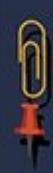
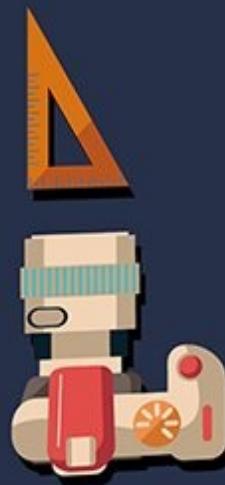


跟老齐学 Python

从入门到精通

电子工业出版社



跟老齐学 Python 从入门到精通

齐伟
编著



人生苦短，我用Python
零基础起步，手把手进阶，辅以实际案例，
Python这样学才简单！



中国工信出版集团

电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

Table of Contents

目录

第零章 预备

Introduction	2.1
关于python的故事	2.2
从小工到专家	2.3
安装python的开发环境	2.4
集成开发环境	2.5

第壹章 基本数据类型

数和四则运算	3.1
除法	3.2
常用数学函数和运算优先级	3.3
写一个简单程序	3.4
字符串(1)	3.5
字符串(2)	3.6
字符串(3)	3.7
字符串(4)	3.8
字符编码	3.9
列表(1)	3.10
列表(2)	3.11
列表(3)	3.12
回顾列表和字符串	3.13
元组	3.14
字典(1)	3.15
字典(2)	3.16
集合(1)	3.17
集合(2)	3.18

第貳章 语句和文件

运算符	4.1
语句(1)	4.2
语句(2)	4.3
语句(3)	4.4
语句(4)	4.5
语句(5)	4.6
文件(1)	4.7
文件(2)	4.8
迭代	4.9
练习	4.10
自省	4.11

第叁章 函数

函数(1)	5.1
函数(2)	5.2
函数(3)	5.3
函数(4)	5.4
函数(5)	5.5
函数(6)	5.6
函数练习	5.7
zip()补充	5.8
命名空间	5.9

第肆章 类

类(1)	6.1
类(2)	6.2
类(3)	6.3
类(4)	6.4
类(5)	6.5
多态和封装	6.6
定制类	6.7
黑魔法	6.8

迭代器	6.9
生成器	6.10
上下文管理器	6.11

第五章 错误和异常

错误和异常(1)	7.1
错误和异常(2)	7.2
错误和异常(3)	7.3

第六章 模块

编写模块	8.1
标准库(1)	8.2
标准库(2)	8.3
标准库(3)	8.4
标准库(4)	8.5
标准库(5)	8.6
标准库(6)	8.7
标准库(7)	8.8
标准库(8)	8.9
第三方库	8.10

第七章 保存数据

将数据存入文件	9.1
MySQL数据库(1)	9.2
MySQL数据库(2)	9.3
MongoDB数据库	9.4
SQLite数据库	9.5
电子表格	9.6

附：网络文摘

如何成为python高手	10.1
ASCII、Unicode、GBK和UTF-8字符编码的区别联系	10.2

大数据全栈式开发语言 – Python	10.3
机器学习编程语言之争，Python夺魁	10.4
Python 2.7.x 和 3.x 版本的重要区别	10.5

This is for everyone.

In the begining when God created the heavens and the earth. the earth was a formless void and darkness covered the face of the deep, while a wind from God swept over the face of the waters. Then God said,"Let there be light"; and there was light. And God saw that the light was good; and God separated the light from the darkness. (GENESIS 1:1-4)

《跟老齐学Python》（入门教程）

From beginner to master.

针对零基础的学习者，试图实现从基础到精通，还要看自己的造化。

原名叫做《零基础学Python》，后来由于图书出版，更名为《跟老齐学Python》。

《跟老齐学Python》一书已经由电子工业出版社出版，于2016年3月1日起，陆续在亚马逊、京东、当当、淘宝等网站发售，请关注，并感谢支持。

此处的在线版与上述印刷的版本有所不同。在印刷出来之后，又发觉有一些地方需要修改，于是就对一部分进行了重写，并且在和朋友们交流后，也会吸收很多意见和建议，用之于修改在线版。其中改动最大的在于，本教程不再包括“实战”部分，而是专门叫做面向零基础的“入门教程”。对于实战部分，我会在适当时候专门撰写。

The fear of the LORD is the beginning of knowledge; fools despise wisdom and instruction.(PROVERBS 1:7)

敬畏耶和华是知识的开端，愚妄人藐视智慧和训诲。

关于Python的故事

如同学习任何一种自然语言比如英语、或者其它编程语言比如汇编一样，总要说一说有关这种语言的事情，有的可能就是八卦，越八卦的越容易传播。当然，以下的所有说法中，难免充满了自恋，因为你看不到说Python的坏话。这也好理解，如果要挑缺点是比较容易的事情，但是找优点，不管是对人还是对其他事物，都是困难的。这也许是人的劣根之所在吧，喜欢挑别人的刺儿，从而彰显自己在那方面高于对方。特别是在我们这个麻将文化充斥的神奇地方，更多了。

废话少说点（已经不少了），进入有关Python的话题。

Python的昨天今天和明天

这个题目有点大了，似乎回顾过去、考察现在、张望未来，都是那些掌握方向的大人物（司机吗？）做的。那就让我们每个人都成为大人物吧。因为如果不回顾一下历史，似乎无法满足学习者的好奇心；如果不考察一下现在，学习者不放心（担心学了之后没有什么用途）；如果不张望一下未来，怎么能吸引（也算是一种忽悠吧）学习者或者未来的开发者呢？

Python的历史

Python的创始人为吉多·范罗苏姆（Guido van Rossum）。关于这个人开发这种语言的过程，很多资料里面都要记录下面的故事：

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。

这个故事我是从维基百科里面直接复制过来的，很多讲Python历史的资料里面，也都转载这段。但是，在我来看，这段故事有点忽悠人的味道。其实，上面这段中提到的，吉多为了打发时间而决定开发Python的说法，来自他自己的这样一段自述：

Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office (a government-run research lab in Amsterdam) would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).
(原文地址：<https://www.python.org/doc/essays/foreword/>)

首先，必须承认，这个哥们儿是一个牛人，非常牛的人。此处献上我的崇拜。

其次，读者千万别认为Python就是一个随随便便就做出来的东西，就是一个牛人一冲动搞出来的东西。人家也是站在巨人的肩膀上的。

第三，牛人在成功之后，往往把奋斗的过程描绘的比较简单，或者是谦虚？或者是让人听起来他更牛？反正，我们看最后结果的时候，很难感受过程中的酸甜苦辣。

不管怎么样，牛人在那时刻开始创立了Python，而且，他更牛的在于具有现代化的思维：开放。通过Python社区，吸引来自世界各地的开发者，参与Python的建设。在这里，请读者一定要联想到Linux和它的创始人芬兰人林纳斯·托瓦兹。两者都秉承“开放”思想，得到了来自世界各地开发者和应用者的欢呼和尊敬。也请大家再联想到另外一个在另外领域秉承开放思想的人——邓小平先生，他让一个封闭的破旧老水车有了更新。

Python的现在

应该说Python现在表现不错。除了在Web开发方面有很多应用之外（当然PHP在这方面也是很不错的），在数据分析、机器学习、大数据、云计算等这些时髦的领域，都有它的身影，并且影响力越来越大了。此外，还有自动化运维、自动化测试。

读者可以到这个网站看一看Python的应用案例：<https://www.python.org/about/success/>。

不过，因为赵国大学教育的问题，致使很多青年才俊对Python了解甚少；更因为赵国的功利化优良传统，青年才俊们最大担心的是学了Python——这种学校老师很少甚至从没有提及的怪东西——没有什么用途，因为才俊们已经通过铺天盖地的广告了解到Android、iOS开发，于是就认为“软件开发==Android or iOS”，其它都过时了——最希望才俊能够跳出四角天空，用自己的头脑思考、用自己的眼睛看世界，形成独立的判断，不要听信广告——也包括我这里的各种对Python的溢美之词。

Python的未来

这个不需要描述，它的未来在所有使用者和学习者手中。

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”，只有内行人才能理解。

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，那就比较一下吧。

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，要阅读非删减版，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。很多小伙伴都会说“嘿，兄弟，你怎么可能不爱上了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。你也许会认为她是很智慧的人，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。

所以，Python值得拥有。

在正式开始学习之前，首先要告诉你成为Python高手的秘诀。

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!

“吃水不忘挖井人”，谁创造了Python，我们一定要感恩并崇拜。

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。

Python已经让人心动了。除了心动，还要行动；只有行动，才能“从小工到专家”。

Do not store up for yourselves treasures on earth, where moth and rust consume and where thieves break in and steal; but store up for yourselves treasures in heaven, where neither moth nor rust consumes and where thieves do not break in and steal. For where your treasure is, there your heart will be also.(MATHEW 10:19-21)

不要为自己积攒财宝在地上，地上有虫子咬，能锈坏，也有贼挖窟窿来偷；只要积攒财宝在天上，天上没有虫子咬，不能锈坏，也没有贼挖窟窿来偷。因为的财宝在哪里，你的心也在哪里。

从小工到专家

这是每个程序员的梦想。

有一本书的名字就是《程序员修炼之道：从小工到专家》，我借用此书的标题。

但是，本书或许能够是你成为专家路上的一块铺路石，如果真能如此，我感到荣幸之至。

关于本书

我曾经在网上写过《零基础学Python（第一版）》，完成之后，发现有一些错误，并且整体结构对零基础的学习者不是很适合。于是，就重新写了本教程，力图为零基础的学习者提供一个入门的教程。

本教程有幸得到了电工业出版社的认可，已经集结成为《跟老齐学Python》一书出版（读者可以在各大网络书店搜索，欢迎购买）。但是，当书出版之后，我又萌生了进一步就该的设计，于是在2016年3月到5月期间，对网络教程进行了再次修订，并且定名为现在的名称——《跟老齐学Python：入门教程》——言外之意，还有别的教程。的确，在我计划之中，还要编写针对相关方面应用的教程。

对于本书，我再次强调，其对象零基础的学习者。之所以再次强调，是因为已经有读者误解了本书的目的。购买了之后，发现比较基础，于是就狂喷一通，并言之凿凿，“还不如看官方文档”。的确，Python的官方文档是最好的。如果能够直接阅读官方文档来学习，当然是一种不错的方法，并且这样的学习者也一定是高手，所以不是本书的读者。就此也建议才俊们，请看完成内容再喷无妨。

或许书中“干货”不多——人体约70%是水分，木乃伊才是干货——水是一种很好的溶剂，它存在书中，目的是别那么枯燥——或许这个目标没有完全实现，只能说是力争了，毕竟编程语言还是不如讲故事精彩。

我也非常欢迎读者能够以心平气和的方式跟我交流，以帮助我改进本书。所以，提供如下联系到我的途径：

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，名称是：qiwsir。地址：<https://github.com/qiwsir>

4. 经常关注我的网站：www.itdiffer.com

现在的年代是一个“东风吹战鼓擂”的年代，能够心平气和讲话的本来就不多，但是气大伤肾，特别是年轻人，一定要小心啦。

Python的版本

关于Python的版本问题，是必须要交代的。

不管出于什么原因，我认为Python给自己搞了两个版本，是败笔。

虽然如此，但幸亏两个版本并非天壤之别，绝大部分是一样的。所以，学习者可以选择任何一种版本进行学习，然后在具体应用的时候，用到什么版本，只要稍加注意，或者到网上搜索一下，即可。

我在这里还整理了一篇文章：[Python2.7.x和3.x版本的重要区别](#)，不知是否愿意阅读？

但是，总有不放心的初学者。

我曾被无数次的拷问：教程是Python 2还是Python 3？

我非常想告诉他什么都支持，但是，我的代码的确是在Python 2下调试的，总不能撒谎吧。于是当我如实奉告的时候，他会说要学习Python 3，转头找那些号称是Python 3的教程。

无奈。

为了迎合学习者胃口，我的教程，从即日起，适合于**Python 3**。

从此，本教程宣称：支持**Python 2**和**Python 3**。如遇到不符合此宣称的地方，请告知我，我立刻修改。

还要说一句，上述宣称的最终解释权归本教程作者。

不管是2还是3，总要从零开始学习，从零开始学，就意味着不需要基础。这个我有信心。

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（作为学习编程的入门语言，我现在最不理解的是用C，因为很多曾经立志学习编程的人学了C语言之后，才知道自己不适合编程。难道是用C来筛选这个行业的从业者吗？）。总之，不用担心自己的所谓基础问题。

这个教程，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。

最后的结论是：学习python，你不用担心基础问题。

特别是看我的教程，我的目标就是要跟你一起从零基础开始，直到高手境界——不是我夸口，是你要有信心。

所以，尽管放胆来学，不用犹豫、不要惧怕。还有一个原因，是因为她优雅。

从小工到专家

有不少学习Python的朋友询问：

“书已经看了，书上的代码也运行过了，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。”

也遇到过一些大学毕业生，虽然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，常常让我大跌眼镜，特别是当他面对真实的工作对象时，所表现出来的能力要比成绩单上的数字差太多了。

我一般会武断地下一个结论：练的少。

因此，从小工到专家，就要多练。当不是盲目地练习，如果找不到方向，可以从阅读代码开始。

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，“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: qjwsir。

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。

之所以run，是要看看这个程序运行结果是什么。除了调试别人的程序，还要调试自己的程序。

调试程序

首先要自己动手写程序。

“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。

为了帮助学习者调试动手敲代码，我正在推出一个项目 [《编程匠艺》训练](#)，可以参加。

在写程序、调试程序的时候，一定会遇到很多问题。怎么办？

办法就是应用网络，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，只能让你成为“砖家”，以至于“砖工”。所以，请用：google.com。

此外，还有其它的好网站，我会陆续向有意成为专家的朋友提供。

成为专家的通道千万条，但这两条路径是真道。

千里之行，始于足下。要学Python，就要有学习的环境。

[总目录](#) | [上节：关于Python的故事](#) | [下节：安装Python的开发环境](#)

But Jesus said to them, "Because of your hardness of heart he wrote this commandment for you. But from the beginning of creation, 'God made them male and female.' 'For this reason a man shall leave his father and mother and be joined to his wife, and the two shall become one flesh.' Therefore what God has joined together, let no one separate."(MARK 10:5-9)

Python安装

不论是谁，只要用Python，就必须配置Python的开发和运行环境。

这环境是什么？它就是若干个软件程序。

不仅Python，任何高级语言都是需要一个自己的编程环境，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面，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。

不论读者是零基础，还是非零基础，不要希望在这里学到很多高深的Python语言技巧，因为这里充满了水分。

“靠，原来是看胡扯的？”

非也。水是生命源泉，一个好的教程，如果没有水分，仅仅是一些干瘪的知识，那么就是一个指令速查手册，难道阅读起来能让你兴趣盎然吗？

在本教程中，我将重点向读者展现学习方法，比如给大家推荐的“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，不少人把互联网等于娱乐网，我衷心希望从你开始，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。

读者看好了，我上面写的很清楚，是“google一下”，不论你是什么派别，只要你立志做一个好一点的程序员，只要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，作为一个追求在技术上有点成就的人，一定要学点上网的技术的，你懂得。

如果你不懂，的确就是是我的读者：零基础。

欲练神功，挥刀自宫。神功是有前提de。

要学Python，不用自宫。Python不用那么残忍的前提，但是，也需要安装点东西才能用。

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Python2.7.x。选哪个都行，本教程两者兼顾，如果没有兼顾到的，可以网上搜一搜。

不用为学习哪个版本而忧愁。如果学了Python2.7，对于Python3，也只是某些地方的小变化了。

下面就一步一步地来安装。如果不是零基础的，可以略过。

在Linux系统中安装

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MacOS，也一样，因为都是UNIX下的蛋。

只是windows有点另类了。也不必惶恐，Python就是跨平台的。

本教程的所有程序，都是在Ubuntu下调试的，没有时间和精力单独再搞windows的，或许会在某些地方提示windows中注意的地方。一般情况下可以放心，总体上跟操作系统关联不紧密。

根据个人喜好，我推荐读者熟悉Linux操作系统，这是很好的。

我用Ubuntu。

只要装了Ubuntu这个操作系统，默认里面就已经把Python安装好了。最新的Ubuntu中可能已经预装了Python的两个版本，你可以选择使用。

接下来就在shell中输入Python（或者Python3，是启动了Python 3），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（“交互模式”，这是一个非常有用的东西，从后面的学习中，你就能体会到，这里是学习Python的主战场）。

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（如果读者下载Python 3或者其它版本，可以到官网www.python.org查看相应版本的源码地址，这里仅仅是举例，不可照抄）：

```
wget http://www.python.org/ftp/python/2.7.6/Python-2.7.6.tgz
```

- 解压源码包

```
tar -zxvf Python-2.7.6.tgz
```

- 编译

```
cd Python-2.7.6
```

```
./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)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

windows系统中安装程序，就是不断地“下一步”。

在windows系统中安装

到[下载页面里面](#)找到你喜欢的版本，然后根据自己的计算机情况，下载相应的安装包，下载完毕，安装过程同其它的windows软件安装方法。

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。

从开始菜单中，你还能找到Python IDEL，打开之后，也是一个交互模式，并且还是一个简单的编辑器。

还有另外一个代表高端的电脑，貌似用Mac OS X的人都很厉害和富有。

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Ubuntu差不多。

按照以上方法，顺利安装成功，只能说明幸运，无它。

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，它能帮助你解决所有问题。当然，加入QQ群或者通过微博，问我也可以。

最后还要交代，你不用纠结是2还是3，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。

不为2还是3浪费时间，但是开发工具的选择要对自己的胃口。

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

"But I say to you that listen, Love your enemies, do good to those who hate you, bless those who curse you, pray for those who abuse you. If anyone strikes you on the cheek, offer the other also; and from anyone who takes away your coat do no withhold even your shirt. Give to everyone who begs from you; and if anyone takes away your goods, do not ask for them again. Do to others as you would have them do to you....Be merciful, just as your Father is merciful."

开发工具

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

这是所有编程语言的惯例。

Hello world

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

- Ubuntu，直接在shell中输入python即可
- windows，从开始菜单中找到IDLE(Python GUI)

进入到如下类似界面：

```
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.
>>>
```

或者是（注意，我在兼顾Python 3了）：

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:54:25) [MSC v.1900 64 bit (AMD64)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

这就是交互模式。将在后面长期使用，会伴随你Python的代码生涯。

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
```

上面的是在Python 2中，下面的是Python 3，注意区别。

```
>>> 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，下面将其内部行为逐一解说。

说明：Python代码中常用到了一个符号：`#`，就是键盘上数字3上面的那个井字符。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。这是交互模式的标志。

```
>>>
```

`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，注意双引号，是英文状态下的。引号不是打印内容，它相当于一个包裹，把打印的内容包起来，统一交给Python。

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

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
```

如果你使用的是Python 3，这里应该写成：

```
>>> print("Hello world")
```

“交互模式”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，开始练字。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”。

集成开发环境

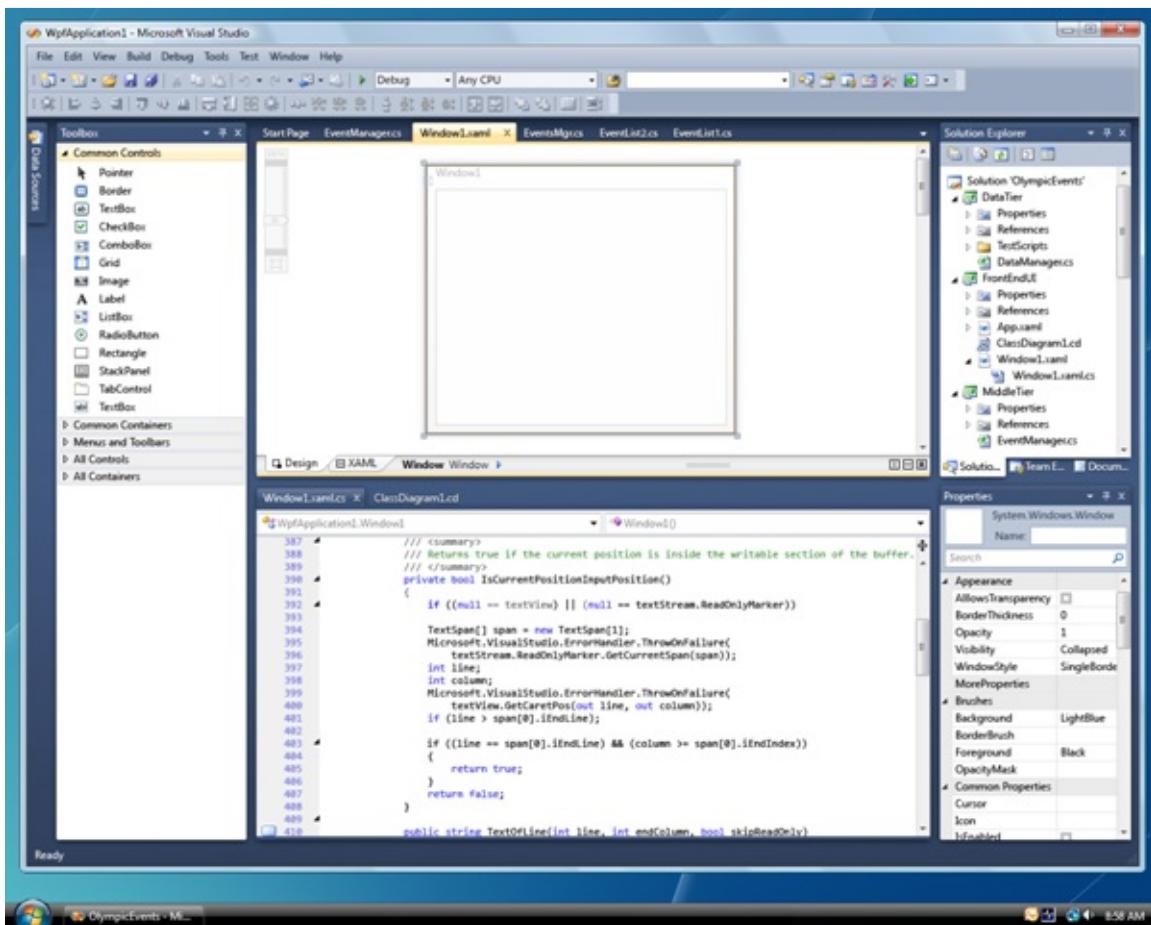
IDE的全称是：Integrated Development Environment，简称IDE，也称为Integration Design Environment、Integration Debugging Environment，翻译成中文叫做“集成开发环境”，在台湾那边叫做“整合開發環境”，它是一种辅助程序员开发用的应用软件。

[维基百科](#)这样对IDE定义：

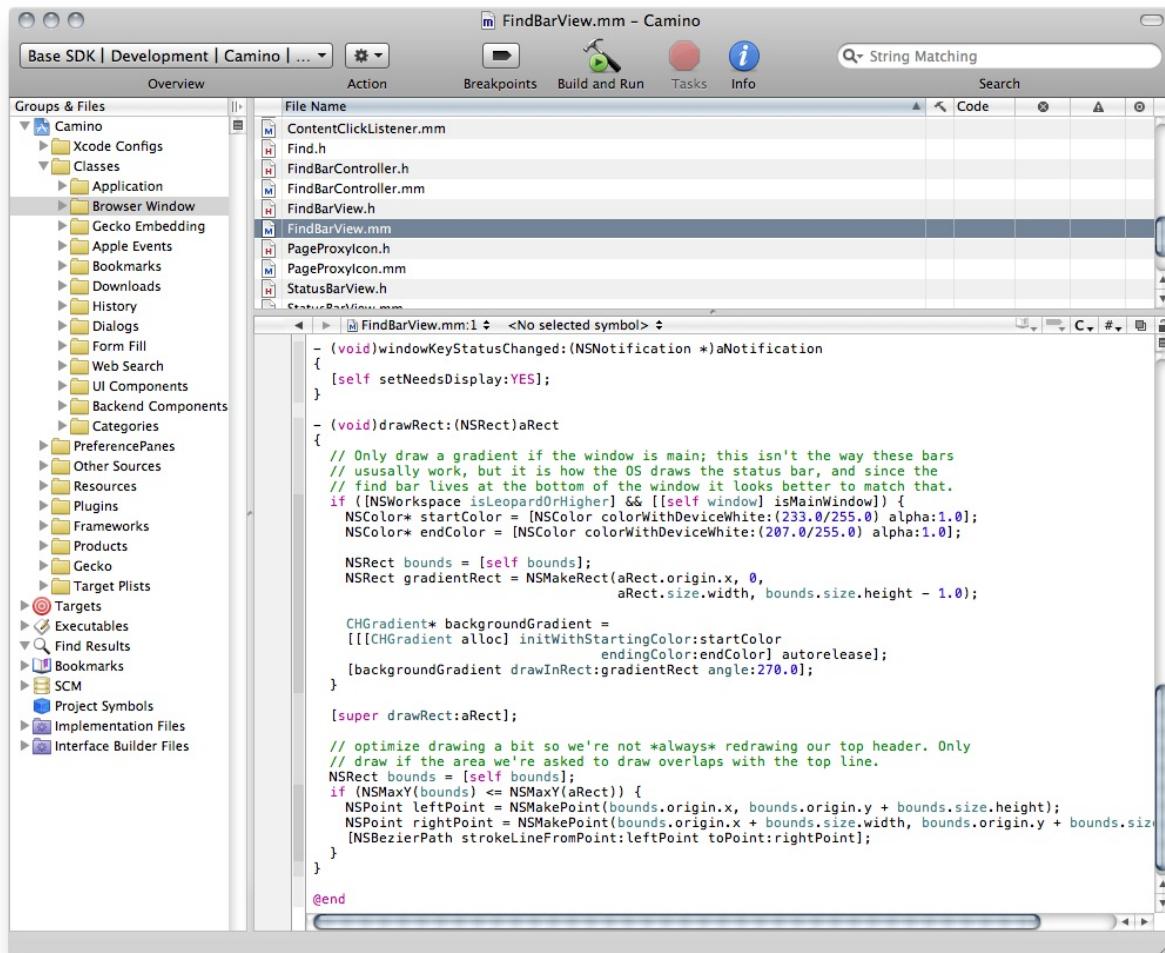
IDE通常包括程式語言編輯器、自動建立工具、通常還包括除錯器。有些IDE包含編譯器／直譯器，如微软的Microsoft Visual Studio，有些则不包含，如Eclipse、SharpDevelop等，这些IDE是通过调用第三方编译器来实现代码的编译工作的。有时IDE还会包含版本控制系统和一些可以设计图形用户界面的工具。许多支援物件導向的现代化IDE还包括了類別瀏覽器、物件檢視器、物件結構圖。雖然目前有一些IDE支援多種程式語言（例如Eclipse、NetBeans、Microsoft Visual Studio），但是一般而言，IDE主要還是針對特定的程式語言而量身打造（例如Visual 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。

集成开发环境



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

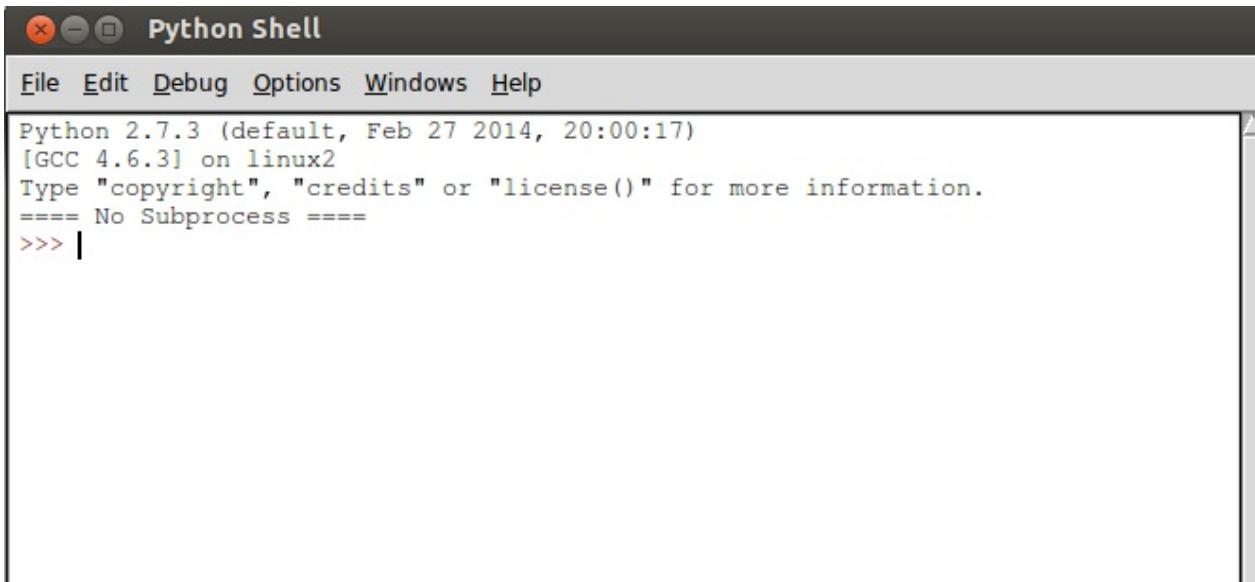
在这里可以提问，可以查看答案。一般如果有问题，先在这里查找，多能找到非常满意的结果，至少有很大启发。

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”菜单中找到“IDLE（Python GUI）”来启动IDLE。启动之后，大概看到这样一个图（也可能是Python 3版本，根据你的版本而定）。



注意：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，跟上面类似的图。

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。

从哪里开始？从计算机的原初功能开始，那就是计算。

For I am not ashamed of the gospel; it is the power of God for salvation to everyone who has faith, to the Jew first and also to the Greek. For in it the righteousness of God is revealed through faith for faith; as it is written, "The one who is righteous will live by faith"

数和四则运算

计算机，原本是用来计算的。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，用int表示。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 2会在其末尾显示一个L，Python 3中干脆把L都省略了，因为本来它也没有什么作用了。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。

这个功能重要，在于Python能自动处理大整数问题，不用担心溢出。什么是“溢出”，随后说明，或者在这里去Google它。

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”，用float表示。

注意，刚才我提到了小数，其实把小数称之为“浮点数”，这种说法并不准确。为此，“知乎”上有专门的解释，请阅读（原文地
址：<http://www.zhihu.com/question/19848808/answer/22219209>：

并不是说小数叫做浮点数。准确的来说：“浮点数”是一种表示数字的标准，整数也可以用浮点数的格式来存储。

当代大部分计算机和程序在处理浮点数时所遵循的标准是由IEEE和ANSI制定的。比如，单精度的浮点数以4个字节来表示，这4个字节可以分为三个部分：1位的符号位(0代表正数，1代表负数)，8位用作指数，最后的23位表示有效数字。

“浮点数”的定义是相对于“定点数”来说的，它们是两种表示小数的方式。

所谓“定点”是指小数点的位置总是在数的某个特定位置。比如在银行系统中，小数点的位置总是在两位小数之前(这两位小数用来表示角和分)。其可以使用BCD码来对小数进行编码。

浮点格式则是基于科学计数法的，它是存储极大或极小数的理想方式。但使用浮点数来表示数据的时候，由于其标准制定方面的原因可能会带来一些问题，例如：某两个不同的整数在单精度浮点数的表示方法下很可能无法区分。

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）。

进制问题此处不是重点，建议读者自行查找资料阅读。

在Python中，每个数字都是真实存在的，相对于我们——人类——来讲，它真实存在，它就是对象（object）。

对象是一个深刻的术语，不管你是否理解，我先这么说，你先听着、用着，逐渐逐渐，就理解深入了。

还要注意，此对象非彼对象，但是学习Python或许在帮助你解决彼对象的时候有帮助。

比如整数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(3.222222)
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'>
```

在Python 3中，看到的是这样的结果：

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

用内建函数能够查看对象的类型。

- <type 'int'> 或者 <class 'int'>，说明3是整数类型（Integer）；
- <type 'float'> 或者 <class 'float'>，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”类型的对象（或者数据），也可以说对象（或数据）类型是整数型，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了这两个对象，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，作用等同于标签的变量 `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的一个重要特征——对象有类型，变量无类型。

理解否？如果没有理解，也不要紧张，继续学习，“发展是硬道理”，在发展中解决问题。

上面的知识，可以用来计算。

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
```

在Python 3中，上面的运算中，除法是有区别的，它们将是这样的：

```
>>> 10 / 2  
5.0  
>>> 10 / 5 + 1  
3.0
```

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

另外，我相信读者已经发现了一个重要的公理：

在计算机中，四则运算和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后辈小子们学习的需要了，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

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
```

观察能力运用到这里，在物理课堂上学的一点不浪费。

找出不一样的地方。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。这意味着什么，你可以继续试验，看看能不能总结出什么规律。后面会用到。

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？（推荐去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”。

大整数

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，无忧愁，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](#)

可以在实验一下大整数相乘。

```
>>> 123456789870987654321122343445567678890098876 * 1233455667789990099876543332387665  
443345566  
15227847719352756287004435258757627727756232836203244433901915893701780160167797618381  
6L
```

Python 3中的计算结果，比上面的少L：

```
>>> 123456789870987654321122343445567678890098876 * 1233455667789990099876543332387665  
443345566  
15227847719352756287004435258757627727756232836203244433901915893701780160167797618381  
6
```

Python自动帮为我们解决大整数问题。这是Python跟很多其它编程语言大不一样的地方，也就是说，你尽可以放心，在Python中，整数的长度是不受限制的（当然，这句话有点绝对了）。

Python解忧愁，可是使用任意大的整数。

所以，选择学习Python就是珍惜光阴了。

强调知识点，有两个符号需要牢记（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(98877654422211223344556677889988776655443322113334445556667778899877654322234
4556678)
<type 'long'>   #是长整数，也是一个整数
```

Python 3的结果是：

```
>>> type(4)
<class 'int'>
>>> type(5.0)
<class 'float'>
>>> type(98877654422211223344556677889988776655443322113334445556667778899877654322234
4556678)
<class 'int'>
```

两个版本有区别，在Python 3中，不再有long类型对象了，都归类为int类型。

浮点数

对于浮点数，通常情况也没有什么太神奇的，不过，有时候会遇到非常大或者非常小的浮点数，这时候通常会使用一种叫做“科学记数法”的方式表示。

```
>>> 9.8 ** -7.2
7.297468937055047e-08
```

在这个例子中，e-08表示的是10的-8次方，这就是科学记数法。当然，也可以直接使用这种方法写数字。

```
>>> a = 2e3
>>> a
2000.0
```

前面说到大整数问题的时候，Python帮我们解决了棘手的问题，使得整数可以无限大，但是，浮点数跟整数不同，它存在上限和下限，如果超出了上下的范围，就会出现溢出问题了。也就是说，如果计算的结果太大或者太小，乃至与已经不在Python的浮点数范围之内，就会有溢出错误。

```
>>> 500.0 ** 100000
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Numerical result out of range')
```

请注意看刚才报错的信息，“out fo range”，就是超出了范围，溢出。所以，在计算中，如果遇到了浮点数，就要小心行事了。对于这种溢出，需要你在编写程序的时候处理，并担当相应的责任。

当然，也要看看Python 3中的结果如何：

```
>>> 500.0 ** 100000
Traceback (most recent call last):
File "<pyshell#19>", line 1, in <module>
    500.0 ** 100000
OverflowError: (34, 'Result too large')
```

浮点数总要小心，它会因为“too large”而“out of range”。更要学会阅读程序中的报错信息，因为后面还会用到，比如除法。

[总目录](#) | [上节：集成开发环境](#) | [下节：除法](#)

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

"I give you a new commandment, that you love one another. Just as I have loved you, you also should love one another. By this everyone will know that you are my disciples, if you have love for one another."(JOHN14:34-35)

除法

除法啰嗦，不仅是Python。

更何况，Python 2和Python 3中的除法还不一样。

更啰嗦了。

所以，读者不要因为我单独列出本节而有怨言。

整数除以整数

进入Python 2交互模式之后，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？Python 2 中的麻烦出来了，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 2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。

这就是Python 2中规定的规则，不用琢磨为什么了。

在Python 3.x中，规则又变了，如果 `1/2`，结果就是`0.5`，也就是说Python 3中的除法是真正的除法了，要取整，只能用 `1//2` 的方式，即 `1//2=0`。

这就是规则，人为规定的，使用者只有顺从，就如同足球比赛的规则一样。

在Python 3中，演示几个除法：

```
>>> 5 / 2
2.5
>>> 7 / 2
3.5
>>> 8 / 2
4.0
```

要想实现类似Python 2中那样的取整除法，Python 3也是可以，这么做：

```
>>> 5 // 2
2
```

浮点数与整数相除

这里还是先讨论Python 2中的“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。

再说Python 3。

除法

前面已经看到，`5 / 2`，得到的就是浮点数 `2.5`；现在，如果当除数或者被除数，有一个或者两个都是浮点数，结果当然还是浮点数。

```
>>> 5.0 / 2
```

看来Python 3的一致性比较好。

但不管是Python 2还是Python 3，都有这种情况：

```
>>> 10.0 / 3  
3.333333333333335
```

这个是不是就有点搞怪了，按照数学知识，应该是 $3.3\overline{3}$ ，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
0.3000000000000004
>>> 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
0.1000000000000003
```

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：

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。

但是，不是什么地方都能“差不多”的。需要精确，用什么方法解决？

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 `numPy` 包和其它用于数学和统计学的包。

列出这些东西，仅仅是让读者能明白，解决问题的方式很多，不必担心。

关于无限循环小数问题，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。

并且常常有现成的“轮子”可是使用。

引用模块解决除法

`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”或者“库”，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：

除法

```
>>> 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。

当然，上述做法是在Python 2中，因为Python 3中天然如此了，没有必要再为此而 `from __future__ import division`。

这就是轮子的力量。

除法的组成有除数、被除数、商和余数，余数也可以单独计算。

余数

计算 `5/2`，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。

操作如下，不论Python 2还是Python 3：

```
>>> 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。

Python不枯燥，因为她多变。

除了使用 `%` 求余数，还有内建函数 `divmod()` ——返回的是商和余数。

```
>>> 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)
```

同样是不区分版本。

四舍五入

最后一个了，一定要坚持，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
```

如何理解 `round()` 内建函数的使用？要建立一个好习惯，并且掌握这个好方法：

```
>>> help(round)
Help on built-in function round in module builtins:

round(...)
    round(number[, ndigits]) -> number

    Round a number to a given precision in decimal digits (default 0 digits).
    This returns an int when called with one argument, otherwise the
    same type as the number. ndigits may be negative.
```

应该能读懂，我相信你。

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，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](#)。

对于计算，远不止这些，还有一个更好用的工具——math。

[总目录](#) | [上节：数和四则运算](#) | [下节：Math模块和运算优先级](#)

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

He has told you, O mortal, what is good; and what does the LORD require of you but to do justice, and to love kindness, and to walk humbly with your God?(MICAH 6:8)

世人哪，耶和华已指示你何为善，他向你所要的是什么呢？只要你行公义，好怜悯，存谦卑的心，与你的神同行。

常用数学函数和运算优先级

数学运算，不仅仅是加减乘除——四则运算是小学数学——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”(module)或者“库”(library)。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(库)默认安装了，这个称之为“标准库”，可以直接使用。如果没有纳入标准库，需要安装之后才能使用。安装方法，特别推荐使用pip来安装。

使用math

math是标准库之一，所以不用安装，可以直接使用。使用方法是：

```
>>> import math
```

不管Python 2还是Python 3，此法通用。

用import就将math引入到当前环境，下面就可以使用它提供的工具了。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', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

`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，也有人叫方法，先不计较名字。总之如果通过`math`，使用其中提供的方法（函数），能够做很多运算。

但，怎么知道每个函数如何使用？

`help()` 是好帮手。

Python是一个非常周到的姑娘，让我们来查看每个函数（方法）的使用方法。

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

不论是Python 2还是Python 3，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`中的`pow`函数的使用方法和相关说明。

1. `pow(x, y)` : 表示这个函数的参数，有两个，也是函数的调用方式
2. `Return x**y (x to the power of y)` : 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。

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`中的任何一个函数的使用方法。

下面是几个常用函数举例，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
```

使用math里的函数，已经能完成大多数基础数学的运算了。Python还嫌不够方便，还提供了几个常见的内建函数，用以数学运算。

常见函数

列举出几个常见函数。

重要声明，如果记不住这些函数也不要紧，先混个脸熟，知道有这些就好了，用的时候再google。

依然是Python 2和Python 3都适用。

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
```

随时复习。

还记得如何了解内建函数的用法吗？

```
>>> 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.
```

这么多函数，再加上加减乘除，支持数学运算的的确不少。数学的混合运算，就是把这些东西放到一个表达式，这时候大家就要区分一下，谁走先？谁最后？

这就是运算优先级。

但不是“领导先走”。

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重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
(experession,...)	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。并且，恰当使用括号，可以让你的表达式更具有可读性。

在程序中，可读性是非常重要的。

“程序”，接下来就要开始写了。

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

You are the light of the world. A city located on a hill cannot be hidden. People do not light a lamp and put it under a basket but on a lampstand, and it gives light to all in the house. In the same way, let your light shine before people, so that they can see your good deeds and give honor to your Father in heaven.(Matthew 5:14-16)

你们是世上的光。城造在山上，是不能隐藏的。人点灯，不放在斗底下，是放在灯台上，就照亮一家的人。你们的光也当这样照在人前，叫他们看见你们的好行为，便将荣耀归给你们在天上的父。

一个简单的程序

学会了四则运算，就可以编程序。

这不是开玩笑，是真的。虽然是简单的程序。

这里说的程序当然不是在交互模式中敲出的几个命令，然后看到结果。那不算编程。

也不要担心学习的东西少而不能编程，因为编程没有那么难。只要你有胆量、有毅力，就一定能写出优秀的程序。

稍安勿躁，下面就开始编写一个真正的但是简单程序。

程序

什么是程序，自维基百科记载：

[computer program and source code](#) (看不懂，很要紧，务必学习好英语，这是你认识世界工具——某国是世界一部分。想当初孙策临终前告诉孙权“外事不明学英语，内事不明学英语”，孙权谨记，才有曹孟德慨叹“生子当如孙仲谋”，因为曹丕和曹植，只学中文，不学英文，虽然这兄弟俩的诗词歌赋很有成就)。

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.

维基百科上还记载这一段中文：[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单独做编译操作（对人而言，不需要执行这个命令），因此她也被称之为解释性语言。但是，这种称呼容易产生误解。

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。

学完Python，你就知晓。

不争论。用得妙就是好。

用IDLE的编程

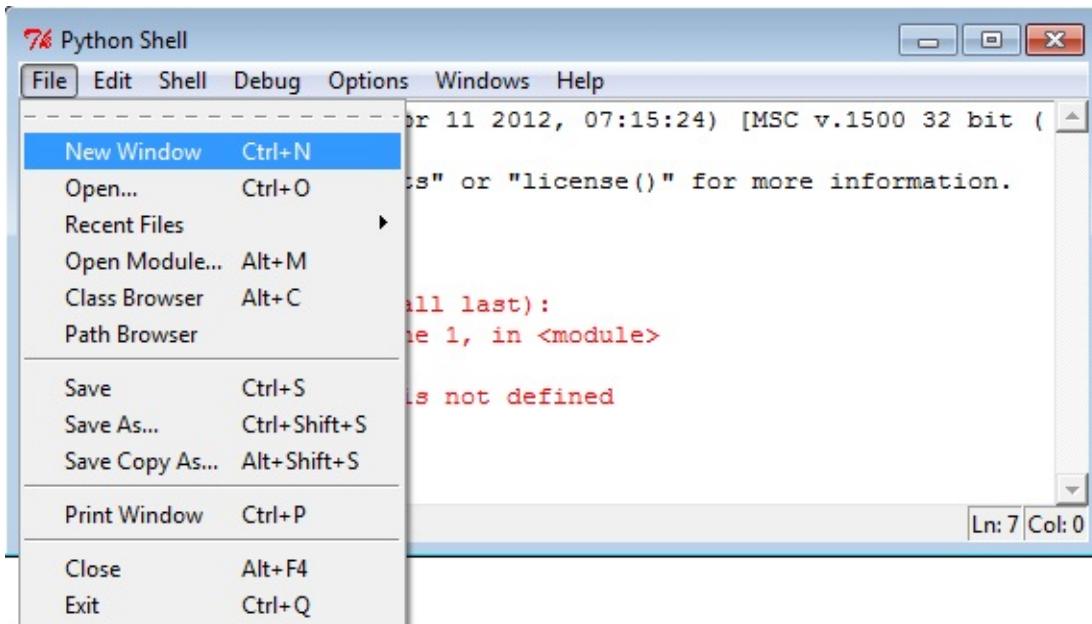
能够写Python程序的工具很多，比如记事本就可以。当然，很多人总希望能用一个专门的编程工具，Python里面自带了一个，作为简单应用是足够了。另外，可以根据自己的喜好用其它的工具，比如我用的是vim，有不少人也用eclipse，还有notepad++，等等。软件领域为编

写一个简单程序

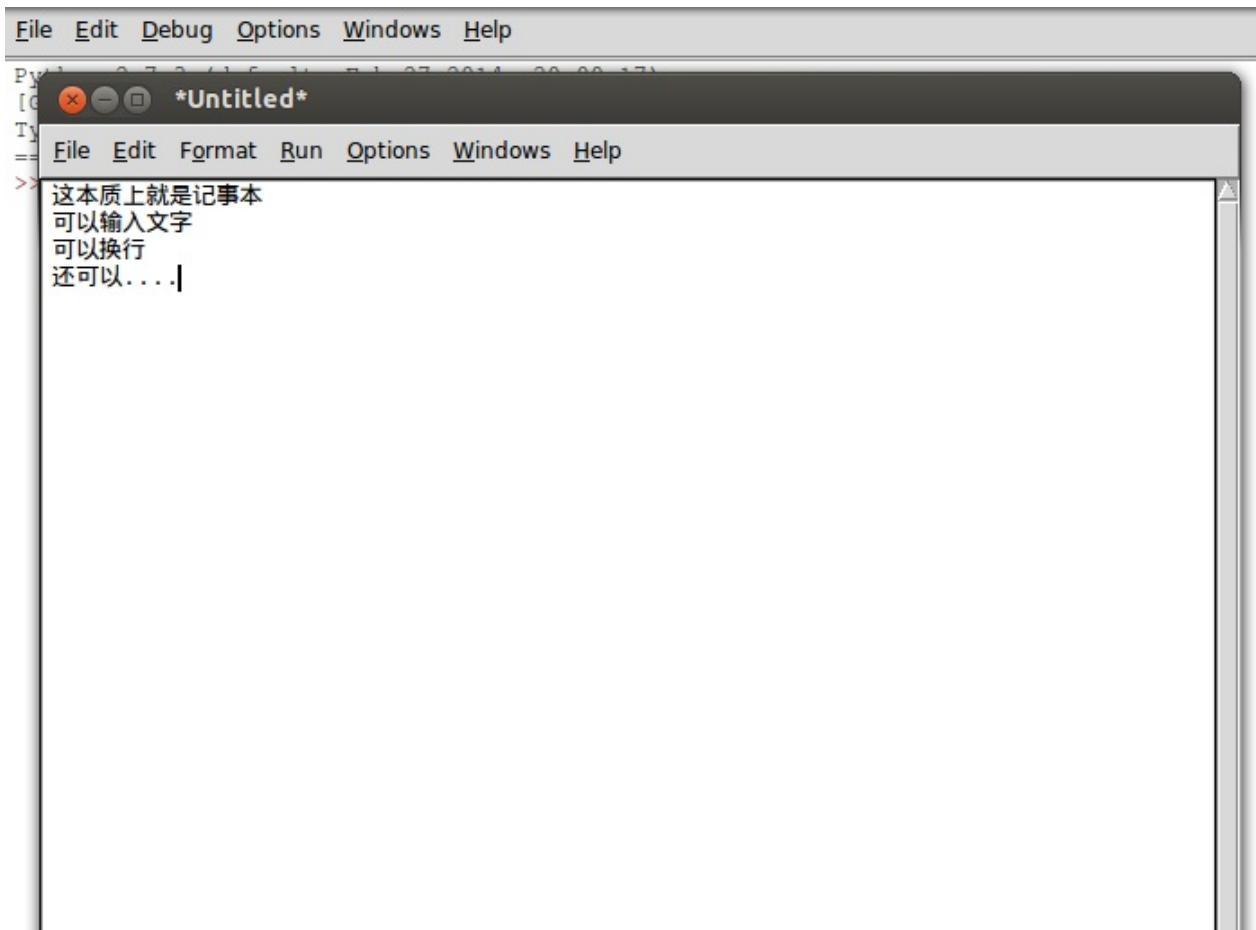
程提供了丰富多彩的工具。

以Python默认的IDLE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。



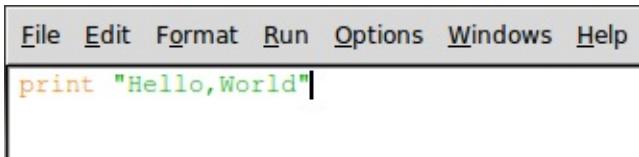
写两个大字：Hello,World

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。

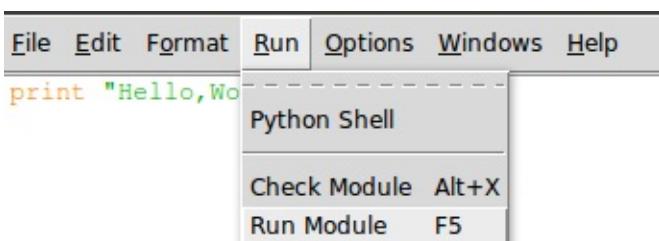
```
print "Hello,World"
```

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，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。

写一个简单程序

```
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，Python 2则输入：`python 105.py`；Python 3则输入：`python3 105.py`。

上面这句话的含义就是告诉计算机，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（我在写这段的时候，正好世界杯），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（什么时候迈出的第一步？）。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 # python 3: 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。

写一个简单程序

下面对这个简单程序进行一一解释。

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

在Linux操作系统中，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。如果是Windows操作系统，则不必写。

```
#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。不管你信还是不信，都应该把程序中的这行删掉，然后运行程序，看看什么结果？

```
"""
请计算：
19+2*4-8/2
"""
```

这一行是给人看的，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。

写一个简单程序

```
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，有一个名字，叫做赋值语句。

`19+2*4-8/2` 是一个表达式，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。指向了右边表达式计算结果。

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
```

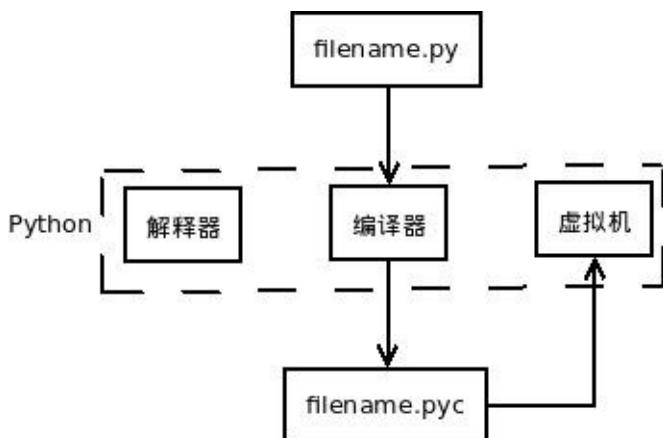
对于Python 2，这还是一个语句，称之为print语句，就是要打印出`a`的值（这种说法不是非常严格，但是通常总这么说。按照严格的说法，是打印变量`a`做对应的对象的值。嫌这种说法啰嗦，就直接说打印`a`的值）。

但是对于Python 3，应该写成 `print(a)`，这里的 `print()` 是一个函数，意思是调用这个函数，将`a`所指向的对象传给此函数。结果同上。

是不是在为看到自己写的第一个程序而欣慰呢？那么计算机是怎么完成计算过程的呢？

编译

在刚才的程序中，那些东西我们可以笼统称之为源代码，最后那个扩展名是 `.py` 的文件是源代码文件。Python是如何执行源代码的呢？



当运行 `.py` 文件的时候，Python会通过编译器，将它编译为 `.pyc` 文件。

写一个简单程序

对，你没有看错。Python中也有编译，只不过它不是你有意识单独来操作的，是你执行程序的时候自动完成的。

然后这个文件就在一个名为虚拟机的东西上运行，这个所谓的虚拟机是专门为Python设计的。

为什么要有虚拟机？

因为有了虚拟机，使得Python可以跨平台的，也就是说你写的Python程序可以不经过修改而在不同才做系统上运行。

Java也不过如此。

如果你没有修改 `.py` 文件，那么每次执行这个程序的时候，就直接运行前面已经生成的 `.pyc` 文件，这样让执行速度就大大提升了，不是每次都要从新编译。

有一些不了解或者不愿意了解Python的人，总认为Python使解释型语言，每次执行程序都要从头到位一行一行解释执行，这是对Python的无知表现。如果你修改了 `.py` 文件，下次执行程序的时候，会自动从新编译。

你根本不用关心 `.pyc` 文件，Python总是自动完成编译过程的。而且，它的代码因为使给机器看的，你也看不懂。不过要注意的是，不要删除它，也不用重命名。

程序搞定，在你感到收获的时候，不要忘了，编程的路我们刚刚开始，后面还有“字符串”。

[总目录](#) | [上节：常用数学函数和运算优先级](#) | [下节：字符串（1）](#)

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)

And since they did not see fit to acknowledge God, God gave them up to a debased mind and things that should not be done. They were filled with every kind of wickedness, evil, covetousness, malice. Full of envy, murder, strife, deceit, craftiness, they are gossips, slanderers, God-haters, insolent, haughty, boastful, inventors of evil, rebellious toward parents, foolish, faithless, heartless, ruthless. They know God's decree, that those who practice such things deserve to die--yet they not only do them but even applaud others who practice them. (ROMANS 1:28-32)

字符串(1)

计算机，当初是为了什么而发明的？

计算！

现如今，我们通常所见之范围内，都在用她做什么？

聊天！看视频！打游戏！处理文档！

显然，已经“忘记初心”，计算机再不是当初的计算机了，成为了“电脑”。也正是忘记初心，变成“电脑”，才成为大众的工具。而支持她作为“电脑”之用的，就是里面的软件。

电脑中的软件，只有少数是用来计算的，多数是不用于计算的。处理文档则是多数功能之一。

软件程序如何处理文档？要再看看自然语言。

如果对自然语言，有很多种，比如英语、法语、汉语等，这是对自然语言的分类。

我还有一种分类方法，固然尚未得到广泛认同，但“真理是掌握在少数人的手里”是我的信念，至少可以让自己成为“民科”。

来自“民科”的分类法：

1. 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），这是第一类；
2. 两个元素并排摆放在一起，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”，这是第二类。

举例：3和5拼接（就是整数求和）在一起是8，这就是第一类；如果是35，那就属于第二类。

只有抽象的原理才能是普适的，所以，用符号的方式概括上述分类：

- 第一类： $\triangle + \square = \circ$
- 第二类： $\triangle + \square = \triangle \square$

很放肆地下一个结论：人类的语言，离不开以上两类，不是第一类就是第二类。

可以鼓掌了。

之所以自鸣得意，是因为了解的太少。

字符串

google，不至于让我盲目。

维基百科的字符串词条早已经有了完整的说明，这个说明是针对一种叫做“字符串”的东西而阐述的：

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]...a[n]`。

伟大的维基百科！它已经把我煞费苦心还自鸣得意的分类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神奇的"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Python数据类型。当然，表示方式还是有区别的。

"I love Python." 'I love Python.' 'I LOVE PYTHON.' 'I LOVE PYTHON.'

