysql 中的 root 用户密码了，否则，Mysql 服务无安全可言了。

```
mysql> GRANT ALL PRIVILEGES ON *.* TO root@localhost IDENTIFIED BY "123456";
```

用 123456 做为 root 用户的密码，应该是非常愚蠢的，如果在真正的项目中，最好别这样做，要用大小写字母与数字混合的密码，且不少于 8 位。

以后如果在登录数据库，就可以用刚才设置的密码了。

运行

安装之后，就要运行它，并操作这个数据库。

```
$ mysql -u root -p  
Enter password:
```

输入数据库的密码，之后出现：

```
Welcome to the MySQL monitor. Commands end with ; or \g.  
Your MySQL connection id is 373  
Server version: 5.5.38-0ubuntu0.14.04.1 (Ubuntu)  
  
Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.  
  
Oracle is a registered trademark of Oracle Corporation and/or its  
affiliates. Other names may be trademarks of their respective  
owners.  
  
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
mysql>
```

看到这个界面内容，就说明你已经进入到数据里面了。接下来就可以对这个数据进行操作。例如：

```
mysql> show databases;  
+-----+  
| Database |  
+-----+  
| information_schema |  
| carstore |  
| cutvideo |  
| itdiffer |  
| mysql |  
| performance_schema |  
| test |  
+-----+
```

用这个命令，就列出了当前已经有的数据库。

对数据库的操作，除了用命令之外，还可以使用一些可视化工具。比如 phpmyadmin 就是不错的。

更多数据库操作的知识，这里就不介绍了，读者可以参考有关书籍。

MySQL 数据库已经安装好，但是 Python 还不能操作它，还要继续安装 Python 操作数据库的模块——Python-MySQLdb

安装 Python-MySQLdb

Python-MySQLdb 是一个接口程序，Python 通过它对 mysql 数据实现各种操作。

在编程中，会遇到很多类似的接口程序，通过接口程序对另外一个对象进行操作。接口程序就好比钥匙，如果要开锁，人直接用手指去捅，肯定是不行的，那么必须借助工具，插入到锁孔中，把锁打开，之后，门开了，就可以操作门里面的东西了。那么打开锁的工具就是接口程序。谁都知道，用对应的钥匙开锁是最好的，如果用别的工具（比如锤子），或许不便利（其实还分人，也就是人开锁的水平，如果是江洋大盗或者小毛贼什么的，擅长开锁，用别的工具也便利了），也就是接口程序不同，编码水平不同，都是考虑因素。

啰嗦这么多，一言蔽之，Python-MySQLdb 就是打开 MySQL 数据库的钥匙。

如果要源码安装，可以这里下载 Python-mysqldb:<https://pypi.Python.org/pypi/MySQL-Python/>

下载之后就可以安装了。

ubuntu 下可以这么做：

```
sudo apt-get install build-essential Python-dev libmysqlclient-dev
sudo apt-get install Python-MySQLdb
```

也可以用 pip 来安装：

```
pip install mysql-Python
```

安装之后，在 python 交互模式下：

```
>>> import MySQLdb
```

如果不报错，恭喜你，已经安装好了。如果报错，恭喜你，可以借着错误信息提高自己的计算机水平了，请求助于 google 大神。

连接数据库

要先找到老婆，才能谈如何养育自己的孩子，同理连接数据库之先要建立数据库。

```
$ mysql -u root -p
Enter password:
```

进入到数据库操作界面：

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 373
Server version: 5.5.38-0ubuntu0.14.04.1 (Ubuntu)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

输入如下命令，建立一个数据库：

```
mysql> create database qiwsirtest character set utf8;
Query OK, 1 row affected (0.00 sec)
```

注意上面的指令，如果仅仅输入：create database qiwsirtest，也可以，但是，我在后面增加了 character set utf8，意思是所建立的数据库 qiwsirtest，编码是 utf-8 的，这样存入汉字就不是乱码了。

看到那一行提示：Query OK, 1 row affected (0.00 sec)，就说明这个数据库已经建立好了，名字叫做:qiwsirtest

数据库建立之后，就可以用 Python 通过已经安装的 mysqlDb 来连接这个名字叫做 qiwsirtest 的库了。

```
>>> import MySQLdb
>>> conn = MySQLdb.connect(host="localhost",user="root",passwd="123123",db="qiwsirtest",port=3306,charset="utf8")
```

逐个解释上述命令的含义：

- host:等号的后面应该填写 mysql 数据库的地址，因为数据库就在本机上（也称作本地），所以使用 localhost，注意引号。如果在其它的服务器上，这里应该填写 ip 地址。一般中小型的网站，数据库和程序都是在同一台服务器（计算机）上，就使用 localhost 了。
- user:登录数据库的用户名，这里一般填写"root",还是要注意引号。当然，如果读者命名了别的用户名，数据库管理者提供了专有用户名，就更改为相应用户。但是，不同用户的权限可能不同，所以，在程序中，如果要操作数据库，还要注意所拥有的权限。在这里用 root，就放心了，什么权限都有啦。不过，这样做，在大型系统中是应该避免的。

- passwd: 上述 user 账户对应的登录 mysql 的密码。我在上面的例子中用的密码是"123123"。不要忘记引号。
- db: 就是刚刚通过 create 命令建立的数据库，我建立的数据库名字是"qiwsirttest"，还是要注意引号。看官如果建立的数据库名字不是这个，就写自己所建数据库名字。
- port: 一般情况，mysql 的默认端口是 3306，当 mysql 被安装到服务器之后，为了能够允许网络访问，服务器（计算机）要提供一个访问端口给它。
- charset: 这个设置，在很多教程中都不写，结果在真正进行数据存储的时候，发现有乱码。这里我将 qiwsirttest 这个数据库的编码设置为 utf-8 格式，这样就允许存入汉字而无乱码了。注意，在 mysql 设置中，utf-8 写成 utf8，没有中间的横线。但是在 Python 文件开头和其它地方设置编码格式的时候，要写成 utf-8。切记！

注：connect 中的 host、user、passwd 等可以不写，只有在写的时候按照 host、user、passwd、db（可以不写）、port 顺序写就可以，端口号 port=3306 还是不要省略的好，如果没有 db 在 port 前面，直接写 3306 会报错。

其实，关于 connect 的参数还不少，下面摘抄来自[mysqlDb 官方文档的内容](#)，把所有的参数都列出来，还有相关说明，请看官认真阅读。不过，上面几个是常用的，其它的看情况使用。

connect(parameters...)

Constructor for creating a connection to the database. Returns a Connection Object. Parameters are the same as for the MySQL C API. In addition, there are a few additional keywords that correspond to what you would pass mysql_options() before connecting. Note that some parameters must be specified as keyword arguments! The default value for each parameter is NULL or zero, as appropriate. Consult the MySQL documentation for more details. The important parameters are:

- host: name of host to connect to. Default: use the local host via a UNIX socket (where applicable).
- user: user to authenticate as. Default: current effective user.
- passwd: password to authenticate with. Default: no password.
- db: database to use. Default: no default database.
- port: TCP port of MySQL server. Default: standard port (3306).
- unix_socket: location of UNIX socket. Default: use default location or TCP for remote hosts.
- conv: type conversion dictionary. Default: a copy of MySQLdb.converters.conversions
- compress: Enable protocol compression. Default: no compression.

- connect_timeout: Abort if connect is not completed within given number of seconds. Default: no timeout (?)
- named_pipe: Use a named pipe (Windows). Default: don't.
- init_command: Initial command to issue to server upon connection. Default: Nothing.
- read_default_file: MySQL configuration file to read; see the MySQL documentation for mysql_options().
- read_default_group: Default group to read; see the MySQL documentation for mysql_options().
- cursorclass: cursor class that cursor() uses, unless overridden. Default: MySQLdb.cursors.Cursor. This must be a keyword parameter.
- use_unicode: If True, CHAR and VARCHAR and TEXT columns are returned as Unicode strings, using the configured character set. It is best to set the default encoding in the server configuration, or client configuration (read with read_default_file). If you change the character set after connecting (MySQL-4.1 and later), you'll need to put the correct character set name in connection.charset.

If False, text-like columns are returned as normal strings, but you can always write Unicode strings.

This must be a keyword parameter.

- charset: If present, the connection character set will be changed to this character set, if they are not equal. Support for changing the character set requires MySQL-4.1 and later server; if the server is too old, UnsupportedError will be raised. This option implies use_unicode=True, but you can override this with use_unicode=False, though you probably shouldn't.

If not present, the default character set is used.

This must be a keyword parameter.

- sql_mode: If present, the session SQL mode will be set to the given string. For more information on sql_mode, see the MySQL documentation. Only available for 4.1 and newer servers.

If not present, the session SQL mode will be unchanged.

This must be a keyword parameter.

- ssl: This parameter takes a dictionary or mapping, where the keys are parameter names used by the mysql_ssl_set MySQL C API call. If this is set, it initiates an SSL connection to the server; if there is no SSL support in the client, an exception is raised. This must be a keyword parameter.

已经完成了数据库的连接。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

MySQL 数据库 (2)

就数据库而言，连接之后就要对其操作。但是，目前那个名字叫做 `qiwsirtest` 的数据仅仅是空架子，没有什么可操作的，要操作它，就必须在里面建立“表”，什么是数据库的表呢？下面摘抄自维基百科对数据库表的简要解释，要想详细了解，需要看官在找一些有关数据库的教程和书籍来看看。

在关系数据库中，数据库表是一系列二维数组的集合，用来代表和储存数据对象之间的关系。它由纵向的列和横向的行组成，例如一个有关作者信息的名为 `authors` 的表中，每个列包含的是所有作者的某个特定类型的信息，比如“姓氏”，而每行则包含了某个特定作者的所有信息：姓、名、住址等等。

对于特定的数据库表，列的数目一般事先固定，各列之间可以由列名来识别。而行的数目可以随时、动态变化，每行通常都可以根据某个（或某几个）列中的数据来识别，称为候选键。

我打算在 `qiwsirtest` 中建立一个存储用户名、用户密码、用户邮箱的表，其结构用二维表格表现如下：

username	password	email
qiwsir	123123	qiwsir@gmail.com

特别说明，这里为了简化细节，突出重点，对密码不加密，直接明文保存，虽然这种方式是很不安全的。但是，有不少网站还都这么做的，这么做的目的是比较可恶的。就让我在这里，仅仅在这里可恶一次。

数据库表

因为直接操作数据部分，不是本教重点，但是关联到后面的操作，为了让读者在阅读上连贯，也快速地说明建立数据库表并输入内容。

```
mysql> use qiwsirtest;
Database changed
mysql> show tables;
Empty set (0.00 sec)
```

用 `show tables` 命令显示这个数据库中是否有数据表了。查询结果显示为空。

下面就用如下命令建立一个数据表，这个数据表的内容就是上面所说明的。

```
mysql> create table users(id int(2) not null primary key auto_increment,username varchar(40),password text,email text);
Query OK, 0 rows affected (0.12 sec)
```

建立的这个数据表名称是：`users`，其中包含上述字段，可以用下面的方式看一看这个数据表的结构。

```
mysql> show tables;
+-----+
| Tables_in_qiwsirtest |
+-----+
| users      |
+-----+
1 row in set (0.00 sec)
```

查询显示，在 qiwsirtest 这个数据库中，已经有一个表，它的名字是：users。

```
mysql> desc users;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra       |
+-----+-----+-----+-----+-----+
| id    | int(2) | NO  | PRI | NULL   | auto_increment |
| username | varchar(40) | YES |   | NULL   |             |
| password | text    | YES |   | NULL   |             |
| email   | text    | YES |   | NULL   |             |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

显示表 users 的结构：

特别提醒：上述所有字段设置仅为演示，在实际开发中，要根据具体情况来确定字段的属性。

如此就得到了一个空表。可以查询看看：

```
mysql> select * from users;
Empty set (0.01 sec)
```

向里面插入点信息，就只插入一条吧。

```
mysql> insert into users(username,password,email) values("qiwsir","123123","qiwsir@gmail.com");
Query OK, 1 row affected (0.05 sec)

mysql> select * from users;
+-----+-----+-----+
| id | username | password | email      |
+-----+-----+-----+
| 1  | qiwsir   | 123123   | qiwsir@gmail.com |
+-----+-----+-----+
1 row in set (0.00 sec)
```

这样就得到了一个有内容的数据库表。

Python 操作数据库

连接数据库，必须的。

```
>>> import MySQLdb
>>> conn = MySQLdb.connect(host="localhost",user="root",passwd="123123",db="qiwsirtest",charset="utf8")
```

Python 建立了与数据的连接，其实是建立了一个 `MySQLdb.connect()` 的实例对象，或者泛泛地称之为连接对象，Python 就是通过连接对象和数据库对话。这个对象常用的方法有：

- `commit()`: 如果数据库表进行了修改，提交保存当前的数据。当然，如果此用户没有权限就作罢了，什么也不会发生。
- `rollback()`: 如果有权限，就取消当前的操作，否则报错。
- `cursor([cursorclass])`: 返回连接的游标对象。通过游标执行 SQL 查询并检查结果。游标比连接支持更多的方法，而且可能在程序中更好用。
- `close()`: 关闭连接。此后，连接对象和游标都不再可用。

Python 和数据之间的连接建立起来之后，要操作数据库，就需要让 Python 对数据库执行 SQL 语句。Python 是通过游标执行 SQL 语句的。所以，连接建立之后，就要利用连接对象得到游标对象，方法如下：

```
>>> cur = conn.cursor()
```

此后，就可以利用游标对象的方法对数据库进行操作。那么还得了解游标对象的常用方法：

名称	描述
<code>close()</code>	关闭游标。之后游标不可用
<code>execute(query,[args])</code>	执行一条 SQL 语句，可以带参数
<code>executemany(query,pseq)</code>	对序列 pseq 中的每个参数执行 sql 语句
<code>fetchone()</code>	返回一条查询结果
<code>fetchall()</code>	返回所有查询结果
<code>fetchmany([size])</code>	返回 size 条结果
<code>nextset()</code>	移动到下一个结果
<code>scroll(value,mode='relative')</code>	移动游标到指定行，如果 mode='relative',则表示从当前所在行移动 value 条,如果 mode='absolute',则表示从结果集的第一行移动 value 条.

插入

例如，要在数据表 users 中插入一条记录，使得:username="Python",password="123456",email="Python@gmail.com"，这样做：

```
>>> cur.execute("insert into users (username,password,email) values (%s,%s,%s)",("Python","123456","Python@gmail.com"))
1L
```

没有报错，并且返回一个"1L"结果，说明有一行记录操作成功。不妨用"mysql>"交互方式查看一下：

```
mysql> select * from users;
+----+-----+-----+
| id | username | password | email      |
+----+-----+-----+
| 1 | qiwsir  | 123123  | qiwsir@gmail.com |
+----+-----+-----+
1 row in set (0.00 sec)
```

咦，奇怪呀。怎么没有看到增加的那一条呢？哪里错了？可是上面也没有报错呀。

特别注意，通过"cur.execute()"对数据库进行操作之后，没有报错，完全正确，但是不等于数据就已经提交到数据库中了，还必须要用到"MySQLdb.connect"的一个属性：commit()，将数据提交上去，也就是进行了"cur.execute()"操作，要将数据提交，必须执行：

```
>>> conn.commit()
```

再到"mysql>"中运行"select * from users"试一试：

```
mysql> select * from users;
+----+-----+-----+
| id | username | password | email      |
+----+-----+-----+
| 1 | qiwsir  | 123123  | qiwsir@gmail.com |
| 2 | python   | 123456  | python@gmail.com |
+----+-----+-----+
2 rows in set (0.00 sec)
```

果然如此。这就如同编写一个文本一样，将文字写到文本上，并不等于文字已经保留在文本文件中了，必须执行"CTRL-S"才能保存。也就是在通过 Python 操作数据库的时候，以"execute()"执行各种 sql 语句之后，要让已经执行的效果保存，必须运行连接对象的"commit()"方法。

再尝试一下插入多条的那个命令"executemany(query,args)"。

```
>>> cur.executemany("insert into users (username,password,email) values (%s,%s,%s)",(("google","111222","g@gmail.co
4L
>>> conn.commit()
```

到"mysql>"里面看结果：

```
mysql> select * from users;
+----+-----+-----+
| id | username | password | email
+----+-----+-----+
| 1 | qiwsir | 123123 | qiwsir@gmail.com |
| 2 | python | 123456 | python@gmail.com |
| 3 | google | 111222 | g@gmail.com |
| 4 | facebook | 222333 | f@face.book |
| 5 | github | 333444 | git@hub.com |
| 6 | docker | 444555 | doc@ker.com |
+----+-----+-----+
6 rows in set (0.00 sec)
```

成功插入了多条记录。在"executemany(query, pseq)"中，query 还是一条 sql 语句，但是 pseq 这时候是一个 tuple，这个 tuple 里面的元素也是 tuple，每个 tuple 分别对应 sql 语句中的字段列表。这句话其实被执行多次。只不过执行过程不显示给我们看罢了。

除了插入命令，其它对数据操作的命令都可用类似上面的方式，比如删除、修改等。

查询

如果要从数据库中查询数据，也用游标方法来操作了。

```
>>> cur.execute("select * from users")
7L
```

这说明从 users 表汇总查询出来了 7 条记录。但是，这似乎有点不友好，告诉我 7 条记录查出来了，但是在哪呢，如果在'mysql>'下操作查询命令，一下就把 7 条记录列出来了。怎么显示 Python 在这里的查询结果呢？

要用到游标对象的 fetchall()、fetchmany(size=None)、fetchone()、scroll(value, mode='relative')等方法。

```
>>> cur.execute("select * from users")
7L
>>> lines = cur.fetchall()
```

到这里，已经将查询到的记录赋值给变量 lines 了。如果要把它们显示出来，就要用到曾经学习过的循环语句了。

```
>>> for line in lines:
...     print line
...
(1L, u'qiwsir', u'123123', u'qiwsir@gmail.com')
(2L, u'python', u'123456', u'python@gmail.com')
(3L, u'google', u'111222', u'g@gmail.com')
(4L, u'facebook', u'222333', u'f@face.book')
(5L, u'github', u'333444', u'git@hub.com')
(6L, u'docker', u'444555', u'doc@ker.com')
(7L, u'\u8001\u95f5', u'9988', u'qiwsir@gmail.com')
```

很好。果然是逐条显示出来了。列位注意，第七条中的 `u'\u8001\u95f5'`, 这里是汉字，只不过由于我的 shell 不能显示罢了，不必惊慌，不必搭理它。

只想查出第一条，可以吗？当然可以！看下面的：

```
>>> cur.execute("select * from users where id=1")
1L
>>> line_first = cur.fetchone()  #只返回一条
>>> print line_first
(1L, u'qiwsir', u'123123', u'qiwsir@gmail.com')
```

为了对上述过程了解深入，做下面实验：

```
>>> cur.execute("select * from users")
7L
>>> print cur.fetchall()
((1L, u'qiwsir', u'123123', u'qiwsir@gmail.com'), (2L, u'python', u'123456', u'python@gmail.com'), (3L, u'google', u'111222', u'
```

原来，用 `cur.execute()` 从数据库查询出来的东西，被“保存在了 `cur` 所能找到的某个地方”，要找出这些被保存的东西，需要用 `cur.fetchall()`（或者 `fetchone` 等），并且找出来之后，做为对象存在。从上面的实验探讨发现，被保存的对象是一个 tuple 中，里面的每个元素，都是一个一个的 tuple。因此，用 `for` 循环就可以一个一个拿出来了。

接着看，还有神奇的呢。

接着上面的操作，再打印一遍

```
>>> print cur.fetchall()
()
```

晕了！怎么什么是空？不是说做为对象已经存在了内存中了吗？难道这个内存中的对象是一次有效吗？

不要着急。

通过游标找出来的对象，在读取的时候有一个特点，就是那个游标会移动。在第一次操作了 print cur.fetchall() 后，因为是将所有的都打印出来，游标就从第一条移动到最后一条。当 print 结束之后，游标已经在最后一条的后面了。接下来如果再次打印，就空了，最后一条后面没有东西了。

下面还要实验，检验上面所说：

```
>>> cur.execute('select * from users')
7L
>>> print cur.fetchone()
(1L, u'qiwsir', u'123123', u'qiwsir@gmail.com')
>>> print cur.fetchone()
(2L, u'python', u'123456', u'python@gmail.com')
>>> print cur.fetchone()
(3L, u'google', u'111222', u'g@gmail.com')
```

这次我不一次全部打印出来了，而是一次打印一条，看官可以从结果中看出来，果然那个游标在一条一条向下移动呢。注意，我在这次实验中，是重新运行了查询语句。

那么，既然在操作存储在内存中的对象时候，游标会移动，能不能让游标向上移动，或者移动到指定位置呢？这就是那个 scroll()

```
>>> cur.scroll(1)
>>> print cur.fetchone()
(5L, u'github', u'333444', u'git@hub.com')
>>> cur.scroll(-2)
>>> print cur.fetchone()
(4L, u'facebook', u'222333', u'f@face.book')
```

果然，这个函数能够移动游标，不过请仔细观察，上面的方式是让游标相对与当前位置向上或者向下移动。即：

cur.scroll(n)，或者，cur.scroll(n,"relative")：意思是相对当前位置向上或者向下移动，n 为正数，表示向下（向前），n 为负数，表示向上（向后）

还有一种方式，可以实现“绝对”移动，不是“相对”移动：增加一个参数"absolute"

特别提醒看官注意的是，在 Python 中，序列对象是的顺序是从 0 开始的。

```
>>> cur.scroll(2,"absolute") #回到序号是 2,但指向第三条
>>> print cur.fetchone()    #打印，果然是
(3L, u'google', u'111222', u'g@gmail.com')

>>> cur.scroll(1,"absolute")
>>> print cur.fetchone()
(2L, u'python', u'123456', u'python@gmail.com')
```

```
>>> cur.scroll(0,"absolute") #回到序号是 0,即指向 tuple 的第一条
>>> print cur.fetchone()
(1L, u'qiwsir', u'123123', u'qiwsir@gmail.com')
```

至此，已经熟悉了 `cur.fetchall()` 和 `cur.fetchone()` 以及 `cur.scroll()` 几个方法，还有另外一个，接这上边的操作，也就是游标在序号是 1 的位置，指向了 tuple 的第二条

```
>>> cur.fetchmany(3)
((2L, u'Python', u'123456', u'python@gmail.com'), (3L, u'google', u'111222', u'g@gmail.com'), (4L, u'facebook', u'222333', u'
```

上面这个操作，就是实现了从当前位置（游标指向 tuple 的序号为 1 的位置，即第二条记录）开始，含当前位置，向下列出 3 条记录。

读取数据，好像有点啰嗦呀。细细琢磨，还是有道理的。你觉得呢？

不过，Python 总是能够为我们着想的，在连接对象的游标方法中提供了一个参数，可以实现将读取到的数据变成字典形式，这样就提供了另外一种读取方式了。

```
>>> cur = conn.cursor(cursorclass=MySQLdb.cursors.DictCursor)
>>> cur.execute("select * from users")
7L
>>> cur.fetchall()
[{'username': u'qiwsir', 'password': u'123123', 'id': 1L, 'email': u'qiwsir@gmail.com'}, {'username': u'my python', 'password': u'
```

这样，在元组里面的元素就是一个一个字典：

```
>>> cur.scroll(0,"absolute")
>>> for line in cur.fetchall():
...     print line["username"]
...
qiwsir
my python
google
facebook
github
docker
老齐
```

根据字典对象的特点来读取了“键-值”。

更新数据

经过前面的操作，这个就比较简单了，不过需要提醒的是，如果更新完毕，和插入数据一样，都需要 commit() 来提交保存。

```
>>> cur.execute("update users set username=%s where id=2",("mypython"))
1L
>>> cur.execute("select * from users where id=2")
1L
>>> cur.fetchone()
(2L, u'mypython', u'123456', u'python@gmail.com')
```

从操作中看出来了，已经将数据库中第二条的用户名修改为 myPython 了，用的就是 update 语句。

不过，要真的实现在数据库中更新，还要运行：

```
>>> conn.commit()
```

这就大事完吉了。

应该还有个小尾巴，那就是当你操作数据完毕，不要忘记关门：

```
>>> cur.close()
>>> conn.close()
```

门锁好了，放心离开。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

mongodb 数据库 (1)

MongoDB 开始火了，这是时代发展的需要。为此，本教程也要涉及到如何用 Python 来操作 mongodb。考虑到读者对这种数据库可能比 mysql 之类的更陌生，所以，要用多一点的篇幅稍作介绍，当然，更完备的内容还是要去阅读专业的 mongodb 书籍。

mongodb 是属于 NoSql 的。

NoSql，全称是 Not Only Sql,指的是非关系型的数据库。它是为了大规模 web 应用而生的，其特征诸如模式自由、支持简易复制、简单的 API、大容量数据等等。

MongoDB 是其一，选择它，主要是因为我喜欢，否则我不会列入我的教程。数说它的特点，可能是：

- 面向文档存储
- 对任何属性可索引
- 复制和高可用性
- 自动分片
- 丰富的查询
- 快速就地更新

也许还能列出更多，基于它的特点，擅长领域就在于：

- 大数据（太时髦了！以下可以都不看，就要用它了。）
- 内容管理和交付
- 移动和社交基础设施
- 用户数据管理
- 数据平台

安装 mongodb

先演示在 ubuntu 系统中的安装过程：

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | sudo tee /etc/apt/sources.list.d/mongodb.list
sudo apt-get update
sudo apt-get install mongodb-10gen
```

如此就安装完毕。上述安装流程来自：[Install MongoDB](#)

如果你用的是其它操作系统，可以到官方网站下载安装程序：<http://www.mongodb.org/downloads>，能满足各种操作系统。

Download and Run MongoDB Yourself

[Current Release](#) [Previous Releases](#) [Development Releases](#)

Production Release (3.0.2)

4/9/2015 [Release Notes](#) [Changelog](#)

Download Source: [tgz](#) | [zip](#)

 Windows

 Linux

 Mac OS X

Solaris

VERSION:

Windows 64-bit 2008 R2+

难免在安装过程中遇到问题，推荐几个资料，供参考：

[window 平台安装 MongoDB](#)

[NoSQL 之【MongoDB】学习（一）：安装说明](#)

[MongoDB 生产环境的安装与配置\(Ubuntu\)](#)

[在 Ubuntu 中安装 MongoDB](#)

[在 Ubuntu 下进行 MongoDB 安装步骤](#)

启动 mongodb

安装完毕，就可以启动数据库。因为本教程不是专门讲数据库，所以，这里不设计数据库的详细讲解，请读者参考有关资料。下面只是建立一个简单的库，并且说明 mongodb 的基本要点，目的在于为后面用 Python 来操作它做个铺垫。

执行 `mongo` 启动 shell，显示的也是 `>`，有点类似 mysql 的状态。在 shell 中，可以实现与数据库的交互操作。

在 shell 中，有一个全局变量 `db`，使用哪个数据库，那个数据库就会被复制给这个全局变量 `db`，如果那个数据库不存在，就会新建。

```
> use mydb
switched to db mydb
> db
mydb
```

