

目 录

致谢

阅前必读

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

第一部分 积小流，至江海

Python环境安装

集成开发环境 (IDE)

数的类型和四则运算

啰嗦的除法

开始真正编程

初识永远强大的函数

玩转字符串(1)

玩转字符串(2)

玩转字符串(3)

眼花缭乱的运算符

从if开始语句的征程

一个免费的实验室

有容乃大的list(1)

有容乃大的list(2)

有容乃大的list(3)

有容乃大的list(4)

list和str比较

画圈还不简单吗

再深点，更懂list

字典，你还记得吗？

字典的操作方法

有点简约的元组

一二三,集合了

集合的关系

Python数据类型总结

深入变量和引用对象

赋值，简单也不简单
坑爹的字符编码
做一个小游戏
不要红头文件(1)
不要红头文件(2)
第二部分 穷千里目，上一层楼
正规地说一句话
print能干的事情
从格式化表达式到方法
复习if语句
用while来循环
难以想象的for
关于循环的小伎俩
让人欢喜让人忧的迭代
大话题小函数(1)
大话题小函数(2)
python文档
重回函数
变量和参数
总结参数的传递
传说中的函数条规
关于类的基本认识
编写类之一创建实例
编写类之二方法
编写类之三子类
编写类之四再论继承
命名空间
类的细节
Import 模块
模块的加载
私有和专有
折腾一下目录
第三部分 昨夜西风，亭台谁登
网站的结构

- 通过Python连接数据库
 - 用Python操作数据库(1)
 - 用Python操作数据库 (2)
 - 用Python操作数据库 (3)
- python开发框架
 - Hello,第一个网页分析
 - 实例分析get和post
 - 问候世界
 - 使用表单和模板
 - 模板中的语法
 - 静态文件以及一个项目框架
 - 模板转义
- 第四部分 暮然回首，灯火阑珊处
 - requests库
 - 比较json/dictionary的库
 - defaultdict 模块和 namedtuple 模块
- 第五部分 Python备忘录
 - 基本的（字面量）值
 - 运算符
 - 常用的内建函数
- 人生苦短，我用Python

致谢

当前文档《老齐 零基础学Python》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-02-21。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地

址：<http://www.bookstack.cn/books/StarterLearningPython>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

阅前必读

- [This is for everyone.](#)
- [零基础学Python](#)
- [第零部分 独上高楼，望尽天涯路](#)
- [第一部分 积小流，至江海](#)
- [第二部分 穷千里目，上一层楼](#)
- [第三部分 昨夜西风，亭台谁登](#)
- [第四部分 暮然回首，灯火阑珊处](#)
- [第五部分 Python备忘录](#)
 - [内容还在不断更新，欢迎follow me。](#)
- [有钱的捧个钱场，有人的捧个人场。如果看官认可本教程，并愿意打赏，当感激涕零。您的资助将是我继续并且奉献更好内容的动力。](#)
- [支付宝账号：qiwsir@126.com](#)
- [扩展阅读\(来自网络文章\)](#)

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](#)

第一部分 积小流，至江海

python的最基础支持

1. [Python环境安装](#): 在不同操作系统的安装方法, 特别演示了如何编译源码安装
2. [集成开发环境 \(IDE\)](#): IDE的使用和 hello world
3. [数的类型和四则运算](#): 数的整型/浮点型, 四则运算和基本的运算函数 (绝对值/四舍五入/取整/幂/开方)
4. [啰嗦的除法](#): 除法(含division)、余数和四舍五入
5. [开始真正编程](#)
6. [初识永远强大的函数](#)
7. [玩转字符串\(1\)](#): 基本概念、字符转义、字符串连接、变量与字符串关系
8. [玩转字符串\(2\)](#)
9. [玩转字符串\(3\)](#)
0. [眼花缭乱的运算符](#)
1. [从if开始语句的征程](#)
2. [一个免费的实验室](#)
3. [有容乃大的list\(1\)](#): 定义、索引、对list反转和追加元素
4. [有容乃大的list\(2\)](#)
5. [有容乃大的list\(3\)](#)
6. [有容乃大的list\(4\)](#)
7. [list和str比较](#)
8. [画圈还不简单吗](#)
9. [再深点, 更懂list](#)
0. [字典, 你还记得吗?](#)
1. [字典的操作方法](#)
2. [有点简约的元组](#)
3. [一二三, 集合了](#)
4. [集合的关系](#)
5. [Python数据类型总结](#)

6. 深入变量和引用对象
7. 赋值，简单也不简单
8. 坑爹的字符编码
9. 做一个小游戏
0. 不要红头文件(1): open, write, close
1. 不要红头文件(2): os.stat, closed, mode, read, readlines, readline

第二部分 穷千里目，上一层楼

python中常用的语句和类

1. 正规地说一句话
2. print能干的事情
3. 从格式化表达式到方法
4. 复习if语句: if基础、三元操作符
5. 用while来循环
6. 难以想象的for
7. 关于循环的小伎俩
8. 让人欢喜让人忧的迭代
9. 大话题小函数(1)
0. 大话题小函数(2)
1. python文档
2. 重回函数
3. 变量和参数
4. 总结参数的传递
5. 传说中的函数条规
6. 关于类的基本认识
7. 编写类之一创建实例

8. [编写类之二方法](#)
9. [编写类之三子类](#)
0. [编写类之四再论继承](#)
1. [命名空间](#)
2. [类的细节](#)
3. [Import 模块](#)
4. [模块的加载](#)
5. [私有和专有](#)
6. [折腾一下目录：os.path.](#)

第三部分 昨夜西风，亭台谁登

用python来做网站

1. [网站的结构](#)：网站组成、MySQL数据库的安装和配置、MySQL的运行
2. [通过Python连接数据库](#)：安装python-MySQLdb，连接MySQL
3. [用Python操作数据库\(1\)](#)：建立连接和游标，并insert and commit
4. [用Python操作数据库\(2\)](#)
5. [用Python操作数据库\(3\)](#)
6. [python开发框架](#)：框架介绍、Tornado安装
7. [Hello, 第一个网页分析](#)：tornado网站的基本结构剖析：improt模块、RequestHandler, HTTPServer, Application, IOLoop
8. [实例分析get和post](#)：get()通过URL得到数据和post()通过get_argument()获取数据
9. [问候世界](#)：利用GAE建立tornado框架网站
0. [使用表单和模板](#)：tornado模板self.render和模板变量传递

1. [模板中的语法](#)：tornado模板中的for, if, set等语法，以{%开始，%}语句{%end%}
2. [静态文件以及一个项目框架](#)：静态目录的建立以及一个基本的项目框架演示
3. [模板转义](#)：模板自动转义以及不转义方法，url字符转义方法

第四部分 暮然回首，灯火阑珊处

python模块（第三方库）说明

1. [requests库](#)：针对http的库，比如post, get等
2. [比较json/dictionary的库](#)：比较两个json，直到最底层，并将结果格式化为dict格式
3. [defaultdict 模块](#)和 [namedtuple 模块](#)：内容如题目

第五部分 Python备忘录

1. [基本的（字面量）值](#)
2. [运算符](#)
3. [常用的内建函数](#)

内容还在不断更新，欢迎follow me。

有钱的捧个钱场，有人的捧个人场。如果看官认可本教程，并愿意打赏，当感激涕零。您的资助将是我继续并且奉献更好内容的动力。

支付宝账号：qiwsir@126.com

感谢支持我的朋友：

陈贤民、李航、王庆、罗苏平、杜明、曹睿、任笠斐、陈艳阳、彭冬碧、范钦武、余威、王志翔

扩展阅读(来自网络文章)

1. [人生苦短，我用Python](#)

第零部分 独上高楼，望尽天涯路

链接

- [唠叨一些关于python的事情](#)
- [开始本栏目的原因](#)

唠叨一些关于python的事情

- 唠叨一些关于Python的事情
 - Python的昨天今天和明天
 - Python的历史
 - Python的现在
 - Python的未来
 - Python的特点
 - python哲学
 - The Zen of Python
 - 准备

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.

唠叨一些关于Python的事情

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（这个我喜欢，可惜多年之后，已经好久没有用过了）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它事物，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，做为刚刚开始学习python的朋友，可千万别认为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。

请列位多向所有倡导“开放”的牛人们表示敬意，是他们让这个世界更好了。他们以行动诠释了热力学第二定律——“熵增原理”。

Python的现在

有一次与某软件公司一个号称是CTO的人谈话，他问我用什么语言开发，我说用Python，估计是我的英语发音不好吧（我这回真的谦虚了一把），他居然听成了Pascal（也是一种高级语言，现在很少用了，曾经是比较流行的教学语言）。呜呼，Python是小众吗？不是，是那家伙眼界不开阔！接触过不少号称CTO的，多数是有几年经验的程序员，并没有以国际视野来看待技术，当然，大牛的CTO还是不少的。总之，不要被外表忽悠了，“不看广告，看疗效”。

首先看一张最近一期的编程语言排行



python在这个榜单中第8，也许看官心理在想：为什么我不去学那个排第一呢？如果您是一个零基础的学习者，我以多年的工作和教学经验正告：还是从入门比较容易的开始吧，python是这样的。等以后，完全可以拓展到其它语言。或许你又问了，php和vb是不是可以呢？他们排名比python靠前。回答是：当然可以。但是，学习一种入门的语言，要多方考虑，或许以后你就不想学别的，想用这个包打天下了，那就只有python。并且，还得看下面的信息：

根据Dice.com一项网上对20000名IT专业人士进行调查的结果：

java类平均工资：91060美元；

python类平均工资：90208美元；

不错，python程序员平均来讲，比java平均工资低，但看看差距，再看看两者的入门门槛，就知道，学习python绝对是一个性价比非常高的投资啦。

Python就是这样，默默地拓展着它的领域。

Python的未来

未来，要靠列位来做了，你学好了，用好了，未来它就光明了。它的未来在你手里。

Python的特点

很多高级语言都宣称自己是简单的、入门容易的，并且具有普适性的。真正做到这些的，不忽悠的，只有Python。有朋友做了一件衬衫，上面写着“生命有限，我用Python”，这说明什么？它有着简单、开发速

度快，节省时间和精力。因为它是开放的，有很多可爱的开发者（为开放社区做贡献的开发者，是最可爱的人），将常用的功能做好了，放在网上，谁都可以拿过来使用。这就是Python，这就是开放。

抄一段严格的描述，来自维基百科：

*Python*是完全面向对象的语言。函数、模块、数字、字符串都是对象。并且完全支持继承、重载、派生、多继承，有益于增强源代码的复用性。*Python*支持重载运算符，因此*Python*也支持泛型设计。相对于*Lisp*这种传统的函数式编程语言，*Python*对函数式设计只提供了有限的支持。有两个标准库（*functools*, *itertools*）提供了*Haskell*和*Standard ML*中久经考验的函数式程序设计工具。

虽然*Python*可能被粗略地分类为“脚本语言”（*script language*），但实际上一些大规模软件开发项目例如*Zope*、*Mnet*及*BitTorrent*，*Google*也广泛地使用它。*Python*的支持者较喜欢称它为一种高级动态编程语言，原因是“脚本语言”泛指仅作简单程序设计任务的语言，如*shell script*、*VBScript*等只能处理简单任务的编程语言，并不能与*Python*相提并论。

*Python*本身被设计为可扩充的。并非所有的特性和功能都集成到语言核心。*Python*提供了丰富的API和工具，以便程序员能够轻松地使用C、C++、*Cython*来编写扩充模块。*Python*编译器本身也可以被集成到其它需要脚本语言的程序内。因此，很多人还把*Python*作为一种“胶水语言”（*glue language*）使用。使用*Python*将其他语言编写的程序进行集成和封装。在*Google*内部的很多项目，例如*Google Engine*使用C++编写性能要求极高的部分，然后用*Python*或*Java/Go*调用相应的模块。《*Python*技术手册》的作者马特利（*Alex Martelli*）说：“这很难讲，不过，2004年，*Python*已在*Google*内部使用，*Google*招募许多*Python*高手，但在这之前就已决定使用*Python*。他们的目的是尽量使用*Python*，在不得已时改用C++；在操控硬件的场合使用C++，在快速开发时候使用*Python*。”

可能里面有一些术语还不是很理解，没关系，只要明白：*Python*是一种很牛的语言，应用简单，功能强大，*google*都在使用。这就足够了，足够让你下决心学习了。

python哲学

*Python*之所以与众不同，还在于它强调一种哲学理念，用黑字表示强调吧：

Python的设计哲学是“优雅”、“明确”、“简单”。

Python开发者的哲学是“用一种方法，最好是只有一种方法来做一件事。在设计Python语言时，如果面临多种选择，Python开发者一般会拒绝花俏的语法，而选择明确没有或者很少有歧义的语法。由于这种设计观念的差异，Python源代码通常具备更好的可读性，并且能够支撑大规模的软件开发。这些准则被称为Python格言。

The Zen of Python

1. `Beautiful is` better than `ugly`.
2. `Explicit is` better than `implicit`.
3. `Simple is` better than `complex`.
4. `Complex is` better than `complicated`.
5. `Flat is` better than `nested`.
6. `Sparse is` better than `dense`.
7. `Readability` counts.
8. `Special` cases aren't `special` enough to break the rules.
9. `Although` practicality beats purity.
10. `Errors` should never pass silently.
11. `Unless` explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.
13. There should be one-- and preferably only one --obvious way to do it.
14. `Although` that way may not be obvious at first unless you're `Dutch`.
15. `Now is` better than `never`.
16. `Although` `never is` often better than `*right*` now.
17. `If` the implementation `is` hard to explain, it's a bad idea.
18. `If` the implementation `is` easy to explain, it may be a good idea.
19. `Namespaces` are one honking great idea -- let's `do` more of those!

上面的诗来自Python官方，已经把前面唠叨的东西做了精美的概括。有中译本，[看这里](#)，本文摘抄一种中文翻译：

1. 优美胜于丑陋，明晰胜于隐晦

2. 简单胜于复杂，复杂胜于繁芜
3. 扁平胜于嵌套，稀疏胜于密集
4. 可读性很重要。
5. 虽然实用性比纯粹性更重要，
6. 但特例并不足以把规则破坏掉。
- 7.
8. 错误状态永远不要忽略，
9. 除非你明确地保持沉默，
10. 直面多义，永不臆断。
- 11.
12. 最佳的途径只有一条，然而他并非显而易见——谁叫你不是荷兰人？
- 13.
14. 置之不理或许会比慌忙应对要好，
15. 然而现在动手远比束手无策更好。
- 16.
17. 难以解读的实现不会是个好主意，
18. 容易解读的或许才是。
- 19.
20. 名字空间就是个顶呱呱好的主意。
- 21.
22. 让我们想出更多的好主意！

准备

已经描述了python的美好，开始学啦，做好如下准备：

- 电脑，必须的。不管是什么操作系统。
- 上网，必须的。没有为什么。

除了这些，还有一条，非常非常重要，写在最后：这是自己的兴趣。

[首页](#) | [下一讲：python安装](#)

开始本栏目的原因

- [为什么要开设此栏目](#)
 - [点击这里，进入本教程的目录](#)

为什么要开设此栏目

这个栏目的名称叫做“零基础学Python”。

现在网上已经有不少学习python的课程，其中也不乏精品。按理说，不缺少我这个基础类型的课程了。但是，我注意到一个问题，不管是课程还是出版的书，大多数是面向已经有一定编程经验的人写的或者讲的，也就是对这些朋友来讲，python已经不是他们的第一门高级编程语言。据我所知，目前国内很多大学都是将C之类的做为学生的第一门语言。

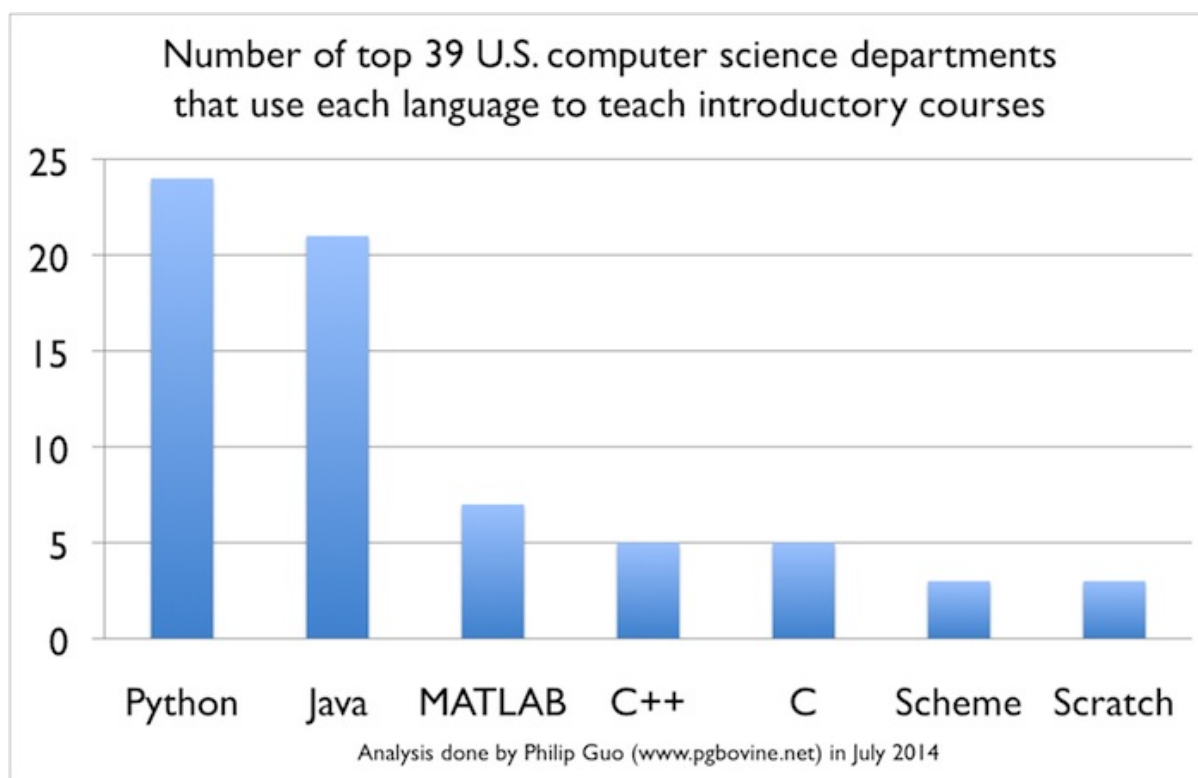
然而，在我看来，python是非常适合做为学习高级语言编程的第一门语言的。有一本书，名字叫《与孩子一起学编程》，这本书的定位，是将python定位为学习者学习的第一门高级编程语言。然而，由于读者对象是孩子，很多“成年人”不屑一顾，当然，里面的讲法与“实战”有点距离，导致以“找工作”、“工作需要”为目标的学习者，认为这本书跟自己要学的方向相差甚远。

为了弥补那本书的缺憾，我在这里推出面向成年人——大学生、或者其他想学习程序但是没有任何编程基础的朋友——学习第一门编程高级语言的教程。将Python做为学习高级语言编程的第一门语言，其优势在于：

- 入门容易，避免了其它语言的繁琐。
- 更接近我们的自然语言和平常的思维方法。
- 学习完这门语言之后，能够直接“实战”——用在工作上。
- 学习完这门语言之后，能够顺利理解并学习其它语言。

- python本身功能强大，一门语言也可以打天下，省却了以后的学习成本。

下面的图示统计显示：Python现在成为美国名校中最流行的编程入门语言。



[点击这里看上图来源](#)

综上，我有了这样一个冲动，做一个栏目，面对零基础要学习Python的朋友，面对将python做为第一门高级语言的朋友。这就是开始本栏目的初衷。

[点击这里，进入本教程的目录](#)

第一部分 积小流，至江海

链接

- [Python环境安装](#)
- [集成开发环境 \(IDE \)](#)
- [数的类型和四则运算](#)
- [啰嗦的除法](#)
- [开始真正编程](#)
- [初识永远强大的函数](#)
- [玩转字符串\(1\)](#)
- [玩转字符串\(2\)](#)
- [玩转字符串\(3\)](#)
- [眼花缭乱的运算符](#)
- [从if开始语句的征程](#)
- [一个免费的实验室](#)
- [有容乃大的list\(1\)](#)
- [有容乃大的list\(2\)](#)
- [有容乃大的list\(3\)](#)
- [有容乃大的list\(4\)](#)
- [list和str比较](#)
- [画圈还不简单吗](#)
- [再深点，更懂list](#)
- [字典，你还记得吗？](#)
- [字典的操作方法](#)
- [有点简约的元组](#)
- [一二三, 集合了](#)

- [集合的关系](#)
- [Python数据类型总结](#)
- [深入变量和引用对象](#)
- [赋值，简单也不简单](#)
- [坑爹的字符编码](#)
- [做一个小游戏](#)
- [不要红头文件\(1\)](#)
- [不要红头文件\(2\)](#)

Python环境安装

- Python安装
 - Linux系统的安装
 - windows系统的安装
 - Mac OS X系统的安装
 - 用ActivePython安装
 - 用源码安装

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

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

刚才又有了一个术语——“程序”，什么是程序？本文就不讲了。如果列为观众不是很理解这个词语，请上网google一下。

注：推荐一种非常重要的学习方法

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

“靠，那看你胡扯吗？”

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

列位看好了，我上面写的很清楚，是google一下，不是让大家去用那个什么度来搜索，那个搜索是专用搜索八卦、假药、以及各种穿着很简单衣服的女孩子照片的。如果你真的要提高自己的技术视野并且专心研究技术问题，请用google。当然，我知道你在用的时候会遇到困难，做为一个要在技术上有点成就的人，一定要学点上网的技术的，你懂得。

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

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。可以说，python3是未来，它比python2.7有进步。但是，现在，还有很多东西没有完全兼容python3。更何况，如果学了python2.7，对于python3，也只是某些地方的小变化了。

所以，我这里是用python2.7为例子来讲授的。

Linux系统的安装

看官所用的计算机是什么操作系统的？自己先弄懂。如果是Linux某个发行版，就跟我同道了。并且我恭喜你，因为以后会安装更多的一些

python库（模块），在这种操作系统下，操作非常简单，当然，如果是iOS，也一样，因为都是UNIX下的蛋。只是windows有点另类了。

不过，没关系，python就是跨平台的。

我以ubuntu 14.04为例，所有用这个操作系统的朋友（肯定很少啦），你们肯定会在shell中输入python，如果看到了>>>，并且显示出python的版本信息，恭喜你，因为你的系统已经自带了python的环境。的确，ubuntu内置了python环境。

我非要自己安装一遍不可。那就这么操作吧：

- 下载源码，目前最新版本是2.7.8，如果以后换了，可以在下面的命令中换版本号

- 源码也可以在网站上下载，具体见前述下载页面

```
wget
```

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

- 解压源码包

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

- 编译

```
cd Python-2.7.8
```

```
./configure --prefix=/usr/local #指定了目录
```

```
make&&make install
```

以上步骤，是我从网上找来的，供参考。因为我的机器早就安装了，不想折腾。安装好之后，进入shell，输入python，会看到如下：

```
1. qw@qw-Latitude-E4300:~$ python
2. Python 2.7.6 (default, Nov 13 2013, 19:24:16) #后来我升级到2.7.8了,
   就是用后面讲到的源码安装方法
3. [GCC 4.6.3] on linux2
4. Type "help", "copyright", "credits" or "license" for more
   information.
5. >>>
```

恭喜你，安装成功了。我用的是python2.7.6，或许你的版本号更高。

windows系统的安装

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

特别注意，安装完之后，需要检查一下，在环境变量是否有python。

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

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

Mac OS X系统的安装

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

如果按照以上方法，顺利安装成功，只能说明幸运，无它。如果没有安装成功，这是提高自己的绝佳机会，因为只有遇到问题才能解决问题，

才能知道更深刻的道理，不要怕，有google，它能帮助列为看官解决所有问题。当然，加入Q Q群或者通过微博，问我也可以。

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

ActivePython

用ActivePython安装

这个ActivePython是一个面向好多种操作系统的Python 套件, 它包含了一个完整的 Python 发布、一个适用于 Python 编程的 IDE 以及一些 Python的。有兴趣的看官可以到其官网浏览：<http://www.activestate.com>

用源码安装

python是开源的，它的源码都在网上。有高手朋友，如果愿意用源码来安装，亦可，请

到：<https://www.python.org/ftp/python/>，下载源码安装。

简单记录一下我的安装方法（我是在linux系统中做的）：

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

7. 祝你幸运

8. 安装完毕

OK! 已经安装好之后，马上就可以开始编程了。

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

[首页](#) | [下一讲](#)

集成开发环境 (IDE)

- 集成开发环境(IDE)
 - 值得纪念的时刻: Hello world
 - 集成开发环境
 - Python的IDE

"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."