不论使用单引号还是双引号，结果都是一样的，都是字符串。

Python 2的用户，可以看到：

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

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

Python 3则是这样的结果：

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

字符串(1)

仔细观察，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，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。

先看 Python 2 下的操作

```
>>> 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。

在 Python 3 中，区别就是 `print()` 是一个函数了。

```
>>> print("good good study, day day up")  
good good study, day day up
```

爱思考，有惊喜；多尝试，有收获。

如果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，就是将错误提示放到 Google 中搜索。

上面的错误原因是什么呢？

字符串(1)

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。

有解吗？有！必须有。

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，就是让某个符号不再表示某个含义，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。

变量能不能指向某个字符串？如果可以则操作会更简单。

变量和字符串

曾记否：变量无类型，对象有类型。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。

整数是对象，通过赋值语句可以设置一个变量指向整数；

字符串是对象，显然也能够通过赋值语句，实现变量指向字符串。

所以字符串对象与某个变量联系在一起：

字符串(1)

```
>>> b = "hello,world"
>>> b
'hello,world'
>>> print b          #Python 2
hello,world
#>>> print(b)        #Python 3
#hello,world
```

检查类型的函数 `type()` 总是在我们需要的时候被想起来。

```
#Python 2
>>> type(a)
<type 'int'>
>>> type(b)
<type 'str'>

#Python 3
>>> type(a)
<class 'int'>
>>> type(b)
<class 'str'>
```

有一种说法：`a`称之为数字型变量，`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。

小学语文老师布置这样一个题目：请用“不约而同”造句。

一小朋友回答：

我问姐姐，“约吗？”，姐姐说，不约儿童。

上面这句话，就是多个字符串连接到了一起。所以，字符串是可以连接起来的。

连接字符串

对数字，用 `+`，可以得到一个新的数字，如：`3+5`，就得到了 `8`。那么对字符串都能进行什么样的操作呢？试试吧：

```
>>> "py" + "thon"
'python'
```

两个字符串可以“相加”，但与数字“相加”不同。实质上是把两个字符串连接起来。

字符串(1)

```
>>> "py" - "thon"          # 在进行这种操作的时候，要斟酌一下其意义？
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           # Python 2的写法，如果是Python 3，请使用print(b + a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

报错了，其错误原因已经打印出来（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。如果是Python 3，报错信息是 TypeError: unsupported operand type(s) for +: 'int' and 'str'。

怎么办？

原来，用 + 所连接的两个对象，必须是同一种类型。如果是不同类型的，则"cannot concatenate"或者"unsupported"。

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，上面的代码，不要使用在Python 3中。

在Pytohn 2中，也要注意，一不小心就会犯错误。包裹着字母 a 的，不是单引号，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——叫做“反引号”。

这种方法，在编程实践中比较少应用，特别是在python 3中，已经把这种方式弃绝了。窃以为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)      # 如果是Python 3，请使用print(b + str(a))
free1989
```

字符串(1)

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——`str()` 是函数，关于函数，后面会详述。其实前面已经遇到过 `int()` 了。比如：

```
>>> a = "250"
>>> type(a)
<type 'str'>      # 对于Pytohn 3，返回值略有差异，前面已经演示过了。
>>> 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。或许看不懂，不用担心，权当混个脸熟。

还有第三种：

```
>>> print b + repr(a)      # 这是Python 2的写法，Python 3则为：>>> print(b + repr(a))
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。

三种解决方法，有区别吗？

首先，`repr()` 和反引号是一致的（Python 3 弃绝了反引号的使用），就不用区别了。

然后区分 `repr()` 和 `str`，不用消耗脑细胞，交给 Google，查询到这样的描述：

字符串(1)

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>

字符串中的困难，不仅在上面所述，还有更多，例如“转义符”，不少错误可能与此有关。

Python转义字符

转义符，已经小试牛刀，便显威力—— what's your name? 。

在字符串中，总会有一些特殊的符号，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。下面表格中列出常用的转义符：

字符串(1)

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，感受实际上是什么样子的。例如：

Python 2 :

```
>>> print "hello.I am qiwsir.\n#这里换行，下一行接续\n... My website is 'http://qiwsir.github.io'."\nhello.I am qiwsir.My website is 'http://qiwsir.github.io'.\n\n>>> print "you can connect me by qq\\weibo\\gmail" #\\是是为了要后面那个\\\nyou can connect me by qq\weibo\gmail
```

Python 3:

```
>>> print("hello.I am qiwsir.\nMy website is 'http://qiwsir.github.io'.")\nhello.I am qiwsir.My website is 'http://qiwsir.github.io'.
```



```
>>> print("you can connect me by qq\\weibo\\\\gmail")\nyou can connect me by qq\\weibo\\\\gmail
```

自己动手，丰衣足食，试试吧。

字符串(1)

`print` 或者 `print()` 解决了显示问题，但是输入怎么办？这就需要 `raw_input()` 或者 `input()` 粉墨登场了。

[总目录](#) | [上节：写一个简单的程序](#) | [下节：字符串（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。

字符串(2)

键盘输入

电脑的智能，一种体现就是可以接受用户通过键盘输入的内容。

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，要了解函数：

- Python 2 : `raw_input()`
- Python 3: `input()`

这是Python的内建函数（built-in function）。关于内建函数，可以分别通过下面的链接查看：

- [Python 2的内建函数](#)
- [Python 3的内建函数](#)

如果仔细对照上面的两个版本的内建函数，会发现还是有差异的。

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？

一种方法是通过网页上的官方内容，点击链接，就能查看该函数的说明文档。

还有一种方法，不知道你是否还记得我在前面使用过的，这里再进行演示。

```
>>> help(raw_input)      #Python 2
>>> help(input)         #Python 3
```

然后就出现，

Python 2中的结果：

```
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.
```

Python 3中的结果：

字符串(2)

```
Help on built-in function input in module builtins:  
  
input(prompt=None, /)  
    Read a string from standard input. The trailing newline is stripped.  
  
    The prompt string, if given, is printed to standard output without a  
    trailing newline before reading input.  
  
    If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise EOFError.  
    On *nix systems, readline is used if available.
```

从中是不是已经清晰地看到了 `raw_input()` 或者 `input()` 的使用方法了。

下面就在交互模式下操练一下这个主管键盘输入的函数。

分别在交互模式下，将这个两个函数操练一下。

```
>>> raw_input("input your name:")          # Python 2  
input your name:python                      #提示输入内容，通过键盘输入`python`  
'python'  
  
>>> input("input your name:")            #Python 3  
input your name:python                      #提示输入内容，通过键盘输入`python`  
'python'
```

输入名字之后，就返回了输入的内容。

返回的结果，也是一个对象（字符串类型的对象），那么就可以用赋值语句，与一个变量关联起来。

```
>>> name = raw_input("input your name:")      #Python 2  
input your name:python  
>>> name  
'python'  
>>> type(name)  
<type 'str'>  
  
>>> name = input("input your name:")        #Python 3  
input your name:python  
>>> name  
'python'  
>>> type(name)  
<class 'str'>
```

而且，返回的结果是 `str` 类型。如果输入的是数字呢？

字符串(2)

```
>>> age = raw_input("How old are you?")           #Python 2
How old are you?10
>>> age
'10'
>>> type(age)
<type 'str'>

>>> age = input("How old are you?")            #Python 2
How old are you?10
>>> age
'10'
>>> type(age)
<class 'str'>
```

返回的结果，仍然是str类型。

所以，Python 2的文档中就明确写出 `raw_input([prompt]) -> string`，意思是它的返回值为字符串。

`print()` 在Python 2和Python 3中，都是一个函数。

特别要提醒的是，`print()` 默认是以 `\n` 结尾的，所以，每次用到 `print` 或者 `print()` 之后，输出内容后面自动带上了 `\n`，于是在打印的结果中就换行了。

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?")          #如果是在Python 3中，更换为input()
age = raw_input("How old are you?")

print "Your name is: ", name                      #Python 3: print("Your na
me is: ", name)
print "You are " + age + " years old."           #Python 3: print("You are " + ag
e + " years old.")

after_ten = int(age) + 10
print "You will be " + str(after_ten) + " years old after ten years."
#Python 3: print("You will be " + str(after_ten) + " years old after ten years.")
```

读者是否能独立调试这个程序？

`print` 语句或者 `print()` 函数，除了打印一个字符串之外，还可以打印字符串拼接结果（拼接之后还是一个字符串，就是比原来长了）。

```
print "You are " + age + " years old."          #Python 2
print("You are " + age + " years old.")          #Python 3
```

注意，那个变量 `age` 必须指向的是字符串类型的对象，如最后的那个语句中：

字符串(2)

```
print "You will be " + str(after_ten) + " years old after ten years."      #Python  
2  
print("You will be " + str(after_ten) + " years old after ten years.")        #Python  
3
```

这句话里面，有一个类型转化，将原本是整数型的对象转化为了str类型。否则，就报错，不信，你可以试试。

同样注意，在 `after_ten = int(age) + 10` 中，因为通过 `raw_input()` 或者 `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。请仔细调试一下，如果没有通过，看报错信息，你能够从中获得修改方向的信息。

通过键盘输入得到的都是字符串，也有的字符串不是通过键盘输入得到的，需要用引号包裹，有时候还要用转义符。但是，有一种方式，能够还原字符串中字符的原始含义。

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。

在一般字符串中，比如

```
>>> print "I like \npython"          #Python 3: 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` 一起组成了换行符 `\n`，即转义了。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  
#Python 3: print(dos)
```

如何避免？

用转义符可以解决：

字符串(2)

```
>>> dos = "c:\\news"
>>> print dos          #Python 3: print(dos)
c:\\news
```

此外，还有一种方法，如：

```
>>> dos = r"c:\\news"
>>> print dos          #Python 3: print(dos)
c:\\news
>>> print r"c:\\news\\python"    #Python 3: 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。

一个字符串，一般可以有多个字符构成，那么可以操作每个字符吗？这就要索引和切片。

[总目录](#) | [上节：字符串\(1\)](#) | [下节：字符串（3）](#)

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)

你们又听见有吩咐古人的话，说：“不可背誓，所起的誓，总要向主谨守”。只是我告诉你们，什么誓都不可起，不可指着天起誓，因为天是神的座位。不可指着地起誓，因为地是他的脚蹬，也不可指着耶路撒冷起誓，因为耶路撒冷是大君的京城。又不可指着你的头起誓，因为你不能使一根头发变黑变白了。你的话，是，就说是。不是，就说不是。若再多说，就是出于那恶者。(Matthew 5:33-37)

字符串(3)

字符串是一个话题中心，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，就变成一个新字符串了；如果这些字符顺序发生变化了，也成了一个新字符串。

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

梁山好汉，从1排到108，就是索引。

再看Python的字符串：

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

有一个字符串，要得到这个字符串的第一个单词 `s`，可以用 `lang[0]`。当然，如果你不愿意让变量`lang`来指向那个字符串，也可以这样做：

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

字符串(3)

效果是一样的。因为`lang`是标签，就指向了 "study python" 字符串。当让执行`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		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。

空格也是一个字符。“无”不完全等于“没有”。

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？

用字符串的一个方法——`index`：

```
>>> 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？但有区别，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，我念到的兄弟站出来。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。”

继续应用前述字符串，`lang[1]` 能够得到字符串的第二个字符`t`，就相当于从字符串中把这个“切”出来了。不过，我们这么“切”却不影响原来字符串的完整性，当然可以理解为将那个字符`t`复制一份拿出来了。

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]`要得到多个（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

字符串(3)

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——前包括，后不包括。

上述，不管是得到一个还是多个，通过索引范围得到字符的过程，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，如果冒号的：

- 前面不写数字，就表示从字符串的第一个开始（包括第一个）；
- 后面的序号不写，就表示到字符串的到最末一个字符结束（包括最后一个）。

`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'
```

不一样。

原因就是前述所说的，如果冒号后面有数字，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'
```

字符串(3)

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，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？用 `id()` 函数查看便知。

字符串有索引，能得到切片。不仅如此，还有更多操作。

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，这是序列共有的操作。

1. `len()` : 求序列长度
2.
 - : 连接2个序列
3.
 - : 重复序列元素
4. `in` : 判断元素是否存在与序列中
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

逐个演示，方能理解：



字符串(3)

```
>>> 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另外一个字符串（这个字符串被称为子字符串），如果包含，就返回 `True`，否则返回 `False`。

最大值和最小值

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

在英文字典中，所有的字母都有一个排序，我们称之为“字典顺序”。

而每个字符，也都通过编码对应着一个数字，它们都会有一定的顺序。`min()` 和 `max()` 就是根据这个顺序获得最小值和最大值，然后对应出相应的字符。读者可以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应编码的数字，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。

字符串(3)

```
>>> 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。能不能根据刚才阐述的比较理解呢？

```
>>> 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
```

如果读者阅读到这里，不知道你是否将上面的各项进行了实际操作？如果操作了，在Python 3中能成功吗？

上面的操作只能适用于Python 2，不适用Python 3。

字符串(3)

Python 3中取消了 `cmp()` 函数。那么在Python 3中怎么比较呢？

```
>>> str1 = "abc"
>>> str1 > str2
False
>>> str1 < str2
True
>>> str1 == str2
False
```

用比较运算符，也可以得到比较结果。

“*”

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。

Python用 `len()` 函数来获得字符串长度，不管Python 2还是Python 3。

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

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，Python 3中返回的结果略有差别。
<type 'int'>
```

对于字符串，作为序列的一种，除了具有上述几种通用的基本操作之外，还有很多别的方法。

[总目录](#) | [上节：字符串\(2\)](#) | [下节：字符串（4）](#)

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)

所以你施舍的时候，不可在你前面吹号，像那假冒为善的人，在会堂里和街道上所行的，故意要得人的荣耀。我实在告诉你们，他们已经得了他们的上次。你施舍的时候，不要叫左手知道右手所作的。要叫你施舍的事行在暗中，你父在暗中察看，必然报答你。(MATTHEW 6:2-4)

字符串(4)

虽然已经对字符串有了了解，但，因为人对字符串的要求比较多，特别是在输出格式上，要满足一定的样式要求。

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”，表示的意思是：

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——print语句（Python 2）、print()函数（Python 3），此前我们已经了解一二了。此处将详述之。

如果将维基百科的定义再通俗化，所谓字符串格式化化，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，并且在显示结果中，字符串要符合空位置所设定的约束条件。

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：

字符串(4)

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

这是曾经较为常用的一种字符串输出方式。注意“曾经”，言下之意，现在不怎么太提倡了。

的确如此，现在提倡使用 `.format()`，这是自Python 2.6开始引入的。所以，从现在开始，本教程详细介绍 `string.format()` 的使用方法，而对用 `%` 进行格式化输出的方式，仅仅局限在上面的样式。读者在阅读其它代码的时候，也能遇到使用 `%` 的，那时候你心中默默说一句“落伍了”，然后翻译成 `string.format()` 即可。

`.format()` 是字符串的一个方法。你在交互模式中，输入 `dir(str)`，会看到如下的内容：

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
 '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mu-
l__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__r-
mul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'cas-
efold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'form-
at_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islow-
er', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lo-
wer', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpar-
tition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

这里所列出来的，就是字符串 `str` 的所有属性和方法。

在这里先下一点毛毛雨。在Python中一切皆对象，所谓对象，就是一个具体东西。桌椅板凳，花花草草，动物植物，包括人，还有程序要找的那个“对象”，都是对象（不用追求严格定义，暂且模模糊糊知道大概即可，大概都不知道，也无妨），凡是对象都有属性和方法（至于属性和方法的定义，待后面说明）。刚才用 `dir(str)` 所返回的结果，就是字符串的属性和方法。

在返回结果中，别的先不看，只看 `format`。有你的慧眼找一找，有没有发现？

继续不要离开交互模式，输入：

```
>>> help(str.format)
```

这是要看字符串的 `format()` 方法的文档。只有通过阅读文档，我们才能了解它使做什么的。

返回结果是：

字符串(4)

```
Help on method_descriptor:

format(...)
    S.format(*args, **kwargs) -> str

    Return a formatted version of S, using substitutions from args and kwargs.
    The substitutions are identified by braces ('{' and '}').
```

`S.format(*args, **kwargs)`，重点看括号里面的，`*args` 表示传入一种类型的参数，`**arg` 表示传入另外一种类型的参数。

又遇到新名词——参数——还是不用搭理它，只管继续——因为“发展是硬道理”！

你暂且阅读文档，不理解也没关系。下面用示例来说明使用方法。

```
>>> "I like {0} and {1}".format("python", "canglaoshi")
'I like python and canglaoshi'
```

在交互模式中，输入了字符串 `"I like {0} and {1}"`，并且其中用 `{0}` 和 `{1}` 占据了两个位置，它们就是占位符。然后使一个非常重要的点`.`。

这个点`.`非常重要，它是从对象引出它的属性或方法的工具，就如同汉语中的“的”一样。

比如“老齐的爱好”，“老齐”是那个对象，“爱好”就是一个方法，两者中间的那个“的”就相当于一个英文的点`.`，于是可以写成 `老齐.爱好`，就可以表明上述的含义了。

`format("python", "canglaoshi")` 是字符串格式化输出的方法，传入了两个字符串，它们分别对应这 `"I like {0} and {1}"` 里的那两个占位符，而且使按照顺序对应的，即第一个参数传入的 `"pytohn"`，对应着 `{0}`，第二参数传入的 `"canglaoshi"` 对应着 `{1}`。

你还可以这样试试，就理解更深刻了。

```
>>> "I like {1} and {0}".format("python", "canglaoshi")
'I like canglaoshi and python'
```

请仔细观察找区别。

`format()` 方法的返回值是一个字符串——`'I like python and canglaoshi'`。

再对照 `help(str.format)` 得到的文档，看一看，是不是理解文档的含义了呢？

一定要会阅读文档，这是学习语言的根本。

既然是“格式化”，就要指定一些格式，让输出的结果符合指定的样式。

```
>>> "I like {0:10} and {1:>15}".format("python", "canglaoshi")
'I like python      and      canglaoshi'
```

字符串(4)

现在有格式了。`{0:10}` 表示第一个位置，有10个字符那么长，并且放在这个位置的字符是左对齐；`{1:>15}` 表示第二个位置，有15个字符那么长，并且放在这个位置的字符是右对齐。

```
>>> "I like {0:^10} and {1:>15}".format("python", "canglaoshi")
'I like    python    and    canglaoshi  '
```

现在是居中对齐了。

```
>>> "I like {0:.2} and {1:^10.4}".format("python", "canglaoshi")
'I like py and      cang      '
```

这个有点复杂，我们一点一点的解释。

`{0:.2}` : 0 说明是第一个位置，对应传入的第一个字符串。`.2` 表示对于传入的字符串，截取前两个，并放到第一个位置，注意的是，在`:`后面和`.`号前面，没有任何数字，意思是该位置的长度自动适应即将放到该位置的字符串。

`{1:^10.4}` : 1 说明是第二个位置，对应传入的第二个字符串。`^` 表示放到该位置的字符串要居中；`10.4` 表示该位置的长度是10个字符串那么长，但，即将放入该位置的字符串应该仅有4个字符那么长，也就是要从传入的字符串`"canglaoshi"` 中截取前四个字符，即为`"cang"`。

再看结果，对照上述解释。

向`format()` 中，除了能够传入字符串，还可以传入数字（包括整数和浮点数），而且也能有各种花样。

```
>>> "She is {0:d} years old and the breast is {1:f}cm".format(28, 90.1415926)
'She is 28 years old and the breast is 90.141593cm'
```

`{0:d}` 表示在第一个位置放一个整数；`{1:f}` 表示在第二个位置放一个浮点数，那么浮点数的小数位数，是默认的。下面在这个基础上，可以再做一些显示格式的优化。

```
>>> "She is {0:4d} years old and the breast is {1:6.2f}cm".format(28, 90.1415926)
'She is    28 years old and the breast is   90.14cm'
```

`{0:4d}` 表示第一个位置的长度是4个字符，并且默认状态下，填充到该位置的整数是右对齐。

`{1:6.2f}` 表示第二个位置的长度使6个字符，并且填充到该位置的浮点数要保留两位小数，默认也是右对齐。

```
>>> "She is {0:04d} years old and the breast is {1:06.2f}cm".format(28, 90.1415926)
'She is 0028 years old and the breast is 090.14cm'
```

字符串(4)

{0:04d} 和 {1:06.2f} 与前述例子不同的在于在声明位置长度的数字前面多了 0，其含义是在数字前面，如果位数不足，则补 0。

以上的输出方式中，我们只讨论了 `format(*args, **kwargs)` 中的 `*args` 部分。还有另外一种方式，则是与 `**kwargs` 有关的（关于这两种参数的含义，本教程后面有专门介绍）。

```
>>> "I like {lang} and {name}".format(lang="python", name="canglaoshi")
'I like python and canglaoshi'
```

一种被称为“字典格式化”，这里仅仅列一个例子。关于“字典”，本教程后续会有的。

```
>>> data = {"name":"Canglaoshi", "age":28}
>>> "{name} is {age}".format(**data)
'Canglaoshi is 28'
```

用 `format()` 做字符串格式化输出，真的很简洁，堪称优雅。

但 `format()` 毕竟只是字符串的方法之一，还有更多方法等待研究。

常用的字符串方法

字符串的方法很多，前面已经通过 `dir(str)` 查看过了。

那么多方法，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()` 函数查看。再举例：

```
>>> 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
```

根据分隔符分割字符串

`split()` 的作用是将字符串根据某个分割符进行分割。

字符串(4)

```
>>> 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

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](#)。

在Python中有下面一些字符串的方法，用来实现各种类型的大小写转化

字符串(4)

- `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()` : S中字符串中所有的单词拼写首字母是否为大写，且其他字母为小写（标题都这么写）

看例子：

字符串(4)

```
>>> 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()      #将所有的字母变成小写字母
>>> 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"
>>> 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
```

再探究一下，可以这么做：

字符串(4)

```
>>> 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
```

用 .join() 拼接字符串

用 + 能够拼接字符串，除了这个还有别的。

当然，+ 也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() 方法拼接字符串，也是一个好选择。

```
>>> 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？

虽然不能把所有方法穷尽，但读者完全可以仿照上述流程研究其它方法。

字符串的问题还要继续，因为中文和英文还有很大区别。

[总目录](#) | [上节：字符串\(3\)](#) | [下节：字符编码](#)

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

So do not worry about tomorrow, for tomorrow will bring worries of its own. Today's trouble is enough for today. (MATTHEW 6:34)

字符编码

其实，标题前面应该加两个字——“坑爹”。

在实践中，字符编码的确是一个“坑”。因为这个世界上，不都是英文。如果都是英文，就没有这个问题了。可是，还有中文、日文等等。

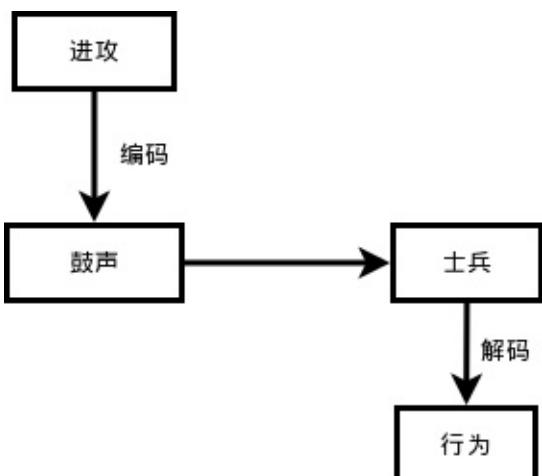
但是，字符编码的确很重要，它不仅仅是计算机的一个基础，还是一个有历史过程的事情。

要从编码开始谈起。

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。李鸿章說：“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。

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。

但计算机的字符编码，不是一蹴而就，而是有一个发展过程的。

ASCII码

计算机里采用二进制，这是毋庸置疑不用解释的了。

20世纪60年代，这是计算机发展的早期，那时候美国是很多领域的老大，计算机上也同样是老大，当然现在也还是老大，未来是不是就要看Chinese People了。老大就要做老大的事情，定规矩肯定是老大的事情，于是美国制定了一套字符编码，解决了英语字符与二进制位之间的对应关系，被称为ASCII码。

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。

英语用128个符号编码就够了，但计算机不是仅仅用于英语。如果用来表示其他语言，128个符号是不够的。于是很多其它国家，都在ASCII码的基础上，搞了很多别的编码，比如汉语里面有了简体中文编码方式GB2312，使用两个字节表示一个汉字。

Unicode

编码方式上，各玩个的，就有点乱，于是就出现了“乱码”。比如电子邮件，发信人和收信人使用的编码方式不一样，收信人就只能看“乱码”了。

网络的发展，让地球都成为一个村了，同一个村里面就不能有很多“方言”，只能有一种，否则“乱了”。

于是[Unicode](#)(呼之出来了，看它的名字，你也知道，就是要统一符号的编码)。

[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。

但[Unicode](#)也不是完美的，存在一些问题。想了解哪些问题，请具体查阅参考文献：[字符编码笔记：ASCII，Unicode和UTF-8](#)

UTF-8

互联网催生了[UTF-8](#)。

[Unicode](#)的实现方式称为[Unicode](#)转换格式（[Unicode Transformation Format](#)，简称为[UTF](#)）——[UTF](#)的含义。

[UTF-8](#))是在互联网上使用最广的一种[Unicode](#)的实现方式。虽然它仅仅是[Unicode](#)的实现方式之一，但它真正一统江湖了。

[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。

有[UTF-8](#)，言外之意还应该有[UTF-n](#)，n是一个别的数字。

的确如此，还有[UTF-16](#)等等，但[UTF-8](#)有很多优点，被广泛接受。

所以，以后，我们在[Python](#)的程序开发中，都要使用[UTF-8](#)编码。

注：参考文献：[字符编码笔记：ASCII，Unicode和UTF-8](#)

看完了一些关于编码的基本知识，再来看[Python](#)中的编码问题。

Python字符编码

[Python](#)编码容易让人迷茫，因为[Python 2](#)和[Python 3](#)还有区别。

如果你在[Python 2](#)中，在交互模式里按照下面的指令执行。

字符编码

```
>>> import sys  
>>> sys.getdefaultencoding()  
'ascii'
```

这说明Python 2的默认是ASCII编码。而Python 3，则不同。

```
>>> import sys  
>>> sys.getdefaultencoding()  
'utf-8'
```

所以，要注意是哪个版本。

之所以如此，根源是Python 2横空出世的时候，Unicode还没有来到这个世界。

在Python中，有两个内建函数，能够实现字符和对应数字之间的转换。

```
>>> ord("Q")  
81  
>>> chr(81)  
'Q'
```

对于英文字母，不同版本的Python没有区别，但是，对于汉字就有区别了。

先看Python 2。如果你用Python 3，此段可以略过。

```
>>> ord("齐")  
  
Traceback (most recent call last):  
  File "<pyshell#4>", line 1, in <module>  
    ord("齐")  
TypeError: ord() expected a character, but string of length 2 found
```

Python 2默认是ASCII编码，齐这个字符已经超出了ASCII的范围，所以要报错。并且，从报错信息中可以看出更多信息。

```
>>> help(ord)  
Help on built-in function ord in module __builtin__:  
  
ord(...)  
    ord(c) -> integer  
  
    Return the integer ordinal of a one-character string.
```

Python 2中，要求ord()的参数所传递的是一个字符，即长度是1，而Python解释器现在认为齐这个汉字是两个字符的长度。

的确如此，汉字占据了两个字节。

```
>>> s = "齐"
>>> len(s)
2
>>> s
'\xc6\xeb'
```

再看Python 3，进行上述操作。

```
>>> ord("齐")
40784
>>> chr(40784)
'齐'

>>> help(ord)
Help on built-in function ord in module builtins:

ord(c, /)
    Return the Unicode code point for a one-character string.
```

区别已经很明显了。

因此，Python 2做了扩展，使用一种新的方式，来声明那个字符串是Unicode编码。

```
>>> t = u'齐'
>>> len(t)
1
>>> t
u'\u9f50'
>>> ord(t)
40784
>>> type(t)
<type 'unicode'>
```

所以，使用Python 2的朋友们要注意了，在编程中，将字符串写成状如 `u'齐'` 是非常必要的，因为这样你使用的就是Unicode编码。

这样，貌似世界就和谐了。

真的吗？

encode和decode

因为种种原因，不可能世界上所有人都按照同一种模式生活，否则就不是“绚丽多彩”的世界了。

“绚丽多彩”的世界，就必须要有encode和decode

- encode：编码

- decode：解码

字符串也有这样两个方法，分别负责编码和解码工作。

在Python 2中，它们分别是 `str.encode()` 和 `str.decode()`。

```
>>> help(str.encode)
Help on method_descriptor:

encode(...)
S.encode([encoding[,errors]]) -> object

Encodes S using the codec registered for encoding. encoding defaults
to the default encoding. errors may be given to set a different error
handling scheme. Default is 'strict' meaning that encoding errors raise
a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and
'xmlcharrefreplace' as well as any other name registered with
codecs.register_error that is able to handle UnicodeEncodeErrors.

>>> help(str.decode)
Help on method_descriptor:

decode(...)
S.decode([encoding[,errors]]) -> object

Decodes S using the codec registered for encoding. encoding defaults
to the default encoding. errors may be given to set a different error
handling scheme. Default is 'strict' meaning that encoding errors raise
a UnicodeDecodeError. Other possible values are 'ignore' and 'replace'
as well as any other name registered with codecs.register_error that is
able to handle UnicodeDecodeErrors.
```

而在Python 3中，它是 `str.encode()`。（细心的读者，有没有看到我叙述上的差别？）

```
>>> help(str.encode)
Help on method_descriptor:

encode(...)
S.encode(encoding='utf-8', errors='strict') -> bytes

Encode S using the codec registered for encoding. Default encoding
is 'utf-8'. errors may be given to set a different error
handling scheme. Default is 'strict' meaning that encoding errors raise
a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and
'xmlcharrefreplace' as well as any other name registered with
codecs.register_error that can handle UnicodeEncodeErrors.
```

在两个版本中，是有区别的，敬请认真阅读文档信息。并且，在Python 3中不再提供 `str.decode()` 方法。想想也是合理的。因为Python 3中默认是UTF-8，就没有必要再解码成别的了吧，那不是要开历史的倒车吗？

所以，在Python 2中，你可以这么玩。

```
>>> a = "赵国"
>>> type(a)
<type 'str'>
>>> len(a)
4
>>> a
'\xd5\xd4\xb9\xfa'
```

这是一个基于ASCII得到的字节串，一个汉字两个字节。

```
>>> b = u'赵国'
>>> type(b)
<type 'unicode'>
>>> len(b)
2
>>> b
u'\u8d75\u56fd'
```

这是基于Unicode的字符串。两个术语：“字节串”和“字符串”，通过这个例子是否有了明晰？

```
>>> c = b.encode("utf-8")
>>> type(c)
<type 'str'>
>>> len(c)
6
>>> c
'\xe8\xb5\xb5\xe5\x9b\xbd'
```

这是其转化为UTF-8编码。请注意，以上三种不同编码下的长度，是不一样的。

```
>>> d = c.decode("utf-8")
>>> type(d)
<type 'unicode'>
>>> len(d)
2
>>> d
u'\u8d75\u56fd'
```

这样又变回去了。

这就是编码的转换方式。

关于编码问题，先到这里，点到为止吧。在编程实践中，如果有纠缠不清的问题，请尽情google，即可解决。

Python 2中如何避免中文是乱码

字符编码

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')
```

经验四：声明字符串直接加u，声明的字符串就是unicode编码的字符串

```
a = u"中"
```

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](#)

至于Python 3，因为天生就是UTF-8，所以对中文友好的。

关于字符串差不多要告一段落了。但是，Python的对象类型，还要继续。接下来“苦力”即将登场。

[总目录](#) | [上节：字符串\(4\)](#) | [下节：列表\(1\)](#)

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)

你们不要论断人，免得你们被论断。因为你们怎样论断人，也必怎样被论断。你们用甚么量器量给人，也必用甚么量器量给你们。(MATTHEW 7:1-2)

列表(1)

已经知道了数字和字符串，用 `type()` 可以得到具体某个对象的类型。

这是两种很基本的对象，由它们可以组成其它的对象。

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。

还学习了变量，如果某个变量指向一个对象，行话是“赋值”。

从现在开始学习一种新的对象——`list`（列表）。下面的文字，请注意了：

列表在**Python**中具有非常强大的功能。

定义

在Python中，用方括号表示一个列表——`[]`。

在方括号里面，可以是数字（整数、浮点数），也可以是字符串，甚至也能够是True/False这种布尔值。

先定义一个空列表看一看。

```
>>> a = []          #定义了一个空列表，并把他赋值给变量a。
>>> type(a)
<type 'list'>      #type()查看变量a所指向对象的类型
>>> bool(a)
False
>>> print 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，列表中什么也没有，就是空的，用 `bool()` 函数来判断，返回False，从而显示它是空的。

不能总玩空的，来点实的吧。

列表(1)

```
>>> 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，定义一个列表类型的对象。

从刚才的例子中，读者是否注意到，列表里面的元素，可以是不同类型的对象，可谓是“有容乃大”，不仅如此，它的元素个数还可以无限大，就是说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Python列表相似的数据类型——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，也就是要么都是整数类型，要么都是字符类型等，不能一个数组中既有整数类型又有字符类型。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中的列表，尽管跟Java中的数组有类似的地方——都是`[]`包裹的——列表中的元素是任意类型的。所以，有一句话说：列表是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，曾经有“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。

列表(1)

```
>>> 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]
['2', 3]
>>> a[1:]
[3, 'qiwsir.github.io']
>>> a[1:2]
[3]
>>> a[2][7:13]    #可以对列表元素做2次切片
'github'
```

列表和字符串两种类型的对象，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
```

我们已经知道，在Python中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']
```

列表(1)

序列的切片，一定要左边的数字小于右边的数字，`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。

还是通过举例来演示反转的方法：

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

对于字符串也可以：

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

是否注意到，不管是字符串还是列表，反转之后，都没有影响原来的对象。

这说明，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列表反转，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
```

列表(1)

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。

操作列表

刚刚提到过，列表是序列。所有的序列，都有几种基本操作。列表也当然如此。

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`

还是前面的列表，

列表(1)

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

- `max()`和`min()`

按照元素的字典顺序进行比较的。

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

- `cmp()`：

跟字符串中提到的 `cmp()` 一样，这个函数仅适用于 Python 2，在 Python 3 中已经被抛弃了。

引用[官方文档] (<https://docs.python.org/2.7/library/functions.html#cmp>)一段话，说明这个函数的意义：

Compare the two objects `x` and `y` and return an integer according to the outcome. The return value is negative if `x < y`, zero if `x == y` and strictly positive if `x > y`.

不用翻译你也能看懂，况且还有例子：

```
>>> 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

列表(1)

```
>>> a = ["good", "python", "I"]
>>> a
['good', 'python', 'I']
>>> a.append("like")          #向列表中追加字符串"like"
>>> a
['good', 'python', 'I', 'like']
>>> a.append(100)            #向列表中追加整数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]`.

所谓追加，即将新的元素加到列表的尾部。

如果您仔细阅读了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]`。这也相当于告诉了我们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),即得到列表的长度
>>> a
['good', 'python', 'I', 'like', 100, 3]
>>> len(a)
6
>>> a[6:]=['xxoo']
>>> a
['good', 'python', 'I', 'like', 100, 3, 'xxoo']
```

到这里，仅仅是列表这座冰山的一角，既然它是“苦力”，可以干的活还很多。

[总目录](#) | [上节：字符编码](#) | [下节：列表\(2\)](#)

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

列表(2)

你们祈求，就给你们；寻找，就寻见；叩门，就给你们开门。因为凡祈求的，就得着；寻找的，就寻见；叩门的，就给他们开门。

所以无论何事，你们愿意人怎样待你们，你们也要怎样待人，因为这就是律法和先知的道理。(MATTHEW 7:7-8,12)

列表(2)

“列表是Python的苦力”，那么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或方法了。

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getslice__',
 '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__',
 '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__setslice__',
 '__sizeof__', '__str__', '__subclasshook__', 'append', 'count', 'extend', 'index',
 'insert', 'pop', 'remove', 'reverse', 'sort']
```

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'
```

这几个都是在编程实践中常常要用到的。

常用的列表函数

append和extend

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

除了将元素追加到列表中，还能够将两个列表合并，或者说将一个列表追加到另外一个列表中。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列表来扩充它，相当于 `a[len(a):] = L`。

列表(2)

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']
```

变量 `la` 指向一个列表对象；变量 `lb` 也指向一个列表对象。为了简单，就说成 `la` 和 `lb` 两个列表。

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
```

仔细观察，能看出什么来吗？

原来，如果 `extend(str)` 的时候，字符串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。

`extend` 的对象是一个列表，如果是字符串，则 Python 会先把它按照字符为单位转化为列表再追加到已知列表后面。

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` 是待并入的列表。

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

列表(2)

```
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。

该文档中出现了**iterable**，什么是**iterable**？这个从现在开始，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
#Python2返回的结果。如果是Python3返
回True.
```

这里用内建函数 `hasattr()` 判断一个字符串是否是可迭代的，在Python 2中返回了 `False`，在Python 3中返回了 `True`。那么，这里似乎有一个矛盾的命题，一个字符串，在不同的Python版本中，为什么不一样呢？请继续阅读。

用同样的方式可以判断：

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

列表(2)

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

如果你使用的是 Python 2，在 `dir(str)` 是无法发现 `__iter__` 的。但是，在 Python 3 中，则可以在 `dir(str)` 的结果中看到 `__iter__`。这也是为什么在 Python 3 中，`hasattr(astr, '__iter__')` 返回 `True` 的原因。

将前面的所有对于字符串的操作，你连贯起来看一下，在 Python 2 中，不认为它是可迭代的，这是针对字符串本身而言，然而如果对它进行了应用于可迭代对象的操作，它又能正常进行，因为 Python 把字符串做了自动转化；因此 Python 3 中干脆顺水推舟，把这个过程一气呵成。让它也具有 `__iter__` 属性了。

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]
```

还要关注列表 `lst` 的变化。`lst` 经过 `extend` 函数操作之后，变成了一个貌似“新”的列表。这句话有点别扭，“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
```

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

列表(2)

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。

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

它把一个字符串 `"itdiffer"` 转化为 `['i', 't', 'd', 'i', 'f', '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。

这里讲述的两个让列表扩容的函数 `append()` 和 `extend()`，它们的共同点是“都能原地修改列表”。

对于“原地修改”还应该增加一个理解——没有返回值。

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
:
>>> print another                                #打印变量another的值。如果是Python3则输入p
print(another)                                    #返回为None，one.extend()没有返回值，即是N
None
one.
>>> one
['good', 'good', 'study', 'day', 'day', 'up']
```

`append()` 和 `extend()` 的区别呢？看下面例子：

列表(2)

```
>>> 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

count() 是一个帮着我们弄清楚列表中元素重复出现次数的方法。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)          #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。

列表(2)

```
>>> 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
```

`x` 是列表中的一个元素，`list.index(x)` 能够检索到该元素在列表中第一次出现的位置。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？

中场休息，下节继续列表的方法。

[总目录](#) | [上节：列表\(1\)](#) | [下节：列表\(3\)](#)

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(3)

"Come to me, all you that are weary and are carrying heavy burdens, and I will give you rest. Take my yoke upon you, and learn from me; for I am gentle and humble in heart, and you will find rest for your souls. For my yoke is easy, and my burden is light."
(MATTHEW 21:28-30)

“凡劳苦担重担的人，可以到我这里来，我就使你们得安息。我心里柔和谦卑，你们当负我的轭，学我的样式，这样，你们心里就必得享安息。因为我的轭是容易的，我的担子是轻省的。”

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

append() 或者 extend() 都是向列表中追加元素，“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。只不过是在任何位置增加一个元素。

[官方文档](#)如是说：

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：

列表(3)

```
>>> 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` 插入到列表中的位置，即将 `x` 插入到索引是 `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5，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。

只不过，这样做的不多罢了。

最后，还要关注，`insert()` 也是对列表原地修改，没有返回值，或者说返回值是`None`。

pop和remove

列表中的元素，不仅能增加，还能被删除。删除列表元素的方法有两个，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.)

读者如果一直跟着我的节奏在学习，应该体会到我们这里的一种学习方法了——先实验，然后总结规律。这是一种物理学的研究方法。

物理学，是科学的基础，特别是它所演化出来的科学的研究方法，更是人类智慧的瑰宝。不忘初心，我是大物理系毕业的。

先实验 `list.remove(x)`，注意看上面的描述。这是一个能够删除列表元素的方法，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']
```

在`all_users`所指向的列表中删除一个元素，一切都符合文档说明的要求，很顺利地完成了。

```
>>> 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
```

如果列表中没有那个元素，非要删除不可，肯定报错。而且报错信息非常明确指出 `x not in list`。这也是文档中已经陈述过的了。

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

仔细观察，变量的名字 `lst`，不是 `list`，最好不要用 `list` 作为变量名字，因为它是Python中内置的对象类型的名字，如果你非要用它做变量名字，很可能引起后续的麻烦。

列表(3)

再仔细观察，这个列表中有两个'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 ...**

所以，对 `remove()` 总结两点：

- 如果正确删除，则删除第一个符合条件的对象。不会有任何反馈。没有消息就是好消息。它是对列表进行原地修改。
- 如果所删除的内容不在列表中，就报错。注意阅读报错信息：`x not in list`

对于删除，能不能更友好一些？在删除之前，先判断一下这个元素是不是在列表中，如果在就删，不在就不删。

如果读者想到这里，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
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——这也正是`Python`语言的特点之一。

对于 `remove()`，还有最后一个要交代的，它对列表的修改也是原地修改，正确实现删除后没有返回值。

另外一个删除 `list.pop([i])` 会怎么样呢？看看文档，做做实验。

列表(3)

```
>>> all_users = ['qiwsir', 'github', 'io', 'algorithm']
>>> all_users.pop()
'algorithm'
>>> all_users
['qiwsir', 'github', 'io']
```

`list.pop([i])`，圆括号里面是 `[i]`，表示这个参数是可选的，如果不写，也就是圆括号为空，默认删除最后一个，并且将删除的元素作为结果返回。提醒读者注意，它有返回值。

如果参数不为空，可以删除指定索引的元素，并将该元素作为返回值。

```
>>> 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      #删除索引起超出范围，就是1不在list的编号范围内
```

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，且对列表原地修改，无返回值
- `list.pop([i])` 中的 `i` 是列表中元素的索引值，可选。为空则删除列表最后一个，否则删除索引为 `i` 的元素。并且将删除元素作为返回值。

给读者留下一个思考题，能不能事先判断一下要删除的元素的索引是不是在列表的长度范围(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。

跟本函数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。

列表(3)

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。输入`help(list.sort)`调用帮助文档，可以得到如下内容（Python2和Python3略有差异）。

```
>>> #Python 2
>>> help(list.sort)
Help on method_descriptor:

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

>>> #Python 3
>>> help(list.sort)
Help on method_descriptor:

sort(...)
    L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*
```

虽然文档说明中略有差异（读者通过本书以前的内容，应该能够理解差异的缘由），但是不影响操作。

```
>>> 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？不知道读者是否用过电子表格，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']
```

列表(3)

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()`，你可以去探究一下用法。

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。

最后，对前文提到的“保留字”基于补充说明。

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。

在Python 2和Python 3中，都可以用下面的方式查看保留字，而且，注意，两个不同版本的保留字还有差别，虽然差别很小。

Python 2中：

```
>>> import keyword
>>> keyword.kwlist
['and', 'as', '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']
>>> len(keyword.kwlist)
31
```

Python 3中：

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
>>> len(keyword.kwlist)
33
```

列表的方法已经结束，但是列表的话题还没有完结，因为它还能和“字符串”搞在一起，需要辨析一番。

[总目录](#) | [上节：列表\(2\)](#) | [下节：列表和字符串](#)

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

Then Peter came to him and said,"Lord, how many times must I forgive my brother who sins against me? As many as seven times?" Jesus said to him,"Not seven times, I tell you, but seventy-seven times?" (MATTHEW 18:21-22)

回顾列表和字符串

列表和字符串两种类型的对象，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的知识复习一下，所谓“温故而知新”。

相同点

都是序列

不管是组成列表的元素，还是组成字符串的字符，都可以从左向右，依次用 `0, 1, 2, ...` 这样的方式建立索引。而要得到一个或多个元素，可以使用切片。

关于序列的基本操作，对两者都适用。

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
```

区别

列表和字符串的最大区别是：列表是可以改变的，字符串是不可变。这个怎么理解呢？

首先看对列表的这些操作，其根源在于列表可以进行修改，即列表是可变的。

回顾列表和字符串

```
>>> 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，如果用在字符串上，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
'WEElcome 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。

在字符串里面的每个元素只能是字符；在列表中，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]]
```

这个列表的元素，是另外三个列表。这样的列表，称之为多维列表。如果读者学习过行列式，这就比较容易理解了。

```
>>> matrix[0][1]  
2
```

当然，列表也可以是这样的：

```
>>> 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'
```

在多维的情况下，里面的list被当成一个元素对待。

列表和字符串转化

符合某些条件的情况下，可以实现列表和字符串之间的转化。会使用到 `split()` 和 `join()`，对这两个函数，已经不陌生了，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)

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，承接前面的操作：

```
>>> 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'
```

回顾列表和字符串

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()
['I', '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'
```

读者是否感到新奇，对于join()函数，其格式是"sep".join(list)，不是list.join(sep)。其实，join()是字符串的方法，不是列表的方法。

```
>>> help(str.join)
Help on method_descriptor:

join(...)
    S.join(iterable) -> str

    Return a string which is the concatenation of the strings in the
    iterable. The separator between elements is S.
```

不过，能传入join()的对象，或者说参数的值，也是有条件的。下面的就不行。

```
>>> a = [1,2,3,'a','b','c']
>>> "+".join(a)
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    "+".join(a)
TypeError: sequence item 0: expected str instance, int found
```

“列表是苦力”，但是暂且让它干这么多。因为更多类型的对象依次登场，先让它到后台。

[总目录](#) | [上节：列表\(3\)](#) | [下节：元组](#)

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

“谁愿为首，就必作你们的仆人。正如人子来，不是要受人的服事，乃是要服事人，并且要舍命，作多人的赎价。”(MATTHEW 20:27-28)

元组

元组是Python中的一种对象类型。它与之前的列表、字符串、整数、浮点数等并列。

但，因为它跟列表接近，经常被忽略。

定义

先看一个例子：

```
>>> s = "abc"  
>>> s  
'abc'
```

这是一个简单的赋值，还可以这样写，这就是Python的与众不同之处。

```
>>> t = 123, 'abc', ["come", "here"]  
>>> t  
(123, 'abc', ['come', 'here'])
```

不仅没有报错，也没有“最后一个有效”，而是将对象放到了一个圆括号里面。

这个带有圆括号的对象，就是一种新的对象（或数据）类型：tuple（元组）。

```
>>> type(t)  
<class 'tuple'>          #这是Python 3的结果，在Python 2中显示：<type 'tuple'>
```

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对象（数据）。

这种类型，可以歪着想，所谓“元”组，就是用“圆”括号啦。

仅从前面那个例子，显而易见得出，元组是序列，这点上跟列表和字符串类似。

但元组中的元素不能更改，这点上跟列表不同，倒是跟str类似；它的元素又可以是任何类型的数据，这点上跟列表相同，但不同于字符串。

元组

```
>>> 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，元组就是一个融合了部分列表和部分字符串属性的杂交产物。此言有理。

索引和切片

元组是序列，因此，元组的基本操作就和列表和字符串相仿。

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元组上的表现，就不一一展示了。读者可以去试试。

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'>
```

元组

以上的例子说明，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。

所有在列表中可以修改列表的方法，在元组中，都失效。因为元组不可修改。

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'))
```

元组用在哪里？

既然它是列表和字符串的杂合，它有什么用途呢？不是用列表和字符串都可以了吗？

在很多时候，的确是用列表和字符串都可以了。但是，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，元组有这类特点，并且是它使用的情景：

- 元组比列表操作速度快。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，请使用元组代替列表。
- 如果对不需要修改的数据进行“写保护”，可以使代码更安全。使用元组而不是列表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，则需要执行元组到列表的转换（需要使用一个特殊的函数）。
- 元组可以在字典（又一种对象类型，后面要讲述）中被用做 `key`，但是列表不行。字典的 `key` 必须是不可变的。元组本身是不可改变的，列表是可变的。
- 元组可以用在字符串格式化中。

元组很简单，不用过多的篇幅说明，但是，依然建议读者把它作为序列，依次按照序列的操作对元组进行实践。

[总目录](#) | [上节：回顾list和str](#) | [下节：字典\(1\)](#)

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

Now I appeal to you, brothers and sisters, by the name of our Lord Jesus Christ, that all of you be in agreement and that there be no divisions among you, but that you be united in the same mind and the same purpose. (1 CORINTHIANS 1:10)

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。

我曾经上过小学，这是事实，那时候曾经用过一本小小的《新华字典》，与我差不多年龄的朋友，也都有同样的记忆，没记错的话应该是1.01元人民币一本。

《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。而是提醒读者想想曾经如何使用字典：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**的对象（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。

字典(1)

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

怎么会这样？！读者能不能给出解答？

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)。

创建字典

创建字典，有多种方法，依次尝试。

方法1：

创建一个空的字典，然后可以向里面加东西。

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

不要小看“空”，“色即是空，空即是色”，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。

还可以创建不空的字典。

```
>>> 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)。

在一个字典中，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(:)英文的冒号，每一对键值之间用英文的逗号(,)隔开。

向已经建立的字典中，增加键值对的一种方法是这样的：

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

字典(1)

用这样的方法可以向一个字典类型的对象中增加“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'
```

访问字典的值

字典对象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'
```

通过“键”能够读取到相应的“值”。在前面的操作中，也显示了，通过“键”能够增加字典中的“值”。

还有，通过“键”能够改变字典中的“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
```

既然字典是键值对的映射，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。

在列表中，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。

字典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，在字典中是通过“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

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。这进一步说明字典的值是可变的。

```
>>> 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
```

字典(1)

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。

```
>>> 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。然后在字典中写好一些数据，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multimap）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。

以上对字典有了基本了解，后面要深入对字典的认识。

[总目录](#) | [上节：元组](#) | [下节：字典\(2\)](#)

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

字典(2)

耶稣对他说：“你要尽心、尽性、尽意，爱主你的神。这是诫命中的第一，且是最大的。其次也相仿，就是要爱人如己。这两条诫命是律法和先知一切道理的总纲。”(MATTHEW 22:37-39)

字典(2)

字典方法

跟前面所讲述的其它对象类似，字典也有一些方法。通过这些方法，能够实现对字典的操作。

这回可不是屠龙之技了，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（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()`，专门查看对象在内存中的位置。

```
>>> 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中。其它的就不演示了，就仅看看字典类型。

字典(2)

```
>>> 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”。

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。

现在有两个字典对象，虽然它们是一样的，但在两个“窝”里面，彼此互不相干。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。

字典(2)

```
>>> 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所对应的字典对象中，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()

字典(2)

```
>>> 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，如果是基本类型的对象（专指数字和字符串），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'}
```

字典(2)

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是。

```
>>> 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”。

字典(2)

```
>>> 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。

`clear()` 后的字典，是将其内容清空，还能够使用 `a = {}` 这种方法。

最后提醒，`clear()` 没有返回值。

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：

字典(2)

```
>>> 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` 就能看到了。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字典的函数，并且这三组有相似的地方。

注意，在Python 3 中，因为已经做了优化，所以不需要有 `iteritems`，`iterkeys` 和 `itervalues` 三个方法。

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。

Python 2中：

```
>>> help(dict.items)

items(...)
D.items() -> list of D's (key, value) pairs, as 2-tuples
```

Python 3中：

```
>>> help(dict.items)
Help on method_descriptor:

items(...)
D.items() -> a set-like object providing a view on D's items
```

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。

Python 2中，`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
```

字典(2)

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。

而Python 3中，`D.items()` 返回的就是一个可迭代的对象，就无需`D.iteritems()`了。

在Python 2中操作。

```
>>> 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。

在python3中，只有`items`, `keys`和`values`方法，返回的也不是`list`, 而是[view对象](#)

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。

字典(2)

```
>>> 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'
```

pop() 的参数，可以是两个，上面的例子中只写了一个。如果写两个，那么就先检查k是不是存在于字典中的键，如果是，就返回它所对应的值，如果不是，就返回参数中的第二个，当然，如果不写第二个参数，就会如同上面举例一样报错。

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'}
```

字典(2)

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，返回的数据格式是元组。

```
>>> 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对象。

字典(2)

```
>>> 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'}
```

列表中以元组为元素，每个元组是一个键值对。

has_key

这个函数的功能是判断字典中是否存在某个键，它目前仅存在于Python 2，在Python 3中取消了这个函数。

```
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
```

在Python 3中，类似判断使用 `k in D`，甚至在Python 2中，这也是很好的方法。

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。

Stay awake and pray that you may not come into the time of trial; the spirit indeed is willing, but the flesh is weak.(MATTHEW 26:41)

集合(1)

已经学习了几种对象类型。

温故而知新。它们是：int/float/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。

读者可能有疑问了，出了这么多的类型，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。

所以，再多的数据类型及其各种方法，都不需要记忆。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)

还有，如果你经常练习，会发现很多东西自然而然就记住了。

在已经学过的不同种类型的对象中：

- 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。

创建集合

集合(1)

元组算是列表和字符串的某些特征的杂合(杂交的都有自己的优势,上一节的末后已经显示了),那么集合则可以堪称是列表和字典的某些特征杂合.

请读者细细品味,这种杂合的特征。

首先要创建集合,其方法是:

```
>>> s1 = set("qiwsir")
>>> s1
set(['q', 'i', 's', 'r', 'w'])
```

把字符串中的字符拆解开,形成集合。

特别注意观察, `qiwsir` 中有两个 `i`,但是在集合中,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,就会过滤掉,剩下不重复的。

在使用 `dir()` 来看看集合的方法,特别从下面找一找有没有 `index`,如果有它,就说明可以索引,否则,集合就没有索引。

```
>>> dir(set)
['__and__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__iand__', '__init__',
 '__ior__', '__isub__', '__iter__', '__ixor__', '__le__', '__len__', '__lt__',
 '__ne__', '__new__', '__or__', '__rand__', '__reduce__', '__reduce_ex__', '__repr__',
 '__ror__', '__rsub__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
 '__subclasshook__', '__xor__', 'add', 'clear', 'copy', 'difference', 'difference_update',
 'discard', 'intersection', 'intersection_update', 'isdisjoint', 'issubset', 'issuperset',
 'pop', 'remove', 'symmetric_difference', 'symmetric_difference_update', 'union',
 'update']
```

请用你那双慧眼,寻找一番。

没有 `index`。

的确没有。所以,集合没有索引,也就没有顺序而言,它不属序列。当你这样操作的时候,

```
>>> s1 = set(['q', 'i', 's', 'r', 'w'])
>>> s1[1]
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    s1[1]
TypeError: 'set' object does not support indexing
```

集合(1)

报错。并且明确告知我们，不支持索引。

除了用 `set()` 来创建集合。还可以使用 `{}` 的方式。

```
>>> s3 = {"facebook", 123}      #通过{}直接创建
>>> s3
set([123, 'facebook'])
```

但是这种方式不提倡使用。因为我们已经将 `{}` 常常用在字典上了，要避免歧义才好。

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'
```

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`”，翻译为“可哈希”。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，类似字符串那样不能原地修改，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列表、字典是不可哈希类型，言外之意，里面的元素都应该是可哈希类型。

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

从前面的 `dir(set)` 结果中，你可以看到不少集合的方法。

为了看的清楚，我把双划线 `_` 开始的先删除掉，剩下的就是：

```
'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。读者完全可以用这种方法自己查看了。

集合(1)

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.
```

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,在字典和集合中都用.但是,如上面的方法建立的是字典,不是集合.
这是python规定的.