除非向这个数据库中增加实质性的内容，否则它是看不到的。

```
> show dbs;
local 0.03125GB
```

向这个数据库增加点东西。mongodb 的基本单元是文档，所谓文档，就类似与 Python 中的字典，以键值对的方式保存数据。

```
> book = {"title": "from beginner to master", "author": "qiwsir", "lang": "python"}
{
    "title" : "from beginner to master",
    "author" : "qiwsir",
    "lang" : "python"
}
> db.books.insert(book)
> db.books.find()
{ "_id" : ObjectId("554f0e3cf579bc0767db9edf"), "title" : "from beginner to master", "author" : "qiwsir", "lang" : "Python" }
```

db 指向了数据库 mydb，books 是这个数据库里面的一个集合（类似 mysql 里面的表），向集合 books 里面插入了一个文档（文档对应 mysql 里面的记录）。“数据库、集合、文档”构成了 mongodb 数据库。

从上面操作，还发现一个有意思的地方，并没有类似 create 之类的命令，用到数据库，就通过 `use xxx`，如果不存在就建立；用到集合，就通过 `db.xxx` 来使用，如果没有就建立。可以总结为“随用随取随建立”。是不是简单的有点出人意料。

```
> show dbs
local 0.03125GB
mydb 0.0625GB
```

当有了充实内容之后，也看到刚才用到的数据库 mydb 了。

在 mongodb 的 shell 中，可以对数据进行“增删改查”等操作。但是，我们的目的是用 Python 来操作，所以，还是把力气放在后面用。

安装 Pymongo

要用 Python 来驱动 mongodb，必须要安装驱动模块，即 Pymongo，这跟操作 mysql 类似。安装方法，我最推荐如下：

```
$ sudo pip install Pymongo
```

如果顺利，就会看到最后的提示：

```
Successfully installed Pymongo
Cleaning up...
```

如果不选择版本，安装的应该是最新版本的，我在本教程测试的时候，安装的是：

```
>>> import Pymongo
>>> pymongo.version
'3.0.1'
```

这个版本在后面给我挖了一个坑。如果读者要指定版本，比如安装 2.8 版本的，可以：

```
$ sudo pip install Pymongo==2.8
```

如果用这个版本，我后面遇到的坑能够避免。

安装好之后，进入到 Python 的交互模式里面：

```
>>> import Pymongo
```

说明模块没有问题。

连接 mongodb

既然 Python 驱动 mongodb 的模块 Pymongo 业已安装完毕，接下来就是连接，也就是建立连接对象。

```
>>> pymongo.Connection("localhost",27017)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'Connection'
```

报错！我在去年做的项目中，就是这样做的，并且网上查看很多教程都是这么连接。

所以，读者如果用的是旧版本的 Pymongo，比如 2.8，仍然可以使用上面的连接方法，如果是像我一样，是用的新的（我安装时没有选版本），就得注意这个问题了。

经验主义害死人。必须看看下面有哪些方法可以用：

```
>>> dir(pymongo)
['ALL', 'ASCENDING', 'CursorType', 'DESCENDING', 'DeleteMany', 'DeleteOne', 'GEO2D', 'GEOHAYSTACK', 'GEOSPHERE', 'IndexType', 'MAX', 'MIN', 'REVERSE', 'SortType']
```

瞪大我的那双浑浊迷茫布满血丝渴望惊喜的眼睛，透过近视镜的玻璃片，怎么也找不到 Connection() 这个方法。原来，刚刚安装的 Pymongo 变了，“他变了”。

不过，我发现了它：MongoClient()

```
>>> client = pymongo.MongoClient("localhost", 27017)
```

很好。Python 已经和 mongodb 建立了连接。

刚才已经建立了一个数据库 mydb，并且在这个库里面有一个集合 books，于是：

```
>>> db = client.mydb
```

或者

```
>>> db = client['mydb']
```

获得数据库 mydb，并赋值给变量 db（这个变量不是 mongodb 的 shell 中的那个 db，此处的 db 就是 Python 中一个寻常的变量）。

```
>>> db.collection_names()
[u'system.indexes', u'books']
```

查看集合，发现了我们已经建立好的那个 books，于是在获取这个集合，并赋值给一个变量 books：

```
>>> books = db["books"]
```

或者

```
>>> books = db.books
```

接下来，就可以操作这个集合中的具体内容了。

编辑

刚刚的 books 所引用的是一个 mongodb 的集合对象，它就跟前面学习过的其它对象一样，有一些方法供我们来驱使。

```
>>> type(books)
<class 'pymongo.collection.Collection'>

>>> dir(books)
['_BaseObject__codec_options', '_BaseObject__read_preference', '_BaseObject__write_concern', '_Collection__create_index', '_Collection__create_indexes', '_Collection__drop', '_Collection__ensure_index', '_Collection__get_index', '_Collection__get_indexes', '_Collection__index_information', '_Collection__max_size', '_Collection__max_write_size', '_Collection__remove', '_Collection__remove_one', '_Collection__size', '_Collection__with_options']
```

这么多方法不会一一介绍，只是按照“增删改查”的常用功能，介绍几种。读者可以使用 help() 去查看每一种方法的使用说明。

```
>>> books.find_one()
{u'lang': u'Python', u'_id': ObjectId('554f0e3cf579bc0767db9edf'), u'author': u'qiwsir', u'title': u'from beginner to master'}
```

提醒读者注意的是，如果你熟悉了 mongodb 的 shell 中的命令，跟 Pymongo 中的方法稍有差别，比如刚才这个，在 mongodb 的 shell 中是这样子的：

```
> db.books.findOne()
{
    "_id" : ObjectId("554f0e3cf579bc0767db9edf"),
    "title" : "from beginner to master",
    "author" : "qiwsir",
    "lang" : "python"
}
```

请注意区分。

目前在集合 books 中，有一个文档，还想再增加，于是插入一条：

新增和查询

```
>>> b2 = {"title": "physics", "author": "Newton", "lang": "english"}
>>> books.insert(b2)
ObjectId('554f28f465db941152e6df8b')
```

成功地向集合中增加了一个文档。得看看结果（我们就是充满好奇心的小孩子，我记得女儿小时候，每个给她照相，拍了一张，她总要看一看。现在我们似乎也是这样，如果不看看，总觉得不放心），看看就是一种查询。

```
>>> books.find().count()
2
```

这是查看当前集合有多少个文档的方式，返回值为 2，则说明有两条文档了。还是要看看内容。

```
>>> books.find_one()
{u'lang': u'python', u'_id': ObjectId('554f0e3cf579bc0767db9edf'), u'author': u'qiwsir', u'title': u'from beginner to master'}
```

这个命令就不行了，因为它只返回第一条。必须要：

```
>>> for i in books.find():
...     print i
...
{u'lang': u'Python', u'_id': ObjectId('554f0e3cf579bc0767db9edf'), u'author': u'qiwsir', u'title': u'from beginner to master'}
{u'lang': u'english', u'title': u'physics', u'_id': ObjectId('554f28f465db941152e6df8b'), u'author': u'Newton'}
```

在 books 引用的对象中有 find() 方法，它返回的是一个可迭代对象，包含着集合中所有的文档。

由于文档是键值对，也不一定每条文档都要结构一样，比如，也可以插入这样的文档进入集合。

```
>>> books.insert({"name":"Hertz"})
ObjectId('554f2b4565db941152e6df8c')
>>> for i in books.find():
...     print i
...
{u'lang': u'Python', u'_id': ObjectId('554f0e3cf579bc0767db9edf'), u'author': u'qiwsir', u'title': u'from beginner to master'}
{u'lang': u'english', u'title': u'physics', u'_id': ObjectId('554f28f465db941152e6df8b'), u'author': u'Newton'}
{u'_id': ObjectId('554f2b4565db941152e6df8c'), u'name': u'Hertz'}
```

如果有多个文档，想一下子插入到集合中（在 mysql 中，可以实现多条数据用一条命令插入到表里面，还记得吗？忘了看[上一节](#)），可以这么做：

```
>>> n1 = {"title":"java", "name":"Bush"}
>>> n2 = {"title":"fortran", "name":"John Warner Backus"}
>>> n3 = {"title":"lisp", "name":"John McCarthy"}
>>> n = [n1, n2, n3]
>>> n
[{'name': 'Bush', 'title': 'java'}, {'name': 'John Warner Backus', 'title': 'fortran'}, {'name': 'John McCarthy', 'title': 'lisp'}]
>>> books.insert(n)
[ObjectId('554f30be65db941152e6df8d'), ObjectId('554f30be65db941152e6df8e'), ObjectId('554f30be65db941152e6df8f')]
```

这样就完成了所谓的批量插入，查看一下文档条数：

```
>>> books.find().count()
6
```

但是，要提醒读者，批量插入的文档大小是有限制的，网上有人说不要超过 20 万条，有人说不要超过 16M B，我没有测试过。在一般情况下，或许达不到上线，如果遇到极端情况，就请读者在使用时多注意了。

如果要查询，除了通过循环之外，能不能按照某个条件查呢？比如查找 'name'='Bush' 的文档：

```
>>> books.find_one({"name":"Bush"})
{u'_id': ObjectId('554f30be65db941152e6df8d'), u'name': u'Bush', u'title': u'java'}
```

对于查询结果，还可以进行排序：

```
>>> for i in books.find().sort("title", pymongo.ASCENDING):
...     print i
...
{u'_id': ObjectId('554f2b4565db941152e6df8c'), u'name': u'Hertz'}
{u'_id': ObjectId('554f30be65db941152e6df8e'), u'name': u'John Warner Backus', u'title': u'fortran'}
```

```
{u'lang': u'python', u'_id': ObjectId('554f0e3cf579bc0767db9edf'), u'author': u'qiwsir', u'title': u'from beginner to master'}
{u'_id': ObjectId('554f30be65db941152e6df8d'), u'name': u'Bush', u'title': u'java'}
{u'_id': ObjectId('554f30be65db941152e6df8f'), u'name': u'John McCarthy', u'title': u'lisp'}
{u'lang': u'english', u'title': u'physics', u'_id': ObjectId('554f28f465db941152e6df8b'), u'author': u'Newton'}
```

这是按照"title"的值的升序排列的，注意 sort() 中的第二个参数，意思是升序排列。如果按照降序，就需要将参数修改为 `Pymongo.DESCENDING`，也可以指定多个排序键。

```
>>> for i in books.find().sort([("name",pymongo.ASCENDING),("name",pymongo.DESCENDING)]):
...     print i
...
{u'_id': ObjectId('554f30be65db941152e6df8e'), u'name': u'John Warner Backus', u'title': u'fortran'}
{u'_id': ObjectId('554f30be65db941152e6df8f'), u'name': u'John McCarthy', u'title': u'lisp'}
{u'_id': ObjectId('554f2b4565db941152e6df8c'), u'name': u'Hertz'}
{u'_id': ObjectId('554f30be65db941152e6df8d'), u'name': u'Bush', u'title': u'java'}
{u'lang': u'python', u'_id': ObjectId('554f0e3cf579bc0767db9edf'), u'author': u'qiwsir', u'title': u'from beginner to master'}
{u'lang': u'english', u'title': u'physics', u'_id': ObjectId('554f28f465db941152e6df8b'), u'author': u'Newton'}
```

读者如果看到这里，请务必注意一个事情，那就是 mongodb 中的每个文档，本质上都是“键值对”的类字典结构。这种结构，一经 Python 读出来，就可以用字典中的各种方法来操作。与此类似的还有一个名为 json 的东西，可以阅读本教程第贰季进阶的第陆章模块中的[《标准库(8)》(http://api.mongodb.org/Python/current/api/bson/json_util.html)中的模块使用说明。

更新

对于已有数据，进行更新，是数据库中常用的操作。比如，要更新 name 为 Hertz 那个文档：

```
>>> books.update({"name":"Hertz"}, {"$set": {"title":"new physics", "author":"Hertz"}})
{u'updatedExisting': True, u'connectionId': 4, u'ok': 1.0, u'err': None, u'n': 1}
>>> books.find_one({"author":"Hertz"})
{u'title': u'new physics', u'_id': ObjectId('554f2b4565db941152e6df8c'), u'name': u'Hertz', u'author': u'Hertz'}
```

在更新的时候，用了一个 `$set` 修改器，它可以用来指定键值，如果键不存在，就会创建。

关于修改器，不仅仅是这一个，还有别的呢。

修改器	描述
\$set	用来指定一个键的值。如果不存在则创建它
\$unset	完全删除某个键
\$inc	增加已有键的值，不存在则创建（只能用于增加整数、长整数、双精度浮点数）
\$push	数组修改器只能操作值为数组，存在 key 在值末尾增加一个元素，不存在则创建一个数组

删除

删除可以用 `remove()` 方法：

```
>>> books.remove({"name":"Bush"})
{u'connectionId': 4, u'ok': 1.0, u'err': None, u'n': 1}
>>> books.find_one({"name":"Bush"})
>>>
```

这是将那个文档全部删除。当然，也可以根据 mongodb 的语法规则，写个条件，按照条件删除。

索引

索引的目的是为了让查询速度更快，当然，在具体的项目开发中，要视情况而定是否建立索引。因为建立索引也是有代价的。

```
>>> books.create_index([("title", pymongo.DESCENDING),])
u'title_-1'
```

我这里仅仅是对 Pymongo 模块做了一个非常简单的介绍，在实际使用过程中，上面知识是很有限的，所以需要读者根据具体应用场景再结合 mongodb 的有关知识去尝试新的语句。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

SQLite 数据库

SQLite 是一个小型的关系型数据库，它最大的特点在于不需要服务器、零配置。在前面的两个服务器，不管是 MySQL 还是 MongoDB，都需要“安装”，安装之后，它运行起来，其实是已经有一个相应的服务器在跑着呢。而 SQLite 不需要这样，首先 Python 已经将相应的驱动模块作为标准库一部分了，只要安装了 Python，就可以使用；另外，它也不需要服务器，可以类似操作文件那样来操作 SQLite 数据库文件。还有一点也不错，SQLite 源代码不受版权限制。

SQLite 也是一个关系型数据库，所以 SQL 语句，都可以在里面使用。

跟操作 mysql 数据库类似，对于 SQLite 数据库，也要通过以下几步：

- 建立连接对象
- 连接对象方法：建立游标对象
- 游标对象方法：执行 sql 语句

建立连接对象

由于 SQLite 数据库的驱动已经在 Python 里面了，所以，只要引用就可以直接使用

```
>>> import sqlite3  
>>> conn = sqlite3.connect("23301.db")
```

这样就得到了连接对象，是不是比 mysql 连接要简化了很多呢。在 `sqlite3.connect("23301.db")` 语句中，如果已经有了那个数据库，就连接上它；如果没有，就新建一个。注意，这里的路径可以随意指定的。

不妨到目录中看一看，是否存在了刚才建立的数据库文件。

```
/2code$ ls 23301.db  
23301.db
```

果然有了一个文件。

连接对象建立起来之后，就要使用连接对象的方法继续工作了。

```
>>> dir(conn)  
['DataError', 'DatabaseError', 'Error', 'IntegrityError', 'InterfaceError', 'InternalError', 'NotSupportedError', 'OperationalError']
```

游标对象

这步跟 mysql 也类似，要建立游标对象。

```
>>> cur = conn.cursor()
```

接下来对数据库内容的操作，都是用游标对象方法来实现了：

```
>>> dir(cur)
```

```
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__iter__', '__new__', '__red
```

是不是看到熟悉的名称了： `close()`, `execute()`, `executemany()`, `fetchall()`

创建数据库表

在 mysql 中，我们演示的是利用 mysql 的 shell 来创建的表。其实，当然可以使用 sql 语句，在 Python 中实现这个功能。这里对 sqlite 数据库，就如此操作一番。

```
>>> create_table = "create table books (title text, author text, lang text)"
>>> cur.execute(create_table)
<sqlite3.Cursor object at 0xb73ed5a0>
```

这样就在数据库 23301.db 中建立了一个表 books。对这个表可以增加数据了：

```
>>> cur.execute('insert into books values ("from beginner to master", "laoqi", "python")')
<sqlite3.Cursor object at 0xb73ed5a0>
```

为了保证数据能够保存，还要（这是多么熟悉的操作流程和命令呀）：

```
>>> conn.commit()
>>> cur.close()
>>> conn.close()
```

支持，刚才建立的那个数据库中，已经有了一个表 books，表中已经有了一条记录。

整个流程都不陌生。

查询

存进去了，总要看看，这算强迫症吗？

```
>>> conn = sqlite3.connect("23301.db")
>>> cur = conn.cursor()
>>> cur.execute('select * from books')
<sqlite3.Cursor object at 0xb73edea0>
>>> print cur.fetchall()
[(u'from beginner to master', u'laoqi', u'python')]
```

批量插入

多增加点内容，以便于做别的操作：

```
>>> books = [("first book","first","c"), ("second book","second","c"), ("third book","second","python")]
```

这回来一个批量插入

```
>>> cur.executemany('insert into books values (?,?,?)', books)
<sqlite3.Cursor object at 0xb73edea0>
>>> conn.commit()
```

用循环语句打印一下查询结果：

```
>>> rows = cur.execute('select * from books')
>>> for row in rows:
...     print row
...
(u'from beginner to master', u'laoqi', u'python')
(u'first book', u'first', u'c')
(u'second book', u'second', u'c')
(u'third book', u'second', u'python')
```

更新

正如前面所说，在 `cur.execute()` 中，你可以写 SQL 语句，来操作数据库。

```
>>> cur.execute("update books set title='physics' where author='first'")
<sqlite3.Cursor object at 0xb73edea0>
>>> conn.commit()
```

按照条件查处来看一看：

```
>>> cur.execute("select * from books where author='first'")
<sqlite3.Cursor object at 0xb73edea0>
>>> cur.fetchone()
(u'physics', u'first', u'c')
```

删除

在 sql 语句中，这也是常用的。

```
>>> cur.execute("delete from books where author='second'")
<sqlite3.Cursor object at 0xb73edea0>
>>> conn.commit()

>>> cur.execute("select * from books")
<sqlite3.Cursor object at 0xb73edea0>
>>> cur.fetchall()
[(u'from beginner to master', u'laoqi', u'python'), (u'physics', u'first', u'c')]
```

不要忘记，在你完成对数据库的操作是，一定要关门才能走人：

```
>>> cur.close()
>>> conn.close()
```

作为基本知识，已经介绍差不多了。当然，在实践的编程中，或许会遇到问题，就请读者多参考官方文档：<http://docs.Python.org/2/library/sqlite3.html>

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

电子表格

一提到电子表格，可能立刻想到的是 excel。殊不知，电子表格，还是“历史悠久”的呢，比 word 要长久多了。根据维基百科的记载整理一个简史：

VisiCalc 是第一个电子表格程序，用于苹果 II 型电脑。由丹·布李克林（Dan Bricklin）和鮑伯·法兰克斯顿（Bob Frankston）发展而成，1979 年 10 月跟著苹果二号电脑推出，成为苹果二号电脑上的「杀手应用软件」。

接下来是 Lotus 1-2-3，由 Lotus Software（美国莲花软件公司）于 1983 年起所推出的电子试算表软件，在 DOS 时期广为个人电脑使用者所使用，是一套杀手级应用软件。也是世界上第一个销售超过 100 万套的软件。

然后微软也开始做电子表格，早在 1982 年，它推出了它的第一款电子制表软件——Multiplan，並在 CP/M 系统上大获成功，但在 MS-DOS 系统上，Multiplan 败给了 Lotus 1-2-3。

1985 年，微软推出第一款 Excel，但它只用于 Mac 系统；直到 1987 年 11 月，微软的第一款适用于 Windows 系统的 Excel 才诞生，不过，它一出来，就与 Windows 系统直接捆绑，由于此后 windows 大行其道，并且 Lotus 1-2-3 迟迟不能适用于 Windows 系统，到了 1988 年，Excel 的销量超过了 1-2-3。

此后就是微软的天下了，Excel 后来又并入了 Office 里面，成为了 Microsoft Office Excel。

尽管 Excel 已经发展了很多代，提供了大量的用户界面特性，但它仍然保留了第一款电子制表软件 VisiCalc 的特性：行、列组成单元格，数据、与数据相关的公式或者对其他单元格的绝对引用保存在单元格中。

由于微软独霸天下，Lotus 1-2-3 已经淡出了人们的视线，甚至于误认为历史就是从微软开始的。

其实，除了微软的电子表格，在 Linux 系统中也有很好的电子表格，google 也提供了不错的在线电子表格（可惜某国内不能正常访问）。

从历史到现在，电子表格都很广泛的用途。所以，Python 也要操作一番电子表格，因为有的数据，或许就是存在电子表格中。

openpyxl

openpyxl 模块是解决 Microsoft Excel 2007/2010 之类版本中扩展名是 Excel 2010 xlsx/xlsm/xltx/xltm 的文件的读写的第三方库。（差点上不来气，这句话太长了。）

安装

安装第三方库，当然用法力无边的 pip install

```
$ sudo pip install openpyxl
```

如果最终看到下面的提示，恭喜你，安装成功。

```
Successfully installed openpyxl jdcal  
Cleaning up...
```

workbook 和 sheet

第一步，当然是要引入模块，用下面的方式：

```
>>> from openpyxl import Workbook
```

接下来就用 `Workbook()` 类里面的方法展开工作：

```
>>> wb = Workbook()
```

请回忆 Excel 文件，如果想不起来，就打开 Excel，我们第一眼看到的是一个称之为工作簿(workbook)的东西，里面有几个 sheet，默认是三个，当然可以随意增删。默认又使用第一个 sheet。

```
>>> ws = wb.active
```

每个工作簿中，至少要有一个 sheet，通过这条指令，就在当前工作簿中建立了一个 sheet，并且它是当前正在使用的。

还可以在这个 sheet 后面追加：

```
>>> ws1 = wb.create_sheet()
```

甚至，还可以加塞：

```
>>> ws2 = wb.create_sheet(1)
```

排在了第二个位置。

在 Excel 文件中一样，创建了 sheet 之后，默认都是以"Sheet1"、"Sheet2"样子来命名的，然后我们可以给其重新命名。在这里，依然可以这么做。

```
>>> ws.title = "Python"
```

ws所引用的sheet对象名字就是"Python"了。

此时，可以使用下面的方式从工作簿对象中得到 sheet

```
>>> ws01 = wb['Python'] #sheet 和工作簿的关系，类似键值对的关系
>>> ws is ws01
True
```

或者用这种方式

```
>>> ws02 = wb.get_sheet_by_name("Python") #这个方法名字也太直接了，方法的参数就是 sheet 名字
>>> ws is ws02
True
```

整理一下到目前为止我们已经完成的工作：建立了工作簿(wb)，还有三个 sheet。还是显示一下比较好：

```
>>> print wb.get_sheet_names()
['Python', 'Sheet2', 'Sheet1']
```

Sheet2 这个 sheet 之所以排在了第二位，是因为在建立的时候，用了一个加塞的方法。这跟 Excel 中差不多，如果 sheet 命名了，就按照那个名字显示，否则就默认为名字是"Sheet1"形状的（注意，第一个字母大写）。

也可以用循环语句，把所有的 sheet 名字打印出来。

```
>>> for sh in wb:
...     print sh.title
...
Python
Sheet2
Sheet1
```

如果读者去 `dir(wb)` 工作簿对象的属性和方法，会发现它具有迭代的特征 `__iter__` 方法。说明，工作簿是可迭代的。

cell

为了能够清楚理解填数据的过程，将电子表中约定的名称以下图方式说明：

	B4		fx	this is a cell		
	A	B	C	D	E	F
1						
2						
3						
4		this is a cell It is B4				
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						

对于 sheet，其中的 cell 是它的下级单位。所以，要得到某个 cell，可以这样：

```
b4 = ws['B4']
```

如果 B4 这个 cell 已经有了，用这种方法就是将它的值赋给了变量 b4；如果 sheet 中没有这个 cell，那么就创建这个 cell 对象。

请读者注意，当我们打开 Excel，默认已经画好了好多 cell。但是，在 Python 操作的电子表格中，不会默认画好那样一个表格，一切都要创建之后才有。所以，如果按照前面的操作流程，上面就是创建了 B4 这个 cell，并且把它作为一个对象被 b4 变量引用。

如果要给 B4 添加数据，可以这么做：

```
>>> ws['B4'] = 4444
```

因为 b4 引用了一个 cell 对象，所以可以利用这个对象的属性来查看其值：

```
>>> b4.value
4444
```

要获得（或者建立并获得）某个 cell 对象，还可以使用下面方法：

```
>>> a1 = ws.cell("A1")
```

或者：

```
>>> a2 = ws.cell(row = 2, column = 1)
```

刚才已经提到，在建立了 sheet 之后，内存中的它并没有 cell，需要程序去建立。上面都是一个一个地建立，能不能一下建立多个呢？比如要类似下面的：

```
|A1|B1|C1| |A2|B2|C2| |A3|B3|C3|
```

就可以如同切片那样来操作：

```
>>> cells = ws["A1":"C3"]
```

可以用下面方法看看创建结果：

```
>>> tuple(ws.iter_rows("A1:C3"))
((<Cell python.A1>, <Cell Python.B1>, <Cell Python.C1>),
 (<Cell python.A2>, <Cell Python.B2>, <Cell Python.C2>),
 (<Cell python.A3>, <Cell Python.B3>, <Cell Python.C3>))
```

这是按照横向顺序数过来的，即 A1–B1–C1，然后下一横行。还可以用下面的循环方法，一个一个地读到每个 cell 对象：

```
>>> for row in ws.iter_rows("A1:C3"):
...     for cell in row:
...         print cell
...
<Cell Python.A1>
<Cell Python.B1>
<Cell Python.C1>
<Cell Python.A2>
<Cell Python.B2>
<Cell Python.C2>
<Cell Python.A3>
<Cell Python.B3>
<Cell Python.C3>
```

也可以用 sheet 对象的 `rows` 属性，得到按照横向顺序依次排列的 cell 对象（注意观察结果，因为没有进行范围限制，所以是目前 sheet 中所有的 cell，前面已经建立到第四行了 B4，所以，要比上面的操作多一个 `row`）：

```
>>> ws.rows
((<Cell python.A1>, <Cell python.B1>, <Cell python.C1>),
 (<Cell python.A2>, <Cell python.B2>, <Cell python.C2>),
 (<Cell python.A3>, <Cell python.B3>, <Cell python.C3>),
 (<Cell python.A4>, <Cell python.B4>, <Cell python.C4>))
```

用 sheet 对象的 `columns` 属性，得到的是按照纵向顺序排列的 cell 对象（注意观察结果）：

```
>>> ws.columns
((<Cell python.A1>, <Cell python.A2>, <Cell python.A3>, <Cell python.A4>),
 (<Cell python.B1>, <Cell python.B2>, <Cell python.B3>, <Cell python.B4>),
 (<Cell python.C1>, <Cell python.C2>, <Cell python.C3>, <Cell python.C4>))
```

不管用那种方法，只要得到了 cell 对象，接下来就可以依次赋值了。比如要将上面的表格中，依次填写上 1,2,3,...

```
>>> i = 1
>>> for cell in ws.rows:
...     cell.value = i
...     i += 1
```

... Traceback (most recent call last): File "", line 2, in AttributeError: 'tuple' object has no attribute 'valu
e'