集成开发环境(IDE)

当安装好python之后，其实就已经可以进行开发了。下面我们开始写第一行python代码。

值得纪念的时刻: Hello world

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

像下面一样：

```
1. Python 2.7.6 (default, Nov 13 2013, 19:24:16)
2. [GCC 4.6.3] on linux2
3. Type "help", "copyright", "credits" or "license" for more
   information.
4. >>>
```

在 `>>>` 后面输入下面内容，并按回车。这就是见证奇迹的时刻。从这

一刻开始，一个从来不懂编程的你，就跨入了程序员行列，不管你的工作是不是编程，你都已经是程序员了，其标志就是你已经用代码向这个世界打招呼了。

```
1. >>> print "Hello, World"
2. Hello, World
```

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

说明：在下面的分解动作中，用到了一个符号：`#`，就是键盘上数字3上面的那个井号。这个符号，在python编程中，表示注释。所谓注释，就是在计算机不执行，只是为了说明某行语句表达什么意思。

```
1. #看到">>>"符号，表示python做好了准备，当代你向她发出指令，让她做什么事情
2.
3. >>>
4.
5. #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
6.
7. >>> print
8.
9. #"Hello,World"是打印的内容，注意，量变的双引号，都是英文状态下的。引号不是打印
   内容，它相当于一个包裹，把打印的内容包起来，统一交给python。
10.
11. >>> print "Hello, World"
12.
13. #上面命令执行的结果。python接收到你要求她所做的事情：打印Hello,World，于是她
   就老老实实在地执行这个命令，丝毫不走样。
14.
15. Hello, World
```

祝贺，伟大的程序员。

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打字：*Hello World*

从此，进入了程序员行列，但是，看官有没有感觉，程序员用的这个工具，就是刚才打印Hello, World的那个cmd或者shell，是不是太简陋了？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个PPT播放自己都不相信的新理念呢，难道我们伟大的程序员，就用这么简陋的工具写出旷世代码吗？

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

IDE就是程序员的工具。

集成开发环境

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

下面就直接抄[维基百科上的说明](#)了：

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。有時IDE還會包含版本控制系統和一些可以設計圖形用戶界面的工具。許多支援物件導向的現代化IDE還包括了類別瀏覽器、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。



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。

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

那么做为零基础的学习者，用什么好呢？

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

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



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

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

后面我们所有的编程，就在这里完成了。这就是伟大程序员用的第一个IDE。

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。且看下回分解。

[首页](#) | [上一讲](#) | [下一讲](#)

数的类型和四则运算

- 用Python计算
 - 复习
 - 四则运算
 - 在计算机中，四则运算和小学数学中学习过的四则运算规则是一样的
 - 几个常见函数
 - 总结

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"

用Python计算

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

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，数学的基础知识各位肯定过关了。

复习

还是先来重温一下伟大时刻，打印hello world.

打开电脑，让python idle运行起来，然后输入：

```
1. >>> print 'Hello, World'
```

2. Hello, World

细心的看官，是否注意到，我在这里用的是单引号，上次用的是双引号。两者效果一样，也就是在这种情况下，单引号和双引号是一样的效果，一定要是成对出现的，不能一半是单引号，另外一半是双引号。

四则运算

按照下面要求，在ide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

```
1. >>> 2+5
2. 7
3. >>> 5-2
4. 3
5. >>> 10/2
6. 5
7. >>> 5*2
8. 10
9. >>> 10/5+1
10. 3
11. >>> 2*3-4
12. 2
```

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

另外，我相信看官已经发现了一个重要的公理：

在计算机中，四则运算和小学数学中学习过的四则运算规则是一样的

要不说是高等动物呢，自己发明的东西，一定要继承自己已经掌握的知识，别跟自己的历史过不去。伟大的科学家们，在当初设计计算机的

时候就想到列位现在学习的需要了，一定不能让后世子孙再学新的运算规则，就用小学数学里面的好了。感谢那些科学家先驱者，泽被后世。

下面计算三个算术题，看看结果是什么

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

看官可能愤怒了，这么简单的题目，就不要劳驾计算机了，太浪费了。

别着急，还是要在ide中运算一下，然后看看结果，有没有不一样？要仔细观察哦。

```
1. >>> 4+2
2. 6
3. >>> 4.0+2
4. 6.0
5. >>> 4.0+2.0
6. 6.0
```

不一样的地方是：第一个式子结果是6，后面两个是6.0。

现在我们就要引入两个数据类型：整数和浮点数

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

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

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

比较好理解，整数，就是小学学过的整数；浮点数，就是小数。如果整数写成小数形式，比如4写成4.0，也就变成了浮点数。

爱学习，就要有探索精神。看官在网上google一下整数，会发现还有另外一个词：长整数（型）。顾名思义，就是比较长的整数啦。在有的语言中，把这个做为单独一类区分开，但是，在python中，我们不用管这个了。只要是整数，就只是整数，不用区分长短（以前版本区分），因为区分没有什么意思，而且跟小学学过的数学知识不协调。

还有一个问题，需要向看官交代一下，眼前可能用不到，但是会总有一些人用这个来忽悠你，当他忽悠你的时候，下面的知识就用到了。

整数溢出问题

这里有一篇专门讨论这个问题的文章，推荐阅读：[整数溢出](#)

对于其它语言，整数溢出是必须正视的，但是，在python里面，看官就无忧愁了，原因就是python为我们解决了这个问题，请阅读拙文：[大整数相乘](#)

ok!看官可以在IDE中实验一下大整数相乘。

```
1. >>>
    123456789870987654321122343445567678890098876*12334556677899900998765
2. 152278477193527562870044352587576277277562328362032444339019158937017
```

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

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

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

- 整数，用int表示，来自单词：integer

- 浮点数，用float表示，就是单词：float

可以用一个命令：`type(object)`来检测一个数是什么类型。

```
1. >>> type(4)
2. <type 'int'>      #4是int，整数
3. >>> type(5.0)
4. <type 'float'>    #5.0是float，浮点数
5. type(9887765442221122334455667788998877665544332211333444555666777889
6. <type 'long'>     #是长整数，也是一个整数
```

几个常见函数

在这里就提到函数，因为这个东西是经常用到的。什么是函数？如果看官不知道此定义，可以去google。貌似是初二数学讲的了。

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

求绝对值

```
1. >>> abs(10)
2. 10
3. >>> abs(-10)
4. 10
5. >>> abs(-1.2)
6. 1.2
```

四舍五入

```
1. >>> round(1.234)
2. 1.0
3. >>> round(1.234, 2)
4. 1.23
5.
```



```

6. >>> #如果不清楚这个函数的用法，可以使用下面方法看帮助信息
7. >>> help(round)
8.
9. Help on built-in function round in module __builtin__:
10.
11. round(...)
12.     round(number[, ndigits]) -> floating point number
13.
14.     Round a number to a given precision in decimal digits (default
15.     0 digits).
16.     This always returns a floating point number. Precision may be
17.     negative.

```

幂函数

```

1. >>> pow(2,3)      #2的3次方
2. 8

```

math模块（对于模块可能还有点陌生，不过不要紧，先按照下面代码实验一下，慢慢就理解了）

```

1. >>> import math      #引入math模块
2. >>> math.floor(32.8)  #取整，不是四舍五入
3. 32.0
4. >>> math.sqrt(4)     #开平方
5. 2.0

```

总结

- python里的加减乘除按照小学数学规则执行
- 不用担心大整数问题，python会自动处理
- `type(object)`是一个有用的东西

[首页](#) | [上一讲](#) | [下一讲](#)

啰嗦的除法

- 啰嗦的除法
 - 整数除以整数
 - 浮点数与整数相除
 - 引用模块解决除法—启用轮子
 - 关于余数
 - 四舍五入

"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)

啰嗦的除法

除法啰嗦，不仅是python。

整数除以整数

看官请在进入python交互模式之后（以后在本教程中，可能不再重复这类的叙述，只要看到>>>，就说明是在交互模式下，这个交互模式，看官可以在ide中，也可以像我一样直接在shell中运行python进入交互模式），练习下面的运算：

```
1. >>> 2/5
2. 0
3. >>> 2.0/5
4. 0.4
5. >>> 2/5.0
6. 0.4
7. >>> 2.0/5.0
8. 0.4
```

看到没有？麻烦出来了（这是在python2.x中），如果从小学数学知识除法，以上四个运算结果都应该是0.4。但我们看到的后三个符合，第一个居然结果是0。why？

因为，在python（严格说是python2.x中，python3会有所变化，具体看官要了解，可以去google）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）。

继续实验，验证这个结论：

```
1. >>> 5/2
2. 2
3. >>> 6/3
4. 2
5. >>> 5/2
6. 2
7. >>> 6/2
8. 3
9. >>> 7/2
10. 3
11. >>> 8/2
12. 4
13. >>> 9/2
14. 4
```

注意：这里是得到整数商，而不是得到含有小数位的结果后“四舍五入”。例如5/2，得到的是商2，余数1，最终5/2=2。并不是对2.5进行四舍五入。

浮点数与整数相除

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

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

出结论之前，还是先做实验：

```
1. >>> 9.0/2
2. 4.5
3. >>> 9/2.0
4. 4.5
5. >>> 9.0/2.0
6. 4.5
7. >>> 8.0/2
8. 4.0
9. >>> 8/2.0
10. 4.0
11. >>> 8.0/2.0
12. 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。

```
1. >>> 10.0/3
2. 3.3333333333333335
```

这个是不是就有点搞怪了，按照数学知识，应该是3.33333...，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。

关于无限循环小数问题，小学都学习了，但是这可不是一个简单问题，看看[维基百科的词条：0.999...](#)，会不会有深入体会呢？

总之，要用python，就得遵循她的规定，前面两条规定已经明确了。

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](#)

说明：以上除法规则，是针对python2，在python3中，将 $5/2$ 和 $5.0/2$ 等同起来了。不过，如果要得到那个整数部分的上，可以用另外一种方式：地板除。

```
1. >>> 9/2
2. 4
3. >>> 9//2
4. 4
```

python总会要提供多种解决问题的方案的，这是她的风格。

引用模块解决除法—启用轮子

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

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

怎么用？可以通过两种形式用：

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

不啰嗦了，实验一个：

```
1. >>> from __future__ import division
2. >>> 5/2
3. 2.5
4. >>> 9/2
5. 4.5
6. >>> 9.0/2
7. 4.5
8. >>> 9/2.0
9. 4.5
```

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

这就是轮子的力量。

关于余数

前面计算 $5/2$ 的时候，商是2，余数是1

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

实验下面的操作：

```
1. >>> 5%2
2. 1
```

```
3. >>> 9%2
4. 1
5. >>> 7%3
6. 1
7. >>> 6%4
8. 2
9. >>> 5.0%2
10. 1.0
```

符号：`%`，就是要得到两个数（可以是整数，也可以是浮点数）相除的余数。

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

```
1. >>> divmod(5,2) #表示5除以2，返回了商和余数
2. (2, 1)
3. >>> divmod(9,2)
4. (4, 1)
5. >>> divmod(5.0,2)
6. (2.0, 1.0)
```

四舍五入

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

动手试试：

```
1. >>> round(1.234567,2)
2. 1.23
3. >>> round(1.234567,3)
4. 1.235
5. >>> round(10.0/3,4)
6. 3.3333
```


简单吧。越简单的时候，越要小心，当你遇到下面的情况，就有点怀疑了：

```
1. >>> round(1.2345,3)
2. 1.234                #应该是：1.235
3. >>> round(2.235,2)
4. 2.23                #应该是：2.24
```

哈哈，我发现了python的一个bug，太激动了。

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

Note:

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

原来真的轮不到我。（垂头丧气状。）

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](#)。

[首页](#) | [上一讲](#) | [下一讲](#)

开始真正编程

- 开始真正编程
 - 用IDLE的编程环境
 - 写两个大字：Hello, World
 - 解一道题目

开始真正编程

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

的确。到目前位置，还不能算编程，只能算会用一些指令（或者叫做命令）来做点简单的工作。并且看官所在的那个IDE界面，也是输入指令用的。

列位稍安勿躁，下面我们就学习如何编写一个真正的程序。工具还是那个IDLE，但是，请大家谨记，对于一个真正的程序来讲，用什么工具是无所谓的，只要能够把指令写进去，比如用记事本也可以。

我去倒杯茶，列位先认真读一读下面一段，关于程序的概念，内容来自维基百科：

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

A computer program, or just a program, is a sequence of instructions, written to perform a specified task with a computer.[1] A computer requires programs to function, typically executing the program's instructions in a central processor.[2] The program has an executable form that the computer can use directly to execute the instructions. The same program in its human-readable source code form, from which

executable programs are derived (e.g., compiled), enables a programmer to study and develop its algorithms. A collection of computer programs and related data is referred to as the software.

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

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

• 计算机程序

计算机程序 (Computer Program) 是指一组指示计算机或其他具有信息处理能力装置每一步动作的指令，通常用某种程序设计语言编写，运行于某种目标体系结构上。打个比方，一个程序就像一个用汉语（程序设计语言）写下的红烧肉菜谱（程序），用于指导懂汉语和烹饪手法的人（体系结构）来做这个菜。通常，计算机程序要经过编译和链接而成为一种人们不易看清而计算机可解读的格式，然后运行。未经编译就可运行的程序，通常称之为脚本程序 (script)。

碧螺春，是我最喜欢的了。有人要送礼给我，请别忘记了。难道我期望列位看官会送吗？哈哈哈

废话少说，开始说程序。程序，简而言之，就是指令的集合。但是，有的程序需要编译，有的不需要。python编写的程序就不需要，因此她也被称之为脚本程序。特别提醒列位，不要认为编译的就好，不编译的就不好；也不要认为编译的就“高端”，不编译的就属于“低端”。有一些做了很多年程序的程序员或者其它什么人，可能会有这样的想法，这是毫无根据的。

不争论。用得妙就是好。

用IDLE的编程环境

操作：File->New window



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



写两个大字：Hello,World

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

直接上代码，就这么一行即可。

```
1. 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，在下拉的里面选择Run Moudle



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

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

成功了吗？成功了也别兴奋，因为还没有到庆祝的时候。

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

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

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



然后在这个shell里面，输入：`python 105.py`

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

我的计算机我做主。于是它给我乖乖地执行了这条命令。如下图：



还在沉默？可以欢呼了，德国队7:1胜巴西队，列看官中，不管是德国

队还是巴西队的粉丝，都可以欢呼，因为你在程序员道路上迈出了伟大的第二步。顺便预测一下，本届世界杯最终冠军应该是：中国队。（有这么扯的吗？）

解一道题目

请计算： $19+2*4-8/2$

代码如下：

```
1. #coding:utf-8
2.
3. """
4. 请计算：19+2*4-8/2
5. """
6.
7. a = 19+2*4-8/2
8. 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：



上面代码中，第一行，不能少，本文件是能够输入汉字的，否则汉字无法输入。

好像还是比较简单。

别着急。复杂的在后面呢。

[首页](#) | [上一讲](#) | [下一讲](#)

初识永远强大的函数

- 永远强大的函数
 - 深入理解函数
 - 变量不仅仅是数
 - 变量本质——占位符
- 给变量赋值
- 建立简单函数
- 建立实用的函数
- `coding:utf-8` 声明本文件中代码的字符集类型是utf-8格式。
初学者如果还不理解，一方面可以去google，另外还可放一放，就先这么抄写下来，以后会讲解。
 - 声明函数的格式为：
- 取名字的学问

永远强大的函数

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，乃至经济学、政治学、社会学等，都已经普遍使用函数。

下面一段来自维基百科（在本教程中，大量的定义来自维基百科，因为它真的很百科）：[函数词条](#)

函数这个数学名词是莱布尼兹在1694年开始使用的，以描述曲线的一个相关量，如曲线的斜率或者曲线上的某一点。莱布尼兹所指的函数现在被称作可导函数，数学家之外的普通人一般接触到的函数即属此类。对于可导函数可以讨论它的极限和导数。此两者描述了函数输出值的变化同输入值变化的关系，是微积分学的基础。

中文的“函数”一词由清朝数学家李善兰译出。其《代数学》书中解释：“凡此變數中函（包含）彼變數者，則此為彼之函數”。

函数，从简单到复杂，各式各样。前面提供的维基百科中的函数词条，里面可以做一个概览。但不管什么样子的函数，都可以用下图概括：



有初中数学水平都能理解一个大概了。这里不赘述。

本讲重点说明用python怎么来做一个函数用一用。

深入理解函数

在中学数学中，可以用这样的方式定义函数： $y=4x+3$ ，这就是一个一次函数，当然，也可以写成： $f(x)=4x+3$ 。其中 x 是变量，它可以代表任何数。

1. 当 $x=2$ 时，代入到上面的函数表达式：
2. $f(2) = 4*2+3 = 11$
3. 所以： $f(2) = 11$

以上对函数的理解，是一般初中生都能打到的。但是，如果看官已经初中毕业了，或者是一个有追求的初中生，还不能局限在上面的理解，还要将函数的理解拓展。

变量不仅仅是数

变量 x 只能是任意数吗？其实，一个函数，就是一个对应关系。看官尝试着将上面表达式的 x 理解为馅饼， $4x+3$ ，就是4个馅饼在加上3（单位是什么，就不重要了），这个结果对应着另外一个东西，那个东西比如说就是iphone。或者说可以理解为4个馅饼加3就对应一个iphone。这就是所谓映射关系。

所以， x ，不仅仅是数，可以是你认为的任何东西。

变量本质——占位符

函数中为什么变量用x？这是一个有趣的问题，自己google一下，看能不能找到答案。

我也不清楚原因。不过，我清楚地知道，变量可以用x，也可以用别的符号，比如y,z,k,i,j...，甚至用alpha,beta,qiwei,qiwsir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到的要复杂。但是，现在我们先不管那些，复杂东西放在以后再说了。现在，就按照初中数学来研究python中的变量

通常使小写字母来命名python中的变量，也可以在其中加上下划线什么的，表示区别。

比如：alpha,x,j,p_beta，这些都可以做为python的变量。

给变量赋值

打开IDLE，实验操作如下：

```
1. >>> a = 2    #注1
2. >>> a        #注2
3. 2
4. >>> b = 3    #注3
5. >>> c = 3
6. >>> b
7. 3
8. >>> c
```

```
9. 3
10. >>>
```

说明：

- 注1：a=2的含义是将一个变量a指向了2这个数，就好比叫做a是的钓鱼的人，通过鱼线，跟一条叫做2的鱼连接者，a通过鱼线就可以导出2
- 注2：相当于要a这个钓鱼的人，顺着鱼线导出那条鱼，看看连接的是哪一条，发现是叫做2的那条傻鱼
- 注3：b=3，理解同上。那么c=3呢？就是这条叫做3的鱼被两个人同时钓到了。

建立简单函数

```
1. >>> a = 2
2. >>> y=3*a+2
3. >>> y
4. 8
```

这种方式建立的函数，跟在初中数学中学习的没有什么区别。当然，这种方式的函数，在编程实践中的用途不大，一般是在学习阶段理解函数来使用的。

别急躁，你在输入a=3, 然后输入y，看看得到什么结果呢？

```
1. >>> a=2
2. >>> y=3*a+2
3. >>> y
4. 8
5. >>> a=3
6. >>> y
7. 8
```

是不是很奇怪？为什么后面已经让a等于3了，结果y还是8。

用前面的钓鱼理论就可以解释了。a和2相连，经过计算，y和8相连了。后面a的连接对象修改了，但是y的连接对象还没有变，所以，还是8。再计算一次，y的连接对象就变了：

```
1. >>> a=3
2. >>> y
3. 8
4. >>> y=3*a+2
5. >>> y
6. 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。

```
1. >>> y=3*a+2
2. Traceback (most recent call last):
3.   File "<stdin>", line 1, in <module>
4.   NameError: name 'a' is not defined
```

注意看错误提示，a是一个变量，提示中告诉我们这个变量没有定义。显然，如果函数中要使用某个变量，不得不提前定义出来。定义方法就是给这个变量赋值。

建立实用的函数

上面用命令方式建立函数，还不够“正规化”，那么就来写一个.py文件吧。

在IDLE中，File->New window

然后输入如下代码：

```
1. #coding:utf-8
2.
3. def add_function(a,b):
4.     c = a+b
5.     print c
6.
7. if __name__=="__main__":
8.     add_function(2,3)
```

然后将文件保存，我把她命名为106-1.py，你根据自己的喜好取个名字。

然后我就进入到那个文件夹，运行这个文件，出现下面的结果，如图：



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：

- `coding:utf-8` 声明本文件中代码的字符集类型是utf-8格式。初学者如果还不理解，一方面可以去google，另外还可放一放，就先这么抄写下来，以后会讲解。
- `def add_function(a,b):` 这里是函数的开始。在声明要建立一个函数的时候，一定要使用def（def 就是英文define的前三个字母），意思就是告知计算机，这里要声明一个函数；`add_function`是这个函数名称，取名字是有讲究的，就好比你的名字一样。在python中取名字的讲究就是要有一定意义，能够从名字中看出这个函数是用来干什么的。从`add_function`这个名字

中，是不是看出她是用来计算加法的呢？(a, b)这个括号里面的是这个函数的参数，也就是函数变量。冒号，这个冒号非常非常重要，如果少了，就报错了。冒号的意思就是下面好开始真正的函数内容了。

- `c=a+b` 特别注意，这一行比上一行要缩进四个空格。这是python的规定，要牢记，不可丢掉，丢了就报错。然后这句话就是将两个参数(变量)相加，结果赋值与另外一个变量c。
- `print c` 还是提醒看官注意，缩进四个空格。将得到的结果c的值打印出来。
- `if name=="main":` 这句话先照抄，不解释。注意就是不缩进了。
- `add_function(2,3)` 这才是真正调用前面建立的函数，并且传入两个参数：a=2, b=3。仔细观察传入参数的方法，就是把2放在a那个位置，3放在b那个位置（所以说，变量就是占位符）。

解牛完毕，做个总结：

声明函数的格式为：

```
1. def 函数名(参数1, 参数2, ..., 参数n):  
2.  
3.     函数体
```

是不是样式很简单呢？

取名字的学问

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。取个好名字，就有好兆头呀。所以孔丘先生说“名不正，言不顺”，歪解：名字不正规化，就不顺。这是歪解，希望不要影

响看官正确理解。不知道大师们是不是能够通过外国人名字预测外国人的吉凶祸福呢？

不管怎样，某国人是很在意名字的，旁边有个国家似乎就不在乎。

python也很在乎名字问题，其实，所有高级语言对名字都有要求。为什么呢？因为如果命名乱了，计算机就有点不知所措了。看python对命名的一般要求。

- 文件名:全小写,可使用下划线
- 函数名:小写，可以用下划线风格单词以增加可读性。如：
`myfunction`, `my_example_function`。注意：混合大小写仅被允许用于这种风格已经占据优势的时候，以便保持向后兼容。
- 函数的参数:如果一个函数的参数名称和保留的关键字(所谓保留关键字，就是python语言已经占用的名称，通常被用来做为已有的函数等的命名了，你如果还用，就不行了。)冲突，通常使用一个后缀下划线好于使用缩写或奇怪的拼写。
- 变量:变量名全部小写，由下划线连接各个单词。如`color = WHITE`, `this_is_a_variable = 1`。

其实，关于命名的问题，还有不少争论呢？最典型的是所谓匈牙利命名法、驼峰命名等。如果你喜欢，可以google一下。以下内容供参考：

- [匈牙利命名法](#)
- [驼峰式大小写](#)
- [帕斯卡命名法](#)
- [python命名的官方要求](#)，如果看官的英文可以，一定要阅读。如果英文稍逊，可以来阅读[中文](#)，不用梯子能行吗？看你命了。

[首页](#) | [上一讲](#) | [下一讲](#)

玩转字符串(1)

- 玩转字符串(1)
 - 字符串
 - 变量连接到字符串
 - 对字符串的简单操作
 - 连接字符串
 - Python转义字符

And since they did not see fit to acknowledge God, God gave them up to a debased mind and things that should no 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)

玩转字符串(1)

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

我的分法：一种是语言中的两个元素（比如两个字）和在一起，出来一个新的元素（比如新的字）；另外一种是两个元素和在一起，知识两个元素并列。比如“好”和“人”，两个元素和在一起是“好人”，而3和5和在一起是8，如果你认为是35，那就属于第二类和法了。

把我的这种分法抽象一下：

- 一种是：△ + □ = ○
- 另外一种是：△ + □ = △ □

我们的语言中，离不开以上两类，不是第一类就是第二类。

太天才了。请鼓掌。

字符串

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

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

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

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

操练一下字符串吧。

```
1. >>> print "good good study, day day up"
2. good good study, day day up
3. >>> print "---good---study---day----up"
4. ---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。也就是在python中，通常用一对双引号、或者单引号来包裹一个字符串。或者说，要定义一个字符串，就用双引号或者单引号。

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

```
1. 小明说"我没有烧圆明园"
```

或者这句

```
1. What's your name?
```

问题非常好，有道理。在python中有一种方法专门解决类似的问题。看下面的例子：

```
1. >>> print "小明说：\"我没有少圆明园\""
2. 小明说"我没有少圆明园"
```

这个例子中，为了打印出那句含有双引号的字符串，也就是双引号是字符串的一部分了，使用了一个符号：\，在python中，将这个符号叫做转义符。本来双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。

下面用转义符在打印第二句话：

```
1. >>> print 'what\'s your name?'
2. what's your name?
```

另外，双引号和单引号还可以嵌套，比如下面的句子中，单引号在双引号里面，虽然没有在单引号前面加转义符，但是它被认为是字符串一部分，而不是包裹字符串的符号

```
1. >>> print "what's your name?"    #双引号包裹单引号，单引号是字符
2. what's your name?
3. >>> print 'what "is your" name'  #单引号包裹双引号，双引号是字符
```

```
4. what "is your" name
```

变量连接到字符串

前面讲过变量了，并且有一个钓鱼的比喻。如果忘记了，请看前一章内容。

其实，变量不仅可以跟数字连接，还能够跟字符串连接。

```
1. >>> a=5
2. >>> a
3. 5
4. >>> print a
5. 5
6. >>> b="hello,world"
7. >>> b
8. 'hello,world'
9. >>> print b
10. hello,world
```

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

```
1. >>> type(a)
2. <type 'int'>
3. >>> type(b)
4. <type 'str'>
```

程序员们经常用一种简单的说法，把a称之为数字型变量，意思就是它能够或者已经跟数字连着呢；把b叫做字符（串）型变量，意思就是它能够或者已经跟字符串连着呢。

对字符串的简单操作

对数字，有一些简单操作，比如四则运算就是，如果3+5，就计算出为8。那么对字符串都能进行什么样的操作呢？试试吧：

```
1. >>> "py"+"thon"
2. 'python'
```

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

```
1. >>> "py"-"thon"
2. Traceback (most recent call last):
3.   File "<stdin>", line 1, in <module>
4.   TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

以上就是对字符串的第一种操作。

连接字符串

在IDLE中按照下面方法操作

```
1. >>> a = "老齐"
2. >>> b= "教python"
3. >>> c = a+b
4. >>> print c
5. 老齐教python
6. >>> c
7. '\xe8\x80\x81\xe9\xbd\x90\xe6\x95\x99python'
```

这是一种最简单连接两个字符串的方法。注意上面例子的最后一行，怎么出现乱码了？那不是乱码，是字符编码的问题。这个你权当没看见好了。不过的确是看见了。请看官google字符编码就知道了。这里推荐一篇非常好的文章：[字符集和字符编码](#)

提示：看官做为学习者，一定要对所学的对象有一种好奇心，比如上面例子中，如果你满足于 `print c`，发现结果跟自己所预料一样，这还远远不够。如果你向下走了一行，就发现一个怪怪的结果了，这就让你在编程路上又前进一大步。所以，要有对世界好奇的心，不断探索、思考和尝试。反正在计算机上尝试，也没有多大成本。最坏的结果是关掉IDLE罢了。

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：

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

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

可以用下面三种方法中的任何一种：

```
1. >>> print b + `a`          #注意，` `是反引号，不是单引号，就是键盘中通常在数字1左边的那个，在英文半角状态下输入的符号
2. free1989
3. >>> print b + str(a)      #str(a)实现将整数对象转换为字符串对象
4. free1989
5. >>> print b + repr(a)     #repr(a)与上面的类似
6. free1989
```

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

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

Almost always use `str` when creating output for end users.