要建立空集合,不得不使用 set() 。

```
>>> s = set()
>>> type(s)
<class 'set'>      #Python 2的返回结果略有差异,为<type 'set'>
```

当然,非空集合,依然可以这样:

```
>>> a_set = {'a', 'i'}      #这回就是set了吧
>>> type(a_set)
<type 'set'>          #Python 3返回: <class 'set'>
```

然后就开始对这个集合使用 add() 方法,并看效果。

```
>>> a_set.add("qiwsir")    #增加一个元素
```

没有报错,就意味着成功。没有返回值,根据我们经验,这属于“原地修改”。

```
>>> a_set
set(['i', 'a', 'qiwsir'])
```

集合(1)

这次经验胜利了。继续洋洋得意地敲代码。

```
>>> 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'])
```

成功继续伴随着。废话！仅仅是刚才的重复罢了。重复是必须的，这样是为了加深印象。

```
>>> b_set.add([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

报错。哪里错了？

遇见错误，不要沮丧。认真阅读报错信息：列表是不可哈希的。洋洋得意中忘记前面强调的：“集合中的元素应该是**hashable**类型”。

耍一个小聪明吧。

```
>>> b_set.add('[1,2,3]')
>>> b_set
set(['[1,2,3]', 'h', 'o', 'n', 'p', 't', 'qiwsir', 'y'])
```

为什么这么一搞就可以了呢？仔细观察，这回不是增加列表了，本质是字符串。

除了上面的增加元素方法之外，还能够从另外一个集合中合并过来元素，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'])
```

如果仅仅是这样的操作，容易误以为 `update` 方法的参数只能是集合。非也。看文档中的描述，这个方法的作用是用原有的集合自身和其它的什么东西构成的新集合更新原来的集合。这句话有点长，可以多读一遍。分解开来，可以理解为：`others`是指的作为参数的不可变对

集合(1)

象，将它和原来的集合组成新的集合，用这个新集合替代原来的集合。举例：

```
>>> s2.update("goo")
>>> s2
set(['github', 'o', 'g', 'qiwsir'])
>>> s2.update((2,3))
>>> s2
set([2, 3, 'g', 'o', 'github', 'qiwsir'])
```

所以，文档的寓意还是比较深刻的。

pop, remove, discard, clear

```
>>> help(set.pop)
pop(...)
    Remove and return an arbitrary set element.
    Raises KeyError if the set is empty.
```

一下变量承接前面的操作，

```
>>> 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'])
```

能不能指定删除某个元素？

```
>>> b_set.pop("n")
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，如果集合已经是空的了，再删除，也报错。这条是帮助文档中告诉我们的，读者可以试试。

要删除指定的元素，怎么办？

集合(1)

```
>>> 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 如果是集合中的元素, 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. (自己在交互模式下 help(set.clear))

集合(1)

```
>>> 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](#) | [上节：字典\(2\)](#) | [下节：集合\(2\)](#)

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

集合(2)

Then Jesus came up and said to them, "All authority in heaven and on earth has been given to me. Therefore go and make disciples of all nations, baptizing them in the name of the Father and the Son and the Holy Spirit, teaching them to obey everything I have commanded you. And remember, I am with you always, to the end of the age."
(MATTHEW 28:18-20)

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")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

报错。“不求成功，但求报错”。从提示信息中可知，这种集合不能修改。

对比看一看，也是复习。这是一个可以原处修改的集合。

```
>>> a_set = set("github")
>>> 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

只有一种关系。

集合(2)

元素要么属于某个集合，要么不属于。

```
>>> aset = set(['h', 'o', 'n', 'p', 't', 'y'])
>>> "a" in aset
False
>>> "h" in aset
True
```

集合与集合的关系

假设两个集合A、B

- A是否等于B，即两个集合的元素是否完全一样

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
```

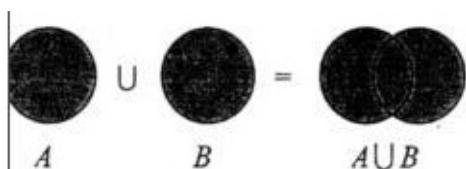
- A是否是B的子集，或者反过来，B是否是A的超集。即A的元素也都是B的元素，但是B的元素比A的元素数量多。

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

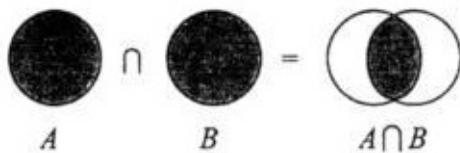


集合(2)

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'])      #Python 3: {'q', 'w', 'r', 'l', 's
', 'a', 'o', 'i'}
>>> a.union(b)
set(['a', 'i', 'l', 'o', 'q', 's', 'r', 'w'])      #Python 3同上
```

- A 、 B 的交集，即 A 、 B 所公有的元素，如下图所示

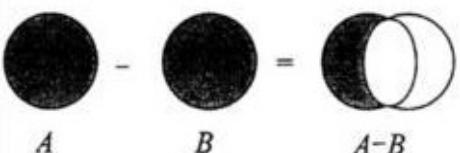


```
>>> a = set(['q', 'i', 's', 'r', 'w'])
>>> b = set(['a', 'q', 'i', 'l', 'o'])
>>> a & b      #两种方式，等价
set(['q', 'i'])      #Python 3: {'q', 'i'}
>>> a.intersection(b)
set(['q', 'i'])      #Python 3同上
```

我在实验的时候，顺手敲了下面的代码，出现的结果如下，能解释一下吗？（思考题）

```
>>> a and b
set(['a', 'q', 'i', 'l', 'o'])      #Python 3: {'q', 'l', 'a', 'o', 'i'}
```

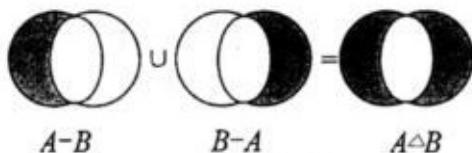
- A 相对 B 的差（补），即 A 相对 B 不同的部分元素，如下图所示



```
>>> a = set(['q', 'i', 's', 'r', 'w'])
>>> b = set(['a', 'q', 'i', 'l', 'o'])
>>> a - b
set(['s', 'r', 'w'])      #Python 3: {'r', 's', 'w'}
>>> a.difference(b)
set(['s', 'r', 'w'])      #Python 3同上
```

- A 、 B 的对称差集，如下图所示

集合(2)



```
>>> 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'])      #Python 3: {'w', 'r', 'l', 's', 'a', 'o'}
```

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](#) | [上节：集合\(1\)](#) | [下节：运算符](#)

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

Jesus answered them, "Those who are well don't need a physician, but those who are sick do. I have not come to call the righteous, but sinners to repentance."(LUKE 5:31-32)

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和其优先级的表格，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，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

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

运算符

所谓比较，这在某国是最常见的了，做家长的经常把自己的孩子跟别人的孩子比较，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对象的都可以比较，比如两个数字、两个字符串等。注意，是两个同一类型的。

不同类型的量可以比较吗？

须明确，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 `a=10` , `b=20` :

运算符	描述	实例
<code>==</code>	等于 - 比较对象是否相等	<code>(a == b)</code> 返回 <code>False</code> 。
<code>!=</code>	不等于 - 比较两个对象是否不相等	<code>(a != b)</code> 返回 <code>True</code> .
<code>></code>	大于 - 返回x是否大于y	<code>(a > b)</code> 返回 <code>False</code> 。
<code><</code>	小于 - 返回x是否小于y	<code>(a < b)</code> 返回 <code>True</code> 。
<code>>=</code>	大于等于 - 返回x是否大于等于y。	<code>(a >= b)</code> 返回 <code>False</code> 。
<code><=</code>	小于等于 - 返回x是否小于等于y。	<code>(a <= b)</code> 返回 <code>True</code> 。

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
```

运算符

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，不是前面说的字典类型的对象。

```
>>> 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

在所有的高级语言中，都有这么一类对象，被称之为布尔型。从这个名称就知道了，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。

请读者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True、False的数据。

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对象间的运算。

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 a=10 , b=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 20。
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 10。
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。

运算符

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。短路求值，能够减少计算量，提高计算布尔表达式的速度。

把上面的计算过程综合一下，在计算 A and B 中，其流程是：

```
if A == False:  
    return False  
else:  
    return bool(B)
```

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。

- or

运算符

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:  
    return bool(B)
```

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  
    return bool(B)   #看B的值，然后就返回B的值做为最终结果。
```

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，前面已经用过了。

复杂的布尔表达式

在进行逻辑判断或者条件判断的时候，不一定都是类似上面例子那样简单的表达式，也可能遇到复杂的表达式。如果你遇到了复杂表达式，最好的方法是使用括号。

```
>>> (4<9) and (5>9)  
False  
>>> not(True and True)  
False
```

运算符

用括号的方法，意义非常明确。当然，布尔运算也有优先级，但是，你不一定记住，或者如果使用括号，根本没有必要记住优先级。

不过为了给学习者一个印象，先面还是以表格行事，按照从高到底的顺序，把布尔运算的优先级列出来，仅供参考，并且没有必要记忆。

顺序	符号
1	$x == y$
2	$x != y$
3	$\text{not } x$
4	$x \text{ and } y$
5	$x \text{ or } y$

最后强调：一定要用括号，不必记忆表格内容。

有了运算符，再应用已经学过的知识，就可以编程了。编程，从语句开始。

[总目录](#) | [上节：集合\(2\)](#) | [下节：语句\(1\)](#)

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)

你们不要论断人，就不被论断；你们不要定人的罪，就不被定罪；你们要饶恕人，就必蒙饶恕。你们要给人，就必有给你们的，并且用十足的升斗，连摇带按、上尖下流地倒在你们怀里；因为你们用甚么量器量给人，也必用甚么量器量给你们。(LUKE 6:37-38)

语句(1)

写程序，就好比小学生学习写作一样，先学习词语，然后造句，在写文章。

到目前为止仅仅学会了一些词语（各种类型的对象），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。（注意：Python 3里，`print`语句改成了`print()`函数。）

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中的语句还是有其特别之处的（别的语言中，也会有自己的特色）。下面就开始娓娓道来。

print

语句(1)

在Python 2中，print是一个语句，但是在Python 3中它是一个函数了。这点请注意。

以Python 2为例，说明print语句。如果说读者使用的是Python 3，请自行将 `print xxx` 修改为 `print(xxx)`，其它不变。

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，如果要在一个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。

或许现在体现得还不是很明显，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语句输出。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。

但是，在Python 3中情况有变。

语句(1)

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
        file: a file-like object (stream); defaults to the current sys.stdout.
        sep:   string inserted between values, default a space.
        end:   string appended after the last value, default a newline.
        flush: whether to forcibly flush the stream.
```

从帮助文档中可以看出，默认的 `end='\n'`，如果不打算换行，可以在使用 `print()` 函数的时候，修改 `end` 这个参数的值。

```
>>> for i in [1,2,3,4]:
    print(i, end=',')

1,2,3,4,
```

`print` 语句或者 `print()` 函数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 `import math`，`math` 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
```

语句(1)

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` 是那个欧拉数；`pi` 就是 π 。

`e`，作为數學常數，是自然對數函數的底數。有時稱它為歐拉數（Euler's number），以瑞士數學家歐拉命名；也有個較鮮見的名字納皮爾常數，以紀念蘇格蘭數學家約翰·納皮爾引進對數。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》）

例子：

```
>>> 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

语句(1)

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（整数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（整数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整数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。

语句(1)

```
>>> 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           #Python 3: 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：

语句(1)

```
>>> 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”。+、-、*、/、%都可以实现类似这种操作。

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'
```

本节只是语句的入门，后面还有更多精彩。

[总目录](#) | [上节：运算符](#) | [下节：语句\(2\)](#)

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

语句(2)

众百姓听的时候，耶稣对门徒说，“你们要防备文士，他们好穿长衣游行，喜爱人在街市上问他们安，又喜爱会堂里的高位，筵席上的首座。他们侵吞寡妇的家产，假意作很长的祷告。这些人要受更重的刑罚。”(LUKE 20:45-47)

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翻译为中文是“如果”。它所发起的就是一个条件语句。

换言之，if是构成条件语句的关键词。

最简单的方式，就是：

```
if bool(conj):
    do something
```

一个非常简单的例子：

```
>>> a = 8
>>> if a == 8:
...     print a          #Python 3: print(a)
...
8
```

简单解释。

`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` (`Python 3: print(a)`) 要有四个空格的缩进。这是Python的特点，称之为语句块。

重要的话说几遍都不为过，“要缩进四个空格”。

唯恐说的不严谨，我还是引用维基百科中的叙述：

Python開發者有意讓違反了縮排規則的程序不能通過編譯，以此來強迫程序员養成良好的編程習慣。並且Python語言利用縮排表示語句塊的開始和結束（Off-side規則），而非使用花括號或者某種關鍵字。增加縮排表示語句塊的開始，而減少縮排則表示語句塊的結束。縮排成爲了語法的一部分。例如if語句。

根據PEP的規定，必須使用4個空格來表示每級縮排。使用Tab字符和其它數目的空格雖然都可以編譯通過，但不符合編碼規範。支持Tab字符和其它数目的空格僅僅是爲兼容很舊的Python程式和某些有問題的編輯程式。

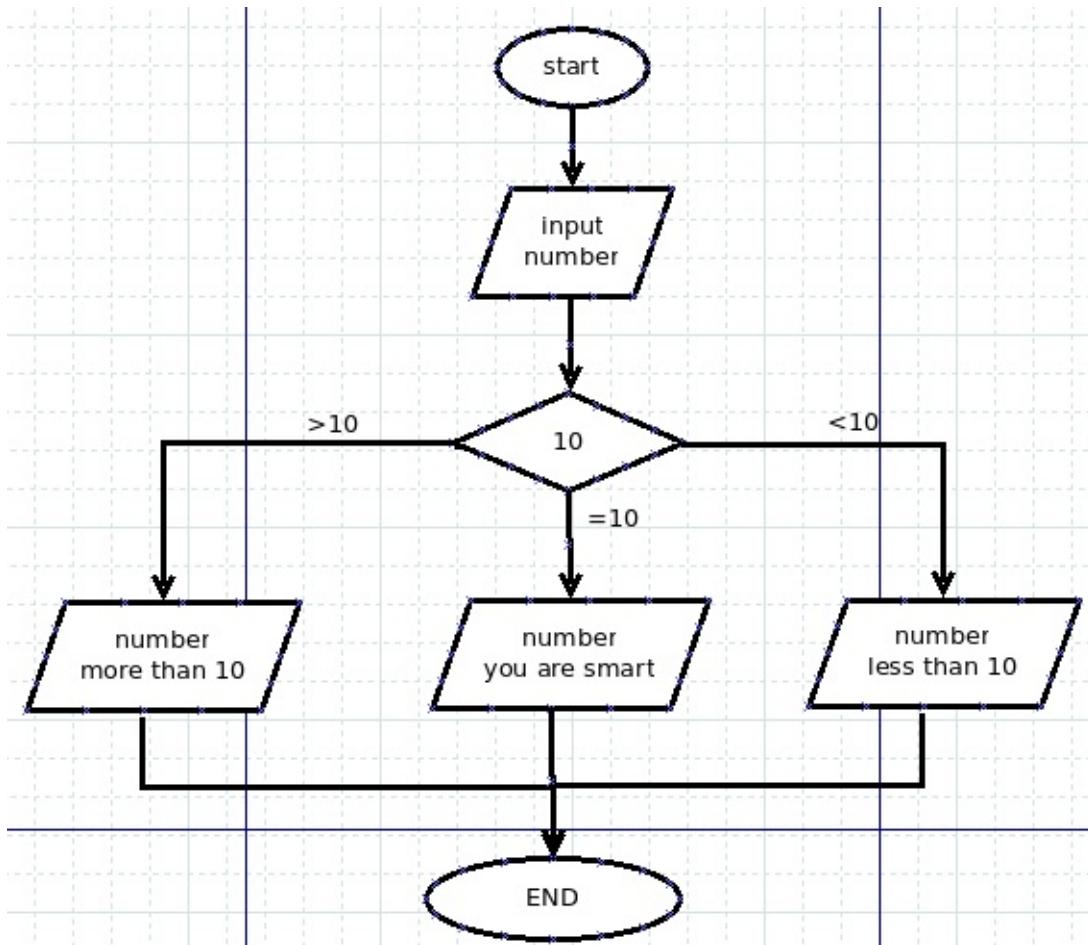
语句(2)

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 ... elif ... else

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，判断该数字和10的大小关系。如果大于10，则输出大于10的提示；如果小于10，则输出小于10的提示；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：

语句(2)

```
if 条件1:  
    语句块1  
elif 条件2:  
    语句块2  
elif 条件3:  
    语句块3  
...  
else:  
    语句块4
```

`elif` 和 `else` 发起的部分，都可以省了，那时就回归到了只有一个 `if` 的情况了。

为了应付多条件判断，就不能省略。

下面我们就不再交互模式中写代码了。打开你选择的写Python代码的编辑器，它一定得是你认为最好的武器。

Python 2的代码如下：

```
#!/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：{}".format(number)  
    print "This number is more than 10."  
elif number < 10:  
    print "您输入的数字是：{}".format(number)  
    print "This number is less than 10."  
else:  
    print "Are you a human?"
```

Python 3的代码如下：

```

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

print("请输入任意一个整数数字：")
number = int(input())

if number == 10:
    print("您输入的数字是:{}.".format(number))
    print("You are SMART.")
elif number > 10:
    print("您输入的数字是:{}.".format(number))
    print("This number is more than 10.")
elif number < 10:
    print("您输入的数字是:{}.".format(number))
    print("This number is less than 10.")
else:
    print("Are you a human?")

```

对上述程序，需要说明几点。

- `#!/usr/bin/env python`。因为我在Ubuntu操作系统中调试的程序，所以这个是必须的。如果你是在windows里面，可以省略。
- `#coding:utf-8`。在程序中有中文，有英文，即便是没有中文，也要声明程序的编码格式是utf-8。
- `raw_input()` 或者 `input()` 得到的是字符串，因为在程序中，要将输入的数字跟整数10进行比较，所以要将该字符串转化为整数，因此使用了 `int()` 函数。

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，在书写在条件 `number == 10` 时，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是整数类型，`number`也是。

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。

语句(2)

```
$ 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.
```

最后，补充说明 `#! /usr/bin/env python` 的含义。

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（如Ubuntu类型的操作系统）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。

上述补充解释不适用于windows系统。

现在读者是否已经明晰，所谓条件语句中的“条件”中，就是各种条件运算表达式或者布尔值，如果是 `True`，就执行该条件下的语句块。

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"
```

先思考一个问题，`if "laoqi"`，这里似乎没有条件表达式，Python怎么判断？

的确没有条件表达式，事实上，它就等同于 `if bool("laoqi")`。

```
>>> name  
'qiwsir'  
>>> name = 'qiwsir' if "" else "python"
```

而这里的 `if ""` 就相当于 `if bool("")`（注意写法上，两个引号之间没有空格）。还要多说一句，这里列举的方式，纯粹是为了理解三元操作符，不具有实战价值。

语句(2)

```
>>> name  
'python'  
>>> name = "qiwsir" if "github" else ""  
>>> name  
'qiwsir'
```

总结一下： $A = Y \text{ if } X \text{ else } Z$

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'
```

`if` 所引起的条件语句使用非常普遍，当然也比较简单。

[总目录](#) | [上节：语句\(1\)](#) | [下节：语句\(3\)](#)

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)

"But be on your guard so that your hearts are not weighted down with dissipation and drunkenness and the worries of this life, and that day close down upon you suddenly like a trap. For it will overtake all who live on the face of the whole earth. But stay alert at all times, praying that you may have strength to escape all these things that must happen, and to stand before the Son of Man."

你们要谨慎，恐怕因贪食醉酒并今生的思虑，累住你们的心，那日子就如同网罗忽然临到你们。因为那日子要这样临到全地上一切居住的人。你们要时时儆醒，常常祈求，是你们能逃避这一切要来的事，得以站立在人子面前。(LUKE 21:34-36)

语句(3)

循环，你我都在其中。

日月更迭，斗转星移，无不是循环；王朝更迭，子子孙孙，从某个角度看也都是循环。

循环是现实生活中常见的现象。

编程语言就是要解决现实问题的，因此也少不了要循环。

for循环

在高级编程语言中，大多数都有**for循环**（**for loop**），关于这种循环，我借用[维基百科中“for loop”的说明](#)，帮助大家理解。

In computer science a for-loop (or simply for loop) is a programming language control statement for specifying iteration, which allows code to be executed repeatedly. The syntax of a for-loop is based on the heritage of the language and the prior programming languages it borrowed from, so programming languages that are descendants of or offshoots of a language that originally provided an iterator will often use the same keyword to name an iterator, e.g., descendants of ALGOL use "for", while descendants of Fortran use "do." There are other possibilities, for example COBOL which uses "PERFORM VARYING".

如果看上面的英文有点难度，可以看下面的翻译，当然是很烂的翻译，绝对达不到“信雅达”的境界，虽然一直追求“德艺双馨”。

在计算机科学中**for循环**是编程语言中针对可迭代对象的语句，它允许代码被重复执行。**for循环**的语法是在对历史上的编程语言继承和借鉴基础上形成的，该语言原来有迭代器，则后来的编程语言也用同样的关键词来实现迭代，比如ALGOL系的使用“**for**”，而Fortan系的使用“**do**”，也有例外，如COBOL用“**PERFORM VARYING**”。

语句(3)

for循环是从ALGOL那里继承过来的。ALGOL（ALGOrithmic Language）是最早的高级编程语言（几乎没有之一），后来的不少高级语言都继承了ALGOL的某些特性，比如Pascal、Ada、C语言等，也包括Python。

维基百科上对此有非常清晰的说明。

The name for-loop comes from the English word for, which is used as the keyword in most programming languages to introduce a for-loop. The term in English dates to ALGOL 58 and was popularized in the influential later ALGOL 60; it is the direct translation of the earlier German für, used in Superplan (1949–1951) by Heinz Rutishauser, who also was involved in defining ALGOL 58 and ALGOL 60. The loop body is executed "for" the given values of the loop variable, though this is more explicit in the ALGOL version of the statement, in which a list of possible values and/or increments can be specified.

在ALGOL中，循环的关键词就是用for。所以，Python要继承这个光荣传统，也用for来实现循环。

于是有了：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；语句块都要四个空格缩进。

从例子中理解**for**循环

下面这个例子似曾相识。

```
>>> hello = "world"  
>>> for i in hello:  
...     print i          #Python 3: print(i)  
...  
w  
o  
r  
l  
d
```

这个例子中实现的就是for循环。下面庖丁解牛：

1. `hello = "world"`，赋值语句，实现变量 `hello` 和字符串 `"world"` 之间的引用关系。
2. `for i in hello:`，`for` 是发起循环的关键词；`i in hello` 是循环规则，字符串类型的对象是序列类型，能够从左到右一个一个地按照索引读出每个字符，于是变量 `i` 就按照

语句(3)

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` 或者 `print(i)`，打印出了字母w；然后循环第二次，让 `i="e"`，执行 `print i` 或者 `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)》中相关内容。

因为可以也通过使用索引，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]          #Python 3: 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。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` 依次得到列表中的各个值。当 `i=0` 时，打印 `hello[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，显示了对字符串的字符依次获取，也涉及了列表，感觉不过瘾呀。

那好，看下面对列表的循环：

```
>>> ls_line = ['Hello', 'I am qiwsi', 'Welcome you', '']

>>> for word in ls_line:
...     print word          #Python 3: print(word)
...
Hello
I am qiwsi
Welcome you

>>> for i in range(len(ls_line)):
...     print ls_line[i]      #Python 3: print(ls_line[i])
...
Hello
I am qiwsi
Welcome you
```

以上两个例子中，是将`for`循环用于字符串和列表，此外，还可以用于字典、元组。例如：

语句(3)

```
>>> d = dict([("website", "www.itdiffer.com"), ("lang", "python"), ("author", "laoqi")]
[])

>>> d
{'website': 'www.itdiffer.com', 'lang': 'python', 'author': 'laoqi'}
>>> for k in d:
    print k      #Python 3: print(k)
```

输出结果是：

```
website
lang
author
```

注意到，上面的循环，其实是读取了字典的key。在字典中，有一个方法，`dict.keys()`，得到的是字典key列表。

```
>>> for k in d.keys():
    print k      #Python 3: print(k)

website
lang
author
```

这种循环方法和上面的循环方法，结果是一样的，但是，这种方法并不提倡，以为它在执行速度上表现欠佳。

如果要获得字典的value怎么办？不要忘记`dict.values()`方法。读者可以自行测试一番。

除了可以单独获得key或者value的循环之外，还可以这么做：

```
>>> for k, v in d.items():
    print k + "-->" + v          #Python 3: print(k+"-->" + v)

website-->www.itdiffer.com
lang-->python
author-->laoqi
```

在工程实践中，你一定会遇到非常大的字典。用上面的方法，要把所有的内容都读入内存，内存东西多了，可能会出麻烦。为此，Python中提供了另外的方法。

```
>>> for k,v in d.iteritems():    #注意：仅在Python2中可用，Python 3中已经做了优化，d.item()即
    print k + "-->" + v

website-->www.itdiffer.com
lang-->python
author-->laoqi
```

语句(3)

这里是循环一个迭代器，迭代器在循环中有很多优势。除了刚才的 `dict.iteritems()` 之外，还有 `dict.itervalues()`，`dict.iterkeys()` 供你选用（以上三个 `dict.iter*` 都只用在Python 2中，Python 3中已经不需要了）。

```
>>> d.iteritems()      #注意：仅在Python2中可用。  
<dictionary-itemiterator object at 0x0000000000322E368>
```

至于对元组的循环，读者自行尝试即可。

除了上述的对象之外，`for`循环还能应用到哪些对象上？能不能用在数字上呢？

```
>>> for i in 321:  
    print i  
  
Traceback (most recent call last):  
  File "<pyshell#48>", line 1, in <module>  
    for i in 321:  
TypeError: 'int' object is not iterable
```

报错了。这说明对于数字不能使用`for`循环。不过，光知道这个还不行，还要看看报错信息。

报错信息中告诉我们，`int` 对象不是可迭代的。言外之意是什么？那就是`for`循环所应用的对象，应该是可迭代的。

那么，怎么判断一个对象是不是可迭代的呢？

```
>>> import collections
```

引入`collections`这个标准库。要判断数字321是不是可迭代的，可以这么做：

```
>>> isinstance(321, collections.Iterable)  
False
```

返回了`False`，说明321这个整数类型的对象，是不可迭代的。再判断一个列表对象。

```
>>> isinstance([1, 2, 3], collections.Iterable)  
True
```

从返回结果，我们知道，列表 `[1, 2, 3]` 是可迭代的。

当然，并不是要你在使用`for`循环之前，非要判断某个对象是否可迭代。因为至此，你已经知道了字符串`str`、列表`list`、字典`dict`、元组`tuple`、集合`set`都是可迭代的。

range(start,stop[, step])

语句(3)

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，返回列表的最后的值不包含 `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，默認為 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，默認是 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：

Python 2:

语句(3)

```
>>> 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]
```

Python 3:

```
>>> range(9)
range(0, 9)
>>> list(range(9))
[0, 1, 2, 3, 4, 5, 6, 7, 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`, 列表中的第一个值是 `start=0`, 第二个值是 `start+1*step=2` (注意, 这里是1, 不是2, 不要忘记, 前面已经讲过, 不论是列表还是字符串, 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:

Python 2 :

语句(3)

```
>>> range(-9)
[]
>>> range(0, -9)
[]
>>> range(0)
[]
```

Python 3:

```
>>> range(-9)
range(0, -9)
>>> list(range(-9))
[]
```

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？

Python 2:

```
>>> range(0, -9, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8]
>>> range(0, -9, -2)
[0, -2, -4, -6, -8]
```

Python 3:

```
>>> range(0, -9, -1)
range(0, -9, -1)
>>> list(range(0, -9, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8]
```

有了这个内置函数，很多事情就简单了。比如：

```
>>> range(0, 101, 2)      #Python 2。如果用Python 3，类似前述演示
[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, 70, 72, 74, 76, 78, 80, 82, 84, 86,
88, 90, 92, 94, 96, 98]
```

100以内的自然数中的偶数组成的列表，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"]`，要得到这个列表的所有序号组成的列表，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
ex)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

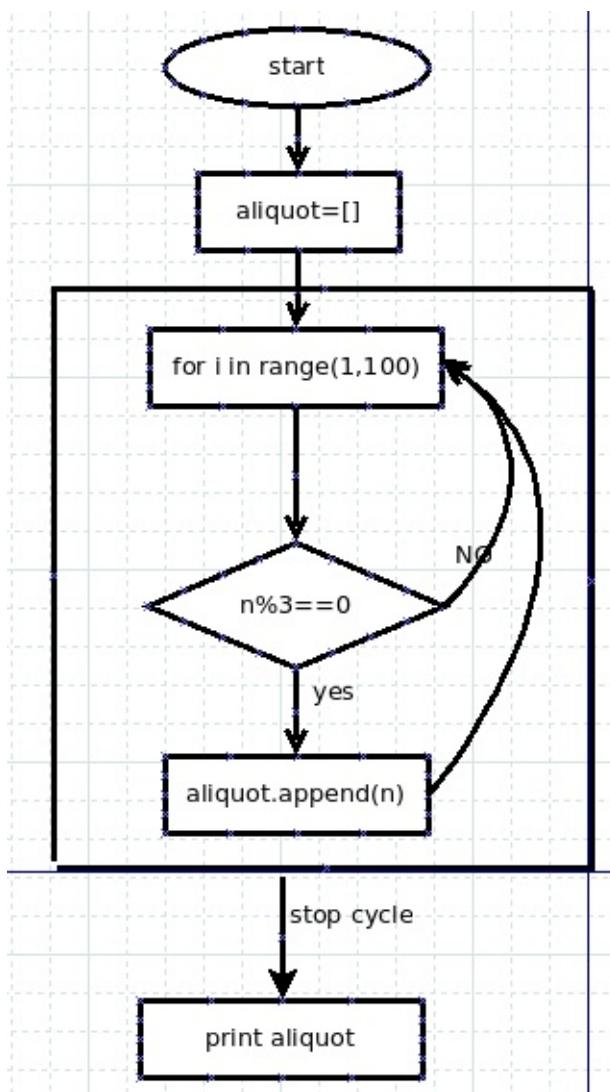
```

再用手指头指着列表里面的元素，数一数，是不是跟结果一样。

例：找出小于100的能够被3整除的正整数。

分析：这个问题有两个限制条件，第一是小于100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 == 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：

语句(3)

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

aliquot = []

for n in range(1,100):
    if n % 3 == 0:
        aliquot.append(n)

print aliquot      #Python 3: 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]
```

for循环在Python中应用广泛，所以，下节还要深入研究这个语句的使用，不要小看它，虽然简单，但内涵深刻。

[总目录](#) | [上节：语句\(2\)](#) | [下节：语句\(4\)](#)

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

"I give you a new commandment, that you love one another. Just as I have loved you, you also should love one another. By this everyone will know that you are my disciples, if you have love for one another." (JOHN 14:34-35)

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(**iterable**)”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，如果是让Python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，而且，用Python编程，还要追求优雅。

语句(4)

于是，`zip()`——一个内建函数姗姗来迟，它可以让同样的问题有不一样的解决途径，看起来更有范儿。

先要看一看`zip()`的文档，以了解它的身世。

在Python 2中，是这样描述的：

```
>>> help(zip)
Help on built-in function zip in module __builtin__:

zip(*args, **kwargs)
    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.
```

在Python 3中，相比Python 2，有一些稍微的差别。

```
>>> help(zip)
Help on class zip in module builtins:

class zip(object)
| zip(iterator1 [, iterator2 [...]]) --> zip object
|
|     Return a zip object whose .__next__() method returns a tuple where
|     the i-th element comes from the i-th iterable argument. The .__next__()
|     method continues until the shortest iterable in the argument sequence
|     is exhausted and then it raises StopIteration.
```

Python 2中，参数是`seq1, seq2, ...`，意思是序列数据；在Python 3中，参数需要时可迭代对象。这点差别，通常是没有影响的，因为序列也是可迭代的。值得关注的是返回值，在Python 2中，返回值是一个列表对象，里面以元组为元素；而Python 3中返回的是一个`zip`对象。

通过实验来理解上面的文档：

Python 2：

```
>>> a = "qiwsir"
>>> b = "github"
>>> zip(a, b)
[('q', 'g'), ('i', 'i'), ('w', 't'), ('s', 'h'), ('i', 'u'), ('r', 'b')]
```

Python 3：

语句(4)

```
>>> zip(a, b)
<zip object at 0x00000000003521D08>
>>> list(zip(a, b))
[('q', 'g'), ('i', 'i'), ('w', 't'), ('s', 'h'), ('i', '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)          #这是Python 2的结果，如果是Python 3，请仿照前面的方式显示查看
[(1, 9), (2, 8), (3, 7)]

>>> m = {"name": "lang"}
>>> n = {"qiwsir": "python"}
>>> zip(m, n)          #这是Python 2的结果，如果是Python 3，请仿照前面的方式显示查看
[("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，它的参数必须是序列，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，做为一个列表的元素。

下面是比较特殊的情况，参数是一个序列对象的时候，生成的结果样子：

```
>>> a ='qiwsir'
>>> c = [1, 2, 3]
>>> zip(c)          #这是Python 2的结果，如果是Python 3，请仿照前面的方式显示查看
[(1,), (2,), (3,)]
>>> zip(a)
[('q',), ('i',), ('w',), ('s',), ('i',), ('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！

语句(4)

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整数类型，不能跟后面的字符串类型相加，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。

语句(4)

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？

```
>>> 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：有一个字典，`myinfor = {"name": "qiwsir", "site": "qiwsir.github.io", "lang": "python"}`，将这个字典转换成：`infor = {"qiwsir": "name", "qiwsir.github.io": "site", "python": "lang"}`

解析：

解法一，用 `for` 循环：

```
>>> myinfor = {"name": "qiwsir", "site": "qiwsir.github.io", "lang": "python"}  
>>> infor = {}  
>>> for k, v in myinfor.items():  
...     infor[v] = k  
...  
>>> infor  
{'python': 'lang', 'qiwsir.github.io': 'site', 'qiwsir': 'name'}
```

解法二，用 `zip()`：

```
>>> 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。

为了能够窥探内部的奥秘，我们将那一行分解开来。

```
>>> myinfor.values()          # 得到字典值的列表  
['python', 'qiwsir', 'qiwsir.github.io']  
>>> myinfor.keys()          # 得到字典键的列表  
['lang', 'name', 'site']  
>>> temp = zip(myinfor.values(), myinfor.keys())      # 压缩成一个列表，每个元素是一个元组，元  
组中第一个是值，第二个是键  
>>> temp  
    # Python 3 中是一个 zip 对象  
[('python', 'lang'), ('qiwsir', 'name'), ('qiwsir.github.io', 'site')]  
>>> dict(temp)                # 这是函数 dict() 的功能，将上述列表转化为字典  
{'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

`enumerate()` 也是内建函数。

本来我们可以通过 `for i in range(len(list))` 的方式得到一个列表的每个元素对应的索引，然后在用 `list[i]` 的方式得到该元素。这是前面用过的路数。

但是，如果要同时得到元素索引和元素怎么办？

```
>>> week = ['monday', 'sunday', 'friday']
>>> 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
...                                         #如果使用Python 3，请自行更换为print(week[i] + ' is ' + str(i))
monday is 0
sunday is 1
friday is 2
```

内建函数`enumerate`，能够实现类似的功能，并且简化。

```
>>> for (i, day) in enumerate(week):
...     print day + ' is ' + str(i)          #Python 3: 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：

语句(4)

```
>>> mylist = ["qiwsir", 703, "python"]
>>> enumerate(mylist)
<enumerate object at 0xb74a63c4>
>>> 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，但是，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,
 'l'), (17, 'a'), (18, 'o'), (19, 's'), (20, 'h'), (21, 'i'), (22, '?'), (23, ' '), (24,
 'C'), (25, 'a'), (26, 'n'), (27, 'g'), (28, 'l'), (29, 'a'), (30, 'o'), (31, 's'), (32,
 'h'), (33, 'i'), (34, ' '), (35, 'i'), (36, 's'), (37, ' '), (38, 'a'), (39, ' '),
 (40, 'g'), (41, 'o'), (42, 'o'), (43, 'd'), (44, ' '), (45, 't'), (46, 'e'), (47, 'a'),
 (48, 'c'), (49, 'h'), (50, 'e'), (51, 'r'), (52, '.')]
```

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。

列表解析

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，并且将结果放在列表中，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有一个非常强大的功能，就是列表解析，它这样使用：

```
>>> 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，读者还不惊叹吗？

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，道出了列表解析的真谛：

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列表，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，感悟一下列表解析的魅力。

语句(4)

```
>>> 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']
```

本节中已经演示过的问题，都能用列表解析来重写。读者不妨试试。

在很多情况下，列表解析的执行效率高，代码简洁明了。是实际写程序中经常被用到的。

现在Python的两个版本，对列表解释上，还是有一点点差别的，请认真看下面的比较操作。

Python 2:

```
>>> i = 1
>>> [ i for i in range(9)]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> i
8
```

Python 3:

```
>>> i = 1
>>> [i for i in range(9)]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> i
1
```

有没有观察到区别？

先 `i = 1`，然后是一个列表解析式，非常巧合的是，列表解析式中也用了变量 `i`。这种情况，在编程中是常常遇到的，我们通常把 `i=1` 中的变量 `i` 称为处于全局命名空间里面（命名空间，是一个新词汇，暂且用起来，后面会讲述），而列表解析式中的变量 `i` 是在列表解析内，称为处在局部命名空间。在Python 3中，`for`循环里的变量不再与全局命名空间的变量有关联了。这种改变，窃以为，是进步。

对于循环，除了`for`之外，还有一个叫做`while`。

[总目录](#) | [上节：语句\(3\)](#) | [下节：语句\(5\)](#)

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)

他们虽然知道神，却不当做神荣耀他，也不感谢他。他们的思念变为虚妄，无知的心就昏暗了。自称为聪明，反成了愚拙；经不能朽坏值神的荣耀变为偶像，仿佛必朽坏的人和飞禽、走兽、昆虫的样式。(ROMANS 1:21-23)

语句(5)

关于循环，Python中除了for，还有一个是while。

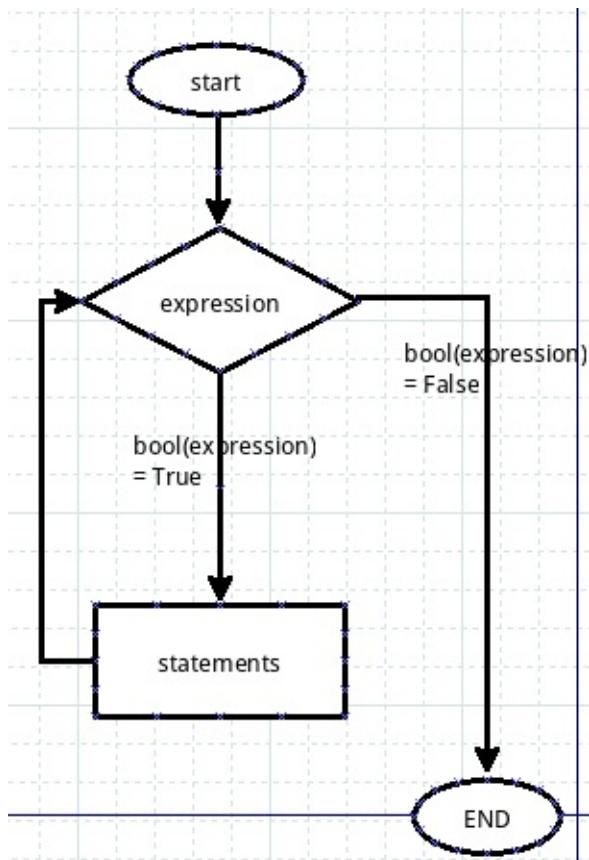
while，翻译成中文是“当...的时候”，这个单词在英语中，常常用来做为时间状语，while ... someone do something，这种类型的说法是有的。

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：')

    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，读者看看是否可以修改？

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:")      #Python 3请使用input()
```

程序用 `num_input` 变量接收了输入的内容。但是，请一定要注意，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：

语句(5)

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. 输入的是是否是整数
2. 如果是整数，是否在1到100之间。

可以这样做：

```
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整数类型，一边是通过输入函数得到的字符串类型数据。在某些情况下可以运行，为什么？读者能理解吗？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(1,100)

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 smaller."
        elif number < int(num_input):
            print "your number is bigger."
        else:
            print "There is something bad, I will not work"

```

上述代码是在Python 2下调试的，如果读者使用的是Python 3，请在前面已经陈述过的几个地方进行修改。

代码供参考，更欢迎读者改进它。

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

```

语句(5)

上述代码在Python 2中调试，如果读者使用Python 3，只需要将print语句更换为 print() 函数即可。

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
```

为了兼顾Python 3，还是重复上面那句话：上述代码在Python 2中调试，如果读者使用Python 3，只需要将print语句更换为 print() 函数即可。

其实，对于 break 和 continue ，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

while...else 有点类似 if ... else ，只需要一个例子就可以理解。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
```

依然要说明：上述代码在Python 2中调试，如果读者使用Python 3，只需要将print语句更换为 `print()` 函数即可。

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."
```

依然要说明：上述代码在Python 2中调试，如果读者使用Python 3，只需要将print语句更换为 `print()` 函数即可。这是很煞风景的一句话。

读者是否能够读懂这段代码的含义？

语句(5)

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(99, 81, -1)` 看看是什么结果？

循环就这么多内容，但它即将迎来一个最大的用处。

[总目录](#) | [上节：语句\(4\)](#) | [下节：文件\(1\)](#)

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)

Be on your guard! If another disciple sins, you must rebuke the offender, and if the same person sins against you seven times a day, and turns back to you seven times and says, 'I repent,' you must forgive. (LUKE17:3-4)

文件(1)

文件，是computer姑娘中非常重要的东西。如在Linux操作系统中，所有的东西都被保存到文件中。

在Python里，它也很重要。只不过，Python 2和Python 3对它的态度有点不同。

在Python 2中，文件也是一种内建类型对象，其地位等同于前面已经学习过的列表、整数、字符串等类型。

在交互模式下，我们可以用 `dir()` 这种已经熟练的方法查看相关属性和方法。

```
>>> dir(file)
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__', '__format__', '__getattribute__', '__hash__', '__init__', '__iter__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'close',
'closed', 'encoding', 'errors', 'fileno', 'flush', 'isatty', 'mode', 'name', 'newlines',
'next', 'read', 'readinto', 'readline', 'readlines', 'seek', 'softspace', 'tell',
'truncate', 'write', 'writelines', 'xreadlines']
```

然而，这一切在Python 3中都变了。在Python 3中，再没有 `file` 这个内建类型了。如果读者使用的是Python 3，执行 `dir(file)`，会报错，并且显示 `NameError: name 'file' is not defined`。

但，这并不妨碍我们对文件的研究。

暂且观察Python 2的 `dir(file)` 结果，里面有 `__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对象是可迭代的(`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。

读文件

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：

文件(1)

```
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`，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          #Python 3: print(line)  
...  
learn python  
  
http://qiwsir.github.io  
  
qiwsir@gmail.com
```

提醒初学者注意，在那个目录里输入了启动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对象进行赋值是一样的。

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，把读到的文件中的每行，赋值给变量 `line`。也可以理解为，用 `for` 循环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。`print line` 或者 `print(line)` 默认情况下，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,        #Python 3: print(line, end='')  
...  
learn python  
http://qiwsir.github.io  
qiwsir@gmail.com
```

如果读者完成了上述操作，紧接着做下面的操作：

文件(1)

```
>>> 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？  
...  
>>>
```

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。后面会对此进行讲解。

在这里，如果要再次读取，那就从新 `f = open('130.txt')` 。有点麻烦！怎奈知识尚不充足。

使用Python 3的读者，这里要注意了，如果你已经执行了 `f = open('130.txt')` ，就是已经建立了一个文件对象，这时候可以使用 `dir()` 来查看这个对象的方法和属性了。

```
>>> f = open("130.txt")  
>>> dir(f)  
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__enter__', '__eq__', '__exit__', '__format__', '__ge__', '__getattribute__', '__g
etstate__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__lt__', '__ne__',
 '__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__
sizeof__', '__str__', '__subclasshook__', '_checkClosed', '_checkReadable', '_checkSee
kable', '_checkWritable', '_finalizing', 'buffer', 'close', 'closed', 'detach', 'encod
ing', 'errors', 'fileno', 'flush', 'isatty', 'line_buffering', 'mode', 'name', 'newlin
es', 'read', 'readable', 'readline', 'readlines', 'seek', 'seekable', 'tell', 'truncat
e', 'writable', 'write', 'writelines']
```

不管两个版本的显示结果有哪些不同，它们相同的是都有 `__iter__` 属性，意味着在两个版本中，文件对象都是可迭代的。

特别提醒，因为我所演示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
```

文件(1)

```
>>> 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的路径分隔符与此不同。）
>>> for line in f:
...     print line
...
learn python

http://qiwsir.github.io

qiwsir@gmail.com
```

读文件，只是针对文件的操作之一，还有创建文件。

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")
>>> nf.close()
```

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。

创建文件，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()` 操作文件，可以有不同的模式。看下表：

文件(1)

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>
```

Python 3中显示的结果信息更丰富、明确。

```
>>> f
<_io.TextIOWrapper name='130.txt' mode='r' encoding='cp936'>
```

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           #Python 3: 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：

文件(1)

```
$ 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。

文件(1)

[总目录](#) | [上节：语句\(5\)](#) | [下节：文件\(2\)](#)

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

文件(2)

你这论断人的，无论你是谁，也无可推诿，你在甚么事上论断人，就在甚么事上定自己的罪。因你这论断人的，自己所行却和别人一样。我们知道这样行的人，神必照真理审判他。(ROMANS 2:1-2)

文件(2)

文件，也是对象。

在Python 2中还是一种内建的类型，在Python 3中被取消了，可是，这并不意味这其地位降低。

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=100
0, st_gid=1000, st_size=69L, st_atime=1407897031, st_mtime=1407734600, st_ctime=140773
4600)

>>> file_stat.st_ctime                #这个是文件创建时间
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, tm_isdst=0)
```

read/readline/readlines

用 open() 能够打开文件，在用for循环，可以将文件的内容读取出来。

但是，在查看文件的属性和方法的时候，会看到三个方法： read, readline, readlines 。

从名称上看，它们应该都是跟“读”有关的，但是，又应该有差别。

的确如此。

还是对比着看一看，并且还要Python 2和Python 3的文档对比。

Python 2 :

```
>>> help(file.read)
Help on method_descriptor:
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.

>>> help(file.readline)
Help on method_descriptor:
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.

>>> help(file.readlines)
Help on method_descriptor:
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.
```

Python 3 :

```
>>> f = open("130.txt")
>>> help(f.read)
Help on built-in function read:
read(size=-1, /) method of _io.TextIOWrapper instance
    Read at most n characters from stream.

    Read from underlying buffer until we have n characters or we hit EOF.
    If n is negative or omitted, read until EOF.

>>> help(f.readline)
Help on built-in function readline:
readline(size=-1, /) method of _io.TextIOWrapper instance
    Read until newline or EOF.

    Returns an empty string if EOF is hit immediately.

>>> help(f.readlines)
Help on built-in function readlines:
readlines(hint=-1, /) method of _io.TextIOWrapper instance
    Return a list of lines from the stream.

    hint can be specified to control the number of lines read: no more
    lines will be read if the total size (in bytes/characters) of all
    lines so far exceeds hint.
```

对照一下上面的说明，三个的异同就显现了。而且，要想了解Python 2和Python 3之间的不同，还可以将两个版本的对照一下。好像也没有什么不同哦。

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 data source. The data source is usually called a file or stream. In general, the EOF is either determined when the reader returns null as seen in Java's BufferedReader,[2] or sometimes people will manually insert an EOF character of their choosing to signal when the file has ended.

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。

文件(2)

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 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.
```

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 an  
d 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.\n'  
>>> f.close()
```

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。

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      #Python 3: 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：

文件(2)

```
>>> 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,           #Python 3: print(line, end='')

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：

文件(2)

```
>>> 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 co
me and my heart burdened be;\n', 'Then, I am still and wait here in the silence,\n', '
Until you come and sit awhile with me.\n', 'You raise me up, so I can stand on mountai
ns;\n', 'You raise me up, to walk on stormy seas;\n', 'I am strong, when I am on your
shoulders;\n', 'You raise me up: To more than I can 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 ,           #Python 3: print(line, end='')
...
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 ,           #Python 3: print(line, end='')
...
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。

文件(2)

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 ,      #Python 3: print(line, end='')  
...  
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，是因为文件是可迭代的对象，直接用 `for` 来迭代即可。

seek

这个函数的功能就是让指针移动。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'
```

现在来看 `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
```

这不是在第17行。在 Python 2 中返回的是 `17L`，很容易让人产生如此误解。但是在 Python 3 中返回的是 `17`，又可能迷茫，单位是什么呢？

文件(2)

读者不妨数一数，是按照字符，'You Raise Me Up\n' 从0开始，到最后，是不是正好为 17。提醒注意的是，如果你用汉语的等文字，就需要注意编码问题了。请查看本教程有关编码的讨论。

```
>>> 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 #Python 3返回：4
```

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 end 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。

前面已经提到了，文件是可迭代的，并且还学过其它可迭代的对象，看来迭代是一个有必要讨论的问题。

[总目录](#) | [上节：文件\(1\)](#) | [下节：迭代](#)

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

你竟任着刚硬不悔改的心，为自己积蓄忿怒，以致神震怒，显他公义审判的日子来到。他必照各人的行为报应各人。凡恒心行善，寻求荣耀、尊贵和不能朽坏之福的，就以永生报应他们；惟有结党不顺从真理，反顺从不义的，就以忿怒、恼恨报应他们。
(ROMANS 2:7-8)

迭代

Bill正在介绍他的项目，嘴里不断蹦出“loop、iterate、traversal、recursion”这些单词，夹杂在汉语汇总。旁边的小白们，都瞠目结舌，“不明觉厉”，心中的敬佩油然而生，

其实，Bill不是真正的大牛，我见过真牛的，他们绝对不会这么说的，他们通常总是轻描淡写，不管我认为多么麻烦的，他们都是举重若轻地解决：

“你就循环一下，然后来个迭代”，

“最后只要花几分钟写个递归就好了”

然后再问“这个你懂吗？”。

哦，这就是真正牛X的程序员。虽然还是不懂。

所以，我们还是老老实实地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），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,          #Python 3: print(i, end=' ')
...
q i w s i r
```

迭代

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()          #Python 3，请用：lst_iter.__next__()
'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：

```
>>> help(iter)
Help on built-in function iter in module builtins:

iter(...)
    iter(iterable) -> iterator
    iter(callable, sentinel) -> iterator

    Get an iterator from an object. In the first form, the argument must
    supply its own iterator, or be a sequence.
    In the second form, the callable is called until it returns the sentinel.
```

`iter()` 函数返回的是一个迭代器对象。关于迭代器对象，本教程会有专门讲授。

```
>>> type(lst_iter)
<type 'listiterator'>      #Python 3: <class 'list_iterator'>
```

所有的迭代器对象，在Python 2中有一个 `next()` 方法，在Python 3中，相应的修改为了 `__next__()`。所以使用不同版本，要注意一下这个方法名称的变更。

迭代器，当然是可迭代的。本教程已经介绍过如果判断一个对象是否为可迭代对象的方法了。

在上面的举例中，`next()` 或者 `__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”，当然不能这样一个一个地敲啦，于是就：

```
>>> while True:
...     print lst_iter.next()          #Python 3: print(lst_iter.__next__())
...
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
StopIteration
```

先不管错误，再来一遍。

```
>>> lst_iter = iter(lst)
>>> while True:
...     print lst_iter.next()          #Python 3: print(lst_iter.__next__())
...
...
q
i
w
s
i
r
Traceback (most recent call last):      #读取到最后一个之后，停止循环
  File "<stdin>", line 2, in <module>
StopIteration
```

看上面的演示的例子会发现，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 `next()` 或者 `__next__()`，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` 的信息（名叫：停止迭代，这分明是警告）。

还要关注迭代器对象的另外一个特点，当对象 `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。

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：

迭代

```
>>> 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
'It's 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
...     print line,      #Python 3: print(line)
...                           #没有打印出什么东西
```

这段代码之所没有打印出东西来，是因为经过前面的操作，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,      #Python 3: print(line,end='')
...
Learn python with qiwsir.
There is free python course.
The website is:
http://qiwsir.github.io
Its language is Chinese.
```

上面过程用`next()`或者`__next__()`也能够读取。

```
>>> f = open("208.txt")
>>> f.next()          #Python 3: 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()
'Its language is Chinese.\n'
>>> f.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

如果用 `next()` 或者 `__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()` 或者 `__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。

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', 'Its 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', 'Its language is Chinese.\n')

>>> "$$".join(open('208.txt'))
'Learn python with qiwsir.\n$$There is free python course.\n$$$The website is:\n$$ht
tp://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'
```

迭代

上述方式，在编程实践中不一定用得上，只是向读者展示一下，并且要明白，可以这么做，不是非要这么做。

最后透露，字典和元组都可以写成类似列表解析式的样式，也可以迭代，读者不妨摸索一下。

[总目录](#) | [上节：文件\(2\)](#) | [下节：练习](#)

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

练习

因为世人都犯了罪，亏缺了神的荣耀。如今却蒙神的恩典，因基督耶稣的救赎，就白白的称义。神设立耶稣作挽回祭，是凭着耶稣的血，藉着人的信，要显明神的义。因为他用忍耐的心，宽容人先时所犯的罪。好在今时显明他的义，使人知道自己为义，也称信耶稣的人为义。(ROMANS 3:23-26)

练习

有学习者问：“看完你的教程，我可以达到什么水平？”

我无语。

同样小学、中学甚至大学，同班同学读同一本书，天天由同一个老师讲课，为什么有的是学渣、有的是学霸？

指望读完某本书，就要达到某个水平，是一种懒汉思维。

我在本教程的开篇已经说过了，从小工到专家的道路，就普通人来讲，“一万小时”的训练是必不可少的，也就是要做练习。

本教程并没有专门提供练习的题目。但并不意味着不需要练习。本节就是一个示例，告诉各位读者，自己找题目做是非常必要的。

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。主要是练习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

练习

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

raw = [1,2,3,4,5,6,7,8,9,0]
print raw      #Python 3: print(raw)

b = raw.pop(0)
raw.append(b)
print raw      #Python 3: 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。这里用到的主要是列表的两个函数 `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。抽出一个，看是几，就将这个数字写到列表中，直到抽出第40个。这样得到的列表是随机了。

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',
'TWOPPI', 'WichmannHill', '_BuiltinMethodType', '_MethodType', '__all__', '__builtin__',
'__doc__', '__file__', '__name__', '__package__', '_acos', '_ceil', '_cos', '_e',
'_exp', '_hashlib', '_hexlify', '_inst', '_log', '_pi', '_random', '_sin', '_sqrt',
'_test', '_test_generator', '_urandom', '_warn', 'betavariate', 'choice', 'division',
'expovariate', 'gammavariate', 'gauss', 'getrandbits', 'getstate', 'jumpahead', 'lognormvariate',
'normalvariate', 'paretovariate', 'randint', 'random', 'randrange', 'sample',
'seed', 'setstate', 'shuffle', 'triangular', 'uniform', 'vonmisesvariate', 'weibullvariate']
```

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          #Python 3不需要这行引入模块的操作
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      #Python 3: 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}".format(ave_num)
#Python 3: print("the average score is:{:.1f}".format(ave_num))
print "There are {}students less than average.".format(less_ave)
#Python 3: print("There are {} students less than average.".format(less_ave))

sorted_score = sorted(score, reverse=True)      #对原列表排序
print sorted_score      #Python 3: 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打一个空格。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，而是把字符串两边的空格都去掉。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
                                #Python 3: print(string)

str_lst = string.split(" ")    #以空格为分割，得到词汇的列表
print str_lst                 #Python 3: print(str_lst)

words = [s.strip() for s in str_lst]  #去除单词两边的空格
print words                   #Python 3: print(words)

new_string = " ".join(words)    #以空格为连接符，将单词链接起来
print new_string               #Python 3: 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，那么列表中单词两边就没有空格了。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。