报错了。什么错误。关键就是没有注意观察上面的结果。tuple 里面是以 tuple 为元素，再里面才是 cell 对象。所以，必须要“时时警醒”，常常谨慎。

```
>>> for row in ws.rows:
...     for cell in row:
...         cell.value = i
...         i += 1
... 
```

如此，就给每个 cell 添加了数据。查看一下，不过要换一个属性：

```
>>> for col in ws.columns:
...     for cell in col:
...         print cell.value
...
1
4
7
10
2
5
8
11
3
6
9
12
```

虽然看着有点不舒服，但的确达到了前面的要求。

保存

把辛苦工作的结果保存一下吧。

```
>>> wb.save("23401.xlsx")
```

如果有同名文件存在，会覆盖。

此时，可以用 Excel 打开这个文件，看看可视化的结果：

	A	B	C
1	1	2	3
2	4	5	6
3	7	8	9
4	10	11	12
-			

读取已有文件

如果已经有一个 .xlsx 文件，要读取它，可以这样做：

```
>>> from openpyxl import load_workbook
>>> wb2 = load_workbook("23401.xlsx")
>>> print wb2.get_sheet_names()
['python', 'Sheet2', 'Sheet1']
>>> ws_wb2 = wb2["python"]
>>> for row in ws_wb2.rows:
...     for cell in row:
...         print cell.value
...
1
2
3
4
5
6
7
8
9
10
11
12
```

很好，就是这个文件。

其它第三方库

针对电子表格的第三方库，除了上面这个 openpyxl 之外，还有别的，列出几个，供参考，使用方法大同小异。

- xlsxwriter：针对 Excel 2010 格式，如 .xlsx，官方网站：<https://xlsxwriter.readthedocs.org/>，这个官方文档写的图文并茂。非常好读。

下面两个用来处理 .xls 格式的电子表表格。

- xlrd：网络文件：<https://secure.simplistix.co.uk/svn/xlrd/trunk/xlrd/doc/xlrd.html?p=4966>
 - xlwt：网络文件：<http://xlwt.readthedocs.org/en/latest/>
-

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

HTML



T



9

实战



unity



HTML



实战

通过前面的学习，已经掌握了 Python 的基本内容，不少读者可能此时已经跃跃欲试，迫切地想用已经掌握的技术去搞点什么东西。

本季就是要出点一些实战的东西。

首先声明，因为仍然是教程，所以在实战中的所有例子，可能举例真正的工程代码要求还有一定的举例，比如可能没有非常优化、或者某些语句和方法使用还有进一步推敲之处。也盼望读者能够指出不足，必改正。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

10

用Tornado做网站

为做网站而准备

作为一个程序猿一定要会做网站。这也不一定吧，貌似是，但是，如果被人问及此事，如果说自己不会，的确羞愧难当呀。所以，本教程要讲一讲如何做网站。

推荐阅读：[History of the World Wide Web](#)

首先，为自己准备一个服务器。这个要求似乎有点过分，作为一个普通的穷苦聊到的程序员，哪里有铜钿来购买服务器呢？没关系，不够买服务器也能做网站，可以购买云服务空间或者虚拟空间，这个在网上搜搜，很多。如果购买这个的铜钿也没有，还可以利用自己的电脑（这总该有了）作为服务服务器。我就是利用一台装有 ubuntu 操作系统的个人电脑作为本教程的案例演示服务器。

然后，要在这个服务器上做一些程序配置。一些必备的网络配置这里就不说了，比如我用的 ubuntu 系统，默认情况都有了。如果读者遇到一些问题，可以搜一下，网上资料多多。另外的配置就是 Python 开发环境，这个应该也有了，前面已经在用了。

接下来，要安装一个框架。本教程中制作网站的案例采用 tornado 框架。

在安装这个框架之前，先了解一些相关知识。

开发框架

对框架的认识，由于工作习惯和工作内容的不同，有很大差异，这里姑且截取[维基百科中的一种定义](#)，之所以要给出一个定义，无非是让看官有所了解，但是是否知道这个定义，丝毫不影响后面的工作。

软件框架（Software framework），通常指的是为了实现某个业界标准或完成特定基本任务的软件组件规范，也指为了实现某个软件组件规范时，提供规范所要求之基础功能的软件产品。

框架的功能类似于基础设施，与具体的软件应用无关，但是提供并实现最为基础的软件架构和体系。软件开发者通常依据特定的框架实现更为复杂的商业运用和业务逻辑。这样的软件应用可以在支持同一种框架的软件系统中运行。

简而言之，框架就是制定一套规范或者规则（思想），大家（程序员）在该规范或者规则（思想）下工作。或者说就是使用别人搭好的舞台，你来做表演。

我比较喜欢最后一句的解释，别人搭好舞台，我来表演。这也就是说，如果在做软件开发的时候，能够减少工作量。就做网站来讲，其实需要做的事情很多，但是如果有了开发框架，很多底层的事情就不需要做了（都有哪些底层的事情呢？读者能否回答？）。

有高手工程师鄙视框架，认为自己编写的才是王道。这方面不争论，框架是开发中很流行的东西，我还是固执地认为用框架来开发，更划算。

Python 框架

有人说 php(什么是 php，严肃的说法，这是另外一种语言，更高雅的说法，是某个活动的汉语拼音简称) 框架多，我不否认，php 的开发框架的确很多很多。不过，Python 的 web 开发框架，也足够使用了，列举几种常见的 web 框架：

- Django: 这是一个被广泛应用的框架。在网上搜索，会发现很多公司在招聘的时候就说要会这个。框架只是辅助，真正的程序员，用什么框架，都应该是根据需要而来。当然不同框架有不同的特点，需要学习一段时间。
- Flask: 一个用 Python 编写的轻量级 Web 应用框架。基于 Werkzeug WSGI 工具箱和 Jinja2 模板引擎。
- Web2py: 是一个为 Python 语言提供的全功能 Web 应用框架，旨在敏捷快速的开发 Web 应用，具有快速、安全以及可移植的数据库驱动的应用，兼容 Google App Engine。
- Bottle: 微型 Python Web 框架，遵循 WSGI，说微型，是因为它只有一个文件，除 Python 标准库外，它不依赖于任何第三方模块。
- Tornado: 全称是 Tornado Web Server，从名字上看就可知道它可以用作 Web 服务器，但同时它也是一个 Python Web 的开发框架。最初是在 FriendFeed 公司的网站上使用，Facebook 收购了之后便开源了出来。
- webpy: 轻量级的 Python Web 框架。webpy 的设计理念力求精简 (Keep it simple and powerful)，源码很简短，只提供一个框架所必须的东西，不依赖大量的第三方模块，它没有 URL 路由、没有模板也没有数据库的访问。

说明：以上信息选自：<http://blog.jobbole.com/72306/>，这篇文章中还有别的框架，由于不是 web 框架，我没有选摘，有兴趣的去阅读。

Tornado

本教程中将选择使用 Tornado 框架。此前有朋友建议我用 Django，首先它是一个好东西。但是，我更愿意用 Tornado，为什么呢？因为……，看下边或许是理由，或许不是。

Tornado 全称 Tornado Web Server，是一个用 Python 语言写成的 Web 服务器兼 Web 应用框架，由 FriendFeed 公司在自己的网站 FriendFeed 中使用，被 Facebook 收购以后框架以开源软件形式开放给大众。看来 Tornado 的出身高贵呀，对了，某国可能风闻有 Facebook，但是要一睹其芳容，还要努力。

用哪个框架，一般是要结合项目而定。我之选用 Tornado 的原因，就是看中了它在性能方面的优异表现。

Tornado 的性能是相当优异的，因为它试图解决一个被称之为“C10k”问题，就是处理大于或等于一万的并发。一万呀，这可是不小的量。(关于 C10K 问题，看官可以浏览：[C10k problem](#))

下表是和一些其他 Web 框架与服务器的对比，供看官参考（数据来源：<https://developers.facebook.com/blog/post/301>）

条件：处理器为 AMD Opteron, 主频 2.4GHz, 4 核

服务	部署	请求/每秒
Tornado	nginx, 4 进程	8213
Tornado	1 个单线程进程	3353
Django	Apache/mod_wsgi	2223
web.py	Apache/mod_wsgi	2066
CherryPy	独立	785

看了这个对比表格，还有什么理由不选择 Tornado 呢？

就是它了——Tornado

安装 Tornado

Tornado 的官方网站：<http://www.tornadoweb.org>

我在自己电脑中（是我目前使用的服务器），用下面方法安装，只需要一句话即可：

```
pip install tornado
```

这是因为 Tornado 已经列入 PyPI，因此可以通过 pip 或者 easy_install 来安装。

如果不用这种方式安装，下面的页面中有可以供看官下载的最新源码版本和安装方式：<https://pypi.python.org/pypi/tornado/>

此外，在 github 上也有托管，看官可以通过上述页面进入到 github 看源码。

我没有在 windows 操作系统上安装过这个东西，不过，在官方网站上有一句话，可能在告诉读者一些信息：

Tornado will also run on Windows, although this configuration is not officially supported and is recommended only for development use.

特别建议，在真正的工程中，网站的服务器还是用 Linux 比较好，你懂得（吗？）。

技术准备

除了做好上述准备之外，还要有点技术准备：

- HTML
- CSS
- JavaScript

我们在后面实例中，不会搞太复杂的界面和 JavaScript(JS) 操作，所以，只需要基本知识即可。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

分析 Hello

打开你写 Python 代码用的编辑器，不要问为什么，把下面的代码一个字不差地录入进去，并命名保存为 hello.py(目录自己任意定)。

```
#!/usr/bin/env Python
#coding:utf-8

import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        greeting = self.get_argument('greeting', 'Hello')
        self.write(greeting + ', welcome you to read: www.itdiffer.com')

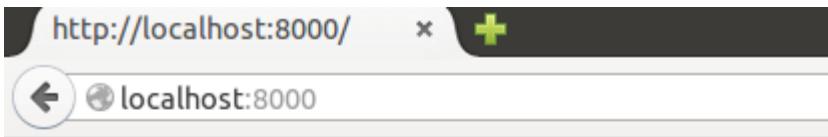
if __name__ == "__main__":
    tornado.options.parse_command_line()
    app = tornado.web.Application(handlers=[(r"/", IndexHandler)])
    http_server = tornado.httpserver.HTTPServer(app)
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()
```

进入到保存 hello.py 文件的目录，执行：

```
$ python hello.py
```

用 Python 运行这个文件，其实就已经发布了一个网站，只不过这个网站太简单了。

接下来，打开浏览器，在浏览器中输入：http://localhost:8000，得到如下界面：



Hello, welcome you to read: www.itdiffer.com

我在 ubuntu 的 shell 中还可以用下面方式运行：

```
$ curl http://localhost:8000/
Hello, welcome you to read: www.itdiffer.com

$ curl http://localhost:8000/?greeting=Qiwsir
Qiwsir, welcome you to read: www.itdiffer.com
```

此操作，读者可以根据自己系统而定。

恭喜你，迈出了决定性一步，已经可以用 Tornado 发布网站了。在这里似乎没有做什么部署，只是安装了 Tornado。是的，不需要多做什么，因为 Tornado 就是一个很好的 server，也是一个开发框架。

下面以这个非常简单的网站为例，对用 tornado 做的网站的基本结构进行解释。

WEB 服务器工作流程

任何一个网站都离不开 Web 服务器，这里所说的不是指那个更计算机一样的硬件设备，是指里面安装的软件，有时候初次接触的看官容易搞混。就来伟大的[维基百科都这么说](#)：

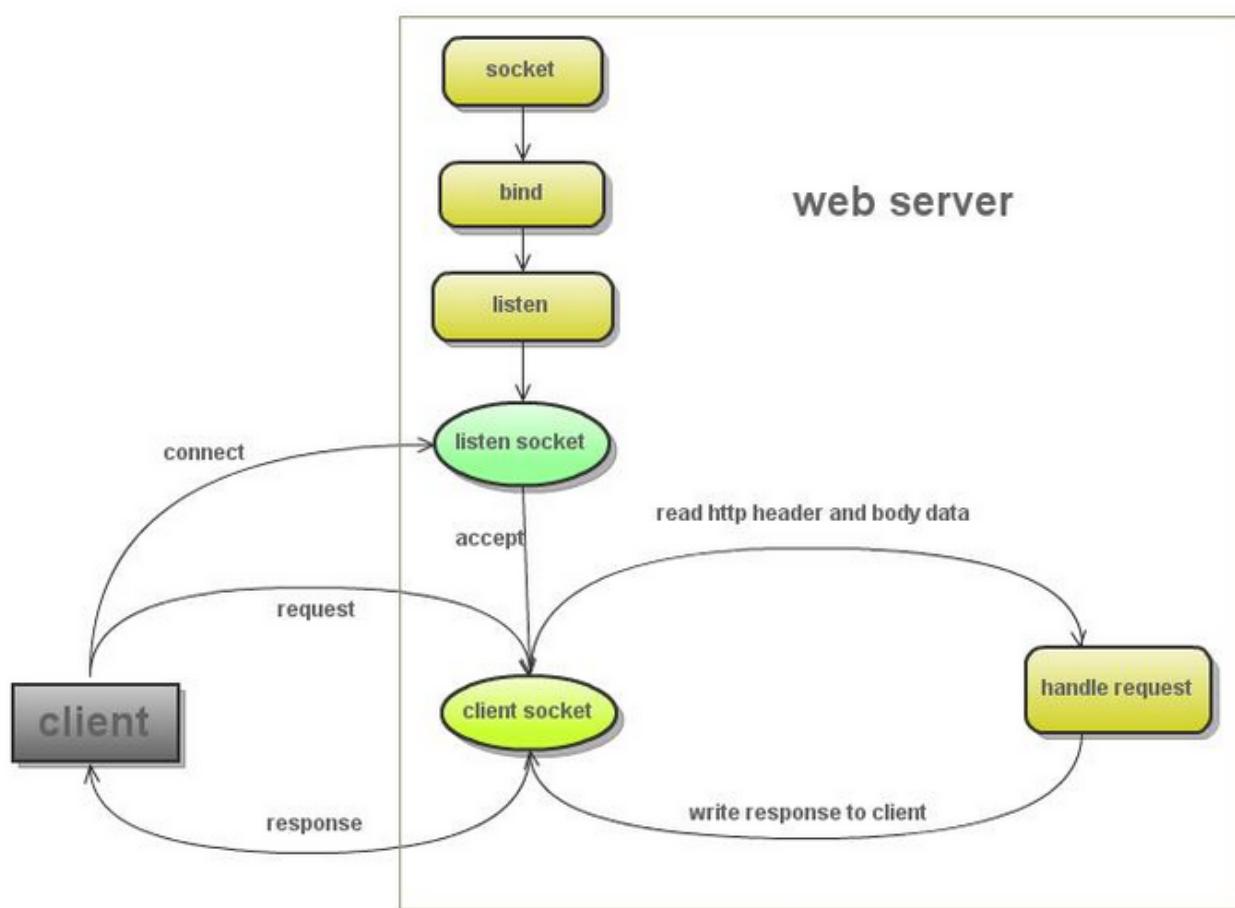
有时，这两种定义会引起混淆，如 Web 服务器。它可能是指用于网站的计算机，也可能是指像 Apache 这样的软件，运行在这样的计算机上以管理网页组件和回应网页浏览器的请求。

在具体的语境中，看官要注意分析，到底指的是什么。

关于 Web 服务器比较好的解释，推荐看看百度百科的内容，我这里就不复制粘贴了，具体可以点击连接查阅：[WEB 服务器](#)

在 WEB 上，用的最多的就是输入网址，访问某个网站。全世界那么多网站网页，如果去访问，怎么能够做到彼此互通互联呢。为了协调彼此，就制定了很多通用的协议，其中 http 协议，就是网络协议中的一种。关于这个协议的介绍，网上随处就能找到，请自己 google.

网上偷来的一张图（从哪里偷来的，我都告诉你了，多实在呀。哈哈。），显示在下面，简要说明 web 服务器的工作过程



偷个彻底，把原文中的说明也贴上：

1. 创建 listen socket，在指定的监听端口，等待客户端请求的到来
2. listen socket 接受客户端的请求，得到 client socket，接下来通过 client socket 与客户端通信
3. 处理客户端的请求，首先从 client socket 读取 http 请求的协议头，如果是 post 协议，还可能要读取客户端上传的数据，然后处理请求，准备好客户端需要的数据，通过 client socket 写给客户端

引入模块

```
import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web
```

这四个都是 Tornado 的模块，在本例中都是必须的。它们四个在一般的网站开发中，都要用到，基本作用分别是：

- `tornado.httpserver`: 这个模块就是用来解决 web 服务器的 http 协议问题，它提供了不少属性方法，实现客户端和服务器端的互通。Tornado 的非阻塞、单线程的特点在这个模块中体现。
- `tornado.ioloop`: 这个也非常重要，能够实现非阻塞 socket 循环，不能互通一次就结束呀。
- `tornado.options`: 这是命令行解析模块，也常用到。
- `tornado.web`: 这是必不可少的模块，它提供了一个简单的 Web 框架与异步功能，从而使其扩展到大量打开的连接，使其成为理想的长轮询。

读者看到这里可能有点莫名其妙，对一些属于不理解。没关系，你可以先不用管它，如果愿意管，就把不理解属于放到 google 立面查看。一定要硬着头皮一字一句地读下去，随着学习和实践的深入，现在不理解的以后就会逐渐领悟理解的。

还有一个模块引入，是用 `from...import` 完成的

```
from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)
```

这两句就显示了所谓“命令行解析模块”的用途了。在这里通过 `tornado.options.define()` 定义了访问本服务器的端口，就是在浏览器地址栏中输入 `http:localhost:8000` 的时候，才能访问本网站，因为 http 协议默认的端口是 80，为了区分，我在这里设置为 8000，为什么要区分呢？因为我的计算机或许你的也是，已经部署了别（或许是 Nginx、Apache）服务器了，它的端口是 80，所以要区分开（也可能是故意不用 80 端口），并且，后面我们还会将 tornado 和 Nginx 联合起来工作，这样两个服务器在同一台计算机上，就要分开喽。

定义请求-处理程序类

```
class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        greeting = self.get_argument('greeting', 'Hello')
        self.write(greeting + ', welcome you to read: www.itdiffer.com')
```

所谓“请求处理”程序类，就是要定义一个类，专门应付客户端（就是你打开的那个浏览器界面）向服务器提出的请求（这个请求也许是要读取某个网页，也许是要将某些信息存到服务器上），服务器要有相应的程序来接收并处理这个请求，并且反馈某些信息（或者是针对请求反馈所要的信息，或者返回其它的错误信息等）。

于是，就定义了一个类，名字是 `IndexHandler`，当然，名字可以随便取了，但是，按照习惯，类的名字中的单词首字母都是大写的，并且如果这个类是请求处理程序类，那么就最好用 `Handler` 结尾，这样在名称上很明确，是干什么的。

类 `IndexHandler` 继承 `tornado.web.RequestHandler`，其中再定义 `get()` 和 `post()` 两个在 `web` 中应用最多的方法的内容（关于这两个方法的详细解释，可以参考：[\[HTTP GET POST 的本质区别详解\]](https://github.com/qiwsir/ITArticles/blob/master/Tornado/DifferenceHttpGetPost.md)），作者在这篇文章中，阐述了两个方法的本质）。

在本例中，只定义了一个 `get()` 方法。

用 `greeting = self.get_argument('greeting', 'Hello')` 的方式可以得到 url 中传递的参数，比如

```
$ curl http://localhost:8000/?greeting=Qiwsir
Qiwsir, welcome you to read: www.itdiffer.com
```

就得到了在 url 中为 `greeting` 设定的值 `Qiwsir`。如果 url 中没有提供值，就是 `Hello`。

官方文档对这个方法的描述如下：

`RequestHandler.get_argument(name, default=, []strip=True)`

Returns the value of the argument with the given name.

If `default` is not provided, the argument is considered to be required, and we raise a `MissingArgumentError` if it is missing.

If the argument appears in the url more than once, we return the last value.

The returned value is always unicode.

接下来的那句 `self.write(greeting + ',weblcome you to read: www.itdiffer.com)'` 中，`write()` 方法主要功能是向客户端反馈信息。也浏览一下官方文档信息，对以后正确理解使用有帮助：

`RequestHandler.write(chunk)[source]`

Writes the given chunk to the output buffer.

To write the output to the network, use the `flush()` method below.

If the given chunk is a dictionary, we write it as JSON and set the Content-Type of the response to be application/json. (if you want to send JSON as a different Content-Type, call set_header after calling write()).

main() 方法

`if __name__ == "__main__"`, 这个方法跟以往执行 Python 程序是一样的。

`tornado.options.parse_command_line()`, 这是在执行 tornado 的解析命令行。在 tornado 的程序中, 只要 import 模块之后, 就会在运行的时候自动加载, 不需要了解细节, 但是, 在 main() 方法中如果有命令行解析, 必须要提前将模块引入。

Application 类

下面这句是重点:

```
app = tornado.web.Application(handlers=[(r"/", IndexHandler)])
```

将 `tornado.web.Application` 类实例化。这个实例化, 本质上是建立了整个网站程序的请求处理集合, 然后它可以被 `HTTPServer` 做为参数调用, 实现 http 协议服务器访问。Application 类的 `__init__` 方法参数形式:

```
def __init__(self, handlers=None, default_host="", transforms=None, **settings):
    pass
```

在一般情况下, `handlers` 是不能为空的, 因为 Application 类通过这个参数的值处理所得到的请求。例如在本例中, `handlers=[(r"/", IndexHandler)]`, 就意味着如果通过浏览器的地址栏输入根路径 (`http://localhost:8000`) 就是根路径, 如果是 `http://localhost:8000/qiwsir`, 就不属于根, 而是一个子路径或目录了), 对应着就是让名字为 `IndexHandler` 类处理这个请求。

通过 `handlers` 传入的数值格式, 一定要注意, 在后面做复杂结构的网站是, 这里就显得重要了。它是一个 `list`, `list` 里面的元素是 `tuple`, `tuple` 的组成包括两部分, 一部分是请求路径, 另外一部分是处理程序的类名称。注意请求路径可以用正则表达式书写(关于正则表达式, 后面会进行简要介绍)。举例说明:

```
handlers = [
    (r"/", IndexHandlers),          #来自根路径的请求用 IndexHandlers 处理
    (r"/qiwsir/(.*)", QiwsirHandlers), #来自 /qiwsir/ 以及其下任何请求 (正则表达式表示任何字符) 都由 QiwsirHandlers 处理
]
```

注意

在这里我使用了 `r"/"` 的样式，意味着就不需要使用转义符，`r` 后面的都表示该符号本来的含义。例如，`\n`，如果单纯这么来使用，就以为着换行，因为符号“\”具有转义功能（关于转义详细阅读[《字符串\(1\)》](#)），当写成 `r"\n"` 的形式是，就不再表示换行了，而是两个字符，`\`和`n`，不会转意。一般情况下，由于正则表达式和`\`会有冲突，因此，当一个字符串使用了正则表达式后，最好在前面加上`'r'`。

关于 Application 类的介绍，告一段落，但是并未完全讲述了，因为还有别的参数设置没有讲，请继续关注后续内容。

HTTPServer 类

实例化之后，Application 对象（用`app`做为标签的）就可以被另外一个类 HTTPServer 引用，形式为：

```
http_server = tornado.httpserver.HTTPServer(app)
```

HTTPServer 是 `tornado.httpserver` 里面定义的类。HTTPServer 是一个单线程非阻塞 HTTP 服务器，执行 HTTPServer 一般要回调 Application 对象，并提供发送响应的接口，也就是下面的内容是跟随上面语句的（`options.port` 的值在 `IndexHandler` 类前面通过 `from...import..` 设置的）。

```
http_server.listen(options.port)
```

这种方法，就建立了单进程的 http 服务。

请看官牢记，如果在以后编码中，遇到需要多进程，请参考官方文档说明：<http://tornado.readthedocs.org/en/latest/httpserver.html#http-server>

IOLoop 类

剩下最后一句了：

```
tornado.ioloop.IOLoop.instance().start()
```

这句话，总是在 `__main__` 的最后一句。表示可以接收来自 HTTP 的请求了。

以上把一个简单的 `hello.py` 剖析。想必读者对 Tornado 编写网站的基本概念已经有了。

如果一头雾水，也不要着急，以来将上面的内容多看几遍。对整体结构有一个基本了解，不要拘泥于细节或者某些词汇含义。然后即继续学习。

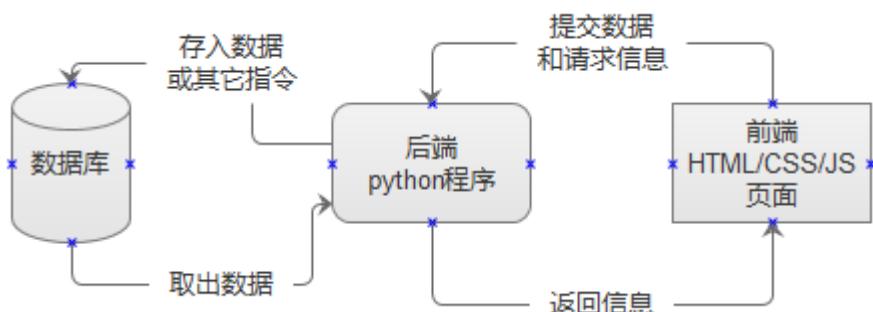
如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

用 tornado 做网站 (1)

从现在开始，做一个网站，当然，这个网站只能算是一个毛坯的，可能很简陋，但是网站的主要元素，它都会涉及到，读者通过此学习，能够了解网站的开发基本结构和内容，并且对前面的知识可以有综合应用。

基本结构

下面是一个网站的基本结构



前端

这是一个不很严格的说法，但是在日常开发中，都这么说。在网站中，所谓前端就是指用浏览器打开之后看到的那部分，它是呈现网站传过来的信息的界面，也是用户和网站之间进行信息交互的界面。撰写前端，一般使用 HTML/CSS/JS，当然，非要用 Python 也不是不可以（例如上节中的例子，就没有用 HTML/CSS/JS），但这势必造成以后维护困难。

MVC 模式是一个非常好的软件架构模式，在网站开发中，也常常要求遵守这个模式。请阅读维基百科的解释：

MVC 模式（Model–View–Controller）是软件工程中的一种软件架构模式，把软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。

MVC 模式最早由 Trygve Reenskaug 在 1978 年提出，是施乐帕罗奥多研究中心（Xerox PARC）在 20 世纪 80 年代为程序语言 Smalltalk 发明的一种软件设计模式。MVC 模式的目的是实现一种动态的程式设计，使后续对程序的修改和扩展简化，并且使程序某一部分的重复利用成为可能。除此之外，此模式通过对复杂度的简化，使程序结构更加直观。软件系统通过对自身基本部分分离的同时也赋予了各个基本部分应有的功能。专业人员可以通过自身的专长分组：

- （控制器 Controller） – 负责转发请求，对请求进行处理。

- (视图 View) – 界面设计人员进行图形界面设计。– (模型 Model) – 程序员编写程序应有的功能（实现算法等等）、数据库专家进行数据管理和数据库设计(可以实现具体的功能)。