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

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

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

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

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

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

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

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

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

以上英文内容来

源: <http://stackoverflow.com/questions/19331404/str-vs-repr-functions-in-python-2-7-5>

Python转义字符

在字符串中，有时需要输入一些特殊的符号，但是，某些符号不能直接

输出，就需要用转义符。所谓转义，就是不采用符号现在之前的含义，而采用另外一含义了。下面表格中列出常用的转义符：

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

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

```

1. >>> print "hello.I am qiwsir.\                #这里换行，下一行接续
2. ... My website is 'http://qiwsir.github.io'."
3. hello.I am qiwsir.My website is 'http://qiwsir.github.io'.
4.
5. >>> print "you can connect me by qq\\weibo\\gmail"  #\\是为了要后面那
   个\
6. you can connect me by qq\weibo\gmail

```

看官自己试试吧。如果有问题，可以联系我解答。

[首页](#) | [上一讲](#) | [下一讲](#)

玩转字符串(2)

- [玩转字符串\(2\)](#)
 - [连接字符串的方法2](#)
- [字符串复制](#)
- [字符串长度](#)
- [字符大小写的转换](#)

玩转字符串(2)

上一章中已经讲到连接两个字符串的一种方法。复习一下：

```
1. >>> a= 'py'
2. >>> b= 'thon'
3. >>> a+b
4. 'python'
```

既然这是一种方法，言外之意，还有另外一种方法。

连接字符串的方法2

在说方法2之前，先说明一下什么是占位符，此前在讲解变量（参数）的时候，提到了占位符，这里对占位符做一个比较严格的定义：

来自[百度百科](#)的定义：

顾名思义，占位符就是先占住一个固定的位置，等着你再往里面添加内容的符号。

根据这个定义，在python里面规定了一些占位符，通过这些占位符来说明那个位置应该填写什么类型的东西，这里暂且了解两个占位符：%d——表示那个位置是整数，%s——表示那个位置应该是字符串。下面看一个具体实例：

```
1. >>> print "one is %d"%1
2. one is 1
```

要求打印(print)的内容中,有一个%d占位符,就是说那个位置应该放一个整数。在第二个%后面,跟着的就是那个位置应该放的东西。这里是一个整数1。我们做下面的操作,就可以更清楚了解了:

```
1. >>> a=1
2. >>> type(a)
3. <type 'int'>      #a是整数
4. >>> b="1"
5. >>> type(b)
6. <type 'str'>      #b是字符串
7. >>> print "one is %d"%a
8. one is 1
9. >>> print "one is %d"%b      #报错了,这个占位符的位置应该放整数,不应该放字符串。
10. Traceback (most recent call last):
11.   File "<stdin>", line 1, in <module>
12.   TypeError: %d format: a number is required, not str
```

同样道理,%s对应的位置应该放字符串,但是,如果放了整数,也可以。只不过是已经转为字符串对待了。但是不赞成这么做。在将来,如果使用mysql(一种数据库)的时候,会要求都用%s做为占位符,这是后话,听听有这么回事即可。

```
1. >>> print "one is %s"%b
2. one is 1
3. >>> print "one is %s"%a      #字符串是包容的
4. one is 1
```

好了。啰嗦半天,占位符是不是理解了呢?下面我们就用占位符来连接字符串。是不是很有意思?

```

1. >>> a = "py"
2. >>> b = "thon"
3. >>> print "%s%s"%(a,b) #注
4. python

```

注：仔细观察，如果两个占位符，要向这两个位置放东西，代表的东西要写在一个圆括号内，并且中间用逗号（半角）隔开。

字符串复制

有一个变量，连接某个字符串，也想让另外一个变量，也连接这个字符串。一种方法是把字符串再写一边，这种方法有点笨拙，对于短的都无所谓了。但是长的就麻烦了。这里有一种字符串复制的方法：

```

1. >>> a = "My name is LaoQi. I like python and can teach you to learn it."
2. >>> print a
3. My name is LaoQi. I like python and can teach you to learn it.
4. >>> b = a
5. >>> print b
6. My name is LaoQi. I like python and can teach you to learn it.
7. >>> print a
8. My name is LaoQi. I like python and can teach you to learn it.

```

复制非常简单，类似与赋值一样。可以理解为那个字符串本来跟a连接着，通过b=a，a从自己手里分处一股绳子给了b，这样两者都可以指向那个字符串了。

字符串长度

要向知道一个字符串有多少个字符，一种方法是从头开始，盯着屏幕数一数。哦，这不是计算机在干活，是键客在干活。键客，不是剑客。剑客是以剑为武器的侠客；而键客是以键盘为武器的侠客。当然，还有贱

客，那是贱人的最高境界，贱到大侠的程度，比如岳不群之流。

键客这样来数字符串长度：

```
1. >>> a="hello"
2. >>> len(a)
3. 5
```

使用的是一个函数`len(object)`。得到的结果就是该字符串长度。

```
1. >>> m = len(a) #把结果返回后赋值给一个变量
2. >>> m
3. 5
4. >>> type(m) #这个返回值（变量）是一个整数型
5. <type 'int'>
```

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中有下面一堆内建函数，用来实现各种类型的大小写转化

- `S.upper()` #S中的字母大写
- `S.lower()` #S中的字母小写
- `S.capitalize()` #首字母大写
- `S.istitle()` #单词首字母是否大写的，且其它为小写，注网友白羽毛指出，这里表述不准确。非常感谢他。为了让看官对这些大小写问题有更深刻理解，我从新写下面的例子，请看官审查。再次感谢白羽毛。
- `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

看例子：

```

1. >>> a = "qiwsir,python"
2. >>> a.upper()          #将小写字母完全变成大写字母
3. 'QIWSIR,PYTHON'
4. >>> a                  #原数据对象并没有改变
5. 'qiwsir,python'
6. >>> b = a.upper()
7. >>> b
8. 'QIWSIR,PYTHON'
9. >>> c = b.lower()      #将所有的小写字母变成大写字母
10. >>> c
11. 'qiwsir,python'
12.
13. >>> a
14. 'qiwsir,python'
15. >>> a.capitalize()    #把字符串的第一个字母变成大写
16. 'Qiwsir,python'
17. >>> a                  #原数据对象没有改变
18. 'qiwsir,python'
19. >>> b = a.capitalize() #新建立了一个
20. >>> b
21. 'Qiwsir,python'
22.
23. >>> a = "qiwsir,github" #这里的问题就是网友白羽毛指出的，非常感谢他。
24. >>> a.istitle()
25. False
26. >>> a = "QIWSIR"        #当全是大写的时候，返回False
27. >>> a.istitle()
28. False
29. >>> a = "qIWSIR"
30. >>> a.istitle()
31. False
32. >>> a = "Qiwsir,github" #如果这样，也返回False
33. >>> a.istitle()
34. False

```

```

35. >>> a = "Qiwsir"           #这样是True
36. >>> a.istitle()
37. True
38. >>> a = 'Qiwsir,Github'   #这样也是True
39. >>> a.istitle()
40. True
41.
42. >>> a = "Qiwsir"
43. >>> a.isupper()
44. False
45. >>> a.upper().isupper()
46. True
47. >>> a.islower()
48. False
49. >>> a.lower().islower()
50. True

```

顺着白羽毛网友指出的，再探究一下，可以这么做：

```

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

```

字符串问题，看来本讲还不能结束。下一讲继续。有看官可能要问了，上面这些在实战中怎么用？我正想为你的，请键客设计一种实战情景，能不能用上所学。

[首页](#) | [上一讲](#) | [下一讲](#)

玩转字符串(3)

- 玩转字符串(3)
 - 给字符串编号
 - 字符串截取
 - 去掉字符串两头的空格
- 练习

玩转字符串(3)

字符串是一个很长的话题，纵然现在开始第三部分，但是也不能完全说尽。因为字符串是自然语言中最复杂的东西，也是承载功能最多的，计算机高级语言编程，要解决自然语言中的问题，让自然语言中完成的事情在计算机上完成，所以，也不得不有更多的话题。

字符串就是一个话题中心。

给字符串编号

在很多很多情况下，我们都要对字符串中的每个字符进行操作（具体看后面的内容），要准确进行操作，必须做的一个工作就是把字符进行编号。比如一个班里面有50名学生，如果这些学生都有学号，老师操作他们将简化很多。比如不用专门找每个人名字，直接通过学号知道谁有没有交作业。

在python中按照这样的顺序对字符串进行编号：从左边第一个开始是0号，向下依次按照整数增加，为1、2...，直到最后一个，在这个过程中，所有字符，包括空格，都进行变好。例如：

```
Hello,world
```

对于这个字符串，从左向右的变好依次是：

```
|0|1|2|3|4|5|6|7|8|9|10|11|
```

```
|H|e|l|l|o|,|w|o|r| |l| |d| |
```

在班级了，老师只要喊出学生的学号，自动有对应的学生站起来。在python里面如何把某个编号所对应的字符调出来呢？看代码：

```
1. >>> a = "Hello, wor ld"
2. >>> len(a)          #字符串的长度是12,说明公有12个字符, 最后一个字符编号是11
3. 12
4. >>> a[0]
5. 'H'
6. >>> a[3]
7. 'l'
8. >>> a[9]
9. ' '
10. >>> a[11]
11. 'd'
12. >>> a[5]
13. ','
```

特别说明，编号是从左边开始，第一个是0。

能不能从右边开始编号呢？可以。这么人见人爱的python难道这点小要求都不满足吗？

```
1. >>> a[-1]
2. 'd'
3. >>> a[11]
4. 'd'
5. >>> a[-12]
6. 'H'
7. >>> a[-3]
8. ' '
```

看到了吗？如果从右边开始，第一个编号是-1, 这样就跟从左边区分开了。也就是a[-1]和a[11]是指向同一个字符。

不管从左边开始还是从右边开始，都能准确找到某个字符。看官喜欢从哪边开始就从哪边开始，或者根据实际使用情况，需要从哪边开始就从哪边开始。

字符串截取

有了编号，不仅仅能够找出某个字符，还能在字符串中取出一部分来。比如，从“hello, wor ld”里面取出“llo”。可以这样操作

```
1. >>> a[2:5]
2. 'llo'
```

这就是截取字符串的一部分，注意：所截取部分的第一个字符（l）对应的编号是（2），从这里开始；结束的字符是（o），对应编号是（4），但是结束的编号要增加1, 不能是4, 而是5. 这样截取到的就是上面所要求的了。

试一试，怎么截取到“, wor”

也就是说，截取a[n,m]，其中n<m，得到的字符是从a[n]开始到a[m-1]

有几个比较特殊的

```
1. >>> a[:]      #表示截取全部
2. 'Hello, wor ld'
3. >>> a[3:]     #表示从a[3]开始，一直到字符串的最后
4. 'lo, wor ld'
5. >>> a[:4]     #表示从字符串开头一直到a[4]前结束
6. 'Hell'
```

去掉字符串两头的空格

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

好吧，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

看官在看下面示例之前，请先自己用上面的内置函数，是否可以？

```
1. >>> b=" hello "  
2. >>> b  
3. ' hello '  
4. >>> b.strip()  
5. 'hello'  
6. >>> b  
7. ' hello '  
8. >>> b.lstrip()  
9. 'hello '  
10. >>> b.rstrip()  
11. ' hello'
```

练习

学编程，必须做练习，通过练习熟悉各种情况下的使用。

下面共同做一个练习：输入用户名，计算机自动向这个用户打招呼。代码如下：

```
1. #coding:utf-8
2.
3. print "please write your name:"
4. name=raw_input()
5. print "Hello,%s"%name
```

这段代码中的raw_input()的含义，就是要用户输入内容，所输入的内容是一个字符串。

其实，上面这段代码存在这改进的地方，比如，如果用户输入的是小写，是不是要将名字的首字母变成大写呢？如果有空格，是不是要去掉呢？等等。或许还有别的，看看能不能在这个练习中，将以前学习过的东西综合应用一下？

[首页](#) | [上一讲](#) | [下一讲](#)

眼花缭乱的运算符

- 眼花缭乱的运算符
 - 算术运算符
 - 比较运算符
 - 逻辑运算符
 - 布尔类型的变量
 - 布尔运算

眼花缭乱的运算符

在计算机高级语言中，运算符是比较多样化的。其实，也都源于我们日常的需要。

算术运算符

前面已经讲过了四则运算，其中涉及到一些运算符：加减乘除，对应的符号分别是： $+$ $-$ $*$ $/$ ，此外，还有求余数的： $%$ 。这些都是算术运算符。其实，算术运算符不止这些。根据中学数学的知识，看官也应该想到，还应该有多项式、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
+	加 - 两个对象相加	10+20 输出结果 30
-	减 - 得到负数或是一个数减去另一个数	10-20 输出结果 -10
*	乘 - 两个数相乘或是返回一个被重复若干次的字符串	10 * 20 输出结果 200
/	除 - x除以y	20/10 输出结果 2

%	取余 - 返回除法的余数	20%10 输出结果 0
**	幂 - 返回x的y次幂	10**2 输出结果 100
//	取整除 - 返回商的整数部分	9//2 输出结果 4 , 9.0//2.0 输出结果 4.0

是不是看着并不陌生呀。这里有一个建议给看官，请打开你的IDLE，依次将上面的运算符实验一下。

列为看官可以根据中学数学的知识，想想上面的运算符在混合运算中，应该按照什么顺序计算。并且亲自试试，是否与中学数学中的规律一致。（应该是一致的，计算机科学家不会另外搞一套让我们和他们一块受罪。）

比较运算符

所谓比较，就是比一比两个东西。这在某国是最常见的了，做家长的经常把自己的孩子跟别人的孩子比较，唯恐自己孩子在某方面差了；官员经常把自己的工资和银行比较，总觉得少了。

在计算机高级语言编程中，任何两个同一类型的量的都可以比较，比如两个数字可以比较，两个字符串可以比较。注意，是两个同一类型的。不同类型的量可以比较吗？首先这种比较没有意义。就好比二两肉和三尺布进行比较，它们谁大呢？这种比较无意义。所以，在真正的编程中，我们要谨慎对待这种不同类型量的比较。

但是，在某些语言中，允许这种无意思的比较。因为它在比较的时候，都是将非数值的转化为了数值类型比较。这个后面我们会做个实验。

对于比较运算符，在小学数学中就学习了一些：大于、小于、等于、不等于。没有陌生的东西，python里面也是如此。且看下表：

以下假设变量a为10，变量b为20：

运算符	描述	实例
-----	----	----

==	等于 - 比较对象是否相等	(a == b) 返回 False。
!=	不等于 - 比较两个对象是否不相等	(a != b) 返回 true。
>	大于 - 返回x是否大于y	(a > b) 返回 False。
<	小于 - 返回x是否小于y	(a < b) 返回 true。
>=	大于等于 - 返回x是否大于等于y。	(a >= b) 返回 False。
<=	小于等于 - 返回x是否小于等于y。	(a <= b) 返回 true。

上面的表格实例中，显示比较的结果就是返回一个true或者false，这是什么意思呢。就是在告诉你，这个比较如果成立，就是为真，返回True，否则返回False，说明比较不成立。

请按照下面方式进行比较操作，然后再根据自己的想象，把比较操作熟练熟练。

```

1. >>> a=10
2. >>> b=20
3. >>> a>b
4. False
5. >>> a<b
6. True
7. >>> a==b
8. False
9. >>> a!=b
10. True
11. >>> a>=b
12. False
13. >>> a<=b
14. True
15. >>> c="5"    #a、c是两个不同类型的量，能比较，但是不提倡这么做。
16. >>> a>c
17. False
18. >>> a<c
19. True

```

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》](#)

下面简单理解一下逻辑问题。

布尔类型的变量

在所有的高级语言中，都有这么一类变量，被称之为布尔型。从这个名称，看官就知道了，这是用一个人的名字来命名的。

乔治·布尔 (*George Boole*, 1815年11月—1864年)，英格兰数学家、哲学家。

乔治·布尔是一个皮匠的儿子，生于英格兰的林肯。由于家境贫寒，布尔不得不在协助养家的同时为自己能受教育而奋斗，不管怎么说，他成了19世纪最重要的数学家之一。尽管他考虑过以牧师为业，但最终还是决定从教，而且不久就开办了自己的学校。

在备课的时候，布尔不满意当时的数学课本，便决定阅读伟大数学家的论文。在阅读伟大的法国数学家拉格朗日的论文时，布尔有了变分法方面的新发现。变分法是数学分析的分支，它处理的是寻求优化某些参数的曲线和曲面。

1848年，布尔出版了《*The Mathematical Analysis of Logic*》，这是他对符号逻辑诸多贡献中的第一次。

1849年，他被任命位于爱尔兰科克的皇后学院（今科克大学或UCC）的数学教授。1854年，他出版了《*The Laws of Thought*》，这是他最著名的著作。在这本书中布尔介绍了现在以他的名字命名的布尔代数。布尔撰写了微分方程和差分方程的课本，这些课本在英国一直使用到19世纪末。

由于其在符号逻辑运算中的特殊贡献，很多计算机语言中将逻辑运算称为布尔运算，将其结果称为布尔值。

请看官认真阅读布尔的生平，立志呀。

布尔所创立的这套逻辑被称之为“布尔代数”。其中规定只有两种值，True和False，正好对应这计算机上二进制数的1和0。所以，布尔代数和计算机是天然吻合的。

所谓布尔类型，就是返回结果为1(True)、0(False)的数据变量。

在python中（其它高级语言也类似，其实就是布尔代数的运算法则），有三种运算符，可以实现布尔类型的变量间的运算。

布尔运算

看下面的表格，对这种逻辑运算符比较容易理解：

（假设变量a为10，变量b为20）

运算符	描述	实例
and	布尔“与” - 如果x为False，x and y返回False，否则它返回y的计算值。	(a and b) 返回 true。
or	布尔“或” - 如果x是True，它返回True，否则它返回y的计算值。	(a or b) 返回 true。
not	布尔“非” - 如果x为True，返回False。如果x为False，它返回True。	not(a and b) 返回 false。

- and

and，翻译为“与”运算，但事实上，这种翻译容易引起望文生义的理解。先说一下正确的理解。A and B，含义是：首先运算A，如果A的值是true，就计算B，并将B的结果返回做为最终结果，如果B是False，那么A and B的最终结果就是False，如果B的结果是True，那么A and B的结果就是True；如果A的值是False，就不计算B了，直接返回A and B的结果为False。

比如：

`4>3 and 4<9`，首先看 `4>3` 的值，这个值是 `True`，再看 `4<9` 的值，是 `True`，那么最终这个表达式的结果为 `True`。

```
1. >>> 4>3 and 4<9
2. 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`。

```
1. >>> 4>3 and 4<2
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。

```
1. >>> 4<3 and 4<2
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。

- or

or，翻译为“或”运算。在A and B中，它是这么运算的：

```
1. if A==True:
2.     return True
3. else:
4.     if B==True:
5.         return True
```

```

6.     else if B==False:
7.         return False

```

上面这段算是伪代码啦。所谓伪代码，就是不是真正的代码，无法运行。但是，伪代码也有用途，就是能够以类似代码的方式表达一种计算过程。

看官是不是能够看懂上面的伪代码呢？下面再增加上每行的注释。这个伪代码跟自然的英语差不多呀。

```

1.  if A==True:           #如果A的值是True
2.      return True       #返回True, 表达式最终结果是True
3.  else:                 #否则, 也就是A的值不是True
4.      if B==True:       #看B的值, 然后就返回B的值做为最终结果。
5.          return True
6.      else if B==False:
7.          return False

```

举例，根据上面的运算过程，分析一下下面的例子，是不是与运算结果一致？

```

1.  >>> 4<3 or 4<9
2.  True
3.  >>> 4<3 or 4>9
4.  False
5.  >>> 4>3 or 4>9
6.  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。

```

1.  >>> not(4>3)
2.  False
3.  >>> not(4<3)

```

4. True

关于运算符问题，其实不止上面这些，还有呢，比如成员运算符in，在后面的学习中会逐渐遇到。

[首页](#) | [上一讲](#) | [下一讲](#)

从if开始语句的征程

- [从if开始语句的征程](#)
 - [什么是语句](#)
 - [if语句](#)
 - [小知识](#)

从if开始语句的征程

一般编程的教材，都是要把所有的变量类型讲完，然后才讲语句。这种讲法，其实不符合学习的特点。学习，就是要循序渐进的。在这点上，我可以很吹一通了，因为我做过教师，研究教育教学，算是有一点心得的。所以，我在这里就开始讲授语句。

什么是语句

在前面，我们已经写了一些.py的文件，这些文件可以用python来运行。那些文件，就是由语句组成的程序。

为了能够严谨地阐述这个概念，我还是要抄一段[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字

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。

if语句

谈到语句，不要被吓住。看下面的例子先：

```
1. if a==4:
2.     print "it is four"
3. else:
4.     print "it is no four"
```

逐句解释一番，注意看注释。在这里给列为看官提醒，在写程序的是由，一定要写必要的注释，同时在阅读程序的时候，也要注意看注释。

```
1. if a==4:                #如果变量a==4是真的，a==4为True，就
2.     print "it is four"  #打印"it is four"。
3. else:                   #否则，即a==4是假的，a==4为False，就
4.     print "it is not four" #打印"it is not four"。
```

以上几句话，就完成了条件判断，在不同条件下做不同的事情。因此，if语句，常被翻译成“条件语句”。

条件语句的基本样式结构：

```
1. if 条件1:
2.     执行的内容1
```

```

3. elif 条件2:
4.     执行的内容2
5. elif 条件3:
6.     执行的内容3
7. else:
8.     执行的内容4

```

执行的内容1、内容2，等，称之为语句块。elif用于多个条件时使用，可以没有。另外，也可以只有if，而没有else。

提醒：每个执行的内容，均以缩进四个空格方式。

例1：输入一个数字，并输出输入的结果，如果这个数字大于10，那么同时输出大于10,如果小于10,同时输出提示小于10,如果等于10,就输出表扬的一句话。

从这里开始，我们的代码就要越来越接近于一个复杂的判断过程了。为了让我们的思维能够更明确上述问题的解决流程，在程序开发过程中，常常要画流程图。什么是流程图，我从另外一个角度讲，就是要让思维过程可视化，简称“思维可视化”。顺便自吹自擂一下，我从2004年就开始在我朝推广思维导图，这就是一种思维可视化工具。自吹到此结束。看这个问题的流程图：



理解了流程图中的含义，就开始写代码，代码实例如下：

```

1. #! /usr/bin/env python
2. #coding:utf-8
3.
4. print "请输入任意一个整数数字："
5.
6. number = int(raw_input())    #通过raw_input()输入的数字是字符串
7.                             #用int()将该字符串转化为整数
8.