练习

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

string = "I love code."
print string      #Python 3: print(string)，以下类似操作都这样处理，不再重复。

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

根据高德纳（Donald Ervin Knuth）的《计算机程序设计艺术》（The Art of Computer Programming），1150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列奥那多（意大利人斐波那契Leonardo Fibonacci），他描述兔子生长的数目时用上了这数列。

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）它们可以生育 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 假设在n月有兔子总共a对，n+1月总共有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）。而新生育出的兔子对数等于所有在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：

```
a[0] = 0          (n=0)
a[1] = 1          (n=1)
a[n] = a[n-1] + a[n-2]  (n>=2)
```

我们要做的事情是用程序计算出n=100时的值。

练习

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？后面会介绍。

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。

练习5

问题描述

在数学中，用 $n!$ 来表示阶乘。例如， $4!=1\times2\times3\times4=24$ 。如果将n个物体排列，有多少种排列方式，那就是 $n!$ 。根据定义， $0!=1$ 。

解析

下面用Python程序来计算阶乘。

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

n = int(raw_input("Enter an interger >=0: "))

fact = 1

for i in range(2, n + 1):
    fact = fact * i

print str(n) + " factorial is " + str(fact)
```

这是用for循环来实现的，当然，你也可以使用while循环来计算阶乘。其代码如下：

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

n = int(raw_input('Enter an integer >= 0: '))

fact = 1
i = 2
while i<=n:
    fact = fact * i
    i += 1

print str(n) + " factorial is " + str(fact)
```

在计算阶乘的时候，如果你输入比较大的整数，也没有问题，因为Python对整数的最大取值没有限制，所以，即时遇到大整数，也可放心使用。

[总目录](#) | [上节：迭代](#) | [下节：自省](#)

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

我们既因信称义，就藉着我们的主耶稣基督，得与神相和。我们又藉着他，因信得进入现在所站的这恩典中，并且欢欢喜喜盼望神的荣耀。不但如此，就是在患难中，也是欢欢喜喜的，因为知道患难生忍耐。忍耐生老练，老练生盼望。盼望不至于羞耻，因为所赐给我们的圣灵，将神的爱浇灌在我们心里。(ROMANS 5:1-5)

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。

本节内容，并不是本教程的必须部分，放到这里，是让学习者对如何应用Python的“自省”机制实现自学，有一些了解罢了。

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()
['GDescriptorBased', 'GInitiallyUnowned', 'GPollableInputStream', 'GPollableOutputStream',
'__builtins__', '__doc__', '__name__', '__package__', 'keyword']
>>> import math
>>> dir()
['GDescriptorBased', 'GInitiallyUnowned', 'GPollableInputStream', 'GPollableOutputStream',
'__builtins__', '__doc__', '__name__', '__package__', 'keyword', 'math']
```

`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__` 看起来像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__)
['ArithmetricError', 'AssertionError', 'AttributeError', 'BaseException', 'BufferError',
 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'E
xception', 'False', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'KeyError', 'Keyboa
rdInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented', 'No
tImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'Referenc
eError', 'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxEr
ror', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError', ''
UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'Unico
deTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivi
sionError', '_', '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs
', 'all', 'any', 'apply', 'ascii', 'basestring', 'bin', 'bool', 'buffer', 'bytearray',
 'bytes', 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'co
pyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'execfil
e', 'exit', 'file', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'h
asattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'intern', 'isinstance', 'issubcl
ass', 'iter', 'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'memoryview',
'min', 'next', 'gettext', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
'quit', 'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round', 'set',
'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type',
'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

`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__', '__getitem__', '__getnewargs__', '__getslice__', '__
gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__
ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__formatter_field_name_sp
lit', '__formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode', 'endsw
ith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islow
er', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition',
 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'sp
litlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

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像现实世界中的对象。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（像运算符，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
['__builtins__', '__doc__', '__name__', '__package__', 'keyword', 'math']
>>> directory = dir      #新变量
>>> directory()          #跟dir()一样的结果
['__builtins__', '__doc__', '__name__', '__package__', 'directory', 'keyword', 'math']
>>> 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', 'DictProxyType', 'DictType', 'DictionaryType', 'EllipsisType',
 'FileType', 'FloatType', 'FrameType', 'FunctionType', 'GeneratorType', 'GetSetDescriptorType',
 'InstanceType', 'IntType', 'LambdaType', 'ListType', 'LongType', 'MemberDescriptorType',
 'MethodType', 'ModuleType', 'NoneType', 'NotImplementedType', 'ObjectType',
 'SliceType', 'StringType', 'StringTypes', 'TracebackType', 'TupleType', 'TypeType',
 'UnboundMethodType', 'UnicodeType', 'XRangeType', '__builtins__', '__doc__', '__file__',
 '__name__', '__package__']

>>> 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 `clist` 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，这个东东我们也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

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.文档的类型有很多种，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种，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](#) | [上节：练习](#) | [下节：函数\(1\)](#)

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

谁能使我们与基督的爱隔绝呢？难道是患难么，是困苦么，是逼迫么，是饥饿么，是赤身露体么，是危险么，是刀剑么。然而靠着爱我们的主，在这一切的事上，已经得胜有余了。因为我深信无论是死、是生、是天使、是掌权的，是有能的，是现在的事，是将来的事，是高处的，是低处的，是别的受造之物，都不能叫我们与神的爱隔绝。这爱是在我们的主基督耶稣里的。(ROMANS 8:35,37-39)

函数(1)

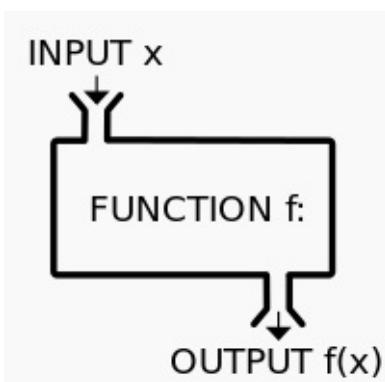
函数，对于人类来讲，能够发展到这个数学思维层次，是一个飞跃。可以说，它的提出，直接加快了现代科技和社会的发展，不论是现代的任何科技门类，乃至于经济学、政治学、社会学等，都已经普遍使用函数。

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，則此爲彼之函數”。

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：



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：

$$f(2) = 4 \cdot 2 + 3 = 11$$

$$\text{所以：} f(2) = 11$$

函数(1)

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。

读者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。很巧，在“知乎”上还真有人询问[这个问题](#)，可以阅读。

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...`，甚至用 `alpha,beta` 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了。现在，就按照初中数学的水平来研究 `Python` 中的变量。

通常使小写字母来命名 `Python` 中的变量，也可以是用下划线连接的多个单词。比如：`alpha`，`x`，`j`，`p_beta`，这些都可以做为 `Python` 的变量。

下面按照纯粹数学的方式，在 `Python` 中建立函数。

```
>>> 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。在纯粹数学中，也常这么用。这种方式在 `Python` 中还有效吗？

既然在上面已经建立了一个函数，那么我就改变变量 `a` 的值，看看得到什么结果。

```
>>> a = 3
>>> y
8
```

函数(1)

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——这跟纯粹数学有所区别了。

用纯粹数学的方式建立函数，对Python不适用，如果非要找个根由，我想可能是“=”造成的，这个符号在数学中是等号，但是在Python中，包括所有的高级编程语言中，是“赋值”。这是我的肤浅理解。更深层的缘由，还在于计算机处理数据的原理与人不同。所以，要有一种新的定义函数的方式

定义函数

在Python中，规定了一种定义函数的格式，下面的举例就是一个函数，以这个函数为例来说明定义函数的格式和调用函数的方法。

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

def add_function(a, b):
    c = a + b
    return c

if __name__ == "__main__":
    result = add_function(2, 3)
    print(result)          #python3: print(result)
```

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。

函数(1)

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`：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。`def` 做在那一行，包括后面的`add_function(a, b)`，被称为函数头。
- `add_function`：这是函数名称。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)`：这是参数列表。要写在括号里面。这是一个变量（参数）列表，其中的变量（参数）指向函数的输入。在这个例子中，函数有两项输入，分别是`a`和`b`。在通常的函数中，输入项没有限定，可以是任意数量，当然也可以没有输入，这时候的参数列表就是一对空着的圆括号`()`，但是，必须得有这个圆括号。
- `:`：这个冒号非常非常重要，如果少了，就报错了。这和前面的语句是类似的，冒号表示函数头结束，下面要开始函数体的内容了。
- `c = a + b`：这一行开始，就是函数体。函数体使一个缩进了四个空格的代码块，完成你需要完成的工作。在这个代码块中，可以使用函数头中的变量，当然，不使用也可以。缩进四个空格。这是Python的规定，要牢记，不可丢掉，丢了就报错。这句话就是将函数头的变量相加，结果赋值与另外一个变量`c`。
- `return c`：还是提醒看官注意，缩进四个空格。`return`是函数的关键字，意思是返回一个值。`return`语句执行时，Python跳出当前的函数并返回到调用这个函数的地方。在下面，有调用这个函数的地方`result = add_function(2, 3)`。但是，函数中的`return`语句也不是必须要写的，如果不写，Python将认为以`return None`来作为结束的。也就是说，如果你的函数中没有`return`，事实上，在调用的时候，Python也会返回一个结果，这个结果就是`None`。
- `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。
- `result = add_function(2, 3)`：这是调用前面建立的函数，并且传入两个值`a=2`和`b=3`。仔细观察传入参数的方法，就是相当于把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）。当函数运行，遇到了`return`语句，就将函数中的结果返回到这里，赋值给`result`。还要啰嗦一句，是“相当于”把2和3分别放在`a`和`b`的位置，这个“相当于”的是有含义的，暂且存疑，后续会讲解。

解牛完毕，做个总结：

定义函数的格式为：

函数(1)

```
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，有人习惯用aaBb的样式，但我不推荐
- 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引用关系，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。

函数(1)

```
>>> 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。`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 `add(x,y)` 之后，也建立了一个 `add(x,y)` 的对象，这个对象与变量 `result` 可以建立引用关系，并且 `add(x,y)` 将运算结果返回。于是，通过 `result` 就可以查看运算结果。

```
>>> add  
<function add at 0x000000000007BAC80>
```

如果使用 `add(x, y)` 的样式，是调用那个函数。但是如果只写函数的名字，不写参数列表，就如同上面那样，我们得到的是该函数在内存汇总的存储信息。你还可以这样做：

```
>>> type(add)  
<class 'function'>      #Python 2下的反馈信息略有差异
```

这说明 `add` 是一个对象，因为只有对象才有类型，并且它是一个 `function` 类。按照我们的经验，对象都可以与一个变量建立引用关系，从而通过那个变量访问对象。

```

>>> r = add
>>> r
<function add at 0x000000000007BAC80>
>>> r(3, 4)
7
>>> add(3, 4)
7
>>> type(r)
<class 'function'>

```

通过赋值语句，变量 `r` 和函数对象建立了引用关系之后，就可以做所有 `add(x, y)` 能做的事情，因为 `r` 就是那个函数的代表。

刚开始接触函数，可能有点吃力。先放松一下，看看“名不正言不顺”的Python版。

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，希望不要影响读者正确理解。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？比如Aoi sola，这个人怎么样？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，比如山本五十六，在名字中间出现数字，就好像我们的张三李四王二麻子那样随便，不过，有一种说法，“山本五十六”的意思是这个人出生时，他父亲56岁，看来跟张三还不一样的。

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。但我不提倡，这是我非常鲜明的观点。
- 函数的参数：命名方式同变量（本质上就是变量）。如果一个参数名称和Python保留的关键字冲突，通常使用一个后缀下划线会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`。

函数(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。（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：

Python2:

```
>>> 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
...
```

Python 3:

函数(1)

```
>>> def add(x, y):
    print("x={}".format(x))
    print("y={}".format(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)
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
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           #Python 3: print("x={}".format(x)), 以下类似, 从略。
...     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
```

函数(1)

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'
```

请读者在闲暇之余用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。

在程序中调用函数，还需要注意一个貌似废话的事项，那就是“先定义，后使用”。说是废话，是因为在理解上似乎当然这样，但是，在实践中，常会遇到此类错误。

```
>>> def foo():
    print('Hello, Teacher Cang!')      #Python 2的使用者请自动调整为print语句
    bar()
```

这里定义了一个函数 `foo()`，在这个函数里面还调用了一个函数 `bar()`，但是这个 `bar()` 函数，此前并没有在什么地方定义。所以，如果调用 `foo()` 函数，就会这样：

```
>>> foo()
Hello, Teacher Cang!
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    foo()
  File "<pyshell#43>", line 3, in foo
    bar()
NameError: name 'bar' is not defined
```

`NameError` 是一种错误信息。错误不可怕，可怕的是不认真看提示信息，只要耐心地认真地阅读提示信息，就能晓得错误原因。提示信息中分明告诉我们，那个 `bar` 没有定义。

所以要必须先定义，后使用。

```
>>> def bar(): pass
```

这就定义了 `bar()`，虽然非常简短，函数体内的代码就一个 `pass`，意思是里面什么也不做，统统地`pass`。然后调用 `foo()`

```
>>> foo()
Hello, Teacher Cang!
```

函数(1)

不再报错了。

虽然将 `bar()` 定义在了 `foo()` 的后面，只要定义了，无论先后，就可以使用。

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。一定要记住复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。模块文件中复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，时不时地回来复习一下，能不断领悟其内涵。

[总目录](#) | [上节：自省](#) | [下节：函数\(2\)](#)

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

函数(2)

爱人不可虚假，恶要厌恶，善要亲近。爱弟兄，要彼此亲热；恭敬人，要彼此推让。殷勤不可懒惰。要心里火热，常常服侍主。在指望中要喜乐，在患难中要忍耐，祷告要恒切。(ROMANS 12:9-12)

函数(2)

返回值

所谓返回值，就是函数向调用函数的地方返回的数据。

编写一个斐波那契数列函数，来说明这个问题。还记得斐波那契数列吗？忘了没关系，看看本教程前面的内容即可。

我这里提供一段参考代码（既然是参考，显然不是唯一正确答案）：

```
#!/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fib，其参数是输入一个整数（但是，你并没有看到我在哪里做了对这个要输入的值的约束，就意味着，你输入非整数，甚至字符串，也使可以的，只是结果会不同，不妨试试吧），然后通过lst = fib(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fib()函数，最后有一个语句return result，意思是将变量result的值返回。返回给谁呢？这要看我们当前在什么位置调用该函数了。

函数(2)

在上面的程序中，以 `lst = fibs(10)` 语句的方式，调用了函数，那么函数就将值返回到当前状态，并记录在内存中，然后把它赋值给变量 `lst`。

注意：上面的函数只返回了一个返回值（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)
```

对这个函数，我们还可以用这样的方式来接收函数的返回值。

```
>>> x, y, z = my_fun()
>>> x
1
>>> y
2
>>> z
3
```

多么神奇。

也不怎么神奇，这也来源于我们前面已经熟知的赋值语句。其效果相当于：

```
>>> x, y, z = a
>>> x, y, z
(1, 2, 3)
```

不是所有的函数都有 `return` 的，比如有的函数，就是执行某个语句或者什么也不做，不需要返回值。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 foo():
...     pass
...
```

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 `pass`——什么也不做，当然是没有 `return` 了。

```
>>> a = foo()
```

我们再看看那个变量 `a`，到底是什么

函数(2)

```
>>> print a      #Python 3: print(a)
None
```

这就是没有 `return` 的函数，事实上返回的是一个 `None`。而 `None`，你有可以理解成没有返回任何东西。

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。

```
>>> foo()
```

特别注意那个 `return`，它还有一个作用，请先观察下面的函数和执行结果，并试图找出其作用。

```
>>> def my_fun():
...     print "I am coding."          #Python 3的用户请修改为print()
...     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`，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`，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` 在这里就有了一个作用，结束正在执行的函数，并离开函数体返回到调用位置，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。”

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，还有比较多的说明，这个被称为“文档”，在文档中主要是说明这个函数的用途。

函数(2)

```
#!/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.
```

函数的属性

任何对象都具有属性，比如“孔乙己的茴香豆”，这里“孔乙己”是一个对象，“茴香豆”是一个属性，世界上“茴香豆”很多，但是这里所说的“茴香豆”是比较特殊的，它归属于“孔乙己”。如果用符号的方式来表示“孔乙己的茴香豆”，一般习惯用句点（英文的）代替中间的“的”字，也就是句点表示了属性的归属，表示为： 孔乙己.茴香豆。

前面已经说过，函数是对象。那么它也有属性。

函数(2)

```
>>> def cang():
    """This is a function of canglaoshi"""
    pass
```

对于这个函数，最熟悉的一个属性就应该是前面提到的函数文档，它可以用句点的方式表示为 `cang.__doc__`。

```
>>> cang.__doc__
'This is a function of canglaoshi'
```

这就体现出这种方式表示属性的优势了，只要对象不同，不管属性的名字是否相同，用句点就可以说明该属性所对应的对象。

还可以为对象增加属性。

```
>>> cang.breast = 90
```

这样就为对象 `cang` 增加了一个属性 `breast`，并且设置该属性的值是90。接下来就可以调用该属性。

```
>>> cang.breast
90
```

还记得我们用来查看对象属性和方法的函数 `dir()` 吗？现在又可以请它出来，一览众属性。

```
>>> dir(cang)
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defaults__',
 '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__get__', '__getattribute__', '__globals__', '__gt__', '__hash__', '__init__', '__kwde
faults__',
 '__le__', '__lt__', '__module__', '__name__', '__ne__', '__new__', '__qualn
ame__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', 'breast']
```

这里列出了所有 `cang` 这个对象的属性和方法，仔细观察，我们刚才用过的 `cang.__doc__` 和刚刚设置的 `cang.breast` 都历历在目。至于这里有很多属性的名字都是用双下划线开始和结束，这类属性可以称之为特殊属性（因为名字样式特殊吗？）。

```
>>> cang.__name__
'cang'
>>> cang.__module__
'__main__'
```

所有这些属性，都可以用句点的方式调用。

参数和变量

函数的参数，还是很有话题的。比如在别的程序员嘴里，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，如果你区别不开，也不会耽误你写代码，这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的面试官问这种问题。

我们就简化一下，笼统地把函数括号里面的变量叫做参数吧，当然你叫变量也无妨，只要大家知道值得是什么东西就好了。虽然这样会引起某些认真的人来喷口水，但也不用担心，反正本书已经声明是很“水”的了。

但如果有人较真，非要让你区分，为了显示你的水平，你可以引用[微软网站](#)上的说明。我认为这段说明高度抽象，而且意义涵盖深远的说明。摘抄过来，请读一读，是否理解。

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。

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词，更混乱了。请稍安勿躁，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。

函数(2)

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，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，请不吝赐教，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“传对象引用”——这又是一种貌似高深的说法。

```
>>> def foo(lst):
...     lst.append(99)
...     return lst
...
>>> x = [1, 3, 5]
>>> y = foo(x)
>>> y
[1, 3, 5, 99]
>>> x
[1, 3, 5, 99]
>>> id(x)
3075464588L
>>> id(y)
3075464588L
```

结合前面学习过的列表能够被原地修改知识，加上刚才说的参数特点，你是不是能理解上面的操作呢？

[总目录](#) | [上节：函数\(1\)](#) | [下节：函数\(3\)](#)

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

圣徒缺乏要帮补，客要一味的宽带。逼迫你们的，要给他们祝福。只要祝福，不可诅咒。与喜乐的人同乐，与哀哭的人要同哭。要彼此同心，不要志气高大，倒要俯就卑微的人。不要自以为聪明。不要以恶报恶，众人以为美的事，要留心去作。若是能行，总要尽力与众人和睦。(ROMANS 12:13-17)

函数(3)

对于函数的参数，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。如果读者了解量子力学——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

世界是不确定的，那么函数参数的个数，也当然有不确定的时候，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          #Python 3请自动修改为print()的格式，下同，从略。
    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被塞入一个元组里面，传给了 `arg`

函数(3)

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      #Python 3: print(args)
...
```

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'})
```

不管是什么，都一股脑地塞进了元组中。

```
>>> foo("python")
('python',)
```

即使只有一个值，也是用元组收集它。特别注意，在元组中，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                  #Python 3: print("x:"+str(x))
...     print "tuple:", args
...
>>> foo(7)
x: 7
tuple: ()
```

这时候 `*args` 收集到的是一个空的元组。

函数(3)

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      #Python 3:  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。

读者到这里可能想了，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。

函数(3)

```
>>> def foo(x,y,z,*args,**kargs):
...     print x      #Python 3用户请修改为print()格式，下同
...     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。

一种优雅的姿势

```
>>> def add(x, y):
...     return x + y
...
>>> add(2, 3)
5
```

这是通常的函数调用时的传值方法。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：

函数(3)

```
>>> 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 "{0} is writing {1}".format (author, name)      #Python 3: print("{0} is writing {1}".format (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。

融会贯通

Python中函数的参数通过赋值的方式来传对象引用。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。

函数(3)

```
>>> def foo(p1, p2, p3):
...     print "p1==>",p1          #Python 3用户修改为print()格式，下同
...     print "p2==>",p2
...     print "p3==>",p3
...
>>> foo("python", 1, ["qiwsir","github","io"])
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

函数(3)

```
>>> 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):
...     print args
...
>>> foo("qiwsir.github.io")
('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。

函数(3)

```
>>> def foo(**args):
...     print args
...
...
>>> foo(1,2,3)
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)
{'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'}
```

对函数的基本内容已经介绍完毕，但是，并不意味着结束，因为还有更深刻的东西没说呢，且看下节。

函数(3)

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

行事为人要端正，好像行在白昼。不可荒宴醉酒，不可好色邪荡，不可争竞嫉妒。总要披戴主耶稣基督，不要为肉体安排，去放纵私欲。

Let us live decently as in the daytime, not in carousing and drunkenness, not in sexual immorality and sensuality, not in discord and jealousy. Instead, put on the Lord Jesus Christ, and make no provision for the flesh to arouse its desires.(ROMANS 13:13-14)

函数(4)

再理解函数

如果把对函数的理解停留在此前的层面，还没有深入到函数的内涵，或者说只能做一些简单的事情，也可能是面临负责问题的时候不得不用冗长的代码解决。

所以，还要对函数进行深入探究。

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。

函数(4)

```
#!/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      #Python 3: 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。

函数(4)

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

"""
the better Fibonacci
"""

memo = {0:0, 1:1}

def fib(n):
    if not n in memo:
        memo[n] = fib(n-1) + fib(n-2)
    return memo[n]

if __name__ == "__main__":
    f = fib(10)
    print f          #Python: 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，而且后者的效率常常比递归要高。所以，我个人的建议是，对使用递归要考虑周密，不小心就永远运行下去了。

传递函数

前面已经多次提到函数也是对象。

对于函数的参数，我们也做了一些探究。通过参数，可以将数字、字符串、列表等等那些已经学习过的Python中默认类型的对象以引用的方式传入函数——也可以传入以后要学习过的自定义类型的对象引用。

阅读了上面两句话，你是否有一个疑惑？都是对象，函数对象的引用能不能作为参数传给函数呢？

看这样一个举例：

```
>>> def bar():
    print "I am in bar()"

>>> def foo(func):
    func()
```

这里定义了两个函数，`bar()` 就是我们熟悉的函数；而 `foo()` 则有些许变化，其参数要求是一个函数，否则函数体内的代码块无法执行 `func()`，因为这就是调用一个函数。

所以，要这样来调用 `foo()` 函数。

函数(4)

```
>>> foo(bar)
I am in bar()
```

下面的例子，是不是可以算一个小的应用呢？

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

def convert(func, seq):
    return [func(i) for i in seq]

if __name__ == "__main__":
    myseq = (111, 3.14, -9.21)
    r = convert(str, myseq)
    print r      #Python 3: print(r)
```

这个例子或者类似的，常常被作为“传递函数”的例子。在 `r = convert(str, myseq)` 里面，`str` 是实现字符串转化的函数 `str()` 的名字。

你当然也可以自己编写一个函数，替换 `str`。

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

def convert(func, seq):
    return [func(i) for i in seq]

def num(n):
    if n%2 == 0:
        return n**n
    else:
        return n*n

if __name__ == "__main__":
    myseq = (3, 4, 5)
    r = convert(num, myseq)
    print r      #Python 3: print(r)
```

在这个例子中，我写了一个 `num(n)` 函数，然后在 `r = convert(num, myseq)` 中使用这个函数的名字 `num`。其实跟前面的举例类似，只是为了让读者更深刻理解所谓“传函数”，使用的是函数名字，不是调用函数——调用函数使用 `num()` 的式样。

嵌套函数

函数不仅可以作为对象传递，还能在函数里面嵌套一个函数。例如：

函数(4)

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

def foo():
    def bar():
        print "bar() is running"      #Python 3用户请修改为print()函数，下同，从略
        print "foo() is running"

foo()      #调用函数
```

上面的代码中，在函数 `foo()` 里面定义了函数 `bar()`，这就是嵌套函数，而 `bar()` 则称为 `foo()` 的内嵌函数，因为它在 `foo()` 的里面定义的。

如果调用 `foo()` 函数，会得到如下结果：

```
foo() is running
```

这说明，在上面的调用方式和内嵌函数写法中，`bar()` 根本就没有被调用，或者说函数 `foo()` 并没有按照从上到下的顺序依次执行其里面的代码。

要想让 `bar()` 这个内嵌函数得到执行，就要在 `foo()` 函数里面显示地调用它，比如：

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

def foo():
    def bar():
        print "bar() is running"
    bar()          #显示调用内嵌函数
    print "foo() is running"

foo()
```

这样的运行结果就是：

```
bar() is running
foo() is running
```

如果我单独调用定义的内嵌函数，是不是可行呢？调用方式就是把上面代码中调用 `foo()`，修改为调用 `bar()`，然后运行，显示结果报错信息 `NameError: name 'bar' is not defined`。

显然这样调用是不行的。因为 `bar()` 函数是定义在 `foo()` 里面的函数，它生效的范围仅局限在 `foo()` 函数体之内，也就是它的作用域是 `foo()` 范围。既然如此，`bar()` 在使用变量的时候也会受到 `foo()` 的拘束了。

函数(4)

```
def foo():
    a = 1
    def bar():
        b = a + 1
        print "b=",b      #Python 3的用户请使用print()
    bar()
    print "a=",a

foo()
#output:
#b= 2
#a= 1
```

在函数 `bar()` 之外但在 `foo()` 之内定义了 `a = 1`，在 `bar()` 中能够被顺利调用。这个关系不难理解，可是如果遇到下面的，就迷茫了。

```
def foo():
    a = 1
    def bar():
        a = a + 1          #修改之处
        print "bar()a=",a
    bar()
    print "foo()a=",a

foo()
```

如果运行这段程序，是会报错的。重要的报错信息是 `UnboundLocalError: local variable 'a' referenced before assignment`。观察 `bar()` 里面，使用了变量 `a`，按照该表达式，Python 解析器认定该变量应是在 `bar()` 内部建立的，而不是引用的外部对象。所以就报错了。

在 Python 3 中，你可以使用 `nonlocal` 关键词，如下演示。

```
def foo():
    a = 1
    def bar():
        nonlocal a
        a = a + 1
        print("bar()a=",a)
    bar()
    print("foo()a=",a)

foo()
#output
#bar()a= 2
#foo()a= 2
```

以上说明了嵌套函数的原理，在编程实践中，怎么用呢？

函数(4)

```
def maker(n):
    def action(x):
        return x ** n
    return action
```

在 `maker()` 函数中，`return action` 返回的是 `action()` 函数对象。

```
f = maker(2)
print f
m = f(3)
print m
```

`f` 所引用的对象是一个函数对象—— `action()` 函数对象，`print f` 就是打印这个函数对象的信息。观察执行结果，对比上述代码，会有所感悟的。

```
<function action at 0x02A39970>
9
```

从这个角度看，嵌套函数，其实能够制作一个动态的函数对象—— `action`。这个话题延伸下去，就是所谓的“闭包”，关于“闭包”的问题，我会在后面跟读者聊一聊。

初识装饰器

至此，我们已经明确，函数——是对象——能够被传递，也能够嵌套。重复一个简单的举例，目的是抛砖引玉。

```
def foo(fun):
    def wrap():
        print "start"          #Python 3用户请自行更换为print()，下同，从略
        fun()
        print "end"
        print fun.__name__
    return wrap

def bar():
    print "I am in bar()"
```

`foo()` 的参数是一个函数，如果我们这样调用此函数：

```
f = foo(bar)
f()
#output:
#start
#I am in bar()
#end
#bar
```

函数(4)

这就是向 `foo()` 传递了函数对象 `bar` ——你已经熟悉的传递函数。对于这个问题，我们可以换一个写法——仅仅是换一个写法。

```
def foo(fun):
    def wrap():
        print "start"
        fun()
        print "end"
        print fun.__name__
    return wrap

@foo          #增加的内容
def bar():
    print "I am in bar()"
```

`@foo` 是一个看起来很奇怪的东西，人们常常把类似这种东西叫做语法糖。

语法糖（Syntactic sugar），也译为糖衣语法，是由英国计算机科学家彼得·兰丁发明的一个术语，指计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。通常来说使用语法糖能够增加程序的可读性，从而减少程序代码出错的机会。（源自《维基百科》）

如果用上面的方式，我们可以这样执行程序：

```
bar()
```

结果是：

```
start
I am in bar()
end
bar
```

以上，就是所谓的装饰器及其应用，`foo()` 是装饰器函数，使用 `@foo` 来装饰 `bar()` 函数。

装饰器本身是一个函数，将被装饰的类（后面会介绍这种东西）或者函数当作参数传递给装饰器函数，如上面所演示的那样。

关于装饰器，后面我们还会遇到。这里是刚刚认识，就如同跟人交往一样，初次见面，姑且简单了解，以后日久天长。

[总目录](#) | [上节：函数\(3\)](#) | [下节：函数\(5\)](#)

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

In the same way, husbands should love their wives as they do their own bodies. He who loves his wife loves himself. For no one ever hates his own body, but he nourishes and tenderly cares for it, just as Christ does for the church, because we are members of his body. "For this reason a man will leave his father and mother and be joined to his wife, and the two will become one flesh." This is a great mystery, and I am applying it to Christ and the church. Each of you, however, should love his wife as himself, and a wife should respect her husband. (EPHESIANS 6:28-33)

函数(5)

“闭包”是一个很酷的名词，不是吗？你听说过“烧包”、“豆包”、“脓包”等词语，“闭包”跟它们比起来，更有点神秘色彩。

什么是闭包

在数学上，有“闭包”，但此处讨论的是计算机高级语言中的“闭包”，维基百科上有这样的定义：

在计算机科学中，闭包（英语：Closure），又称词法闭包（Lexical Closure）或函数闭包（function closures），是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。所以，有另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。闭包在运行时可以有多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。

闭包的概念出现于60年代，最早实现闭包的程序语言是Scheme。之后，闭包被广泛使用于函数式编程语言如ML语言和LISP。很多命令式程序语言也开始支持闭包。

上面的定义是很严格的，也是比较难理解的。所以，要用简单的例子来说明。

毋庸置疑，下面这段程序是能够顺利运行的。

```
a = 3

def foo():
    print a      #Python 3: print(a)

foo()
```

a = 3 定义的变量在函数里面能够被调用，但是反过来，如下所示：

```
def foo():
    a = 3

    print a      #Python 3: print(a)
```

函数(5)

这段程序会毋庸置疑地报错了。其原因就可以用变量的作用域来解释了，详细见[命名空间](#)。

在函数 `foo()` 里面可以直接使用函数外面的 `a = 3`，但是在函数 `foo()` 外面不能使用它里面的所定义的 `a = 3`。根据作用域的关系，是合情合理的。然而，也许在某种特殊情况下，我们需要在函数外面使用函数里面的变量，怎么办？

```
def foo():
    a = 3
    def bar():
        return a
    return bar

f = foo()
print f()      #Python 3: print(f())
#output:
3
```

用上面的方式，就实现了在函数外面得到函数里面所定义的对象。这种写法的本质就是嵌套函数。

在函数 `foo()` 里面，有 `a = 3` 和另外一个函数 `bar()`，它们两个都在函数 `foo()` 的环境里面，但是，它们两个是互不统属的，所以变量 `a` 相对函数 `bar()` 是自由变量，并且在函数 `bar()` 中应用了这个自由变量——函数 `bar()` 就是我们所定义的闭包。

闭包是一个函数，并且这个函数具有以下特点：

- 定义在另外一个函数里面（嵌套函数）
- 引用其所在函数环境的自由变量

从上述代码的运行效果上看，通过闭包，能够在定义自由变量 `a = 3` 的环境 `foo()` 之外的地方得到该自由变量所引用的对象，或者说 `foo()` 执行完毕，但 `a = 3` 依然可以在 `f()` 即 `bar()` 函数中存在，而没有被收回。所以，`print f()` 才得到了其结果。

使用闭包

为什么要是用闭包？

如果不使用必要，也能编程，这是确认无疑的。

只不过，在某些时候，需要对事务做更高层次的抽象，这就可能用到闭包。

比如要写一个关于抛物线的函数。如不使用闭包，对于读者来讲应该能够轻易完成，现在使用闭包的方式，可以这么做。

函数(5)

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

def parabola(a, b, c):
    def para(x):
        return a*x**2 + b*x + c
    return para

p = parabola(2, 3, 4)
print p(5)          #Python 3: print(p(5))
```

在上面的函数中，`p = parabola(2, 3, 4)` 定义了一个抛物线的函数对象——状如 $y = 2x^2 + 3x + 4$ ，如果要计算 $x = 5$ 时，该抛物线函数的值，只需要 `p(5)` 即可。这种写法是不是让函数只用起来更简洁？

读者在学习了[类](#)的有关知识之后，再回来阅读这个闭包的应用，会认识到，此处以 `p = parabola(2, 3, 4)` 的形式，就如同类中创建实例一样。可以利用上面的函数创建多个实例，也就是得到多个不同的抛物线函数对象。

这就是闭包应用的典型案例之一。

另外，装饰器，本质上就是闭包的一种应用——可以再次阅读[装饰器](#)

当然，闭包在实践中还有其它方面的应用，作为入门教程，此处不做深究。读者如果有意愿，可以去Google有关内容，有不少大神在这方面撰写了文章。

[总目录](#) | [上节：函数\(4\)](#) | [下节：函数\(6\)](#)

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

Ever since the creation of the world his eternal power and divine nature, invisible though they are, have been understood and seen through the things he has made. So they are without excuse; for they knew God, they did not honor him as God or give thanks to him, but they became futile in their thinking, and their senseless minds were darkened. Claiming to be wise, they became fools; and they exchanged the glory of the immortal God for images resembling a mortal human being or birds or four-footed animals or reptiles. (ROMANS 1:20-23)

函数(6)

几个特殊函数

在python中，有几个特别的函数，它们常常被看做是Python能够进行所谓“函数式编程”的见证，虽然我认为Python不可能走上那条发展道路。

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。

函数(6)

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，也不一定麻烦，或者相差无几。

因此，在编程实践中，可以不用这些特殊函数，但本着艺不压身的想法，还是要介绍，并且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):
...     x += 3
...     return x
...
>>> numbers = range(10)
>>> numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> new_numbers = []
>>> for i in numbers:
...     new_numbers.append(add(i))
...
>>> 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)。

```
>>> 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)，这一行 lambda x:x+3 就完成 add(x) 函数体里面的两行。还可以写这样的例子：

函数(6)

```
>>> g = lambda x, y: x + y
>>> 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：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，要打印一列表，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]

>>> map(add, numbers)
[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)`。

函数(6)

`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，塞入到`func`那个函数里面，并将`func`的返回值依次存到一个列表中。

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：

- 对可迭代对象中的每个元素，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列表。
- 如果参数很多，则对那些参数并行执行`function`。

例如：

函数(6)

```
>>> 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]
```

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。

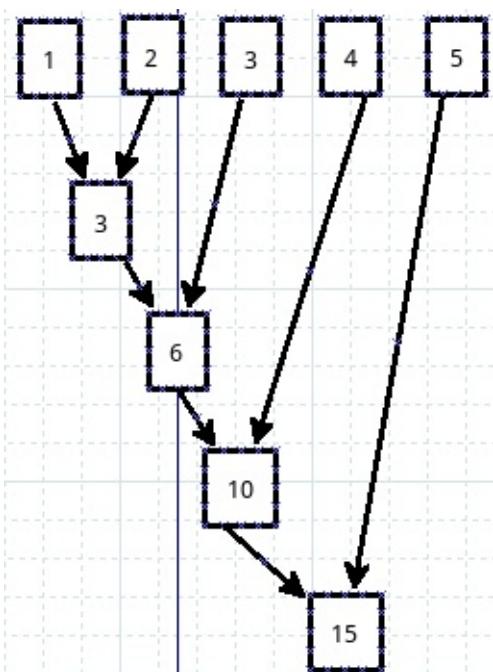
reduce

首先声明：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。

再看这个：

```
>>> reduce(lambda x,y: x+y,[1, 2, 3, 4, 5])
15
```

请仔细观察，是否能够看出是如何运算的呢？画一个图：



函数(6)

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]
```

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。

函数(6)

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]+...`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)]

>>> #这样也可以：new_tuple = (x*y for x,y in zip(a,b))，与上面的区别，后续会讲到
>>> 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
```

在Python 2中，如果使用map/reduce之类，可能或遇到性能不稳定的情况，如果是Python 3，就放心多了，不仅性能稳定，而且速度足够快。

但是，我还是更推荐使用列表解析，因为它可读性更好，你无法保证队友都跟你一样。

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 resu
lt 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。

不过，编程的时候，往往不能靠单纯自己的喜好，还得考虑队友。

[总目录](#) | [上节：函数\(5\)](#) | [下节：函数练习](#)

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

信心软弱的，你们要接纳，但不要辩论所疑惑的事。有人信百物都可吃，但那软弱的，只吃蔬菜。吃的人不可轻看不吃的人，不吃的人不可论断吃的人；因为神已经收纳他了。(ROMANS 14:1-3)

函数练习

仅仅知道函数的知识，是远远不够的，必须要勤练习、多敲代码，才能有体会，熟能生巧，熟能有感悟。

首先声明，一下对每个问题的解决方案，不一定是最优的。

读者在完成本节练习的时候，请遵守如下规则（全看你的自我控制能力了，自控能力强的胜出，不要欺骗哦，因为上帝在看着你呢）：

1. 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. 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（2000BC）古巴比伦的数学家就能解一元二次方程了。在大约公元前480年，中国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：

Python 2:

函数练习

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

"""
solving a quadratic equation
"""

from __future__ import division #Python2中为了确保除法得到的结果是精确的，导入python未来支持
                                的语言特征division(精确除法)
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."
```

Python 3:

```

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

"""
solving a quadratic equation
"""

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:{}".format(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

```

得到了方程的根。

但是，如果再认真思考，发现上述代码是有很多改进空间的：

- 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 `delta<0`，是不是写成复数更好？

读者是否还有其它改进呢？希望你能优化它，并分享你的成果。

至少要完成上述改进，可能需要其它的有关知识，甚至还没有介绍。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是一个开放的语言，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，就知道年级一共有多少人了。

现在我们就帮着老师来做成绩统计，这项工作，对学霸来讲是快乐的，因为可以提前享受成功的喜悦；对学渣来讲是痛并快乐着，因为晚痛不如早痛。

所以，要快乐地敲代码。

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。

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：

Python 2:

```
#!/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 = round(sum_scores/len(score_values), 2) # round(a,2) 保留两位小数
    return average
```

函数练习

```
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 "Xuezha is: ",xuezha          #学渣们
```

Python 3:

```
#!/usr/bin/env python
# coding=utf-8
"""
统计考试成绩
"""

def average_score(scores):
    """
    统计平均分.
    """
    score_values = scores.values()
    sum_scores = sum(score_values)
```

函数练习

```
average = round(sum_scores/len(score_values), 2) # round(a,2) 保留两位小数
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:{}" .format(ave))      #平均分

    sor = sorted_score(examine_scores)
    print("list of the scores:{}" .format(sor))        #成绩表

    xueba = max_score(examine_scores)
    print("Xueba is:{}" .format(xueba))                #学霸们

    xuezha = min_score(examine_scores)
    print("Xuzha is:{}" .format(xuezha))               #学渣们
```

保存为20502.py，然后运行：

```
$ python 20502.py
the average score is: 80.22
list of the scores: [('facebook', 99), ('apple', 99), ('amazon', 99), ('google', 98),
('alibaba', 80), ('android', 76), ('IBM', 70), ('baidu', 52), ('yahoo', 49)]
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。能不能优化一下？

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。

依照惯例，读者先做，然后我提供参考代码，最后思考如何优化。

质数（Prime number），又称素数，指在大于1的自然数中，除了1和该数自身外，无法被其他自然数整除的数（也可定义为只有1与该数本身两个因数的数）。

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，都可表示成两个素数之和。”。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, 101) if is_prime(i)]      #从2开始，因为1显然不是质数
    print(primes)  #Python 3: 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, 8
3, 89, 97]
```

打印出了100以内的质数。

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，还有好多中，读者可以自己创造或者网上搜索一些，拓展思路。

网友frankwang分享一段关于素数的代码，供各位参考：

```
def find_primes(n):
    primes_list = []
    for x in range(2, n+1):
        is_prime = True
        for y in range(2, int(x**0.5) + 1): #x**0.5 相当于math.sqrt(x)
            if x % y == 0:
                is_prime = False
                break
        if is_prime:
            primes_list.append(x)

    print(primes_list)      #Python 2: print primes_list

if __name__ == "__main__":
    max = int(input('Find primes up to: '))
    find_primes(max)
```

代码保存后运行，打印结果如下：

```
Find primes up to: 100
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 8
3, 89, 97]
```

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。

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

zip()补充

因为耶和华赐人智慧，知识和聪明都由他口而出。他给正直人存留真智慧，给行为纯正的人作盾牌，为要保守公平人的路，庇护虔敬人的道。你也必明白仁义、公平、正直，一切的善道。智慧必入你心，你的灵要以知识为美。谋略必护卫你，聪明必保守你，要救你脱离恶道（注：或作“恶人的道”），脱离说乖谬话的人。（箴言书 2:6-12）

zip()补充

在《语句（4）》中，对zip()进行了阐述，但是，由于篇幅限制，没有阐述的太完整。所以，本讲再次将它拿出来，希望能够有一个完成的独立阐述，以便学习者参考。

内建函数zip()

zip()是一个内建函数，对它的描述为：

zip(*iterables)

Make an iterator that aggregates elements from each of the iterables.

Returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.

zip()的参数是可迭代对象。例如：

```
>>> colors = ["red", "green", "blue"]
>>> values = [234, 12, 89, 65]
>>> for col, val in zip(colors, values):
...     print (col, val)      #Python 3: print((col, val))
...
('red', 234)
('green', 12)
('blue', 89)
```

注意观察，zip()自动进行了匹配，并且抛弃不对应的项。

参数 *iterables

这是zip() 的雕虫小技。

文档中显示zip() 的参数使*iterables，这是什么意思呢？

在《函数（3）》中，讲述“参数收集”和“另外一种传值”方法时，遇到过类似的参数，把其中一个例子再拿出来欣赏：

zip()补充

```
>>> def add(x,y):
...     return x + y
...
>>> add(2,3)
5
>>> bars = (2,3)
>>> add(*bars)
5
```

zip() 也类似上面示例中构造的那个 add() 函数。

```
>>> dots = [(1, 2), (3, 4), (5, 6)]
>>> x, y = zip(*dots)
>>> x
(1, 3, 5)
>>> y
(2, 4, 6)
```

利用这个功能，就比较容易实现矩阵的转置了。补充：关于矩阵的知识，可以参阅维基百科词条：[矩阵](#)

```
>>> m = [[1, 2],
...        [3, 4],
...        [5, 6]]
>>> zip(*m)      #Python 3: list(zip(*m))
[(1, 3, 5),
 (2, 4, 6)]
```

下面再看一个有点绚丽的：

```
>>> seq = range(1, 10)
>>> zip(*[iter(seq)]*3)      #Python 3: list(zip(*[iter(seq)]*3))
[(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

感觉有点太炫酷了，不是太好理解。其实，分解一下，就是：

```
>>> x = iter(range(1, 10))
>>> zip(x, x, x)      # Python 3: list(zip(x, x, x) )
[(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

这种炫酷的代码，我不提倡应用到编程实践中，这里仅仅是展示一下 zip() 的使用罢了。

关于在字典中使用 zip() 就不做过多介绍了，因为在《语句（4）》 中已经做了完整阐述。

更酷的示例

最后，展示一个来自网络的示例，或许在某些时候用一下，能够人前炫耀。

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [2, 2, 9, 0, 9]
>>> map(lambda pair: max(pair), zip(a, b))           #Python 3: list(map(lambda pair: max
(pair), zip(a, b)))
[2, 2, 9, 4, 9]
```

参考：

- <http://pavdmyt.com/python-zip-fu/>
- <https://bradmontgomery.net/blog/2013/04/01/pythons-zip-map-and-lambda/>

[总目录](#) | [上节：函数练习](#) | [下节：命名空间](#)

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

Let the word of Christ dwell in you richly in all wisdom; teaching and admonishing one another in psalms and hymns and spiritual songs, singing with grrace in your hearts tto the Lord. And whatsoever ye do in word or deed, do all in the name of the Lord Jesus, giving thanks to God and the Father by him. (COLOSSIANS 3:14-15)

命名空间

命名空间，英语叫做namespace，是很多编程语言中都会出现的术语。所以，有必要了解。

全局变量和局部变量

全局变量和局部变量，是理解命名空间的起始。

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 x=9，在函数的前面也有一个 x=2。

```
x = 2

def funcx():
    x = 9
    print "this x is in the funcx:-->", x          #Python 3请自动修改为print()，下同，从略

funcx()
print "-----"
print "this x is out of funcx:-->", x
```

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
```

从输出中可以看出，运行 funcx()，输出了 funcx() 里面的变量 x 引用的对象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。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，能够统率函数内外。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混乱。内外有别，在程序中一定要注意的。

局部变量和全局变量是在不同的范围内起作用。所谓不同范围，就是变量产生作用的区域，简称作用域。

作用域

所谓作用域，是“名字与实体的绑定保持有效的那部分计算机程序”（引自《维基百科》），用直白的方式说，就是程序中变量与对象存在关联的那段程序。如果用前面的例子说明，`x = 2` 和 `x = 9` 是处在两个不同的作用域中。

通常，把作用域还分为静态作用域和动态作用域两种，虽然 Python 是所谓的动态语言（不很严格的划分），但它的作用域属于静态作用域，意即 Python 中变量的作用域由它在程序中的位置决定，如同上面例子中的 `x = 9` 位于函数体内，它的作用域和 `x = 2` 就不同。

那么，Python 的作用域是怎么划分的呢？可以划分为四个层级：

1. Local：局部作用域，或称本地作用域
2. Enclosing：嵌套作用域
3. Global：全局作用域
4. Built-in：内建作用域

对于一个变量，Python 也是按照上述从前到后的顺序，在不同作用域中查找。在刚才的例子中，对于 `x`，首先搜索的是函数体内的本地作用域，然后是函数体外的全局作用域。

```

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

def outer_foo():
    a = 10
    def inner_foo():
        a = 20
        print "inner_foo,a=", a      #a=20
                                    #Python 3的读者，请自行修改为print()

rint()函数形式，下同，从略
    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

```

仔细观察上述程序和运行结果，你会看出对变量在不同范围进行搜索的规律的。

在Python程序中，变量的作用域是有在函数、类中才能被改变，或者说，如果不是在函数或者类中，比如在循环或者条件语句中，变量都是在同一层级的作用域中。可以再次参考上述的示例，并且可以在上述示例中修改，检测你的理解。

命名空间

“命名空间是对作用域的一种特殊的抽象”（引自《维基百科》）。下面就继续理解这种抽象。

先看来自《维基百科》的定义，这个定义通俗易懂。

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）提供一隱藏區域標識符的機制。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視作隱式命名空间，它们和可見性、可訪問性和对象生命周期不可分割的联系在一起。

在这段定义中，已经非常清晰地描述了命名空间的含义，特别是如果你已经理解了作用域之后，对命名空间就没有什么陌生感了。

为了凸显命名空间之对Python程序员的重要价值，请在交互模式下，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!
```

这就是所谓《python之禅》,请看最后一句: Namespaces are one honking great idea -- let's do more of those!

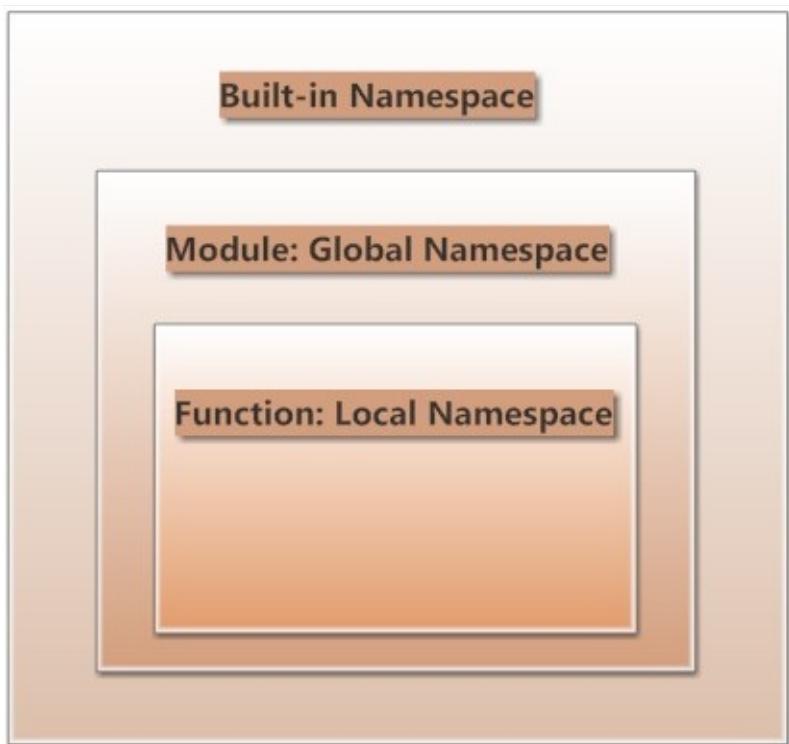
简而言之,命名空间是从所定义的命名到对象的映射集合。

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:

1. 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。
2. 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。
3. 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。

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字典是一样的。

根据习惯，读者一定已经猜测到了，如果访问全局命名空间，可以使用 `print globals()`。

对于不同的命名空间，除了存在查找的顺序之外，还有不同的生命周期，即什么时候它存在，什么时候它消失了。对此，在理解上比较简单，那就是哪个部分被读入内存，它相应的命名空间就存在了。比如程序启动，内置命名空间就创建，一直到程序结束；而其它的，比如本地命名空间，就是在函数调用时开始创建，函数执行结束或者抛出异常时结束。

关于命名空间，读者还需要在日后的开发实践中慢慢体会，它会融会到你的编程过程中，有时候你是觉察不到的，正所谓“随风潜入夜，润物细无声”。

[总目录](#) | [上节：zip\(\)补充](#) | [下节：类\(1\)](#)

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

你这个人，为什么论断弟兄呢？又为什么轻看弟兄呢？因我们都要站在神的台前。

所以我们不可再彼此论断，宁可定意，谁也不给弟兄下绊脚跌人之物。(ROMANS 14:10,13)

类(1)

类，这个词如果你第一次听到，把它作为一个单独的名词，会感觉怪怪的，因为在汉语中，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术语有了解。

“行百里路者半九十”，如果读者坚持阅读到本书的这个章节，已经对Python有了初步感受，而“类”就是能够让你在Python学习进程中再上台阶的标志。所以，一定要硬着头皮耐心地继续学下去。

术语

所谓“术语”，可以粗浅地理解为某个领域的“行话”，比如在物理学里面，有专门定义的“质量”、“位移”、“速度”等，这些术语有的跟日常生活中的俗称名字貌似一样，但是所指有所不同。

“术语”的主要特征是具有一定的稳定性，并且严谨、简明，不是流行语。在谈到OOP的时候，会遇到一些术语，需要先明确它们的含义。

本节没有特别声明的术语定义均来自《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——一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。

我们不去追究翻译的名称，因为这是专家们的事情。我们需要知道的是“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。

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。

于是就需要“述而不作”的我了，我的任务就是就把大师的话化简，虽然化简了之后可能在严谨性上就不足了，但对于初学者来讲，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。那时候应当抛弃本书，去阅读大师的作品。

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，方法更常被使用）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。

为了体现“深入浅出”的道理，还是讲故事吧。

既然万物都是对象，那么，某个具体的人也是对象，这是当然的事情。假设这个具体的人就是德艺双馨的苍老师，她是一个对象。这个苍老师具有哪些特征呢？我错了，写到这里发现不能用苍老师为对象的例子，因为容易让读者不专心学习了。我换一个吧，选定的对象就是某个王美女（这个王美女完全是虚构的，请不要对号入座，更不要想入非非，如果雷同，纯属巧合，想入非非的后果自行担负）。

王美女这个对象具有某些特征，眼睛，大；腿，长；皮肤，白。当然，既然是美女，肯定还有别的明显特征，读者可以自己假设去。如果用“对象”的术语来说明，就说这些特征都是她的属性。也就是说属性是一个对象所具有的特征，或曰：是什么。。

类(1)

王美女除了具有上面的特征之外，她还能做一些事情，比如她能唱歌、会吹拉弹唱等。这些都是她能够做的事情。用“对象”的术语来说，就是她的“方法”。即方法就是对象能够做什么。

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。这一系统大概是最先的基于容量架构（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。

至此，如果读者把前面的文字逐句读过，没有跳跃，则姑且认为你已经对“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：

类(1)

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，读者或许有一个认识，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
```

从这里开始编写一个类，不过这次我们暂时不用Python，而是用貌似伪代码，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，并约定，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。

类(1)

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()
```

用这样一种表达方式，就是将“美女类”实例化了，或者说创建了一个实例“王美女”。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，得到的值就是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，这个王美女（实例）的皮肤（属性）就是黑色（值）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？如果没有理解，请用苍老师实例化美女类，你一定能理解的。

当然，对类的认识不能仅仅停留在此，还要真刀真枪地写代码。

[总目录](#) | [上节：函数练习](#) | [下节：类\(2\)](#)

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

弟兄们，那些离间你们，叫你们跌倒，背乎所学之道的人，我劝你们要留意躲避他们。因为这样的人不服侍我们的主基督，只服侍自己的肚腹，用花言巧语诱惑那些老实人的心。(ROMANS 16:17-18)

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

Python是一个不断发展的高级语言（似乎别的语言也是不断发展的，甚至于自然语言也是），导致了Python 2和Python 3两个版本。

就是在Python 2中，还有“新式类”和“旧式类（也叫做经典类）”之分。这可真够分裂的。

新式类是Python 2.2引进的，在此后的版本中，我们一般用的都是新式类。

但是在Python 3中没有这种新旧的问题，它只是“类”。所以，如果你使用的是Python 3，并且也没有兴趣了解历史问题，可以跳过本节内容。不过，如果这样，如果有一天遇到老代码，你会后悔的。

书归正传，转到Python 2中，先写一个极简的旧式类。

```
>>> class AA:  
...     pass
```

这就是一个旧式类。关于定义类的方法，下面会详细说明。读者姑且囫囵吞枣，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（它的另外一个名字是“经典类”，“旧”的时间长了就变成“经典”了）。

然后，将这个类实例化或者说建立一个实例（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，这跟调用函数的方法是一样的。

接下来做如下操作：