所谓“前端”，就对大概对应着 View 部分，之所以说是大概，因为 MVC 是站在一个软件系统的角度进行划分的，上图中的前后端，与其说是系统部分的划分，不如严格说是系统功能的划分。

前端所实现的功能主要有：

- 呈现内容。这些内容是根据 url，由后端从数据库中提取出来的。前端将其按照一定的样式呈现出来。另外，有一些内容，不是后端数据库提供的，是写在前端的。
- 用户与网站交互。现在的网站，这是必须的，比如用户登录。当用户在指定的输入框中输入信息之后，该信息就是被前端提交给后端，后端对这个信息进行处理之后，在一般情况下都要再反馈给前端一个处理结果，然后前端呈现给用户。

后端

这里所说的后端，对应着 MVC 中的 Controller 和 Model 的部分或者全部功能，因为在我们的图中，“后端”是一个狭隘的概念，没有把数据库放在其内。

不在这些术语上纠结。

在我们这里，后端就是用 Python 写的程序。主要任务就是根据需要处理由前端发过来的各种请求，根据请求的处理结果，一方面操作数据库（对数据库进行增删改查），另外一方面把请求的处理结果反馈给前端。

数据库

工作比较单一，就是面对后端的 Python 程序，任其增删改查。

关于 Python 如何操作数据库，在本教程的第贰季第柒章中已经有详细的叙述，请读者阅览。

一个基本框架

上节中，显示了一个只能显示一行字的网站，那个网站由于功能太单一，把所有的东西都写到一个文件中。在真正的工程开发中，如果那么做，虽然不是不可，但开发过程和后期维护会遇到麻烦，特别是不便于多人合作。

所以，要做一个基本框架。以后网站就在这个框架中开发。

建立一个目录，在这个目录中建立一些子目录和文件。

```
/.  
|
```

```

handlers
|
methods
|
statics
|
templates
|
application.py
|
server.py
|
url.py

```

这个结构建立好，就摆开了一个做网站的架势。有了这个架势，后面的事情就是在这个基础上添加具体内容了。当然，还可以用另外一个更好听的名字，称之为设计。

依次说明上面的架势中每个目录和文件的作用（当然，这个作用是我规定的，读者如果愿意，也可以根据自己的意愿来任意设计）：

- handlers：我准备在这个文件夹中放前面所说的后端 Python 程序，主要处理来自前端的请求，并且操作数据库。
- methods：这里准备放一些函数或者类，比如用的最多的读写数据库的函数，这些函数被 handlers 里面的程序使用。
- statics：这里准备放一些静态文件，比如图片，css 和 javascript 文件等。
- templates：这里放模板文件，都是以 html 为扩展名的，它们将直接面对用户。

另外，还有三个 Python 文件，依次写下如下内容。这些内容的功能，已经在上节中讲过，只是这里进行分门别类。

url.py 文件

```

#!/usr/bin/env Python
# coding=utf-8
"""

the url structure of website
"""

import sys  #utf-8, 兼容汉字
reload(sys)
sys.setdefaultencoding("utf-8")

from handlers.index import IndexHandler #假设已经有了

```

```
url = [
    (r'/', IndexHandler),
]
```

url.py 文件主要是设置网站的目录结构。`from handlers.index import IndexHandler`，虽然在 handlers 文件夹还没有什么东西，为了演示如何建立网站的目录结构，假设在 handlers 文件夹里面已经有了一个文件 index.py，它里面还有一个类 IndexHandler。在 url.py 文件中，将其引用过来。

变量 url 指向一个列表，在列表中列出所有目录和对应的处理类。比如 `(r'/', IndexHandler)`，就是约定网站根目录的处理类是 IndexHandler，即来自这个目录的 get() 或者 post() 请求，均有 IndexHandler 类中相应方法来处理。

如果还有别的目录，如法炮制。

application.py 文件

```
#!/usr/bin/env Python
# coding=utf-8

from url import url

import tornado.web
import os

settings = dict(
    template_path = os.path.join(os.path.dirname(__file__), "templates"),
    static_path = os.path.join(os.path.dirname(__file__), "statics")
)

application = tornado.web.Application(
    handlers = url,
    **settings
)
```

从内容中可以看出，这个文件完成了对网站系统的基本配置，建立网站的请求处理集合。

`from url import url` 是将 url.py 中设定的目录引用过来。

`setting` 引用了一个字典对象，里面约定了模板和静态文件的路径，即声明已经建立的文件夹"templates"和"statics"分别为模板目录和静态文件目录。

接下来的 application 就是一个请求处理集合对象。请注意 `tornado.web.Application()` 的参数设置：

```
| tornado.web.Application(handlers=None, default_host="", transforms=None, **settings)
```

关于 settings 的设置，不仅仅是文件中的两个，还有其它，比如，如果填上 `debug = True` 就表示出于调试模式。调试模式的好处就在于有利于开发调试，但是，在正式部署的时候，最好不要用调试模式。其它更多的 settings 可以参看官方文档：[tornado.web—RequestHandler and Application classes](#)

server.py 文件

这个文件的作用是将 tornado 服务器运行起来，并且囊括前面两个文件中的对象属性设置。

```
#!/usr/bin/env Python
# coding=utf-8

import tornado.ioloop
import tornado.options
import tornado.httpserver

from application import application

from tornado.options import define, options

define("port", default = 8000, help = "run on the given port", type = int)

def main():
    tornado.options.parse_command_line()
    http_server = tornado.httpserver.HTTPServer(application)
    http_server.listen(options.port)

    print "Development server is running at http://127.0.0.1:%s" % options.port
    print "Quit the server with Control-C"

    tornado.ioloop.IOLoop.instance().start()

if __name__ == "__main__":
    main()
```

此文件中的内容，在[上节](#)已经介绍，不再赘述。

如此这般，就完成了网站架势的搭建。

后面要做的是向里面添加内容。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

用 tornado 做网站 (2)

既然摆好了一个网站的架势，下面就可以向里面填内容。

连接数据库

要做的网站，有数据库支持，虽然这不是必须的，但是如果做一个功能强悍的网站，数据库就是必须的了。

接下来的网站，我暂且采用 mysql 数据库。

怎么连接 mysql 数据呢？其方法跟《mysql 数据库(1)》中的方法完全一致。为了简单，我也不新建数据库了，就利用已经有的那个数据库。

在上一节中已经建立的文件夹 methods 中建立一个文件 db.py，并且参考《mysql 数据库 (1)》(页 0)的内容，分别建立起连接对象和游标对象。代码如下：

```
#!/usr/bin/env Python
# coding=utf-8

import MySQLdb

conn = MySQLdb.connect(host="localhost", user="root", passwd="123123", db="qiwsirtest", port=3306, charset="utf8") #

cur = conn.cursor() #游标对象
```

用户登录

前端

很多网站上都看到用户登录功能，这里做一个简单的登录，其功能描述为：

当用户输入网址，呈现在眼前的是一个登录界面。在用户名和密码两个输入框中分别输入了正确的用户名和密码之后，点击确定按钮，登录网站，显示对该用户的欢迎信息。

用图示来说明，首先呈现下图：

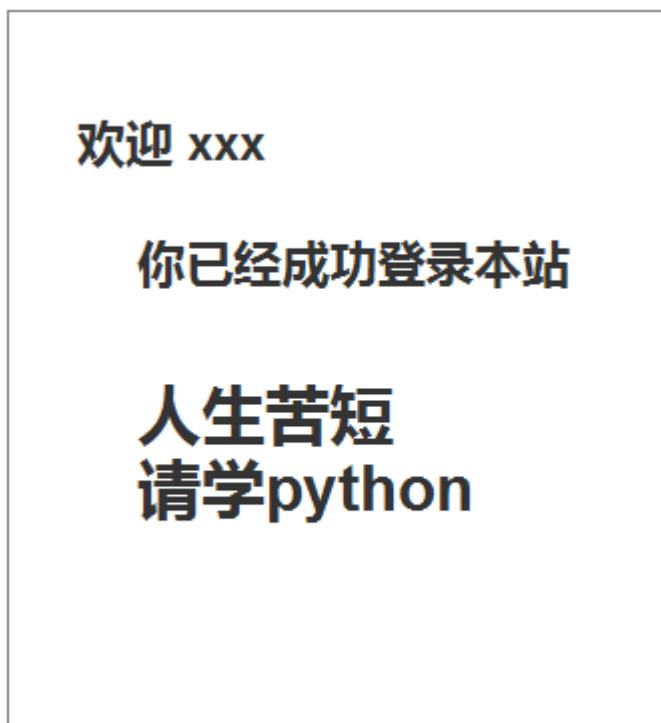
用户登录界面

用户名 :

密 码 :

尚未注册的用户 , 请注册

用户点击“登录”按钮，经过验证是合法用户之后，就呈现这样的界面：



先用 HTML 写好第一个界面。进入到 templates 文件，建立名为 index.html 的文件：

```
<!DOCTYPE html>
<head>
```

```

<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1" />
<title>Learning Python</title>
</head>
<body>
<h2>Login</h2>
<form method="POST">
<p><span>UserName:</span><input type="text" id="username"/></p>
<p><span>Password:</span><input type="password" id="password" /></p>
<p><input type="BUTTON" value="LOGIN" id="login" /></p>
</form>
</body>

```

这是一个很简单前端界面。要特别关注 `<meta name="viewport" content="width=device-width, initial-scale=1" />`，其目的在将网页的默认宽度(viewport)设置为设备的屏幕宽度(width=device-width)，并且原始缩放比例为 1.0(initial-scale=1)，即网页初始大小占屏幕面积的 100%。这样做的目的，是让在电脑、手机等不同大小的屏幕上，都能非常好地显示。

这种样式的网页，就是“自适应页面”。当然，自适应页面绝非是仅仅有这样一行代码就完全解决的。要设计自适应页面，也就是要进行“响应式设计”，还需要对 CSS、JS 乃至其它元素如表格、图片等进行设计，或者使用一些响应式设计的框架。这个目前暂不讨论，读者可以网上搜索有关资料阅读。

一提到要能够在手机上，读者是否想到了 HTML5 呢，这个被一些人热捧、被另一些人蔑视的家伙，毋庸置疑，现在已经得到了越来越广泛的应用。

HTML5 是 HTML 最新的修订版本，2014 年 10 月由万维网联盟（W3C）完成标准制定。目标是取代 1999 年所制定的 HTML 4.01 和 XHTML 1.0 标准，以期能在互联网应用迅速发展的时候，使网络标准达到符合当代的网络需求。广义论及 HTML5 时，实际指的是包括 HTML、CSS 和 JavaScript 在内的一套技术组合。

响应式网页设计（英语：Responsive web design，通常缩写为 RWD），又称为自适应网页设计、回应式网页设计。是一种网页设计的技术做法，该设计可使网站在多种浏览设备（从桌面电脑显示器到移动电话或其他移动产品设备）上阅读和导航，同时减少缩放、平移和滚动。

如果要看效果，可以直接用浏览器打开网页，因为它是 .html 格式的文件。

引入 jQuery

虽然完成了视觉上的设计，但是，如果点击那个 login 按钮，没有任何反应。因为它还仅仅是一个孤立的页面，这时候需要一个前端交互利器——javascript。

对于 javascript，不少人对它有误解，总认为它是从 java 演化出来的。的确，两个有相像的地方。但 javascript 和 java 的关系，就如同“雷峰塔”和“雷锋”的关系一样。详细读一读来自维基百科的诠释。

JavaScript，一种直译式脚本语言，是一种动态类型、弱类型、基于原型的语言，内置支持类。它的解释器被称为 JavaScript 引擎，为浏览器的一部分，广泛用于客户端的脚本语言，最早是在 HTML 网页上使用，用来给 HTML 网页增加动态功能。然而现在 JavaScript 也可被用于网络服务器，如 Node.js。

在 1995 年时，由网景公司的布兰登·艾克，在网景导航者浏览器上首次设计实现而成。因为网景公司与昇阳公司合作，网景公司管理层希望它外观看起来像 Java，因此取名为 JavaScript。但实际上它的语义与 Self 及 Scheme 较为接近。

为了获取技术优势，微软推出了 JScript，与 JavaScript 同样可在浏览器上运行。为了统一规格，1997 年，在 ECMA（欧洲计算机制造商协会）的协调下，由网景、昇阳、微软和 Borland 公司组成的工作组确定统一标准：ECMA-262。因为 JavaScript 兼容于 ECMA 标准，因此也称为 ECMAScript。

但是，我更喜欢用 jQuery，因为它的确让我省了不少事。

jQuery 是一套跨浏览器的 JavaScript 库，简化 HTML 与 JavaScript 之间的操作。由约翰·雷西格（John Resig）在 2006 年 1 月的 BarCamp NYC 上发布第一个版本。目前是由 Dave Methvin 领导的开发团队进行开发。全球前 10,000 个访问最高的网站中，有 65% 使用了 jQuery，是目前最受欢迎的 JavaScript 库。

在 index.html 文件中引入 jQuery 的方法有多种。

原则上将，可以在 HTML 文件的任何地方引入 jQuery 库，但是通常放置的地方在 html 文件的开头 `<head>...</head>` 中，或者在文件的末尾 `</body>` 以内。放在开头，如果所用的库比较大、比较多，在载入页面时时间相对长点。

第一种引入方法，是国际化的一种：

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
```

这是直接从 jQuery CDN(Content Delivery Network)上直接引用，好处在于如果这个库更新，你不用任何操作，就直接使用最新的了。但是，如果在你的网页中这么用了，如果在某个有很多自信的国家上网，并且没有梯子，会发现网页几乎打不开，就是因为连接上面那个地址的通道是被墙了。

当然，jQuery CDN 不止一个，比如官方网站的：

```
<script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
```

第二种引入方法，就是将 jQuery 下载下来，放在指定地方（比如，与自己网站在同一个存储器中，或者自己可以访问的另外服务器）。到官方网站 (<https://jqueryui.com/>) 下载最新的库，然后将它放在已经建立的 statics 目录内，为了更清楚区分，可以在里面建立一个子目录 js，jquery 库放在 js 子目录里面。下载的时候，建议下载以 min.js 结尾的文件，因为这个是经过压缩之后，体积小。

我在 statics/js 目录中放置了下载的库，并且为了简短，更名为 jquery.min.js。

本来可以用下面的方法引入：

```
<script src="statics/js/jquery.min.js"></script>
```

如果这样写，也是可以的。但是，考虑到 tornado 的特点，用下面方法引入，更具有灵活性：

```
<script src="{{static_url("js/jquery.min.js")}}></script>
```

不仅要引入 jquery，还需要引入自己写的 js 指令，所以要建立一个文件，我命名为 script.js，也同时引用过来。虽然目前这个文件还是空的。

```
<script src="{{static_url("js/script.js")}}></script>
```

这里用的 static_url 是一个函数，它是 tornado 模板提供的一个函数。用这个函数，能够制定静态文件。之所以用它，而不是用上面的那种直接调用的方法，主要原因是如果某一天，将静态文件目录 statics 修改了，也就是不指定 statics 为静态文件目录了，定义别的目录为静态文件目录。只需要在定义静态文件目录那里修改（定义静态文件目录的方法请参看上一节），而其它地方的代码不需要修改。

编写 js

先写一个测试性质的东西。

用编辑器打开 statics/js/script.js 文件，如果没有就新建。输入的代码如下：

```
$(document).ready(function(){
    alert("good");
    $("#login").click(function(){
        var user = $("#username").val();
        var pwd = $("#password").val();
        alert("username: "+user);
    });
});
```

由于本教程不是专门讲授 javascript 或者 jquery，所以，在 js 代码部分，只能一带而过，不详细解释。

上面的代码主要实现获取表单中 id 值分别为 username 和 password 所输入的值，alert 函数的功能是把值以弹出菜单的方式显示出来。

hanlers 里面的程序

是否还记得在上一节中，在 url.py 文件中，做了这样的设置：

```
from handlers.index import IndexHandler #假设已经有了

url = [
    (r'/', IndexHandler),
]
```

现在就去把假设有了的那个文件建立起来，即在 `handlers` 里面建立 `index.py` 文件，并写入如下代码：

```
#!/usr/bin/env Python
# coding=utf-8

import tornado.web

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        self.render("index.html")
```

当访问根目录的时候（不论输入 `localhost:8000`，还是 `http://127.0.0.1:8000`，或者网站域名），就将相应的请求交给了 `handlers` 目录中的 `index.py` 文件中的 `IndexHandler` 类的 `get()` 方法来处理，它的处理结果是呈现 `index.html` 模板内容。

`render()` 函数的功能在于向请求者反馈网页模板，并且可以向模板中传递数值。关于传递数值的内容，在后面介绍。

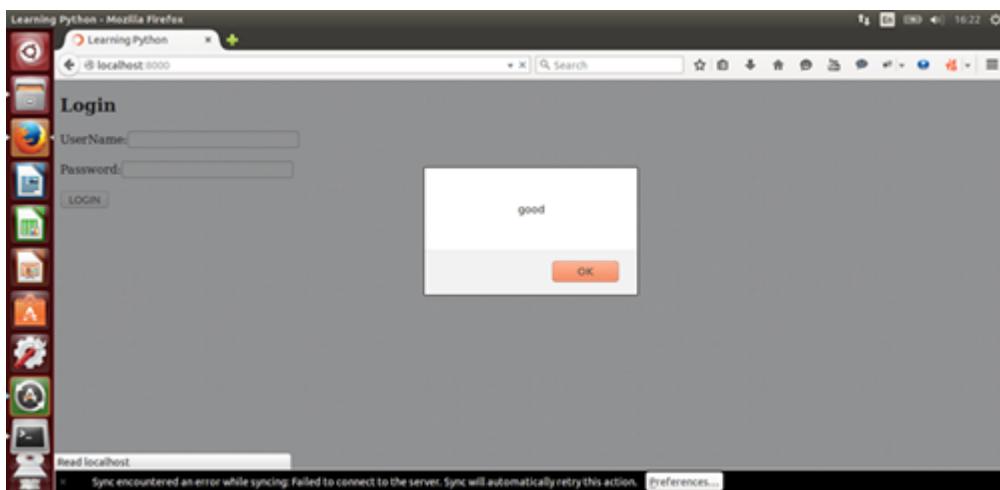
上面的文件保存之后，回到 `handlers` 目录中。因为这里面的文件要在别处被当做模块引用，所以，需要在这里建立一个空文件，命名为 `__init__.py`。这个文件非常重要。在[编写模块](#)一节中，介绍了引用模块的方法。但是，那些方法有一个弊端，就是如果某个目录中有多个文件，就显得麻烦了。其实 Python 已经想到这点了，于是就提供了 `__init__.py` 文件，只要在该目录中加入了这个文件，该目录中的其它 `.py` 文件就可以作为模块被 Python 引入了。

至此，一个带有表单的 tornado 网站就建立起来了。读者可以回到上一级目录中，找到 `server.py` 文件，运行它：

```
$ python server.py
Development server is running at http://127.0.0.1:8000
Quit the server with Control-C
```

如果读者在前面的学习中，跟我的操作完全一致，就会在 shell 中看到上面的结果。

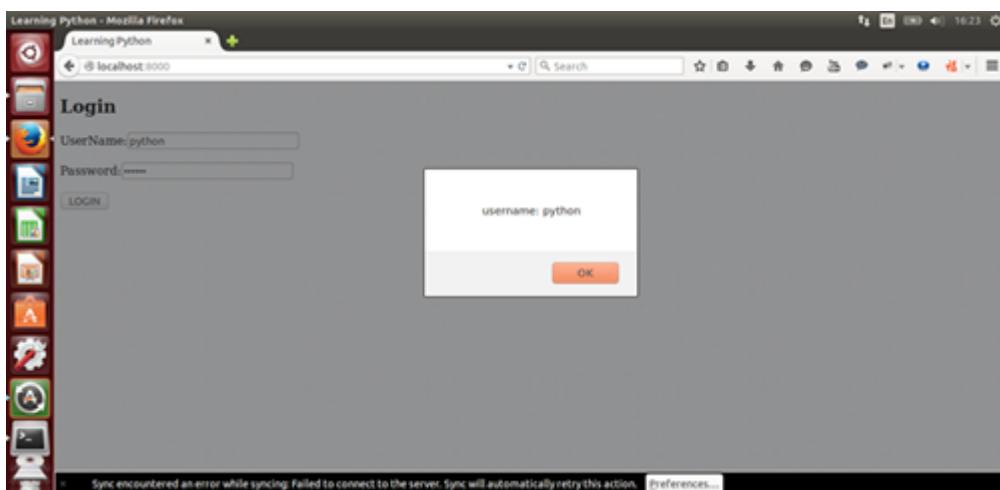
打开浏览器，输入 `http://localhost:8000` 或者 `http://127.0.0.1:8000`，看到的应该是：



这就是 script.js 中的开始起作用了，第一句是要弹出一个对话框。点击“确定”按钮之后，就是：



在这个页面输入用户名和密码，然后点击 Login 按钮，就是：



一个网站有了雏形。不过，当提交表单的反应，还仅仅停留在客户端，还没有向后端传递客户端的数据信息。请继续学习下一节。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我, 请通过支付宝: qiwsir@126.com,不胜感激。

用 tornado 做网站 (3)

数据传输

在已经建立了前端表单之后，就要实现前端和后端之间的数据传递。在工程中，常用到一个被称之为 ajax() 的方法。

关于 ajax 的故事，需要浓墨重彩，因为它足够精彩。

ajax 是 “Asynchronous Javascript and XML”（异步 JavaScript 和 XML）的缩写，在它的发展历程中，汇集了众家贡献。比如微软的 IE 团队曾经将 XHR(XML HttpRequest) 用于 web 浏览器和 web 服务器间传输数据，并且被 W3C 标准采用。当然，也有其它公司为 Ajax 技术做出了贡献，虽然它们都被遗忘了，比如 Oddpost，后来被 Yahoo! 收购并成为 Yahoo! Mail 的基础。但是，真正让 Ajax 大放异彩的 google 是不能被忽视的，正是 google 在 Gmail、Suggest 和 Maps 上大规模使用了 Ajax，才使得人们看到了它的魅力，程序员由此而兴奋。

技术总是在不断进化的，进化的方向就是用着越来越方便。

回到上一节使用的 jQuery，里面有 ajax() 方法，能够让程序员方便的调用。

ajax() 方法通过 HTTP 请求加载远程数据。

该方法是 jQuery 底层 AJAX 实现。简单易用的高层实现见 \$.get, \$.post 等。\$.ajax() 返回其创建的 XMLHttpRequest 对象。大多数情况下你无需直接操作该函数，除非你需要操作不常用的选项，以获得更多的灵活性。

最简单的情况下，\$.ajax() 可以不带任何参数直接使用。

在上文介绍 Ajax 的时候，用到了一个重要的术语——“异步”，与之相对应的叫做“同步”。我引用来自[阮一峰的网络日志](#)中的通俗描述：

"同步模式"就是上一段的模式，后一个任务等待前一个任务结束，然后再执行，程序的执行顺序与任务的排列顺序是一致的、同步的；"异步模式"则完全不同，每一个任务有一个或多个回调函数（callback），前一个任务结束后，不是执行后一个任务，而是执行回调函数，后一个任务则是不等前一个任务结束就执行，所以程序的执行顺序与任务的排列顺序是不一致的、异步的。

"异步模式"非常重要。在浏览器端，耗时很长的操作都应该异步执行，避免浏览器失去响应，最好的例子就是 Ajax 操作。在服务器端，"异步模式"甚至是唯一的模式，因为执行环境是单线程的，如果允许同步执行所有 http 请求，服务器性能会急剧下降，很快就会失去响应。

看来，`ajax()` 是前后端进行数据传输的重要角色。

承接上一节的内容，要是用 `ajax()` 方法，需要修改 `script.js` 文件内容即可：

```
$(document).ready(function(){
    $("#login").click(function(){
        var user = $("#username").val();
        var pwd = $("#password").val();
        var pd = {"username":user, "password":pwd};
        $.ajax({
            type:"post",
            url:"/",
            data:pd,
            cache:false,
            success:function(data){
                alert(data);
            },
            error:function(){
                alert("error!");
            },
        });
    });
});
```

在这段代码中，`var pd = {"username":user, "password":pwd};` 意即将得到的 `user` 和 `pwd` 值，放到一个 json 对象中（关于 json，请阅读[《标准库\(8\)》](#)），形成了一个 json 对象。接下来就是利用 `ajax()` 方法将这个 json 对象传给后端。

jQuery 中的 `ajax()` 方法使用比较简单，正如上面代码所示，只需要 `$.ajax()` 即可，不过需要对立面的参数进行说明。

- `type`: post 还是 get。关于 post 和 get 的区别，可以阅读：[\[HTTP POST GET 本质区别详解 href="http://github.com/qiwsir/ITArticles/blob/master/Tornado/DifferenceHttpPost.md"\]](http://github.com/qiwsir/ITArticles/blob/master/Tornado/DifferenceHttpPost.md)
- `url`: post 或者 get 的地址
- `data`: 传输的数据，包括三种：（1）html 拼接的字符串；（2）json 数据；（3）form 表单经 `serialize()` 序列化的。本例中传输的就是 json 数据，这也是经常用到的一种方式。
- `cache`: 默认为 true，如果不允许缓存，设置为 false.
- `success`: 请求成功时执行回调函数。本例中，将返回的 data 用 `alert` 方式弹出来。读者是否注意到，我在很多地方都用了 `alert()` 这个东西，目的在于调试，走一步看一步，看看得到的数据是否如自己所要。也是有点不自信呀。

- error: 如果请求失败所执行的函数。

后端接受数据

前端通过 ajax 技术，将数据已 json 格式传给了后端，并且指明了对象目录 "/"，这个目录在 url.py 文件中已经做了配置，是由 handlers 目录的 index.py 文件的 IndexHandler 类来出来。因为是用 post 方法传的数据，那么在这个类中就要有 post 方法来接收数据。所以，要在 IndexHandler 类中增加 post()，增加之后的完善代码是：