```



```

9.  if number == 10:
10.     print "您输入的数字是:%d"%number
11.     print "You are SMART."
12.  elif number > 10:
13.     print "您输入的数字是:%d"%number
14.     print "This number is more than 10."
15.  elif number < 10:
16.     print "您输入的数字是:%d"%number
17.     print "This number is less than 10."
18.  else:
19.     print "Are you a human?"

```

特别提醒看官注意，前面我们已经用过raw_input()函数了，这个是获得用户在界面上输入的信息，而通过它得到的是字符串类型的数据。可以在IDLE中这样检验一下：

```

1.  >>> a=raw_input()
2.  10
3.  >>> a
4.  '10'
5.  >>> type(a)
6.  <type 'str'>
7.  >>> a=int(a)
8.  >>> a
9.  10
10. >>> type(a)
11. <type 'int'>

```

刚刚得到的那个a就是str类型，如果用int()转换一下，就变成int类型了。

看来int()可以将str类型的数字转换为int类型，类似，是不是有这样的结论呢：str()可以将int类型的数字转化为str类型。建议看官实验一下。

上述程序的后面，就是依据条件进行判断，不同条件下做不同的事情

了。需要提醒的是在条件中：`number == 10`，为了阅读方便，在`number`和`==`之间有一个空格最好了，同理，后面也有一个。这里的`10`，是`int`类型，`number`也是`int`类型。

上面的程序不知道是不是搞懂了？如果没有，可以通过QQ跟我联系，我的QQ公布一下：26066913，或者登录我的微博，通过微博跟我联系，当然还可以发邮件啦。我看到您的问题，会答复的。在github上跟我互动，是我最欢迎的。

最后，给看官留一个练习题目：

课后练习：开发一个猜数字游戏的程序。即程序在某个范围内指定一个数字，比如在0到9范围内指定一个数字，用户猜测程序所指定的数字大小。

请看官自己编写。我们会在后面讨论这个问题。

小知识

不知道各位是否注意到，上面的那段代码，开始有一行：

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

这是什么意思呢？

这句话以`#`开头，表示本来不在程序中运行。这句话的用途是告诉机器寻找到该设备上的python解释器，操作系统使用它找到的解释器来运行文件中的程序代码。有的程序里写的是`/usr/bin python`，表示python解释器在`/usr/bin`里面。但是，如果写成`/usr/bin/env`，则表示要通过系统搜索路径寻找python解释器。不同系统，可能解释器的位置不同，所以这种方式能够让代码更将拥有可移植性。对了，以上是对Unix系列操作系统而言。对与windows系统，这句话就当不存

在。

[首页](#) | [上一讲](#) | [下一讲](#)

一个免费的实验室

- 一个免费的实验室
 - 走进Python实验室
 - 交互模式下进行实验
 - 通过变量直接显示其内容
 - 缩进
 - 报错
 - 探索

一个免费的实验室

在学生时代，就羡慕实验室，老师在里面可以鼓捣各种有意思的东西。上大学的时候，终于有机会在实验室做大量实验了，因为我是物理系，并且，遇到了一位非常令我尊敬的老师——高老师，让我在他的实验室里面，把所有已经破旧损坏的实验仪器修理装配好，并且按照要求做好实验样例。经过一番折腾，才明白，要做好实验，不仅仅花费精力，还有不菲的设备成本呢。后来工作的时候，更感觉到实验设备费用之高昂，因此做实验的时候总要小心翼翼。

再后来，终于发现原来计算机是一个最好的实验室。在这里做实验成本真的很低呀。

扯的远了吧。不远，现在就扯回来。学习Python，也要做实验，也就是尝试性地看看某个命令到底什么含义。通过实验，研究清楚了，才能在编程实践中使用。

怎么做Python实验呢？

走进Python实验室

在《集成开发环境(IDE)》一章中，我们介绍了Python的IDE时，给大家推荐了IDLE，进入到IDLE中，看到>>>符号，可以在后面输入一行指令。其实，这就是一个非常好的实验室。

另外一个实验室就是UNIX操作系统（包含各种Linux和Mac OSx）的shell，在打开shell之后，输入python，出现如下图所示：



如果看官是用windows的，也能够通过cmd来获得上图类似的界面，依然是输入python，之后得到界面。

在上述任何一个环境中，都可以输入指令，敲回车键运行并输出结果。

在这里你可以随心所欲实验。

交互模式下进行实验

前面的各讲中，其实都使用了交互模式。本着循序渐进、循环上升的原则，本讲应该对交互模式进行一番深入和系统化了。

通过变量直接显示其内容

从例子开始：

```
1. >>> a="http://qiwsir.github.io"
2. >>> a
3. 'http://qiwsir.github.io'
4. >>> print a
5. http://qiwsir.github.io
```

当给一个变量a赋值于一个字符串之后，输入变量名称，就能够打印出字符串，和print a具有同样的效果。这是交互模式下的一个特点，如

果在文件模式中，则不能，只有通过print才能打印变量内容。

缩进

```
1. >>> if bool(a):
2.     ...     print "I like python"
3.     ...
4. I like python
```

对于if语句，在上一讲《[从if开始语句的征程](#)》中，已经注意到，if下面的执行语句要缩进四个空格。在有的python教材中，说在交互模式下不需要缩进，可能是针对python3或者其它版本，我使用的是python2.7，的确需要缩进。上面的例子就看出来了。

看官在自己的机器上测试一下，是不是需要缩进？

报错

在一个广告中看到过这样一句话：程序员的格言，“不求最好，只求报错”。报错，对编程不是坏事。如何对待报错呢？

一定要认真阅读所提示的错误信息。

还是上面那个例子，我如果这样写：

```
1. >>> if bool(a):
2.     ... print "I like python"
3.     File "<stdin>", line 2
4.         print "I like python"
5.             ^
6. IndentationError: expected an indented block
```

从错误信息中，我们可以知道，第二行错了。错在什么地方呢？

python非常人性化就在这里，告诉你错误在什么地方：

IndentationError: expected an indented block

意思就是说需要一个缩进块。也就是我没有对第二行进行缩进，需要缩进。

另外，顺便还要提醒，>>>表示后面可以输入指令，...表示当前指令没有结束。要结束并执行，需要敲击两次回车键。

探索

如果看官对某个指令不了解，或者想试试某种操作是否可行，可以在交互模式下进行探索，这种探索的损失成本非常小，充其量就是报错。而且从报错信息中，我们还能得到更多有价值的内容。

例如，在《[眼花缭乱的运算符](#)》中，提到了布尔运算，其实，在变量的类型中，除了前面提到的整数型、字符串型，布尔型也是一种，那么布尔型的变量有什么特点呢？下面就探索一下：

```
1. >>> a
2. 'http://qiwsir.github.io'
3. >>> bool(a)      #布尔型,用bool()表示,就类似int(),str(),是一个内置函数
4. True
5. >>> b=""
6. >>> bool(b)
7. False
8. >>> bool(4>3)
9. True
10. >>> bool(4<3)
11. False
12. >>> m=bool(b)
13. >>> m
14. False
15. >>> type(m)
16. <type 'bool'>
17. >>>
```

从上面的实验可以看出，如果对象是空，返回False，如果不是，则返回True；如果对象是False，返回False。上面探索，还可以扩展到其它情况。看官能不能通过探索，总结出bool()的特点呢？

有容乃大的list(1)

- 有容乃大的list(1)
 - 定义
 - list索引
 - 对list反转
 - 对list的操作
 - 追加元素

有容乃大的list(1)

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

前面还学习了变量，如果某个变量跟一个int类型的数据用线连着（行话是：赋值），那么这个变量我们就把它叫做int类型的变量；有时候还没赋值呢，是准备让这个变量接收int类型的数据，我们也需要将它声明为int类型的变量。不过，在python里面有一样好处，变量不用提前声明，随用随命名。

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

LIST在python中具有非常强大的功能。

定义

在python中，用方括号表示一个list，[]

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

```

1. >>> a=[]           #定义了一个变量a，它是list类型，并且是空的。
2. >>> type(a)
3. <type 'list'>       #用内置函数type()查看变量a的类型，为list
4. >>> bool(a)         #用内置函数bool()看看list类型的变量a的布尔值，因为是空的，
                        所以为False
5. False
6. >>> print a         #打印list类型的变量a
7. []

```

不能总玩空的，来点实的吧。

```

1. >>> a=['2',3,'qiwsir.github.io']
2. >>> a
3. ['2', 3, 'qiwsir.github.io']
4. >>> type(a)
5. <type 'list'>
6. >>> bool(a)
7. True
8. >>> print a
9. ['2', 3, 'qiwsir.github.io']

```

用上述方法，定义一个list类型的变量和数据。

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

list索引

尚记得在《玩转字符串(3)》中，曾经给字符串进行编号，然后根据编号来获取某个或者某部分字符，这样的过程，就是“索引”(index)。

```
1. >>> url = "qiwsir.github.io"
2. >>> url[2]
3. 'w'
4. >>> url[:4]
5. 'qiws'
6. >>> url[3:9]
7. '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。

```
1. >>> a
2. ['2', 3, 'qiwsir.github.io']
3. >>> a[0]      #索引序号也是从0开始
4. '2'
5. >>> a[1]
6. 3
7. >>> [2]
8. [2]
9. >>> a[:2]     #跟str中的类似，切片的范围是：包含开始位置，到结束位置之前
10. ['2', 3]     #不包含结束位置
11. >>> a[1:]
12. [3, 'qiwsir.github.io']
13. >>> a[-1]    #负数编号从右边开始
14. 'qiwsir.github.io'
15. >>> a[-2]
16. 3
17. >>> a[:]
18. ['2', 3, 'qiwsir.github.io']
```

对list反转

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

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

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，是比较容易理解和阅读的，特别推荐之：

```
1. >>> list(reversed(alst))
2. [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：

```
1. >>> help(reversed)
2. Help on class reversed in module __builtin__:
3.
4. class reversed(object)
5. |   reversed(sequence) -> reverse iterator over values of the
   |   sequence
6. |
7. |   Return a reverse iterator
```

它返回一个可以迭代的对象，不过是已经将原来的序列对象反转了。比如：

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

很好，很强大，特别推荐使用。

对list的操作

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

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

前面在对str进行操作的时候，有一些内置函数，比如s.strip()，这是去掉左右空格的内置函数，也是str的方法。按照一贯制的对称法则，对list也会有一些操作方法。

追加元素

```

1. >>> a = ["good","python","I"]
2. >>> a
3. ['good', 'python', 'I']
4. >>> a.append("like")           #向list中添加str类型"like"
5. >>> a
6. ['good', 'python', 'I', 'like']
7. >>> a.append(100)              #向list中添加int类型100
8. >>> a
9. ['good', 'python', 'I', 'like', 100]
```

官方文档这样描述list.append()方法

```
list.append(x)
```

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

从以上描述中，以及本部分的标题“追加元素”，是不是能够理解

`list.append(x)`的含义呢？即将新的元素`x`追加到`list`的尾部。

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

```

1. >>> a
2. ['good', 'python', 'I', 'like', 100]
3. >>> a[len(a):]=[3]          #len(a),即得到list的长度，这个长度是指list中的元
    素个数。
4. >>> a
5. ['good', 'python', 'I', 'like', 100, 3]
6. >>> len(a)
7. 6
8. >>> a[6:]=['xxoo']
9. >>> a
10. ['good', 'python', 'I', 'like', 100, 3, 'xxoo']

```

顺便说一下`len()`，这个是用来获取`list`, `str`等类型的数据长度的。在字符串讲解的时候也提到了。

```

1. >>> name = 'yeashape'
2. >>> len(name)          #str的长度，是字符的个数
3. 8
4. >>> a=[1,2,'a','b']    #list的长度，是元素的个数
5. >>> len(a)
6. 4
7. >>> b=['yeashape']
8. >>> len(b)
9. 1

```

下一讲继续`list`，有容乃大。

[首页](#) | [上一讲](#) | [下一讲](#)

有容乃大的list(2)

- 有容乃大的list(2)
 - 对list的操作
 - list的长度
 - 合并list
 - list中某元素的个数
 - 元素在list中的位置

有容乃大的list(2)

对list的操作

list的长度

还记得str的长度怎么获得吗？其长度是什么含呢？那种方法能不能用在list上面呢？效果如何？

做实验：

```
1. >>> name = 'qiwsir'
2. >>> type(name)
3. <type 'str'>
4. >>> len(name)
5. 6
6. >>> lname = ['sir', 'qi']
7. >>> type(lname)
8. <type 'list'>
9. >>> len(lname)
10. 2
11. >>> length = len(lname)
12. >>> length
13. 2
```



```
14. >>> type(length)
15. <type 'int'>
```

实验结论：

- `len(x)`，对于list一样适用
- 得到的是list中元素个数
- 返回值是int类型

合并list

《有容乃大的list(1)》中，对list的操作提到了`list.append(x)`，也就是将某个元素x 追加到已知的一个list后边。

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

```
list.extend(L)
```

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

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

官方文档的这句话翻译过来：

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

英语太烂，翻译太差。直接看例子，更明白

```
1. >>> la
2. [1, 2, 3]
3. >>> lb
```

```

4. ['qiwsir', 'python']
5. >>> la.extend(lb)
6. >>> la
7. [1, 2, 3, 'qiwsir', 'python']
8. >>> lb
9. ['qiwsir', 'python']

```

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

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

```

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

```

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

如果extend的对象是数值型，则报错。

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

不过，别忘记了前面官方文档的后半句话，它的意思是：

```

1. >>> la

```

```

2. [1, 2, 3, 'a', 'b', 'c']
3. >>> lb
4. ['qiwsir', 'python']
5. >>> la[len(la):]=lb
6. >>> la
7. [1, 2, 3, 'a', 'b', 'c', 'qiwsir', 'python']

```

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

联想到到[上一讲](#)中的一个list函数`list.append()`, 这里的`extend`函数也是将另外的元素（只不过这个元素是列表）增加到一个已知列表中, 那么两者有什么不一样呢? 看下面例子:

```

1. >>> lst = [1,2,3]
2. >>> lst.append(['qiwsir', 'github'])
3. >>> lst
4. [1, 2, 3, ['qiwsir', 'github']] #append的结果
5. >>> len(lst)
6. 4
7.
8. >>> lst2 = [1,2,3]
9. >>> lst2.extend(['qiwsir', 'github'])
10. >>> lst2
11. [1, 2, 3, 'qiwsir', 'github'] #extend的结果
12. >>> len(lst2)
13. 5

```

`append`是整建制地追加, `extend`是个体化扩编。

list中某元素的个数

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

```
list.count(x)
```

Return the number of times x appears in the list.

一定要不断实验，才能理解文档中精炼的表达。

```
1. >>> la = [1,2,1,1,3]
2. >>> la.count(1)
3. 3
4. >>> la.append('a')
5. >>> la.append('a')
6. >>> la
7. [1, 2, 1, 1, 3, 'a', 'a']
8. >>> la.count('a')
9. 2
10. >>> la.count(2)
11. 1
12. >>> la.count(5)      #NOTE:la中没有5,但是如果用这种方法找,不报错,返回的是
    数字0
13. 0
```

元素在list中的位置

《有容乃大的list(1)》中已经提到，可以将list中的元素，从左向右依次从0开始编号，建立索引（如果从右向左，就从-1开始依次编号），通过索引能够提取出某个元素，或者某几个元素。就是如这样做：

```
1. >>> la
2. [1, 2, 3, 'a', 'b', 'c', 'qiwsir', 'python']
3. >>> la[2]
4. 3
5. >>> la[2:5]
6. [3, 'a', 'b']
7. >>> la[:7]
8. [1, 2, 3, 'a', 'b', 'c', 'qiwsir']
```

如果考虑反过来的情况，能不能通过某个元素，找到它在list中的编号呢？

看官的需要就是python的方向，你想到，python就做到。

```

1. >>> la
2. [1, 2, 3, 'a', 'b', 'c', 'qiwsir', 'python']
3. >>> la.index(3)
4. 2
5. >>> la.index('a')
6. 3
7. >>> la.index(1)
8. 0
9. >>> la.index('qi')      #如果不存在，就报错
10. Traceback (most recent call last):
11.   File "<stdin>", line 1, in <module>
12.   ValueError: 'qi' is not in list
13. >>> la.index('qiwsir')
14. 6

```

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

依然是上一条官方解释：

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

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

是不是说的非常清楚明白了？

先到这里，下讲还继续有容乃大的list。

[首页](#) | [上一讲](#) | [下一讲](#)

有容乃大的list(3)

- 有容乃大的list(3)
 - 对list的操作
 - 向list中插入一个元素
 - 删除list中的元素

有容乃大的list(3)

现在是讲lis的第三章了。俗话说，事不过三，不知道在开头，我也不知道这一讲是不是能够把基础的list知识讲完呢。哈哈。其实如果真正写文章，会在写完之后把这句话删掉的。而我则是完全像跟看官聊天一样，就不删除了。

继续。

对list的操作

向list中插入一个元素

前面有一个向list中追加元素的方法，那个追加是且只能是将新元素添加在list的最后一个。如：

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

从这个操作，就可以说明list是可以随时改变的。这种改变的含义只它的大小即所容纳元素的个数以及元素内容，可以随时直接修改，而不用进行转换。这和str有着很大的不同。对于str，就不能进行字符的

追加。请看官要注意比较，这也是str和list的重要区别。

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

我特别引导列为看官要通过[官方文档来理解](#)：

```
list.insert(i, x)
```

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

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

根据官方文档的说明，我们做下面的实验，请看官从实验中理解：

```
1. >>> all_users
2. ['qiwsir', 'github', 'io']
3. >>> all_users.insert("python")      #list.insert(i,x),要求有两个参数,
    少了就报错
4. Traceback (most recent call last):
5.   File "<stdin>", line 1, in <module>
6. TypeError: insert() takes exactly 2 arguments (1 given)
7.
8. >>> all_users.insert(0,"python")
9. >>> all_users
10. ['python', 'qiwsir', 'github', 'io']
11.
12. >>> all_users.insert(1,"http://")
13. >>> all_users
14. ['python', 'http://', 'qiwsir', 'github', 'io']
15.
16. >>> length = len(all_users)
17. >>> length
18. 5
```



```

19.
20. >>> all_users.insert(length, "algorithm")
21. >>> all_users
22. ['python', 'http://', 'qiwsir', 'github', 'io', 'algorithm']

```

小结：

- `list.insert(i, x)`, 将新的元素x 插入到原list中的list[i] 前面
- 如果`i==len(list)`, 意思是在后面追加, 就等同于 `list.append(x)`

删除list中的元素

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

```
list.remove(x)
```

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

```
list.pop([i])
```

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

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

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

报错。

```

1. >>> all_users
2. ['python', 'http://', 'qiwsir', 'github', 'io', 'algorithm']
3. >>> all_users.remove("http://")
4. >>> all_users          #的确是把"http://"删除了
5. ['python', 'qiwsir', 'github', 'io', 'algorithm']
6.
7. >>> all_users.remove("tianchao")          #原list中没有“tianchao”，要删除，就报错。
8. Traceback (most recent call last):
9.   File "<stdin>", line 1, in <module>
10. ValueError: list.remove(x): x not in list

```

注意两点：

- 如果正确删除，不会有任何反馈。没有消息就是好消息。
- 如果所删除的内容不在list中，就报错。注意阅读报错信息：x not in list

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

如果看官想到这里，就是在编程的旅程上一进步。python的确让我们这么做。

```

1. >>> all_users
2. ['python', 'qiwsir', 'github', 'io', 'algorithm']
3. >>> "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
4. True
5.
6. >>> if "python" in all_users:
7. ...     all_users.remove("python")
8. ...     print all_users
9. ... else:

```

```

10. ...     print "'python' is not in all_users"
11. ...
12. ['qiwsir', 'github', 'io', 'algorithm']      #删除了"python"元素
13.
14. >>> if "python" in all_users:
15. ...     all_users.remove("python")
16. ...     print all_users
17. ... else:
18. ...     print "'python' is not in all_users"
19. ...
20. '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。

另外一个删除`list.pop([i])`会怎么样呢？看看文档，做做实验。

```

1. >>> all_users
2. ['qiwsir', 'github', 'io', 'algorithm']
3. >>> all_users.pop()      #list.pop([i]),圆括号里面是[i],表示这个序号是可选的
4. 'algorithm'              #如果不写,就如同这个操作,默认删除最后一个,并且将该结果返回
5.
6. >>> all_users
7. ['qiwsir', 'github', 'io']
8.
9. >>> all_users.pop(1)      #指定删除编号为1的元素"github"
10. 'github'
11.
12. >>> all_users
13. ['qiwsir', 'io']
14. >>> all_users.pop()
15. 'io'
16.
17. >>> all_users            #只有一个元素了,该元素编号是0
18. ['qiwsir']

```

```
19. >>> 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
20. Traceback (most recent call last):
21.   File "<stdin>", line 1, in <module>
22. IndexError: pop index out of range      #删除索引超出范围，就是1不在list的编号范围之内
```

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

list是一个有意思的东西，内涵丰富。看来下一讲还要继续讲list。并且可能会做一个有意思的游戏。请期待。

[首页](#) | [上一讲](#) | [下一讲](#)

有容乃大的list(4)

- 有容乃大的list(4)
 - 对list的操作
 - range(start, stop)生成数字list
 - 排排坐，分果果
 - 一个非常重要的方法

有容乃大的list(4)

list的话题的确不少，而且，在编程中，用途也非常多。

有看官可能要问了，如果要生成一个list，除了要把元素一个一个写上之外，有没有能够让计算机自己按照某个规律生成list的方法呢？

如果你提出了这个问题，充分说明你是一个“懒人”，不过这不是什么坏事情，这个世界就是因为“懒人”的存在而进步。“懒人”其实不懒。

对list的操作

range(start, stop)生成数字list

range(start, stop[, step])是一个内置函数。

要研究清楚一些函数特别是内置函数的功能，建议看官首先要明白内置函数名称的含义。因为在python中，名称不是随便取的，是代表一定意义的。关于取名字问题，可以看参考本系列的：[永远强大的函数](#)中的《取名字的学问》部分内容。

range

n. 范围；幅度；排；山脉

vi. （在...内）变动；平行，列为一行；延伸；漫游；射程达到

vt. 漫游；放牧；使并列；归类于；来回走动

在具体实验之前，还是按照管理，摘抄一段[官方文档的原话](#)，让我们能够深刻理解之：

*This is a versatile function to create lists containing arithmetic progressions. It is most often used in for loops. The arguments must be plain integers. If the step argument is omitted, it defaults to 1. If the start argument is omitted, it defaults to 0. The full form returns a list of plain integers [start, start + step, start + 2 * step, ...]. If step is positive, the last element is the largest start + i * step less than stop; if step is negative, the last element is the smallest start + i * step greater than stop. step must not be zero (or else ValueError is raised).*

从这段话，我们可以得出关于range()函数的以下几点：

- 这个函数可以创建一个数字元素组成的列表。
- 这个函数最常用于for循环（关于for循环，马上就要涉及到了）
- 函数的参数必须是整数，默认从0开始。返回值是类似[start, start + step, start + 2*step, ...]的列表。
- step默认值是1。如果不写，就是按照此值。
- 如果step是正数，返回list的最最后的值不包含stop值，即start+istep这个值小于stop；如果step是负数，start+istep的值大于stop。
- step不能等于零，如果等于零，就报错。

在实验开始之前，再解释range(start, stop[, step])的含义：

- start：开始数值，默认为0，也就是如果不写这项，就是认为start=0
- stop：结束的数值，必须要写的。
- step：变化的步长，默认是1，也就是不写，就是认为步长为1。坚决不能为0

实验开始，请以各项对照前面的讲述：

```

1. >>> range(9)                #stop=9, 别的都没有写, 含义就是range(0,9,1)
2. [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
3. >>> range(0,9)
4. [0, 1, 2, 3, 4, 5, 6, 7, 8]
5. >>> range(0,9,1)
6. [0, 1, 2, 3, 4, 5, 6, 7, 8]
7.
8. >>> range(1,9)                #start=1
9. [1, 2, 3, 4, 5, 6, 7, 8]
10.
11. >>> range(0,9,2)            #step=2, 每个元素等于start+i*step,
12. [0, 2, 4, 6, 8]

```

仅仅解释一下range(0,9,2)

- 如果是从0开始, 步长为1, 可以写成range(9)的样子, 但是, 如果步长为2, 写成range(9,2)的样子, 计算机就有点糊涂了, 它会认为start=9, stop=2。所以, 在步长不为1的时候, 切忌, 要把start的值也写上。
- start=0, step=2, stop=9。list中的第一个值是start=0, 第二个值是start+1step=2 (注意, 这里是1, 不是2, 不要忘记, 前面已经讲过, 不论是list还是str, 对元素进行编号的时候, 都是从0开始的), 第n个值就是start+(n-1)step。直到小于stop前的那个值。

熟悉了上面的计算过程, 看看下面的输入谁是什么结果?