```
>>> type(AA)  
<type 'classobj'>
```

类(2)

AA() 是调用类，但是 AA 指的就是那个类对象。

这一点非常类似于函数，比如函数 foo() ，而 foo 就是那个函数对象。在这里，“一切皆对象”应该再次浮现在你的脑海里。

type(AA) 返回的是这个类对象的属性—— classobj —— AA 是一个类对象。

```
>>> aa.__class__  
<class '__main__.AA' at 0xb71f017c>
```

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。

```
>>> type(aa)  
<type 'instance'>
```

解读一下上面含义：

type(aa) 是要看实例 aa 的类型，显示的结果是 instance ，是告诉我们它是一个实例。

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。

看看我们已经熟悉的一个对象。

```
>>> a = 7  
>>> a.__class__  
<type 'int'>  
>>> type(a)  
<type 'int'>
```

对于整数7，毫无疑问，它是对象。用两种方式查看类型，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'>
```

类(2)

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。

“喜新厌旧”是编程界的传统。所以，旧式类就不是我们讨论的内容了。

在本书此后的内容中，所有Python 2代码中的类，都是新式类。

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python 3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。但在本书中，如果在Python 2里面定义新式类，都会采用第一种定义方法。

创建类

前面我们创建了很简单的类，虽然没有什么太大的用途，但也是创建了类。为了更一般化地说明如何创建类，下面这个类，更具有通常类的结构。

Python2：

类(2)

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

class Person(object):
    """
    This is a sample of class.
    """

    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def color(self, color):
        d = {}
        d[self.name] = color
        return d
```

Python 3：

```
class Person:
    """
    This is a sample of class.
    """

    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def color(self, color):
        d = {}
        d[self.name] = color
        return d
```

这是一个具有“大众脸”的类，下面对它进行逐条解释。

Python 2的代码中，使用了 `class Person(object)`，代表着是新式类。而Python 3的代码中，不需要显示地继承`object`了。

`class Person`，这是在声明创建一个名为"Person"的类，其关键词是 `class` ——对照着学习，创建函数的关键词是 `def` ——新旧知识类比，更容易理解。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`。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。对于Python 2，因为是要继承`object`，所以要在后面有一

类(2)

个类似函数的参数列表那样的样式 (object) , Python 3则不需要,但是,不论是Python 2还是Python 3,要继承其它父类的时候,都要在后面写上 (father_class_name) ——当然,继承的问题是后文了。这一切结束之后,本行的最后就是冒号 : 。

声明了类的名字(或者也包括继承的父类),然后就是类里面的代码块。秉承函数的做法,类里面的代码块,相对类定义类的那一行,也是要缩进四个空格——四个空格,尽可能不适用tab键,老老实实敲四个空格,否则后患无穷,绝非危言耸听。

再看类里面的代码,那些东西看起来并不陌生,你一眼就认出它们了——都是 def 这个关键词开头——就是已经学习过的函数。没错,它们就是函数。不过,仔细观察,会发现这些函数有点跟以往的函数不同,它们的参数中,都有 self 。这点差别,也是类中这种函数的特色,为了跟以往的函数有所区别,所以很多程序员喜欢把类里面的函数叫做“方法”——暂且不要纠结在名称上,虽然我看到过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。

在上述代码示例中,函数(方法)的参数必须包括 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。

下面对类里面的每个方法(函数)做一个简要的阐述。

def __init__(self, name) ,这个方法是比较特殊的。当从命名方式上看,就不一般——用双下划线开头和结尾——这样的方法,在类里面还有很多,我们统称为“特殊方法”。对于 __init__() 这个特殊方法,通常还给它取了一个名字——构造函数——这是通常的叫法,但是我觉得这个名字不好,在本书中我把它叫做做初始化函数,因为从字面意义上说,它对应的含义是初始化,并且在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,Python中的真正构造函数是 __new__() 。

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面有水。在 Python的类中,初始化就担负着类似的工作。在类实例化时首先就执行初始化函数。

此例子中的初始化函数的参数除了 self ,还有一个 name ,在这个类被实例化的同时,要传给它一个值。

在初始化函数里面, self.name = name 的含义是要建立实例的一个属性,这个属性的名字也是 name ,这个属性的值等于参数 name 所传入的值。特别注意,这里的属性 self.name 和参数 name 是纯属巧合,你也可以设置成 self.xxx = name ,只不过这样写,总感觉不是很方便。

def get_name(self) 和 def color(self, color) 是类里面的另外两个方法,这两个方法的除了第一个参数必须是 self 之外,别的跟函数就没有什么区别了。只是需要关注的是两个方法中都用到了 self.name ,这个属性只能在类里面这样使用。

类(2)

以上就将我们所建立的类进行了简要的分析。这也是建立一个类的基本方法。

实例

类是对象的定义，实例才是真实的物件。比如“人”是一个类，但是“人”终究不是具体的某个活体，只有“张三”、“李四”才是具体的物件，但他们具有“人”所定义的属性和方法。“张三”、“李四”就是“人”的实例。

承接前面的类，先写出调用该类的代码。

Python 2:

```
if __name__ == "__main__":
    girl = Person("canglaoshi")
    print girl.name
    name = girl.get_name()
    print name
    her_color = girl.color("white")
    print her_color
```

Python 3:

```
if __name__ == "__main__":
    girl = Person("canglaoshi")
    print(girl.name)
    name = girl.get_name()
    print(name)
    her_color = girl.color("white")
    print(her_color)
```

`girl = Person('canglaoshi')` 是利用上面的类创建实例。

创建实例，调用类 `Person()`，首先就执行初始化函数，初始化函数有两个参数 `self` 和 `name`，其中 `self` 是默认参数，不需要传值；`name` 则需要给它传值，所以用 `Person('canglaoshi')` 的样式，就是为初始化函数中的 `name` 参数传值了，即 `name = 'canglaoshi'`。

`girl` 就是一个实例，它有属性和方法。

先说属性。实例化过程中，首先要执行 `__init__()`，并通过参数 `name`，使得实例属性 `self.name = 'canglaoshi'`。这里先稍微提一下 `self` 的作用，它实质上就是实例对象本身，当你用实例调用方法的时候，由解释器将那个实例传递给方法，所以不需要显示地为这个参数传值。那么 `self.name` 也顺理成章地是实例的属性了。所以 `print girl.name` 或者 `print(girl.name)` 的结果应该是 `canglaoshi`。

这就是初始化的功能。简而言之，通过初始化函数，确定了这个实例的“基本属性”（实例是什么样子的）。

类(2)

`girl.get_name()` 是通过实例来调用方法，也可以说建立了实例 `girl`，这个实例就具有了 `get_name()` 方法。虽然在类里面，该方法的第一个参数是 `self`，跟前面所述原因一样，通过实例调用该方法——实例方法——的时候，不需要显示地为 `self` 传递值，所以，在这里就不需要写任何参数。观察类中这个方法的代码可知，它的功能就是返回实例属性 `self.name` 的值，所以 `print name` 或者 `print(name)` 的结果是 `canglaoshi`。

`girl.color("white")` 之所以要给参数传值，是因为 `def color(self, color)` 中有参数 `color`。另外，这个方法里面也使用了 `self.name` 实例属性。最终该方法返回的是一个字典。所以 `print her_color` 或者 `print(her_color)` 的结果是 `{'canglaoshi': 'white'}`。

刚才以 `girl = Person("canglaoshi")` 的方式，建立了一个实例，仿照它，还可以建立更多的实例，比如 `boy = Person("zhongsan")` 等等。也就是一个类，可以建立多个实例。所以“类提供默认行为，是实例的工厂”（源自Learning Python），这句话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。

这就是通过类建立实例，并且通过实例来调用其属性和方法的过程。

[总目录](#) | [上节：类\(1\)](#) | [下节：类\(3\)](#)

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

世人凭自己的智慧，既不认识神，神就乐意用人所当作愚拙的道理拯救那些信的人，这就是神的智慧了。犹太人是要神迹，希腊人是求智慧；我们却是传钉十字架的基督。(1 CORINTHIANS 1:21-22)

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

在交互模式下，创建一个简单的类。

```
>>> class A(object):      #Python 3: class A:  
...           x = 7  
...
```

这个类中的代码，没有任何方法，只有一个 `x = 7`，当然，如果愿意还可以写别的。

先不用管为什么，继续在交互模式中敲代码。

```
>>> A.x  
7
```

`A` 是刚刚建立的类的名字，`x` 是那个类中的一个变量，它引用的对象是整数 `7`。通过 `A.x` 的方式，就能得到整数 `7`。像这样的，类中的 `x` 被称为类的属性，而 `7` 是这个属性的值。`A.x` 是调用类属性的方式。

这里谈到了“属性”，不要忽视这个词语，它是用在很多领域的一个术语，比如哲学有一些哲学家认为，属性所描述的是种类性质（例如颜色），属性的性质就是所谓的值（比如红色）。

哲学真的是“科学之母”呀，把上面的理解套用过来，就是我们现在讨论的类属性。`x` 是类 `A` 的性质，它的值是 `7`。如果还抽象，我就不得不寄出大招了。

```
>>> class Girl(object):      #Python 3: class Girl:  
...           breast = 90
```

在真实世界中，`breast` 就是 `Girl` 的属性，你到某度上去搜索一下有关的关键词，就发现 `breast` 是一个重要属性了。所以，如果要建立类 `Girl`，它必须有 `breast` 属性。那么这个属性的值，就关系到某类 `Girl` 的颜值了。这里的类 `Girl` 都是 `breast` 为 `90` 的，你可以想象其颜值高低了，是否符合你的喜好。

大招一出，为之一振，顿时困意消退。下面就请回到前面那个类 `A`，继续枯燥地学习。

类(3)

如果要调用类的某个属性——简称：类属性——其方法就是用半角的英文句号 . ，如前面所演示的 A.x 。类属性仅与其所定义的类绑定，并且这种属性，本质上说就是类中的变量，它的值不依赖于任何实例，只是由类中所写的变量赋值语句确定，比如 Girl 类，不管你用这个类建立的实例是东施还是西施，类属性 Girl.breast 都是90。所以，这个类属性还有别的名字，如静态变量或者静态数据。

在本书中，已经多次提到“万物皆对象”的观念，类也不例外，它也是对象。只有对象，才具有属性和方法。而属性是可以修改或者增加、删除的。既然如此，对刚才的类 A 或者类 Girl ，你都可以对目前其有的属性进行修改，也可以增加新的属性。

```
>>> A.y = 9  
>>> A.y  
9
```

对类 A 增加了一个新的属性，并且赋给了值。然后删除一个已有属性。

```
>>> del A.x  
>>> A.x  
Traceback (most recent call last):  
  File "<pyshell#14>", line 1, in <module>  
    A.x  
AttributeError: type object 'A' has no attribute 'x'
```

A.x 属性删除之后，再调用，就出现异常了。但是 A.y 依然存在。

```
>>> A.y  
9
```

你也可以修改当前已有的类属性。

```
>>> Girl.breast = 40  
>>> Girl.breast  
40
```

breast 是我们在类 Gril 中自己定义的属性，其实在一个类建立的同时，Python也让这个类具有了一些默认的属性。可以用我们熟知的 dir() 来查看类的所有属性（也包括方法）。

```
>>> dir(Girl)  
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',  
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',  
 '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',  
 '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'breast']
```

仔细观察，上面有一个特殊的属性 __dict__ ，用“特殊”来修饰这个属性，是因为它也是以双下划线开头和结尾，就类似你已经见过的一个特殊方法 __init__() 一样。在类里面，凡以双下划线开头和结尾命名的属性或者方法，我们都会说它是“特殊”的。

类(3)

对于Python 2和Python 3，上述显示结果会略有不同，其差别主要在于特殊属性和方法上。所以读者在自己的版本中操作的时候，不要执拗于上述结果。但是，自定义的属性 `breast` 是肯定都存在的。

```
>>> Girl.__dict__
mappingproxy({'breast': 40, '__weakref__': <attribute '__weakref__' of 'Girl' objects>,
  '__dict__': <attribute '__dict__' of 'Girl' objects>, '__module__': '__main__', '__doc__':
None})
```

对于不同版本的Python，上述显示结果也有所差异，但你都能看到属性的详细信息。类的特殊属性 `__dict__` 所显示的是这个类的属性名称以及该属性的完整数据。

下面列出类的几种特殊属性的含义，读者可以一一查看。

- `c.__name__`：以字符串的形式，返回类的名字，注意这时候得到的仅仅是一个字符串，它不是一个类对象
- `c.__doc__`：显示类的文档
- `c.__base__`：类C的所有父类。如果是按照上面方式定义的类，应该显示 `object`，因为以上所有类都继承了它。等到学习了“继承”，再来看这个属性，内容就丰富了
- `c.__dict__`：以字典形式显示类的所有属性
- `c.__module__`：类所在的模块

稍微解释 `c.__module__`。承接前面建立的类 `Girl`，做如下操作：

```
>>> Girl.__module__
'__main__'
```

说明这个类所述的模块是 `__main__`，换个角度，类 `Girl` 的全称是 `__main__.Girl`。还记得我们曾经用过标准库中的`math`吗？它是一个模块，它下面的内容都是用类似 `math.pi` 的样式读取。两者是同样道理，只不过模块名称变化了。

```
>>> from math import sin
>>> sin.__module__
'math'
```

这里的 `sin` 全称就是 `math.sin`，所以你也可以 `import math`，然后使用 `math.sin`。以上两者是一样的道理。

最后对类属性进行总结：

1. 类属性跟类绑定，可以自定义、删除、修改值，也可以随时增加类属性
2. 类属性不因为实例变化而发生变化
3. 每个类都有一些特殊属性，通常情况下特殊属性是不需要修改的，虽然有的特殊属性可以修改，比如 `c.__doc__`

类(3)

对于类，除了属性，还有方法。但是类中的方法，因为牵扯到实例，所以，我们还是通过研究实例，理解类中的方法

创建实例

创建实例并不是困难的事情，只需要通过调用类就能实现。

```
>>> canglaoshi = Girl()
>>> canglaoshi
<__main__.Girl object at 0x0000000003726C18>
```

这就创建了一个实例 `canglaoshi`。

请读者注意，调用类的方法和调用函数的方法类似。如果仅仅写 `Girl()` 也是创建了一个实例，如下所示：

```
>>> Girl()
<__main__.Girl object at 0x00000000035262E8>
```

而 `canglaoshi = Girl()` 本质上就是将变量 `canglaoshi` 与实例对象 `Girl()` 建立引用关系，这种关系同以前见过的赋值语句 `a = 2` 是同样的效果。

那么，一个实例的建立过程是怎样进行的呢？

再次启用上节中写的类。

Python2：

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

class Person(object):
    """
    This is a sample of class.
    """

    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def color(self, color):
        d = {}
        d[self.name] = color
        return d
```

Python 3：

类(3)

```
class Person:  
    """  
    This is a sample of class.  
    """  
  
    def __init__(self, name):  
        self.name = name  
  
    def get_name(self):  
        return self.name  
  
    def color(self, color):  
        d = {}  
        d[self.name] = color  
        return d
```

实例还是用 `girl = Person('canglaoshi')`，当然，你建立别的实例也是可以的。利用一个类，可以建立无限多个实例，所以有一种说法，类是实例的工厂。

创建实例，就是调用类。当类被调用之后：

1. 创建实例对象；
2. 检查是否有——专业的说法：是否实现—— `__init__()` 方法。如果没有，则返回实例对象；
3. 如果有 `__init__()`，则调用该方法，并且将实例对象作为第一个参数 `self` 传递进去

`__init__()` 作为一个特殊方法，是比较特殊的，在它里面，一般是规定一些属性或者做一些初始化要让类具有的一些特征，但是，它没有 `return` 语句。观察别的方法都是可以有这个的。

```
class Foo:  
    def __init__(self):  
        print("I am in init())")  
        return 1  
  
bar = Foo()
```

假如写了上面的代码，在初始化函数中有 `return` 语句，又会如何？

```
I am in init()  
Traceback (most recent call last):  
File "F:/MyGitHub/StarterLearningPython/2code/20801p3.py", line 7, in <module>  
    bar = Foo()  
TypeError: __init__() should return None, not 'int'
```

这就是运行结果，出现了异常，并且明确告知“`init()` should return None”，所以不能有 `return`，如果非要有，也得是 `return None`，索性就不要写了。

类(3)

由此可知，对于 `__init__()` 初始化函数，除了第一个参数是并且必须有 `self`、不能有 `return` 语句这两个特征之外，其它方面和普通函数就没有什么异样了。比如参数和里面的属性，你可以这样来做：

```
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"
```

实例创建好之后，就要研究关于实例的内容，首先看实例属性

实例属性

与类属性类似，实例所具有的属性叫做“实例属性”。

还是用那个简单的类，虽然有点枯燥。

```
>>> class A(object):      #Python 3: class A:  
...     x = 7  
...  
>>> foo = A()
```

类已经有一个属性 `A.x = 7`，那么由这个类所建立的实例也具有这个属性。

```
>>> foo.x  
7
```

除了 `foo.x` 这个属性之外，实例也具有其它的属性和方法，依然能使用 `dir()` 方法来查看。

```
>>> dir(foo)  
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',  
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',  
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',  
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'x']
```

不同Python版本显示结果稍微有差异，但是那个名为 `x` 的属性是都有的，这点上还是跟类属性类似的。并且实例也有一些特殊属性，比如：

```
>>> foo.__dict__  
mappingproxy({'__dict__': <attribute '__dict__' of 'A' objects>, '__doc__': None, '__weakref__': <attribute '__weakref__' of 'A' objects>, '__module__': '__main__', 'x': 7})
```

类(3)

实例属性和类属性的一个最大不同，在于实例属性可以随意更改，不用有什么担心（前面我们建议，尽可能不要修改类属性）。

```
>>> foo.x += 1  
>>> foo.x  
8
```

这是把实例属性修改了。但是，类属性并没有因为实例属性修改而变化，正如前文所讲，类属性是跟类绑定，不受实例影响。

```
>>> A.x  
7
```

上述结果说明“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，旧的不就显现出来了？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`。其实，在这里你也完全可以依照变量和对象的关系来理解上述实例属性和数值（对象）的关系。

但是，类属性能够影响实例属性，这是因为实例就是通过调用类来建立的。

```
>>> A.x += 1  
>>> A.x  
8  
>>> foo.x  
8
```

如果是同一个属性`x`，实例属性跟着类属性而改变。

类(3)

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。

如果增加一个类属性，相应的也增加了一个实例属性，

```
>>> A.y = "hello"
>>> foo.y
'hello'
```

反过来，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。

类(3)

以上所显示的实例属性或者类属性，都源自于类中的变量所引用的值，或者说是静态数据，尽管能够通过类或者实例增加新的属性，其值也是静态的。

还有一类实例属性的生成方法，就是在实例创建的时候，通过 `__init__()` 初始化函数建立，这种建立则是动态的。

self 的作用

类里面的任何方法，第一个参数是 `self`，但是在创建实例的时候，似乎没有这个参数什么事儿（不显示地写出来），那么 `self` 是干什么的呢？

`self` 是一个很神奇的参数。

将前文的 `Person` 类简化一下，

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

class Person(object):      #Python 3: class Person:
    def __init__(self, name):
        self.name = name
        print self           #Python 3: print(self)
        print type(self)     #Python 3: print(type(self))
```

其它部分省略。

当创建实例时候，首先要执行构造函数，同时就打印新增的两条。结果是：

```
>>> girl = Person("canglaoshi")
<__main__.Person object at 0x0000000003146C50>
<class '__main__.Person'>
```

这说明 `self` 就是类 `Person` 的实例，再看看刚刚建立的那个实例 `girl`。

```
>>> girl
<__main__.Person object at 0x0000000003146C50>
>>> type(girl)
<class '__main__.Person'>
```

`self` 和 `girl` 所引用的实例对象一样。

当创建实例的时候，实例变量作为第一个参数，被 Python 解释器悄悄地传给了 `self`，所以我们说在初始化函数中的 `self.name` 就是实例的属性。

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

类(3)

是可以的。

```
>>> girl.name  
'canglaoshi'
```

这是我们得到的实例属性，但是，在类的外面不能这样用：

```
>>> self.name  
Traceback (most recent call last):  
  File "<pyshell#23>", line 1, in <module>  
    self.name  
NameError: name 'self' is not defined
```

数据流转

将类实例化，通过实例来执行各种方法，应用实例的属性，是最常见的操作。

所以，对此过程中的数据流转一定要弄明白。

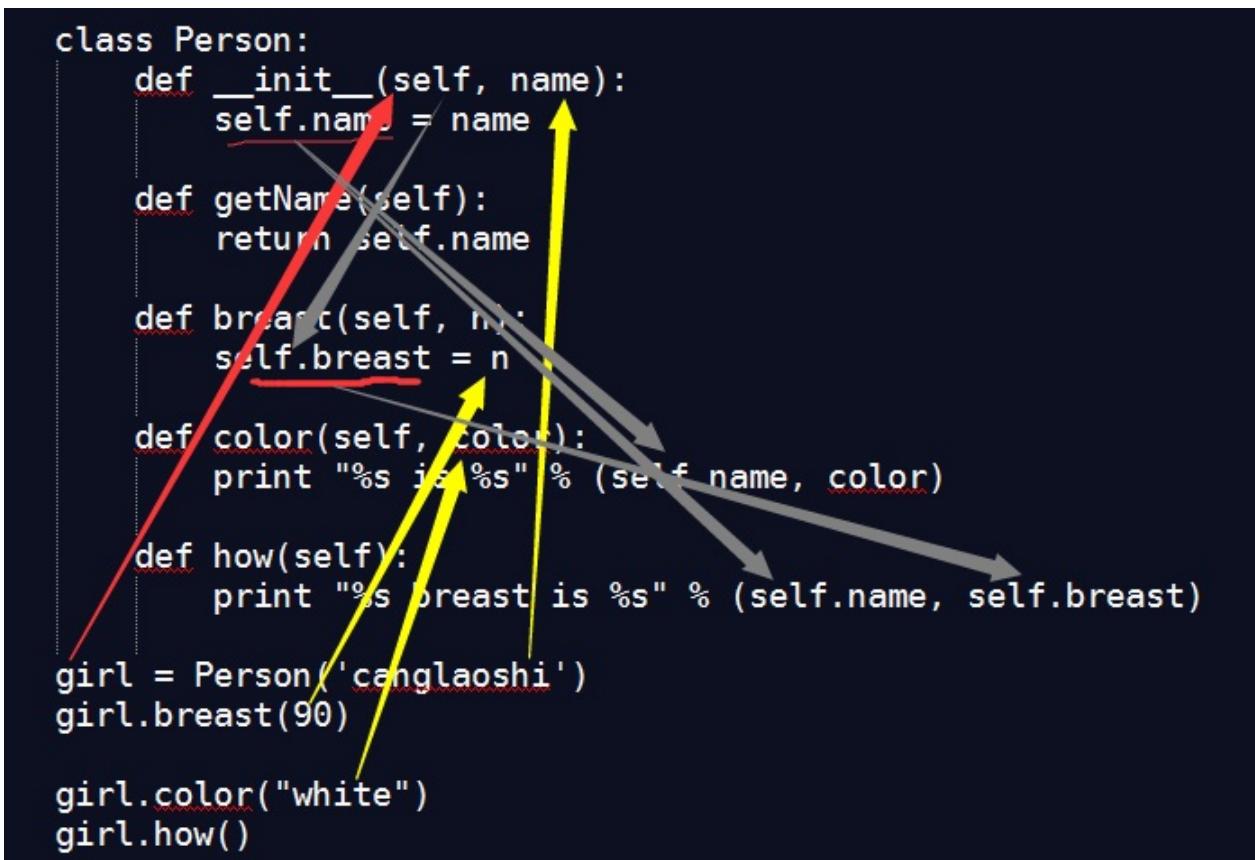
把前文的类再稍微修改，如下：

```
#!/usr/bin/env python  
# coding=utf-8  
  
class Person(object):      #Python 3: 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)          #Python 3: print("{0} is {1}".f  
        ormat(self.name, color))  
  
    def how(self):  
        print "%s breast is %s" % (self.name, self.breast)      #Python 3: print("{0}  
} breast is {1}".format(self.name, self.breast))  
  
girl = Person('canglaoshi')  
girl.breast(90)  
  
girl.color("white")  
girl.how()
```

运行后结果：

```
$ python 20701.py
canglaoshi is white
canglaoshi 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。



创建实例 `girl = Person('canglaoshi')`，注意观察图上的箭头方向。`girl` 这个引用实例对象的变量传给了 `self`，即 `self` 也引用了实例对象。简化理解为：`self` 是实例（不求准确，只求表面现象），实例变量主外，`self` 主内。

"canglaoshi" 是一个具体的数据，通过初始化函数中的 `name` 参数，传给 `self.name` ——准确地说谁传了对象引用给实例的属性 `name`。前面已经讲过，`self` 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 "canglaoshi"。

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，这个参数要明显写上，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。

因为引用实例对象的变量 `girl` 和 `self`，所以，在类里面也可以用 `girl` 代替 `self`。例如，做如下修改：

类(3)

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

class Person(object):    #Python 3: 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('canglaoshi')
name = girl.getName()
print name
```

运行之后，打印：

```
canglaoshi
```

这个例子说明，在实例化之后，`girl` 和 `self` 都引用了实例对象。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。

[总目录](#) | [上节：类\(2\)](#) | [下节：类\(4\)](#)

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(4)

爱人不可虚假，恶要厌恶，善要亲近。爱弟兄，要彼此亲热；恭敬人，要彼此推让。殷勤不可懒惰。要心里火热，常常服侍主。在指望中要有喜乐，在患难中要有忍耐，祷告要恒切。圣徒缺乏要帮补，客要一味地款待。逼迫你们的，要给他们祝福，只要祝福，不可诅咒。与喜乐的人要同乐，与哀哭的人要同哭。要彼此同心，不要志气高大，倒要俯就卑微的人。不要自以为聪明。不要以恶报恶。众人以为美的事，要留心去作。

(ROMANS 12:9-17)

类(4)

方法

在类里面，除了属性，就是方法，当然还有注释和文档，但计算机不看它们的，只是人看的。

关于方法，在通常情况下用实例调用。但是，跟方法有关的一些深入的话题，还需要辨析。

绑定方法和非绑定方法

除了特殊方法，类中的其它的普通方法，是经常要用到的，所以，要对这些普通方法进行研究。

```
>>> class Foo:          #Python 2: class Foo(object):
    def bar(self):
        print("This is a normal method of class.")      #Python 2 使用print 语句

>>> f = Foo()
>>> f.bar()
This is a normal method of class.
```

在类 `Foo` 中，方法 `bar()` 本质上是一个函数，只不过这个函数的第一个参数必须是 `self` ——在类中给它另外一个名字，叫“方法”——跟函数相比，没有本质的不同。

当建立了实例之后，用实例来调用这个方法的时候，因为Python解释器把实例已经作为第一参数隐式地传给了该方法，所以就不需要显示地写出 `self` 参数了——这个观点反复强调，就是让读者理解 `self` 就是实例。

如果要把实例显示地传给方法，可以用下面的方式进行，

```
>>> Foo.bar(f)
This is a normal method of class.
```

用这种方式，跟证实了前述观点，即实例化之后，`self` 和实例 `f` 是相同的。通常，我们在类里面使用 `self`，类外面使用 `f` 这个实例，两者有分工。

类(4)

如果在用类调用方法的时候，不传实例，会怎样？

```
>>> Foo.bar()
```

这样执行，不同的Python版本，报错信息有所不同。

Python 2的报错信息是：

```
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    Foo.bar()
TypeError: unbound method bar() must be called with Foo instance as first argument (go
t nothing instead)
```

而Python 3下报错信息变成了：

```
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    Foo.bar()
TypeError: bar() missing 1 required positional argument: 'self'
```

不管你是用什么版本，最好都阅读上述两个报错信息。在Python 2的报错信息中，告诉我们 `bar()` 是非绑定方法，它必须以 `Foo` 的实例作为第一个参数；Python 3的报错信息也是告诉我们 `bar()` 缺少一个必须的参数 `self`，它也是一个实例。所以，不管哪个版本，都要传一个实例。

Python中一切皆对象——又是老生常谈，都是因为此观念之重要。类 `Foo` 的方法 `bar()` 也是对象——函数对象，那么，我们就可以像这样来获得该对象了。

```
>>> Foo.bar
<unbound method Foo.bar>      #Python 3的显示结果:<function Foo.bar at 0x00000000006AAC8
0>
```

在Python 2的结果中，可以很清晰看出，通过类调用的方法对象，是一个非绑定方法——`unbound method`，又遇到这个词语了。

此外，还可以通过实例来得到该对象。

```
>>> f.bar
<bound method Foo.bar of <__main__.Foo object at 0x02A9BFB0>>
```

用实例来得到这个方法对象，不管是Python 2还是Python 3，结果是一样的。在这里我们看到的是`bound method`——绑定方法。

下面就要逼近`unbound method`和`bound method`的概念本质了。

类(4)

在类 `Foo` 的属性中，有一个 `__dict__` 的特殊属性，前文已经介绍过了。我们使用它，来窥探内部信息。

```
>>> Foo.__dict__['bar']
<function bar at 0x02AA98F0>      #Python 3显示结果：<function Foo.bar at 0x000000000006AAC
80>
```

从这个层面进一步说明 `bar` 是一个函数对象。

这个似乎是已经熟悉的了。

下面我们再看一个新的东西——描述器。

什么是描述器？

Python中有几个特殊方法比较特殊，它们分别是 `__get__()`、`__set__()` 和 `__delete__()`，简单地说，有这些方法的对象叫做描述器。

描述器是属性、实例方法、静态方法、类方法和继承中使用的 `super` 的背后实现机制，它在 Python中使用广泛。这句话中的那些生疏的名词，以后都会用到，稍安勿躁。

如何使用？

上述三个特殊方法，可以用下面的方式来使用——所谓的描述器协议。

```
descr.__get__(self, obj, type=None) --> value

descr.__set__(self, obj, value) --> None

descr.__delete__(self, obj) --> None
```

关于描述器的内容，本节不重点阐述，这里提及它，目的是要解决“绑定方法”和“非绑定方法”的问题，所以，读者如果有兴趣深入了解描述器，可以去google。

弱水三千，只取一瓢。我们在这里也只看 `__get__()`。

```
>>> Foo.__dict__['bar'].__get__(None, Foo)
<unbound method Foo.bar>      #Python 3显示结果：<function Foo.bar at 0x000000000006AAC
80>
```

对照描述器协议，我将 `self` 赋以了 `None`，其返回结果和 `Foo.bar` 的返回结果是一样的。让 `self` 为 `None` 的意思就是没有给定的实例，因此该方法被认为非绑定方法（`unbound method`）。

如果给定一个实例呢？

```
>>> Foo.__dict__['bar'].__get__(f, Foo)
<bound method Foo.bar of <__main__.Foo object at 0x02A9BFB0>>
```

类(4)

这时候的显示结果和 `f.bar` 是相同的。

综上所述，可以认为：

- 当通过类来获取方法的时候，得到的是非绑定方法对象。
- 当通过实例获取方法的时候，得到的是绑定方法对象。

所以，通常用实例调用的方法，都是绑定方法。那么非绑定方法在哪里会用到呢？当学习“继承”相关内容的时候，它会再次登场。

静态方法和类方法

先看下面的代码

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

class Foo(object):          #Python 3: class Foo:
    one = 0

    def __init__(self):
        Foo.one = Foo.one + 1

def get_class_attr(cls):
    return cls.one

if __name__ == "__main__":
    f1 = Foo()
    print "f1:", Foo.one      #Python 3: print("f1:"+str(Foo.one)), 下同, 从略
    f2 = Foo()
    print "f2:", Foo.one

    print get_class_attr(Foo)
```

在上述代码中，有一个函数 `get_class_attr()`，这个函数的参数我用 `cls`，从函数体的代码中看，要求它引用的对象应该具有属性 `one`，这就说明，不是随便一个对象就可以的。恰好，就是这么巧，我在前面定义的类 `Foo` 中，就有 `one` 这个属性。于是乎，我在调用这个函数的时候，就直接将该类对象传给了它 `get_class_attr(Foo)`。

其运行结果如下：

```
f1: 1
f2: 2
2
```

在这个程序中，函数 `get_class_attr()` 写在了类的外面，但事实上，函数只能调用前面写的那个类对象，因为不是所有对象都有那个特别的属性的。所以，这种写法，使得类和函数的耦合性太强了，不便于以后维护。这种写法是应该避免的。避免的方法就是把函数与类融为一体。于是就有了下面的写法。

类(4)

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

class Foo(object):      #Python 3: class Foo:
    one = 0

    def __init__(self):
        Foo.one = Foo.one + 1

    @classmethod
    def get_class_attr(cls):
        return cls.one

if __name__ == "__main__":
    f1 = Foo()
    print "f1:", Foo.one
    f2 = Foo()
    print "f2:", Foo.one

    print f1.get_class_attr()
    print "f1.one", f1.one
    print Foo.get_class_attr()

    print "***** 10"
    f1.one = 8
    Foo.one = 9
    print f1.one
    print f1.get_class_attr()
    print Foo.get_class_attr()
```

在这个程序中，出现了 `@classmethod` ——装饰器——在函数那部分遇到过了。需要注意的是 `@classmethod` 所装饰的方法的参数中，第一个参数不是 `self`，这是和我们以前看到的类中的方法是有区别的。这里我使用了参数 `cls`，你用别的也可以，只不过习惯用 `cls`。

再看对类的使用过程。先贴出上述程序的执行结果：

```
f1: 1
f2: 2
2
f1.one 2
2
*****
8
9
9
```

分别建立两个实例，此后类属性 `Foo.one` 的值是2，然后分别通过实例和类来调用 `get_class_attr()` 方法（没有显示写 `cls` 参数），结果都相同。

类(4)

当修改类属性和实例属性，再次通过实例和类调用 `get_class_attr()` 方法，得到的依然是类属性的结果。这说明，装饰器 `@classmethod` 所装饰的方法，其参数 `cls` 引用的对象是类对象 `Foo`。

至此，可以下一个定义了。

所谓类方法，就是在类里面定义的方法，该方法由装饰器 `@classmethod` 所装饰，其第一个参数 `cls` 所引用的是这个类对象，即将类本身作为引用对象传入到此方法中。

理解了类方法之后，用同样的套路理解另外一个方法——静态方法。还是先看代码——一个有待优化的代码。

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

T = 1

def check_t():
    T = 3
    return T

class Foo(object):      #Python 3: class Foo:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        if check_t():
            return self.name
        else:
            return "no person"

if __name__ == "__main__":
    f = Foo("canglaoshi")
    name = f.get_name()
    print name      #Python 3: print(name)
```

先观察上面的程序，发现在类 `Foo` 里面使用了外面定义的函数 `check_t()`。这种类和函数的关系，也是由于有密切关系，从而导致程序维护有困难，于是在和前面同样的理由之下，就出现了下面比较便于维护的程序。

```

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

T = 1

class Foo(object):      #Python 3: class Foo:
    def __init__(self, name):
        self.name = name

    @staticmethod
    def check_t():
        T = 1
        return T

    def get_name(self):
        if self.check_t():
            return self.name
        else:
            return "no person"

if __name__ == "__main__":
    f = Foo("canglaoshi")
    name = f.get_name()
    print name      #Python 3: print(name)

```

经过优化，将原来放在类外面的函数，移动到了类里面，也就是函数 `check_t()` 现在位于类 `Foo` 的命名空间之内了。但是，不是简单的移动，还要在这个函数的前面加上 `@staticmethod` 装饰器，并且要注意的是，虽然这个函数位于类的里面，跟其它的方法不同，它不以 `self` 为第一个参数。当使用它的时候，可以通过实例调用，比如 `self.check_t()`；也可以通过类调用这个方法，比如 `Foo.check_t()`。

从上面的程序可以看出，尽管 `check_t()` 位于类的命名空间之内，它却是一个独立的方法，跟类没有什么关系，仅仅是为了免除前面所说的维护上的困难，写在类的作用域内的普通函数罢了。但，它的存在也是有道理的，以上的例子就是典型说明。当然，在类的作用域里面的时候，前面必须要加上一个装饰器 `@staticmethod`。我们将这种方法也给予命名，称之为静态方法。

方法，是类的重要组成部分。本节专门讲述了方法中的几种特殊方法，它们为我们使用类的方法提供了更多便利的工具。但是，类的重要特征之一——继承，还没有亮相。

[总目录](#) | [上节：类\(3\)](#) | [下节：类\(5\)](#)

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

你们仍是属肉体的，因为在你们中间有嫉妒分争，这岂不是属乎肉体，照着世人的样子行吗？…我栽种了，亚波罗浇灌了，惟有神叫他生长。(1 CORINTHIANS 3:3,6)

类(5)

继承

继承——OOP的三个特征：多态、继承、封装——是类的重要内容。

继承，也是人的贪欲。

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，“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，就一夜之间变成富n代；如果继承了“革命先烈的光荣传统”，就红色？

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。

概念

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，例如网友令狐虫持有这样的观点：

类(5)

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。

在Python 2 中，我们这样定义新式类：

```
class NewStyle(object):
    pass
```

这就是典型的继承。这个类继承了 `object`，`object` 是所有类的父类。这种定义是从Python 2.2开始的，它解决了以往的类和类型的不统一的问题，自那以后，类就是一种数据类型了。

发展到Python 3，类的定义变为：

```
class NewStyle:
    pass
```

不再显示地写出 `object`，是因为Python 3中的所有类，都隐式地继承了 `object`。

总而言之，`object` 就是所有类的父类。

单继承

这是只从一个父类那里继承。

```
>>> class P(object):          #Python 3: class P:
    pass

>>> class C(P):
    pass
```

寥寥数“键”，就实现了继承。

类(5)

类 `P` 是一个通常的类，只不过在Python的两个版本中，定义样式稍微不同罢了。类 `C`（注意字母大写）则是定义的一个子类，它用 `C(P)` 的形式继承了类 `P`——称之为父类——虽然父类什么也没有。

子类 `C` 继承父类 `P` 的方式就是在类名称后面的括号里面写上父类的名字，不管是Python的那个版本。既然继承了父类，那么父类的一切都带入到了子类，所以在Python 2中就没有必要重复写 `object` 了，它已经通过父类 `P` 被继承到子类 `C` 了；Python 3中，要显式的写上父类的名字，除了 `object`，它不会隐式继承任何其它类。

```
>>> C.__base__  
<class '__main__.P'>
```

还记得类的一个特殊属性吗？由 `C.__base__` 可以得到类的父类。刚才的操作，就显示出类 `C` 的父类是 `P`。

为了深入理解“继承”的作用，让父类做一点点事情。

```
>>> class P(object):      #Python 3: class P:  
    def __init__(self):  
        print "I am a rich man."      #Python 3: print("I am a rich man.")  
  
>>> class C(P):  
    pass  
  
>>> c = C()  
I am a rich man.
```

父类 `P` 中增加了初始化函数，然后子类 `C` 继承它。我们已经熟知，当建立实例的时候，首先要执行类中的初始化函数。因为子类 `C` 继承了父类，就把父类中的初始化函数拿到了子类里面，所以在 `c = C()` 的时候，执行了父类中定义的初始化函数——这就是继承，而且是从一个父类那里继承来的，所以也称之为单继承。

看一个比较完成的程序示例。

类(5)

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

class Person(object):      #Python 3: class Person:
    def __init__(self, name):
        self.name = name

    def height(self, m):
        h = dict([("height", m),])
        return h

    def breast(self, n):
        b = dict([("breast", n),])
        return b

class Girl(Person):
    def get_name(self):
        return self.name

if __name__ == "__main__":
    cang = Girl("canglaoshi")
    print cang.get_name()      #Python 3: print(cang.get_name()), 下同, 从略
    print cang.height(160)
    print cang.breast(90)
```

上面这个程序，保存之后运行：

```
canglaoshi
{'height': 160}
{'breast': 90}
```

对以上程序进行解释：

首先定义了一个类 `Person`，把它作为父类。然后定义了一个子类 `Girl`，继承了 `Person`。

在子类 `Girl` 中，只写了一个方法 `get_name()`，但是因为是继承了 `Person`，那么 `Girl` 就全部拥有了 `Person` 中的方法和属性。子类 `Girl` 的方法 `get_name()` 中，使用了属性 `self.name`，但是在类 `Girl` 中，并没有什么地方显示创建了这个属性，就是因为继承 `Person` 类，在父类中有初始化函数。所以，当使用子类创建实例的时候，必须传一个参数 `cang = Girl("canglaoshi")`，然后调用实例方法 `cang.get_name()`。对于实例方法 `cang.height(160)`，也是因着继承的缘故使然。

在上面的程序中，子类 `Girl` 里面没有与父类 `Person` 重复的属性和方法，但有时候，会遇到这样的情况。

类(5)

```
class Girl(Person):
    def __init__(self):
        self.name = "Aoi sola"

    def get_name(self):
        return self.name
```

在子类里面，也写了一个初始化函数，并且定义了一个实例属性 `self.name = "Aoi sola"`。在父类中，也有初始化函数。在这种情况下，再次执行程序。

在Python 2中出现异常：

```
TypeError: __init__() takes exactly 1 argument (2 given)
```

Python 3中也有异常：

```
TypeError: __init__() takes 1 positional argument but 2 were given
```

不管哪个版本中的异常信息，都告诉我们，创建实例的时候，传入的参数个数多了。根源在于，子类 `Girl` 中的初始化函数，只有一个 `self`。因为跟父类中的初始化函数重名，虽然继承了父类，但是将父类中的初始化函数覆盖了，导致父类中的 `__init__()` 在子类中不再实现。所以，实例化子类，不应该再显式地传参数。

```
if __name__ == "__main__":
    cang = Girl()          #不在显示地传参数
    print cang.get_name()    #Python 3: print(cang.get_name()), 下同, 从略
    print cang.height(160)
    print cang.breast(90)
```

如此修改之后，再运行，则显示结果：

```
Aoi sola
{'height': 160}
{'breast': 90}
```

从结果中不难看出，如果子类中的方法或属性覆盖了父类（即与父类同名），那么就不在继承父类的该方法或者属性。

像这样，子类 `Girl` 里面有与父类 `Person` 同样名称的方法和属性，也称之为对父类相应部分的重写。重写之后，父类的相应部分不再被继承到子类，没有重写的部分，在子类中依然被继承，从上面程序可以看出来此结果。

还有一种可能存在，就是重写之后，如果要在子类中继承父类中相应部分，怎么办？

调用覆盖的方法

类(5)

承接前面的问题和程序，可以对子类 Girl 做出这样的修改。

```
class Girl(Person):
    def __init__(self, name):
        Person.__init__(self, name)
        self.real_name = "Aoi sola"

    def get_name(self):
        return self.name
```

请读者注意观察 Girl 的初始化方法，与前面的有所不同。为了能够使用父类的初始化方法，以类方法的方式调用 Person.__init__(self, name)。另外，在子类的 __init__() 的参数中，要增加相应的参数 name。这样就回答了前面的问题。

实例化子类，以下面的方式运行程序：

```
if __name__ == "__main__":
    cang = Girl("canglaoshi")
    print cang.real_name
    print cang.get_name()
    print cang.height(160)
    print cang.breast(90)
```

执行结果为：

```
Aoi sola
canglaoshi
{'height': 160}
{'breast': 90}
```

就这样，使用类方法的方式，将父类中被覆盖的方法再次在子类中实现。

但上述方式有一个问题，如果父类的名称因为某种目前你无法预料的原因修改了，子类中该父类的名称也要修改，有如果程序比较复杂或者忘记了，就会出现异常。于是乎，就有了更巧妙的方法—— super。再重写子类。

```
class Girl(Person):
    def __init__(self, name):
        #Person.__init__(self, name)
        super(Girl, self).__init__(name)
        self.real_name = "Aoi sola"

    def get_name(self):
        return self.name
```

仅仅修改一处，将 Person.__init__(self, name) 修改为 super(Girl, self).__init__(name)。执行程序后，显示的结果与以前一样。

类(5)

关于 `super`，有人做了非常深入的研究，推荐读者阅读 [《Python's super\(\) considered super!》](#)，文中已经探究了 `super` 的工作过程。读者如果要深入了解，可以阅读这篇文章。

多重继承

前面所说的继承，父类都只有一个。但，继承可以来自多个“父”，这就是多重继承。

所谓多重继承，就是指某一个子类的父类，不止一个，而是多个。比如：

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

class Person(object):      #Python 3: class Person:
    def eye(self):
        print "two eyes"      #Python 3: print("two eyes"),下同,从略

    def breast(self, n):
        print "The breast is: ",n

class Girl(object):        #Python 3: class Gril:
    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。

类(5)

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):      #Python 3: class K1:
    def foo(self):
        print "K1-foo"  #Python 3: print("K1-foo")，下同，从略

class K2(object):      #Python 3: class K2:
    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”。

Python 2的新式类，以及Python 3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。

导致新式类和Python 3中继承顺序较旧式类有所变化，其原因是mro（Method Resolution Order）算法，读者对此若有兴趣，可以到网上搜索关于这个算法的内容进行了解。

[总目录](#) | [上节：类\(4\)](#) | [下节：多态和封装](#)

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

爱是恒久忍耐，又有恩慈；爱是不嫉妒，爱是不自夸，不张狂，不作害羞的事，不求自己的益处，不轻易发怒，不计算人的恶，不喜欢不义，只喜欢真理；凡事包容，凡事相信，凡事盼望，凡事忍耐。(1 CORINTHIANS 13:4-7)

多态和封装

“多态”和“封装”是OOP的重要特征——前面说的“继承”也是。但是，对于Python而言，对这两个的理解也有很多不同。建议读者“吃百家宴”，到网上搜一搜有关话题，不少人写了文章来讨论。

多态

这里我仅仅针对初学者，按照自己的理解，谈谈零基础学Python的读者可以怎样理解“多态”，因为“多态”就如同其名字一样，在理解上也是“多态”的。

先来看这样的例子：

```
>>> "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()` 的参数类型。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\(5\)](#)中对此的解释。

```
>>> f(2,3)
5
>>> f("qiw","sir")
'qiwsir'
>>> f(["python","java"],["c++","lisp"])
['python', 'java', 'c++', 'lisp']
```

这里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，都能得到不报错的结果。

以上，就体现了“多态”——同一种行为具有不同表现形式和形态的能力，换一种说法，就是对象多种表现形式的体现。

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
```

本书由于不属于这种概念争论范畴，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，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），是指物件導向程式執行時，相同的訊息可能會送給多個不同的類別之物件，而系統可依據物件所屬類別，引發對應類別的方法，而有不同的行為。簡單來說，所謂多型意指相同的訊息給予不同的物件會引發不同的動作稱之。

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()`。

上述的种种多态表现，皆因为Python是一种解释型的语言，不需要进行预编译，只在运行时才确定状态。所以，Python就被认为天生是一种多态的语言。也有人持相反观点，认为Python不支持多态，在理由中，也用了上述内容。看来，看着半杯水，的确能够有不同的结论——“还有半杯水呢！”和“还剩半杯水了！”。

争论，让给思想者。我们，围观。

为了让读者能够进一步理解Python的多态特点，必须要比较，不跟世界上三分之二尚处于水深火热的劳苦大众比较，怎能体会到自己生活在幸福的祖国大花园内。

《Thinking in Java》的作者Bruce Eckel在2003年5月2日发表了一篇题为《Strong Typing vs. Strong Testing》的博客，在其中将Java和Python的多态特征进行了比较，在此我选摘部分内容，重温大师的论述。

先来欣赏大师所撰写的一段Java代码：

```
// Speaking pets in Java:
interface Pet {
    void speak();
}

class Cat implements Pet {
    public void speak() { System.out.println("meow!"); }
}

class Dog implements Pet {
    public void speak() { System.out.println("woof!"); }
}

public class PetSpeak {
    static void command(Pet p) { p.speak(); }
    public static void main(String[] args) {
        Pet[] pets = { new Cat(), new Dog() };
        for(int i = 0; i < pets.length; i++)
            command(pets[i]);
    }
}
```

如果读者没有学习过Java，对上述代码理解可能不是很顺畅，这不重要。主要观察 `command(Pet p)`，这种写法意味着函数 `command()` 所能接受的参数的类型必须是 `Pet` 类型，其它类型不行。所以，必须创建 `interface Pet` 这个接口并且类 `Cat` 和 `Dog` 继承它，然后

才能upcast them to the generic command() method。（原文：I must create a hierarchy of Pet, and inherit Dog and Cat so that I can upcast them to the generic command() method.）

与上面的代码相对应，大师提供了Python代码，如下所示：

```
# Speaking pets in Python:  
class Pet:  
    def speak(self): pass  
  
class Cat(Pet):  
    def speak(self):  
        print "meow!"  
  
class Dog(Pet):  
    def speak(self):  
        print "woof!"  
  
def command(pet):  
    pet.speak()  
  
pets = [ Cat(), Dog() ]  
  
for pet in pets:  
    command(pet)
```

注意这段Python代码中的 `command()` 函数，其参数 `pet` 并没有要求必须是前面的 `Pet` 类型（注意区分大小写），仅仅是一个名字为 `pet` 的对象引用罢了。Python不关心引用的对象是什么类型，只要改对象有 `speak()` 方法即可。提醒读者注意的是，因为历史原因（2003年），大师当时写的是旧式类。

根据我们对Python的理解，上面代码中的类 `Pet` 其实是多余的。是的，大师也这么认为，只是因为大师当时是完全模仿Java程序而写的。随后，大师就修改了上面的代码。

```
# Speaking pets in Python, but without base classes:
class Cat:
    def speak(self):
        print "meow!"

class Dog:
    def speak(self):
        print "woof!"

class Bob:
    def bow(self):
        print "thank you, thank you!"
    def speak(self):
        print "hello, welcome to the neighborhood!"
    def drive(self):
        print "beep, beep!"

def command(pet):
    pet.speak()

pets = [ Cat(), Dog(), Bob() ]

for pet in pets:
    command(pet)
```

不仅去掉了没什么用的类 `Pet`，又增加了一个新的类 `Bob`，这个类根本不是如 `Cat` 和 `Dog` 那样的类型，只是它碰巧也有一个名字为 `speak()` 的方法罢了。但是，也依然能够在 `command()` 函数中被调用。

这就是Python中的多态特点，大师Brue Eckel通过非常有说明了的代码说明了Java和Python的区别，并充分展示了Python中的多态特征。

诚如前面所述，Python不检查传入对象的类型（上面大师所写的代码中非常清晰表明了这点），这种方式被称之为“隐式类型”（latent typing）或者“结构式类型”（structural typing），也被通俗的称为“鸭子类型”(duck typing)，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，“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。”

鸭子类型就意味着可以向任何对象发送任何消息，语言只关心该对象能否接受该消息，不强求该对象是否某一种特定的类型——该对象的多态表现。

对于Python的这种特征，有一批程序员不接受，他们认为在程序被执行的时候，可能收到错误的对象，而且这种错误还可能潜伏在程序的某个角落。因此在编程领域就有了“强类型”（如Java）和“弱类型”（如Python）之争。

对于此类争论，大师Brue Eckel在上面所提到的博客中，给出了非常明确的回答。下面原文恭录于此：

Strong testing, not strong typing.

So this, I assert, is an aspect of why Python works. C++ tests happen at compile time (with a few minor special cases). Some Java tests happen at compile time (syntax checking), and some happen at run time (array-bounds checking, for example). Most Python tests happen at runtime rather than at compile time, but they do happen, and that's the important thing (not when). And because I can get a Python program up and running in far less time than it takes you to write the equivalent C++/Java/C# program, I can start running the real tests sooner: unit tests, tests of my hypothesis, tests of alternate approaches, etc. And if a Python program has adequate unit tests, it can be as robust as a C++, Java or C# program with adequate unit tests (although the tests in Python will be faster to write).

读大师的话，醍醐灌顶，豁然开朗，再也不去参与那些浪费唾沫的争论了。

顺便再告诉读者，从发表于2003年5月2日的《Strong Typing vs. Strong Testing》中可以看出，大师在那时已经开始在授课的过程中给学生使用Python了。2003年，那时候赵国程序员，有多少知道这个星球上有一种名为Python的计算机高级语言。

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

“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)是对具体对象的一种抽象，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

class ProtectMe(object):          #Python 3: class ProtectMe:
    def __init__(self):
        self.me = "qiwsir"
        self.__name = "kivi"

    def __python(self):
        print "I love Python."      #Python 3: print("I love Python."), 下同, 从略

    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

class ProtectMe(object):      #Python 3: 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      #Python 3: 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](#) | [上节：类\(5\)](#) | [下节：定制类](#)

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

Who are you to pass judgment on servants of another? It is before their own lord that they stand or fall. And they will be upheld, for the Lord is able to make them stand.

定制类

类是对象，类也是对象类型。字符串、列表、字典等是Python中内置的对象类型，除此之外，我们可以编写类，自定义对象类型。

类和对象类型

如果时至今日，你还没有充分理解类和对象类型的问题，可以再看看如下内容。

```
>>> class C1(object): pass      #Python 3: class C1: pass

>>> class C2(object): pass      #Python 3: class C2: pass

>>> a = C1()
>>> b = C2()
>>> type(a)
<class '__main__.C1'>
>>> type(b)
<class '__main__.C2'>
```

`type()` 是我们已经知晓了的内建函数，它返回的是对象类型。`a = C1()`，是实例化，创建了一个实例，也是一个赋值语句，将变量 `a` 与类 `C1()` 建立了引用关系，这就和以前 `a = 2` 的效果是一样的。所以，我们可以通过 `type(a)` 来得到实例或者说是这个变量所引用对象的类型。

在Python中，还有一个函数，专门来判断一个对象是不是另一个给定类的实例。

```
>>> help(isinstance)
Help on built-in function isinstance in module __builtin__:

isinstance(...)
    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.).
```

这是Python 2下的帮助文档信息，Python 3下的内容与之类似。

从 `isinstance()` 的名字上就能知道它是干什么的。

定制类

用它可以判断一个对象是否是一个类或者子类的实例，如果第二个参数是类型，也可以判断是否为该类型。

```
>>> isinstance(a, C1)
True
>>> isinstance(a, C2)
False
```

a 是类 c1 的实例，不是 c2 的实例。类似操作，还可以这么做：

```
>>> m = 1
>>> isinstance(m, int)
True
>>> isinstance(m, float)
False
```

用以前的话说，m 所引用的对象是整数型，简说成 m 是整数型。但是，如果从 instance() 的操作中看，m 和 a 是等效的，你可以认为 a 是C1类型的对象。

由此，我们进一步理解了类就是一种对象类型的。

定制类

必须要定制类，因为这个世界太复杂。

定制类，就要用到类的特殊方法，比如初始化函数 `__init__`，虽然用途很广泛，但仅仅用它还嫌不够，还要用到其它的特殊方法。

在Python的官方网站上，专门有介绍[Special method names](#)的章节，读者可以去仔细阅读。恕我不一一介绍。

本节中，仅根据例子中的问题，使用某个特殊方法。

```

#!/usr/bin/env python

class RoundFloat(object):      #Python 3: class RoundFloat:
    def __init__(self, val):
        assert isinstance(val, float), "value must be a float."
        self.value = round(val, 2)

    def __str__(self):
        return "{:.2f}".format(self.value)

    __repr__ = __str__

if __name__ == "__main__":
    r = RoundFloat(2.185)
    print r                  #Python 3: print(r)
    print type(r)            #Python 3: print(type(r))

```

上述程序中的类 `RoundFloat` 的作用是定义了一种两位小数的浮点数类型，利用这个类，能够得到两位小数的浮点数。

在初始化函数中 `assert isinstance(val, float), "value must be a float."` 是对输入的数据类型进行判断，如果不是浮点数就会抛出异常提示。关于 `assert`（断言）可以参看后续内容。

方法 `__str__()` 是一个特殊方法。实现这个方法，目的就是能够得到打印的内容。这里就是将前面四舍五入保留了两位小数的浮点数，以小数点后有两位小数的形式输出。

`__repr__ = __str__` 的含义是在类被调用，即向变量提供 `__str__()` 里的内容。

执行程序，结果是：