```
#!/usr/bin/env Python
# coding=utf-8

import tornado.web

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        self.render("index.html")

    def post(self):
        username = self.get_argument("username")
        password = self.get_argument("password")
        self.write(username)
```

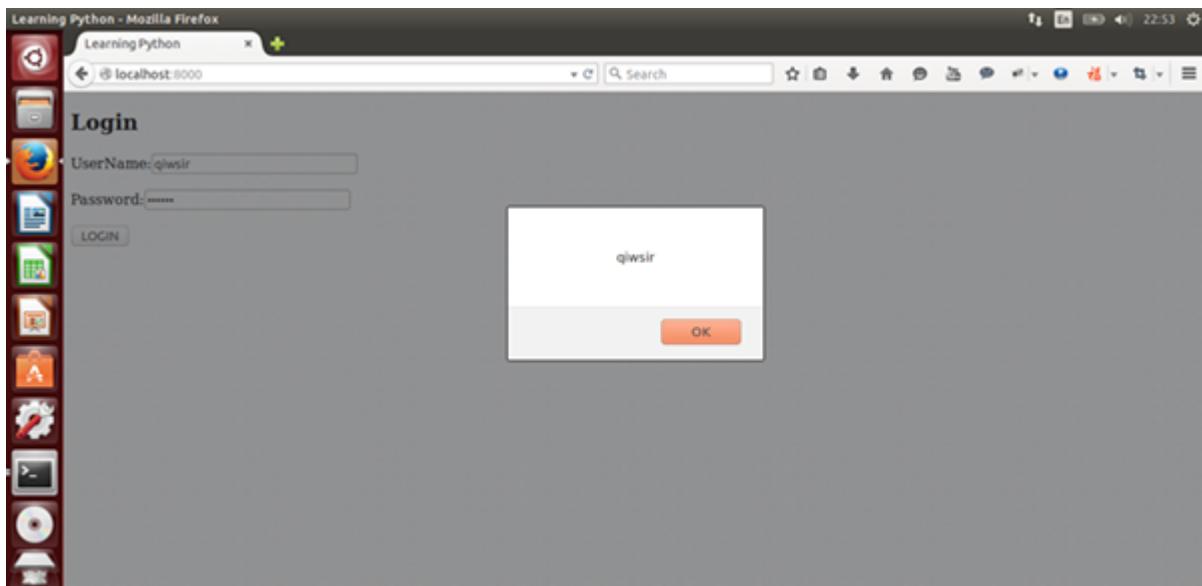
在 post() 方法中，使用 get_argument() 函数来接收前端传过来的数据，这个函数的完整格式是 `get_argument(name, default=[], strip=True)`，它能够获取 name 的值。在上面的代码中，name 就是从前端传到后端的那个 json 对象的键的名字，是哪个键就获取该键的值。如果获取不到 name 的值，就返回 default 的值，但是这个值默认是没有的，如果真的没有就会抛出 HTTP 400。特别注意，在 get 的时候，通过 get_argument() 函数获得 url 的参数，如果是多个参数，就获取最后一个的值。要想获取多个值，可以使用 `get_arguments(name, strip=True)`。

上例中分别用 get_argument() 方法得到了 username 和 password，并且它们都是 unicode 编码的数据。

tornado.web.RequestHandler 的方法 write()，即上例中的 `self.write(username)`，是后端向前端返回数据。这里返回的实际上是一个字符串，也可返回 json 字符串。

如果读者要查看修改代码之后的网站效果，最有效的方式先停止网站（`ctrl+c`），在从新执行 `Python server.py` 运行网站，然后刷新浏览器即可。这是一种较为笨拙的方法。一种灵巧的方法是开启调试模式。是否还记得？在设置 setting 的时候，写上 `debug = True` 就表示是调试模式了（参阅：[用 tornado 做网站 \(1\)](#)）。但是，调试模式也不是十全十美，如果修改模板，就不会加载，还需要重启服务。反正重启也不麻烦，无妨啦。

看看上面的代码效果：



这是前端输入了用户名和密码之后，点击 login 按钮，提交给后端，后端再向前端返回数据之后的效果。就是我们想要的结果。

验证用户名和密码

按照流程，用户在前端输入了用户名和密码，并通过 ajax 提交到了后端，后端借助于 `get_argument()` 方法得到了所提交的数据（用户名和密码）。下面要做的事情就是验证这个用户名和密码是否合法，其体现在：

- 数据库中是否有这个用户
- 密码和用户先前设定的密码（已经保存在数据库中）是否匹配

这个验证工作完成之后，才能允许用户登录，登录之后才能继续做某些事情。

首先，在 `methods` 目录中（已经有了一个 `db.py`）创建一个文件，我命名为 `readdb.py`，专门用来存储读数据用的函数（这种划分完全是为了明确和演示一些应用方法，读者也可以都写到 `db.py` 中）。这个文件的代码如下：

```
#!/usr/bin/env Python
# coding=utf-8

from db import *

def select_table(table, column, condition, value):
    sql = "select " + column + " from " + table + " where " + condition + "=" + value + ""
    cur.execute(sql)
```

```
lines = cur.fetchall()  
return lines
```

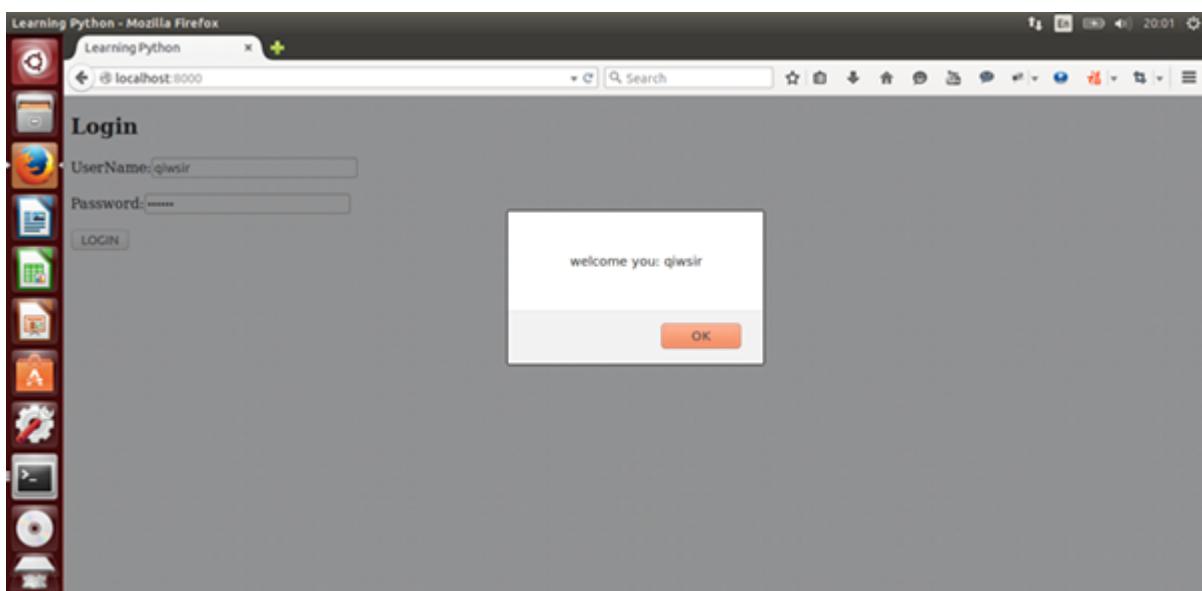
上面这段代码，建议读者可以写上注释，以检验自己是否能够将以往的知识融会贯通地应用。恕我不再解释。

有了这段代码之后，就进一步改写 index.py 中的 post() 方法。为了明了，将 index.py 的全部代码呈现如下：

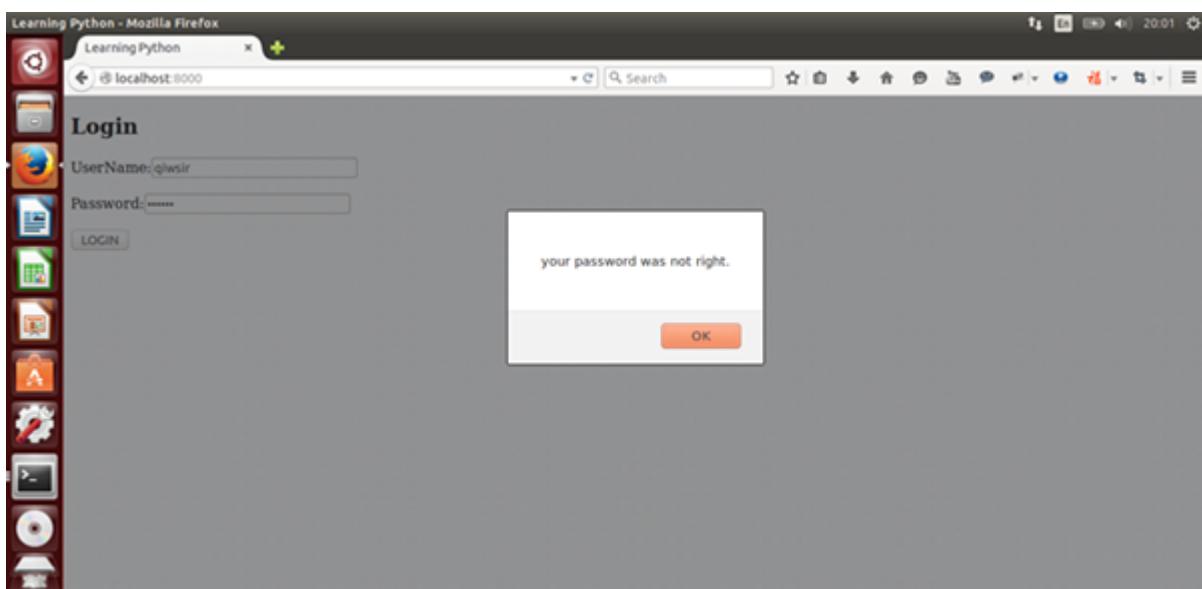
```
#!/usr/bin/env Python  
# coding=utf-8  
  
import tornado.web  
import methods.readdb as mrd  
  
class IndexHandler(tornado.web.RequestHandler):  
    def get(self):  
        self.render("index.html")  
  
    def post(self):  
        username = self.get_argument("username")  
        password = self.get_argument("password")  
        user_infos = mrd.select_table(table="users", column="*", condition="username", value=username)  
        if user_infos:  
            db_pwd = user_infos[0][2]  
            if db_pwd == password:  
                self.write("welcome you: " + username)  
            else:  
                self.write("your password was not right.")  
        else:  
            self.write("There is no thi user.")
```

特别注意，在 methods 目录中，不要缺少了 `__init__.py` 文件，才能在 index.py 中实现 `import methods.readdb as mrd`。

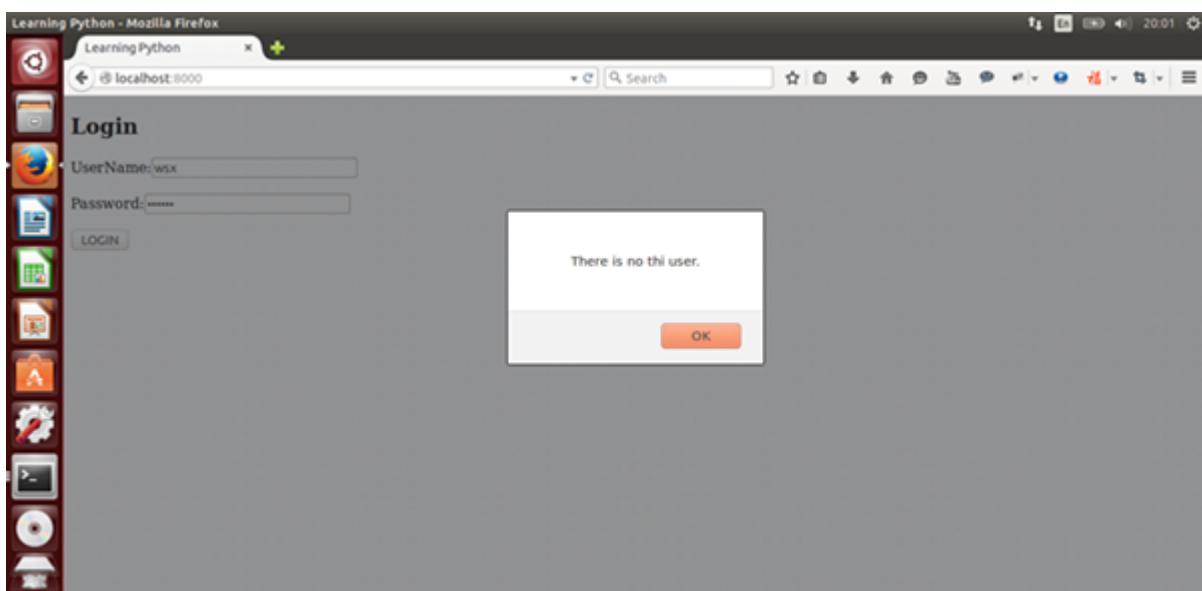
代码修改到这里，看到的结果是：



这是正确输入用户名（所谓正确，就是输入的用户名和密码合法，即在数据库中有该用户名，且密码匹配），并提交数据后，反馈给前端的欢迎信息。



如果输入的密码错误了，则如此提示。



这是随意输入的结果，数据库中无此用户。

需要特别说明一点，上述演示中，数据库中的用户密码并没有加密。关于密码加密问题，后续要研究。

总目录 (页 0)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

用 tornado 做网站 (4)

模板

已经基本了解前端向和后端如何传递数据，以及后端如何接收数据的过程和方法之后。我突然发现，前端页面写的太难看了。俗话说“外行看热闹，内行看门道”。程序员写的网站，在更多时候是给“外行”看的，他们可没有耐心来看代码，他们看的就是界面，因此界面是否做的漂亮一点点，是直观重要的。

其实，也不仅仅是漂亮的原因，因为前端页面，还要显示从后端读取出来的数据呢。

恰好，tornado 提供比较好用的前端模板(tornado.template)。通过这个模板，能够让前端编写更方便。

render()

render() 方法能够告诉 tornado 读入哪个模板，插入其中的模板代码，并返回结果给浏览器。比如在 IndexHandler 类中 get() 方法里面的 `self.render("index.html")`，就是让 tornado 到 templates 目录中找到名为 index.html 的文件，读出它的内容，返回给浏览器。这样用户就能看到 index.html 所规定的页面了。当然，在前面所写的 index.html 还仅仅是 html 标记，没有显示出所谓“模板”的作用。为此，将 index.html 和 index.py 文件做如下改造。

```
#!/usr/bin/env Python
# coding=utf-8

import tornado.web
import methods.readdb as mrd

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        usernames = mrd.select_columns(table="users", column="username")
        one_user = usernames[0][0]
        self.render("index.html", user=one_user)
```

index.py 文件中，只修改了 get() 方法，从数据库中读取用户名，并且提出一个用户（one_user），然后通过 `self.render("index.html", user=one_user)` 将这个用户名放到 index.html 中，其中 `user=one_user` 的作用就是传递对象到模板。

提醒读者注意的是，在上面的代码中，我使用了 `mrd.select_columns(table="users",column="username")`，也就是说必须要在 methods 目录中的 readdb.py 文件中有一个名为 `select_columns` 的函数。为了使读者能够理解，贴出已经修改之后的 readdb.py 文件代码，比上一节多了函数 `select_columns`：

```
#!/usr/bin/env Python
# coding=utf-8

from db import *

def select_table(table, column, condition, value):
    sql = "select " + column + " from " + table + " where " + condition + "=" + value + ""
    cur.execute(sql)
    lines = cur.fetchall()
    return lines

def select_columns(table, column):
    sql = "select " + column + " from " + table
    cur.execute(sql)
    lines = cur.fetchall()
    return lines
```

下面是 index.html 修改后的代码：

```
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Learning Python</title>
</head>
<body>
    <h2> 登录页面</h2>
    <p>用用户名为: {{user}}登录</p>
    <form method="POST">
        <p><span>UserName:</span><input type="text" id="username"/></p>
        <p><span>Password:</span><input type="password" id="password" /></p>
        <p><input type="BUTTON" value="登录" id="login" /></p>
    </form>
    <script src="{{static_url('js/jquery.min.js')}}"></script>
    <script src="{{static_url('js/script.js')}}"></script>
</body>
```

`<p>用用户名为: {{user}}登录</p>`，这里用了 `{{ }}` 方式，接受对应的变量引导来的对象。也就是在首页打开之后，用户应当看到有一行提示。如下图一样。



图中箭头是我为了强调后来加上去的，箭头所指的，就是从数据库中读取出来的用户名，借助于模板中的双大括号 {{ }} 显示出来。

{{ }} 本质上是占位符。当这个 html 被执行的时候，这个位置会被一个具体的对象（例如上面就是字符串 qiwsir）所替代。具体是哪个具体对象替代这个占位符，完全是由 render() 方法中关键词来指定，也就是 render() 中的关键词与模板中的占位符包裹着的关键词一致。

用这种方式，修改一下用户正确登录之后的效果。要求用户正确登录之后，跳转到另外一个页面，并且在那个页面中显示出用户的完整信息。

先修改 url.py 文件，在其中增加一些内容。完整代码如下：

```
#!/usr/bin/env Python
# coding=utf-8
"""

the url structure of website
"""

import sys
reload(sys)
sys.setdefaultencoding("utf-8")

from handlers.index import IndexHandler
from handlers.user import UserHandler

url = [
    (r'/', IndexHandler),
```

```
(r'/user', UserHandler),
]
```

然后就建立 handlers/user.py 文件，内容如下：

```
#!/usr/bin/env Python
# coding=utf-8

import tornado.web
import methods.readdb as mrd

class UserHandler(tornado.web.RequestHandler):
    def get(self):
        username = self.get_argument("user")
        user_infos = mrd.select_table(table="users", column="*", condition="username", value=username)
        self.render("user.html", users = user_infos)
```

在 get() 中使用 `self.get_argument("user")`，目的是要通过 url 获取参数 user 的值。因此，当用户登录后，得到正确返回值，那么 js 应该用这样的方式载入新的页面。

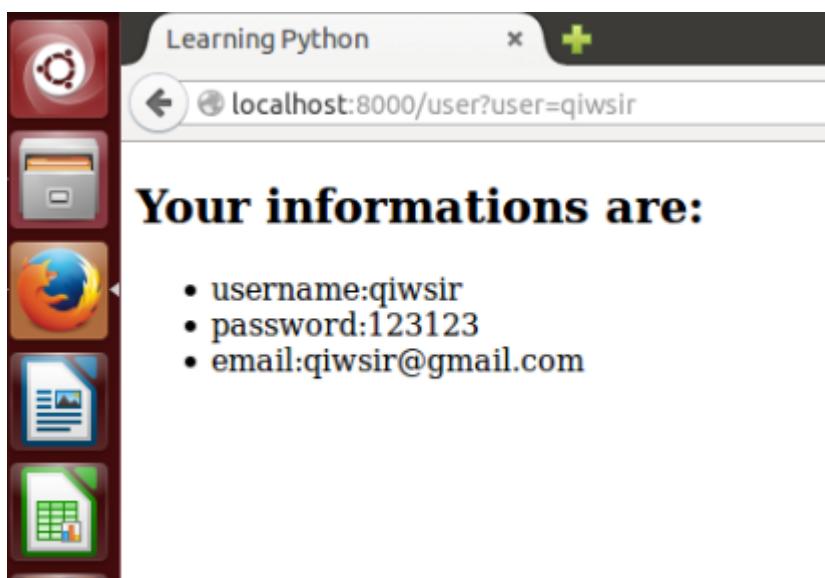
注意：上述的 user.py 代码为了简单突出本将要说明的，没有对 user_infos 的结果进行判断。在实际的编程中，这要进行判断或者使用 try...except。

```
$(document).ready(function(){
    $("#login").click(function(){
        var user = $("#username").val();
        var pwd = $("#password").val();
        var pd = {"username":user, "password":pwd};
        $.ajax({
            type:"post",
            url:"/",
            data:pd,
            cache:false,
            success:function(data){
                window.location.href = "/user?user="+data;
            },
            error:function(){
                alert("error!");
            },
        });
    });
});
```

接下来是 user.html 模板。注意上面的代码中，user_infos 引用的对象不是一个字符串了，也就是传入模板的不是一个字符串，是一个元组。对此，模板这样来处理它。

```
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Learning Python</title>
</head>
<body>
    <h2>Your informations are:</h2>
    <ul>
        {% for one in users %}
            <li>username:{{one[1]}}</li>
            <li>password:{{one[2]}}</li>
            <li>email:{{one[3]}}</li>
        {% end %}
    </ul>
</body>
```

显示的效果是：



在上面的模板中，其实用到了模板语法。

模板语法

在模板的双大括号中，可以写类似 Python 的语句或者表达式。比如：

```
>>> from tornado.template import Template
>>> print Template("{{ 3+4 }}").generate()
7
>>> print Template("{{ 'python'[0:2] }}").generate()
py
```

```
>>> print Template("{{ '-'join(str(i) for i in range(10)) }}").generate()
0-1-2-3-4-5-6-7-8-9
```

意即如果在模板中，某个地方写上 `{{ 3+4 }}` ，当那个模板被 `render()` 读入之后，在页面上该占位符的地方就显示 `7` 。这说明 tornado 自动将双大括号内的表达式进行计算，并将其结果以字符串的形式返回到浏览器输出。

除了表达式之外，Python 的语句也可以在表达式中使用，包括 `if`、`for`、`while` 和 `try`。只不过要有一个语句开始和结束的标记，用以区分那里是语句、哪里是 HTML 标记符。

语句的形式： `{{% 语句 %}}`

例如：

```
{{% if user=='qiwsir' %}}
  {{ user }}
{{% end %}}
```

上面的举例中，第一行虽然是 `if` 语句，但是不要在后面写冒号了。最后一行一定不能缺少，表示语句块结束。将这一个语句块放到模板中，当被 `render` 读取此模板的时候，tornado 将执行结果返回给浏览器显示，跟前面的表达式一样。实际的例子可以看上图输出结果和对应的循环语句。

转义字符

虽然读者现在已经对字符转义问题不陌生了，但是在网站开发中，它还将是一个令人感到麻烦的问题。所谓转义字符（Escape Sequence）也称字符实体(Character Entity)，它的存在是因为在网页中 `<, >` 之类的符号，是不能直接被输出的，因为它们已经被用作了 HTML 标记符了，如果在网页上用到它们，就要转义。另外，也有一些字符在 ASCII 字符集中没有定义（如版权符号“©”），这样的符号要在 HTML 中出现，也需要转义字符（如“©”对应的转义字符是“©”）。

上述是指前端页面的字符转义，其实不仅前端，在后端程序中，因为要读写数据库，也会遇到字符转义问题。

比如一个简单的查询语句： `select username, password from usertable where username='qiwsir'` ，如果在登录框中没有输入 `qiwsir`，而是输入了 `a;drop database;` ，这个查询语句就变成了 `select username, password from usertable where username=a; drop database;` ，如果后端程序执行了这条语句会怎么样呢？后果很严重，因为会 `drop database`，届时真的是欲哭无泪了。类似的情况还很多，比如还可以输入 `<input type="text">`，结果出现了一个输入框，如果是 `<form action="...">`，会造成跨站攻击了。这方面的问题还不少呢，读者有空可以到网上搜一下所谓 sql 注入问题，能了解更多。

所以，后端也要转义。

转义是不是很麻烦呢？

Tornado 为你着想了，因为存在以上转义问题，而且会有粗心的程序员忘记了，于是 Tornado 中，模板默认为自动转义。这是多么好的设计呀。于是所有表单输入的，你就不用担心会遇到上述问题了。

为了能够体会自动转义，不妨在登录框中输入上面那样字符，然后可以用 print 语句看一看，后台得到了什么。

print 语句，在 Python3 中是 print() 函数，在进行程序调试的时候非常有用。经常用它把要看个究竟的东西打印出来。

自动转义是一个好事情，但是，有时候会不需要转义，比如想在模板中这样做：

```
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Learning Python</title>
</head>
<body>
    <h2>登录页面</h2>
    <p>用用户名为：{{user}}登录</p>
    <form method="POST">
        <p><span>UserName:</span><input type="text" id="username"/></p>
        <p><span>Password:</span><input type="password" id="password" /></p>
        <p><input type="BUTTON" value="登录" id="login" /></p>
    </form>
    {% set website = "<a href='http://www.itdiffer.com'>welcome to my website</a>" %}
    {{ website }}
    <script src="{{static_url('js/jquery.min.js')}}"></script>
    <script src="{{static_url('js/script.js')}}"></script>
</body>
```

这是 index.html 的代码，我增加了 `{% set website = "welcome to my website" %}`，作用是设置一个变量，名字是 website，它对应的内容是一个做了超链接的文字。然后在下面使用这个变量 `{{ website }}`，本希望能够出现的是有一行字“welcome to my website”，点击这行字，就可以打开对应链接的网站。可是，看到了这个：



下面那一行，把整个源码都显示出来了。这就是因为自动转义的结果。这里需要的是不转义。于是可以将 {{ website }} 修改为：

```
{% raw website %}
```

表示这一行不转义。但是别的地方还是转义的。这是一种最推荐的方法。



如果你要全转义，可以使用：

```
{% autoescape None %}  
{{ website }}
```

貌似省事，但是我不推荐。

几个备查函数

下面几个函数，放在这里备查，或许在某些时候用到。都是可以使用在模板中的。

- `escape(s)`: 替换字符串 s 中的 &、<、> 为他们对应的 HTML 字符。
- `url_escape(s)`: 使用 `urllib.quote_plus` 替换字符串 s 中的字符为 URL 编码形式。
- `json_encode(val)`: 将 val 编码成 JSON 格式。
- `squeeze(s)`: 过滤字符串 s，把连续的多个空白字符替换成一个空格。

此外，在模板中也可以使用自己编写的函数。但不常用。所以本教程就不啰嗦这个了。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

用 tornado 做网站 (5)

模板继承

用前面的方法，已经能够很顺利地编写模板了。读者如果留心一下，会觉得每个模板都有相同的部分内容。在 Python 中，有一种被称之为“继承”的机制（请阅读本教程第贰季第肆章中的[类 \(4\)](#)中有关“继承”讲述]），它的作用之一就是能够让代码重用。

在 tornado 的模板中，也能这样。

先建立一个文件，命名为 base.html，代码如下：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Learning Python</title>
</head>
<body>
    <header>
        {% block header %}{% end %}</header>
    <content>
        {% block body %}{% end %}</content>
    <footer>
        {% set website = "<a href='http://www.itdiffer.com'>welcome to my website</a>" %}
        {% raw website %}
    </footer>
    <script src="{{static_url("js/jquery.min.js")}}></script>
    <script src="{{static_url("js/script.js")}}></script>
</body>
</html>
```

接下来就以 base.html 为父模板，依次改写已经有的 index.html 和 user.html 模板。

index.html 代码如下：

```
{% extends "base.html" %}

{% block header %}
```

```
<h2>登录页面</h2>
<p>用用户名为: {{user}}登录</p>
{% end %}
{% block body %}
<form method="POST">
<p><span>UserName:</span><input type="text" id="username"/></p>
<p><span>Password:</span><input type="password" id="password" /></p>
<p><input type="BUTTON" value="登录" id="login" /></p>
</form>
{% end %}
```

user.html 的代码如下：

```
{% extends "base.html" %}

{% block header %}
<h2>Your informations are:</h2>
{% end %}

{% block body %}
<ul>
{% for one in users %}
<li>username:{{one[1]}}</li>
<li>password:{{one[2]}}</li>
<li>email:{{one[3]}}</li>
{% end %}
</ul>
{% end %}
```

看以上代码，已经没有以前重复的部分了。`{% extends "base.html" %}` 意味着以 `base.html` 为父模板。在 `base.html` 中规定了形式如同 `{% block header %}{% end %}` 这样的块语句。在 `index.html` 和 `user.html` 中，分别对块语句中的内容进行了重写（或者说填充）。这就相当于在 `base.html` 中做了一个结构，在子模板中按照这个结构填内容。

CSS

基本上的流程已经差不多了，如果要美化前端，还需要使用 `css`，它的使用方法跟 `js` 类似，也是在静态目录中建立文件即可。然后把下面这句加入到 `base.html` 的 `<head></head>` 中：

```
<link rel="stylesheet" type="text/css" href="{{static_url('css/style.css')}}">
```