```
1. >>> 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：

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

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？

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

有了这个内置函数，很多事情就简单了。比如：

```
1. >>> range(0, 100, 2)
2. [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34,
    36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68,
    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list，就非常简单地搞定了。

思考一个问题，现在有一个列表，比如是

`["I", "am", "a", "pythoner", "I", "am", "learning", "it", "with", "qiwsir"]`，要得到这个list的所有序号组成的list，但是不能一个一个用手指头来数。怎么办？

请沉思两分钟之后，自己实验一下，然后看下面。

```

1. >>> pythoner
2. ['I', 'am', 'a', 'pythoner', 'I', 'am', 'learning', 'it', 'with',
   'qiwsir']
3. >>> py_index = range(len(pythoner))    #以len(pythoner)为stop的值
4. >>> py_index
5. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```

再用手指头指着pythoner里面的元素，数一数，是不是跟结果一样。

排排坐，分果果

排序，不管在现实还是在网络上都是随处可见的。梁山好汉要从第一个排序到第108个，这是一个不很容易搞定的活。

前面提到的内置函数range()得到的结果，就是一个排好序的。对于一个没有排好序的list，怎么排序呢？

有两个方法可以实现对list的排序：

- list.sort(cmp=None, key=None, reverse=False)
- sorted(iterable[, cmp[, key[, reverse]])

通过下面的实验，可以理解如何排序的方法

```

1. >>> number = [1,4,6,2,9,7,3]
2. >>> number.sort()
3. >>> number
4. [1, 2, 3, 4, 6, 7, 9]
5.
6. >>> number = [1,4,6,2,9,7,3]
7. >>> number
8. [1, 4, 6, 2, 9, 7, 3]
9. >>> sorted(number)
10. [1, 2, 3, 4, 6, 7, 9]

```

```
11.  
12. >>> number = [1,4,6,2,9,7,3]  
13. >>> number  
14. [1, 4, 6, 2, 9, 7, 3]  
15. >>> number.sort(reverse=True)    #开始实现倒序  
16. >>> number  
17. [9, 7, 6, 4, 3, 2, 1]  
18.  
19. >>> number = [1,4,6,2,9,7,3]  
20. >>> number  
21. [1, 4, 6, 2, 9, 7, 3]  
22. >>> sorted(number,reverse=True)  
23. [9, 7, 6, 4, 3, 2, 1]
```

其实，在高级语言中，排序是一个比较热门对的话题，如果有兴趣的读者，可以到我写的[有关算法](#)中查看有关排序的话题。

至此，有关list的基本操作的内置函数，就差不多了。不过最后，还要告诉看官们一个学习方法。因为python的内置函数往往不少，有时候光凭教程，很难学到全部，那么，最关键地是要自己会查找都有哪些函数可以用。怎么查找呢？

一个非常重要的方法

假设有一个list，如何知道它所拥有的内置函数呢？请用`help()`，帮助我吧。

```
1. >>> help(list)
```

就能够看到所有的关于list的函数，以及该函数的使用方法。

[首页](#) | [上一讲](#) | [下一讲](#)

list和str比较

- list和str比较
 - 相同点
 - 都属于序列类型的数据
 - 区别
 - 多维list
 - list和str转化
 - `str.split()`
 - `"[sep]".join(list)`
 - 公告：

list和str比较

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

相同点

都属于序列类型的数据

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

例如：

```
1. >>> welcome_str = "Welcome you"
2. >>> welcome_str[0]
3. 'W'
```

```

4. >>> welcome_str[1]
5. 'e'
6. >>> welcome_str[len(welcome_str)-1]
7. 'u'
8. >>> welcome_str[:4]
9. 'Welc'
10. >>> a = "python"
11. >>> a*3
12. 'pythonpythonpython'
13.
14. >>> git_list = ["qiwsir","github","io"]
15. >>> git_list[0]
16. 'qiwsir'
17. >>> git_list[len(git_list)-1]
18. 'io'
19. >>> git_list[0:2]
20. ['qiwsir', 'github']
21. >>> b = ['qiwsir']
22. >>> b*7
23. ['qiwsir', 'qiwsir', 'qiwsir', 'qiwsir', 'qiwsir', 'qiwsir',
    'qiwsir']

```

对于此类数据，下面一些操作是类似的：

```

1. >>> first = "hello,world"
2. >>> welcome_str
3. 'Welcome you'
4. >>> first+", "+welcome_str    #用+号连接str
5. 'hello,world,Welcome you'
6. >>> welcome_str                #原来的str没有受到影响，即上面的+号连接后从新
    生成了一个字符串
7. 'Welcome you'
8. >>> first
9. 'hello,world'
10.
11. >>> language = ['python']
12. >>> git_list
13. ['qiwsir', 'github', 'io']

```

```

14. >>> language + git_list      #用+号连接list, 得到一个新的list
15. ['python', 'qiwsir', 'github', 'io']
16. >>> git_list
17. ['qiwsir', 'github', 'io']
18. >>> language
19. ['python']
20.
21. >>> len(welcome_str)      #得到字符数
22. 11
23. >>> len(git_list)        #得到元素数
24. 3

```

区别

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

首先看对list的这些操作，其特点是在原处将list进行了修改：

```

1. >>> git_list
2. ['qiwsir', 'github', 'io']
3.
4. >>> git_list.append("python")
5. >>> git_list
6. ['qiwsir', 'github', 'io', 'python']
7.
8. >>> git_list[1]
9. 'github'
10. >>> git_list[1] = 'github.com'
11. >>> git_list
12. ['qiwsir', 'github.com', 'io', 'python']
13.
14. >>> git_list.insert(1, "algorithm")
15. >>> git_list
16. ['qiwsir', 'algorithm', 'github.com', 'io', 'python']
17.
18. >>> git_list.pop()

```

```

19. 'python'
20.
21. >>> del git_list[1]
22. >>> git_list
23. ['qiwsir', 'github.com', 'io']

```

以上这些操作，如果用在str上，都会报错，比如：

```

1. >>> welcome_str
2. 'Welcome you'
3.
4. >>> welcome_str[1]='E'
5. Traceback (most recent call last):
6. File "<stdin>", line 1, in <module>
7. TypeError: 'str' object does not support item assignment
8.
9. >>> del welcome_str[1]
10. Traceback (most recent call last):
11. File "<stdin>", line 1, in <module>
12. TypeError: 'str' object doesn't support item deletion
13.
14. >>> welcome_str.append("E")
15. Traceback (most recent call last):
16. File "<stdin>", line 1, in <module>
17. AttributeError: 'str' object has no attribute 'append'

```

如果要修改一个str，不得不这样。

```

1. >>> welcome_str
2. 'Welcome you'
3. >>> welcome_str[0]+"E"+welcome_str[2:] #从新生成一个str
4. 'WEelcome you'
5. >>> welcome_str #对原来的没有任何影响
6. 'Welcome you'

```

其实，在这种做法中，相当于从新生成了一个str。

多维list

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

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

以上显示了多维list以及访问方式。在多维的情况下，里面的list也跟一个前面元素一样对待。

list和str转化

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：

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

得到了对这个内置函数的完整说明。特别强调：这是一种非常好的学习方法


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

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

不管是否看懂上面这段话，都可以看例子。还是希望看官能够理解上面的内容。

```
1. >>> line = "Hello.I am qiwsir.Welcome you."
2.
3. >>> line.split(".")      #以英文的句点为分隔符，得到list
4. ['Hello', 'I am qiwsir', 'Welcome you', '']
5.
6. >>> 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.
7. ['Hello', 'I am qiwsir.Welcome you.']
8.
9. >>> name = "Albert Ainstain"    #也有可能用空格来做为分隔符
10. >>> name.split(" ")
11. ['Albert', 'Ainstain']
```

下面的例子，让你更有点惊奇了。

```
1. >>> 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
2. >>> print s      #输出之后的样式
3. I am, writing
4. python  book on line
5. >>> s.split()    #用split(),但是括号中不输入任何参数
6. ['I', 'am,', 'writing', 'python', 'book', 'on', 'line']
```

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

“[sep]”.join(list)

join可以说是split的逆运算，举例：

```
1. >>> name
2. ['Albert', 'Ainstain']
3. >>> "".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
4. 'AlbertAinstain'
5. >>> ".".join(name)     #以英文的句点做为连接分隔符
6. 'Albert.Ainstain'
7. >>> " ".join(name)     #以空格做为连接的分隔符
8. 'Albert Ainstain'
```

回到上面那个神奇的例子中，可以这么使用join。

```
1. >>> s = "I am, writing\npython\tbook on line"
2. >>> print s
3. I am, writing
4. python  book on line
5. >>> s.split()
6. ['I', 'am,', 'writing', 'python', 'book', 'on', 'line']
7. >>> " ".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？
8. 'I am, writing python book on line'
```

公告：

有朋友愿意学习python, 恭请到我的github上follower我，并且可以给我发邮件，也可以在微博上关注我。更多有关信息请看：[易水禾：http://qiwsir.github.io](http://qiwsir.github.io)

[首页](#) | [上一讲](#) | [下一讲](#)

画圈还不简单吗

- [画圈还不简单吗？](#)
 - [简单的for循环例子](#)
 - [上一个台阶](#)

画圈还不简单吗？

画圈？换一个说法就是循环。循环，是高级语言编程中重要的工作。现实生活中，很多事情都是在循环，日月更迭，斗转星移，无不是循环；王朝更迭，寻常百姓，也都是循环。

在python中，循环有一个语句：for语句。

简单的for循环例子

```
1. >>> hello = "world"
2. >>> for i in hello:
3. ...     print i
4. ...
5. w
6. o
7. r
8. l
9. d
```

上面这个for循环是怎么工作的呢？

1. hello这个变量引用的是"world"这个str类型的数据
2. 变量 i 通过hello找到它所引用的"world", 然后从第一字符开始，依次获得该字符的引用。
3. 当 i="w"的时候，执行print i，打印出了字母w，结束之后循

环第二次，让 `i="e"`，然后执行`print i`，打印出字母e，如此循环下去，一直到最后一个字符被打印出来，循环自动结束

顺便补充一个`print`的技巧，上面的打印结果是竖着排列，也就是每打印一个之后，就自动换行。如果要让打印的在一行，可以用下面的方法，在打印的后面加一个逗号（英文）

```
1. >>> for i in hello:
2.     ...     print i,
3.     ...
4. w o r l d
5.
6. >>> for i in hello:
7.     ...     print i+",",      #为了美观，可以在每个字符后面加一个逗号分割
8.     ...
9. w, o, r, l, d,
10. >>>
```

因为可以通过使用索引编号（偏移量）做为下表，得到某个字符。所以，还可以通过下面的循环方式实现上面代码中同样功能：

```
1. >>> for i in range(len(hello)):
2.     ...     print hello[i]
3.     ...
4. w
5. o
6. r
7. l
8. 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”每个字母的编号，即偏移量。

3. `for i in range(len(hello))`,就相当于`for i in [0,1,2,3,4]`,让*i*依次等于list中的各个值。当*i*=0时,打印`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list,感觉不过瘾呀。那好,看下面对list的循环:

```
1. >>> ls_line
2. ['Hello', 'I am qiwsir', 'Welcome you', '']
3. >>> for word in ls_line:
4. ...     print word
5. ...
6. Hello
7. I am qiwsir
8. Welcome you
9.
10. >>> for i in range(len(ls_line)):
11. ...     print ls_line[i]
12. ...
13. Hello
14. I am qiwsir
15. Welcome you
```

上一个台阶

我们已经理解了for语句的基本工作流程,如果写一个一般化的公式,可以这么表示:

1. `for` 循环规则:
2. 操作语句

用for语句来解决一个实际问题。

例：找出100以内的能够被3整除的正整数。

分析：这个问题有两个限制条件，第一是100以内的正整数，根据前面所学，可以用`range(1,100)`来实现；第二个是要解决被3整除的问题，假设某个正整数`n`，这个数如果能够被3整除，也就是`n%3`(%是取余数)为0.那么如何得到`n`呢，就是要用for循环。

以上做了简单分析，要实现流程，还需要细化一下。按照前面曾经讲授过的一种方法，要画出问题解决的流程图。



下面写代码就是按图索骥了。

代码：

```
1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  aliquot = []
5.
6.  for n in range(1,100):
7.      if n%3 == 0:
8.          aliquot.append(n)
9.
10. print aliquot
```

代码运行结果：

```
1.  [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]
```

这里仅仅列举一个简单的例子，看官可以在这个例子基础上深入：打印某范围内的偶数/奇数等。

如果要对list的循环进行深入了解的，可以到专门撰写的[python and algorithm](#)里面阅读有关文章

再深点，更懂list

- 再深点，更懂list
 - list解析
 - enumerate

再深点，更懂list

对于list，由于她的确非常非常庞杂，在python中应用非常广泛，所以，虽然已经介绍完毕了基础内容，这里还要用一讲深入一点点，往往越深入越...

list解析

先看下面的例子，这个例子是想得到1到9的每个整数的平方，并且将结果放在list中打印出来

```
1. >>> power2 = []
2. >>> for i in range(1,10):
3. ...     power2.append(i*i)
4. ...
5. >>> power2
6. [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

python有一个非常有意思的功能，就是list解析，就是这样的：

```
1. >>> squares = [x**2 for x in range(1,10)]
2. >>> squares
3. [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

看到这个结果，看官还不惊叹吗？这就是python，追求简洁优雅的python！

其官方文档中有这样一段描述，道出了list解析的真谛：

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

还记得[前面一讲](#)中的那个问题吗？

找出100以内的能够被3整除的正整数。

我们用的方法是：

```
1. aliquot = []
2.
3. for n in range(1,100):
4.     if n%3 == 0:
5.         aliquot.append(n)
6.
7. print aliquot
```

好了。现在用list解析重写，会是这样的：

```
1. >>> aliquot = [n for n in range(1,100) if n%3==0]
2. >>> aliquot
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]
```

震撼了。绝对牛X！

再来一个，是网友[ccbikai](#)提供的，比牛X还牛X。

```
1. >>> print range(3,100,3)
2. [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]
```

这就是python有意思的地方，也是计算机高级语言编程有意思的地

方，你只要动脑筋，总能找到惊喜的东西。

其实，不仅仅对数字组成的list，所有的都可以如此操作。请在平复了激动的心之后，默默地看下面的代码，感悟一下list解析的魅力。

```
1. >>> mybag = [' glass', ' apple', 'green leaf ']    #有的前面有空格，有的后面有空格
2. >>> [one.strip() for one in mybag]                #去掉元素前后的空格
3. ['glass', 'apple', 'green leaf']
```

enumerate

这是一个有意思的内置函数，本来我们可以通过for i in range(len(list))的方式得到一个list的每个元素编号，然后在用list[i]的方式得到该元素。如果要同时得到元素编号和元素怎么办？就是这样的了：

```
1. >>> for i in range(len(week)):
2.     ...     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
3.     ...
4. monday is 0
5. sunday is 1
6. friday is 2
```

python中提供了一个内置函数enumerate，能够实现类似的功能

```
1. >>> for (i,day) in enumerate(week):
2.     ...     print day+' is '+str(i)
3.     ...
4. monday is 0
5. sunday is 1
6. friday is 2
```

算是一个有意思的内置函数了，主要是提供一个简单快捷的方法。

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。

```
1. >>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
2. >>> list(enumerate(seasons))
3. [(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
4. >>> list(enumerate(seasons, start=1))
5. [(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

在这里有类似(0, 'Spring')这样的东西，这是另外一种数据类型，待后面详解。

下面将enumerate函数和list解析联合起来，同时显示，在进行list解析的时候，也可以包含进函数（关于函数，可以参考的章节有：[初始强大的函数](#)，[重回函数](#)）。

```
1. >>> def treatment(pos, element):
2. ...     return "%d: %s"%(pos,element)
3. ...
4. >>> seq = ["qiwsir","qiwsir.github.io","python"]
5. >>> [ treatment(i, ele) for i,ele in enumerate(seq) ]
6. ['0: qiwsir', '1: qiwsir.github.io', '2: python']
```

看官也可以用[小话题大函数](#)中的lambda函数来写上面的代码：

```
1. >>> seq = ["qiwsir","qiwsir.github.io","python"]
2. >>> foo = lambda i,ele:"%d:%s"%(i,ele)           #lambda函数，给代码带
来简介
```

```
3. >>> [foo(i,ele) for i,ele in enumerate(seq)]
4. ['0:qiwsir', '1:qiwsir.github.io', '2:python']
```

[首页](#) | [上一讲](#) | [下一讲](#)

字典，你还记得吗？

- 字典，你还记得吗？
 - 概述
 - 创建dict
 - 访问dict的值
 - 知识

字典，你还记得吗？

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，乃至于已经有手机版的各种字典了。。我曾经用过一本小小的《新华字典》。

《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，不是为了叙旧。而是提醒看官想想我们如何使用字典：先查索引（不管是拼音还是偏旁查字），然后通过索引找到相应内容。

这种方法能够快捷的找到目标。

在python中，也有一种数据与此相近，不仅相近，这种数据的名称就叫做dictionary，翻译过来是字典，类似于前面的int/str/list，这种类型数据名称是:dict

依据管理，要知道如何建立dict和它有关属性方法。

因为已经有了此前的基础，所以，学这个就可以加快了。

前面曾经建议看官一个很好的学习探究方法，比如想了解str的有关属性方法，可以在交互模式下使用：

```
1. >>>help(str)
```

将得到所有的有关内容。

现在换一个，使用dir，也能得到相同的结果。只是简单一些罢了。请在交互模式下：

```
1. >>> dir(dict)
2. ['__class__', '__cmp__', '__contains__', '__delattr__',
    '__delitem__', '__doc__', '__eq__', '__format__', '__ge__',
    '__getattr__', '__getitem__', '__gt__', '__hash__',
    '__init__', '__iter__', '__le__', '__len__', '__lt__', '__ne__',
    '__new__', '__reduce__', '__reduce_ex__', '__repr__',
    '__setattr__', '__setitem__', '__sizeof__', '__str__',
    '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'has_key',
    'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop',
    'popitem', 'setdefault', 'update', 'values', 'viewitems',
    'viewkeys', 'viewvalues']
```

以__（双下划线）开头的先不管。看后面的。如果要想深入了解，可以这样：

```
1. >>> help(dict.values)
```

然后出现：

```
1. Help on method_descriptor:
2.
3. values(...)
4.     D.values() -> list of D's values
5. (END)
```

也就是在这里显示出了values这个内置函数的使用方法。敲击键盘上的q键退回。

概述

python中的dict具有如下特点：

- dict是可变的
- dict可以存储任意数量的Python对象
- dict可以存储任何python数据类型
- dict以：key:value，即“键：值”对的形式存储数据，每个键是唯一的。
- dict也被称为关联数组或哈希表。

以上诸条，如果还不是很理解，也没有关系，通过下面的学习，特别是通过各种实验，就能理解了。

创建dict

话说创建dict的方法可是远远多于前面的int/str/list，为什么会多呢？一般规律是复杂点的东西都会有多种渠道生成，这也是从安全便捷角度考虑吧。

方法1：

创建一个空的dict，这个空dict，可以在以后向里面加东西用。

```
1. >>> mydict = {}
2. >>> mydict
3. {}
```

创建有内容的dict。


```

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

```

“name”: “qiwsir” 就是一个键值对，前面的name叫做键（key），后面的qiwsir是前面的键所对应的值（value）。在一个dict中，键是唯一的，不能重复；值则是对应于键，值可以重复。键值之间用(:)英文的分号，每一对键值之间用英文的逗号(,) 隔开。

```

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

```

如下，演示了从一个空的dict开始增加内容的过程：

```

1. >>> mydict = {}
2. >>> mydict
3. {}
4. >>> mydict["site"] = "qiwsir.github.io"
5. >>> mydict[1] = 80
6. >>> mydict[2] = "python"
7. >>> mydict["name"] = ["zhangsan", "lisi", "wangwu"]
8. >>> mydict
9. {1: 80, 2: 'python', 'site': 'qiwsir.github.io', 'name':
    ['zhangsan', 'lisi', 'wangwu']}
10.
11. >>> mydict[1] = 90    #如果这样，则是修改这个键的值
12. >>> mydict
13. {1: 90, 2: 'python', 'site': 'qiwsir.github.io', 'name':
    ['zhangsan', 'lisi', 'wangwu']}

```

方法2：

```
1. >>> name = ("first", "Google"), ("second", "Yahoo")           #这是另外一种
    数据类型，称之为元组，后面会讲到
2. >>> website = dict(name)
3. >>> website
4. {'second': 'Yahoo', 'first': 'Google'}
```

方法3:

```
1. 这个方法，跟上面的不同在于使用fromkeys
2.
3. >>> website = {}.fromkeys(("third", "forth"), "facebook")
4. >>> website
5. {'forth': 'facebook', 'third': 'facebook'}
```

需要提醒的是，这种方法是从新建立一个dict。

访问dict的值

因为dict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。

```
1. >>> person
2. {'name2': 'qiwsir', 'name': 'qiwsir', 'language': 'python', 'site':
    'qiwsir.github.io'}
3. >>> person['name']
4. 'qiwsir'
5. >>> person['language']
6. 'python'
7. >>> site = person['site']
8. >>> print site
9. qiwsir.github.io
```

如同前面所讲，通过键能够增加dict中的值，通过键能够改变dict中的值，通过键也能够访问dict中的值。

看官可以跟list对比一下。如果我们访问list中的元素，可以通过索引值得到（list[i]），如果是让机器来巡回访问，就可以用for语句。复习一下：

```
1. >>> person_list = ["qiwsir", "Newton", "Boolean"]
2. >>> for name in person_list:
3. ...     print name
4. ...
5. qiwsir
6. Newton
7. Boolean
```

那么，dict是不是也可以用for语句来循环访问呢？当然可以，来看例子：

```
1. >>> person
2. {'name2': 'qiwsir', 'name': 'qiwsir', 'language': 'python', 'site':
   'qiwsir.github.io'}
3. >>> for key in person:
4. ...     print person[key]
5. ...
6. qiwsir
7. qiwsir
8. python
9. qiwsir.github.io
```

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。

[首页](#) | [上一讲](#) | [下一讲](#)

字典的操作方法

- `dict()`的操作方法
 - 嵌套
 - 获取键、值
 - 其它几种常用方法

`dict()`的操作方法

`dict`的很多方法跟`list`有类似的地方，下面一一道来，并且会跟`list`做一个对比

嵌套

嵌套在`list`中也存在，就是元素是`list`，在`dict`中，也有类似的样式：

```
1. >>> a_list = [[1,2,3],[4,5],[6,7]]
2. >>> a_list[1][1]
3. 5
4. >>> a_dict = {1:
    {"name":"qiwsir"},2:"python","email":"qiwsir@gmail.com"}
5. >>> a_dict
6. {1: {'name': 'qiwsir'}, 2: 'python', 'email': 'qiwsir@gmail.com'}
7. >>> a_dict[1]['name']      #一个嵌套的dict访问其值的方法：一层一层地写出键
8. 'qiwsir'
```

获取键、值

在[上一讲](#)中，已经知道可以通过`dict`的键得到其值。例上面的例子。

还有别的方法得到键值吗？有！python一般不是只有一个方法实现某

个操作的。

```

1. >>> website =
    {1:"google","second":"baidu",3:"facebook","twitter":4}
2.
3. >>>#用d.keys()的方法得到dict的所有键，结果是list
4. >>> website.keys()
5. [1, 'second', 3, 'twitter']
6.
7. >>>#用d.values()的方法得到dict的所有值，如果里面没有嵌套别的dict，结果是list
8. >>> website.values()
9. ['google', 'baidu', 'facebook', 4]
10.
11. >>>#用items()的方法得到了一组一组的键值对，
12. >>>#结果是list，只不过list里面的元素是元组
13. >>> website.items()
14. [(1, 'google'), ('second', 'baidu'), (3, 'facebook'), ('twitter',
    4)]

```

从上面的结果中，我们就可以看出，还可以用for语句循环得到相应内容。例如：

```

1. >>> for key in website.keys():
2.     ...     print key,type(key)
3.     ...
4. 1 <type 'int'>
5. second <type 'str'>
6. 3 <type 'int'>
7. twitter <type 'str'>
8.
9. >>>#下面的方法和上面的方法是一样的
10. >>> for key in website:
11.     ...     print key,type(key)
12.     ...
13. 1 <type 'int'>
14. second <type 'str'>

```

```
15. 3 <type 'int'>
16. twitter <type 'str'>
```

以下两种方法等效：

```
1. >>> for value in website.values():
2.     ...     print value
3.     ...
4. google
5. baidu
6. facebook
7. 4
8.
9. >>> for key in website:
10.     ...     print website[key]
11.     ...
12. google
13. baidu
14. facebook
15. 4
```

下面的方法又是等效的：

```
1. >>> for k,v in website.items():
2.     ...     print str(k)+":"+str(v)
3.     ...
4. 1:google
5. second:baidu
6. 3:facebook
7. twitter:4
8.
9. >>> for k in website:
10.     ...     print str(k)+":"+str(website[k])
11.     ...
12. 1:google
13. second:baidu
14. 3:facebook
```

```
15. twitter:4
```

下面的方法也能得到键值，不过似乎要多敲键盘

```
1. >>> website
2. {1: 'google', 'second': 'baidu', 3: 'facebook', 'twitter': 4}
3. >>> website.get(1)
4. 'google'
5. >>> website.get("second")
6. 'baidu'
```

其它几种常用方法

dict中的方法在这里不做过多的介绍，因为前面一节中已经列出来类，看官如果有兴趣可以一一尝试。下面列出几种常用的

```
1. >>> len(website)
2. 4
3. >>> website
4. {1: 'google', 'second': 'baidu', 3: 'facebook', 'twitter': 4}
5.
6. >>> new_web = website.copy()    #拷贝一份，这个拷贝也叫做浅拷贝，对应着还有深拷贝。
7. >>> new_web                      #两者区别，可以google一下。
8. {1: 'google', 'second': 'baidu', 3: 'facebook', 'twitter': 4}
```