```

2.19
<class '__main__.RoundFloat'>

```

如果是 `RoundFloat(2.185)`，返回的结果是 `2.00`。

对比看，`int(2.34)` 和 `RoundFloat(2.185)` 完全等效，即 `int` 是对象类型，也是数据转换的函数；`RoundFloat` 具有同样的功能。`RoundFloat` 就是我们新定义的对象类型。

仿照上面的做法，我们还可以定制一个专门显示分数的类。

如你所知，如果在Python中直接输入状如`3/2`，它不会是一个分数，而是按照除法进行处理。但，分数的显示和使用，是显而易见的，Python的内置对象类型中又没有分数类型（不仅Python，相当多的高级语言都没有）。所以，有必要自定义一个相关的类型。

仿照前面定制类的方式，写出这样一段代码。

```

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

class Fraction(object):      #Python 3: class Fraction:
    def __init__(self, number, denom=1):
        self.number = number
        self.denom = denom

    def __str__(self):
        return str(self.number) + '/' + str(self.denom)

    __repr__ = __str__

if __name__ == "__main__":
    f = Fraction(2, 3)
    print f      #Python 3: print(f)

#output: 2/3

```

类 `Fraction` 就是自定义的分数类型。由此可见，自定义类是相当重要和必要的。

在这个基础上，继续将分数问题深入研究——分数相加。 $1/2 + 1/3 = 5/6$ ，计算过程如下：

1. 通分，即分母为原来两个分数的分母的最小公倍数，得到 $3/6 + 2/6$ ；
2. 分子相加，得到上述两个分数的和。

这样，我们将问题分解，找出一个关键，就是“通分”，而通分的关键是找出两个整数的最小公倍数。

如何找最小公倍数？步骤如下：

1. 计算两个数的最大公约数，假设a和b，最大公约数（greatest common divisor）用 $\text{gcd}(a, b)$ 表示；
2. 最小公倍数和最大公约数的关系是： $\text{lcm}(a, b) = |a * b| / \text{gcd}(a, b)$ ， $\text{lcm}(a, b)$ 表示这两个数的最小公倍数（lowest common multiple）。

读者不妨从新审视一番上述问题解决的思路。原始问题是计算两个分数的加法，然后将这个问题分解，再将分解之后的问题再分解。最终我们解决问题的基石是计算最大公约数。像这样解决问题的方法，我们称之为分治法，即一个复杂问题，分解为若干个简单问题，然后把简单问题组合起来，就解决了那个复杂问题——分而治之。

分解到最小的问题，就可以用编写函数的方式解决了。所以，先计算最大公约数和最小公倍数。

定制类

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

def gcd(a, b):          #最大公约数
    if not a > b:
        a, b = b, a
    while b != 0:
        remainder = a % b
        a, b = b, remainder
    return a

def lcm(a, b):          #最小公倍数
    return (a * b) / gcd(a,b)

if __name__ == "__main__":
    print gcd(8, 20)      #Python 3: print(gcd(8, 20))
    print lcm(8, 20)      #Python 3: print(lcm(8, 20))

#output:
#4
#40
```

如此，完成了最小公倍数的计算。然后，在前面定制的分数类的基础上，就可以制作两个分数相加的计算了。

```

#!/usr/bin/env python
#coding: utf-8
def gcd(a, b):
    if not a > b:
        a, b = b, a
    while b != 0:
        remainder = a % b
        a, b = b, remainder
    return a

def lcm(a, b):
    return (a * b) / gcd(a,b)

class Fraction(object):      #Python 3: class Frraction:
    def __init__(self, number, denom=1):
        self.number = number
        self.denom = denom

    def __str__(self):
        return str(self.number) + '/' + str(self.denom)

    __repr__ = __str__

    def __add__(self, other):
        lcm_num = lcm(self.denom, other.denom)
        number_sum = (lcm_num / self.denom * self.number) + (lcm_num / other.denom * other.number)
        return Fraction(number_sum, lcm_num)

if __name__ == "__main__":
    m = Fraction(1, 3)
    n = Fraction(1, 2)
    s = m + n
    print m,"+",n,"=",s

```

较之以前，增加了一个特殊方法 `__add__()`，它就是实现相加的特殊方法。在类中，有规定了加减乘除等运算的特殊方法。

在Python中，如果要实现某种运算，必须要有运算符，这是毫无疑问已经很熟悉的了。但是，这些运算符之所以能够被使用，都是因为有一些特殊方法才得以实现的。以下表格中列出几种常见运算符所对应的特殊方法，供参考。

定制类

二元运算符	特殊方法
+	<code>__add__</code> , <code>__radd__</code>
-	<code>__sub__</code> , <code>__rsub__</code>
*	<code>__mul__</code> , <code>__rmul__</code>
/	<code>__div__</code> , <code>__rdiv__</code> , <code>__truediv__</code> , <code>__rtruediv__</code>
//	<code>__floordiv__</code> , <code>__rfloordiv__</code>
%	<code>__mod__</code> , <code>__rmod__</code>
**	<code>__pow__</code> , <code>__rpow__</code>
<<	<code>__lshift__</code> , <code>__rlshift__</code>
>>	<code>__rshift__</code> , <code>__rrshift__</code>
&	<code>__and__</code> , <code>__rand__</code>
==	<code>__eq__</code>
!=,<>	<code>__ne__</code>
>	<code>__get__</code>
<	<code>__lt__</code>
>=	<code>__ge__</code>
<=	<code>__le__</code>

以“+”为例，不论是实现 `1 + 2` 还是 `'abc' + 'xyz'`，都是要执行 `1.__add__(2)` 或者 `'abc'.__add__('xyz')` 操作。也就是两个对象是否能进行加法运算，首先就要看相应的对象是否有 `__add__()` 方法（读者不妨在交互模式中使用 `dir()`，看一看整数、字符串是否有 `__add__()` 方法），一旦相应的对象有 `__add__()` 方法，即使这个对象从数学上不可加，我们都可以用加法的形式，来表达 `obj.__add__()` 所定义的操作。在Python中，运算符起到简化书写的功能，但它依靠特殊方法实现。

所以，在刚才自定义的类 `Fraction` 中，为了实现分数加法，我们重写了 `__add__()` 方法，也可以称之为运算符重载（对于Python是否支持重载，也是一个争论话题）。

就这样，我们解决了分数相加的问题。

但，上述加法还不是很完美，还可以有很多优化的地方，比如分数结果要化成最简分数等等。真正要做好一个分数运算的类，还有很多工作。

不过，在Python中，其实不用你自己做了，由高手做好。标准库中就有相应模块解决此问题。

```
>>> from fractions import Fraction
>>> m, n = Fraction(1, 3), Fraction(1, 2)
>>> m + n
Fraction(5, 6)
>>> print m + n      #Python 3: print(m + n)
5/6
>>> a, b = Fraction(1, 3), Fraction(1, 6)
>>> print a + b      #Python 3: print(a + b)
1/2
```

Python的魅力之一，就是它强大的标准库和第三方库，让你省心省力。

[总目录](#) | [上节：多态和封装](#) | [下节：黑魔法](#)

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

我们既蒙怜悯，受了这职分，就不丧胆，乃将那些暗昧可耻的事弃绝了，不行诡诈，不谬讲神的道理，只将真理表明出来，好在神面前把自己荐与各人的良心。(2 CORINTHIANS 4:1-2)

黑魔法

围绕类的话题，真实说也说不完，仅特殊方法，除了前面遇到过的 `__init__()`，`__new__()`，`__str__()` 等之外，还有很多。虽然它们仅仅是在某些特殊情景中使用，但是，因为本教程是“From Beginner to Master”。当然，不是学习了类的更多特殊方法就能达到Master水平，但是这是通往Master的一步。

本节试图再介绍一些点“黑魔法”，既能窥探到Python的更高境界，也能感受到Master的未来能力。俗话说“艺不压身”，还是多学点好。

优化内存的 `__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__', '__slots__', '__str__', '__subclasshook__', 'flower', 'tree']
```

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，可以通过实例属性赋值。

```
>>> t.flower = "haitanghua"
>>> t.flower
'haiwanghua'
```

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。

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，一句话概括，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。

属性拦截

有时候，访问某个类或者实例属性，它不存在，就会异常。对于异常，总是要处理的。就好像“寻隐者不遇”，却被童子“遥指杏花村”，将你“拦截”了，不至于因为“不遇”而垂头丧气。

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。

用例子说明。

```
>>> class A(object):          #Python 3: class A:
...     def __getattr__(self, name):
...         print "You use getattr"
...         #Python 3: print("You use getattr"), 下同, 从
略
...     def __setattr__(self, name, value):
...         print "You use setattr"
...         self.__dict__[name] = value
...
```

类 A 除了两个方法，没有别的了。

```
>>> a = A()
>>> a.x
You use getattr
```

a.x 这个实例属性，本来是不存在的，但是，由于类中有了 `__getattr__(self, name)` 方法，当发现属性 x 不存在于对象的 `__dict__` 中的时候，就调用了 `__getattr__`，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，如果再调用这个属性：

```
>>> 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)`，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]` 样式。因为如果用 `return self.__dict__[name]` 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 getattribute
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 getattribute
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，的确是“黑魔法”。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):      #Python 3: class Rectangle:
    """
    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()          #Python 3: print(r.getSize())
    r.setSize( (30, 40) )
    print r.width              #Python 3: print(r.width)
    print r.length             #Python 3: 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):          #Python 3: class Rectangle:
    """
    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      #Python 3: print(r.size)
    r.size = 30, 40
    print r.width     #Python 3: print(r.width)
    print r.length    #Python 3: 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。

如果要对于这种黑魔法有更深的理解，可以阅读：[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。

至此，是否注意到，我们使用了很多以双下划线开头和结尾的方法或者属性，比如 `__dict__`，`__init__()` 等。在Python中，用这种方法表示特殊的方法和属性，当然，这是一个惯例，之所以这样做，主要是确保这些特殊的名字不会跟你自己所定义的名称冲突，我们自己定义名称的时候，是绝少用双划线开头和结尾的。如果你需要重写这些方法，当然是可以的。

[总目录](#) | [上节：定制类](#) | [下节：迭代器](#)

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

For freedom Christ has set us free. Stand firm, therefore, and do not submit again to a yoke of slavery.

基督释放了我们，叫我们得以自由，所以要站立得稳，不要再被奴仆的轭挟制。
(GALATIANS 5:1)

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》。

```
>>> hasattr(list, '__iter__')
True
```

不仅仅是列表，文件、字典都有一个名为 `__iter__` 的方法，这说明它们都是可迭代的。

`__iter__()` 是对象的一个特殊方法，它是迭代规则(iterator protocol)的基础，有了它，说明对象是可迭代的。

跟迭代有关的一个内建函数 `iter()`，它的文档中这样描述：

```
>>> help(iter)
Help on built-in function iter in module __builtin__:

iter(...)
    iter(collection) -> iterator
    iter(callable, sentinel) -> iterator

    Get an iterator from an object. In the first form, the argument must
    supply its own iterator, or be a sequence.
    In the second form, the callable is called until it returns the sentinel.
```

这个函数前文介绍过，它返回一个迭代器对象。比如：

```
>>> lst = [1, 2, 3, 4]
>>> iter_lst = iter(lst)
>>> iter_lst
<listiterator object at 0x02BE8D50>      #Python 3返回结果:<list_iterator object at 0
x00000000034CD6D8>
```

从返回结果中可以看出，`iter_lst` 引用的是迭代器对象。那么，`iter_lst` 和 `lst` 有区别吗？

```
>>> hasattr(lst, "__iter__")
True
>>> hasattr(iter_lst, "__iter__")
True
```

它们都有 `__iter__`，这是相同点，说明它们都是可迭代的。

但是：

Python 2：

```
>>> hasattr(lst, "next")
False
>>> hasattr(iter_lst, "next")
True
```

Python 3：

```
>>> hasattr(lst, "__next__")
False
>>> hasattr(iter_lst, "__next__")
True
```

这就是两者的区别。我们像 `iter_lst` 所引用的对象那样，具有 `next()`（Python 2）或者 `__next__()`（Python 3）方法的对象，称之为迭代器对象。显见，迭代器对象必然是可迭代的，反之则不然。

Python 3中迭代器对象实现的是 `__next__()` 方法，不是 `next()`。并且，在Python 3中有一个内建函数 `next()`，可以实现 `next(it)` 访问迭代器，这相当于Python 2中的 `it.next()`（`it`是迭代对象）。

为了体现Python强悍，自己写一个迭代器对象。

```

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

"""
the interator as range()
"""

class MyRange(object):    #Python 3: class MyRange:
    def __init__(self, n):
        self.i = 1
        self.n = n

    def __iter__(self):
        return self

    def next(self):          #Python 3: def __next__(self):
        if self.i <= self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()

if __name__ == "__main__":
    x = MyRange(7)
    print [i for i in x]      #Python 3中使用print()函数，下同，从略

```

将代码保存，并运行，结果是：

```
[1, 2, 3, 4, 5, 6, 7]
```

以上代码的含义，是自己仿写了类似 `range()` 的类，但是跟 `range()` 又有所不同，除了结果不同之外，还有：

- 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，即意味着它是可迭代的。
- 实现 `next()` 或者 `__next__()` 方法，从而使得这个对象是迭代器对象，并且方法中判断，在不满足条件的时候要发起 `StopIteration()` 异常。

再来看 `range()`（以下仅仅限于 Python 2）：

```
>>> a = range(7)
>>> hasattr(a, "__iter__")
True
>>> hasattr(a, "next")
False
>>> print a
[0, 1, 2, 3, 4, 5, 6]
```

所以我写的类和 `range()` 还是有很大区别的。

为了能深入理解迭代器的工作过程，我们这样来操作：

```
if __name__ == "__main__":
    x = MyRange(3)
    print "self.n=",x.n,";", "self.i=",x.i      #Python 3中使用print()函数，下同，从略
    x.next()
    print "self.n=",x.n,";", "self.i=",x.i
    x.next()
    print "self.n=",x.n,";", "self.i=",x.i
    x.next()
    print "self.n=",x.n,";", "self.i=",x.i
    x.next()
    print "self.n=",x.n,";", "self.i=",x.i
```

运行结果如下：

```
self.n= 3 ; self.i= 1
self.n= 3 ; self.i= 2
self.n= 3 ; self.i= 3
self.n= 3 ; self.i= 4

Traceback (most recent call last):
  File "F:\MyGitHub\StarterLearningPython\2code\21401.py", line 32, in <module>
    x.next()
  File "F:\MyGitHub\StarterLearningPython\2code\21401.py", line 21, in next
    raise StopIteration()
StopIteration
```

当 `next()` 或者 `__next__()` 中的 `self.i <= self.n` 为假，就 `raise StopIteration()`，结束迭代过程。

还记得斐波那契数列吗？前文已经多次用到，这里我们再次使用它，不过是要用它来做一个迭代器对象。

```

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

compute Fibonacci by iterator
"""

class Fibs(object):      #Python 3: class Fibs:
    def __init__(self, max):
        self.max = max
        self.a = 0
        self.b = 1

    def __iter__(self):
        return self

    def next(self):      #Python 3: 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__":
    fibs = Fibs(5)
    print list(fibs)      #Python 3: print(list(fibs))

```

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？

以上演示了迭代器的一个具体应用。综合本节上面的内容和前文对迭代的讲述，对迭代器做一个概括：

1. 在 Python 中，迭代器是遵循迭代协议的对象。
2. 可以使用 `iter()` 以从任何序列得到迭代器（如 `list`, `tuple`, `dictionary`, `set` 等）。
3. 编写类，实现 `__iter__()` 方法，以及 `next()`（Python 2）或 `__next__()`（Python 3）
 - 当没有元素时，则引发 `StopIteration` 异常。
4. 如果有很多值，列表就会占用太多的内存，而迭代器则占用更少内存。
5. 迭代器从第一个元素开始访问，直到所有的元素被访问完结束，只能往前不会后退。

迭代器不仅实用，也很有趣。看下面的操作：

```
>>> my_lst = [x**x for x in range(4)]
>>> my_lst
[1, 1, 4, 27]
>>> for i in my_lst: print i      #Python 3: print(i)

1
1
4
27
>>> for i in my_lst: print i

1
1
4
27
```

我连续两次调用列表 `my_lst` 进行循环，都能正常进行。这个列表相当于一个耐用品，可以反复使用。

在Python中，除了列表解析式，还可以做元组解析式，方法非常简单：

```
>>> my_tup = (x**x for x in range(4))
>>> my_tup
<generator object <genexpr> at 0x02B7C2B0>
>>> for i in my_tup: print i

1
1
4
27
>>> for i in my_tup: print i
```

对于 `my_tup`，我们已经看到，它是generator对象，关于这个名称先不管它，后面会讲解。当把它用到循环中，它明显是一次性用品，只能使用一次，再次使用，就什么也不显示了。

```
>>> type(my_lst)
<type 'list'>
>>> type(my_tup)
<type 'generator'>
```

`my_lst` 和 `my_tup` 是两种不同的对象，并且 `my_tup` 也不是元组，它是一个generator。其它先不管，请读者在你的Python交互模式中输入 `dir(my_tup)`，如果是Python 2，请查找是否有 `_iter_` 和 `next`；如果是Python 3则查看是否有 `_iter_` 和 `_next_`。答案是肯定的。这也是 `my_lst` 和 `my_tup` 所引用对象的区别。

因此，`my_tup` 引用的是一个迭代器对象。它的 `next()` 或者 `_next_()` 方法，使得它只能向前。

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()` 的结果是一个列表。但是，如果用 `help(xrange)` 查看（仅限于Python 2）：

```
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()` 类似 `range()`，但返回的不是列表。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__', '__len__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__reversed__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

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。

上述论述仅适用于Python 2，因为在Python 3里面，将 `range()` 优化了，相当于Python 2里面 `xrange()`，所以，在Python 3中就不再有 `xrange()`。

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))      #适用于Python 2，Python 3中的range()已经具有了Python 2的xrange()功能
[(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 `range(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](#) | [上节：黑魔法](#) | [下节：生成器](#)

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

圣灵所结的果子，就是仁爱、喜乐、和平、忍耐、恩慈、良善、信实、温柔、节制。这样的事，没有律法禁止。凡属基督耶稣的人，是已经把肉体连肉体的邪情私欲同钉在十字架上了。我们若是靠圣灵得生，就当靠圣灵行事。不要贪图虚名，彼此惹气，互相嫉妒。(GALATIANS 5:22-26)

生成器

上节中，我们曾经做过这样的操作：

```
>>> my_tup = (x**x for x in range(4))
>>> my_tup
<generator object <genexpr> at 0x02B7C2B0>
```

`generator`，翻译过来是生成器。

生成器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。

既然在探讨“迭代器”的时候出现了生成器，这就说明生成器和迭代器有着一定的渊源关系。

没错！生成器必须是可迭代的，它首先是迭代器。

但，它毕竟还是生成器，具有生成器的特质。

定义生成器

定义生成器，必须使用 `yield` 关键词。`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，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，里面有 `yield` 发起的三个语句。下面看如何使用它：

```
>>> ge = g()
>>> ge
<generator object g at 0xb7200edc>
>>> type(ge)
<type 'generator'>
```

调用函数，得到了一个生成器(generator)对象。

Python 2:

```
>>> dir(ge)
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__',
 '__init__', '__iter__', '__name__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'close', 'gi_code', 'gi_frame',
 'gi_running', 'next', 'send', 'throw']
```

Python 3:

```
>>> dir(ge)
['__class__', '__del__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
 '__lt__', '__name__', '__ne__', '__new__', '__next__', '__qualname__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'close', 'gi_code', 'gi_frame', 'gi_running', 'gi_yieldfrom', 'send', 'throw']
```

在这里看到了 `__iter__()` 和 `next()` 或 `__next__()`，虽然我们在函数体内并没有显示地写出 `__iter__()`、`next()` 和 `__next__()`，仅仅写了 `yield` 语句，它就已经成为迭代器了。

既然如此，当然可以：

```
>>> ge.next()          #Python 3: ge.__next__(), 下同, 从略
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` 关键词的函数是一个生成器对象，这个生成器对象也是迭代器。

于是可以这样定义：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()`：返回生成器之外。
2. `ge.next()`：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

函数返回值，本来已经有了一个 `return`，现在又出现了 `yield`，这两个有什么区别？

为了区别，我们写两个没有什么用途的函数：

```
>>> def r_return(n):
...     print "You taked me."          #Python 3: print("You taked me."),下同,从略
...     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."      #Python 3: print("You taked me.")，下同，从略
...     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()           #Python 3: yy.__next__(), 下同，从略
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。

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 ,      #Python 3: print(i, end=',')

```

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？读者可以将本书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。

最后一句，你在编程中，不用生成器也可以。

[总目录](#) | [上节：迭代器](#) | [下节：上下文管理器](#)

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

要使人晓得智慧和训诲，分辨通达的言语。使人处事，领受智慧、仁义、公平、正直的训诲。使愚人灵明、使少年人有知识和谋略。使智慧人听见、增长学问、使聪明人得着智谋、使人明白箴言和譬喻、懂得智慧人的言词和谜语。敬畏耶和华使知识的开端，愚妄人藐视智慧和训诲。

上下文管理器

在《文件(1)》中提到，如果要打开文件，一种比较好的方法是使用 `with` 语句，因为这种方法，不仅结构简单，更重要的是不用再单独去判断某种异常情况，也不用专门去执行文件关闭的指令了。

本节对这个有点神奇的 `with` 进行深入剖析。

概念

跟 `with` 相关的有一些概念，需要必须澄清。

上下文管理

如果把它作为一个概念来阐述，似乎有点多余，因为从字面上也可以有一丝的体会，但是，我要说的是，那点直觉的体会不一定等于理性的严格定义，特别是周遭事物越来越复杂的时候。

“上下文”的英文是context，在网上检索了一下关于“上下文”的说法，发现没有什么严格的定义，另外，不同的语言环境，对“上下文管理”有不同的说法。根据我个人的经验和能看到的某些资料，窃以为可以把“上下文”理解为某一些语句构成的一个环境（也可以说是代码块），所谓“管理”就是要在这个环境中做一些事情，做什么事情呢？就Python而言，是要将前面某个语句（“上文”）干的事情独立成为对象，然后在后面（“下文”）中使用这个对象来做事情。

上下文管理协议

英文是Context Management Protocol，既然是协议，就应该是包含某些方法的东西，大家都按照这个去做（协商好了的东西）。Python中的上下文管理协议中必须包含 `__enter__()` 和 `__exit__()` 两个方法。

看这两个方法的名字，估计读者也能领悟一二了（名字不是随便取的，这个和某个岛国取名字的方法不同，当然，现在人家也不是随便取了）。

上下文管理器

网上能够找到的最通常的说法是：上下文管理器是支持上下文管理协议的对象，这种对象实现了 `__enter__()` 和 `__exit__()` 方法。

这个简洁而准确的定义，一般情况下一些高手是理解了。如果读者有疑惑，就说明...，我还是要把一个高雅的定义通俗化更好一些。

在Python中，下面的语句，也存在上下文，但它们是一气呵成执行的。

```
>>> name = "laoqi"
>>> if name == "laoqi":
...     print name      #Python 3: print(name)
...
laoqi
>>> if name == "laoqi":
...     for i in name:
...         print i,      #Python 3: print(i, end=', ')
...
l a o q i
```

以上两个例子中，“上文”进行了判断，然后“下文”执行，从上而下，已经很通畅了。还有不那么通畅的，就是下面的情况。

```
>>> f = open("a.txt", "w")
>>> f.write("hello")
>>> f.write("python")
>>> f.close()
```

在这个示例中，当 `f = open("a.txt", "w")` 之后，其实这句话并没有如同前面的示例中那样被“遗忘”，它是让计算机运行到一种状态——文件始终处于打开状态——然后在这种状态中进行后面的操作，直到 `f.close()` 为止，这种状态才结束。

在这种情况下，我们就可以使用“上下文管理器”（英文：Context Manager），用它来获得“上文”状态对象，然后在“下文”使用它，并在整个过程执行完毕来收场。

更Python一点的说法，可以说是在某任务执行之初，上下文管理器做好执行准备，当任务（代码块）执行完毕或者中间出现了异常，上下文管理器负责结束工作。

这么好的一个东西，是Python2.5以后才进来的。

必要性

刚才那个向文件中写入hello和python两个单词的示例，如果你觉得在工程中也可以这样做，就大错特错了。因为它存在隐含的问题，比如在写入了hello之后，不知道什么原因，后面的python不能写入了，最能说服你的是恰好遇到了“磁盘已满”——虽然这种情况的概率可能比抓奖券还小，但作为严谨的程序员，是必须要考虑的，这也是程序复杂之原因，这时候后面的操作就出现了异常，无法执行，文件也不能close。解决这个问题的方法是用 `try ... finally ...` 语句，读者一定能写出来。

不错，的确解决了。

问题继续，如果要从一个文件读内容，写入到另外一个文件中，下面的样子你觉得如何？

首先建立一个文件，名称为23501.txt，里面的内容如下：

```
$ cat 23501.txt
hello laoqi
www.itdiffer.com
```

然后写出下面的代码，实现上述目的：

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

read_file = open("23501.txt")
write_file = open("23502.txt", "w")

try:
    r = read_file.readlines()
    for line in r:
        write_file.write(line)
finally:
    read_file.close()
    write_file.close()
```

如果你不知道“上下文管理器”，这样做无可厚非，可偏偏现在已经知道了，所以，从今以后这样做就不是最优的了，因为它可以用“上下文管理器”写的更好。所以，用 `with` 语句改写之后，就是很优雅的了。

```
with open("23501.txt") as read_file, open("23503.txt", "w") as write_file:
    for line in read_file.readlines():
        write_file.write(line)
```

跟前面的对比一下，是不是有点惊叹了？！所以，你可以理直气壮地说“我用 Python”。

可见上下文管理器是必要的，因为它让代码优雅了，当然优雅只是表象，还有更深层次的含义，继续阅读下面的内容能有深入体会。

深入理解

前面已经说了，上下文管理器执行了 `__enter__()` 和 `__exit__()` 方法，可是在 `with` 语句中哪里看到了这两个方法呢？

为了解把这个问题解释清楚，需要先做点别的操作，虽然工程中一般不需要做。

```

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

class ContextManagerOpenDemo(object):

    def __enter__(self):
        print "Starting the Manager."          #Python 3: print("Starting the Manager.")

    def __exit__(self, *others):
        print "Exiting the Manager."         #Python 3: print("Exiting the Manager.")

    with ContextManagerOpenDemo():
        print "In the Manager."           #Python 3: print("In the Manager.")

```

在上面的代码示例中，我们写了一个类 `ContextManagerOpenDemo()`，你就把它理解为我自己写的 `open()` 吧，当然使最简版本了。在这个类中，`__enter__()` 方法和 `__exit__()` 方法都比较简单，就是要检测是否执行该方法。

然后用 `with` 语句来执行，目的是按照“上下文管理器”的解释那样，应该首先执行类中的 `__enter__()` 方法，它总是在进入代码块前被调用的，接着就执行代码块——`with` 语句下面的内容，当代码块执行完毕，离开的时候又调用类中的 `__exit__()`。

检验一下，是否按照上述理想路径执行。

```

$ python 23502.py
Starting the Manager.
In the Manager.
Exiting the Manager.

```

果然如此。执行结果已经基本显示了上下文管理器的工作原理。

为了让它更接近 `open()`，需要再进一步改写，让它能够接受参数，以便于指定打开的文件。

```

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

class ContextManagerOpenDemo(object):
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        print "Starting the Manager."          #Python 3: print("Starting the Manager.")
        self.open_file = open(self.filename, self.mode)
        return self.open_file

    def __exit__(self, *others):
        self.open_file.close()
        print "Exiting the Manager."         #Python 3: print("Exiting the Manager.")

with ContextManagerOpenDemo("23501.txt", 'r') as reader:
    print "In the Manager."           #Python 3: print("In the Manager.")
    for line in reader:
        print line                  #Python 3: print(line)

```

这段代码的意图主要是：

1. 通过 `__init__()` 能够读入文件名和打开模式，以使得看起来更接近 `open()`；
2. 当进入语句块时，先执行 `__enter__()` 方法，把文件打开，并返回该文件对象；
3. 执行代码块内容，打印文件内容；
4. 离开代码块的时候，执行 `__exit__()` 方法，关闭文件。

运行结果是：

```

$ python 23502.py
Starting the Manager.
In the Manager.
hello laoqi

www.itdiffer.com

Exiting the Manager.

```

在上述代码中，我们没有显示地写捕获异常的语句，不管在代码块执行时候遇到什么异常，都是要离开代码块，那么就立刻让 `__exit__()` 方法接管了。

如果要把异常显现出来，也使可以，可以改写 `__exit__()` 方法。例如：

```

def __exit__(self, exc_type, exc_value, exc_traceback):
    return False

```

当代码块出现异常，则由 `__exit__()` 负责善后清理，如果返回 `False`，如上面的示例，则异常让 `with` 之外的语句逻辑来处理，这是通常使用的方法；如果返回 `True`，意味着不对异常进行处理。

从上面我们自己写的类和方法中，已经了解了上下文管理器的运行原理了。那么，`open()` 跟它有什么关系吗？

为了能清楚地查看，我们需要建立一个文件对象，并且使用 `dir()` 来看看是否有我们所期盼的东西。

```
>>> f = open("a.txt")
>>> dir(f)
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__', '__format__', '__
getattribute__', '__hash__', '__init__', '__iter__', '__new__', '__reduce__', '__reduc
e_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'clos
e', 'closed', 'encoding', 'errors', 'fileno', 'flush', 'isatty', 'mode', 'name', 'newl
ines', 'next', 'read', 'readinto', 'readline', 'readlines', 'seek', 'softspace', 'tell
', 'truncate', 'write', 'writelines', 'xreadlines']
```

读者是否运用你那迷迷糊糊的火眼金睛看到了两个已经很面熟的方法名称了？如果你找到了，你就心知肚明了。

在 `with` 语句中还有一个 `as`，虽然在上面示例中没有显示，但是一般我们还是不抛弃它的，它的作用就是将返回的对象付给一个变量，以便于以后使用。

contextlib 模块

Python 中的这个模块使上下文管理中非常好用的东东，这也是标准库中的一员，不需要另外安装了。

```
>>> import contextlib
>>> dir(contextlib)
['GeneratorContextManager', '__all__', '__builtins__', '__doc__', '__file__', '__n
ame__', '__package__', 'closing', 'contextmanager', 'nested', 'sys', 'warn', 'wraps']
```

常用的是 `contextmanger`、`closing` 和 `nested`。

contextlib.closing()

要想知道 `contextlib.closing()` 的使用方法，最常用的方法就是 `help()`，这是我们的一贯做法。

```
Help on class closing in module contextlib:

class closing(__builtin__.object)
| Context to automatically close something at the end of a block.
|
| Code like this:
|
|     with closing(<module>.open(<arguments>)) as f:
|         <block>
|
```

以上省略了部分内容。

有一种或许常用到的情景，就是连接数据库，并返回一个数据库对象，在使用完之后关闭数据库连接，其形状如下：

```
with contextlib.closing(CreateDB()) as db:
    db.query()
```

以上不是可运行的代码，只是一个架势，读者如果在编码中使用，需要根据实际情况改写。

当数据库语句 `db.query()` 结束之后，数据库连接自动关闭。

contextlib.nested()

`nested` 的汉语意思是“嵌套的，内装的”，从字面上读者也可能理解了，这个方法跟嵌套有关。前面有一个示例，是从一个文件读取，然后写入到另外一个文件。我不知道读者是否想过可以这么写：

```
with open("23501.txt") as read_file:
    with open("23503.txt", "w") as write_file:
        for line in read_file.readlines():
            write_file.write(line)
```

此种写法不是不行，但是不提倡，因为它太不Pythoner了。其实这里就涉及到了嵌套，因此可以使用 `contextlib.nested` 重写。

```
with contextlib.nested(open("23501.txt", "r"), open("23503.txt", "w")) as (read_file,
write_file):
    for line in read_file.readlines():
        write_file.write(line)
```

这是一种不错的写法，当然，在本节最前面所用到的写法，也是可以的，只要不用刚才那种嵌套。

contextlib.contextmanager

`contextlib.contextmanager`是一个装饰器，它作用于生成器函数（也就是带有`yield`的函数），一单生成器函数被装饰以后，就返回一个上下文管理器，即`contextlib.contextmanager`因为装饰了一个生成器函数而产生了`__enter__()`和`__exit__()`方法。例如：

特别要提醒，被装饰的生成器函数只能产生一个值，否则就会抛出`RuntimeError`异常；如果有`as`子句，则所产生的值，会通过`as`子句赋给某个变量，就如同前面那样，例如下面的示例（本示例来自：<http://www.ibm.com/developerworks/cn/opensource/os-cn-pythonwith/index.html>）。

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

from contextlib import contextmanager

@contextmanager
def demo():
    print "before yield."
    yield "contextmanager demo."
    print "after yield."

with demo() as dd:
    print "the word is: %s" % dd
```

尊重引用的文章，所以上述代码就不再注释Python 3下如何修改了。如果读者是使用Python 3的，可以自行将代码中的`print`语句修改为`print()`函数式样。

运行结果是：

```
$ python 23504.py
before yield.
the word is: contextmanager demo.
after yield.
```

为了好玩，再借用网上的一个示例，理解这个装饰器的作用（下面代码来自：

<http://preshing.com/20110920/the-python-with-statement-by-example/>），代码中用到了`cairo`模块，该模块的安装方法是：

```
sudo apt-get install libcairo2-dev
```

如果是windows操作系统，可以到官方网站下载：<http://cairographics.org/>

所执行的代码如下：

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

import cairo
from contextlib import contextmanager

@contextmanager
def saved(cr):
    cr.save()
    try:
        yield cr
    finally:
        cr.restore()

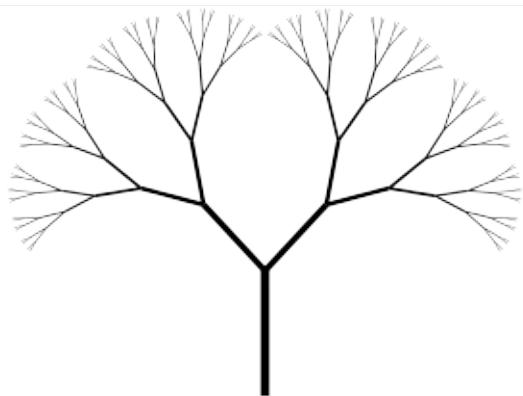
def tree(angle):
    cr.move_to(0, 0)
    cr.translate(0, -65)
    cr.line_to(0, 0)
    cr.stroke()
    cr.scale(0.72, 0.72)
    if angle > 0.2:
        for a in [-angle, angle]:
            with saved(cr):
                cr.rotate(a)
                tree(angle * 0.75)

surf = cairo.ImageSurface(cairo.FORMAT_ARGB32, 280, 204)
cr = cairo.Context(surf)
cr.translate(140, 203)
cr.set_line_width(5)
tree(0.75)
surf.write_to_png('fractal-tree.png')
```

不过，我感到很奇怪，我得到的图片是这样的：



而原文中得到的图片是这样的：



请读者指正。

[总目录](#) | [上节：生成器](#) | [下节：错误和异常\(1\)](#)

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

因为各人必担当自己的担子。在道理上受教的，当把一切需用的供给施教的人。不要自欺，神是轻慢不得的。人种的是什么，收的也是什么。顺着情欲撒种的，必从情欲收败坏；顺着圣灵撒种的，必从圣灵收永生。我们行善，不可丧志，若不灰心，到了时候就要收成。(GALATIANS 6:5-9)

错误和异常(1)

对于程序在执行过程中因为错误或者别的原因而中止的现象，已经看过多次了，那些都可以归为“错误和异常”现象。本章就要对这种现象进行近距离观察。

错误

不管是小白还是高手，在编写程序的时候，错误往往是难以避免的。可能是因为语法用错了，也可能是拼写做了，当然还可能其它莫名其妙的错误，比如冒号写成了全角的了，等等。总之，编程中有相当一部分工作就是要不停地修改错误。

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。

另一种常见错误是逻辑错误。逻辑错误可能是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，于是抛出提示信息，即为异常。

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          #Python 3: ZeroDivisionError: division by zero
```

错误和异常(1)

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。

在刚才的例子中，明确告诉我们一场的类型是 `ZeroDivisionError`，并且对此异常类型做了解释（虽然Python 2和Python 3的解释不完全一致，但意思是一样的）。

为什么0不能作除数（分母）？虽然小学生都知道不能作，但是不一定知道为什么不能作。读者对此有兴趣思考思考吗？

下表中列出常见的异常

异常	描述
<code>NameError</code>	尝试访问一个没有申明的变量
<code>ZeroDivisionError</code>	除数为0
<code>SyntaxError</code>	语法错误
<code>IndexError</code>	索引超出序列范围
<code>KeyError</code>	请求一个不存在的字典关键字
<code>IOError</code>	输入输出错误（比如你要读的文件不存在）
<code>AttributeError</code>	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中变量虽然不需要在使用变量之前先声明类型，但也需要对变量进行赋值，然后才能使用。不被赋值的变量，不能再Python中存在，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
```

错误和异常(1)

你或许已经有足够的信心，貌似这样简单的错误在你的程序中是不会出现的，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。它只在程序运行之前出现。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和KeyError

```
>>> 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要对你的Computer进行全身搜查。所以，Python会按照你指定位置去找，找不到就异常。

AttributeError

错误和异常(1)

```
>>> class A(object): pass          #Python 3: class A: 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。

Python内建的异常也不仅仅上面几个，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.
#Python 3:
#TypeError: 'str' object cannot be interpreted as an integer
```

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

如果在程序运行过程中，出现了“抛出异常”的现象，程序就会中止运行。这样的程序是不“健壮”的，“健壮”的程序应该是不为各种“异常”所击倒，所以，要在程序里面对各种异常进行处理。

Python 2：

```
#!/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
```

Python 3 :

```

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

while 1:
    print("this is a division program.")
    c = input("input 'c' continue, otherwise logout:")
    if c == 'c':
        a = input("first number:")
        b = 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”，当异常出现时就会导致程序中止。

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 子句中有异常可，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。

错误和异常(1)

`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."          #Python 3: "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")
```

故意出现0做分母的情况。这时候 `is_raise = False`，则会：

错误和异常(1)

```
$ 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      #Python 3的信息略有差异
```

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](#) | [上节：生成器](#) | [下节：错误和异常\(2\)](#)

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

你们得救是本乎恩，也因着信。这并不是出于自己，乃是神所赐的；也不是出于行为，免得有人自夸。(EPHESIANS 2:8-9)

错误和异常(2)

处理多个异常

`try...except...` 是处理异常的基本方式。在此基础上，还可有扩展，能够处理多个异常。

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，由不同的 `except` 子句处理。

Python 2:

```
#!/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
```

Python 3:

错误和异常(2)

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

while 1:
    print("this is a division program.")
    c = input("input 'c' continue, otherwise logout:")
    if c == 'c':
        a = input("first number:")
        b = 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，try里面遇到一个异常，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 "*****"
```

错误和异常(2)

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。

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好。希望把它打印出来。但是程序还能不能中断，怎么办？

Python提供了一种方式，将上面代码修改如下：

Python 2:

```
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
```

Python 3:

```
while 1:  
    print("this is a division program.")  
    c = input("input 'c' continue, otherwise logout:")  
    if c == 'c':  
        a = input("first number:")  
        b = input("second number:")  
        try:  
            print(float(a)/float(b))  
            print("*****")  
        except (ZeroDivisionError, ValueError) as 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  
$
```

注意Python 3中的写法 `except (ZeroDivisionError, ValueError) as e:`

在上面程序中，只处理了两个异常，还可能有更多的异常，如果要处理，怎么办？可以这样：`execpt:` 或者 `except Exception, e`、`except Exception as 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。

错误和异常(2)

```
>>> try:  
...     print "I am try"          #Python 3: print("I am try"), 下同, 从略  
... 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，只是如果输入的有误，就不断要求重新输入，直到输入正确并得到了结果，才不再要求输入内容，然后程序结束。

在看下面的参考代码之前，读者是否可以先自己写一段并调试？看看结果如何。

Python 2：

```
#!/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
```

Python 3：

```
#!/usr/bin/env python
# coding=utf-8
while 1:
    try:
        x = input("the first number:")
        y = input("the second number:")

        r = float(x)/float(y)
        print(r)
    except Exception as 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。

程序中的 `except Exception, e` 或 `except Exception as e:` 的含义是不管什么异常，这里都会捕获，并且传给变量 `e`，然后用 `print e` 或者 `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`，最终都要执行它。因此一种说法是将 `finally` 用在可能的异常后进行清理。比如：

错误和异常(2)

```
>>> x = 10

>>> try:
...     x = 1/0
... except Exception, e:          #Python 3:  except Exception as e:
...     print e                  #Python 3: print(e)
... finally:
...     print "del x"           #Python 3:  print(e)
...     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
```

[总目录](#) | [上节：错误和异常\(1\)](#) | [下节：错误和异常\(3\)](#)

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

凡事谦虚、温柔、忍耐，用爱心互相宽容，用平和彼此联络，竭力保守圣灵所赐合而为一的心。(EPHESIANS 4:2-3)

错误和异常(3)

assert

从代码中理解 `assert`。

```
>>> assert 1==1
>>> assert 1==0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

`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，就抛出异常。

```
#!/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."      #Python 3: print("balance is not en
ough.")
```

上面的程序中，`deposit()` 和 `withdraw()` 方法的参数 `amount` 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，引用维基百科中对“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）：

错误和异常(3)

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，用于通知外界该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](#) | [上节：错误和异常\(2\)](#) | [下节：编写模块](#)

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

生气却不可犯罪，不可含怒到日落，也不可给魔鬼留地步。从前偷窃的，不要再偷。总要劳力，亲手作正经事，就可有余，分给那缺少的人。污秽的言语，一句不可出口，只要随事说造就人的好话，叫听见的人得益处。(EPHESIANS 4:26-29)

随着对Python学习的深入，其优点日渐突出，让读者也感觉到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（标准库），还有海量的第三方模块（或者包、库），并且很多开发者还在不断贡献自己开发的新模块（或者包、库），正是有了这么强大的“模块自信”，Python才被很多人钟爱。并且这种方式也正在不断被其它更多语言所借鉴，几乎成为普世行为了（不知道Python是不是首倡者）。

“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。

编写模块

想必读者已经熟悉了 import 语句，曾经有这样一个例子：

```
>>> import math  
>>> math.pow(3, 2)  
9.0
```

这里的 math 就是Python标准库中的一个，用import引入这个模块，然后可以使用它里面的函数（方法），比如这个 pow() 函数。显然，这里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。

请读者注意，我们会在实践中用到模块、库、包这些名词。它们有区别吗？有！只不过，现在我们暂时不区分，就笼统地说，阅读下面的内容，就理解它们之间的区分了。

模块是程序

“模块是程序”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侯将它引用过来，节省精力，不需要重写雷同的代码。

编写模块

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 import sys，不过由于 sys 是Python标准库之一，所以不用特别告诉Python解释器其位置。

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，请注意文件路径的写法。

```
>>> import pm
>>> pm.lang
'python'
```

本来在pm.py文件中有一个赋值语句，即 lang = "python"，现在将pm.py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”+“.”+“属性或类、方法名称”来访问变量 pm.py 中的东西。当然，如果不存在的，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，查看是不是多了一个扩展名是.pyc的文件？应该是。

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：[直譯器](#)

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

编写模块

本来复杂的多样化的世界非得划分为“黑白”的结果，喜欢用“非此即彼”的思维方式考虑问题，比如一提到“日本人”，除了苍老师，都该杀。这基本上是小孩子的思维方法，可惜在某国大行其道。

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。当然，苍老师是德艺双馨的，无可辩驳、毋庸置疑。

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](#)

有了 `.pyc` 文件后，每次运行就不需要重新让解释器来编译 `.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。

是否还记得前面写有关程序然后执行时常常要用到 `if __name__ == "__main__"`。那时我们直接用 `python filename.py` 的格式来运行该程序，此时我们也同样有了 `.py` 文件，不过是作为模块引入的。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。

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

def lang():
    return "python"

if __name__ == "__main__":
    print lang()      #Python 3: print(lang())
```

沿用先前的做法：

```
$ 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'
```

用 `dir()` 来查看它：

```
>>> 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"`，即属性 `__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，不必写 `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/pplib.py  
[sudo] password for qw:  
  
$ ls /usr/lib/python2.7/dist-packages/pm*  
/usr/lib/python2.7/dist-packages/pplib.py
```

文件放到了指定位置。看下面的：

```
>>> import pplib  
>>> pplib.lang  
<function lang at 0xb744372c>  
>>> pplib.lang()  
'python'
```

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。虽然这种方法在前面使用了，但其实是很不常用。因为它也有麻烦的地方，比如在交互模式下，如果关闭当前的terminal了，再开启（或者从新开启一个），还得重新告知。

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 PYTHONPATH = "$PYTHONPATH : /home/qw/python"`，然后保存退出即可。

环境变量更改之后，用户下次登录时生效，如果想立刻生效，则要执行下面的语句（此处感谢[Hsinwe](#)朋友的指正）：

```
$ source /etc/profile
```

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。

但是，问题并没有结束。正如[Hsinwe](#)所指出的那样，我上面的操作使进入了模块所在的目录，如果进入别的目录呢？能不能正常引入呢？这是一个非常好的问题，恭请各位读者来试一试。

__all__ 在模块中的作用

上面的模块虽然比较简单，但是已经显示了编写模块和在程序中导入模块的基本方式。在实践中，所编写的模块也许更复杂一点，比如，我写了这么一个模块，并把其文件命名为pp.py

```

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

public_variable = "Hello, I am a public variable."
_private_variable = "Hi, I am a private variable."

def public_teacher():
    print "I am a public teacher, I am from JP."      #Python 3: print("I am a public teacher, I am from JP.")

def _private_teacher():
    print "I am a private teacher, I am from CN."     #Python 3: print("I am a private teacher, I am from CN.")

```

接下来就是熟悉的操作了，进入到交互模式中。`pp.py` 这个文件就是一个模块，这个模块中包含了变量和函数。

```

>>> import sys
>>> sys.path.append("~/Documents/StarterLearningPython/2code/pp.py")
>>> import pp
>>> from pp import *
>>> public_variable
'Hello, I am a public variable.'
>>> _private_variable
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name '_private_variable' is not defined

```

编写模块

变量 `public_variable` 能够被使用，但是另外一个变量 `_private_variable` 不能被调用，先观察一下两者的区别，后者是以单下划线开头的，这样的是私有变量。而 `from pp import *` 的含义是“希望能访问模块（pp）中有权限访问的全部名称”，那些被视为私有的变量或者函数或者类，当然就没有权限被访问了。

再如：

```
>>> public_teacher()
I am a public teacher, I am from JP.
>>> _private_teacher()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_private_teacher' is not defined
```

这不是绝对的，如果要访问具有私有性质的东西，可以这样做啦。

```
>>> import pp
>>> pp._private_teacher()
I am a private teacher, I am from CN.
>>> pp._private_variable
'Hi, I am a private variable.'
```

下面再对 `pp.py` 文件改写，增加一点东西。

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

__all__ = ['_private_variable', 'public_teacher']

public_variable = "Hello, I am a public variable."
_private_variable = "Hi, I am a private variable."

def public_teacher():
    print "I am a public teacher, I am from JP."      #Python 3: print("I am a public te
acher, I am from JP.")

def _private_teacher():
    print "I am a private teacher, I am from CN."    #Python 3:  print("I am a private
teacher, I am from CN.")
```

在修改之后的 `pp.py` 中，增加了 `__all__` 属性以及相应的值，在列表中包含了一个私有变量的名字和一个函数的名字。这是在告诉引用本模块的解释器，这两个东西是有权限被访问的，而且只有这两个东西。

```
>>> import sys
>>> sys.path.append("~/Documents/StarterLearningPython/2code/pp.py")
>>> from pp import *
>>> _private_variable
'Hi, I am a private variable.'
```

果然，曾经不能被访问的私有变量，现在能够访问了。

```
>>> public_variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'public_variable' is not defined
```

因为这个变量没有在 `__all__` 的值中，虽然以前曾经被访问到过，但是现在就不行了。

```
>>> public_teacher()
I am a public teacher, I am from JP.
>>> _private_teacher()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_private_teacher' is not defined
```

这只不过是再次说明前面的结论罢了。当然，如果以 `import pp` 引入模块，再用 `pp._private_teacher` 的方式是一样有效的。

包或者库

顾名思义，包或者库，应该是比“模块”大的。也的确如此，一般来讲，一个“包”里面会有多个模块，当然，“库”是一个更大的概念了，比如Python标准库中的每个库都有好多个包，每个包都有若干个模块。

一个包是由多个模块组成，即多个 `.py` 的文件，那么这个所谓“包”也就是我们熟悉的一个目录罢了。现在就需要解决如何引用某个目录中的模块问题了。解决方法就是在该目录中放一个 `__init__.py` 文件。`__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。

例如，我建立了一个目录，名曰：`packageqi`，里面依次放了 `pm.py` 和 `pp.py` 两个文件，然后建立一个空文件 `__init__.py`。

接下来，我需要导入这个包 (`package_qi`) 中的模块。

下面这种方法，是很清晰明了的。

```
>>> import package_qi.pm
>>> package_qi.pm.lang()
'python'
```

另外一种方法，貌似简短，但如果多了，恐怕有点难以分辨了。

```
>>> from package_qi import pm
>>> pm.lang()
'python'
```

在后续制作网站的实战中，还会经常用到这种方式，届时会了解更多。请保持兴趣继续阅读，不要半途而废，不然疑惑得不到解决，好东西就看不到了。

[总目录](#) | [上节：错误和异常\(3\)](#) | [下节：标准库\(1\)](#)

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)

一切苦毒、恼恨、忿怒、嚷闹、毁谤，并一切的恶毒，都当从你们中间除掉。并要以恩慈相待，存怜悯的心，彼此饶恕，正如神在基督里饶恕了你们一样(EPHESIANS 4:31-32)

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”。

那些在安装Python时就默认已经安装好的模块被统称为“标准库”。

熟悉标准库编程之必须。

引用的方式

所有模块都服从下述引用方式，是最基本的、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，以标准库pprint为例。

以 pprint.pprint() 的方式使用模块中的一种方法，这种方法能够让字典格式化输出。看看结果是不是比原来更容易阅读了呢？

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，太多了看着头晕眼花，不容易阅读。

这是用 import pprint 样式引入模块，并以 . 点号（英文半角）的形式引用其方法。

还有下面的方式：

```
>>> from pprint import pprint
```

意思是从 `pprint` 模块中之将 `pprint()` 引入，之后就可以直接使用它了。

```
>>> 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模块中的所有可用的内容。但是，这样造成的结果是可读性不是很好，并且，不管是不在程序中用上，都拿过来，是不是太贪婪了？贪婪的结果是消耗内存。所以，这种方法，可以用于常用的并且模块属性或方法不是很多的情况。莫贪婪。

诚然，如果很明确使用模块中的哪些方法或属性，那么使用类似 `from modulename import name1, name2, name3...` 也未尝不可。需要再次提醒的是不能因为引入了模块而降低了可读性，让别人不知道呈现在眼前的方法是从何而来。

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，且要过了若干时间，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

标准库(1)

继续以 `pprint` 为例，深入研究：

```
>>> import pprint
>>> dir(pprint)
['PrettyPrinter', '_StringIO', '__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__', '_commajoin', '__id__', '__len__', '__perfcheck__', '__recursion__', '__safe_repr__', '__sorted__', '__sys__', '__type__', 'isreadable', 'isrecursive', 'pformat', ' pprint', 'saferrepr', 'warnings']
```

对 `dir()` 并不陌生。从结果中可以看到 `pprint` 的属性和方法。其中有有的是双划线、单划线开头的。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', 'saferrepr', 'warnings']
```

针对这几个，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)
```

前面是用 `import pprint` 方式引入模块的。

```
Help on function isreadable in module pprint:

isreadable(object)
    Determine if saferrepr(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）。

再回头看看 `dir(pprint)` 的结果：

```
>>> pprint.__all__
['pprint', 'pformat', 'isreadable', 'isrecursive', 'saferrepr', '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，事实上这些东西很少在引入模块时被使用。

帮助、文档和源码

你能记住每个模块的属性和方法吗？比如前面刚刚查询过的 `pprint` 模块中的属性和方法，现在能背诵出来吗？我的记忆力不行，都记不住。所以，我非常喜欢使用 `dir()` 和 `help()`，这也是本书从开始到现在，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？

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
```

标准库(1)

在这个文件的开始部分，所有类、方法和 `import` 之前，写一个用三个引号包括的字符串，这就是文档。

```
>>> import sys  
>>> sys.path.append("~/Documents/VBS/StarterLearningPython/2code")  
>>> import pm  
>>> print pm.__doc__ #Python 3: 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者的习惯。

对于Python的标准库和第三方模块，不仅要看帮助信息和文档，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__` 属性，它就告诉我们这个模块的位置：

```
>>> print pprint.__file__ #Python 3: 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，根据显示的目录，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源码。

事实证明，这种标准库中的源码是质量最好的。阅读高质量的代码，是提高编程水平的途径之一。

[总目录](#) | [上节：编写模块](#) | [下节：标准库\(2\)](#)

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

标准库(2)

所以你们要效法神，好像蒙慈爱的儿女一样。也要凭爱心行事，正如基督爱我们，为我们舍了自己，当作馨香的供物与祭物献与神。至于淫乱并一切污秽，或是贪婪，在你们中间连提都不可，方合圣徒的体统。(EPHESIANS 5:1-3)

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，我将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，前面已经使用过 `sys.path.append()`。

```
>>> import sys  
>>> print sys.__doc__
```

显示了 `sys` 的基本文档，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` 函数和属性中，选择常用的（应该说是我觉得常用的）来说明。

sys.argv

`sys.argv`是专门用来向python解释器传递参数，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 PYTHONDONTWRITEBYTECODE=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]          #Python 3的读者，请自行修改为print()函数形式，  
下同，从略
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`。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。

如果这样来试试：

标准库(2)

```
$ 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']
```

在这里用 `sys.argv[1]` 得到的就是 `beginner`，依次类推。

sys.exit()

这个方法的意思是退出当前程序。

```
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。

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

import sys

for i in range(10):
    if i == 5:
        sys.exit()
    else:
        print i      #Python 3: print(i)
```

这段程序的运行结果就是：

```
$ python 22102.py
0
1
2
3
4
```

在大多数函数中会用到 `return`，其含义是终止当前的函数，并向调用函数的位置返回相应值（如果没有就是 `None`）。但是 `sys.exit()` 的含义是退出当前程序——不仅仅是函数，并发起 `SystemExit` 异常。这就是两者的区别了。

标准库(2)

如果使用 `sys.exit(0)` 表示正常退出。若需要在退出的时候有一个对人友好的提示，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，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。不在赘述。

sys.stdout

`sys.stdin`，`sys.stdout`，`sys.stderr` 这三个有相似之处，它们所实现的都是类文件流，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中的函数功能对照，`sys.stdin` 获得输入（等价于Python 2中的`raw_input()`，Python 3中的`input()`），`sys.stdout` 负责输出。

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。

此处仅就 `sys.stdout` 做一个简要说明。

```
>>> for i in range(3):
...     print i          #Python 3: print(i)
...
0
1
2
```

`print` 语句或者函数，你一定很熟悉，此前用的很多。并且，特别说明，在默认情况下，不管是语句还是函数，最后都是有 `\n` 换行的。这种输入，本质上是用 `sys.stdout.write(object + '\n')` 实现。

```
>>> import sys
>>> for i in range(3):
...     sys.stdout.write(str(i))
...
012>>>
```