当然，要在 `style.css` 中写一个样式，比如：

```
body {
    color:red;
}
```

然后看看前端显示什么样子了，我这里是这样的：



关注字体颜色。

至于其它关于 CSS 方面的内容，本教程就不重点讲解了。读者可以参考关于 CSS 的资料。

至此，一个简单的基于 tornado 的网站就做好了，虽然它很丑，但是它很有前途。因为读者只要按照上述的讨论，可以在里面增加各种自己认为可以增加的内容。

建议读者在上述学习基础上，可以继续完成下面的几个功能：

- 用户注册
- 用户发表文章
- 用户文章列表，并根据文章标题查看文章内容
- 用户重新编辑文章

在后续教程内容中，也会涉及到上述功能。

cookie 和安全

cookie 是现在网站重要的内容，特别是当有用户登录的时候。所以，要了解 cookie。维基百科如是说：

Cookie (复数形態 Cookies)，中文名稱為小型文字檔案或小甜餅，指某些網站為了辨別用戶身份而儲存在用戶本地終端 (Client Side) 上的數據（通常經過加密）。定義於 RFC2109。是網景公司的前雇員 Lou Montulli 在 1993 年 3 月的發明。

关于 cookie 的作用，维基百科已经说的非常详细了（读者还能正常访问这么伟大的网站吗？）：

因为 HTTP 协议是无状态的，即服务器不知道用户上一次做了什么，这严重阻碍了交互式 Web 应用程序的实现。在典型的网上购物场景中，用户浏览了几个页面，买了一盒饼干和两瓶饮料。最后结帐时，由于 HTTP 的无状态性，不通过额外的手段，服务器并不知道用户到底买了什么。所以 Cookie 就是用来绕开 HTTP 的无状态性的“额外手段”之一。服务器可以设置或读取 Cookies 中包含信息，借此维护用户跟服务器会话中的状态。

在刚才的购物场景中，当用户选购了第一项商品，服务器在向用户发送网页的同时，还发送了一段 Cookie，记录着那项商品的信息。当用户访问另一个页面，浏览器会把 Cookie 发送给服务器，于是服务器知道他之前选购了什么。用户继续选购饮料，服务器就在原来那段 Cookie 里追加新的商品信息。结帐时，服务器读取发送来的 Cookie 就行了。

Cookie 另一个典型的应用是当登录一个网站时，网站往往会请求用户输入用户名和密码，并且用户可以勾选“下次自动登录”。如果勾选了，那么下次访问同一网站时，用户会发现没输入用户名和密码就已经登录了。这正是因为前一次登录时，服务器发送了包含登录凭据（用户名加密码的某种加密形式）的 Cookie 到用户的硬盘上。第二次登录时，（如果该 Cookie 尚未到期）浏览器会发送该 Cookie，服务器验证凭据，于是不必输入用户名和密码就让用户登录了。

和任何别的事物一样，cookie 也有缺陷，比如来自伟大的维基百科也列出了三条：

1. cookie 会被附加在每个 HTTP 请求中，所以无形中增加了流量。
2. 由于在 HTTP 请求中的 cookie 是明文传递的，所以安全性成问题。（除非用 HTTPS）
3. Cookie 的大小限制在 4KB 左右。对于复杂的存储需求来说是不够用的。

对于用户来讲，可以通过改变浏览器设置，来禁用 cookie，也可以删除历史的 cookie。但就目前而言，禁用 cookie 的可能不多了，因为她总要在网上买东西吧。

Cookie 最让人担心的还是由于它存储了用户的个人信息，并且最终这些信息要发给服务器，那么它就会成为某些人的目标或者工具，比如有 cookie 盗贼，就是搜集用户 cookie，然后利用这些信息进入用户账号，达到个人的某种不可告人之目的；还有被称之为 cookie 投毒的说法，是利用客户端的 cookie 传给服务器的机会，修改传回去的值。这些行为常常是通过一种被称为“跨站指令脚本(Cross site scripting)”（或者跨站指令码）的行为方式实现的。伟大的维基百科这样解释了跨站脚本：

跨网站脚本 (Cross-site scripting, 通常简称为 XSS 或跨站脚本或跨站脚本攻击) 是一种网站应用程序的安全漏洞攻击, 是代码注入的一种。它允许恶意用户将代码注入到网页上, 其他用户在观看网页时就会受到影响。这类攻击通常包含了 HTML 以及用户端脚本语言。

XSS 攻击通常指的是通过利用网页开发时留下的漏洞, 通过巧妙的方法注入恶意指令代码到网页, 使用户加载并执行攻击者恶意制造的网页程序。这些恶意网页程序通常是 JavaScript, 但实际上也可以包括 Java, VBS script, ActiveX, Flash 或者甚至是普通的 HTML。攻击成功后, 攻击者可能得到更高的权限 (如执行一些操作)、私密网页内容、会话和 cookie 等各种内容。

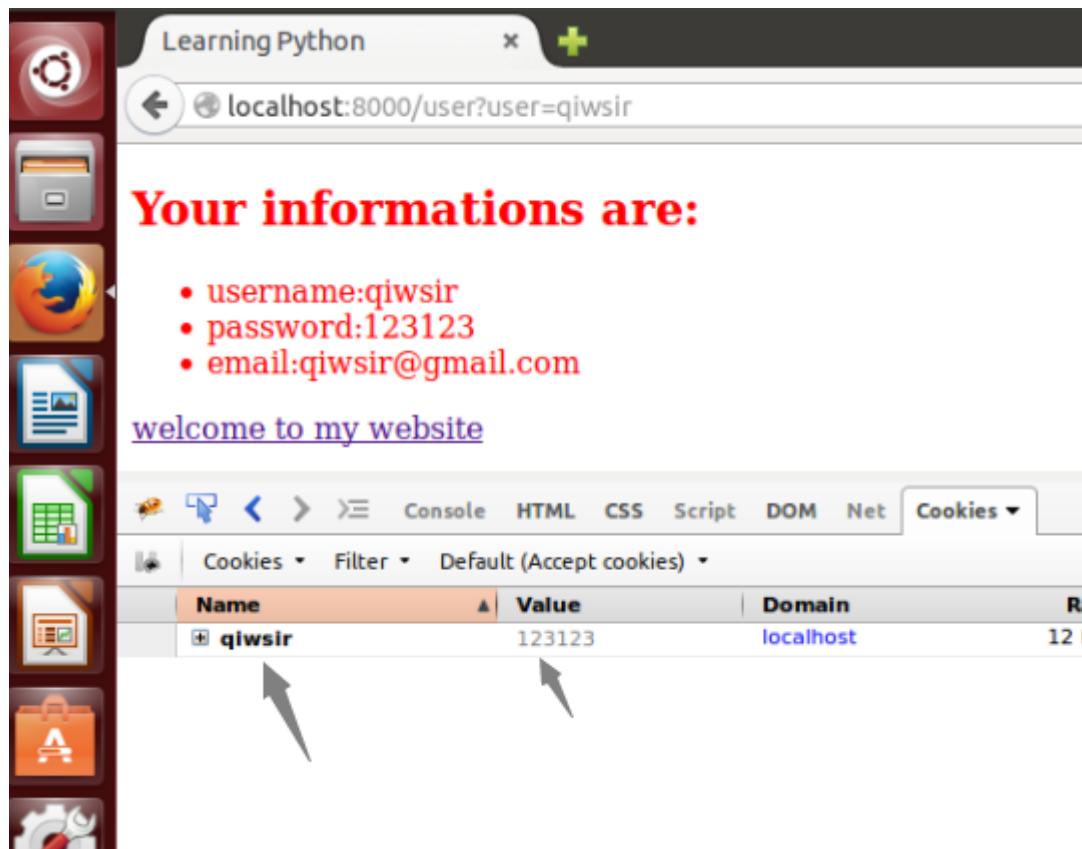
cookie 是好的, 被普遍使用。在 tornado 中, 也提供对 cookie 的读写函数。

`set_cookie()` 和 `get_cookie()` 是默认提供的两个方法, 但是它是明文不加密传输的。

在 index.py 文件的 IndexHandler 类的 post() 方法中, 当用户登录, 验证用户名和密码后, 将用户名和密码存入 cookie, 代码如下:

```
def post(self):
    username = self.get_argument("username")
    password = self.get_argument("password")
    user_infos = mrd.select_table(table="users", column="*", condition="username=%s", value=username)
    if user_infos:
        db_pwd = user_infos[0][2]
        if db_pwd == password:
            self.set_cookie(username, db_pwd)
            self.write(username)
        else:
            self.write("your password was not right.")
    else:
        self.write("There is no thi user.")
```

上面代码中, 较以前只增加了一句 `self.set_cookie(username, db_pwd)`, 在回到登录页面, 等候之后就成为:



看图中箭头所指，从左开始的第一个是用户名，第二个是存储的该用户密码。将我在登录时的密码就以明文的方式存储在 cookie 里面了。

明文存储，显然不安全。

tornado 提供另外一种安全的方法：set_secure_cookie() 和 get_secure_cookie()，称其为安全 cookie，是因为它以明文加密方式传输。此外，跟 set_cookie() 的区别还在于，set_secure_cookie() 执行后的 cookie 保存在磁盘中，直到它过期为止。也是因为这个原因，即使关闭浏览器，在失效时间之间，cookie 都一直存在。

要是用 set_secure_cookie() 方法设置 cookie，要先在 application.py 文件的 setting 中进行如下配置：

```
setting = dict(
    template_path = os.path.join(os.path.dirname(__file__), "templates"),
    static_path = os.path.join(os.path.dirname(__file__), "statics"),
    cookie_secret = "bZJc2sWbQLKos6GkHn/VB9oXwQt8S0R0kRvJ5/xJ89E=",
)
```

其中 `cookie_secret = "bZJc2sWbQLKos6GkHn/VB9oXwQt8S0R0kRvJ5/xJ89E="` 是为此增加的，但是，它并不是真正的加密，仅仅是一个障眼法罢了。

因为 tornado 会将 cookie 值编码为 Base-64 字符串，并增加一个时间戳和一个 cookie 内容的 HMAC 签名。所以，cookie_secret 的值，常常用下面的方式生成（这是一个随机的字符串）：

```
>>> import base64, uuid
>>> base64.b64encode(uuid.uuid4().bytes)
'w8yZud+kRHIP9uABEXaQiA=='
```

如果嫌弃上面的签名短，可以用 `base64.b64encode(uuid.uuid4().bytes + uuid.uuid4().bytes)` 获取。这里得到的是一个随机字符串，用它作为 cookie_secret 值。

然后修改 index.py 中设置 cookie 那句话，变成：

```
self.set_secure_cookie(username, db_pwd)
```

重新跑一个，看看效果。

The screenshot shows a web browser window with the URL `localhost:8000/user?user=qiwsir`. The page content displays "Your informations are:" followed by a list of user details: "username:qiwsir", "password:123123", and "email:qiwsir@gmail.com". Below this is a link "welcome to my website". To the right of the browser is a screenshot of the Firebug developer tools, specifically the Cookies panel. It lists a single cookie named "qiwsir" with the value "2|1:0|10:1432736714|6:qiwsir|8:MTIzMTIz|ecff9a4606197da3f2d9c4f72456a3c364f6ced73253d9bc42e4fe410a0ee320". An arrow points from the browser's cookie value to the corresponding row in the Firebug cookies list.

啊哈，果然“密”了很多。

如果要获取此 cookie，用 `self.get_secure_cookie(username)` 即可。

这是不是就安全了。如果这样就安全了，你太低估黑客们的技术实力了，甚至于用户自己也会修改 cookie 值。所以，还不安全。所以，又有了 `httponly` 和 `secure` 属性，用来防范 cookie 投毒。设置方法是：

```
self.set_secure_cookie(username, db_pwd, httponly=True, secure=True)
```

要获取 cookie，可以使用 `self.set_secure_cookie(username)` 方法，将这句放在 `user.py` 中某个适合的位置，并且可以用 `print` 语句打印出结果，就能看到变量 `username` 对应的 cookie 了。这时候已经不是那个“密”过的，是明文显示。

用这样的方法，浏览器通过 SSL 连接传递 cookie，能够在一定程度上防范跨站脚本攻击。

XSRF

XSRF 的含义是 Cross-site request forgery，即跨站请求伪造，也称之为"one click attack"，通常缩写成 CSRF 或者 XSRF，可以读作"sea surf"。这种对网站的攻击方式跟上面的跨站脚本（XSS）似乎相像，但攻击方式不一样。XSS 利用站点内的信任用户，而 XSRF 则通过伪装来自受信任用户的请求来利用受信任的网站。与 XSS 攻击相比，XSRF 攻击往往不大流行（因此对其进行防范的资源也相当稀少）和难以防范，所以被认为比 XSS 更具危险性。

读者要详细了解 XSRF，推荐阅读：[CSRF | XSRF 跨站请求伪造](#)

对于防范 CSRF 的方法，上面推荐阅读的文章中有明确的描述。还有一点需要提醒读者，就是在开发应用时需要深谋远虑。任何会产生副作用的 HTTP 请求，比如点击购买按钮、编辑账户设置、改变密码或删除文档，都应该使用 post() 方法。这是良好的 RESTful 做法。

另一个新名词：REST。这是一种 web 服务实现方案。伟大的维基百科中这样描述：

表徵性狀態傳輸（英文：Representational State Transfer，简称 REST）是 Roy Fielding 博士在 2000 年他的博士论文中提出来的一种软件架构风格。目前在三种主流的 Web 服务实现方案中，因为 REST 模式与复杂的 SOAP 和 XML-RPC 相比更加简洁，越来越多的 web 服务开始采用 REST 风格设计和实现。例如，Amazon.com 提供接近 REST 风格的 Web 服务进行图书查找；雅虎提供的 Web 服务也是 REST 风格的。

更详细的内容，读者可网上搜索来了解。

此外，在 tornado 中，还提供了 CSRF 保护的方法。

在 application.py 文件中，使用 xsrf_cookies 参数开启 CSRF 保护。

```
setting = dict(
    template_path = os.path.join(os.path.dirname(__file__), "templates"),
    static_path = os.path.join(os.path.dirname(__file__), "statics"),
    cookie_secret = "bZJc2sWbQLKos6GkHn/VB9oXwQt8S0R0kRvJ5/xJ89E=",
    xsrf_cookies = True,
)
```

这样设置之后，Tornado 将拒绝请求参数中不包含正确的 _xsrf 值的 post/put/delete 请求。tornado 会在后面悄悄地处理 _xsrf cookies，所以，在表单中也要包含 CSRF 令牌以确保请求合法。比如 index.html 的表单，修改如下：

```
{% extends "base.html" %}

{% block header %}
<h2>登录页面</h2>
<p>用用户名为：{{user}}登录</p>
{% end %}

{% block body %}
<form method="POST">
  {% raw xsrf_form_html() %}
  <p><span>UserName:</span><input type="text" id="username"/></p>
  <p><span>Password:</span><input type="password" id="password" /></p>
  <p><input type="BUTTON" value="登录" id="login" /></p>
</form>
{% end %}
```

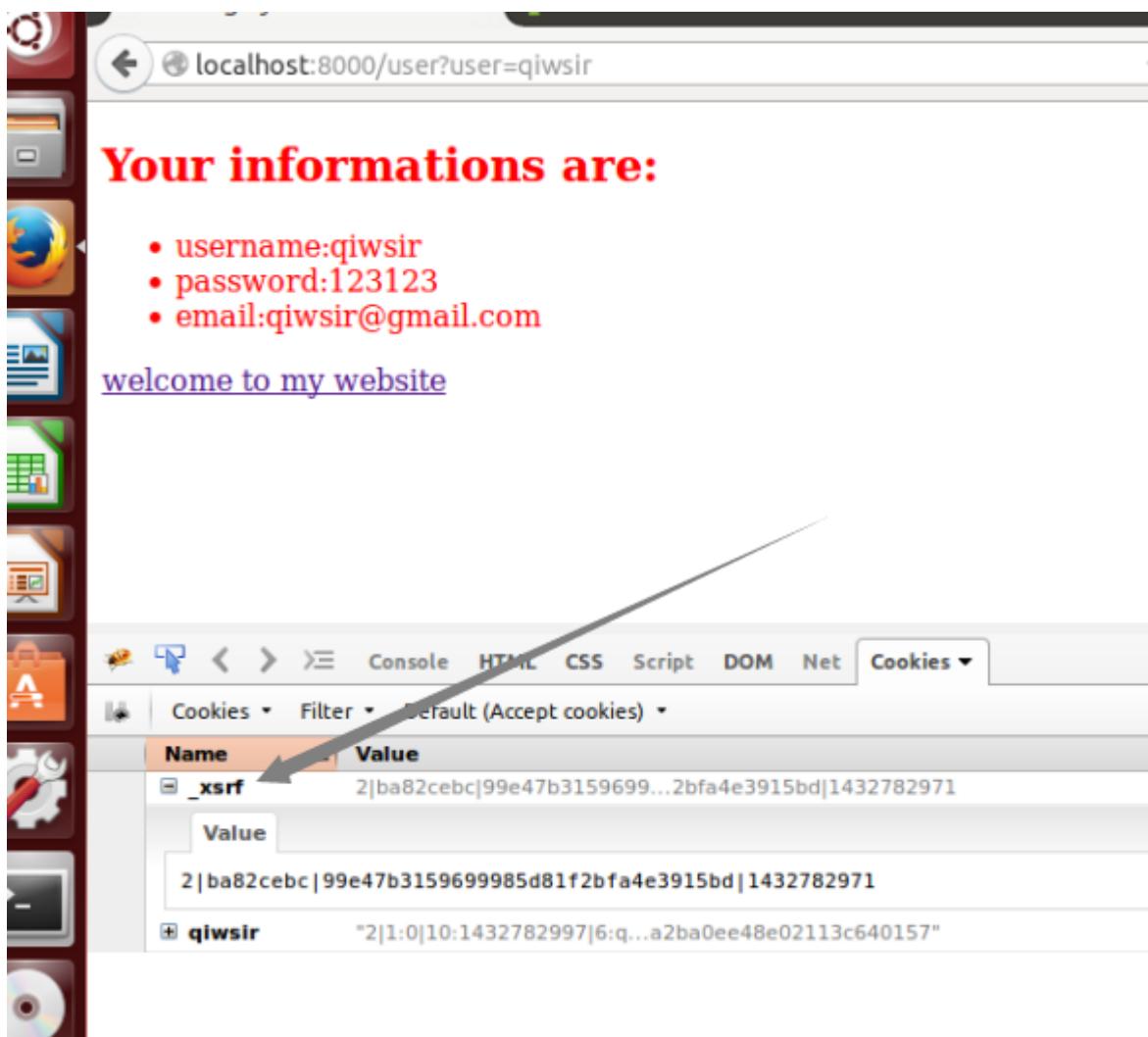
{% raw xsrf_form_html() %} 是新增的，目的就在于实现上面所说的授权给前端以合法请求。

前端向后端发送的请求是通过 ajax()，所以，在 ajax 请求中，需要一个 _xsrf 参数。

以下是 script.js 的代码

```
function getCookie(name){  
    var x = document.cookie.match("\b" + name + "=([^;]*)\b");  
    return x ? x[1]:undefined;  
}  
  
$(document).ready(function(){  
    $("#login").click(function(){  
        var user = $("#username").val();  
        var pwd = $("#password").val();  
        var pd = {"username":user, "password":pwd, "_xsrf":getCookie("_xsrf")};  
        $.ajax({  
            type:"post",  
            url:"/",  
            data:pd,  
            cache:false,  
            success:function(data){  
                window.location.href = "/user?user="+data;  
            },  
            error:function(){  
                alert("error!");  
            },  
        });  
    });  
});
```

函数 getCookie() 的作用是得到 cookie 值，然后将这个值放到向后端 post 的数据中 var pd = {"username":user, "password":pwd, "_xsrf":getCookie("_xsrf")};。运行的结果：



这是 tornado 提供的 CSRF 防护方法。是不是这样做就高枕无忧了呢？没这么简单。要做好一个网站，需要考虑的事情还很多。特别推荐阅读[WebAppSec/Secure Coding Guidelines](#)

常常听到人说做个网站怎么那么简单，客户用这种说辞来压低价格，老板用这种说辞来缩短工时成本，从上面的简单叙述中，你觉得网站还是随便几个页面就完事了吗？除非那个网站不是给人看的，是在那里摆着的。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

用 tornado 做网站 (6)

在[上一节](#)中已经对安全问题进行了描述，另外一个内容是不能忽略的，那就是用户登录之后，对当前用户状态（用户是否登录）进行判断。

用户验证

用户登录之后，当翻到别的目录中时，往往需要验证用户是否处于登录状态。当然，一种比较直接的方法，就是在转到每个目录时，都从 cookie 中把用户信息，然后传到后端，跟数据库验证。这不仅是直接的，也是基本的流程。但是，这个过程如果总让用户自己来做，框架的作用就显不出来了。tornado 就提供了一种用户验证方法。

为了后面更工程化地使用 tornado 编程。需要将前面的已经有的代码进行重新梳理。我只是将有修改的文件代码写出来，不做过多解释，必要的有注释，相信读者在前述学习基础上，能够理解。

在 handler 目录中增加一个文件，名称是 base.py，代码如下：

```
#! /usr/bin/env python
# coding=utf-8

import tornado.web

class BaseHandler(tornado.web.RequestHandler):
    def get_current_user(self):
        return self.get_secure_cookie("user")
```

在这个文件中，目前只做一个事情，就是建立一个名为 BaseHandler 的类，然后在里面放置一个方法，就是得到当前的 cookie。在这里特别要向读者说明，在这个类中，其实还可以写不少别的东西，比如你就可以将数据库连接写到这个类的初始化 `__init__()` 方法中。因为在其它的类中，我们要继承这个类。所以，这样一个架势，就为读者以后的扩展增加了冗余空间。

然后把 index.py 文件改写为：

```
#!/usr/bin/env Python
# coding=utf-8

import tornado.escape
import methods.readdb as mrd
from base import BaseHandler

class IndexHandler(BaseHandler): #继承 base.py 中的类 BaseHandler
```

```

def get(self):
    usernames = mrd.select_columns(table="users",column="username")
    one_user = usernames[0][0]
    self.render("index.html", user=one_user)

def post(self):
    username = self.get_argument("username")
    password = self.get_argument("password")
    user_infos = mrd.select_table(table="users",column="*",condition="username",value=username)
    if user_infos:
        db_pwd = user_infos[0][2]
        if db_pwd == password:
            self.set_current_user(username) #将当前用户名写入 cookie, 方法见下面
            self.write(username)
        else:
            self.write("-1")
    else:
        self.write("-1")

def set_current_user(self, user):
    if user:
        self.set_secure_cookie('user', tornado.escape.json_encode(user)) #注意这里使用了 tornado.escape.json_encode()
    else:
        self.clear_cookie("user")

class ErrorHandler(BaseHandler): #增加了一个专门用来显示错误的页面
    def get(self): #但是后面不单独讲述, 读者可以从源码中理解
        self.render("error.html")

```

在 index.py 的类 IndexHandler 中, 继承了 BaseHandler 类, 并且增加了一个方法 set_current_user() 用于将用户名写入 cookie。请读者特别注意那个 tornado.escape.json_encode() 方法, 其功能是:

tornado.escape.json_encode(value) JSON-encodes the given Python object.