删除键值对的方法有两个，但是两者有一点区别

```
1. >>>#d.pop(key)，根据key删除相应的键值对，并返回该值
2. >>> new_web.pop('second')
3. 'baidu'
4.
5. >>> del new_web[3]              #没有返回值，如果删除键不存在，返回错误
6. >>> new_web
7. {1: 'google', 'twitter': 4}
8. >>> del new_web[9]
```



```

9. Traceback (most recent call last):
10. File "<stdin>", line 1, in <module>
11. KeyError: 9

```

用`d.update(d2)`可以把`d2`合并到`d`中。

```

1. >>> cnweb
2. {'qq': 'first in cn', 'python': 'qiwsir.github.io', 'alibaba':
   'Business'}
3. >>> website
4. {1: 'google', 'second': 'baidu', 3: 'facebook', 'twitter': 4}
5.
6. >>> website.update(cnweb)    #把cnweb合并到website内
7. >>> website                  #变化了
8. {'qq': 'first in cn', 1: 'google', 'second': 'baidu', 3:
   'facebook', 'python': 'qiwsir.github.io', 'twitter': 4, 'alibaba':
   'Business'}
9. >>> cnweb                    #not changed
10. {'qq': 'first in cn', 'python': 'qiwsir.github.io', 'alibaba':
    'Business'}

```

在本讲最后，要提醒看官，在python3中，dict有不少变化，比如能够进行字典解析，就类似列表解析那样，这可是非常有意思的东西哦。

[首页](#) | [上一讲](#) | [下一讲](#)

有点简约的元组

- 有点简约的元组
 - 像list那样访问元素和切片
 - tuple用在哪里？

有点简约的元组

关于元组，上一讲中涉及到了这个名词。本讲完整地讲述它。

先看一个例子：

```
1. >>>#变量引用str
2. >>> s = "abc"
3. >>> s
4. 'abc'
5.
6. >>>#如果这样写，就会是...
7. >>> t = 123, 'abc', ["come", "here"]
8. >>> t
9. (123, 'abc', ['come', 'here'])
```

上面例子中看到的变量t，并没有报错，也没有“最后一个有效”，而是将对象做为一个新的数据类型：tuple（元组），赋值给了变量t。

元组是用圆括号括起来的，其中的元素之间用逗号隔开。（都是英文半角）

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

```
1. >>> t = 1, "23", [123, "abc"], ("python", "learn")    #元素多样性，近list
2. >>> t
```

```

3. (1, '23', [123, 'abc'], ('python', 'learn'))
4.
5. >>> t[0] = 8                                     #不能原地修改, 近str
6. Traceback (most recent call last):
7.   File "<stdin>", line 1, in <module>
8. TypeError: 'tuple' object does not support item assignment
9.
10. >>> t.append("no")
11. Traceback (most recent call last):
12.   File "<stdin>", line 1, in <module>
13. AttributeError: 'tuple' object has no attribute 'append'
14.     >>>

```

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

像list那样访问元素和切片

先复习list中的一点知识：

```

1. >>> one_list = ["python", "qiwsir", "github", "io"]
2. >>> one_list[2]
3. 'github'
4. >>> one_list[1:]
5. ['qiwsir', 'github', 'io']
6. >>> for word in one_list:
7. ...     print word
8. ...
9. python
10. qiwsir
11. github
12. io
13. >>> len(one_list)
14. 4

```

下面再实验一下，上面的list如果换成tuple是否可行

```

1. >>> t
2. (1, '23', [123, 'abc'], ('python', 'learn'))
3. >>> t[2]
4. [123, 'abc']
5. >>> t[1:]
6. ('23', [123, 'abc'], ('python', 'learn'))
7. >>> for every in t:
8. ...     print every
9. ...
10. 1
11. 23
12. [123, 'abc']
13. ('python', 'learn')
14. >>> len(t)
15. 4
16.
17. >>> t[2][0]      #还能这样呀，哦对了，list中也能这样
18. 123
19. >>> t[3][1]
20. 'learn'

```

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

分别用**list()**和**tuple()**能够实现两者的转化：

```

1. >>> t
2. (1, '23', [123, 'abc'], ('python', 'learn'))
3. >>> t1s = list(t)      #tuple-->list
4. >>> t1s
5. [1, '23', [123, 'abc'], ('python', 'learn')]
6.
7. >>> t_tuple = tuple(t1s)      #list-->tuple
8. >>> t_tuple
9. (1, '23', [123, 'abc'], ('python', 'learn'))

```

tuple用在哪里？

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

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

一般认为，tuple有这类特点，并且也是它使用的情景：

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

[首页](#) | [上一讲](#) | [下一讲](#)

一二三,集合了

- 一二三,集合了
 - 创建set
 - 更改set
 - 增加元素
 - 删除
 - 知识

一二三,集合了

回顾一下已经了解的数据类型:int/str/bool/list/dict/tuple
还真的不少了.

不过,python是一个发展的语言,没准以后还出别的呢.看官可能有疑问了,出了这么多的数据类型,我也记不住呀,特别是里面还有不少方法.

不要担心记不住,你只要记住爱因斯坦说的就好了.

爱因斯坦在美国演讲,有人问:“你可记得声音的速度是多少?你如何记下许多东西?”

爱因斯坦轻松答道:“声音的速度是多少,我必须查辞典才能回答.因为我从来不记在辞典上已经印着的东西,我的记忆力是用来记忆书本上没有的东西。”

多么霸气的回答,这回答不仅仅霸气,更是在告诉我们一种方法:只要能够通过某种方法查找到的,就不需要记忆.

那么,上面那么多数据类型的各种方法,都不需要记忆了,因为它们都可以通过下述方法但不限于这些方法查到(这句话的逻辑还是比较严密的,包括但不限于...)

- 交互模式下用dir()或者help()

- google(不推荐Xdu,原因自己体会啦)

为了能够在总体上对已经学习过的数据类型有了解,我们不妨做如下分类:

1. 是否为序列类型:即该数据的元素是否能够索引.其中序列类型的包括str/list/tuple
2. 是否可以原处修改:即该数据的元素是否能够原处修改(特别提醒看官,这里说的是原处修改问题,有的资料里面说str不能修改,也是指原处修改问题.为了避免误解,特别强调了原处).能够原处修改的是list/dict(特别说明,dict的键必须是不可修改的,dict的值可原处修改)

什么原处修改?看官能不能在交互模式下通过实例解释一下?

到这里,看官可千万不要以为本讲是复习课.本讲的主要内容不是复习,主要内容是要向看官介绍一种新的数据类型:集合(set).彻底晕倒了,到底python有多少个数据类型呢?又多出来了一个.

从基本道理上说,python中的数据类型可以很多,因为每个人都可以自己定义一种数据类型.但是,python官方认可或者说内置的数据类型,就那么几种了.基本上今天的set讲完,就差不多了.在以后的开发过程中,包括今天和以往介绍的数据类型,是常用的.当然,自己定义一个也可以,但是用原生的更好.

创建set

tuple算是list和str的杂合(杂交的都有自己的优势,上一节的末后已经显示了),那么set则可以堪称是list和dict的杂合.

set拥有类似dict的特点:可以用{}花括号来定义;其中的元素没有序列,也就是是非序列类型的数据;而且,set中的元素不可重复,这就类似

dict的键。

set也有继承了一点list的特点:如可以原处修改(事实上是一种类别的set可以原处修改,另外一种不可以)。

下面通过实验,进一步理解创建set的方法:

```

1. >>> s1 = set("qiwsir") #把str中的字符拆解开,形成set.特别注意观察:qiwsir
    中有两个i
2. >>> s1 #但是在s1中,只有一个i,也就是不能重复
3. set(['q', 'i', 's', 'r', 'w'])
4.
5. >>> s2 = set([123, "google", "face", "book", "facebook", "book"]) #通
    过list创建set.不能有重复,元素可以是int/str
6. >>> s2
7. set(['facebook', 123, 'google', 'book', 'face']) #元
    素顺序排列不是按照指定顺序
8.
9. >>> s3 = {"facebook", 123} #通过{}直接创建
10. >>> s3
11. set([123, 'facebook'])

```

再大胆做几个探究,请看官注意观察结果:

```

1. >>> s3 = {"facebook", [1, 2, 'a'],
    {"name": "python", "lang": "english"}, 123}
2. Traceback (most recent call last):
3.   File "<stdin>", line 1, in <module>
4. TypeError: unhashable type: 'dict'
5.
6. >>> s3 = {"facebook", [1, 2], 123}
7. Traceback (most recent call last):
8.   File "<stdin>", line 1, in <module>
9. TypeError: unhashable type: 'list'

```

从上述实验中,可以看出,通过{}无法创建含有list/dict元素的set。

继续探索一个情况：

```

1. >>> s1
2. set(['q', 'i', 's', 'r', 'w'])
3. >>> s1[1] = "I"
4. Traceback (most recent call last):
5.   File "<stdin>", line 1, in <module>
6. TypeError: 'set' object does not support item assignment
7.
8. >>> s1
9. set(['q', 'i', 's', 'r', 'w'])
10. >>> lst = list(s1)
11. >>> lst
12. ['q', 'i', 's', 'r', 'w']
13. >>> lst[1] = "I"
14. >>> lst
15. ['q', 'I', 's', 'r', 'w']

```

上面的探索中,将set和list做了一个对比,虽然说两者都能够做原处修改,但是,通过索引编号(偏移量)的方式,直接修改,list允许,但是set报错.

那么,set如何修改呢?

更改set

还是用前面已经介绍过多次的自学方法,把set的有关内置函数找出来,看看都可以对set做什么操作.

```

1. >>> dir(set)
2. ['__and__', '__class__', '__cmp__', '__contains__', '__delattr__',
   '__doc__', '__eq__', '__format__', '__ge__', '__getattr__',
   '__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']
```

为了看的清楚,我把双划线__开始的先删除掉(后面我们会有专题讲述这些):

```
'add', 'clear', 'copy', 'difference', 'difference_update', 'discard',
'intersection', 'intersection_update', 'isdisjoint', 'issubset',
'issuperset', 'pop', 'remove', 'symmetric_difference',
'symmetric_difference_update', 'union', 'update'
```

然后用help()可以找到每个函数的具体使用方法,下面列几个例子:

增加元素

```
1. >>> help(set.add)
2.
3. Help on method_descriptor:
4.
5. add(...)
6. Add an element to a set.
7. This has no effect if the element is already present.
```

下面在交互模式这个最好的实验室里面做实验:

```
1. >>> a_set = {} #我想当然地认为这样也可以建立一个set
2. >>> 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.
3. Traceback (most recent call last):
4.   File "<stdin>", line 1, in <module>
5. AttributeError: 'dict' object has no attribute 'add'
6. >>> type(a_set) #type之后发现,计算机认为我建立的是一个dict
```

```
7. <type 'dict'>
```

特别说明一下, {}这个东西, 在dict和set中都用. 但是, 如上面的方法建立的是dict, 不是set. 这是python规定的. 要建立set, 只能用前面介绍的方法了.

```
1. >>> a_set = {'a', 'i'}           #这回就是set了吧
2. >>> type(a_set)
3. <type 'set'>                     #果然
4.
5. >>> a_set.add("qiwsir")          #增加一个元素
6. >>> a_set                        #原处修改, 即原来的a_set引用对象已经改变
7. set(['i', 'a', 'qiwsir'])
8.
9. >>> b_set = set("python")
10. >>> type(b_set)
11. <type 'set'>
12. >>> b_set
13. set(['h', 'o', 'n', 'p', 't', 'y'])
14. >>> b_set.add("qiwsir")
15. >>> b_set
16. set(['h', 'o', 'n', 'p', 't', 'qiwsir', 'y'])
17.
18. >>> b_set.add([1,2,3])           #这样做是不行滴, 跟前面一样, 报错.
19. Traceback (most recent call last):
20.   File "<stdin>", line 1, in <module>
21. TypeError: unhashable type: 'list'
22.
23. >>> b_set.add('[1,2,3]')          #可以这样!
24. >>> b_set
25. set(['[1,2,3]', 'h', 'o', 'n', 'p', 't', 'qiwsir', 'y'])
```

除了上面的增加元素方法之外, 还能够从另外一个set中合并过来元素, 方法是set.update(s2)

```
1. >>> help(set.update)
```

```

2. update(...)
3.     Update a set with the union of itself and others.
4.
5. >>> s1
6. set(['a', 'b'])
7. >>> s2
8. set(['github', 'qiwsir'])
9. >>> s1.update(s2)      #把s2的元素并入到s1中.
10. >>> s1                #s1的引用对象修改
11. set(['a', 'qiwsir', 'b', 'github'])
12. >>> s2                #s2的未变
13. set(['github', 'qiwsir'])

```

删除

```

1. >>> help(set.pop)
2. pop(...)
3.     Remove and return an arbitrary set element.
4.     Raises KeyError if the set is empty.
5.
6. >>> b_set
7. set(['[1,2,3]', 'h', 'o', 'n', 'p', 't', 'qiwsir', 'y'])
8. >>> b_set.pop()      #从set中任意选一个删除,并返回该值
9. '[1,2,3]'
10. >>> b_set.pop()
11. 'h'
12. >>> b_set.pop()
13. 'o'
14. >>> b_set
15. set(['n', 'p', 't', 'qiwsir', 'y'])
16.
17. >>> b_set.pop("n")   #如果要指定删除某个元素,报错了.
18. Traceback (most recent call last):
19.   File "<stdin>", line 1, in <module>
20. 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的, `pop()` 不能有参数.此外,如果 `set` 是空的了,也报错.这条是帮助信息告诉我们的,看官可以试试.

要删除指定的元素,怎么办?

```
1. >>> help(set.remove)
2.
3. remove(...)
4.     Remove an element from a set; it must be a member.
5.
6.     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:

```
1. >>> a_set
2. set(['i', 'a', 'qiwsir'])
3. >>> a_set.remove("i")
4. >>> a_set
5. set(['a', 'qiwsir'])
6. >>> a_set.remove("w")
7. Traceback (most recent call last):
8.   File "<stdin>", line 1, in <module>
9.   KeyError: 'w'
```

跟 `remove(obj)` 类似的还有一个 `discard(obj)`:

```
1. >>> help(set.discard)
2.
3. discard(...)
4.     Remove an element from a set if it is a member.
5.
6.     If the element is not a member, do nothing.
```

与`help(set.remove)`的信息对比,看看有什么不同。`discard(obj)`中的`obj`如果是`set`中的元素,就删除,如果不是,就什么也不做,do nothing. 新闻就要对比着看才有意思呢. 这里也一样.

```
1. >>> a_set.discard('a')
2. >>> a_set
3. set(['qiwsir'])
4. >>> a_set.discard('b')
5. >>>
```

在删除上还有一个绝杀,就是`set.clear()`,它的功能是:Remove all elements from this set.(看官自己在交互模式下`help(set.clear)`)

```
1. >>> a_set
2. set(['qiwsir'])
3. >>> a_set.clear()
4. >>> a_set
5. set([])
6. >>> bool(a_set)      #空了,bool一下返回False.
7. 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.

[首页](#) | [上一讲](#) | [下一讲](#)

集合的关系

- 集合的关系
 - 冻结的集合
 - 集合运算
 - 元素与集合的关系
 - 集合与集合的纠结

集合的关系

冻结的集合

前面一节讲述了集合的基本概念，注意，那里所涉及到的集合都是可原处修改的集合。还有一种集合，不能在原处修改。这种集合的创建方法是：

```

1. >>> f_set = frozenset("qiwsir")      #看这个名字就知道了frozen，冻结的
    set
2. >>> f_set
3. frozenset(['q', 'i', 's', 'r', 'w'])
4. >>> f_set.add("python")              #报错
5. Traceback (most recent call last):
6.   File "<stdin>", line 1, in <module>
7. AttributeError: 'frozenset' object has no attribute 'add'
8.
9. >>> a_set = set("github")             #对比看一看，这是一个可以原处修改的
    set
10. >>> a_set
11. set(['b', 'g', 'i', 'h', 'u', 't'])
12. >>> a_set.add("python")
13. >>> a_set
14. set(['b', 'g', 'i', 'h', 'python', 'u', 't'])
  
```


集合运算

先复习一下中学数学（准确说是高中数学中的一点知识）中关于集合的一点知识，主要是唤起那痛苦而青涩美丽的回忆吧，至少对我是。

元素与集合的关系

元素是否属于某个集合。

```
1. >>> aset
2. set(['h', 'o', 'n', 'p', 't', 'y'])
3. >>> "a" in aset
4. False
5. >>> "h" in aset
6. True
```

集合与集合的纠结

假设两个集合A、B

- A是否等于B，即两个集合的元素完全一样

在交互模式下实验

```
1. >>> a
2. set(['q', 'i', 's', 'r', 'w'])
3. >>> b
4. set(['a', 'q', 'i', 'l', 'o'])
5. >>> a == b
6. False
7. >>> a != b
8. 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。

实验一下

```

1. >>> a
2. set(['q', 'i', 's', 'r', 'w'])
3. >>> c
4. set(['q', 'i'])
5. >>> c<a      #c是a的子集
6. True
7. >>> c.issubset(a)    #或者用这种方法，判断c是否是a的子集
8. True
9. >>> a.issuperset(c) #判断a是否是c的超集
10. True
11.
12. >>> b
13. set(['a', 'q', 'i', 'l', 'o'])
14. >>> a<b      #a不是b的子集
15. False
16. >>> a.issubset(b)    #或者这样做
17. False

```

- A、B的并集，即A、B所有元素，如下图所示



```

1. >>> a
2. set(['q', 'i', 's', 'r', 'w'])
3. >>> b
4. set(['a', 'q', 'i', 'l', 'o'])
5. >>> a | b      #可以有两种方式，结果一样
6. set(['a', 'i', 'l', 'o', 'q', 's', 'r', 'w'])
7. >>> a.union(b)
8. set(['a', 'i', 'l', 'o', 'q', 's', 'r', 'w'])

```

- A、B的交集，即A、B所公有的元素，如下图所示



```

1. >>> a
2. set(['q', 'i', 's', 'r', 'w'])
3. >>> b
4. set(['a', 'q', 'i', 'l', 'o'])
5. >>> a & b          #两种方式，等价
6. set(['q', 'i'])
7. >>> a.intersection(b)
8. set(['q', 'i'])

```

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

```

1. >>> a and b
2. set(['a', 'q', 'i', 'l', 'o'])

```

- A相对B的差（补），即A相对B不同的部分元素，如下图所示



```

1. >>> a
2. set(['q', 'i', 's', 'r', 'w'])
3. >>> b
4. set(['a', 'q', 'i', 'l', 'o'])
5. >>> a - b
6. set(['s', 'r', 'w'])
7. >>> a.difference(b)
8. set(['s', 'r', 'w'])

```

-A、B的对称差集，如下图所示



```

1. >>> a
2. set(['q', 'i', 's', 'r', 'w'])
3. >>> b
4. set(['a', 'q', 'i', 'l', 'o'])

```

```
5. >>> a.symmetric_difference(b)
6. set(['a', 'l', 'o', 's', 'r', 'w'])
```

以上是集合的基本运算。在编程中，如果用到，可以用前面说的方法查找。不用死记硬背。

[首页](#) | [上一讲](#) | [下一讲](#)

Python数据类型总结

- Python的数据类型总结
 - 主日崇拜
 - 忘记背后，努力面前，向着标杆直跑

Python的数据类型总结

前面已经洋洋洒洒地介绍了不少数据类型。不能再不顾一切地向前冲了，应当总结一下。这样让看官能够从总体上对这些数据类型有所了解，如果能够有一览众山小的感觉，就太好了。

下面的表格中列出了已经学习过的数据类型，也是python的核心数据类型之一部分，这些都被称之为内置对象。

对象，就是你面对的所有东西都是对象，看官要逐渐熟悉这个称呼。所有的数据类型，就是一种对象。英文单词是*object*，直接的汉语意思是物体，这就好像我们在现实中一样，把很多我们看到和用到的都可以统称为“东西”一样。“东西”就是“对象”，就是*object*。在编程中，那个所谓面向对象，也可以说成“面向东西”，是吗？容易有歧义吧。

对象类型	举例
int/float	123, 3.14
str	'qiwsir.github.io'
list	[1, [2, 'three'], 4]
dict	{'name': "qiwsir", "lang": "python"}
tuple	(1, 2, "three")
set	set("qi"), {"q", "i"}

不论任何类型的数据，只要动用`dir(object)`或者`help(obj)`就能够在交互模式下查看到有关的函数，也就是这样能够查看相关帮助文档了。举例：

```
1. >>> dir(dict)
```

看官需要移动鼠标，就能够看全(下面的本质上就是一个list):

```
1. ['__class__', '__cmp__', '__contains__', '__delattr__',
    '__delitem__', '__doc__', '__eq__', '__format__', '__ge__',
    '__getattr__', '__getitem__', '__gt__', '__hash__',
    '__init__', '__iter__', '__le__', '__len__', '__lt__', '__ne__',
    '__new__', '__reduce__', '__reduce_ex__', '__repr__',
    '__setattr__', '__setitem__', '__sizeof__', '__str__',
    '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'has_key',
    'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop',
    'popitem', 'setdefault', 'update', 'values', 'viewitems',
    'viewkeys', 'viewvalues']
```

先略过__双下划线开头的哪些，看后面的，就是dict的内置函数。至于详细的操作方法，通过类似`help(dict.pop)`的方式获得。这是前面说过的，再说一遍，加深印象。

我的观点：学习，重要的是学习方法，不是按部就班的敲代码。

今天既然是复习，就要在原来基础上提高一点。所以，也要看看上面那些以双下划线_开头的东西，请看官找一下，有没有发现这个：“__doc__”。这是什么，它是一个文件，里面记录了对当前所查看的对象的详细解释。可以在交互模式下这样查看：

```
1. >>> dict.__doc__
```

显示应该是这样的：

```
"dict() -> new empty dictionary\n
dict(mapping) -> new dictionary
initialized from a mapping object's\n
(key, value)
pairs\n
dict(iterable) -> new dictionary
initialized as if via:\n
d = {}\n
for k, v in iterable:\n
d[k] = v\n
dict(**kwargs) -> new
dictionary initialized with the
name=value pairs\n
in the keyword argument list.
For example: dict(one=1, two=2)"
```

注意看上面乱七八糟的英文中，是不是有\n符号，这是什么？前面在讲

述字符串的时候提到了转义符号\，这是换一行。也就是说，如果上面的文字，按照排版要求，应该是这样的（当然，在文本中，如果打开，其实就是排好版的样子）。

```
"dict() -> new empty dictionary
dict(mapping) -> new dictionary initialized from a mapping object's
(key, value) pairs
dict(iterable) -> new dictionary initialized as if via:
d = {}
for k, v in iterable:
d[k] = v
dict(**kwargs) -> new dictionary initialized with the name=value pairs
in the keyword argument list. For example: dict(one=1, two=2)"
```

可能排版还是不符合愿意。不过，看官也大概能看明白了。我要说的不是排版，要说的是告诉看官一种查看某个数据类型含义的方法，就是通过obj.__doc__文件来看。

嘿嘿，其实有一种方法，可以看到排版的结果的：

```
1. >>> print dict.__doc__
2. dict() -> new empty dictionary
3. dict(mapping) -> new dictionary initialized from a mapping object's
4.     (key, value) pairs
5. dict(iterable) -> new dictionary initialized as if via:
6.     d = {}
7.     for k, v in iterable:
8.         d[k] = v
9. dict(**kwargs) -> new dictionary initialized with the name=value
    pairs
10.     in the keyword argument list. For example: dict(one=1, two=2)
```

上面那么折腾一下，就是为了凑篇幅，不然这个总结的东西太少了。

总之，只要用这种方法，你就能得到所有帮助文档，随时随地。如果可以上网，到官方网站，是另外一种方法。

还需要再解释别的吗？都多余了。唯一需要的是看官要能会点英语。不过我相信看官能够读懂，我这个二把刀都不如的英语水平，还能凑合看呢，何况看官呢？

总结不是意味着结束，是意味着继往开来。精彩还在后面，这里只是休息。今天还是周日。

主日崇拜

腓立比書 Philippians (3:13-14)

*Brethren, I count not myself to have apprehended: but this one thing I do, forgetting those things which are behind, and reaching forth unto those things which are before,
I press toward the mark for the prize of the high calling of God in Christ Jesus.*

弟兄們、我不是以為自己已經得著了。我只有一件事、就是忘記背後努力面前的，
向著標竿直跑、要得神在基督耶穌裡從上面召我來得的獎賞。

忘记背后，努力面前，向着标杆直跑

深入变量和引用对象

- 深入变量和引用对象
 - 变量和对象
 - is和==的效果

深入变量和引用对象

今天是2014年8月4日，这段时间灾祸接连发生，显示不久前昆山的工厂爆炸，死伤不少，然后是云南地震，也有死伤。为所有在灾难中受伤的人们献上祷告。

在《永远强大的函数》那一讲中，老齐我(<http://qiwsir.github.io>)已经向看官们简述了一下变量，之后我们就一直在使用变量，每次使用变量，都要有一个操作，就是赋值。本讲再次提及这个两个事情，就是要让看官对变量和赋值有一个知其然和知其所以然的认识。当然，最后能不能达到此目的，主要看我不是说的通俗易懂了。如果您没有明白，就说明我说的还不够好，可以联系我，我再为您效劳。