跟前面的不同，是因为没有加上那个 `\n`。

```
>>> 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"      #Python 3: print("Learn Pytho
n: 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
```

以上，对标准库中的 `sys` 有了粗浅的了解。更详细内容，请读者运用本书已经反复使用的 `dir()`、`help()` 去探究，当然还有Google。

copy

前面对浅拷贝和深拷贝做了研究，这里再次提出，温故知新。

```
>>> 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，这个例子跟以前讨论浅拷贝和深拷贝略有不同，请读者认真推敲结果，并对照代码。

标准库(2)

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

import copy

class MyCopy(object):    #Python 3: class MyCopy:
    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: {0}\n slice: {1}\n list(): {2}\n copy(): {3}\n deepcopy(): {4}\n".for
rmat(a,b,c,d,e)
#Python 3:
#print("original: {0}\n slice: {1}\n list(): {2}\n copy(): {3}\n deepcopy(): {4}\n".fo
rmat(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]
```

尽在不言中，请读者认真对照上面的显示结果，体会深拷贝和浅拷贝。

[总目录](#) | [上节：标准库\(1\)](#) | [下节：标准库\(3\)](#)

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

你们要谨慎行事，不要像愚昧人，当像智慧人。要爱惜光阴，因为现今的世代邪恶。不要作糊涂人，要明白主的旨意如何。不要醉酒，酒能使人放荡，乃要被圣灵充满。

(EPHESIANS 5:15-18)

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，有时候感觉很神秘。

```
>>> import os
>>> dir(os)
['EX_CANTCREATE', 'EX_CONFIG', 'EX_DATAERR', 'EX_IOERR', 'EX_NOHOST', 'EX_NOINPUT', 'EX_NOPERM', 'EX_NOUSER', 'EX_OK', 'EX_OSERR', 'EX_OSFILE', 'EX_PROTOCOL', 'EX_SOFTWARE', 'EX_TEMPFAIL', 'EX_UNAVAILABLE', 'EX_USAGE', 'F_OK', 'NGROUPS_MAX', 'O_APPEND', 'O_ASYNC', 'O_CREAT', 'O_DIRECT', 'O_DIRECTORY', 'O_DSYNC', 'O_EXCL', 'O_LARGEFILE', 'O_NDELAY', 'O_NOATIME', 'O_NOCTTY', 'O_NOFOLLOW', 'O_NONBLOCK', 'O_RDONLY', 'O_RDWR', 'O_RSYNC', 'O_SYNC', 'O_TRUNC', 'O_WRONLY', 'P_NOWAIT', 'P_NOWAITO', 'P_WAIT', 'R_OK', 'SEEK_CUR', 'SEEK_END', 'SEEK_SET', 'ST_APPEND', 'ST_MANDLOCK', 'ST_NOATIME', 'ST_NODEV', 'ST_NODIRATIME', 'ST_NOEXEC', 'ST_NOSUID', 'ST_RDONLY', 'ST_RELATIME', 'ST_SYNCHRONOUS', 'ST_WRITE', 'TMP_MAX', 'UserDict', 'WCONTINUED', 'WCOREDUMP', 'WEXITSTATUS', 'WIFCONTINUED', 'WIFEXITED', 'WIFSIGNALED', 'WIFSTOPPED', 'WNOHANG', 'WSTOPSIG', 'WTERMSIG', 'WUNTRACED', 'W_OK', 'X_OK', '__Environ', '__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__', '__copy_reg__', '__execvpe__', '__exists__', '__exit__', '__get_exports_list__', '__make_stat_result__', '__make_statvfs_result__', '__pickle_stat_result__', '__pickles_statvfs_result__', '__spawnvef__', 'abort', 'access', 'altsep', 'chdir', 'chmod', 'chown', 'chroot', 'close', 'closerange', 'confstr', 'confstr_names', 'ctermid', 'curdir', 'defpath', 'devnull', 'dup', 'dup2', 'environ', 'errno', 'error', 'execl', 'execle', 'execl', 'execle', 'execv', 'execve', 'execvp', 'execvpe', 'extsep', 'fchdir', 'fchmod', 'fchown', 'fdatasync', 'fdopen', 'fork', 'forkpty', 'fpathconf', 'fstat', 'fstatvfs', 'fsync', 'ftruncate', 'getcwd', 'getcwdu', 'getegid', 'getenv', 'geteuid', 'getgid', 'getgroups', 'getloadavg', 'getlogin', 'getpgid', 'getpgrp', 'getpid', 'getppid', 'getresgid', 'getresuid', 'getsid', 'getuid', 'initgroups', 'isatty', 'kill', 'killpg', 'lchown', 'linesep', 'link', 'listdir', 'lseek', 'lstat', 'major', 'makedev', 'makedirs', 'minor', 'mkdir', 'mkfifo', 'mknod', 'name', 'nice', 'open', 'openpty', 'pardir', 'path', 'pathconf', 'pathconf_names', 'pathsep', 'pipe', 'popen', 'popen2', 'popen3', 'popen4', 'putenv', 'read', 'readlink', 'remove', 'removedirs', 'rename', 'renames', 'rmdir', 'sep', 'setegid', 'seteuid', 'setgid', 'setgroups', 'setpgid', 'setpgrp', 'setregid', 'setresgid', 'setresuid', 'setreuid', 'setsid', 'setuid', 'spawnl', 'spawnle', 'spawnlpe', 'spawnnv', 'spawnve', 'spawnvp', 'spawnvpe', 'stat', 'stat_float_times', 'stat_result', 'statvfs', 'statvfs_result', 'sterror', 'symlink', 'sys', 'sysconf', 'sysconf_names', 'system', 'tcgetpgrp', 'tcsetpgrp', 'tempnam', 'times', 'tmpfile', 'tmpnam', 'ttynname', 'umask', 'uname', 'unlink', 'unsetenv', 'urandom', 'utime', 'wait', 'wait3', 'wait4', 'waitpid', 'walk', 'write']
```

这么多内容不能都介绍，列出来纯粹是要吓唬你一下，先混个脸熟，将来用到哪个了，可以到这里来找。

下面选择几个介绍一番，目的是不断强化我已经常用但是不知道你是否熟悉的学习方法，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

在对文件进行操作的时候，`open()` 这个内建函数可以打开文件。但是，如果对文件进行改名、删除操作，就要是用`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别的名称。

```
>>> 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称，第二个是打算修改成为的文件名。

然后查看，能够看到这个文件。

```
$ ls new*
newtemp.py
```

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）。

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。

```
Help on built-in function rename in module posix:

rename(...)
    rename(old, new)

    Rename a file or directory.
```

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)).
```

为了测试，先建立一些文件。

```
$ 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] Is a directory: '/home/qw/Documents/VBS/StarterLearningPython/2cod  
e/rd'
```

报错了。

我打算将这个目录下的所剩文件删光光，但这么做不行。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，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：显示目录中的内容（包括文件和子目录）。

```
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，且不显示目录中某些隐藏文件或子目录。

```
>>> 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：当前工作目录；**os.chdir**：改变当前工作目录

这两个函数怎么用？唯有通过 `help()` 看文档啦。请读者自行看看。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

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，从看文档开始一点一点地理解。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。下面把刚刚建立的这个目录删除了，毫无疑问它是空的。

```
>>> 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。但事实上行得通。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，再看当前的工作目录：

```
>>> 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，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？但显示就是这样。

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'
```

不存在的目录也被建立起来，直到最右边的目录为止。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

不管是哪种操作系统，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()`

标准库(3)

```
>>> p = os.getcwd()      #当前目录  
>>> p  
'/home/qw/Documents/VBS/StarterLearningPython'
```

显示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=1430224935, st_mtime=1430224935, st_ctime=1430224935)
```

再指定一个文件：

```
>>> pf = p + "/README.md"
```

显示此文件的信息：

```
>>> 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=1429580969, st_mtime=1429580969, st_ctime=1429580969)
```

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姑娘是友好的，但对读者可能不友好。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，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），或者在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，对于某些人（比如程序员）在某些情况下，命令是不错的选项，甚至是离不开的。

`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，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  221.md  index.md
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  222.md  n001.md
03.md      103.md  107.md  111.md  115.md  119.md  123.md  127.md  1code        203.md  2
07.md  211.md  215.md  219.md  2code      README.md
0images  104.md  108.md  112.md  116.md  120.md  124.md  128.md  1images  204.md  208.
md  212.md  216.md  220.md  2images
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，读者可以查阅有关专门资料。

`os.system()` 是一个用途很多的方法。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 initialised

.....
```

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系统，那么就要非常小心了，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 \ 代表转义。比较简单的一个方法是用 `r"c:\user\firfox.exe"` 的样式，因为在 `r" "` 中的，都被认为是原始字符。而且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。读者按照这些提示，看看能不能完成用 `os.system()` 启动firefox的操作。

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。

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](#) | [上节：标准库\(2\)](#) | [下节：标准库\(4\)](#)

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)

凡事不可结党，不可贪图虚浮的荣耀，只要存心谦卑，各人看别人比自己强。各人不要单顾自己的事，也要顾别人的事。(PHILIPPIANS 2:3-4)

标准库(4)

heapq

堆 (heap) ，是一种数据结构，引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出来的内容。

堆的实现是通过构造二叉堆，也就是一种二叉树。

基础知识

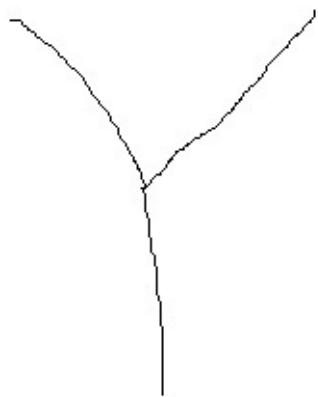
这是一棵在苏州很常见的香樟树，马路两边、公园里随处可见，特别是在艳阳高照的时候，它的树荫能把路面遮盖。



但是，在编程中，我们常说的树是这样的：

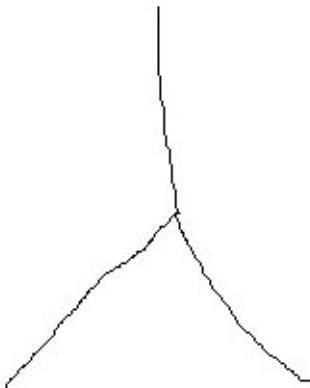


这是一棵“根”在上面树，也是编程中常说的树。为什么这样呢？我想主要是画着更方便吧。上面那棵树虽然根在上面了，还完全是写实的作品，本人做为一名隐姓埋名多年的抽象派画家，不喜欢这样的树，我画出来的是这样的：

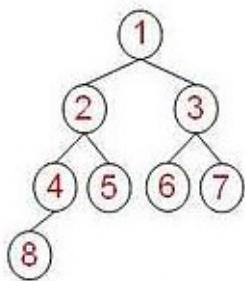


这棵树有两根枝权，可不要小看这两根枝权哦，《道德经》上不是说“一生二，二生三，三生万物”。一就是下面那个干，二就是两个枝权，每个枝权还可以看做下一个一，然后再有两个枝权，如此不断重复（这简直就是递归呀），就成为了一棵大树。

这棵树画成这样就更符合编程的习惯了，可以向下不断延伸。



并且给它一个正规的名字：二叉树。

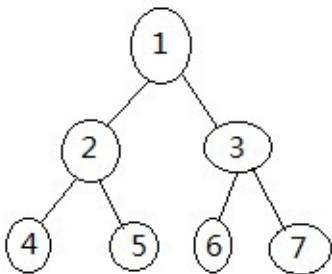


这个也是二叉树，完全脱胎于我所画的后现代抽象主义作品。但是略有不同，这幅图在各个枝权上显示的是数字。这种类型的“树”就编程语言中所说的二叉树，维基百科曰：

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”。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：

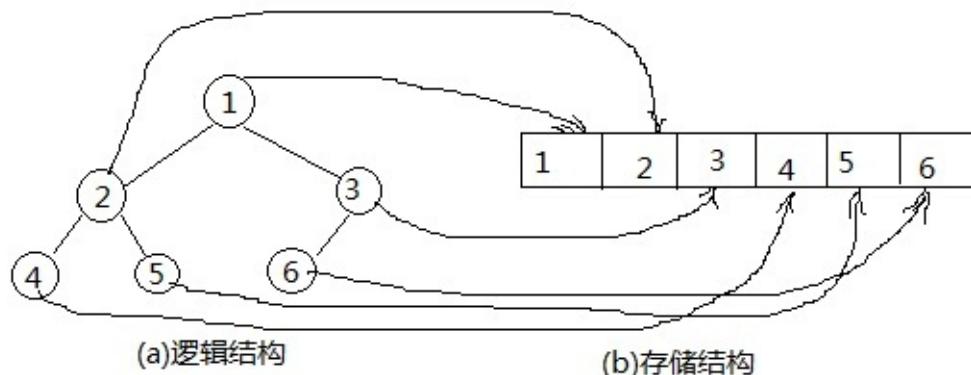


下面讨论的对象是通过二叉树实现的，其具有如下特点：

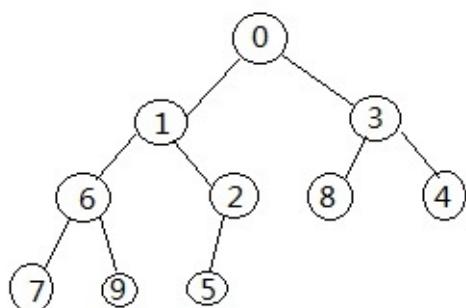
- 节点的值大于等于（或者小于等于）任何子节点的值。
- 节点左子树和右子树是一个二叉堆。如果父节点的值总大于等于任何一个子节点的值，其为最大堆；若父节点值总小于等于子节点值，为最小堆。上面图示中的完全二叉树，就表示一个最小堆。

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', 'heappreplace', 'merge', 'nlargest', 'nsmallest', 'heappushpop']
```

依次查看这些函数的使用方法。

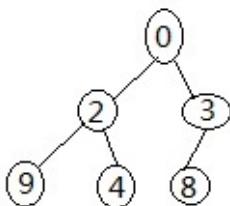
heappush(heap, x)：将x压入堆heap

```
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]
```

请读者注意上面的操作，在向堆增加数值的时候并没有严格按照什么顺序，是随意的。但是，当查看堆的数据时，显示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，看看是不是一棵二叉树。



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。

标准库(4)

```
>>> h1 = [2, 4, 6, 8, 9, 0, 1, 5, 3]
>>> heapq.heapify(h1)
>>> h1
[0, 3, 1, 4, 9, 6, 2, 5, 8]
```

经过这样的操作，列表 `h1` 就变成了堆（堆的顺序和列表不同），可以对 `h1`（堆）使用 `heappop()` 或者 `heappush()` 等函数了。否则，不可。

```
>>> heapq.heappop(h1)
0
>>> heapq.heappop(h1)
1
>>> h1
[2, 3, 5, 4, 9, 6, 8]
>>> heapq.heappush(h1, 9)
>>> h1
[2, 3, 5, 4, 9, 6, 8, 9]
```

不要认为堆里面只能放数字，举例中之所以用数字，是因为对它的逻辑结构比较好理解。

```
>>> heapq.heappush(h1, "q")
>>> h1
[2, 3, 5, 4, 9, 6, 8, 9, 'q']
>>> heapq.heappush(h1, "w")
>>> h1
[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有哪些呢？排序是一个应用方面。一提到排序，读者肯定想到的是 `sorted()` 或者列表中的 `sort()`，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。但如果使用堆排序，相对于其他排序，也有自己的优势。不同的排序方法有不同的特点，读者可以自行深入研究不同排序的优劣。

deque

有这样一个问题：一个列表，比如是 `[1, 2, 3]`，在最右边增加一个数字。

标准库(4)

这也太简单了，不就是用 `append()` 追加一个吗？

这是简单。但，得寸进尺，能不能在最左边增加一个数字呢？

这个应该有办法，不过得想想了。读者在向下阅读之前候，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是因为 `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字。

```
>>> qlst = deque(lst)
```

这是必需的，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])
```

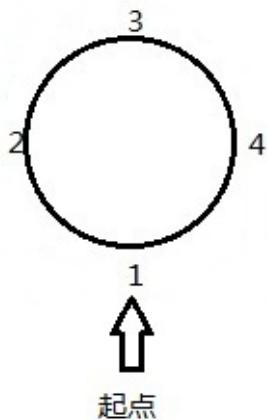
这样操作多么容易呀。继续看删除：

```
>>> qlst.pop()
5
>>> qlst
deque([7, 1, 2, 3, 4])
>>> qlst.popleft()
7
>>> qlst
deque([1, 2, 3, 4])
```

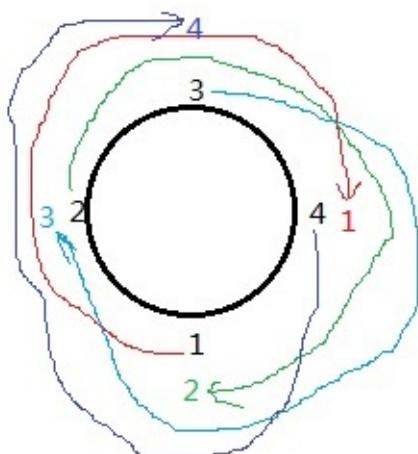
删除也分左右。下面这个，请读者仔细观察。

```
>>> qlst.rotate(3)
>>> qlst
deque([2, 3, 4, 1])
```

`rotate()`的功能是将`[1, 2, 3, 4]`的首尾连起来，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])
```

如果参数是负数，那么就逆时针转。

在`deque`中，还有 `extend()` 和 `extendleft()` 方法。读者可自己调试。

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

凡所行的，都不要发怨言、起争论，使你们无可指摘，诚实无伪，在这弯曲悖谬的世代作神无瑕疵的儿女。你们显在这世代中，好像明光照耀，将生命的道表明出来。
(PHILIPPIANS 2:14-15)

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”这个术语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`年的年历，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

Mo	Tu	We	Th	Fr	Sa	Su
				1	2	3
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	

February

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	

March

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

April

Mo	Tu	We	Th	Fr	Sa	Su
				1	2	3
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			

May

Mo	Tu	We	Th	Fr	Sa	Su
					1	2
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

June

Mo	Tu	We	Th	Fr	Sa	Su
				1	2	3
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
				29	30	

July

Mo	Tu	We	Th	Fr	Sa	Su
				1	2	3
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		

August

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		

September

Mo	Tu	We	Th	Fr	Sa	Su
				1	2	3
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

October

Mo	Tu	We	Th	Fr	Sa	Su
				1	2	3
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	

November

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		

December

Mo	Tu	We	Th	Fr	Sa	Su
				1	2	3
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			

其它部分就是按照上面的样式，将2015年度的各个月份日历完全显示出来。

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)

标准库(5)

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。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。

monthrange(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.monthrange(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 模块很常用，比如记录某个程序运行时间的长短等，下面一道来其中的方法。

- `time()`

```
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時間，或稱POSIX時間是UNIX或類UNIX系統使用的時間表示方式：從協調世界時 1970年1月1日0時0分0秒起至現在的總秒數，不考慮閏秒

現時大部分使用UNIX的系統都是32位元的，即它們會以32位二進制數字表示時間。但是它們最多只能表示至協調世界時間2038年1月19日3時14分07秒（二進制：01111111 11111111 11111111 11111111，0x7FFF:FFFF），在下一秒二進制數字會是10000000 00000000 00000000，（0x8000:0000），這是負數，因此各系統會把時間誤解作1901年12月13日20時45分52秒（亦有說回歸到1970年）。這時可能會令軟體發生問題，導致系統癱瘓。

目前的解決方案是把系統由32位元轉為64位元系統。在64位系統下，此時間最多可以表示到292,277,026,596年12月4日15時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=124, tm_isdst=0)
```

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：

标准库(5)

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
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, tm_isdst=0)
```

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=124, tm_isdst=0)
```

格林威治標準時間（中國大陸翻譯：格林尼治平均時間或格林尼治標準時間，台、港、澳翻譯：格林威治標準時間；英语：Greenwich Mean Time，GMT）是指位於英國倫敦郊區的皇家格林威治天文台的標準時間，因為本初子午線被定義在通過那裡的經線。

还有更友好的，请继续阅读。

asctime()

标准库(5)

```
>>> 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(10000000)
>>> 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=12, tm_isdst=0)
>>> 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(10000000)
'Mon Jan 12 21:46:40 1970'
```

跟前面得到的结果是一样的，只不过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。

mkttime()

`mkttime()` 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, t
m_wday=1, tm_yday=125, tm_isdst=0)
>>> time.mkttime(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）。

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。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）
%y	去掉世纪的年份	00-99，如"15"
%Y	完整的年份	如"2015"
%j	指定日期是一年中的第几天	001-366
%m	返回月份	01-12
%b	本地简化月份的名称	简写英文月份
%B	本地完整月份的名称	完整英文月份
%d	该月的第几日	如5月1日返回"01"
%H	该日的第几时（24小时制）	00-23
%I	该日的第几时（12小时制）	01-12
%M	分钟	00-59
%S	秒	00-59
%U	在该年中的第多少星期（以周日为一周起点）	00-53
%W	同上，只不过是以周一为起点	00-53
%w	一星期中的第几天	0-6
%Z	时区	在中国大陆测试，返回CST，即China Standard Time
%x	日期	日/月/年
%X	时间	时:分:秒
%c	详细日期时间	日/月/年 时:分:秒
%%	'%'字符	'%'字符
%p	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, tm_isdst=-1)
```

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，有时调用起来感觉不是很直接，于是又出来了一个 `datetime` 模块，供程序猿和程序员们选择使用。

`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)
```

其实这里生成了一个日期对象，然后操作这个对象的各种属性。可以用 `print`，以获得更佳视觉：

```
>>> print today      #Python 3: print(today)
2015-05-05
>>> print today.ctime()      #Python 3: print(today.ctime())
Tue May  5 00:00:00 2015
>>> print today.timetuple()      #Python 3: 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, tm_isdst=-1)
>>> print today.toordinal()      #Python 3: 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          #Python 3: 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。

对 `now` 增加5个小时；

```
>>> b = now + datetime.timedelta(hours=5)
>>> print b          #Python 3: print(b)
2015-05-05 14:22:43.142520
```

增加两周；

```
>>> c = now + datetime.timedelta(weeks=2)
>>> print c          #Python 3: print(c)
2015-05-19 09:22:43.142520
```

计算时间差：

```
>>> d = c - b
>>> print d          #Python 3: print(d)
13 days, 19:00:00
```

[总目录](#) | [上节：标准库\(4\)](#) | [下节：标准库\(6\)](#)

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

弟兄们，我不是以为自己已经得着了，我只有一件事，就是忘记背后，努力面前的，向着标杆直跑，要得神在基督耶稣里从上面召我来得的奖赏。(PHILIPPIANS 3:13-14)

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：

在Python 2中，这样操作：

```
>>> import urllib  
>>> itdiffer = urllib.urlopen("http://www.itdiffer.com")
```

但是如果读者使用的是Python 3，必须换个姿势：

```
>>> import urllib.request  
>>> itdiffer = urllib.request.urlopen("http://www.itdiffer.com")
```

这样就已经把我的网站www.itdiffer.com首页的内容拿过来了，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。

```
>>> print itdiffer.read()          #Python 3: print(itdiffer.read())  
<!DOCTYPE HTML>  
<html>  
  <head>  
    <title>I am Qiwsir</title>  
  ....//因为内容太多，下面就省略了
```

这样就完成了对网页的抓取。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。

Python 2:

```
>>> dir(urllib)
['ContentTooShortError', 'FancyURLopener', 'MAXFTPCACHE', 'URLopener', '__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__', '__version__', '_asciire', '_ftpliberrors', '_have_ssl', '_hexdig', '_hextochr', '_hostprog', '_is_unicode', '_localhost', '_noheaders', '_nportprog', '_passwdprog', '_portprog', '_queryprog', '_safe_map', '_safe_quoters', '_tagprog', '_tishost', '_typeprog', '_urlopener', '_userprog', '_valueprog', 'addbase', 'addclosehook', 'addinfo', 'addinfourl', 'always_safe', 'base64', 'basejoin', 'c', 'ftpcache', 'ftpliberrors', 'ftpwrapper', 'getproxies', 'getproxies_environment', 'i', 'localhost', 'noheaders', 'os', 'pathname2url', 'proxy_bypass', 'proxy_bypass_environment', 'quote', 'quote_plus', 're', 'reporthook', 'socket', 'splitattr', 'splithost', 'splitnport', 'splitpasswd', 'splitport', 'splitquery', 'splittag', 'splittype', 'splituser', 'splitvalue', 'ssl', 'string', 'sys', 'test1', 'tishost', 'time', 'toBytes', 'unquote', 'unquote_plus', 'unwrap', 'url2pathname', 'urlopen', 'urlencode', 'urlretrieve']
```

Python 3:

```
>>> dir(urllib.request)
['AbstractBasicAuthHandler', 'AbstractDigestAuthHandler', 'AbstractHTTPHandler', 'BaseHandler', 'CacheFTPHandler', 'ContentTooShortError', 'DataHandler', 'FTPHandler', 'FancyURLopener', 'FileHandler', 'HTTPBasicAuthHandler', 'HTTPCookieProcessor', 'HTTPDefaultErrorHandler', 'HTTPDigestAuthHandler', 'HTTPError', 'HTTPErrorProcessor', 'HTTPHandler', 'HTTPPasswordMgr', 'HTTPPasswordMgrWithDefaultRealm', 'HTTPPasswordMgrWithPriorAuth', 'HTTPRedirectHandler', 'HTTPSHandler', 'MAXFTPCACHE', 'OpenerDirector', 'ProxyBasicAuthHandler', 'ProxyDigestAuthHandler', 'ProxyHandler', 'Request', 'URLError', 'URLopener', 'UnknownHandler', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '__version__', '__cut_port_re', '__ftpliberrors', '__have_ssl', '__localhost', '__noheaders', '__opener', '__parse_proxy', '__proxy_bypass_macosx_sysconf', '__randombytes', '__safe_gethostname', '__tishost', '__url_tempfiles', 'addclosehook', 'addinfourl', 'base64', 'bisect', 'build_opener', 'collections', 'contextlib', 'email', 'ftpcache', 'ftpliberrors', 'ftpwrapper', 'getproxies', 'getproxies_environment', 'getproxies_registry', 'hashlib', 'http', 'install_opener', 'io', 'localhost', 'noheaders', 'os', 'parse_http_list', 'parse_keqv_list', 'pathname2url', 'posixpath', 'proxy_bypass', 'proxy_bypass_environment', 'proxy_bypass_registry', 'quote', 're', 'request_host', 'socket', 'splitattr', 'splithost', 'splitpasswd', 'splitport', 'splitquery', 'splittag', 'splittype', 'splituser', 'splitvalue', 'ssl', 'sys', 'tempfile', 'tishost', 'time', 'to_bytes', 'unquote', 'unquote_to_bytes', 'unwrap', 'url2pathname', 'urlopen', 'urljoin', 'urlparse', 'urlretrieve', 'urllsplit', 'urlunparse', 'warnings']
```

选几个常用的介绍，如果读者用到其它的，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，然后就如同在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.
```

查看文档信息，在Python 2下使用 `help(urllib.urlopen)`，在Python 3下使用 `help(urllib.request.urlopen)`。两者查询结果略有差异，上述显示的是Python 2下的查询结果。

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，即变量 `itdiffer` 引用了得到的类文件对象，依然可以用老办法 `dir(itdiffer)` 查看它的属性和方法，但在不同的Python版本下，显示结果是有所不同的，区别的原因是两个版本对文件对象的不同处理。

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的编码要求，要对`url`进行编码和解码。在进行web开发的时候特别要注意。`urllib` 或者 `urllib.request` 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。下面的操作是在Python 2中进行的，

```
>>> 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'
```

如果是Python 3的读者，请注意，该方法不在前述所引用的 `urllib.request` 中，尽管它里面有 `quote()` 方法，但最好的操作是 `import urllib.parse`，所以，Python 3下应该这么操作：

```
>>> import urllib.parse
>>> du = 'http://www.itdiffer.com/name=python book'
>>> urllib.parse.quote(du)
'http%3A//www.itdiffer.com/name%3Dpython%20book'
>>> urllib.parse.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：

Google search results for '零基础 python':

- 《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。

Python 2:

```
>>> dup = urllib.quote_plus(du)
>>> urllib.unquote_plus(dup)
'http://www.itdiffer.com/name=python book'
```

Python 3:

```
>>> dup = urllib.parse.quote_plus(du)
>>> urllib.parse.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。

Python 2:

```
>>> urllib.urlencode({"name":"qiwsir","web":"itdiffer.com"})
'web=itdiffer.com&name=qiwsir'
```

Python 3:

```
>>> urllib.parse.urlencode({"name":"qiwsir","web":"itdiffer.com"})
'name=qiwsir&web=itdiffer.com'
```

如果将来你要做一个网站，上面的方法或许会用到。

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。

以下是在Python 2中的操作：

```
>>> import urllib
>>> urllib.urlretrieve("http://www.itdiffer.com/images/me.jpg", "me.jpg")
('me.jpg', <httplib.HTTPMessage instance at 0xb6ecb6cc>)
```

如果在Python 3中，则要使用 `urllib.request`：

```
>>> import urllib.request
>>> urllib.request.urlretrieve("http://www.itdiffer.com/images/me.jpg", "me.jpg")
('me.jpg', <http.client.HTTPMessage object at 0x000000000395A160>)
```

`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（[Python 2文档](#)，[Python 3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

标准库(6)

- **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
#Python 3
#import urllib.request

def go(a,b,c):
    per = 100.0 * a * b / c
    if per > 100:
        per = 100
    print "%.2f%%" % per

url = "http://ww2.sinaimg.cn/mw690/8e4023f8gw1f34gs20b4ij20qo0zkthw.jpg"
local = "/home/qw/Pictures/g.jpg"
urllib.urlretrieve(url, local, go)
#Python 3
#urllib.request.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，读者自行查看（或许在本书出版的时候，这张什么的图片已经看不到了，你应该把这视为正常现象，可以换一张图片地址）。

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', 'FTPHandler', 'FileHandler', 'HTTPBasicAuthHandler', 'HTTPCookieProcessor', 'HTTPDefaultErrorHandler', 'HTTPDigestAuthHandler', 'HTTPError', 'HTTPErrorProcessor', 'HTTPHandler', 'HTTPPPasswordMgr', 'HTTPPPasswordMgrWithDefaultRealm', 'HTTPRedirectHandler', 'HTTPSShandler', 'OpenerDirector', 'ProxyBasicAuthHandler', 'ProxyDigestAuthHandler', 'ProxyHandler', 'Request', 'StringIO', 'URLError', 'UnknownHandler', '__builtins__', '__doc__', '__file__', '__name__', '__package__', '__version__', '__cut_port_re', '__opener__', '__parse_proxy__', '__safe_gethostbyname__', 'addinfourl', 'base64', 'bisect', 'build_opener', 'ftplibwrapper', 'getproxies', 'hashlib', 'httplib', 'install_opener', 'localhost', 'mimetypes', 'os', 'parse_http_list', 'parse_keqv_list', 'posixpath', 'proxy_bypass', 'quote', 'random', 'randombytes', 're', 'request_host', 'socket', 'splitattr', 'splithost', 'splitpasswd', 'splitport', 'splittag', 'splittype', 'splituser', 'splitvalue', 'sys', 'time', 'toBytes', 'unquote', 'unwrap', 'url2pathname', 'urlopen', 'urlparse', 'warnings']
```

比较常用的比如 `urlopen()` 跟 `urllib.urlopen()` 是完全类似的。

但是，要注意，上述言论仅仅是针对Python 2的，在Python 3中，已经没有 `urllib2` 这个模块了，取代它的是 `urllib.request`。

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：

Python 2:

```
>>> req = urllib2.Request("http://www.itdiffer.com")
```

Python 3:

标准库(6)

```
>>> import urllib.request  
>>> req = urllib.request.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

Python 2:

```
>>> response = urllib2.urlopen(req)  
>>> page = response.read()  
>>> print page
```

Python 3:

```
>>> response = urllib.request.urlopen(req)  
>>> page = response.read()  
>>> print(page)
```

显示结果从略。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。

Python 2:

```
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 = urllib2.urlopen(req)     #接受反馈的信息  
the_page = response.read()         #读取反馈的内容
```

Python 3:

```
import urllib.request  
import urllib.parse  
  
url = 'http://www.itdiffer.com/register.py'  
  
values = {'name' : 'qiwsir', 'location' : 'China', 'language' : 'Python' }  
  
data = urllib.parse.urlencode(values)      # 编码  
req = urllib.request.Request(url, data)    # 发送请求同时传data表单  
response = urllib.request.urlopen(req)     #接受反馈的信息  
the_page = response.read()               #读取反馈的内容
```

标准库(6)

如果读者照抄上面的程序，然后运行代码，肯定报错。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，为了让程序运行，读者可以开发接受数据的程序。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，并且在这个例子中是以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)      #Python 3: req = urllib.request.Request(u  
rl, data, headers)
```

再用·urlopen()·方法访问：

```
response = urllib2.urlopen(req)                #Python 3: response = urllib.request.urlopen(re  
q)
```

除了上面演示之外，urllib2 或者 urllib.request 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](#) | [上节：标准库\(5\)](#) | [下节：标准库\(7\)](#)

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

你们要靠主常常喜乐；我再说，你们要喜乐。当叫众人知道你们谦让的心。主已经近了。应当一无挂虑，只要凡事藉着祷告、祈求和感谢，将你们所要的告诉神。神所赐出人意外的平安，比在基督耶稣里面保守你们的心怀意念。(PHILIPPIANS 4:4-7)

标准库(7)

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

如果要对XML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XML，可以阅读[w3school的教程](#)

XML的重要在于它是用来传输数据的，因此，特别是在web编程中，经常要用到的。有了它让数据传输变得简单了。这么重要，Python当然支持。

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研究。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`。

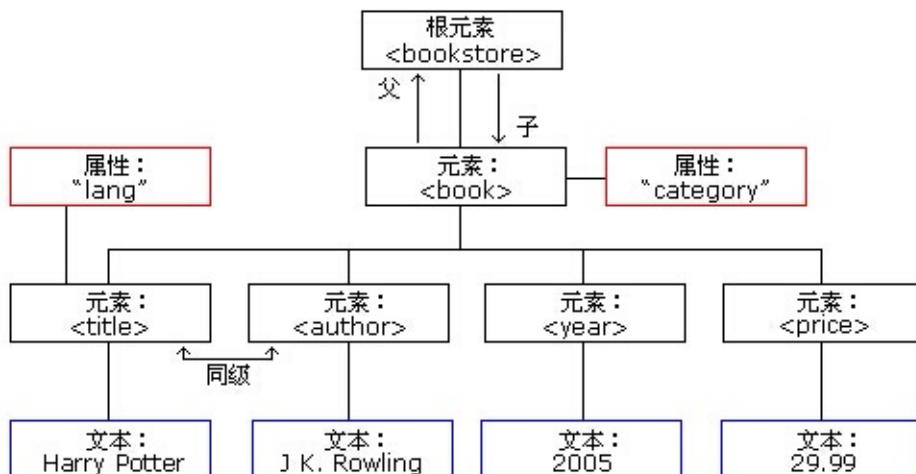
如果读者使用的是Python 2，可以像这样引入模块：

```
try:
    import xml.etree.cElementTree as ET
except ImportError:
    import xml.etree.ElementTree as ET
```

如果是Python 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Python 3相对Python 2有了很大进步。

遍历查询

先要做一个XML文档。图省事，就用w3school中的一个例子：



这是一棵树，先把这棵树写成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>

```

将其保存并命名为22601.xml的文件，接下来就是以它为对象，练习各种招数了。

```
>>> import xml.etree.ElementTree as ET
```

如果读者使用Python 2，推荐使用如前所述的 `try...except...` 方式引入模块；如果是Python 3，按照刚才的方式引入即可。

```

>>> tree = ET.ElementTree(file="22601.xml")
>>> tree
<ElementTree object at 0xb724cc2c>

```

建立起XML解析树对象。然后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      #Python 3: 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'
```

再深入一层，就有内容了：

```
>>> 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      #Python 3: 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      #Python 3: 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遍。

```
>>> for ele in tree.iter():
...     print ele.tag, ele.attrib      #Python 3: 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      #Python 3: print(ele.text)
...
Everyday Italian
Harry Potter
Learning XML
```

利用 `.findall()` 方法，也可以是实现查找功能：

```
>>> 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      #Python 3: 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      #Python 3: 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，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      #Python 3: 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.ElementTree 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      #Python 3: 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)

```

标准库(7)

```
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 = ""
        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": "BProcesser"})

#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 firest conte
nt")

#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/ch
ain"), {"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>
        <processer name="AProcesser" file="lib64/A.so"
            path="/tmp">
        </processer>
        <processer name="BProcesser" file="lib64/B.so" value="fordelete">
        </processer>
        <processer name="BProcesser" file="lib64/B.so22222222"/>

    <services>
        <service name="search" prefix="/bin/search?">
            output_formatter="OutPutFormatter:service_inc">

                <chain sequence="chain1"/>
                <chain sequence="chain2"></chain>
            </service>
        <service name="update" prefix="/bin/update?">
            <chain sequence="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 first content</person>
        </processer>
        <processer age="1" file="lib64/B.so22222222" name="BProcesser">
            <person age="15" money="200000">this is the first content</person>
        </processer>

    <services>
        <service name="search" output_formatter="OutPutFormatter:service_inc"
            prefix="/bin/search?">

            <chain sequence="chain2" />
        </service>
        <service name="update" prefix="/bin/update?">
            <chain sequence="chain3" value="fordelete">new text</chain>
        </service>
    </services>
    </processors>
</framework>

```

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

所以你们既是神的选民，圣洁蒙爱的人，就要存怜悯、恩慈、谦虚、温柔、忍耐的心。倘若这人与那人有嫌隙，总要彼此包容，彼此饶恕；主怎么饶恕了你们，你们也要怎样饶恕人。在这一切之外，要存着爱心，爱心就是联络全德的。又要叫基督的平安在你们心里作主，你们也为此蒙召，归为一体，且要存感谢的心。(COLOSSIANS 3:12-15)

标准库(8)

JSON

就传递数据而言，XML是一种选择，还有另外一种——JSON，它是一种轻量级的数据交换格式，如果读者要做web编程，则会用到它的。根据维基百科的相关内容，对JSON做如下简介：

JSON（JavaScript Object Notation）是一種由道格拉斯·克羅克福特構想設計、輕量級的資料交換語言，以文字為基礎，且易於讓人閱讀。儘管JSON是Javascript的一個子集，但JSON是獨立於語言的文本格式，並且採用了類似於C語言家族的一些習慣。

关于JSON更为详细的内容，可以参考其官方网站：<http://www.json.org>

从上述网站摘取部分内容，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']
#Python 3的显示结果如下：
['dump', 'dumps', 'load', 'loads', 'JSONDecoder', 'JSONDecodeError', 'JSONEncoder']
```

encoding: dumps()

```
>>> data = [{"name": "qiwsir", "lang": ("python", "english"), "age": 40}]
>>> data
[{"lang": ('python', 'english'), 'age': 40, 'name': 'qiwsir'}]
>>> data_json = json.dumps(data)
>>> data_json
'[{"lang": ["python", "english"], "name": "qiwsir", "age": 40}]'
```

`encoding`的操作是比较简单的，请注意观察 `data` 和 `data_json` 的不同——`lang`的值从元组变成了列表，还有不同：

```
>>> type(data_json)
<type 'str'>
>>> type(data)
<type 'list'>
```

decoding: loads()

`decoding`的过程也像上面一样简单：

```
>>> new_data = json.loads(data_json)
>>> new_data
[{"lang": ["python", "english"], "age": 40, "name": "qiwsir"}]
```

需要注意的是，解码之后，并没有将元组还原。

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          #Python 3: 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。但现在是所谓“大数据”时代了，随便一个什么业务都在说自己是大数据，显然不能总让JSON很小，事实上真正的大数据，再“大”的JSON也不行了。前面的操作方法是将数据都读入内存，如果数据太大了就会内存溢出。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()          #Python 3: 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 a
    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

```


你们要爱惜光阴，用智慧与外人交往。你们的言语要常常带着和气，好像用盐调和，就可知道怎样回答各人。(COLOSSIANS 4:5-6)

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Pythoner，即使你达到了master的水平，在做某个事情之前最好在网上搜一下是否有标准库或者第三方库替你完成。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

安装第三方库的方法有几种，不同方法有不同的优缺点，读者可以根据自己的喜好或者实际的工作情景来选择。

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（注意：github不是源码的唯一来源，只不过很多源码都在这个网站上，我也喜欢罢了），得到源码之后，在本地安装。

如果你下载的是一个文件包，即得到的源码格式为zip、tar.zip、tar.bz2的压缩文件，需要先解压缩，然后进入其目录；如果你能熟练使用git命令，可以直接从github中clone源码到本地计算机上，然后进入该目录。

通常会看见一个setup.py的文件。

```
python setup.py install
```

在这里可能对某些操作系统的读者就漠视了，因为我用的是Ubuntu，读者可以根据自己的操作系统确定安装方法。

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。

这种安装方法有时候麻烦一些，但是比较灵活，主要体现在：

- 可以下载安装自己选定的任意版本的第三方库，比如最新版，或者更早的某个版本，所以在某些有特殊需要的时候，常常使用这种方式安装。
- 通过安装设置可以指定安装目录，自由度比较高。

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。网站的地址：<https://pypi.python.org/pypi>

pip是一个以Python计算机程序语言写成的软件包管理系统，它可以安装和管理软件包，另外不少的软件包也可以在“Python软件包索引”（英语：Python Package Index，简称PyPI）中找到。（源自《维基百科》）

首先，要安装pip。读者可以先检查一下，在你的操作系统中是否已经有了pip，因为有的操作系统，或者已经预先安装了，或者在安装Python的时候安装了。如果你确信没有安装，就要针对你的操作系统进行安装，例如在Ubuntu中：

Python 2：

```
sudo apt-get install python-pip
```

Python 3：

```
sudo apt-get install python3-pip
```

当然，也可以这里下载文件[get-pip.py](https://bootstrap.pypa.io/get-pip.py)，然后执行 `python get-pip.py` 来安装。这个方法也适用于windows。

pip就这样安装好了，非常简单吧。

然后你就可以淋漓尽致地安装第三方库了，之所以如此，是因为只需要执行 `pip install XXXXXX` (`XXXXXX`代表第三方库的名字) 即可。当然前提是那个库已经在PyPI里面了。

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

Python 2 :

```
pip install requests
```

Python 3 :

```
pip3 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', 'Timeout', 'TooManyRedirects', 'URLRequired', '__author__', '__build__', '__builtins__', '__copyright__', '__doc__', '__file__', '__license__', '__name__', '__package__', '__path__', '__title__', '__version__', 'adapters', 'api', 'auth', 'certs', 'codes', 'compat', 'cookies', 'delete', 'exceptions', 'get', 'head', 'hooks', 'logging', 'models', 'options', 'packages', 'patch', 'post', 'put', 'request', 'session', 'sessions', 'status_codes', 'structures', 'utils']
```

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='buqj70k7f9rrg51emsatveda2', port=None, port_specified=False, domain='www.1world0x00.com', domain_specified=False, domain_initial_dot=False, path '/', path_specified=True, secure=False, expires=None, discard=True, comment=None, comment_url=None, rest={}, rfc2109=False)]>
```

仔细观察，是不是看到了cookie的name和value，结合对网络有关知识的理解，是不是有一种豁然开朗恍然大悟的感觉？

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; path=/', 'expires': 'Thu, 19 Nov 1981 08:52:00 GMT', 'keep-alive': 'timeout=15, max=500', 'server': 'Apache/2.2.15 (CentOS)', 'connection': 'Keep-Alive', 'pragma': 'no-cache', 'cache-control': 'no-store, no-cache, must-revalidate', 'post-check=0, pre-check=0', 'date': 'Mon, 10 Nov 2014 01:39:03 GMT', 'content-type': 'text/html; charset=UTF-8', 'x-pingback': 'http://www.1world0x00.com/index.php?action/xmlrpc'}
```

```
>>> r.encoding
'UTF-8'
```

```
>>> r.status_code
200
```

这些都是在客户端看到的网页基本属性。

下面这个比较长，是网页的内容，仅仅截取部分显示：

```
>>> 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
'\xef\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    <title>1world0x00sec</title>\n    <link rel="stylesheet" href="http://www.1world0x00.com/usr/themes/default/style.min.css">\n    <link .....'
```

以二进制的方式打开服务器并返回数据。

post请求

假如你向某个服务器发送一些数据，可能会谁用post的方式，用requests模块实现这种请求比较简单，只需要传递一个字典给data参数。

```
>>> import requests
>>> payload = {"key1": "value1", "key2": "value2"}
>>> r = requests.post("http://httpbin.org/post")
>>> r1 = requests.post("http://httpbin.org/post", data=payload)
```

r没有加data的请求，得到的效果是：

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Content-Length": "0",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.4.3 CPython/2.7.8 Windows/7",
    "X-Request-Id": "19ed80fc-ffe6-4dc0-b83a-08dba09daf88"
  },
  "json": null,
  "origin": "118.113.116.160",
  "url": "http://httpbin.org/post"
}
```

r1为data提供了值，再看效果：

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "key1": "value1",
    "key2": "value2"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Content-Length": "23",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.4.3 CPython/2.7.8 Windows/7",
    "X-Request-Id": "b8ba897f-44c9-4922-b157-562e0cf07bcd"
  },
  "json": null,
  "origin": "118.113.116.160",
  "url": "http://httpbin.org/post"
}
```

新闻比较看才有意思，代码也如此。比较上面两个结果，发现后者当data被赋值之后，在结果中form的值即为data所传入的数据，它就是post给服务器的内容。喵...

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，一般用到字典类型的数据。

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

通过一个实例，展示第三方模块的应用方法，其实没有什么特殊的地方，只要安装了，就和用标注库模块一样了。

根据我的个人经验，第三方模块常常在某个方面做得更好，或者性能更优化，所以，不要将其放在我们的视野之外。

[总目录](#) | [上节：标准库\(8\)](#) | [下节：存入文件](#)

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

要常常喜乐，不住地祷告，凡事谢恩，因为这是神在基督耶稣里向你们所定的旨意。不要消灭圣灵的感动，不要藐视先知的讲论。但要凡事察验，善美的要持守，各样的恶事要禁戒不作。(1 THESSALONIANS 5:16-22)

将数据存入文件

在《[文件\(1\)](#)》和《[文件\(2\)](#)》中，已经学习了如何读写文件。

程序执行结果，就是产生一些数据，一般情况下，这些数据要保存到磁盘中，最简单的方法就是写入到某个文件。但是，如果仅仅是简单地把数据写入文件，不是最佳的存储机构。为此，就有了诸多不同的数据存储方式，这些方式不仅能够保证数据被存储，还能够让数据便于读取，此外，还有很多其它方面的优势。

简而言之，就是要将存储的对象格式化（或者叫做序列化），才好存好取。这就有点类似集装箱的作用。

pickle

pickle是标准库中的一个模块，在Python 2中还有一个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。

而在Python 3中，你只需要 `import pickle` 即可，因为它已经在Python 3中具备了Python 2中的cpickle同样的性能。

```
>>> import pickle
>>> integers = [1, 2, 3, 4, 5]
>>> f = open("22901.dat", "wb")
>>> pickle.dump(integers, f)
>>> f.close()
```

用 `pickle.dump(integers, f)` 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，在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对象。
- `file`：要写入的文件。可以更广泛地理解为拥有 `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。

换一种数据格式，并且做对比：

将数据存入文件

```
>>> import pickle
>>> d = {}
>>> integers = range(9999)
>>> d["i"] = integers      #下面将这个字典类型的对象存入文件

>>> 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)    #Python 3: print("{0:d}, {1:d}, {2:.2f}").format(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。

将数据保存入文件，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    #Python 3: print(integers)
[1, 2, 3, 4, 5]
```

再看看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()
```

如果是自己定义的对象，是否可以用上述方式存入文件并读出来呢？看下面的例子：

将数据存入文件

```
>>> import cPickle as pickle          #这是Python 2的引入方式，如果是Python 3，直接使用import
pickle
>>> 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      #Python 3: print("my book is: ", se
lf.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
qc_builtin__
object
qNtRq}qUnameqU<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
qc_builtin__
object
qNtRq}qUnameqU<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。先看一下基本写、读操作。

```
>>> 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，这很接近数据库的样式了。下面是读。

```
>>> s = shelve.open("22901.db")
>>> name = s["name"]
>>> print name      #Python 3: print(name)
www.itdiffer.com
>>> contents = s["contents"]
>>> print contents    #Python 3: print(contents)
{'second': 'day day up', 'first': 'base knowledge'}
```

看到输出的内容，你一定想到，肯定可以用 `for` 语句来读，想到了就用代码来测试，这就是 Python 交互模式的便利之处。

```
>>> 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。所建立的对象被变量 `s` 所引用，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：

将数据存入文件

```
>>> 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          #Python 3: 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](#) | [上节：第三方库](#) | [下节：mysql数据库\(1\)](#)

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

我们在你们那里的时候，曾吩咐你们说，若有人不肯作工，就不可吃饭。因我们听说，在你们中间有人不按规矩而行，什么工都不作，反倒专管闲事。我们靠主耶稣基督，吩咐、劝解这样的人，要安静作工，吃自己的饭。(2 THESSALONIANS 3:10-12)

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，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，地球上的数据库主要有三种：

- 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) ...