如果要查看源码, 可以阅读: <http://www.tornadoweb.org/en/branch2.3/escape.html>

这样做的本质是把 user 转化为 json, 写入到了 cookie 中。如果从 cookie 中把它读出来, 使用 user 的值时, 还会用到:

tornado.escape.json_decode(value) Returns Python objects for the given JSON string

它们与 [json 模块中的 dump\(\)、load\(\)](#) 功能相仿。

接下来要对 user.py 文件也进行重写:

```

#!/usr/bin/env Python
# coding=utf-8

import tornado.web
import tornado.escape
import methods.readdb as mrd
from base import BaseHandler

class UserHandler(BaseHandler):
    @tornado.web.authenticated
    def get(self):
        #username = self.get_argument("user")
        username = tornado.escape.json_decode(self.current_user)
        user_infos = mrd.select_table(table="users", column="*", condition="username", value=username)
        self.render("user.html", users = user_infos)

```

在 `get()` 方法前面添加 `@tornado.web.authenticated`，这是一个装饰器，它的作用就是完成 `tornado` 的认证功能，即能够得到当前合法用户。在原来的代码中，用 `username = self.get_argument("user")` 方法，从 `url` 中得到当前用户名，现在把它注释掉，改用 `self.current_user`，这是和前面的装饰器配合使用的，如果它的值为假，就根据 `setting` 中的设置，寻找 `login_url` 所指定的目录（请关注下面对 `setting` 的配置）。

由于在 `index.py` 文件的 `set_current_user()` 方法中，是将 `user` 值转化为 `json` 写入 `cookie` 的，这里就得用 `username = tornado.escape.json_decode(self.current_user)` 解码。得到的 `username` 值，可以被用于后一句中的数据库查询。

`application.py` 中的 `setting` 也要做相应修改：

```

#!/usr/bin/env Python
# coding=utf-8

from url import url

import tornado.web
import os

setting = dict(
    template_path = os.path.join(os.path.dirname(__file__), "templates"),
    static_path = os.path.join(os.path.dirname(__file__), "statics"),
    cookie_secret = "bZJc2sWbQLKos6GkHn/VB9oXwQt8S0R0kRvJ5/xJ89E=",
    xsrf_cookies = True,
    login_url = '/',
)

application = tornado.web.Application(

```

```
handlers = url,  
**setting  
)
```

与以前代码的重要区别在于 `login_url = '/'`, 如果用户不合法, 根据这个设置, 会返回到首页。当然, 如果有单独的登录界面, 比如是 `/login`, 也可以 `login_url = '/login'`。

如此完成的是用户登录到网站之后, 在页面转换的时候实现用户认证。

为了演示本节的效果, 我对教程的源码进行修改。读者在阅读的时候, 可以参照源码。

总目录 (页 0)

如果你认为有必要打赏我, 请通过支付宝: qiwsir@126.com, 不胜感激。

用 tornado 做网站 (7)

到上一节结束，其实读者已经能够做一个网站了，但是，仅仅用前面的技术来做的网站，仅能算一个小网站，在《[为做网站而准备](#)》中，说明之所以选 tornado，就是因为它能够解决 c10k 问题，即能够实现大用户量访问。

要实现大用户量访问，必须要做的就是：异步。除非你是很土的土豪。

相关概念

同步和异步

有不少资料对这两个概念做了不同角度和层面的解释。在我看来，一个最典型的例子就是打电话和发短信。

- 打电话就是同步。张三给李四打电话，张三说：“是李四吗？”当这个信息被张三发出，提交给李四，就等待李四的响应（一般会听到“是”，或者“不是”），只有得到了李四返回的信息之后，才能进行后续的信息传送。
- 发短信是异步。张三给李四发短信，编辑了一句话“今晚一起看老齐的零基础学 Python”，发送给李四。李四或许马上回复，或许过一段时间，这段时间多长也不定，才回复。总之，李四不管什么时候回复，张三可以听到短信铃声为提示查看短信。

以上方式理解“同步”和“异步”不是很精准，有些地方或有牵强。要严格理解，需要用严格一点的定义表述（以下表述参照了[知乎](#)上的回答）：

同步和异步关注的是消息通信机制 (synchronous communication/ asynchronous communication)

所谓同步，就是在发出一个“调用”时，在没有得到结果之前，该“调用”就不返回。但是一旦调用返回，就得返回值了。换句话说，就是由“调用者”主动等待这个“调用”的结果。

而异步则是相反，“调用”在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在“调用”发出后，“被调用者”通过状态、通知来通知调用者，或通过回调函数处理这个调用。

可能还是前面的打电话和发短信更好理解。

阻塞和非阻塞

“阻塞和非阻塞”与“同步和异步”常常被换为一谈，其实它们之间还是有差别的。如果按照一个“差不多”先生的思维方法，你也可以不那么深究它们之间的学理上的差距，反正在你的程序中，会使用就可以了。不过，必要的严谨还是需要的，特别是我写这个教程，要装扮的让别人看来自己懂，于是就再引用知乎上的说明（我个人认为，别人已经做的挺好的东西，就别重复劳动了，“拿来主义”，也不错。或许你说我抄袭和山寨，但是我明确告诉你来源了）：

阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态。

阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

按照这个说明，发短信就是显然的非阻塞，发出去一条短信之后，你利用手机还可以干别的，乃至于再发一条“老齐的课程没意思，还是看 PHP 刺激”也是可以的。

关于这两组基本概念的辨析，不是本教程的重点，读者可以参阅这篇文章：<http://www.cppblog.com/conversation/archive/2009/05/13/82879.html>，文章作者做了细致入微的辨析。

tornado 的同步

此前，在 tornado 基础上已经完成的 web，就是同步的、阻塞的。为了更明显的感受这点，不妨这样试一试。

在 handlers 文件夹中建立一个文件，命名为 sleep.py

```
#!/usr/bin/env python
# coding=utf-8

from base import BaseHandler

import time

class SleepHandler(BaseHandler):
    def get(self):
        time.sleep(17)
        self.render("sleep.html")

class SeeHandler(BaseHandler):
    def get(self):
        self.render("see.html")
```

其它的事情，如果读者对我在《用 tornado 做网站 (1)》中所讲述的网站框架熟悉，应该知道如何做了，不熟悉，请回头复习。

sleep.html 和 see.html 是两个简单的模板，内容可以自己写。别忘记修改 url.py 中的目录。

然后的测试稍微复杂一点点，就是打开浏览器之后，打开两个标签，分别在两个标签中输入 `localhost:8000/sleep`（记为标签 1）和 `localhost:8000/see`（记为标签 2），注意我用的是 8000 端口。输入之后先不要点击回车去访问。做好准备，记住切换标签可以用“ctrl-tab”组合键。

1. 执行标签 1，让它访问网站；
2. 马上切换到标签 2，访问网址。
3. 注意观察，两个标签页面，是不是都在显示正在访问，请等待。
4. 当标签 1 不呈现等待提示（比如一个正在转的圆圈）时，标签 2 的表现如何？几乎同时也访问成功了。

建议读者修改 sleep.py 中的 `time.sleep(17)` 这个值，多试试。很好玩的吧。

当然，这是比较笨拙的方法，本来是可以通过测试工具完成上述操作比较的。怎奈要用别的工具，还要进行介绍，又多了一个分散精力的东西，故用如此笨拙的方法，权当有一个体会。

异步设置

tornado 本来就是一个异步的服务框架，体现在 tornado 的服务器和客户端的网络交互的异步上，起作用的是 `tornado.ioloop.IOLoop`。但是如果的客户端请求服务器之后，在执行某个方法的时候，比如上面的代码中执行 `get()` 方法的时候，遇到了 `time.sleep(17)` 这个需要执行时间比较长的操作，耗费时间，就会使整个 tornado 服务器的性能受限了。

为了解决这个问题，tornado 提供了一套异步机制，就是异步装饰器 `@tornado.web.asynchronous`：

```
#!/usr/bin/env Python
# coding=utf-8

import tornado.web
from base import BaseHandler

import time

class SleepHandler(BaseHandler):
    @tornado.web.asynchronous
    def get(self):
        tornado.ioloop.IOLoop.instance().add_timeout(time.time() + 17, callback=self.on_response)
```

```
def on_response(self):
    self.render("sleep.html")
    self.finish()
```

将 sleep.py 的代码如上述一样改造，即在 get() 方法前面增加了装饰器 `@tornado.web.asynchronous`，它的作用在于将 tornado 服务器本身默认的设置 `_auto_finish` 值修改为 `false`。如果不用这个装饰器，客户端访问服务器的 get() 方法并得到返回值之后，两只之间的连接就断开了，但是用了 `@tornado.web.asynchronous` 之后，这个连接就不关闭，直到执行了 `self.finish()` 才关闭这个连接。

`tornado.ioloop.IOLoop.instance().add_timeout()` 也是一个实现异步的函数，`time.time() + 17` 是给前面函数提供一个参数，这样实现了相当于 `time.sleep(17)` 的功能，不过，还没有完成，当这个操作完成之后，就执行回调函数 `on_response()` 中的 `self.render("sleep.html")`，并关闭连接 `self.finish()`。

过程清楚了。所谓异步，就是要解决原来的 `time.sleep(17)` 造成的服务器处理时间长，性能下降的问题。解决方法如上描述。

读者看这个代码，或许感觉有点不是很舒服。如果有这么一点感觉，是正常的。因为它里面除了装饰器之外，用到了一个回调函数，它让代码的逻辑不是平铺下去，而是被分割为了两段。第一段是 `tornado.ioloop.IOLoop.instance().add_timeout(time.time() + 17, callback=self.on_response)`，用 `callback=self.on_response` 来使用回调函数，并没有如同改造之前直接 `self.render("sleep.html")`；第二段是回调函数 `on_response(self)`，要在这个函数里面执行 `self.render("sleep.html")`，并且以 `self.finish()` 结尾以关闭连接。

这还是执行简单逻辑，如果复杂了，不断地要进行“回调”，无法让逻辑顺利延续，那面会“眩晕”了。这种现象被业界成为“代码逻辑拆分”，打破了原有逻辑的顺序性。为了让代码逻辑不至于被拆分的七零八落，于是就出现了另外一种常用的方法：

```
#!/usr/bin/env Python
# coding=utf-8

import tornado.web
import tornado.gen
from base import BaseHandler

import time

class SleepHandler(tornado.web.RequestHandler):
    @tornado.gen.coroutine
    def get(self):
        yield tornado.gen.Task(tornado.ioloop.IOLoop.instance().add_timeout, time.time() + 17)
        #yield tornado.gen.sleep(17)
        self.render("sleep.html")
```

从整体上看，这段代码避免了回调函数，看着顺利多了。

再看细节部分。

首先使用的是 `@tornado.gen.coroutine` 装饰器，所以要在前面有 `import tornado.gen`。跟这个装饰器类似的是 `@tornado.gen.engine` 装饰器，两者功能类似，有一点细微差别。请阅读[官方对此的解释](#)：

This decorator(指 engine) is similar to coroutine, except it does not return a Future and the callback argument is not treated specially.

`@tornado.gen.engine` 是古时候用的，现在我们都使用 `@tornado.gen.coroutine` 了，这个是在 tornado 3.0 以后开始。在网上查阅资料的时候，会遇到一些使用 `@tornado.gen.engine` 的，但是在你使用或者借鉴代码的时候，就勇敢地将其修改为 `@tornado.gen.coroutine` 好了。有了这个装饰器，就能够控制下面的生成器的流程了。

然后就看到 `get()` 方法里面的 `yield` 了，这是一个生成器（参阅本教程《[生成器](#)》）。返回后，最后使用 `yield` 得到了一个生成器，先把流程挂起，等完全完毕，再唤醒继续执行。要提醒读者，生成器都是异步的。

其实，上面啰嗦一对，可以用代码中注释了的一句话来代替 `yield tornado.gen.sleep(17)`，之所以扩所，就是为了顺便看到 `tornado.gen.Task()` 方法，因为如果读者在看古老的代码时候，会遇到。但是，后面你写的时候，就不要那么啰嗦了，请用 `yield tornado.gen.sleep()`。

至此，基本上对 tornado 的异步设置有了概览，不过，上面的程序在实际中没有什么价值。在工程中，要让 tornado 网站真正异步起来，还要做很多事情，不仅仅是如上面的设置，因为很多东西，其实都不是异步的。

实践中的异步

以下各项同步（阻塞）的，如果在 tornado 中按照之前的方式只用它们，就是把 tornado 的非阻塞、异步优势削减了。

- 数据库的所有操作，不管你的数据是 SQL 还是 noSQL，`connect`、`insert`、`update` 等
- 文件操作，打开，读取，写入等
- `time.sleep`，在前面举例中已经看到了
- `smtplib`，发邮件的操作
- 一些网络操作，比如 tornado 的 `httpclient` 以及 `pycurl` 等

除了以上，或许在编程实践中还会遇到其他的同步、阻塞实践。仅仅就上面几项，就是编程实践中经常会遇到的，怎么解决？

聪明的大牛程序员帮我们做了扩展模块，专门用来实现异步/非阻塞的。

- 在数据库方面，由于种类繁多，不能一一说明，比如 mysql，可以使用[adb](#)模块来实现 python 的异步 mysql 库；对于 mongodb 数据库，有一个非常优秀的模块，专门用于在 tornado 和 mongodb 上实现异步操作，它就是 motor。特别贴出它的 logo，我喜欢。官方网站：<http://motor.readthedocs.org/en/stable/>上的安装和使用方法都很详细。



- 文件操作方面也没有替代模块，只能尽量控制好 IO，或者使用内存型（Redis）及文档型（MongoDB）数据库。
- time.sleep() 在 tornado 中有替代：`tornado.gen.sleep()` 或者 `tornado.ioloop.IOLoop.instance().add_timeout`，这在前面代码已经显示了。
- smtp 发送邮件，推荐改为 `tornado-smtp-client`。
- 对于网络操作，要使用 `tornado.httpclient.AsyncHTTPClient`。

其它的解决方法，只能看到问题具体说了，甚至没有很好的解决方法。不过，这里有一个列表，列出了足够多的库，供使用者选择：[Async Client Libraries built on tornado.ioloop](#)，同时这个页面里面还有很多别的链接，都是很好的资源，建议读者多看看。

教程到这里，读者是不是要思考一个问题，既然对于 mongodb 有专门的 motor 库来实现异步，前面对于 tornado 的异步，不管是哪个装饰器，都感觉麻烦，有没有专门的库来实现这种异步呢？这不是异想天开，还真有。也应该有，因为这才体现python的特点。比如[greenlet-tornado](#)，就是一个不错的库。读者可以浏览官方网站深入了解（为什么对 mysql 那么不积极呢？按理说应该出来好多支持 mysql 异步的库才对）。

必须声明，前面演示如何在 tornado 中设置异步的代码，仅仅是演示理解设置方法。在工程实践中，那个代码的意义不到。为此，应该有一个近似于实践的代码示例。是的，的确应该有。当我正要写这样的代码时候，在网上发现一篇文章，这篇文章阻止了我写，因为我要写的那篇文章的作者早就写好了，而且我认为表述非常到位，示例也详细。所以，我不得不放弃，转而推荐给读者这篇好文章：

举例：<http://emptysqua.re/blog/refactoring-tornado-coroutines/>

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

HTML



TP



11

科学计算



unity



HTML



为计算做准备

闲谈

计算机姑娘是擅长进行科学计算的，本来她就是做这个的，只不过后来人们让她处理了很多文字内容罢了，以至于现在有一些人认为她是用来打字写文章的（变成打字机了），忘记了她最擅长的计算。

每种编程语言都能用来做计算，区别在于编程过程中，是否有足够的工具包供给。比如用汇编，就得自己多劳动，如果用 Fortran，就方便得多了。不知道读者是否听说过 Fortran，貌似古老，现在仍被使用。（以下引文均来自维基百科）

Fortran 语言是为了满足数值计算的需求而发展出来的。1953 年 12 月，IBM 公司工程师约翰·巴科斯（J. Backus）因深深体会编写程序很困难，而写了一份备忘录给董事长斯伯特·赫德（Cuthbert Hurd），建议论为 IBM704 系统设计全新的电脑语言以提升开发效率。当时 IBM 公司的顾问冯·诺伊曼强烈反对，因为他任认为不切实际而且根本不必要。但赫德批准了这项计划。1957 年，IBM 公司开发出第一套 FORTRAN 语言，在 IBM704 电脑上运作。历史上第一支 FORTRAN 程式在马里兰州的西屋贝地斯核电厂实验室。1957 年 4 月 20 日星期五的下午，一位 IBM 软件工程师决定在电厂内编译第一支 FORTRAN 程式，当程式码输入后，经过编译，印表机列出一行讯息：“原始程式错误……右侧括号后面没有逗号”，这让现场人员都感到讶异，修正这个错误后，印表机输出了正确结果。而西屋电器公司因此意外地成为 FORTRAN 的第一个商业用户。1958 年推出 FORTRAN II，几年后又推出 FORTRAN III，1962 年推出 FORTRAN IV 后，开始广泛被使用。目前最新版是 Fortran 2008。

还有一个广为应用的不得不说，那就是 matlab，一直以来被人称赞。

MATLAB（矩阵实验室）是 MATrix LABoratory 的缩写，是一款由美国 The MathWorks 公司出品的商业数学软件。MATLAB 是一种用于算法开发、数据可视化、数据分析以及数值计算的高级技术计算语言和交互式环境。除了矩阵运算、绘制函数/数据图像等常用功能外，MATLAB 还可以用来创建用户界面及与调用其它语言（包括 C,C++,Java,Python 和 FORTRAN）编写的程序。

但是，它是收费的商业软件，虽然在某国这个无所谓。

还有 R 语言，也是在计算领域被多多使用的。

R 语言，一种自由软件程式语言与操作环境，主要用于统计分析、绘图、数据挖掘。R 本來是由来自新西兰奥克兰大学的 Ross Ihaka 和 Robert Gentleman 开发（也因此称为 R），现在由“R 开发核心团队”负责开发。R 是基于 S 语言的一个 GNU 计划项目，所以也可以当作 S 语言的一种实现，通常用 S 语言编写的代码都可以不作修改的在 R 环境下运行。R 的语法是來自 Scheme。

最后要说的就是 Python，近几年使用 Python 的领域不断扩张，包括在科学计算领域，它已经成为了一种趋势。在这个过程中，虽然有不少人诟病 Python 的这个慢那个解释动态语言之类（这种说法是值得讨论的），但是，依然无法阻挡 Python 在科学计算领域大行其道。之所以这样，就是因为它是 Python。

- 开源，就这一条就已经足够了，一定要用开源的东西。至于为什么，本教程前面都阐述过了。
- 因为开源，所以有非常棒的社区，里面有相当多支持科学计算的库，不用还等待何时？
- 简单易学，这点对那些不是专业程序员来讲非常重要。我就接触到一些搞天文学和生物学的研究者，他们正在使用 Python 进行计算。
- 在科学计算上如果用了 Python，能够让数据跟其它的比如 web 无缝对接，这不是很好的吗？

当然，最重要一点，就是本教程是讲 Python 的，所以，在科学计算这块肯定不会讲 Fortran 或者 R，一定得是 Python。

安装

如果读者使用 Ubuntu 或者 Debian，可以这样来安装：

```
sudo apt-get install Python-numpy Python-scipy Python-matplotlib ipython ipython-notebook Python-pandas Python-
```

一股脑把可能用上的都先装上。在安装的时候，如果需要其它的依赖，你会明显看到的。

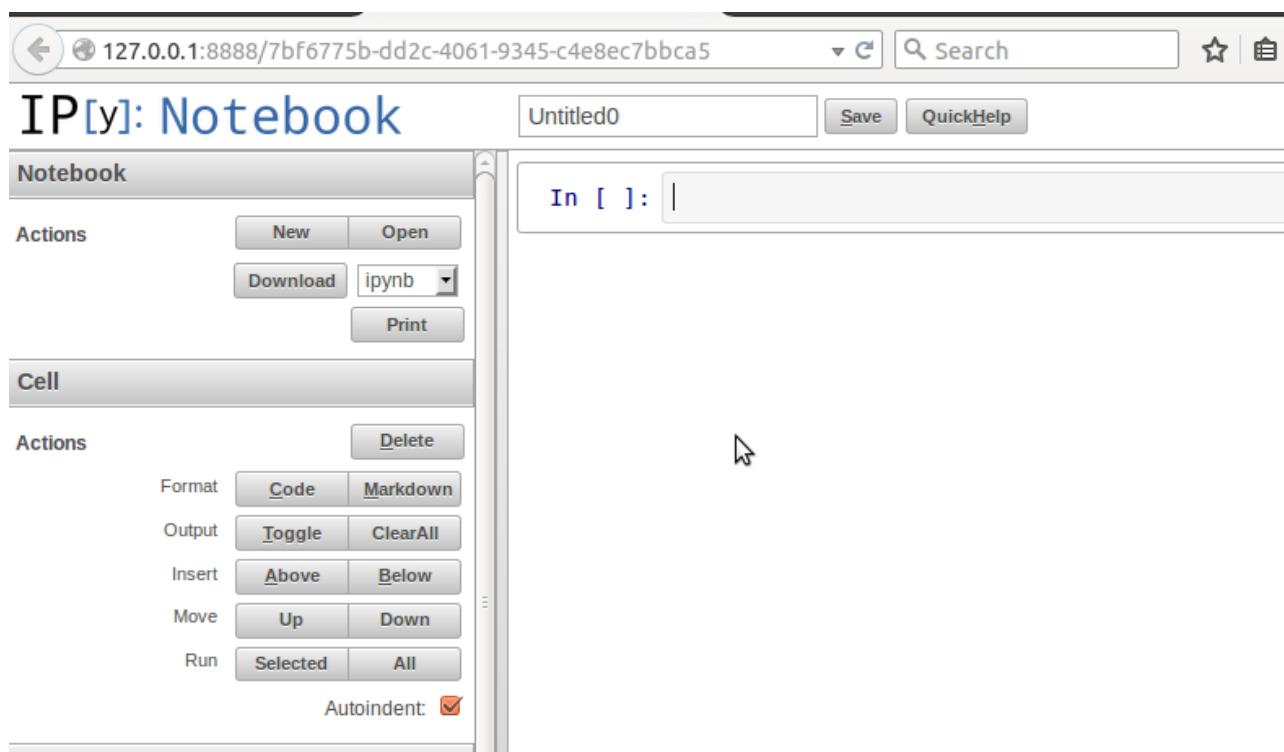
如果是别的系统，比如 windows 类，请自己网上查找安装方法吧，这里内容不少，最权威的是看官方网站列出的安装：<http://www.scipy.org/install.html>

基本操作

在科学计算中，业界比较喜欢使用 ipython notebook，前面已经安装。在 shell 中执行

```
ipython notebook --pylab=inline
```

得到下图的界面，这是在浏览器中打开的：



在 In 后面的编辑区，可以写 Python 语句。然后按下 SHIFT+ENTER 或者 CTRL+ENTER 就能执行了，如果按下 ENTER，不是执行，是在当前编辑区换行。

The screenshot shows the notebook interface with several cells. Cell [1] contains the input 'In [1]: 2+3' and the output 'Out[1]: 5'. Cell [2] contains the input 'In [2]: print "hello world"' and the output 'hello world'. A new cell 'In []:' is visible at the bottom.

```

Untitled0
Save QuickHelp

In [1]: 2+3
Out[1]: 5

In [2]: print "hello world"
hello world

In [ ]:

```

Ipython Notebook 是一个非常不错的编辑器，执行之后，直接显示出来输入内容和输出的结果。当然，错误是难免的，它会：

```
In [9]: import random  
a = [random.randint(10)]  
print a
```

↳

```
-----  
TypeError Traceback (most recent call last)  
/home/qw/Documents/StarterLearningPython/<ipython-input-9-44c3f4700939> in <module>()  
  1 import random  
----> 2 a = [random.randint(10)]  
  3 print a  
  
TypeError: randint() takes exactly 3 arguments (2 given)
```

注意观察图中的箭头所示，直接标出有问题的行。返回编辑区，修改之后可继续执行。

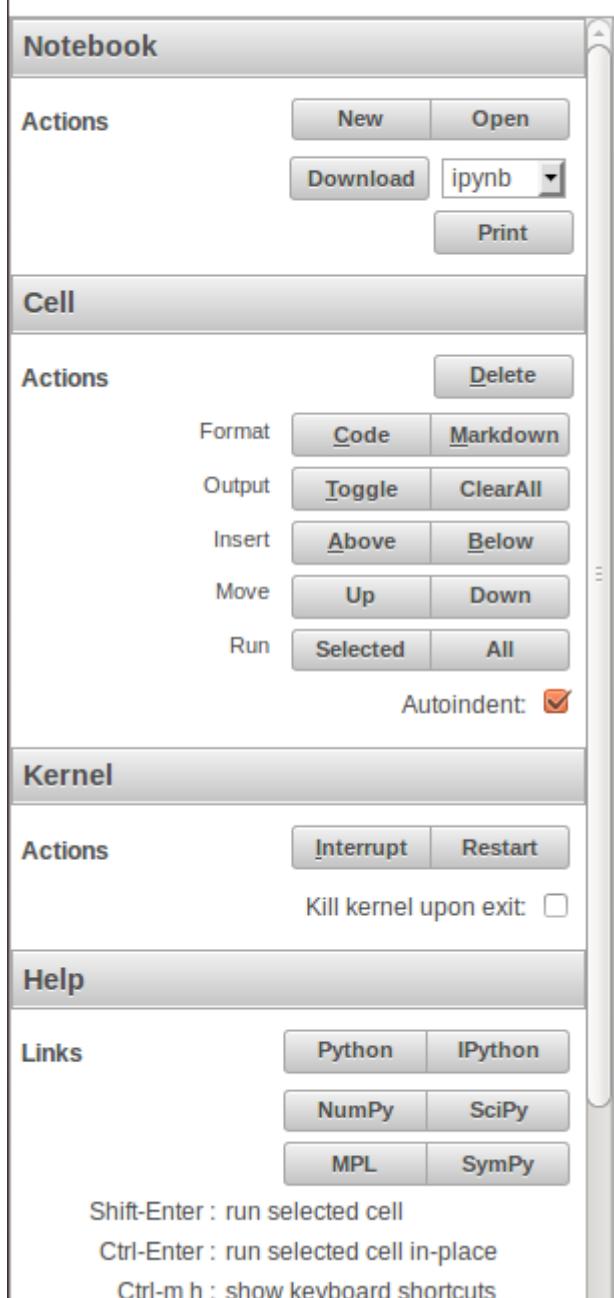
```
In [10]: import random  
a = [random.randint(1,10)]  
print a
```

↳

[10]

不要忽视左边的辅助操作，能够让你在使用 ipython notebook 的时候更方便。

IP[y]: Notebook



除了在网页中之外，如果你已经喜欢上了 Python 的交互模式，特别是你用的计算机中有一个 shell 的东西，更是棒了。于是可以：

```
$ ipython
```

进入了一个类似于 Python 的交互模式中，如下所示：

```
In [1]: print "hello, pandas"
```

```
hello, pandas
```

In [2]:

或者说 ipython 同样是一个不错的交互模式。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我, 请通过支付宝: qiwsir@126.com, 不胜感激。

Pandas 使用 (1)

Pandas 是基于 NumPy 的一个非常好用的库，正如名字一样，人见人爱。之所以如此，就在于不论是读取、处理数据，用它都非常简单。

基本的数据结构

Pandas 有两种自己独有的基本数据结构。读者应该注意的是，它固然有着两种数据结构，因为它依然是 Python 的一个库，所以，Python 中有的数据类型在这里依然适用，也同样还可以使用类自己定义数据类型。只不过，Pandas 里面又定义了两种数据类型：Series 和 DataFrame，它们让数据操作更简单了。

以下操作都是基于：

```
In [1]: from pandas import Series, DataFrame
        import pandas as pd
```

为了省事，后面就不在显示了。并且如果你跟我一样是使用 ipython notebook，只需要开始引入模块即可。

Series

Series 就如同列表一样，一系列数据，每个数据对应一个索引值。比如这样一个列表：[9, 3, 8]，如果跟索引值写到一起，就是：

data	9	3	8
index	0	1	2

这种样式我们已经熟悉了，不过，在有些时候，需要把它竖过来表示：

index	data
0	9
1	3
2	8

上面两种，只是表现形式上的差别罢了。

Series 就是“竖起来”的 list：

```
In [2]: s = Series([100, "PYTHON", "Soochow", "Qiwsir"])
s
```

```
Out[2]: 0      100
        1    PYTHON
        2   Soochow
        3    Qiwsir
```

另外一点也很像列表，就是里面的元素的类型，由你任意决定（其实是由需要来决定）。

这里，我们实质上创建了一个 Series 对象，这个对象当然就有其属性和方法了。比如，下面的两个属性依次可以显示 Series 对象的数据值和索引：

```
In [3]: s.values
```

```
Out[3]: array([100, 'PYTHON', 'Soochow', 'Qiwsir'], dtype=object)
```

```
In [4]: s.index
```

```
Out[4]: Int64Index([0, 1, 2, 3], dtype=int64)
```

列表的索引只能是从 0 开始的整数，Series 数据类型在默认情况下，其索引也是如此。不过，区别于列表的是，Series 可以自定义索引：

```
s2 = Series([100, "PYTHON", "Soochow", "Qiwsir"], index=["mark", "title", "university", "name"])
s2
```

mark	100
title	PYTHON
university	Soochow
name	Qiwsir

```
In [6]: s2.index
```

```
Out[6]: Index(['mark', 'title', 'university', 'name'], dtype=object)
```

自定义索引，的确比较有意思。就凭这个，也是必须的。

每个元素都有了索引，就可以根据索引操作元素了。还记得 list 中的操作吗？Series 中，也有类似的操作。先看简单的，根据索引查看其值和修改其值：

```
In [7]: s2["name"]
```

```
Out[7]: 'Qiwsir'
```

```
In [8]: s2["name"] = "AOI"
```

```
In [9]: s2
```

```
Out[9]: mark          100
         title        PYTHON
         university   Soochow
         name          AOI
```

这是不是又有点类似 dict 数据了呢？的确如此。看下面理解了。

读者是否注意到，前面定义 Series 对象的时候，用的是列表，即 Series() 方法的参数中，第一个列表就是其数据值，如果需要定义 index，放在后面，依然是一个列表。除了这种方法之外，还可以用下面的方法定义 Series 对象：

```
In [15]: sd = {"python":8000, "c++":8100, "c#":4000}
         s4 = Series(sd)
         s4
```

```
Out[15]: c#      4000
         c++    8100
         python 8000
```

现在是否理解为什么前面那个类似 dict 了？因为本来就是可以这样定义的。

这时候，索引依然可以自定义。Pandas 的优势在这里体现出来，如果自定义了索引，自定的索引会自动寻找原来的索引，如果一样的，就取原来索引对应的值，这个可以简称为“自动对齐”。

```
In [19]: s6 = Series(sd, index=["java", "python", "c++", "c#"])
         s6
```

```
Out[19]: java      NaN
         python   8000
         c++     8100
         c#      4000
```