变量和对象

在《[learning python](#)》那本书里面，作者对变量、对象和引用的关系阐述的非常明了。我这里在很大程度上是受他的启发。感谢作者Mark Lutz先生的巨著。

应用《[learning python](#)》中的一个观点：变量无类型，对象有类型

在python中，如果要使用一个变量，不需要提前声明，只需要在用的时候，给这个变量赋值即可。这里特别强调，只要用一个变量，就要给

这个变量赋值。

所以，像这样是不行的。

```
1. >>> x
2. Traceback (most recent call last):
3.   File "<stdin>", line 1, in <module>
4. NameError: name 'x' is not defined
```

反复提醒：一定要注意看报错信息。如果光光地写一个变量，而没有赋值，那么python认为这个变量没有定义。赋值，不仅仅是给一个非空的值，也可以给一个空值，如下，都是允许的

```
1. >>> x = 3
2. >>> lst = []
3. >>> word = ""
4. >>> my_dict = {}
```

在前面讲述中，我提出了一个类比，就是变量通过一根线，连着对象（具体就可能是一个int/list等），这个类比被很多人接受了，算是我老齐的首创呀。那么，如果要用一种严格的语言来描述，变量可以理解为一个系统表的元素，它拥有过指向对象的命名空间。太严肃了，不好理解，就理解我那个类比吧。变量就是存在系统中的一个东西，这个东西有一种能力，能够用一根线与某对象连接，它能够钓鱼。

对象呢？展开想象。在机器的内存中，系统分配一个空间，这里面就放着所谓的对象，有时候放数字，有时候放字符串。如果放数字，就是int类型，如果放字符串，就是str类型。

接下来的事情，就是前面说的变量用自己所拥有的能力，把对象和自己连接起来（指针连接对象空间），这就是引用。引用完成，就实现了赋值。



看到上面的图了吧，从图中就比较鲜明的表示了变量和对象的关系。所以，严格地将，只有放在内存空间中的对象（也就是数据）才有类型，而变量是没有类型的。这么说如果还没有彻底明白，就再打一个比喻：变量就好比钓鱼的人，湖水里就好像内存，里面有好多鱼，有各种各样的鱼，它们就是对象。钓鱼的人（变量）的任务就是用某种方式（鱼儿引诱）把自己和鱼通过鱼线连接起来。那么，鱼是有类型的，有鲢鱼、鲫鱼、带鱼（带鱼也跑到湖水了了，难道是淡水带鱼？呵呵，就这么扯淡吧，别较真），钓鱼的人（变量）没有这种类型，他钓到不同类型的鱼。

这个比喻太烂了。凑合着理解吧。看官有好的比喻，别忘记分享。

同一个变量可以同时指向两个对象吗？绝对不能脚踩两只船。如果这样呢？

```
1. >>> x = 4
2. >>> x = 5
3. >>> x
4. 5
```

变量x先指向了对象4，然后指向对象5，当后者放生的时候，自动跟第一个对象4接触关系。再看x，引用的对象就是5了。那么4呢？一旦没有变量引用它了，它就变成了孤魂野鬼。python是很吝啬的，它绝对不允许在内存中存在孤魂野鬼。凡是这些东西都被看做垃圾，而对垃圾，python有一个自动的回收机制。

在网上找了一个图示说明，很好，引用过来（来源：

<http://www.linuxidc.com/Linux/2012-09/69523.htm>）

```
1. >>> a = 100          #完成了变量a对内存空间中的对象100的引用
```

如下图所示：



然后，又操作了：

```
1. >>> a = "hello"
```

如下图所示：



原来内存中的那个100就做为垃圾被收集了。而且，这个收集过程是python自动完成的，不用我们操心。

那么，python是怎么进行垃圾收集的呢？在[Quora](#)上也有人问这个问题，我看那个回答很精彩，做个链接，有兴趣的读一读吧。[Python \(programming language\): How does garbage collection in Python work?](#)

is和==的效果

以上过程的原理搞清楚了，下面就可以深入一步了。

```
1. >>> l1 = [1, 2, 3]
2. >>> l2 = l1
```

这个操作中，l1和l2两个变量，引用的是一个对象，都是[1, 2, 3]。何以见得？如果通过l1来修改[1, 2, 3]，l2引用对象也修改了，那么就证实这个观点了。

```
1. >>> l1[0] = 99      #把对象变为[99, 2, 3]
2. >>> l1              #变了
3. [99, 2, 3]
```

```

4. >>> l2                #真的变了吔
5. [99, 2, 3]

```

再换一个方式：

```

1. >>> l1 = [1, 2, 3]
2. >>> l2 = [1, 2, 3]
3. >>> l1[0] = 99
4. >>> l1
5. [99, 2, 3]
6. >>> l2
7. [1, 2, 3]

```

l1和l2貌似指向了同样的一个对象[1, 2, 3]，其实，在内存中，这是两块东西，互不相关。只是在内容上一样。就好像是水里长的一样的两条鱼，两个人都钓到了，当不是同一条。所以，当通过l1修改引用对象的后，l2没有变化。

进一步还能这么检验：

```

1. >>> l1
2. [1, 2, 3]
3. >>> l2
4. [1, 2, 3]
5. >>> l1 == l2          #两个相等，是指内容一样
6. True
7. >>> l1 is l2          #is 是比较两个引用对象在内存中的地址是不是一样
8. False                 #前面的检验已经说明，这是两个东东
9.
10. >>> l3 = l1           #顺便看看如果这样，l3和l1应用同一个对象
11. >>> l3
12. [1, 2, 3]
13. >>> l3 == l1
14. True
15. >>> l3 is l1         #is的结果是True
16. True

```

某些对象，有copy函数，通过这个函数得到的对象，是一个新的还是引用到同一个对象呢？看官也可以做一下类似上面的实验，就晓得了。比如：

```

1. >>> l1
2. [1, 2, 3]
3. >>> l2 = l1[:]
4. >>> l2
5. [1, 2, 3]
6. >>> l1[0] = 22
7. >>> l1
8. [22, 2, 3]
9. >>> l2
10. [1, 2, 3]
11.
12. >>> adict = {"name":"qiwsir","web":"qiwsir.github.io"}
13. >>> bdict = adict.copy()
14. >>> bdict
15. {'web': 'qiwsir.github.io', 'name': 'qiwsir'}
16. >>> adict["email"] = "qiwsir@gmail.com"
17. >>> adict
18. {'web': 'qiwsir.github.io', 'name': 'qiwsir', 'email':
    'qiwsir@gmail.com'}
19. >>> bdict
20. {'web': 'qiwsir.github.io', 'name': 'qiwsir'}

```

不过，看官还有小心有点，python不总按照前面说的方式出牌，比如小数字的时候

```

1. >>> x = 2
2. >>> y = 2
3. >>> x is y
4. True
5. >>> x = 200000
6. >>> y = 200000
7. >>> x is y      #什么道理呀，小数字的时候，就用缓存中的。

```

```
8. False
9.
10. >>> x = 'hello'
11. >>> y = 'hello'
12. >>> x is y
13. True
14. >>> x = "what is you name?"
15. >>> y = "what is you name?"
16. >>> x is y      #不光小的数字，短的字符串也是
17. False
```

赋值是不是简单地就是等号呢？从上面得出来，=的作用就是让变量指针指向某个对象。不过，还可以再深入一些。走着瞧吧。

赋值，简单也不简单

- 赋值，简单也不简单
 - 变量命名
 - 赋值语句
 - 增强赋值

赋值，简单也不简单

变量命名

在《[初识永远强大的函数](#)》一文中，有一节专门讨论“取名字的学问”，就是有关变量名称的问题，本着温故而知新的原则，这里要复习：

名称格式：（下划线或者字母）+（任意数目的字母，数字或下划线）

注意：

1. 区分大小写
 2. 禁止使用保留字
 3. 遵循通常习惯
- 以单一下划线开头的变量名（`_x`）不会被`from module import *`语句导入的。
 - 前后有下划线的变量名（`x`）是系统定义的变量名，对解释器有特殊意义。
 - 以两个下划线开头，但结尾没有两个下划线的变量名（`__x`）是类本地（压缩）变量。
 - 通过交互模式运行时，只有单个下划线变量（`_`）会保存最后的表达式结果。

需要解释一下保留字，就是python里面保留了一些单词，这些单词不能让用户来用作变量名称。都有哪些呢？(python2和python3少有差别，但是总体差不多)

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

需要都记住吗？当然不需要了。一方面，可以在网上随手查到，另外，还能这样：

```
1. >>> not = 3
2.   File "<stdin>", line 1
3.       not = 3
4.           ^
5.   SyntaxError: invalid syntax
6.
7. >>> pass = "hello,world"
8.   File "<stdin>", line 1
9.       pass = "hello,world"
10.          ^
11.  SyntaxError: invalid syntax
```

在交互模式的实验室中，用保留字做变量，就报错了。当然，这时候就要换名字了。

以上原则，是基本原则。在实际编程中，大家通常还这样做，以便让程序更具有可读性：

- 名字具有一定的含义。比如写：n = "qiwsir"，就不如写：name = "qiwsir"更好。
- 名字不要误导别人。比如用account_list指一组账号，就会被人误解为是list类型的数据，事实上可能是也可能不是。所以这时候最好换个名称，比如直接用accounts。

- 名字要有意义的区分，有时候你可能会用到a1, a2之类的名字，最好不要这么做，换个别的方式，通过字面能够看出一定的区分来更好。
- 最好是名称能够读出来，千万别自己造英文单词，也别乱用缩写什么的，特别是贵国的，还喜欢用汉语拼音缩写来做为名字，更麻烦了，还不如全拼呢。最好是用完整的单词或者公认的不会引起歧义的缩写。
- 单个字母和数字就少用了，不仅是显得你太懒惰，还会因为在一段代码中可能有很多个单个的字母和数字，为搜索带来麻烦，别人也更不知道你的i和他理解的i是不是一个含义。

总之，取名字，讲究不少。不论如何，要记住一个标准：明确

赋值语句

对于赋值语句，看官已经不陌生了。任何一个变量，在python中，只要想用它，就要首先赋值。

语句格式：变量名称 = 对象

[上一节](#)中也分析了赋值的本质。

还有一种赋值方式，叫做隐式赋值，通过import、from、del、class、for、函数参数。等模块导入，函数和类的定义，for循环变量以及函数参数都是隐式赋值运算。这方面的东西后面会徐徐道来。

```
1. >>> name = "qiwsir"
2.
3. >>> name, website = "qiwsir", "qiwsir.github.io"    #多个变量，按照顺序依次赋值
4. >>> name
5. 'qiwsir'
6. >>> website
```

```
7. 'qiwsir.github.io'
8.
9. >>> name, website = "qiwsir"    #有几个变量，就对应几个对象，不能少，也不能多
10. Traceback (most recent call last):
11.   File "<stdin>", line 1, in <module>
12. ValueError: too many values to unpack
```

如果这样赋值，也得两边数目一致：

```
1. >>> one,two,three,four = "good"
2. >>> one
3. 'g'
4. >>> two
5. 'o'
6. >>> three
7. 'o'
8. >>> four
9. 'd'
```

这就相当于把good分拆为一个一个的字母，然后对应着赋值给左边的变量。

```
1. >>> [name,site] = ["qiwsir","qiwsir.github.io"]
2. >>> name
3. 'qiwsir'
4. >>> site
5. 'qiwsir.github.io'
6. >>> name,site = ("qiwsir","qiwsir.github.io")
7. >>> name
8. 'qiwsir'
9. >>> site
10. 'qiwsir.github.io'
```

这样也行呀。

其实，赋值的样式不少，核心就是将变量和某对象对应起来。对象，可

以用上面的方式，也可以是这样的

```
1. >>> site = "qiwsir.github.io"
2. >>> name, main = site.split(".")[0], site.split(".")[1]    #还记得
    str.split(<sep>)这个东东吗？忘记了，google一下吧。
3. >>> name
4. 'qiwsir'
5. >>> main
6. 'github'
```

增强赋值

这个东西听名字就是比赋值强的。

在python中，将下列的方式称为增强赋值：

增强赋值语句	等价于语句
<code>x+=y</code>	<code>x = x+y</code>
<code>x-=y</code>	<code>x = x-y</code>
<code>x*=y</code>	<code>x = x*y</code>
<code>x/=y</code>	<code>x = x/y</code>

其它类似结构：`x&=y` `x|=y` `x^=y` `x%=y` `x>>=y` `x<<=y`
`x**=y` `x//=y`

看下面的例子，有一个list，想得到另外一个列表，其中每个数比原来list中的大2。可以用下面方式实现：

```
1. >>> number
2. [1, 2, 3, 4, 5]
3. >>> number2 = []
4. >>> for i in number:
5. ...     i = i+2
6. ...     number2.append(i)
7. ...
```

```
8. >>> number2
9. [3, 4, 5, 6, 7]
```

如果用上面的增强赋值， $i = i+2$ 可以写成 $i += 2$ ，试一试吧：

```
1. >>> number
2. [1, 2, 3, 4, 5]
3. >>> number2 = []
4. >>> for i in number:
5. ...     i += 2
6. ...     number2.append(i)
7. ...
8. >>> number2
9. [3, 4, 5, 6, 7]
```

这就是增强赋值。为什么用增强赋值？因为 $i += 2$ ，比 $i = i+2$ 计算更快，后者右边还要拷贝一个 i 。

上面的例子还能修改，别忘记了list解析的强大功能呀。

```
1. >>> [i+2 for i in number]
2. [3, 4, 5, 6, 7]
```

坑爹的字符编码

- 坑爹的字符编码
 - 编码
 - 计算机中的字符编码
 - encode和decode
 - python中如何避免中文是乱码

坑爹的字符编码

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

```
1. >>> name = '老齐'
2. >>> name
3. '\xe8\x80\x81\xe9\xbd\x90'
```

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

```
1. >>> name = "qiwsir"
2. >>> name
3. 'qiwsir'
```

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

编码

什么是编码？这是一个比较玄乎的问题。也不好下一个普通定义。我看

到有的教材中有定义，不敢说他的定义不对，至少可以说不容易理解。

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



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

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

举一个似乎遥远，其实不久前人们都在使用的东西做例子：[电报](#)

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

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

商务。”天津官电局于庚子遭乱全毁。1887年，台湾巡抚刘铭传敷设了福州至台湾的海底电缆，是中国首条海底电缆。1884年，北京电报开始建设，采用“安设双线，由通州展至京城，以一端引入署中，专递官信，以一端择地安置用便商民”，同年8月5日，电报线路开始建设，所有电线杆一律漆成红色。8月22日，位于北京崇文门外大街西的喜鹊胡同的外城商用电报局开业。同年8月30日，位于崇文门内泡子和以西的吕公堂开局，专门收发官方电报。

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

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

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

电报中的编码被称为**摩尔斯电码**，英文是**Morse Code**

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

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

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

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



我瞪着眼看了老长时间，这两行不是一样的吗？

不管这个了，总之，这就是编码。

计算机中的字符编码

先抄一段[维基百科对字符编码](#)的解释：

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

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

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

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

上面的引文中已经说了，现在我们用的编码标准，已经不是ASCII了，我上大学那时候老师讲的还是ASCII呢（最坑爹的是贵国的大学教育，前几天面试一个大学毕业生，计算机专业的，他告诉我他的老师给他们讲的就是ASCII为编码标准呢，我说你别埋汰老师了，你去看看教材，今天这哥们真给我发短信了，告诉我教材上就是这么说的。），时代变迁，现在已经变成了Unicode了，那么什么是Unicode编码呢？还是抄一段来自[维基百科](#)的说明（需要说明一下，本讲不是我qiwsir在讲，是维基百科在讲，我只是一个配角，哈哈）

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

Unicode伴随着通用字符集的标准而发展，同时也以书本的形式对外发表。Unicode至今仍在

不断增修，每个新版本都加入更多新的字符。目前最新的版本为7.0.0，已收入超过十万个字符（第十万个字符在2005年获采纳）。Unicode涵盖的数据除了视觉上的字形、编码方法、标准的字符编码外，还包含了字符特性，如大小写字母。

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

什么是utf-8，还是看[维基百科](#)上怎么说的吧

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

不再多引用了，如果要看更多，请到[原文](#)。

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

encode和decode

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

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

python2默认的编码是ascii，通过encode可以将对象的编码转换为

指定编码格式，而decode是这个过程的逆过程。

做一个实验，才能理解：

```
1. >>> a = "中"
2. >>> type(a)
3. <type 'str'>
4. >>> a
5. '\xe4\xb8\xad'
6. >>> len(a)
7. 3
8.
9. >>> b = a.decode()
10. >>> b
11. u'\u4e2d'
12. >>> type(b)
13. <type 'unicode'>
14. >>> len(b)
15. 1
```

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

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

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

反过来，一个unicode编码的字符串，也可以转换为字节串。

```
1. >>> c = b.encode('utf-8')
2. >>> c
3. '\xe4\xb8\xad'
4. >>> type(c)
5. <type 'str'>
6. >>> c == a
7. True
```

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

python中如何避免中文是乱码

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

首先，提倡使用utf-8编码方案，因为它跨平台不错。

经验一：在开头声明：

```
1. # -*- 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：

```
1. # 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()

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

```
2. 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)

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

我还收集了网上的一篇文章，也挺好的，推荐给看官：[Python2.x的中文显示方法](#)

最后告诉给我，如果用python3，坑爹的编码问题就不烦恼了。

做一个小游戏

- 做一个小游戏
 - 游戏内容：猜数字游戏
 - 游戏过程描述
 - 分析
 - 随机选择一个数

做一个小游戏

在讲述有关list的时候，提到做游戏的事情，后来这个事情一直没有接续。不是忘记了，是在想在哪个阶段做最合适。经过一段时间学习，看官已经不是纯粹小白了，已经属于python初级者了。现在就是开始做那个游戏的时候了。

游戏内容：猜数字游戏

太简单了吧。是的，游戏难度不大，不过这个游戏中蕴含的东西可是值得玩味的。

游戏过程描述

1. 程序运行起来，随机在某个范围内选择一个整数。
2. 提示用户输入数字，也就是猜程序随机选的那个数字。
3. 程序将用户输入的数字与自己选定的对比，一样则用户完成游戏，否则继续猜。
4. 使用次数少的用户得胜。

分析

在任何形式的程序开发之前，不管是大还是小，都要进行分析。即根据功能需求，将不同功能点进行分解。从而确定开发过程。我们现在做一个很小的程序，也是这样来做。

随机选择一个数

要实现随机选择一个数字，可以使用python中的一个随机函数：`random`。下面对这个函数做简要介绍，除了针对本次应用之外，还扩展点，也许别处看官能用上。

还是要首先强化一种学习方法，就是要学会查看帮助文档。

```

1. >>> import random #这个是必须的，因为不是内置函数
2. >>> dir(random)
3. ['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random',
    'SG_MAGICCONST', 'SystemRandom', 'TWOPI', 'WichmannHill',
    '_BuiltinMethodType', '_MethodType', '__all__', '__builtins__',
    '__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']
4. >>> help(random.randint)
5.
6. Help on method randint in module random:
7.
8. randint(self, a, b) method of random.Random instance
9.     Return random integer in range [a, b], including both end
    points.
```

耐心地看文档，就明白怎么用了。不过，还是把主要的东西列出来，但

仍然建议看官在看每个函数的使用之前，在交互模式下通过help来查看文档。

随机整数：

```
1. >>> import random
2. >>> random.randint(0, 99)
3. 21
```

随机选取0到100间的偶数：

```
1. >>> import random
2. >>> random.randrange(0, 101, 2)
3. 42
```

随机浮点数：

```
1. >>> import random
2. >>> random.random()
3. 0.85415370477785668
4. >>> random.uniform(1, 10)
5. 5.4221167969800881
```

随机字符：

```
1. >>> import random
2. >>> random.choice('qiwsir.github.io')
3. 'g'
```

多个字符中选取特定数量的字符：

```
1. >>> import random
2. random.sample('qiwsir.github.io', 3)
3. ['w', 's', 'b']
```


随机选取字符串：

```
1. >>> import random
2. >>> random.choice ( ['apple', 'pear', 'peach', 'orange', 'lemon'] )
3. 'lemon'
```

洗牌：把原有的顺序打乱，按照随机顺序排列

```
1. >>> import random
2. >>> items = [1, 2, 3, 4, 5, 6]
3. >>> random.shuffle(items)
4. >>> items
5. [3, 2, 5, 6, 4, 1]
```

有点多了。不过，本次实验中，值用到了`random.randint()`即可。多出来是买一送一的（哦。忘记了，没有人买呢，本课程全是白送的）。

关键技术点之一已经突破。可以编程了。再梳理一下流程。画个图展示：

（备注：这里我先懒惰一下吧，看官能不能画出这个程序的流程图呢？特别是如果是一个初学者，流程图一定要自己画哦。刚才看到网上一个朋友说自己学编程，但是逻辑思维差，所以没有学好。其实，画流程图就是帮助提高逻辑思维的一种好方式，请画图吧。）

图画好了，按照直观的理解，下面的代码是一个初学者常常写出来的（老鸟们不要喷，因为是代表初学者的）。

```
1. #!/usr/bin/env python
2. #coding:utf-8
3.
4. import random
5.
```

```
6. number = random.randint(1,100)
7.
8. print "请输入一个100以内的自然数："
9.
10. input_number = raw_input()
11.
12. if number == int(input_number):
13.     print "猜对了, 这个数是："
14.     print number
15. else:
16.     print "错了。"
```

上面的程序已经能够基本走通，但是，还有很多缺陷。

最明显的就是只能让人猜一次，不能多次。怎么修改，能够多次猜呢？动动脑筋之后看代码，或者看官在自己的代码上改改，能不能实现多次猜测？

另外，能不能增强一些友好性呢，让用户知道自己输入的数是大了，还是小了。

根据上述修改想法，新代码如下：

```
1. #!/usr/bin/env python
2. #coding:utf-8
3.
4. import random
5.
6. number = random.randint(1,100)
7.
8. print "请输入一个100以内的自然数："
9.
10. input_number = raw_input()
11.
12. if number == int(input_number):
13.     print "猜对了, 这个数是："
14.     print number
```

```
15. elif number > int(input_number):
16.     print "小了"
17.     input_number = raw_input()
18. elif number < int(input_number):
19.     print "大了"
20.     input_number = raw_input()
21. else:
22.     print "错了。"
```

嗯，似乎比原来进步一点点，因为允许用户输入第二次了。同时也告诉用户输入的是大还是小了。但，这也不行呀。应该能够输入很多次，直到正确为止。

是的。这就要用到一个新的东西：循环。如果看官心急，可以google一下while或者for循环，来进一步完善这个游戏，如果不着急，可以等等，随后我也会讲到这部分。

这个游戏还没有完呢，即使用了循环，后面还会继续。

[首页](#) | [上一讲](#) | [下一讲](#)

不要红头文件(1)

- 不要红头文件(1)
 - 打开文件
 - 创建文件
 - 使用with

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)

不要红头文件(1)

这两天身体不给力，拖欠了每天发讲座的约定，看官见谅。

红头文件，是某国特别色的东西，在python里不需要，python里要处理的是计算机中的文件，包括文本的、图片的、音频的、视频的等等，还有不少没见过的扩展名的，在linux中，不是所有的东西都被保存到文件中吗？文件，在python中，是一种对象，就如同已经学习过的字符串、数字等一样。

先要在交互模式下查看一下文件都有哪些属性：

```
1. >>> dir(file)
2. ['__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']
```

然后对部分属性进行详细说明，就是看官学习了。

打开文件

在某个文件夹下面建立了一个文件，名曰：130.txt，并且在里面输入了如下内容：

```
1. learn python
2. http://qiwsir.github.io
3. qiwsir@gmail.com
```

此文件一共三行。

下图显示了这个文件的存储位置：



在上面截图中，我在当前位置输入了python（我已经设置了环境变量，如果你没有，需要写全启动python命令路径），进入到交互模式。在这个交互模式下，这样操作：

```
1. >>> f = open("130.txt")      #打开已经存在的文件
2. >>> for line in f:
3. ...     print line
4. ...
5. learn python
6.
7. http://qiwsir.github.io
8.
9. qiwsir@gmail.com
```

提醒初学者注意，在那个文件夹输入了启动python交互模式的命令，那么，如果按照上面的方法 `open("130.txt")` 打开文件，就意味着这个文件130.txt是在当前文件夹内的。如果要打开其它文件夹内的文件，

请用相对路径或者绝对路径来表示，从而让python能够找到那个文件。

将打开的文件，赋值个变量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）。

接下来，用for来读取文件中的内容，就如同读取一个前面已经学过的序列对象一样，如list、str、tuple，把读到的文件中的每行，赋值给变量line。也可以理解为，for循环是一行一行地读取文件内容。每次扫描一行，遇到行结束符号\n表示本行结束，然后是下一行。

从打印的结果看出，每一样跟前面看到的文件内容中的每一行是一样的。只是行与行之间多了一空行，前面显示文章内容的时候，没有这个空行。或许这无关紧要，但是，还要深究一下，才能豁然。

在原文中，每行结束有本行结束符号\n，表示换行。在for语句汇总，`print line`表示每次打印完line的对象之后，就换行，也就是打印完line的对象之后会增加一个\n。这样看来，在每行末尾就有两个\n，即：\n\n，于是在打印中就出现了一个空行。