概况

MySQL是一个使用非常广泛的数据库，很多网站都使用它。关于这个数据库有很多传说，例如[维基百科](#)上有这么一段：

MySQL（官方发音为英语发音：/maɪ̯ ,ɛskju:ˈɛl/ "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不小的网站。

安装

你的电脑或许不会天生就有MySQL，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。本书的目标是from beginner to master，不管是是不是真的master，总要装得像，Linux能够给你撑门面，这也是推荐使用Ubuntu的原因。

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，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（root）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
```

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，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。以后如果再登录数据库，就可以用刚才设置的密码了。

运行

安装完就要运行它，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>
```

恭喜你，已经进入到数据库操作界面了，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           |
+-----+
```

show databases;（最后的半角分号别忘记了）命令表示列出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`。

python-MySQLdb 或 pymysql

`python-MySQLdb` 是一个接口程序，Python通过它对MySQL数据实现各种操作。但是，这仅适用于Python 2，如果你使用Python 3，那就要用到 `pymysql` 了。

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，直接用手指去捅肯定是不行的，必须借助工具插入到锁孔中，把锁打开，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，

MySQL数据库(1)

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（当然，具有特殊开锁能力的人除外），也就是接口程序，编码水平等都是考虑因素。

`python-MySQLdb / pymysql` 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。

在Ubuntu操作系统下还可以用软件仓库来安装。

```
sudo apt-get install build-essential python-dev libmysqlclient-dev
```

然后可以：

```
sudo apt-get install python-MySQLdb
```

最推荐是使用pip：

```
pip install mysql-python
```

如果是Python 3，则要安装 `pymysql`：

```
pip3 install pymysql
```

下面的演示中，我使用Python 2，如果读者使用Python 3，请自行 `import pymysql`，其它函数部分，基本一致，所以就不重复写代码了。请读者务必注意。

安装之后，在Python 2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，那么恭喜你，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：

```
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，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)`，说明这个数据库已经建立好了，名字叫`qiwsirtest`。

数据库建立之后，就可以用Python通过已经安装的 `python-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")
```

下面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，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`，不过，这样种做法在大型系统中是应该避免的。
- **passwd**：`user`账户登录MySQL的密码。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"`qiwsirtest`"，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，`MySQLdb`的默认端口是3306，当`MySQLdb`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（服务器管理员可以进行端口配置）。
- **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。这里将`qiwsirtest`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`MySQLdb`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`。

注：`connect`中的所有参数，可以只按照顺序把值写入。但是，我推荐读者还是按照上面的方式，以免出了乱子自己还糊涂。

其实，关于connect的参数还有别的，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.

至此，已经完成了数据库的连接。

[总目录](#) | [上节：将数据存入文件](#) | [下节：mysql数据库\(2\)](#)

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

"So do not worry about tomorrow, for tomorrow will bring worries of its own. Today's trouble is enough for today." (MATTHEW 7:34)

MySQL数据库(2)

就数据库而言，连接之后就要对其操作。但是，目前名字叫作`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？下面摘抄维基百科对数据库表的简要解释。

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。

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。据小道消息，有的网站居然用明文保存密码，这么做是比较可恶的。就让我在这里，仅仅在这里可恶一次。

数据库表

因为直接操作数据库不是本书重点，但是关联到后面的操作，为了让读者在阅读上连贯，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。

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)default charset=utf8;
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)
```

向里面插入一条信息。

```

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。

操作数据库

首先要连接数据库。

```

>>> import MySQLdb
>>> conn = MySQLdb.connect(host="localhost",user="root",passwd="123123",db="qiwsirttest"
",charset="utf8")

```

Python建立了与数据库的连接，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了。

Python和数据库之间的连接建立起来之后，若要操作它，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
close()	关闭游标。之后游标不可用
execute(query[,args])	执行一条SQL语句，可以带参数
executemany(query, pseq)	对序列pseq中的每个参数执行sql语句
fetchone()	返回一条查询结果
fetchall()	返回所有查询结果
fetchmany([size])	返回size条结果
nextset()	移动到下一个结果
scroll(value,mode='relative')	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。不妨进入到"mysql>"交互方式查看。

```
mysql> select * from users;
+----+-----+-----+
| id | username | password | email      |
+----+-----+-----+
|  1 | qiwfir  | 123123   | qiwfir@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"的一个方法：`commit()`，将数据提交上去，也就是进行了 `cur.execute()` 操作，要将数据提交才能有效改变数据库的内容。

```
>>> conn.commit()
```

再到"mysql>"中运行 "select * from users" 试一试：

```
mysql> select * from users;
+----+-----+-----+
| id | username | password | email
+----+-----+-----+
| 1  | qiwfir  | 123123   | qiwfir@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。所有以 `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.com"),("facebook","222333","f@face.book"),("github","333444 ", "git@hub.com"),("docker","444555","doc@ker.com")))
4L
>>> conn.commit()
```

到"mysql>"里面看结果：

```
mysql> select * from users;
+----+-----+-----+
| id | username | password | email
+----+-----+-----+
| 1  | qiwfir  | 123123   | qiwfir@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` 这时候是一个元组，特别注意括号——一环套一环的括号，这个元组里面的元素也是元组，每个元组分别对应sql语句中的字段列表。

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，7条记录在哪里呢，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()
```

至此已经将查询到的记录赋值给变量 `lines` 了。如果要将它们显示出来，就要用到曾经学习过的循环语句。

```
>>> for line in lines:
...     print line          #Python 3: 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\u9f50', u'9988', u'qiwsir@gmail.com')
```

很好，果然逐条显示出来了。请读者注意，第七条中的 `u'\u8001\u9f50'` 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，再看下面：

```
>>> cur.execute("select * from users where id=1")
1L
>>> line_first = cur.fetchone()      #只返回一条
>>> print line_first          #Python 3: 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()          #Python 3: 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'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\u9f50', u'9988', u'qiwsir@gmail.com'))
```

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，作为对象存在。从上面的实验探讨发现，返回值是一个元组对象，里面的每个元素，都是一个一个的元组。因此，用 `for` 循环就可以一个一个拿出来了。

继续，可以看到神奇。接着上面的操作，再打印一遍。

```
>>> print cur.fetchall()          #Python 3:  print(cur.fetchall())
()
```

晕了！怎么什么是空？不是说作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，这就是神奇所在。

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。这如同什么？能不能与本书前面已经有的知识关联起来？

下面还要实验，检验上面所说：

```
>>> cur.execute('select * from users')
7L
>>> print cur.fetchone()          #Python 3:  print(cur.fetchone() )
(1L, u'qiwsir', u'123123', u'qiwsir@gmail.com')
>>> print cur.fetchone()          #Python 3:  print(cur.fetchone() )
(2L, u'python', u'123456', u'python@gmail.com')
>>> print cur.fetchone()          #Python 3:  print(cur.fetchone() )
(3L, u'google', u'111222', u'g@gmail.com')
```

这次不再一次性全部打印出来了，而是一次打印一条，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，这次实验中重新运行了查询语句。

那么，既然操作存储在内存中的对象时游标会移动，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，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”。

但在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()        #Python 3:    print(cur.fetchone())
(3L, u'google', u'111222', u'g@gmail.com')

>>> cur.scroll(1, "absolute")
>>> print cur.fetchone()        #Python 3:    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()        #Python 3:    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——`cur.fetchmany()`，在前面操作的基础上继续。

```
>>> 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'f@face.book'))
```

上面这个操作，就是实现了从当前位置（游标指向序号为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。

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。

```
>>> 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'123456', 'id': 2L, 'email': u'python@gmail.com'}, {'username': u'google', 'password': u'111222', 'id': 3L, 'email': u'g@gmail.com'}, {'username': u'facebook', 'password': u'222333', 'id': 4L, 'email': u'f@face.book'}, {'username': u'github', 'password': u'333444', 'id': 5L, 'email': u'git@hub.com'}, {'username': u'docker', 'password': u'444555', 'id': 6L, 'email': u'doc@ker.com'}, {'username': u'\u8001\u9f50', 'password': u'9988', 'id': 7L, 'email': u'qiwsir@gmail.com'}]
```

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"]      #Python 3: print(line["username"])
...
qiwsir
mypython
google
...
```

根据字典对象的特点来读取了“键-值”。

更新数据

熟悉了前面的操作，再到底这里，一切都显得简单了，但仍要提醒的是，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的更新，还要运行：

```
>>> conn.commit()
```

还有个小尾巴，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](#) | 上节：[mysql数据库\(1\)](#) | 下节：[mongodb数据库](#)

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

大哉！敬虔的奥秘，无人不以为然，就是：神在肉身显现，被圣灵称义，被天使看见，被传于外邦，被世人信服，被接在荣耀里。(1 TIMOTHY 3:16)

MongoDB数据库(1)

MongoDB开始火了，这是时代发展的需要。为此，在这里也要探讨一下如何用Python来操作此数据库。考虑到读者对这种数据库的了解可能比关系型数据库陌生，所以，要用多一点的篇幅来介绍。

MongoDB是属于NoSql的。

NoSql（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。

MongoDB是NoSql其一，选择它，主要是因为我喜欢，下面说说它的特点。

- 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

基于它的特点，擅长的领域就在于：

- 大数据（太时髦了！以下可以都不看，有这么一条就足够了）
- 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。上述安装流程可以参考：[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>，该网站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: [tar](#) | [zip](#)

Windows

Linux

Mac OS X

Solaris

VERSION:

Windows 64-bit 2008 R2+



如果在安装过程中遇到了问题，建议去问Google大神（如果有读者心存疑虑或者愤愤不平，请不要发怒，这是我的个人建议，不同意可以略过，我当然也尊重读者的个人选择）。

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

安装完毕就可以启动数据库。因为本书不是专门讲数据库，所以这里不涉及数据库的详细讲解，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，那个数据库作为对象被赋给这个全局变量`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，会看到刚才用到的数据库mydb了。

在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...
```

写本书的时候，安装版本号如下，如果读者的版本不一样，也无大碍。

```
>>> import pymongo
>>> pymongo.version
'3.0.1'
```

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
```

安装好之后，进入到Python的交互模式：

```
>>> import pymongo
```

说明模块没有问题。

连接

既然Python驱动MongoDB的模块pymongo业已安装完毕，接下来就是连接，即建立连接对象。

```
>>> pymongo.Connection("localhost",27017)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'Connection'
```

报错！我在写本书之前做项目时，就是按照上面方法连接的，读者可以查一下，会发现很多教程是这么连接的。但是，眼睁睁地看到了报错。

所以，一定要注意这里的坑。

如果读者用的是旧版本的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', 'GE02D', 'GEOHAYSTACK', 'GEOSPHERE', 'HASHED', 'IndexModel', 'InsertOne', 'MAX_SUPPORTED_WIRE_VERSION', 'MIN_SUPPORTED_WIRE_VERSION', 'MongoClient', 'MongoReplicaSetClient', 'OFF', 'ReadPreference', 'ReplaceOne', 'ReturnDocument', 'SLOW_ONLY', 'TEXT', 'UpdateMany', 'UpdateOne', 'WriteConcern', '__builtins__', '__doc__', '__file__', '__name__', '__package__', '__path__', '_cmessage', 'auth', 'bulk', 'client_options', 'collection', 'command_cursor', 'common', 'cursor', 'cursor_manager', 'database', 'errors', 'get_version_string', 'has_c', 'helpers', 'ismaster', 'message', 'mongo_client', 'mongo_replica_set_client', 'monitor', 'monotonic', 'network', 'operations', 'periodic_executor', 'pool', 'read_preferences', 'response', 'results', 'server', 'server_description', 'server_selectors', 'server_type', 'settings', 'son_manipulator', 'ssl_context', 'ssl_support', 'thread_util', 'topology', 'topology_description', 'uri_parser', 'version', 'version_tuple', 'write_concern']
```

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`，“他变了”。

不过，我发现了 `MongoClient()`，真乃峰回路转。

```
>>> 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`，于是再获取这个集合，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', '_Collection__create_index', '_Collection__database', '_Collection__find_and_modify', '_Collection__full_name', '_Collection__name', '__call__', '__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__getattribute__', '__getitem__', '__hash__', '__init__', '__iter__', '__module__', '__ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_command', '_count', '_delete', '_insert', '_socket_for_primary_reads', '_socket_for_reads', '_socket_for_writes', '_update', 'aggregate', 'bulk_write', 'codec_options', 'count', 'create_index', 'create_indexes', 'database', 'delete_many', 'delete_one', 'distinct', 'drop', 'drop_index', 'drop_indexes', 'ensure_index', 'find', 'find_and_modify', 'find_one', 'find_one_and_delete', 'find_one_and_replace', 'find_one_and_update', 'full_name', 'group', 'index_information', 'initialize_ordered_bulk_op', 'initialize_unordered_bulkop', 'inline_map_reduce', 'insert', 'insert_many', 'insert_one', 'list_indexes', 'map_reduce', 'name', 'next', 'options', 'parallel_scan', 'read_preference', 'reindex', 'remove', 'rename', 'replace_one', 'save', 'update', 'update_many', 'update_one', 'with_options', 'write_concern']
```

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，MongoDB的shell中的命令与pymongo中的方法有时候会稍有差别，务必小心。比如刚才这个，在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，于是就进入到了“增删改查”的常规操作。

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，每次给她照相，每拍了一张，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          #Python 3: 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，不一定每条文档都要结构一样，比如，也可以在集合中插入这样的文档。

```
>>> 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），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
```

提醒读者，批量插入的文档大小是有限制的，有人说不要超过20万条，有人说不要超过16MB，我没有测试过。在一般情况下，或许达不到上限，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          #Python 3: 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          #Python 3: 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'}
```

如果读者看到这里，请务必注意，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，但是，如果用Python读过来之后，无法直接用JSON中的 `json.dumps()` 方法操作文档。其中一种解决方法就是将文档中的 `'_id'` “键/值”对删除（例如：`del doc['_id']`），然后使用 `json.dumps()` 即可。读者也可是使用 `json_util` 模块，因为它是“Tools for using Python's json module with BSON documents”，请阅读 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，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
<code>\$set</code>	用来指定一个键的值。如果不存在则创建它
<code>\$unset</code>	完全删除某个键
<code>\$inc</code>	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）
<code>\$push</code>	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，稍一演示，读者必会。

```
>>> 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，是否建立索引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'
```

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。

然而，敬虔加上知足的心便是大利了，因为我们没有带什么到世上来，也不能带什么去，只要有衣有食，就当知足。但那些想要发财的人，就陷在迷惑、落在网罗和许多无知有害的私欲里，叫人沉在败坏和灭亡中。贪财是万恶之根。有人贪恋钱财，就被引诱离了真道，用许多愁苦把自己刺透了。(1 TIMOTHY 6:6-10)

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。前面的两个数据库，不管是MySQL还是MongoDB，都需要“安装”，安装之后，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。并且在学过MySQL的基础上，理解本节能容就容易多了。

```
>>> 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")` 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', 'InternalE
rror', 'NotSupportedError', 'OperationalError', 'ProgrammingError', 'Warning', '__call_
__', '__class__', '__delattr__', '__doc__', '__enter__', '__exit__', '__format__', '__
getattribute__', '__hash__', '__init__', '__new__', '__reduce__', '__reduce_ex__', '__
repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'close', 'commit'
, 'create_aggregate', 'create_collation', 'create_function', 'cursor', 'enable_load_ex
tension', 'execute', 'executemany', 'executescript', 'interrupt', 'isolation_level', 'i
terdump', 'load_extension', 'rollback', 'row_factory', 'set_authorizer', 'set_progres
s_handler', 'text_factory', 'total_changes']
```

游标对象

这一步跟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：

```
>>> dir(cur)
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__':
, '__init__', '__iter__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setat
tr__', '__sizeof__', '__str__', '__subclasshook__', 'arraysize', 'close', 'connection'
, 'description', 'execute', 'executemany', 'executescript', 'fetchall', 'fetchmany', 'f
etchone', 'lastrowid', 'next', 'row_factory', 'rowcount', 'setinputsizes', 'setoutput
size']
```

看到熟悉的名称了：`close()`, `execute()`, `executemany()`, `fetchall()`

创建数据库表

面对SQLite数据库，读者曾经数据的sql指令都可以照样使用。

```
>>> 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，还要如下操作（这是多么熟悉的操作流程和命令呀）：

```
>>> conn.commit()
>>> cur.close()
>>> conn.close()
```

在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。

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()          #Python 3: 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()
```

用循环语句打印查询结果：

```
>>> rows = cur.execute('select * from books')
>>> for row in rows:
...     print row          #Python 3: 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

删除也是操作数据库必须的动作。

```
>>> 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，在完成对数据库的操作后，一定要关门才能走人：

```
>>> cur.close()
>>> conn.close()
```

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：<https://docs.python.org/2/library/sqlite3.html>

[总目录](#) | [上节：MongoDB数据库](#) | [下节：电子表格](#)

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

人若自洁，脱离卑贱的事，就必作贵重的器皿，成为圣洁，合乎主用，预备行各样的善事。你要逃避少年的私欲，同那清心祷告主的人追求公义、信德、仁爱、和平。惟有那愚拙无学问的辩论，总要弃绝，因为知道这等事是起争竞的。(2 TIMOTHY 2:21-23)

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“历史悠久”，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·法蘭克斯頓（Bob Frankston）發展而成，1979年10月跟著蘋果二號電腦推出，成爲蘋果二號電腦上的「殺手應用軟體」。

接下来是Lotus 1-2-3，由Lotus Software（美國蓮花軟體公司）於1983年起所推出的電子試算表軟體，在DOS時期廣爲個人電腦使用者所使用，是一套殺手級應用軟體。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電子制表軟件—Multiplan，並在CP/M系統上大獲成功，但在MS-DOS系統上，Multiplan敗給了Lotus 1-2-3。

1985年，微软推出第一款Excel，但它只用於Mac系統；直到1987年11月，微软的第一款適用於Windows系統的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Lotus1-2-3遲遲不能適用於Windows系統，到了1988年，Excel的銷量超過了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電子制表軟件VisiCalc的特性：行、列組成單元格，數據、與數據相關的公式或者對其他單元格的絕對引用保存在單元格中。

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，因为有些数据，就存在电子表格中。

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。

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，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，还可以插队：

```
>>> ws2 = wb.create_sheet(1)
```

在第二个位置插入了一个sheet。

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 3: print(wb.get_sheet_names())
['python', 'Sheet2', 'Sheet1']
```

Sheet2之所以排在了第二位，是因为在建立的时候，用了一个插队的方法。这跟在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 3: print(sh.title)
...
python
Sheet2
Sheet1
```

如果读者 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__`。说明，工作簿对象是可迭代的。

cell

为了能够清楚理解向电子表格中增加数据的过程，将电子表中约定的名称以下图方式说明：

电子表格

B4		fx	this is a cell				
	A	B	C	D	E	F	G
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操作的电子表格的情况下，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：

电子表格

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>))
```

这是按照横向顺序读过来的，即A1-B1-C1，作为一个元组，然后读下一横行，再组成一个元组。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      #Python 3: 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，所以是当前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）：

电子表格

```
>>> 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 'value'
```

报错了。关键是没有注意观察上面的结果。元组里面是以元组为元素，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，不过可以换一个属性：

```
>>> 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 3: 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            #Python 3: 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](#) | [上节：SQLite数据库](#) | [下节：实战-引](#)

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

如何成为Python高手

这篇文章是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.secnetix.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>

ASCII、Unicode、GBK和UTF-8字符编码的区别联系

2015-05-08 实验楼

很久很久以前，有一群人，他们决定用8个可以开合的晶体管来组合成不同的状态，以表示世界上的万物。他们看到8个开关状态是好的，于是他们把这称为“字节”。

再后来，他们又做了一些可以处理这些字节的机器，机器开动了，可以用字节来组合出很多状态，状态开始变来变去。他们看到这样是好的，于是它们就这机器称为“计算机”。

开始计算机只在美国用。八位的字节一共可以组合出 $256(2\text{的}8\text{次方})$ 种不同的状态。他们把其中的编号从0开始的32种状态分别规定了特殊的用途，一但终端、打印机遇上约定好的这些字节被传过来时，就要做一些约定的动作，如：

遇上 $0x10$, 终端就换行；
 遇上 $0x07$, 终端就向人们嘟嘟叫；
 遇上 $0x1b$, 打印机就打印反白的字，或者终端就用彩色显示字母。

他们看到这样很好，于是就把这些 $0x20$ 以下的字节状态称为“控制码”。他们又把所有的空格、标点符号、数字、大小写字母分别用连续的字节状态表示，一直编到了第127号，这样计算机就可以用不同字节来存储英语的文字了。

大家看到这样，都感觉很好，于是大家都把这个方案叫做 ANSI的“Ascii”编码（American Standard Code for Information Interchange，美国信息互换标准代码）。当时世界上所有的计算机都用同样的ASCII方案来保存英文文字。

后来，就像建造巴比伦塔一样，世界各地的都开始使用计算机，但是很多国家用的不是英文，他们的字母里有许多是ASCII里没有的，为了可以在计算机保存他们的文字，他们决定采用127号之后的空位来表示这些新的字母、符号，还加入了很多画表格时需要用下到的横线、竖线、交叉等形式，一直把序号编到了最后一个状态255。

从128到255这一页的字符集被称“扩展字符集”。从此之后，贪婪的人类再没有新的状态可以用了，美帝国主义可能没有想到还有第三世界国家的人们也希望可以用到计算机吧！

等中国人们得到计算机时，已经没有可以利用的字节状态来表示汉字，况且有6000多个常用汉字需要保存呢。

但是这难不倒智慧的中国人民，我们不客气地把那些127号之后的奇异符号们直接取消掉，规定：

一个小于127的字符的意义与原来相同，但两个大于127的字符连在一起时，就表示一个汉字；

前面的一个字节（他称之为高字节）从0xA1用到 0xF7，后面一个字节（低字节）从0xA1到0xFE；

这样我们就可以组合出大约7000多个简体汉字了。

在这些编码里，我们还把数学符号、罗马希腊的字母、日文的假名们都编进去了，连在 ASCII 里本来就有的数字、标点、字母都统统重新编了两个字节长的编码，这就是常说的“全角”字符。

而原来在127号以下的那些就叫“半角”字符了。

中国人民看到这样很不错，于是就把这种汉字方案叫做“GB2312”。GB2312 是对 ASCII 的中文扩展。

但是中国的汉字太多了，我们很快就发现有许多人的人名没有办法在这里打出来，特别是某些很会麻烦别人的国家领导人。

于是我们不得不继续把 GB2312 没有用到的码位找出来老实不客气地用上。

后来还是不够用，于是干脆不再要求低字节一定是127号之后的内码，只要第一个字节是大于127就固定表示这是一个汉字的开始，不管后面跟的是不是扩展字符集里的内容。结果扩展之后的编码方案被称为 GBK 标准，GBK 包括了 GB2312 的所有内容，同时又增加了近20000个新的汉字（包括繁体字）和符号。

后来少数民族也要用电脑了，于是我们再扩展，又加了数千个新的少数民族的字，GBK 扩成了 GB18030。从此之后，中华民族的文化就可以在计算机时代中传承了。

中国的程序员们看到这一系列汉字编码的标准是好的，于是通称他们叫做“DBCS”（Double Byte Character Set 双字节字符集）。

在 DBCS 系列标准里，最大的特点是两字节长的汉字字符和一字节长的英文字符并存于同一套编码方案里，因此他们写的程序为了支持中文处理，必须要注意字符串里的每一个字节的值，如果这个值是大于127的，那么就认为一个双字节字符集里的字符出现了。

那时候凡是受过加持，会编程的计算机僧侣们都要每天念下面这个咒语数百遍：“一个汉字算两个英文字符！一个汉字算两个英文字符……”

因为当时各个国家都像中国这样搞出一套自己的编码标准，结果互相之间谁也不懂谁的编码，谁也不支持别人的编码。

连大陆和台湾这样只相隔了150海里，使用着同一种语言的兄弟地区，也分别采用了不同的 DBCS 编码方案——当时的中国人想让电脑显示汉字，就必须装上一个“汉字系统”，专门用来处理汉字的显示、输入的问题，但是那个台湾的愚昧封建人士写的算命程序就必须加装另一套支持 BIG5 编码的什么“倚天汉字系统”才可以用，装错了字符系统，显示就会乱了套！这怎么办？而且世界民族之林中还有那些一时用不上电脑的穷苦人民，他们的文字又怎么办？真是计算机的巴比伦塔命题啊！

正在这时，大天使加百列及时出现了一个叫 ISO（国际标准化组织）的国际组织决定着手解决这个问题。

他们采用的方法很简单：

废了所有的地区性编码方案，重新搞一个包括了地球上所有文化、所有字母和符号 的编码！

他们打算叫它“Universal Multiple-Octet Coded Character Set”，简称 UCS，俗称“unicode”。

unicode开始制订时，计算机的存储器容量极大地发展了，空间再也不成为问题了。于是 ISO 就直接规定必须用两个字节，也就是16位来统一表示所有的字符，对于ASCII里的那些“半角”字符，unicode保持其原编码不变，只是将其长度由原来的8位扩展为16位，而其他文化和语言的字符则全部重新统一编码。由于“半角”英文符号只需要用到低8位，所以其高8位永远是0，因此这种大气的方案在保存英文文本时会多浪费一倍的空间。

这时候，从旧社会里走过来的程序员开始发现一个奇怪的现象：他们的strlen函数靠不住了，一个汉字不再是相当于两个字符了，而是一个！是的，从unicode开始，无论是半角的英文字母，还是全角的汉字，它们都是统一的“一个字符”！同时，也都是统一的“两个字节”，请注意“字符”和“字节”两个术语的不同：

“字节”是一个8位的物理存储单元，

而“字符”则是一个文化相关的符号。

在unicode中，一个字符就是两个字节。一个汉字算两个英文字符的时代已经快过去了。

unicode同样也不完美，这里就有两个的问题，

一个是，如何才能区别unicode和ascii？计算机怎么知道三个字节表示一个符号，而不是分别表示三个符号呢？

第二个问题是，我们已经知道，英文字母只用一个字节表示就够了，如果unicode统一规定，每个符号用三个或四个字节表示，那么每个英文字母前都必然有二到三个字节是0，这对于存储空间来说是极大的浪费，文本文件的大小会因此大出二三倍，这是难以接受的。

unicode在很长一段时间内无法推广，直到互联网的出现，为解决unicode如何在网络上传输的问题，于是面向传输的众多 UTF（UCS Transfer Format）标准出现了，顾名思义：

UTF-8就是每次8个位传输数据，

而UTF-16就是每次16个位。

UTF-8就是在互联网上使用最广的一种unicode的实现方式，这是为传输而设计的编码，并使编码无国界，这样就可以显示全世界上所有文化的字符了。

UTF-8最大的一个特点，就是它是一种变长的编码方式。

它可以使用1~4个字节表示一个符号，根据不同的符号而变化字节长度，当字符在ASCII码的范围时，就用一个字节表示，保留了ASCII字符一个字节的编码做为它的一部分，（注意的是unicode一个中文字符占2个字节，而UTF-8一个中文字符占3个字节）。

从unicode到uft-8并不是直接的对应，而是要过一些算法和规则来转换。

Unicode符号范围 | UTF-8编码方式

(十六进制)	(二进制)	—————	0000 0000-0000
007F 0xxxxxxx	0000 0080-0000	07FF 110xxxxx	10xxxxxx 0000 0800-0000
1110xxxx	10xxxxxx	10xxxxxx 0001 0000-0010	FFFF 11110xxx 10xxxxxx 10xxxxxx
10xxxxxx			

之前一直对字符编码很模糊，网查资料被忽悠地晕头转向，看完这篇风趣的文章，把之前模糊的知识点串联起来，并稍加总结，我和我的小伙伴们都明白了！

via:<http://toutiao.io/>

原文链接：<http://dengo.org/archives/901>

大数据全栈式开发语言 – Python

来源：<http://www.techug.com/full-stack-python>

前段时间，ThoughtWorks 在深圳举办一次社区活动上，有一个演讲主题叫做“Fullstack JavaScript”，是关于用 JavaScript 进行前端、服务器端，甚至数据库（MongoDB）开发，一个 Web 应用开发人员，只需要学会一门语言，就可以实现整个应用。

受此启发，我发现 Python 可以称为大数据全栈式开发语言。因为 Python 在云基础设施，DevOps，大数据处理等领域都是炙手可热的语言。

领域	流行语言
云基础设施	Python, Java, Go
DevOps	Python, Shell, Ruby , Go
网络爬虫	Python, PHP, C++
数据处理	Python, R, Scala

就像只要会 JavaScript 就可以写出完整的 Web 应用，只要会 Python，就可以实现一个完整的大数据处理平台。

云基础设施

这年头，不支持云平台，不支持海量数据，不支持动态伸缩，根本不敢说自己是做大数据的，顶多也就敢跟人说是做商业智能（BI）。

云平台分为私有云和公有云。私有云平台如日中天的 OpenStack，就是 Python 写的。曾经的追赶者 CloudStack，在刚推出时大肆强调自己是 Java 写的，比 Python 有优势。结果，搬石砸脚，2015 年初，CloudStack 的发起人 Citrix 宣布加入 OpenStack 基金会，CloudStack 眼看着就要寿终正寝。

如果嫌麻烦不想自己搭建私有云，用公有云，不论是 AWS，GCE，Azure，还是阿里云，青云，在都提供了 Python SDK，其中 GCE 只提供 Python 和 JavaScript 的 SDK，而青云只提供 Python SDK。可见各家云平台对 Python 的重视。

提到基础设施搭建，不得不提 Hadoop，在今天，Hadoop 因为其 MapReduce 数据处理速度不够快，已经不再作为大数据处理的首选，但是 HDFS 和 Yarn——Hadoop 的两个组件——倒是越来越受欢迎。Hadoop 的开发语言是 Java，没有官方提供 Python 支持，不过有很多第

三方库封装了 Hadoop 的 API 接口（`pydoop`, `hadoopy` 等等）。

Hadoop MapReduce 的替代者，是号称快上 100 倍的 Spark，其开发语言是 Scala，但是提供了 Scala, Java, Python 的开发接口，想要讨好那么多用 Python 开发的数据科学家，不支持 Python，真是说不过去。HDFS 的替代品，比如 GlusterFS, Ceph 等，都是直接提供 Python 支持。Yarn 的替代者，Mesos 是 C++ 实现，除 C++ 外，提供了 Java 和 Python 的支持包。

DevOps

DevOps 有个中文名字，叫做开发自运维。互联网时代，只有能够快速试验新想法，并在第一时间，安全、可靠的交付业务价值，才能保持竞争力。DevOps 推崇的自动化构建/测试/部署，以及系统度量等技术实践，是互联网时代必不可少的。

自动化构建是因应用而易的，如果是 Python 应用，因为有 `setuptools`, `pip`, `virtualenv`, `tox`, `flake8` 等工具的存在，自动化构建非常简单。而且，因为几乎所有 Linux 系统都内置 Python 解释器，所以用 Python 做自动化，不需要系统预安装什么软件。

自动化测试方面，基于 Python 的 Robot Framework 企业级应用最喜欢的自动化测试框架，而且和语言无关。Cucumber 也有很多支持者，Python 对应的 Lettuce 可以做到完全一样的事情。Locust 在自动化性能测试方面也开始受到越来越多的关注。

自动化配置管理工具，老牌的如 Chef 和 Puppet，是 Ruby 开发，目前仍保持着强劲的势头。不过，新生代 Ansible 和 SaltStack——均为 Python 开发——因为较前两者设计更为轻量化，受到越来越多开发者的欢迎，已经开始给前辈们制造了不少的压力。

在系统监控与度量方面，传统的 Nagios 逐渐没落，新贵如 Sensu 大受好评，云服务形式的 New Relic 已经成为创业公司的标配，这些都不是直接通过 Python 实现的，不过 Python 要接入这些工具，并不困难。

除了上述这些工具，基于 Python，提供完整 DevOps 功能的 PaaS 平台，如 Cloudify 和 Deis，虽未成气候，但已经得到大量关注。

网络爬虫

大数据的数据从哪里来？除了部分企业有能力自己产生大量的数据，大部分时候，是需要靠爬虫来抓取互联网数据来做分析。

网络爬虫是 Python 的传统强势领域，最流行的爬虫框架 Scrapy，HTTP 工具包 `urllib2`，HTML 解析工具 `beautifulsoup`, XML 解析器 `lxml`，等等，都是能够独当一面的类库。

不过，网络爬虫并不仅仅是打开网页，解析 HTML 这么简单。高效的爬虫要能够支持大量灵活的并发操作，常常要能够同时几千甚至上万个网页同时抓取，传统的线程池方式资源浪费比较大，线程数上千之后系统资源基本上就全浪费在线程调度上了。Python 由于能够很好的

支持协程（Coroutine）操作，基于此发展起来很多并发库，如 Gevent，Eventlet，还有 Celery 之类的分布式任务框架。被认为是比 AMQP 更高效的 ZeroMQ 也是最早就提供了 Python 版本。有了对高并发的支持，网络爬虫才真正可以达到大数据规模。

抓取下来的数据，需要做分词处理，Python 在这方面也不逊色，著名的自然语言处理程序包 NLTK，还有专门做中文分词的 Jieba，都是做分词的利器。

数据处理

万事俱备，只欠东风。这东风，就是数据处理算法。从统计理论，到数据挖掘，机器学习，再到最近几年提出来的深度学习理论，数据科学正处于百花齐放的时代。数据科学家们都用什么编程？

如果是在理论研究领域，R 语言也许是最受数据科学家欢迎的，但是 R 语言的问题也很明显，因为是统计学家们创建了 R 语言，所以其语法略显怪异。而且 R 语言要想实现大规模分布式系统，还需要很长一段时间的工程之路要走。所以很多公司使用 R 语言做原型试验，算法确定之后，再翻译成工程语言。

Python 也是数据科学家最喜欢的语言之一。和 R 语言不同，Python 本身就是一门工程性语言，数据科学家用 Python 实现的算法，可以直接用在产品中，这对于大数据初创公司节省成本是非常有帮助的。正式因为数据科学家对 Python 和 R 的热爱，Spark 为了讨好数据科学家，对这两种语言提供了非常好的支持。

Python 的数据处理相关类库非常多。高性能的科学计算类库 NumPy 和 SciPy，给其他高级算法打了非常好的基础，matplotlib 让 Python 画图变得像 Matlab 一样简单。Scikit-learn 和 Milk 实现了很多机器学习算法，基于这两个库实现的 PyLearn2，是深度学习领域的重要成员。Theano 利用 GPU 加速，实现了高性能数学符号计算和多维矩阵计算。当然，还有 Pandas，一个在工程领域已经广泛使用的大数据处理类库，其 DataFrame 的设计借鉴自 R 语言，后来又启发了 Spark 项目实现了类似机制。

对了，还有 iPython，这个工具如此有用，以至于我差点把他当成标准库而忘了介绍。iPython 是一个交互式 Python 运行环境，能够实时看到每一段 Python 代码的结果。默认情况下，iPython 运行在命令行，可以执行 ipython notebook 在网页中运行。用 matplotlib 绘制的图可以直接嵌入式的显示在 iPython Notebook 中。

iPython Notebook 的笔记本文件可以共享给其他人，这样其他人就可以在自己的环境中重现你的工作成果；如果对方没有运行环境，还可以直接转换成 HTML 或者 PDF。

为什么是 Python

正是因为应用开发工程师、运维工程师、数据科学家都喜欢 Python，才使得 Python 成为大数据系统的全栈式开发语言。

对于开发工程师而言，Python 的优雅和简洁无疑是最大的吸引力，在 Python 交互式环境中，执行`import this`，读一读 Python 之禅，你就明白 Python 为什么如此吸引人。Python 社区一直非常有活力，和 NodeJS 社区软件包爆炸式增长不同，Python 的软件包增长速度一直比较稳定，同时软件包的质量也相对较高。有很多人诟病 Python 对于空格的要求过于苛刻，但正是因为这个要求，才使得 Python 在做大型项目时比其他语言有优势。OpenStack 项目总共超过 200 万行代码，证明了这一点。

对于运维工程师而言，Python 的最大优势在于，几乎所有 Linux 发行版都内置了 Python 解释器。Shell 虽然功能强大，但毕竟语法不够优雅，写比较复杂的任务会很痛苦。用 Python 替代 Shell，做一些复杂的任务，对运维人员来说，是一次解放。

对于数据科学家而言，Python 简单又不失强大。和 C/C++ 相比，不用做很多的底层工作，可以快速进行模型验证；和 Java 相比，Python 语法简洁，表达能力强，同样的工作只需要 1/3 代码；和 Matlab，Octave 相比，Python 的工程成熟度更高。不止一个编程大牛表达过，Python 是最适合作为大学计算机科学编程课程使用语言——MIT 的计算机入门课程就是使用的 Python——因为 Python 能够让人学到编程最重要的东西——如何解决问题。

顺便提一句，微软参加 2015 年 PyCon，高调宣布提高 Python 在 Windows 上的编程体验，包括 Visual Studio 支持 Python，优化 Python 的 C 扩展在 Windows 上的编译等等。脑补下未来 Python 作为 Windows 默认组件的场景。

机器学习编程语言之争，Python夺魁

来源：<http://www.infoq.com/cn/news/2015/09/Python>

随着科技的发展，拥有高容量、高速度和多样性的大数据已经成为当今时代的主题词。数据科学领域中所采用的机器学习编程语言大相径庭。究竟哪种语言最适合机器学习成为争论不休的话题。近日，密西根州立大学的博士生Sebastian Raschka再次发起了机器学习编程语言之争，分析了自己选择Python的原因。

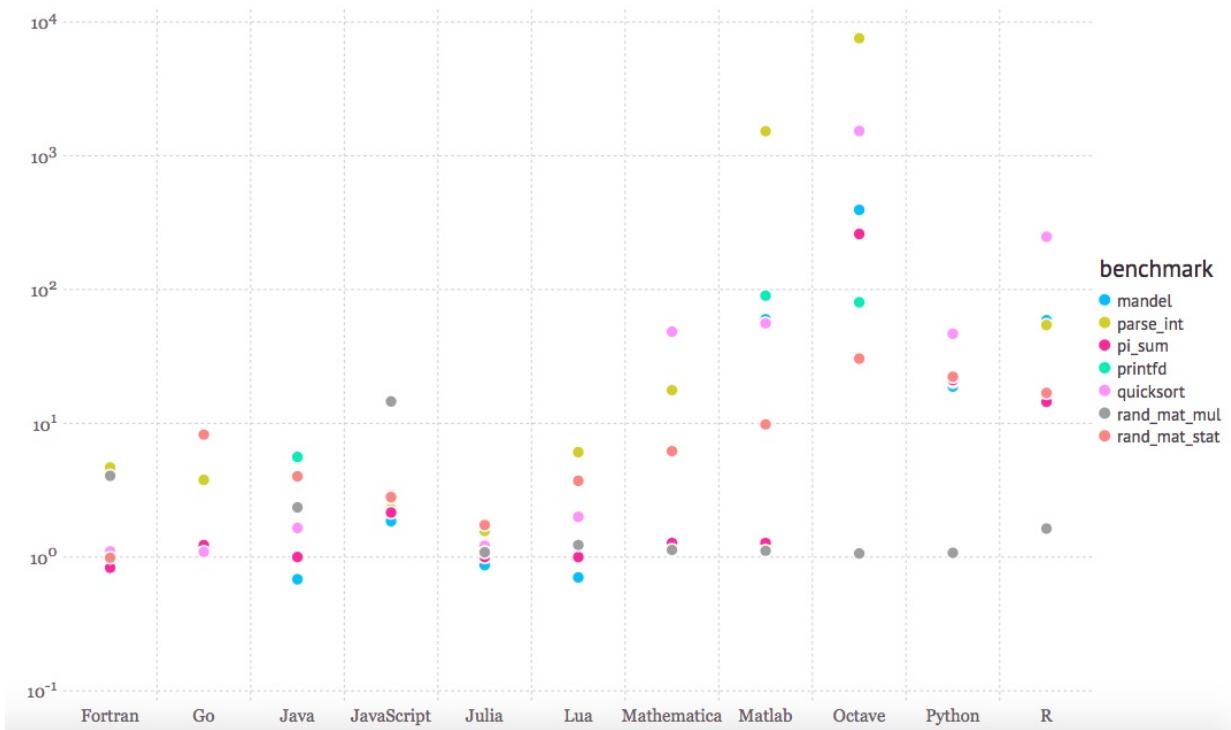
目前，机器学习牵涉的编程语言十分多样，包括了MATLAB、Julia、R、Perl、Python、Ruby等等。首先，Raschka定义了语言好坏的原则：一门好的语言应该使得编写、调试和执行代码的总时间最短。然后，他花大篇幅讲述了一个简单的道理——没有一门语言是对所有情况、所有人都十分适合的。

Roberto Waltman：“在这种唯一正确的方式下，面向对象的Spaghetti code就是Lasagna code。”

针对具体的应用场景以及参与项目的人员，开发团队或个人要对任务需求进行评估，再结合人员对于各个语言的熟悉程度和语言生态环境的情况等等，作出选择。接下来，Raschka开始正式点评各个语言相对与Python在机器学习时的优缺点。

表现平平的MATLAB

作为机器学习、模式识别等方面经常使用的工具，MATLAB被放在了第一个分析。Raschka认为，MATLAB在实现机器学习算法时要比Python或者NumPy更加自然。很多高校也在计算机科学相关的课程中教授MATLAB语言。然而，MATLAB也存在很多的缺点：价格昂贵、非开源、性能表现平平、语法不符合程序员的习惯等。例如，MATLAB中矩阵乘积运算操作为X.dot(Y)，而Python为X@W，更加简洁、明了。



各种语言相对于C的测试性能——C的性能为1.0，越小越好。从上图可以看出，MATLAB的性能比Python、Go、Java等语言要差很多。Raschka也提到，上图中Python的性能也表现不好。但是，现在性能强劲的GPU为Python提供了强大的后盾。在2010年的时候，Python中的[Theano库](#)在CPU上运行时，其速度是NumPy的1.8倍。而Theano在GPU上运行时，其速度就是NumPy的11倍。因此，在目前更高性能的GPU的帮助下，Python性能已经有了更大的改进。

貌似很强大的Julia

作为一门新型的语言，Julia在设计之处就充分借鉴了C/Ruby/Python等语言的经验，试图重新融合众家之长，为科学计算提供一个有力的工具。Raschka充分认可了该语言。然而，对于其未来Julia是否会流行，Raschka持保留态度。

Bjarne Stroustrup：世界上只有两种语言——大家一直在抱怨的和无人问津的。

Raschka认为一个语言是否有用又很大程度上取决于其是否流行。只有被广大编程人员所熟知的编程语言，相关的库才会更多，遇到问题时也能够方便的找到答案，且易于团队合作和代码分享。语言本身也会随着社区的关注不断得以改善。在流行度上，Julia并没有什么优势。

R语言本身无错

接下来，Raschka又提到了R语言。其实，Raschka之前使用过R语言，而且还专门撰写了一部有关R语言中Heat map的书。而且，他自己也不认为R语言存在任何令人不满意的地方。据Spectrum IEEE统计，在2015年排名前十的编程语言中，R语言是提升最快的。近期，微软对R语言表现出极大的兴趣。并且很快，微软的Azure便开始支持R语言了。

Alan J. Perlis：当某人说“我想要一门编程语言。它要能够根据我的目标自动生成代码”。让他拿根棒棒糖，一边玩去吧！

Raschka总结R和Python之间的不同在于，R是统计学家开发的（数学）语言，而Python是计算领域科学家开发，可以应用到统计学的编程语言。这就是说，二者都可以很好的应用于数据科学或者机器学习，但Python的语法更容易被程序员所接受。

Perl的没落

Raschka简单分析了Perl语言。作为一门古老的语言，Perl已经不可避免的走向了没落。

Raschka表示，Perl曾经是他学习的第一门语言。但是，Perl目前主要用于简单的脚本编写，已经很少应用在机器学习的算法编写方面。

老而弥坚的Python

最后，Raschka夸赞了自己所喜爱的Python语言。尽管Python已经走过了20多年的风雨历程，Raschka认为它仍然可以继续流行下去。而且，Raschka还不遗余力的列出了自己最喜欢的Python工具——NumPy、Theano、scikit-learn、matplotlib等。

至于Ruby、Java、Scala、Lua等，Raschka没有亲身体验，暂未点评。

对于Raschka的言论，很多人表示赞同，也有不少人反对。有人表示，自己从R切换到Python，主要原因也是Python更方便与团队之外的人进行合作和沟通。也有人表示，作为一个新入门的程序员，利用R进行数据科学的编程可以利用很多现成的库，十分得心应手。更有与Python打过8年交道的资深程序员表示，Julia十分具有吸引力，愿意花时间去尝试该语言。亲爱的读者朋友，你眼中最好的机器学习编程语言又是哪个呢？

Python2.7.x和3.x版本的重要区别

很多刚刚学习python的朋友，都纠结于目前Python的两个版本，虽然我已经对此进行了阐述（请阅读《[Python安装](#)》），但是，能够认真阅读的人不多。很多人是一目十行，看个大概罢了，重要观点往往忽视。

这里我选取网上一文章，该文讲述了一些具体的区别。供读者参考。

文章来源：<http://blog.jobbole.com/80006/>

许多Python初学者都会问：我应该学习哪个版本的Python。对于这个问题，我的回答通常是“先选择一个最适合你的Python教程，教程中使用哪个版本的Python，你就用那个版本。等学得差不多了，再来研究不同版本之间的差别”。

但如果想要用Python开发一个新项目，那么该如何选择Python版本呢？我可以负责任的说，大部分Python库都同时支持Python 2.7.x和3.x版本的，所以不论选择哪个版本都是可以的。但为了在使用Python时避开某些版本中一些常见的陷阱，或需要移植某个Python项目时，依然有必要了解一下Python两个常见版本之间的主要区别。

__future__ 模块

Python3.x引入了一些与Python2不兼容的关键字和特性，在Python2中，可以通过内置的 __future__ 模块导入这些新内容。如果你希望在Python2环境下写的代码也可以在Python 3.x中运行，那么建议使用 __future__ 模块。例如，如果希望在Python2中拥有Python3.x的整数除法行为，可以通过下面的语句导入相应的模块。

```
from __future__ import division
```

下表列出了 __future__ 中其他可导入的特性：

特性	可选版本	强制版本	效果
nested_scopes	2.1.0b1	2.2	PEP 227: <i>Statically Nested Scopes</i>
generators	2.2.0a1	2.3	PEP 255: <i>Simple Generators</i>
division	2.2.0a2	3.0	PEP 238: <i>Changing the Division Operator</i>
absolute_import	2.5.0a1	3.0	PEP 328: <i>Imports: Multi-Line and Absolute/Relative</i>
with_statement	2.5.0a1	2.6	PEP 343: <i>The “with” Statement</i>
print_function	2.6.0a2	3.0	PEP 3105: <i>Make print a function</i>
unicode_literals	2.6.0a2	3.0	PEP 3112: <i>Bytes literals in Python 3000</i>

示例：

```
from platform import python_version
```

print 函数

虽然print语法是Python3中一个很小的改动，且应该已经广为人知，但依然值得提一下：Python2中的print语句被Python3中的print()函数取代，这意味着在Python3中必须用括号将需要输出的对象括起来。

在Python2中使用额外的括号也是可以的。但反过来在Python3中想以Python2的形式不带括号调用print函数时，会触发 `SyntaxError`。

Python 2

```
print 'Python', python_version()
print 'Hello, World!'
print('Hello, World!')
print "text", ; print 'print more text on the same line'
```

```
Python 2.7.6
Hello, World!
Hello, World!
text print more text on the same line
```

Python 3

```
print('Python', python_version())
print('Hello, World!')

print("some text,", end="")
print(' print more text on the same line')

Python 3.4.1
Hello, World!
some text, print more text on the same line

print 'Hello, World!'

File "<ipython-input-3-139a7c5835bd>", line 1
print 'Hello, World!'
^
SyntaxError: invalid syntax
```

注意：

在Python中，带不带括号输出"Hello World"都很正常。但如果在圆括号中同时输出多个对象时，就会创建一个元组，这是因为在Python2中，print是一个语句，而不是函数调用。

```
print 'Python', python_version()
print('a', 'b')
print 'a', 'b'

Python 2.7.7
('a', 'b')
a b
```

整数除法

由于人们常常会忽视Python3在整数除法上的改动（写错了也不会触发Syntax Error），所以在移植代码或在Python2中执行Python3的代码时，需要特别注意这个改动。

Python 2.7.x 和 3.x 版本的重要区别

所以，我还是会在Python3的脚本中尝试用float(3)/2或 3/2.0代替3/2，以此来避免代码在Python2环境下可能导致的错误（或与之相反，在Python 2脚本中用 `from __future__ import division` 来使用Python3的除法）。

Python 2

```
print 'Python', python_version()
print '3 / 2 =', 3 / 2
print '3 // 2 =', 3 // 2
print '3 / 2.0 =', 3 / 2.0
print '3 // 2.0 =', 3 // 2.0

Python 2.7.6
3 / 2 = 1
3 // 2 = 1
3 / 2.0 = 1.5
3 // 2.0 = 1.0
```

Python 3

```
print('Python', python_version())
print('3 / 2 =', 3 / 2)
print('3 // 2 =', 3 // 2)
print('3 / 2.0 =', 3 / 2.0)
print('3 // 2.0 =', 3 // 2.0)

Python 3.4.1
3 / 2 = 1.5
3 // 2 = 1
3 / 2.0 = 1.5
3 // 2.0 = 1.0
```

Unicode

Python2有基于ASCII的str()类型，其可通过单独的unicode()函数转成unicode类型，但没有byte类型。

而在Python3中，终于有了Unicode（utf-8）字符串，以及两个字节类：bytes和bytearrays。

Python 2

```
print 'Python', python_version()

Python 2.7.6

print type(unicode('this is like a python3 str type'))

<type 'unicode'>

print type(b'byte type does not exist')

<type 'str'>

print 'they are really' + b' the same'

they are really the same

print type(bytarray(b'bytarray oddly does exist though'))

<type 'bytarray'>
```

Python 3

```
print('Python', python_version())
print('strings are now utf-8 u03BCnicou0394é!')

Python 3.4.1
strings are now utf-8 μnicοΔέ!

print('Python', python_version(), end="")
print(' has', type(b' bytes for storing data'))

Python 3.4.1 has <class 'bytes'>

print('and Python', python_version(), end="")
print(' also has', type(bytarray(b'bytarrays')))

and Python 3.4.1 also has <class 'bytarray'>

'note that we cannot add a string' + b'bytes for data'

TypeError Traceback (most recent call last)
<ipython-input-13-d3e8942ccf81> in <module>()
----> 1 'note that we cannot add a string' + b'bytes for data'

TypeError: Can't convert 'bytes' object to str implicitly
```

xrange

在Python2.x中，经常会用 `xrange()` 创建一个可迭代对象，通常出现在“for循环”或“列表/集合/字典推导式”中。

Python 2.7.x 和 3.x 版本的重要区别

这种行为与生成器非常相似（如“惰性求值”），但这里的xrange-iterable无尽的，意味着可能在这个xrange上无限迭代。

由于xrange的“惰性求知”特性，如果只需迭代一次（如for循环中），range()通常比xrange()快一些。不过不建议在多次迭代中使用range()，因为range()每次都会在内存中重新生成一个列表。

在Python 3中，range()的实现方式与xrange()函数相同，所以就不存在专用的xrange()（在Python 3中使用xrange()会触发NameError）。

```
import timeit

n = 10000
def test_range(n):
    return for i in range(n):
        pass

def test_xrange(n):
    for i in xrange(n):
        pass
```

Python 2

```
print 'Python', python_version()

print 'ntiming range()'
%timeit test_range(n)

print 'nntiming xrange()'
%timeit test_xrange(n)

Python 2.7.6

timing range()
1000 loops, best of 3: 433 µs per loop

timing xrange()
1000 loops, best of 3: 350 µs per loop
```

Python 3

```

print('Python', python_version())

print('ntiming range()')
%timeit test_range(n)

Python 3.4.1

timing range()
1000 loops, best of 3: 520 µs per loop

print(xrange(10))

-----
NameError Traceback (most recent call last)
in ()
----> 1 print(xrange(10))

NameError: name 'xrange' is not defined

```

Python 3 中的 range 对象中的 `__contains__` 方法

另一个值得一提的是，在 Python 3.x 中，range 有了一个新的 `__contains__` 方法。`__contains__` 方法可以有效的加快 Python 3.x 中整数和布尔型的“查找”速度。

```

x = 10000000
def val_in_range(x, val):
    return val in range(x)

def val_in_xrange(x, val):
    return val in xrange(x)

print('Python', python_version())
assert(val_in_range(x, x/2) == True)
assert(val_in_range(x, x//2) == True)
%timeit val_in_range(x, x/2)
%timeit val_in_range(x, x//2)

Python 3.4.1
1 loops, best of 3: 742 ms per loop
1000000 loops, best of 3: 1.19 µs per loop

```

根据上面的 `timeit` 的结果，查找整数比查找浮点数要快大约 6 万倍。但由于 Python 2.x 中的 `range` 或 `xrange` 没有 `__contains__` 方法，所以在 Python 2 中的整数和浮点数的查找速度差别不大。

Python 2.7.x 和 3.x 版本的重要区别

```
print 'Python', python_version()

assert(val_in_xrange(x, x/2.0) == True)
assert(val_in_xrange(x, x/2) == True)
assert(val_in_range(x, x/2) == True)
assert(val_in_range(x, x//2) == True)
%timeit val_in_xrange(x, x/2.0)
%timeit val_in_xrange(x, x/2)
%timeit val_in_range(x, x/2.0)
%timeit val_in_range(x, x/2)
```

```
Python 2.7.7
1 loops, best of 3: 285 ms per loop
1 loops, best of 3: 179 ms per loop
1 loops, best of 3: 658 ms per loop
1 loops, best of 3: 556 ms per loop
```

下面的代码证明了 Python 2.x 中没有 `__contains__` 方法：

```
print('Python', python_version())
range.__contains__

Python 3.4.1
<slot wrapper '__contains__' of 'range' objects

print('Python', python_version())
range.__contains__

Python 2.7.7
-----
AttributeError Traceback (most recent call last)
<ipython-input-7-05327350dafb> in <module>()
1 print 'Python', python_version()
----> 2 range.__contains__

AttributeError: 'builtin_function_or_method' object has no attribute '__contains__'

print('Python', python_version())
xrange.__contains__

Python 2.7.7
-----
AttributeError Traceback (most recent call last)
in ()
1 print 'Python', python_version()
----> 2 xrange.__contains__

AttributeError: type object 'xrange' has no attribute '__contains__'
```

关于 Python 2 中 xrange() 与 Python 3 中 range() 之间的速度差异的一点说明：

有读者指出了 Python 3 中的 range() 和 Python 2 中 xrange() 执行速度有差异。由于这两者的实现方式相同，因此理论上执行速度应该也是相同的。这里的速度差别仅仅是因为 Python 3 的总体速度就比 Python 2 慢。

```
def test_while():
    i = 0
    while i < 20000:
        i += 1
    return

print('Python', python_version())
%timeit test_while()

Python 3.4.1
%timeit test_while()
100 loops, best of 3: 2.68 ms per loop

print 'Python', python_version()
%timeit test_while()

Python 2.7.6
1000 loops, best of 3: 1.72 ms per loop
```

触发异常

Python 2 支持新旧两种异常触发语法，而 Python 3 只接受带括号的语法（不然会触发 SyntaxError）：

Python 2

```
print 'Python', python_version()

Python 2.7.6

raise IOError, "file error"

-----
IOError Traceback (most recent call last)
<ipython-input-8-25f049caebb0> in <module>()
----> 1 raise IOError, "file error"

IOError: file error

raise IOError("file error")

-----
IOError Traceback (most recent call last)
<ipython-input-9-6f1c43f525b2> in <module>()
----> 1 raise IOError("file error")

IOError: file error
```

Python 3

```
print('Python', python_version())

Python 3.4.1

raise IOError, "file error"

File "<ipython-input-10-25f049caebb0>", line 1
raise IOError, "file error"
^
SyntaxError: invalid syntax
The proper way to raise an exception in Python 3:

print('Python', python_version())
raise IOError("file error")

Python 3.4.1

-----
OSError Traceback (most recent call last)
<ipython-input-11-c350544d15da> in <module>()
1 print('Python', python_version())
----> 2 raise IOError("file error")

OSError: file error
```

异常处理

Python 3 中的异常处理也发生了一点变化。在 Python 3 中必须使用“as”关键字。

Python 2

```
print 'Python', python_version()
try:
    let_us_cause_a_NameError
except NameError, err:
    print err, '--> our error message'

Python 2.7.6
name 'let_us_cause_a_NameError' is not defined --> our error message
```

Python 3

```
print('Python', python_version())
try:
    let_us_cause_a_NameError
except NameError as err:
    print(err, '--> our error message')

Python 3.4.1
name 'let_us_cause_a_NameError' is not defined --> our error message
```

next() 函数和.next() 方法

由于会经常用到 next() (.next()) 函数（方法），所以还要提到另一个语法改动（实现方面也做了改动）：在 Python 2.7.5 中，函数形式和方法形式都可以使用，而在 Python 3 中，只能使用 next() 函数（试图调用 .next() 方法会触发 AttributeError）。

Python 2

```
print 'Python', python_version()
my_generator = (letter for letter in 'abcdefg')
next(my_generator)
my_generator.next()

Python 2.7.6
'b'
```

Python 3

```
print('Python', python_version())
my_generator = (letter for letter in 'abcdefg')
next(my_generator)

Python 3.4.1
'a'

my_generator.next()

-----
AttributeError Traceback (most recent call last)
<ipython-input-14-125f388bb61b> in <module>()
----> 1 my_generator.next()

AttributeError: 'generator' object has no attribute 'next'
```

For循环变量与全局命名空间泄漏

好消息是：在Python 3.x中，for循环中的变量不再会泄漏到全局命名空间中了！

这是Python 3.x中做的一个改动，在“What’s New In Python 3.0”中有如下描述：

“列表推导不再支持[... for var in item1, item2, ...]这样的语法，使用[... for var in (item1, item2, ...)]代替。还要注意列表推导有不同的语义：现在列表推导更接近list()构造器中的生成器表达式这样的语法糖，特别要注意的是，循环控制变量不会再泄漏到循环周围的空间中了。”

Python 2

```
print 'Python', python_version()

i = 1
print 'before: i =', i

print 'comprehension: ', [i for i in range(5)]

print 'after: i =', i

Python 2.7.6
before: i = 1
comprehension: [0, 1, 2, 3, 4]
after: i = 4
```

Python 3

```
print('Python', python_version())

i = 1
print('before: i =', i)

print('comprehension:', [i for i in range(5)])

print('after: i =', i)

Python 3.4.1
before: i = 1
comprehension: [0, 1, 2, 3, 4]
after: i = 1
```

比较无序类型

Python 3 中另一个优秀的改动是，如果我们试图比较无序类型，会触发一个 `TypeError`。

Python 2

```
print 'Python', python_version()
print "[1, 2] > 'foo' = ", [1, 2] > 'foo'
print "(1, 2) > 'foo' = ", (1, 2) > 'foo'
print "[1, 2] > (1, 2) = ", [1, 2] > (1, 2)

Python 2.7.6
[1, 2] > 'foo' = False
(1, 2) > 'foo' = True
[1, 2] > (1, 2) = False
```

Python 3

```
print('Python', python_version())
print("[1, 2] > 'foo' = ", [1, 2] > 'foo')
print("(1, 2) > 'foo' = ", (1, 2) > 'foo')
print("[1, 2] > (1, 2) = ", [1, 2] > (1, 2))

Python 3.4.1
-----
TypeError Traceback (most recent call last)
<ipython-input-16-a9031729f4a0> in <module>()
 1 print('Python', python_version())
 2 print("[1, 2] > 'foo' = ", [1, 2] > 'foo')
 3 print("(1, 2) > 'foo' = ", (1, 2) > 'foo')
 4 print("[1, 2] > (1, 2) = ", [1, 2] > (1, 2))
TypeError: unorderable types: list() > str()
```

通过 `input()` 解析用户的输入

幸运的是，Python 3改进了`input()`函数，这样该函数就会总是将用户的输入存储为`str`对象。在Python 2中，为了避免读取非字符串类型会发生的一些危险行为，不得不使用`raw_input()`代替`input()`。

Python 2

```
Python 2.7.6
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> my_input = input('enter a number: ')

enter a number: 123

>>> type(my_input)
<type 'int'>

>>> my_input = raw_input('enter a number: ')

enter a number: 123

>>> type(my_input)
<type 'str'>
```

Python 3

```
Python 3.4.1
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> my_input = input('enter a number: ')
enter a number: 123
>>> type(my_input)
<class 'str'>
```

返回可迭代对象，而不是列表

在`xrange`一节中可以看到，某些函数和方法在Python 3中返回的是可迭代对象，而不像在Python 2中返回列表。

由于通常对这些对象只遍历一次，所以这种方式会节省很多内存。然而，如果通过生成器来多次迭代这些对象，效率就不高了。

此时我们的确需要列表对象，可以通过`list()`函数简单的将可迭代对象转成列表。

Python 2

```
print 'Python', python_version()

print range(3)
print type(range(3))

Python 2.7.6
[0, 1, 2]
<type 'list'>
```

Python 3

```
print('Python', python_version())
print(range(3))
print(type(range(3)))
print(list(range(3)))

Python 3.4.1
range(0, 3)
<class 'range'>
[0, 1, 2]
```

下面列出了Python 3中其他不再返回列表的常用函数和方法：

- zip()
- map()
- filter()
- 字典的.key()方法
- 字典的.value()方法
- 字典的.item()方法

更多关于Python 2和Python 3的文章

下面列出了其他一些可以进一步了解Python 2和Python 3的优秀文章，

//迁移到 Python 3

- [Should I use Python 2 or Python 3 for my development activity?](#)
- [What's New In Python 3.0](#)
- [Porting to Python 3](#)
- [Porting Python 2 Code to Python 3](#)
- [How keep Python 3 moving forward](#)

// 对Python 3的褒与贬

- 10 awesome features of Python that you can't use because you refuse to upgrade to Python 3
- 关于你不想知道的所有Python3 unicode特性
- Python 3 正在毁灭 Python
- Python 3 能振兴 Python
- Python 3 is fine