在 sd 中，只有 'python':8000, 'c++':8100, 'c#':4000，没有"java"，但是在索引参数中有，于是其它能够“自动对齐”的照搬原值，没有的那个"java"，依然在新 Series 对象的索引中存在，并且自动为其赋值 NaN。在 Pandas 中，如果没有值，都对齐赋给 NaN。来一个更特殊的：

```
In [17]: ilst = ["java", "perl"]
s5 = Series(sd, index=ilst)
s5
```

```
Out[17]: java      NaN
          perl     NaN
```

新得到的 Series 对象索引与 sd 对象一个也不对应，所以都是 NaN。

Pandas 有专门的方法来判断值是否为空。

```
In [20]: pd.isnull(s6)
```

```
Out[20]: java        True
          python     False
          C++        False
          C#         False
```

```
In [21]: pd.notnull(s6)|
```

```
Out[21]: java        False
          python     True
          C++        True
          C#         True
```

此外，Series 对象也有同样的方法：

```
In [22]: s6.isnull()
```

```
Out[22]: java        True
          python     False
          C++        False
          C#         False
```

其实，对索引的名字，是可以从新定义的：

```
In [38]: s6.index = ["p1", "p2", "p3", "p4"]
s6
```

```
Out[38]: p1      NaN
          p2      8000
          p3      8100
          p4      4000
```

对于 Series 数据，也可以做类似下面的运算（关于运算，后面还要详细介绍）：

```
In [10]: s3 = Series([3,9,4,7], index=['a','b','c','d'])
s3
```

```
Out[10]: a    3
          b    9
          c    4
          d    7
```

```
In [11]: s3[s3 > 5]
```

```
Out[11]: b    9
          d    7
```

```
In [12]: s3 * 5
```

```
Out[12]: a    15
          b    45
          c    20
          d    35
```

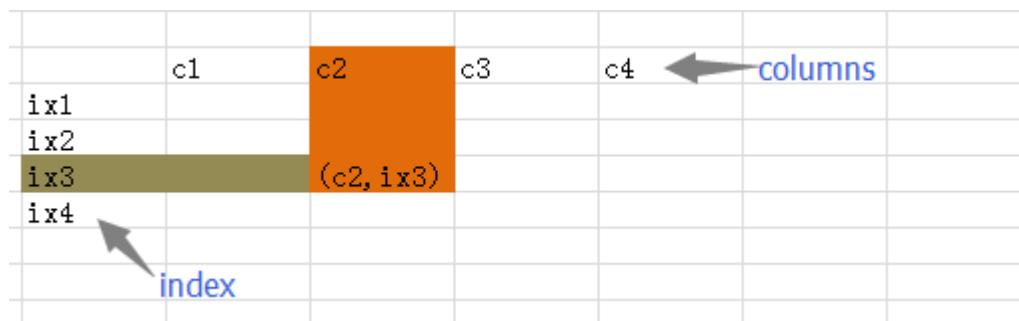
```
In [28]: s5 + s6
```

```
Out[28]: c#      NaN
          c++     8188
          java    NaN
          python  8030
```

上面的演示中，都是在 ipython notebook 中进行的，所以截图了。在学习 Series 数据类型同时了解了 ipython notebook。对于后面的所有操作，读者都可以在 ipython notebook 中进行。但是，我的讲述可能会在 Python 交互模式中进行。

DataFrame

DataFrame 是一种二维的数据结构，非常接近于电子表格或者类似 mysql 数据库的形式。它的竖行称之为 columns，横行跟前面的 Series 一样，称之为 index，也就是说可以通过 columns 和 index 来确定一个主句的位置。（有人把 DataFrame 翻译为“数据框”，是不是还可以称之为“筐”呢？向里面装数据嘛。）



下面的演示，是在 Python 交互模式下进行，读者仍然可以在 ipython notebook 环境中测试。

```
>>> import pandas as pd
>>> from pandas import Series, DataFrame

>>> data = {"name":["yahoo","google","facebook"], "marks":[200,400,800], "price":[9, 3, 7]}
>>> f1 = DataFrame(data)
>>> f1
   marks name  price
0    200 yahoo     9
1    400 google    3
2    800 facebook   7
```

这是定义一个 DataFrame 对象的常用方法——使用 dict 定义。字典的“键”（"name", "marks", "price"）就是 DataFrame 的 columns 的值（名称），字典中每个“键”的“值”是一个列表，它们就是那一竖列中的具体填充数据。上面的定义中没有确定索引，所以，按照惯例（Series 中已经形成的惯例）就是从 0 开始的整数。从上面的结果中很明显表示出来，这就是一个二维的数据结构（类似 excel 或者 mysql 中的查看效果）。

上面的数据显示中，columns 的顺序没有规定，就如同字典中键的顺序一样，但是在 DataFrame 中，columns 跟字典键相比，有一个明显不同，就是其顺序可以被规定，向下面这样做：

```
>>> f2 = DataFrame(data, columns=['name','price','marks'])
>>> f2
   name  price  marks
0  yahoo     9    200
1  google    3    400
2 facebook   7    800
```

跟 Series 类似的，DataFrame 数据的索引也能够自定义。

```
>>> f3 = DataFrame(data, columns=['name', 'price', 'marks', 'debt'], index=['a','b','c','d'])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/pymodules/python2.7/pandas/core/frame.py", line 283, in __init__
  mgr = self._init_dict(data, index, columns, dtype=dtype)
File "/usr/lib/pymodules/python2.7/pandas/core/frame.py", line 368, in _init_dict
```

```

mgr = BlockManager(blocks, axes)
File "/usr/lib/pymodules/python2.7/pandas/core/internals.py", line 285, in __init__
    self._verify_integrity()
File "/usr/lib/pymodules/python2.7/pandas/core/internals.py", line 367, in _verify_integrity
    assert(block.values.shape[1:] == mgr_shape[1:])
AssertionError

```

报错了。这个报错信息就太不友好了，也没有提供什么线索。这就是交互模式的不利之处。修改之，错误在于 index 的值——列表——的数据项多了一个，data 中是三行，这里给出了四个项（['a','b','c','d']）。

```

>>> f3 = DataFrame(data, columns=['name', 'price', 'marks', 'debt'], index=['a','b','c'])
>>> f3
   name  price  marks  debt
a  yahoo     9    200   NaN
b  google     3    400   NaN
c facebook    7    800   NaN

```

读者还要注意观察上面的显示结果。因为在定义 f3 的时候，columns 的参数中，比以往多了一项('debt')，但是这项在 data 这个字典中并没有，所以 debt 这一竖列的值都是空的，在 Pandas 中，空就用 NaN 来代表了。

定义 DataFrame 的方法，除了上面的之外，还可以使用“字典套字典”的方式。

```

>>> newdata = {"lang":{"firstline":"python","secondline":"java"}, "price":{"firstline":8000}}
>>> f4 = DataFrame(newdata)
>>> f4
   lang  price
firstline  python  8000
secondline  java   NaN

```

在字典中就规定好数列名称（第一层键）和每横行索引（第二层字典键）以及对应的数据（第二层字典值），也就是在字典中规定好了每个数据格子中的数据，没有规定的都是空。

```

>>> DataFrame(newdata, index=["firstline","secondline","thirdline"])
   lang  price
firstline  python  8000
secondline  java   NaN
thirdline   NaN    NaN

```

如果额外确定了索引，就如同上面显示一样，除非在字典中有相应的索引内容，否则都是 NaN。

前面定义了 DataFrame 数据（可以通过两种方法），它也是一种对象类型，比如变量 f3 引用了一个对象，它的类型是 DataFrame。承接以前的思维方法：对象有属性和方法。

```

>>> f3.columns
Index(['name', 'price', 'marks', 'debt'], dtype=object)

```

DataFrame 对象的 columns 属性，能够显示所有的 columns 名称。并且，还能用下面类似字典的方式，得到某竖列的全部内容（当然包含索引）：

```
>>> f3['name']
a    yahoo
b    google
c  facebook
Name: name
```

这是什么？这其实就是一个 Series，或者说，可以将 DataFrame 理解为是有一个一个的 Series 组成的。

一直耿耿于怀没有数值的那一列，下面的操作是统一给那一列赋值：

```
>>> f3['debt'] = 89.2
>>> f3
   name  price  marks  debt
a  yahoo     9     200  89.2
b  google     3     400  89.2
c facebook    7     800  89.2
```

除了能够统一赋值之外，还能够“点对点”添加数值，结合前面的 Series，既然 DataFrame 对象的每竖列都是一个 Series 对象，那么可以先定义一个 Series 对象，然后把它放到 DataFrame 对象中。如下：

```
>>> sdebt = Series([2.2, 3.3], index=["a","c"])  #注意索引
>>> f3['debt'] = sdebt
```

将 Series 对象(sdebt 变量所引用) 赋给 f3['debt']列，Pandas 的一个重要特性——自动对齐——在这里起作用了，在 Series 中，只有两个索引 ("a","c")，它们将和 DataFrame 中的索引自动对齐。于是乎：

```
>>> f3
   name  price  marks  debt
a  yahoo     9     200  2.2
b  google     3     400  NaN
c facebook    7     800  3.3
```

自动对齐之后，没有被复制的依然保持 NaN。

还可以更精准的修改数据吗？当然可以，完全仿照字典的操作：

```
>>> f3["price"]["c"] = 300
>>> f3
   name  price  marks  debt
a  yahoo     9     200  2.2
b  google     3     400  NaN
c facebook   300     800  3.3
```

这些操作是不是都不陌生呀，这就是 Pandas 中的两种数据对象。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

Pandas 使用 (2)

特别向读者生命，本教程因为篇幅限制，不能将有关 pandas 的内容完全详细讲述，只能“抛砖引玉”，向大家做一个简单介绍，说明其基本使用方法。当读者在实践中使用的时候，如果遇到问题，可以结合相关文档或者 google 来解决。

读取 csv 文件

关于 csv 文件

csv 是一种通用的、相对简单的文件格式，在表格类型的数据中用途很广泛，很多关系型数据库都支持这种类型文件的导入导出，并且 excel 这种常用的数据表格也能和 csv 文件之间转换。

逗号分隔值（Comma-Separated Values, CSV，有时也称为字符分隔值，因为分隔字符也可以不是逗号），其文件以纯文本形式存储表格数据（数字和文本）。纯文本意味着该文件是一个字符序列，不含必须像二进制数字那样被解读的数据。CSV 文件由任意数目的记录组成，记录间以某种换行符分隔；每条记录由字段组成，字段间的分隔符是其它字符或字符串，最常见的是逗号或制表符。通常，所有记录都有完全相同的字段序列。

从上述维基百科的叙述中，重点要解读出“字段间分隔符”“最常见的是逗号或制表符”，当然，这种分隔符也可以自行制定。比如下面这个我命名为 marks.csv 的文件，就是用逗号（必须是半角的）作为分隔符：

```
name,physics,python,math,english
Google,100,100,25,12
Facebook,45,54,44,88
Twitter,54,76,13,91
Yahoo,54,452,26,100
```

其实，这个文件要表达的事情是（如果转化为表格形式）：

	A	B	C	D	E
1	name	physics	python	math	english
2	Google		100	100	25
3	Facebook		45	54	44
4	Twitter		54	76	13
5	Yahoo		54	452	26
6					100

普通方法读取

最简单、最直接的就是 open() 打开文件：

```
>>> with open("./marks.csv") as f:
...     for line in f:
...         print line
...
name,physics,python,math,english
Google,100,100,25,12
Facebook,45,54,44,88
Twitter,54,76,13,91
Yahoo,54,452,26,100
```

此方法可以，但略显麻烦。

Python 中还有一个 csv 的标准库，足可见 csv 文件的使用频繁了。

```
>>> import csv
>>> dir(csv)
['Dialect', 'DictReader', 'DictWriter', 'Error', 'QUOTE_ALL', 'QUOTE_MINIMAL', 'QUOTE_NONE', 'QUOTE_NONNUMERIC']
```

什么时候也不要忘记这种最佳学习方法。从上面结果可以看出，csv 模块提供的属性和方法。仅仅就读取本例子中的文件：

```
>>> import csv
>>> csv_reader = csv.reader(open("./marks.csv"))
>>> for row in csv_reader:
...     print row
...
['name', 'physics', 'python', 'math', 'english']
['Google', '100', '100', '25', '12']
['Facebook', '45', '54', '44', '88']
['Twitter', '54', '76', '13', '91']
['Yahoo', '54', '452', '26', '100']
```

算是稍有改善。

用 Pandas 读取

如果对上面的结果都有点不满意的话，那么看看 Pandas 的效果：

```
>>> import pandas as pd
>>> marks = pd.read_csv("./marks.csv")
>>> marks
   name  physics  python  math  english
0  Google     100     100    25      12
1 Facebook      45      54    44      88
2 Twitter       54      76    13      91
3 Yahoo         54     452    26     100
```

看了这样的结果，你还不感觉惊讶吗？你还不喜欢上 Pandas 吗？这是多么精妙的显示。它是什么？它就是一个 DataFrame 数据。

还有另外一种方法：

```
>>> pd.read_table("./marks.csv", sep=",")
   name  physics  python  math  english
0  Google     100     100    25      12
1 Facebook      45      54    44      88
2 Twitter       54      76    13      91
3 Yahoo         54     452    26     100
```

如果你有足够的好奇心来研究这个名叫 DataFrame 的对象，可以这样：

```
>>> dir(marks)
['T', '_AXIS_ALIASES', '_AXIS_NAMES', '_AXIS_NUMBERS', '__add__', '__and__', '__array__', '__array_wrap__', '__cl...
```

一个一个浏览一下，通过名字可以直到那个方法或者属性的大概，然后就可以根据你的喜好和需要，试一试：

```
>>> marks.index
Int64Index([0, 1, 2, 3], dtype=int64)
>>> marks.columns
Index(['name', 'physics', 'python', 'math', 'english'], dtype=object)
>>> marks['name'][1]
'Facebook'
```

这几个是让你回忆一下上一节的。从 DataFrame 对象的属性和方法中找一个，再尝试：

```
>>> marks.sort(column="python")
   name  physics  python  math  english
1 Facebook      45      54    44      88
```

```

2 Twitter    54   76  13   91
0 Google     100   100  25   12
3 Yahoo      54   452  26   100

```

按照竖列"Python"的值排队，结果也是很让人满意的。下面几个操作，也是常用到的，并且秉承了 Python 的一贯方法：

```

>>> marks[:1]
   name physics python math english
0 Google    100   100  25   12
>>> marks[1:2]
   name physics python math english
1 Facebook   45   54  44   88
>>> marks["physics"]
0 100
1 45
2 54
3 54
Name: physics

```

可以说，当你已经掌握了通过 `dir()` 和 `help()` 查看对象的方法和属性时，就已经掌握了 pandas 的用法，其实何止 pandas，其它对象都是如此。

读取其它格式数据

csv 是常用来存储数据的格式之一，此外常用的还有 MS excel 格式的文件，以及 json 和 xml 格式的数据等。它们都可以使用 pandas 来轻易读取。

`.xls` 或者 `.xlsx`

在下面的结果中寻觅一下，有没有跟 excel 有关的方法？

```

>>> dir(pd)
['DataFrame', 'DataMatrix', 'DateOffset', 'DateRange', 'ExcelFile', 'ExcelWriter', 'Factor', 'HDFStore', 'Index', 'Int64Index', 'M

```

虽然没有类似 `read_csv()` 的方法（在网上查询，有的资料说有 `read_xls()` 方法，那时老黄历了），但是有 `ExcelFile` 类，于是乎：

```

>>> xls = pd.ExcelFile("./marks.xlsx")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/pymodules/python2.7/pandas/io/parsers.py", line 575, in __init__

```

```
from openpyxl import load_workbook
ImportError: No module named openpyxl
```

我这里少了一个模块，看报错提示，用 pip 安装 openpyxl 模块： sudo pip install openpyxl 。继续：

```
>>> xls = pd.ExcelFile("./marks.xlsx")
>>> dir(xls)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__module__', ...
>>> xls.sheet_names
['Sheet1', 'Sheet2', 'Sheet3']
>>> sheet1 = xls.parse("Sheet1")
>>> sheet1
   0  1  2  3  4
0  5 100 100 25 12
1 6 45 54 44 88
2 7 54 76 13 91
3 8 54 452 26 100
```

结果中，columns 的名字与前面 csv 结果不一样，数据部分是同样结果。从结果中可以看到，sheet1 也是一个 DataFrame 对象。

对于单个的 DataFrame 对象，如何通过属性和方法进行操作，如果读者理解了本教程从一开始就贯穿进来的思想——利用 dir() 和 help() 或者到官方网站，看文档！——此时就能比较轻松地进行各种操作了。下面的举例，纯属是为了增加篇幅和向读者做一些诱惑性广告，或者给懒惰者看看。当然，肯定是不完全，也不能在实践中照搬。基本方法还在刚才交代过的思想。

如果遇到了 json 或者 xml 格式的数据怎么办呢？直接使用本教程第贰季第陆章中 [《标准库 \(7\) \(页 0\)](#) 中的方法，再结合 Series 或者 DataFrame 数据特点读取。

此外，还允许从数据库中读取数据，首先就是使用本教程第贰季第柒章中阐述的各种数据库（[《MySQL 数据库 \(1\)》\(页 0\)](#)）连接和读取方法，将相应数据查询出来，并且将结果（结果通常是列表或者元组类型，或者是字符串）按照前面讲述的 Series 或者 DataFrame 类型数据进行组织，然后就可以对其操作。

[总目录 \(页 0\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

处理股票数据

这段时间某国股市很火爆，不少砖家在分析股市火爆的各种原因，更有不少人看到别人挣钱眼红了，点钞票杀入股市。不过，我还是很淡定的，因为没钱，所以不用担心任何股市风险临到。

但是，为了体现本人也是与时俱进的，就以股票数据为例子，来简要说明 pandas 和其它模块在处理数据上的应用。

下载 yahoo 上的数据

或许你稀奇，为什么要下载 yahoo 上的股票数据呢？国内网站上不是也有吗？是有。但是，那时某国内的。我喜欢 yahoo，因为她曾经吸引我，注意我说的是www.yahoo.com，不是后来被阿里巴巴收购并拆散的那个。



虽然 yahoo 的世代渐行渐远，但她终究是值得记忆的。所以，我要演示如何下载 yahoo 财经栏目中的股票数据。

```
In [1]: import pandas
In [2]: import pandas.io.data

In [3]: sym = "BABA"

In [4]: finace = pandas.io.data.DataReader(sym, "yahoo", start="2014/11/11")
In [5]: print finace.tail(3)
      Open    High     Low   Close  Volume  Adj Close
Date
2015-06-17  86.580002  87.800003  86.480003  86.800003  10206100  86.800003
2015-06-18  86.970001  87.589996  86.320000  86.750000  11652600  86.750000
2015-06-19  86.510002  86.599998  85.169998  85.739998  10207100  85.739998
```

下载了阿里巴巴的股票数据（自 2014 年 11 月 11 日以来），并且打印最后三条。

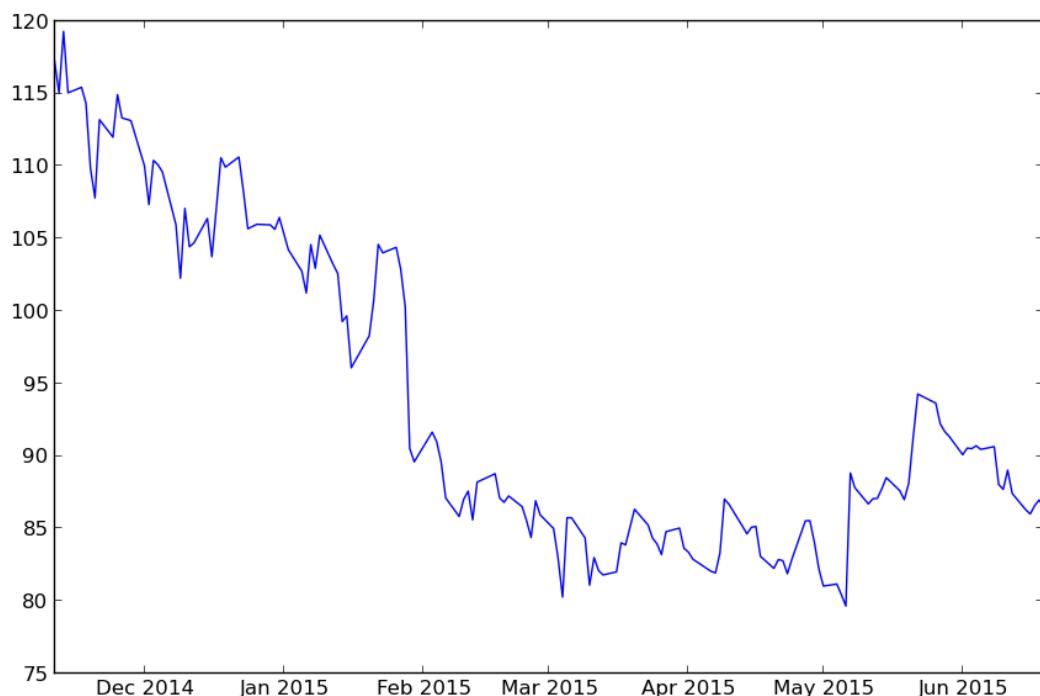
画图

已经得到了一个 DataFrame 对象，就是前面已经下载并用 finace 变量引用的对象。

```
In[6]: import matplotlib.pyplot as plt
In [7]: plt.plot(finace.index, finace["Open"])
Out[]: [<matplotlib.lines.Line2D at 0xa88e5cc>]

In [8]: plt.show()
```

于是乎出来了下图：



从图中可以看出阿里巴巴的股票自从 2014 年 11 月 11 日到 2015 年 6 月 19 日的股票开盘价变化。看来那个所谓的“光棍节”得到了股市的认可，所以，在此我郑重地建议阿里巴巴要再造一些节日，比如 3 月 3 日、4 月 4 日，还好，某国还有农历，阳历用完了用农历。可以维持股票高开高走了。

阿里巴巴的事情，我就不用操心了。

上面指令中的 `import matplotlib.pyplot as plt` 是个此前没有看到的。`matplotlib` 模块是 Python 中绘制二维图形的模块，是最好的模块。本教程在这里展示了它的一个小小地绘图功能，读者就一下看到阿里巴巴“光棍节”的力量，难道还不能说明 `matplotlib` 的强悍吗？很可惜，`matplotlib` 的发明者——John Hunter 已经于 2012 年 8 月 28 日因病医治无效英年早逝，这真是天妒英才呀。为了缅怀他，请读者访问官方网站：matplotlib.org，并认真学习这个模块的使用。

经过上面的操作，读者可以用 `dir()` 这个以前常用的法宝，来查看 `finace` 所引用的 `DataFrame` 对象的方法和属性等。只要运用此前不断向大家演示的方法——`dir+help` ——就能够对这个对象进行操作，也就是能够对该股票数据进行各种操作。

再次声明，本课程仅仅是稍微演示一下相关操作，如果读者要深入研习，恭请寻找相关的专业书籍资料阅读学习。

[总目录](#) | [上节：Pandas 使用 \(2\)](#)

如果你认为有必要打赏我，请通过支付宝：qiwsir@126.com,不胜感激。

HTML



T
P

12

结尾



unity



HTML



如何成为 Python 高手

这篇文章主要是对我收集的一些文章的摘要。因为已经有很多比我有才华的人写出了大量关于如何成为优秀 Python 程序员的好文章。

我的总结主要集中在四个基本题目上：

- 函数式编程，
- 性能，
- 测试，
- 编码规范。

如果一个程序员能将这四个方面的内容知识都吸收消化，那他/她不管怎样都会有巨大的收获。

函数式编程

命令式的编程风格已经成为事实上的标准。命令式编程的程序是由一些描述状态转变的语句组成。虽然有时候这种编程方式十分的有效，但有时也不尽如此(比如复杂性) —— 而且，相对于声明式编程方式，它可能会显得不是很直观。

如果你不明白我究竟是在说什么，这很正常。这里有一些文章能让你脑袋开窍。但你要注意，这些文章有点像《骇客帝国》里的红色药丸 —— 一旦你尝试过了函数式编程，你就永远不会回头了。

- <http://www.amk.ca/python/writing/functional>
- http://www.secretnix.de/olli/Python/lambda_functions.hawk
- <http://docs.python.org/howto/functional.html>

性能

你会看到有如此多的讨论都在批评这些“脚本语言”(Python, Ruby)是如何的性能低下，可是你却经常的容易忽略这样的事实：是程序员使用的算法导致了程序这样拙劣的表现。

这里有一些非常好的文章，能让你知道 Python 的运行时性能表现的细节详情，你会发现，通过这些精炼而且有趣的语言，你也能写出高性能的应用程序。而且，当你的老板质疑 Python 的性能时，你别忘了告诉他，这世界上第二大的搜索引擎就是用 Python 写成的 —— 它叫做 Youtube(参考 Python 摘录)

- <http://jaynes.colorado.edu/PythonIdioms.html>
- <http://wiki.python.org/moin/PythonSpeed/PerformanceTips>

测试

如今在计算机科学界，测试可能是一个最让人不知所措的主题了。有些程序员能真正的理解它，十分重视 TDD (测试驱动开发)和它的后继者 BDD (行为驱动开发)。而另外一些根本不接受，认为这是浪费时间。那么，我现在将告诉你：如果你不曾开始使用 TDD/BDD，那你错过了很多最好的东西！

这并不只是说引入了一种技术，可以替换你的公司里那种通过愚蠢的手工点击测试应用程序的原始发布管理制度，更重要的是，它是一种能够让你深入理解你自己的业务领域的工具 —— 真正的你需要的、你想要的攻克问题、处理问题的方式。如果你还没有这样做，请试一下。下面的这些文章将会给你一些提示：

- <http://www.oreillynet.com/lpt/a/5463>
- <http://www.oreillynet.com/lpt/a/5584>
- http://wiki.cacr.caltech.edu/danse/index.php/Unit_testing_and_Integration_testing
- <http://docs.python.org/library/unittest.html>

编码规范

并非所有的代码生来平等。有些代码可以被另外的任何一个好的程序员读懂和修改。但有些却只能被读，而且只能被代码的原始作者修改 —— 而且这也只是在他或她写出了这代码的几小时内可以。为什么会这样？因为没有经过代码测试(上面说的)和缺乏正确的编程规范。

下面的文章给你描述了一个最小的应该遵守的规范合集。如果按照这些指导原则，你将能编写出更简洁和漂亮的代码。作为附加效应，你的程序会变得可读性更好，更容易的被你和任何其他人修改。

- <http://www.python.org/dev/peps/pep-0008/>
- http://www.fantascienza.net/leonardo/ar/python_best_practices.html

那就去传阅这些资料吧。从坐在你身边的人开始。也许在下一次程序员沙龙或编程大会的时候，也已经成为一名 Python 编程高手了！

祝你学习旅途顺利。

本文来源：<http://blogread.cn/it/article/3892?f=wb>



中国最大的IT职业在线教育平台



更多信息请访问 

<http://wiki.jikexueyuan.com/project/start-learning-python/>