```
1. >>> f = open('130.txt')
2. >>> for line in f:
3. ...     print line,      #后面加一个逗号，就去掉了原来默认增加的\n了，看看，
   ...     少了空行。
4. ...
5. learn python
6. http://qiwsir.github.io
7. qiwsir@gmail.com
```

在进行上述操作的时候，有没有遇到这样的情况呢？

```
1. >>> f = open('130.txt')
2. >>> for line in f:
3. ...     print line,
```

```

4. ...
5. learn python
6. http://qiwsir.github.io
7. qiwsir@gmail.com
8.
9. >>> 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，
10. ...     print line2      #然后打印，结果就什么也显示，这是什么问题？
11. ...
12. >>>

```

如果看官没有遇到上面问题，可以试试。遇到了，这就解惑。不是什么错误，是因为前一次已经读取了文件内容，并且到了文件的末尾了。再重复操作，就是从末尾开始继续读了。当然显示不了什么东西，但是python并不认为这是错误，因为后面就会讲到，或许在这次读取之前，已经又向文件中追加内容了。那么，如果要再次读取怎么办？就重新来一边好了。

特别提醒看官，因为当前的交互模式是在该文件所在目录启动的，所以，就相当于这个实验室和文件130.txt是同一个目录，这时候我们打开文件130.txt，就认为是在本目录中打开，如果文件不是在本目录中，需要写清楚路径。

比如：在上一级目录中

(~/Documents/ITArticles/BasicPython)，加入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。



```

1. >>> f = open("130.txt")
2. Traceback (most recent call last):
3.   File "<stdin>", line 1, in <module>
4. IOError: [Errno 2] No such file or directory: '130.txt'
5.
6. >>> f = open("./codes/130.txt")      #必须得写上路径了（注意，windows的路

```

径是\隔开，需要转义。对转义符，看官看以前讲座)

```

7. >>> for line in f:
8.     ...     print line
9.     ...
10. learn python
11.
12. http://qiwsir.github.io
13.
14. qiwsir@gmail.com
15.
16. >>>

```

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？

```

1. >>> nf = open("131.txt", "w")
2. >>> nf.write("This is a file")

```

就这样创建了一个文件？并写入了文件内容呢？看看再说：



真的就这样创建了新文件，并且里面有那句话呢。

看官注意了没有，这次我们同样是用open()这个函数，但是多了个“w”，这是在告诉python用什么样的模式打开文件。也就是说，用open()打开文件，可以有不同的模式打开。看下表：

模式	描述
r	以读方式打开文件，可读取文件信息。
w	以写方式打开文件，可向文件写入信息。如文件存在，则清空该文件，再写入新内容
a	以追加模式打开文件（即一打开文件，文件指针自动移到文件末尾），如果文件不存在则创建
r+	以读写方式打开文件，可对文件进行读和写操作。

w+	消除文件内容，然后以读写方式打开文件。
a+	以读写方式打开文件，并把文件指针移到文件尾。
b	以二进制模式打开文件，而不是以文本模式。该模式只对Windows或Dos有效，类Unix的文件是用二进制模式进行操作的。

从表中不难看出，不同模式下打开文件，可以进行相关的读写。那么，如果什么模式都不写，像前面那样呢？那样就是默认为r模式，只读的方式打开文件。

```

1. >>> f = open("130.txt")
2. >>> f
3. <open file '130.txt', mode 'r' at 0xb7530230>
4. >>> f = open("130.txt", "r")
5. >>> f
6. <open file '130.txt', mode 'r' at 0xb750a700>

```

可以用这种方式查看当前打开的文件是采用什么模式的，上面显示，两种模式是一样的效果。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

```

1. >>> fp = open("131.txt")
2. >>> for line in fp:          #原来这个文件里面的内容
3. ...     print line
4. ...
5. This is a file
6. >>> 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？
7. >>> fp.write("My name is qiwsir.\nMy website is qiwsir.github.io")
    #再查看内容
8. >>> fp.close()

```

查看文件内容：

```
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
2. My name is qiwsir.
3. 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

```
1. >>> fp = open("131.txt", "a")
2. >>> fp.write("\nAha, I like program\n") #向文件中追加
3. >>> 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：

```
1. $ cat 131.txt
2. My name is qiwsir.
3. My website is qiwsir.github.io
4. Aha, I like program
```

其它项目就不一一讲述了。看官可以自己实验。

本讲先到这里，明天继续文件。感冒药吃了，昏昏欲睡。

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。

```
1. >>> with open("130.txt", "a") as f:
```

```
2. ...     f.write("\nThis is about 'with...as...'")
3. ...
4. >>> with open("130.txt", "r") as f:
5. ...     print f.read()
6. ...
7. learn python
8. http://qiwsir.github.io
9. qiwsir@gmail.com
10. hello
11.
12. This is about 'with...as...'
13. >>>
```

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。

[首页](#) | [上一讲](#) | [下一讲](#)

不要红头文件(2)

- 不要红头文件(2)
 - 文件的属性
 - 文件的有关状态
 - 文件的内置函数

不要红头文件(2)

在前面学习了基本的打开和建立文件之后，就可以对文件进行多种多样的操作了。请看官要注意，文件，不是什么特别的东西，就是一个对象，如同对待此前学习过的字符串、列表等一样。

文件的属性

所谓属性，就是能够通过一个文件对象得到的东西。

```
1. >>> f = open("131.txt", "a")
2. >>> f.name
3. '131.txt'
4. >>> f.mode      #显示当前文件打开的模式
5. 'a'
6. >>> f.closed    #文件是否关闭，如果关闭，返回True；如果打开，返回False
7. False
8. >>> f.close()   #关闭文件的内置函数
9. >>> f.closed
10. True
```

文件的有关状态

很多时候，我们需要获取一个文件的有关状态（有时候成为属性，但是这里的文件属性和上面的文件属性是不一样的，可是，我觉得称之为文

件状态更好一点)，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，能够解决此问题：

```

1. >>> import os
2. >>> file_stat = os.stat("131.txt")      #查看这个文件的状态
3. >>> 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。
4. posix.stat_result(st_mode=33204, st_ino=5772566L, st_dev=2049L,
    st_nlink=1, st_uid=1000, st_gid=1000, st_size=69L,
    st_atime=1407897031, st_mtime=1407734600, st_ctime=1407734600)
5.
6. >>> file_stat.st_ctime                  #这个是文件创建时间
7. 1407734600.0882277                      #换一种方式查看这个时间
8. >>> import time
9. >>> time.localtime(file_stat.st_ctime)  #这回看清楚了。
10. 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)

```

以上关于文件状态和文件属性的内容，在对文件的某些方面进行判断和操作的时候或许会用到。特别是文件属性。比如在操作文件的时候，我们经常要首先判断这个文件是否已经关闭或者打开，就需要用到file.closed这个属性来判断了。

文件的内置函数

```

1. >>> dir(file)
2. ['__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']
3. >>>

```

这么多内置函数，不会都讲述，只能捡着重点的来实验了。

```
1. >>> f = open("131.txt", "r")
2. >>> f.read()
3. 'My name is qiwsir.\nMy website is qiwsir.github.io\nAha,I like
   program\n'
4. >>>
```

`file.read()`能够将文件中的内容全部读取过来。特别注意，这是返回一个字符串，而且是将文件中的内容全部读到内存中。试想，如果内容太多是不是就有点惨了呢？的确是，千万不要去读大个的文件。

```
1. >>> content = f.read()
2. >>> type(content)
3. <type 'str'>
```

如果文件比较大了，就不要一次都读过来，可以转而一行一行地，用 `readline`

```
1. >>> f = open("131.txt", "r")
2. >>> f.readline()          #每次返回一行，然后指针向下移动
3. 'My name is qiwsir.\n'
4. >>> f.readline()          #再读，再返回一行
5. 'My website is qiwsir.github.io\n'
6. >>> f.readline()
7. 'Aha,I like program\n'
8. >>> f.readline()          #已经到最后一行了，再读，不报错，返回空
9. ''
```

这个方法，看官是不是觉得太慢了呢？有没有痛快点的呢？有，请挥刀自宫，不用自宫，也能用 `readlines`。注意区别，这个是复数，言外之意就是多行啦。

```
1. >>> f = open("131.txt", "r")
2. >>> content = f.readlines()
```

```

3. >>> cont
4. ['My name is qiwsir.\n', 'My website is qiwsir.github.io\n', 'Aha,I
   like program\n']
5. >>> type(cont)
6. <type 'list'>
7. >>> for line in cont:
8. ...     print line
9. ...
10. My name is qiwsir.
11.
12. My website is qiwsir.github.io
13.
14. Aha,I like program

```

从实验中我们可以看到，`readlines`和`read`有一样之处，都是将文件内容一次性读出来，存放在内存，但是两者也有区别，`read`返回的是`str`类型，`readlines`返回的是`list`，而且一行一个元素，因此，就可以通过`for`逐行打印出来了。

在`print line`中，注意观察`list`里面的每个元素，最后都是一个`\n`结尾，所以打印的结果会有空行。其原因前面已经介绍过了，忘了的朋友请回滚到[上一讲](#)

不过，还是要提醒列位，太大的文件不用都读到内存中。对付大点的文件，还是推荐这么做：

```

1. >>> f = open("131.txt", "r")
2. >>> f
3. <open file '131.txt', mode 'r' at 0xb757c230>
4. >>> type(f)
5. <type 'file'>
6. >>> for line in f:
7. ...     print line
8. ...
9. My name is qiwsir.
10.

```

```

11. My website is qiwsir.github.io
12.
13. Aha,I like program

```

以上都是读文件的内置函数和方法。除了读，就是要写。所谓写，就是将内容存入到文件中。用到的内置函数是write。但是，要写入文件，还要注意打开文件的模式，可以是w，也可以是a，看具体情况而定。

```

1. >>> f = open("131.txt", "a")      #因为这个文件已经存在，我又不想清空，用追加的模式
2. >>> f.write("There is a baby.")    #这句话应该放到文件最后
3. >>> f.close()                      #请看官注意，写了之后，一定要及时关闭文件。才能代表真正写入

```

看看写的效果：

```

1. >>> f = open("131.txt", "r")
2. >>> for line in f.readlines():
3. ...     print line
4. ...
5. My name is qiwsir.
6.
7. My website is qiwsir.github.io
8.
9. Aha,I like program
10.
11. There is a baby.                #果然增加了这一行

```

以上是关于文件的基本操作。其实对文件远远不知这些，有兴趣的看官可以google一下pickle这个模块，是一个很好用的东西。

第二部分 穷千里目，上一层楼

链接

- [正规地说一句话](#)
- [print能干的事情](#)
- [从格式化表达式到方法](#)
- [复习if语句](#)
- [用while来循环](#)
- [难以想象的for](#)
- [关于循环的小伎俩](#)
- [让人欢喜让人忧的迭代](#)
- [大话题小函数\(1\)](#)
- [大话题小函数\(2\)](#)
- [python文档](#)
- [重回函数](#)
- [变量和参数](#)
- [总结参数的传递](#)
- [传说中的函数条规](#)
- [关于类的基本认识](#)
- [编写类之一创建实例](#)
- [编写类之二方法](#)
- [编写类之三子类](#)
- [编写类之四再论继承](#)
- [命名空间](#)
- [类的细节](#)
- [Import 模块](#)

- [模块的加载](#)
- [私有和专有](#)
- [折腾一下目录](#)

正规地说一句话

- 正规地说一句话
 - 再谈赋值语句
 - 序列赋值

正规地说一句话

小孩子刚刚开始学说话的时候，常常是一个字一个字地开始学，比如学说“饺子”，对他/她来讲，似乎有点难度，大人也聪明，于是就简化了，用“饺饺”来代替，其实就是让孩子学会一个字就能表达。当然，从教育学的角度，有人不赞成这种方法。这个此处不讨论了。如果对比学习编程，就好像是前面已经学习过的那些各种类型的数据（对应这自然语言中的单个字、词），要表达一个完整的意思，或者让计算机完成一个事情（动作），不得不通过一句话，这句话就是语句，它是按照一定规则组织起来的。自然语言中的一句话，按照主谓宾的语法方式组织，计算机编程中的语句，也是按照一定的语法要求进行组织。

虽然在第一部分中，已经零星涉及到语句问题，并且在不同场合也进行了一些应用。毕竟不那么系统。本部分，就比较系统地介绍python中的语句。

为了有总括的印象，先看看python中都包括哪些语句：

- 赋值语句
- if语句，当条件成立时运行语句块。经常与else, elif（相当于else if）配合使用。
- for语句，遍历列表、字符串、字典、集合等迭代器，依次处理迭代器中的每个元素。
- while语句，当条件为真时，循环运行语句块。

- try语句。与except, finally, else配合使用处理在程序运行中出现的异常情况。
- class语句。用于定义类型。
- def语句。用于定义函数和类型的方法。
- pass语句。表示此行为空，不运行任何操作。
- assert语句。用于程序调适阶段时测试运行条件是否满足。
- with语句。Python2.6以后定义的语法，在一个场景中运行语句块。比如，运行语句块前加锁，然后在语句块运行退出后释放锁。
- yield语句。在迭代器函数内使用，用于返回一个元素。
- raise语句。抛出一个异常。
- import语句。导入一个模块或包。常用写法：`from module import name`, `import module as name`, `from module import name as anothername`

特别说明，以上划分也不是很严格，有的内容，有的朋友不认为属于语句。这没关系，反正就是那个东西，在编程中使用。不纠结于名词归类上。总之这些都是要掌握的，才能顺利编程呢。

再谈赋值语句

还记得[赋值，简单也不简单](#)那一讲中所提到的赋值语句吗？既然谈语句，就应该从这个开始，一方面复习，另外一方面，希望能够深点，深点的感觉总是很好的（我说的是理解python，思无邪。前面有一个关于list的内容：[再深点，更懂list](#)，就有喜欢看玩笑的看官思邪了。哈哈。）

```
1. >>> qiwsir = 1
2. >>> python = 2
3. >>> x, y = qiwsir, python    #相当于x=qiwsir,y=python
4. >>> x
5. 1
```

```
6. >>> y
7. 2
8. >>> x, y                                #输出的是tuple
9. (1, 2)
10. >>> [x, y]                             #这就是一个list
11. [1, 2]
12.
13. >>> [a, b] = [qiwsir, python]
14. >>> a
15. 1
16. >>> b
17. 2
18. >>> a, b
19. (1, 2)
20. >>> [a, b]
21. [1, 2]
```

换一种方式，以上两种赋值方法交叉组合一下：

```
1. >>> [c, d] = qiwsir, python
2. >>> c
3. 1
4. >>> d
5. 2
6. >>> c, d
7. (1, 2)
8. >>> f, g = [qiwsir, python]
9. >>> f
10. 1
11. >>> g
12. 2
13. >>> f, g
14. (1, 2)
```

居然也行。其实，从这里我们就看出来了，赋值，就是对应着将左边的变量和右边的对象关联起来。

有这样一个有趣的问题，如果 $a=3$, $b=4$ ，想把这两个变量的值调换一下，也就是 $a=4$, $b=3$ 。在有的高级语言中，是要先引入另外一个变量 c 做为中间变量，就是这样：

```
1. a = 3
2. b = 4
3. c = a    #即c=3
4. a = b    #a=4
5. b = c    #b=3
```

初学者可能有点糊涂。就是我和你两只手都托着一个箱子，现在我们两个要换一下箱子，但是两个手都被占用了，无法换（当然，要求箱子不能落地，也不要放在桌子上之类的）。于是再找一个名曰张三的人来，他空着两只手，那么我先把箱子给张三，我就空出来了，然后接你的箱子，你的箱子就到我手里了。我的那个箱子现在张三手里呢，你接过来，于是我们两个就换了箱子了。

只所以这么啰嗦，就是因为两个没有更多的手。但是，这不是python，python有更多的手。她可以这样：

```
1. >>> qiwsir = 100
2. >>> python = 200
3. >>> qiwsir, python = python, qiwsir
4. >>> qiwsir
5. 200
6. >>> python
7. 100
```

有点神奇，python是三头六臂的。

序列赋值

其实上面实验的赋值，本质上就是序列赋值。只不过这里再强化一番罢

了。如果左边的变量是序列，右边的对象也是序列，两者将一一对应地进行赋值。

```

1. >>> [a, b, c] = (1, 2, 3)    #左右序列一一对应，左边是变量，右边是对象
2. >>> a
3. 1
4. >>> b
5. 2
6. >>> c
7. 3
8. >>> (a,b,c) = [1,2,3]
9. >>> a
10. 1
11. >>> b
12. 2
13. >>> c
14. 3
15. >>> [a,b,c] = "qiw"      #不要忘了了，str也是序列类型的数据
16. >>> a
17. 'q'
18. >>> b
19. 'i'
20. >>> c
21. 'w'
22. >>> (a,b,c) = "qiw"
23. >>> a,c
24. ('q', 'w')
25. >>> a,b,c = 'qiw'        #与前面等价
26. >>> a,b
27. ('q', 'i')
28. >>> a,b = 'qiw'          #报错了，因为左边和右边不是一一对应
29. Traceback (most recent call last):
30.   File "<stdin>", line 1, in <module>
31. ValueError: too many values to unpack
32.
33. >>> (a,b),c = "qi","wei"   #注意观察，这样的像是是如何对应的
34. >>> a,b,c
35. ('q', 'i', 'wei')

```

```
36. >>> string = "qiwsir"
37. >>> a,b,c = string[0],string[1],string[2]    #取切片也一样
38. >>> a,b,c
39. ('q', 'i', 'w')
40. >>> (a,b),c = string[:2],string[2:]
41. >>> a,b,c
42. ('q', 'i', 'wsir')
```

从实验中，可以看出，要搞清楚这种眼花缭乱的赋值，就仅仅扣住“一一对应”这个命脉即可。

如果看官用python3，在赋值上还有更多有意思的东西呢。不过，本讲座用的还是python2。

print能干的事情

- [print能干的事情](#)
 - [eval\(\)](#)
 - [print详解](#)
 - [%r是万能的吗？](#)
 - [再扩展](#)

print能干的事情

print的一些基本用法，在前面的讲述中也涉及一些，本讲是在复习的基础上，尽量再多点内容。

eval()

在print干事情之前，先看看这个东东。不是没有用，因为说不定某些时候要用到。

```

1. >>> help(eval)           #这个是一招鲜，凡是不理解怎么用，就用这个看文档
2.
3. Help on built-in function eval in module __builtin__:
4.
5. eval(...)
6.     eval(source[, globals[, locals]]) -> value
7.
8.     Evaluate the source in the context of globals and locals.
9.     The source may be a string representing a Python expression
10.    or a code object as returned by compile().
11.    The globals must be a dictionary and locals can be any mapping,
12.    defaulting to the current globals and locals.
13.    If only globals is given, locals defaults to it.
```

能看懂更好了，看不懂也没有关系。看我写的吧。哈哈。概括一下，

`eval()`是把字符串中符合python表达式的东西计算出来。意思就是：

```
1. >>> 3+4           #这是一个表达式，python会根据计算法则计算出结果来
2. 7
3. >>> "3+4"         #这是一个字符串，python就不计算里面的内容了，虽然里面是一个
                        符合python规范的表达式
4. '3+4'
5. >>> eval("3+4")    #这里就跟上面不一样了，就把字符串里面的表达式计算出来了
6. 7
```

下面再看一个字符串“相加”的例子：

```
1. >>> "qiwsir"+"github.io"
2. 'qiwsir.github.io'
3. >>> "'qiwsir'+'github.io'"    #字符串里面，python是不会进行“计算”的
4. "'qiwsir'+'github.io'"
5. >>> eval("'qiwsir'+'github.io'") #eval()做的事情完全不一样，它会把字符串
                                     里面的计算出来
6. 'qiwsir.github.io'
```

顺便再说一下另外一个跟`eval()`有点类似的函数：`exec()`，这个函数专门来执行字符串或文件里面的python语句。

```
1. >>> exec "print 'hello, qiwsir'"
2. hello, qiwsir
3. >>> "print 'hello, qiwsir'"
4. "print 'hello, qiwsir'"
```

print详解

`print`命令在编程实践中用的比较多，特别是要向看看程序运行到某个时候产生了什么结果了，必须用`print`来输出，或者说，本讲更宽泛地说，就要说明白把程序中得到的结果输出问题。

比较简单的输出，前面已经涉及到过了：

```
1. >>> name = 'qiwsir'
2. >>> room = 703
3. >>> website = 'qiwsir.github.io'
4. >>> print "MY name is:%s\nMy room is:%d\nMy website is:%s"%
    (name,room,website)
5. MY name is:qiwsir
6. My room is:703
7. My website is:qiwsir.github.io
```

其中，%s,%d就是占位符。

```
1. >>> a = 3.1415926
2. >>> print "%d"%a      #%d只能输出整数,int类型
3. 3
4. >>> print "%f"%a      #%f输出浮点数
5. 3.141593
6. >>> print "%.2f"%a    #按照要求输出小数位数
7. 3.14
8. >>> print "%.9f"%a    #如果要求的小数位数过多，后面就用0补全
9. 3.141592600
10. >>> b = 3
11. >>> print "%4d"%b     #如果是整数，这样写要求该整数占有四个位置，于是在前面增
    加三个空格
12.      3                #而不是写成0003的样式
```

换一种范式，写成这样，就跟上面有点区别了。

```
1. >>> import math        #引入数学模块
2. >>> print "PI=%f"%math.pi #默认，将圆周率打印成这个样子
3. PI=3.141593
4. >>> print "PI=%10.3f"%math.pi #约束一下，这个的含义是整数部分加上小数点和
    小数部分共计10位，并且右对齐
5. PI=      3.142
6. >>> print "PI=%-10.3f"%math.pi #要求显示的左对齐，其余跟上面一样
7. PI=3.142
```

```

8. >>> print "PI=%06d"%int(math.pi) #整数部分的显示, 要求共6位, 这样前面用0补足了。
9. PI=000003

```

其实, 跟对上面数字操作类似, 对字符串也可以做一些约束输出操作。看下面实验, 最好看官也试试。

```

1. >>> website
2. 'qiwsir.github.io'
3. >>> print "%.3s"%website
4. qiw
5. >>> print "%.3s"%(3,website)
6. qiw
7. >>> print "%7.3s"%website
8.      qiw
9. >>> print "%-7.3s"%website
10. qiw

```

总体上, 跟对数字的输出操作类似。不过, 在实际的操作中, 这些用的真的不是很多, 至少在我这么多年的代码生涯中, 用到上面复杂操作的, 就是现在给列位展示的时候, 充其量用一用对float类型的数据输出小数位数的操作, 其它的输出操作, 以默认的那种方式居多。请看官在这里鄙夷我的无知吧。

行文到此, 提醒列位, 如果用python3的, 请用print(), 要加个括号。

print有一个特点, 就是输出的时候, 每行后面都自动加上一个换行符号\n, 这个在前面已经有所提及。

```

1. >>> website
2. 'qiwsir.github.io'
3. >>> for word in website.split("."):
4.     ...     print word
5.     ...

```

```

6. qiwsir
7. github
8. io
9. >>> for word in website.split("."):
10. ...     print word,          #注意，加了一个逗号，输出形式就变化了吧。
11. ...
12. qiwsir github io

```

%r是万能的吗？

我曾经说过，懒人改变世界，特别是在敲代码的领域。于是就有人问了，前面一会儿是%s，一会儿是%d，麻烦，有没有一个万能的？于是网上就有人给出答案了，%r就是万能的。看实验：

```

1. >>> import math
2. >>> print "PI=%r"%math.pi
3. PI=3.141592653589793
4. >>> print "Pi=%r"%int(math.pi)
5. Pi=3

```

真的是万能呀！别着急，看看这个，你是不是就糊涂了？

```

1. >>> print "Pi=%s"%int(math.pi)
2. Pi=3

```

当然，这样就肯定出错了：

```

1. >>> print "p=%d"%pi
2. Traceback (most recent call last):
3.   File "<stdin>", line 1, in <module>
4. TypeError: %d format: a number is required, not str

```

如果看到这里，看官有点糊涂是很正常的，特别是那个号称万能的%r和%s，怎么都能够对原本属于%d的进行正常输出呢？

其实，不管是%r还是%s(%d)都是把做为整数的对象转化为字符串输出了，而不是输出整数。但是%r和%s是有点区别的，本讲对这个暂不做深入研究，只是说明这样的对应：%s→str();%r→repr()，什么意思呢？就是说%s调用的是str()函数把对象转化为str类型，而%r是调用了repr()将对象转化为字符串。关于两者的区别请参考：[Difference between str and repr in Python](#)，下面是一个简单的例子，演示一下两者区别：

```
1. >>> import datetime
2. >>> today = datetime.date.today()
3. >>> today
4. datetime.date(2014, 8, 15)
5. >>> str(today)
6. '2014-08-15'
7. >>> repr(today)
8. 'datetime.date(2014, 8, 15)'
```

最后要表达我的一个观点，没有什么万能的，一切都是根据实际需要而定。

关于更多的输出格式占位符的说明，这个页面中有一个表格，可惜没有找到中文的，如果看官找到中文的，请共享一下呀：[string formatting](#)

再扩展

```
1. >>> myinfo
2. {'website': 'qiwsir.github.io', 'name': 'qiwsir', 'room': 703}
3. >>> print "qiwsir is in %(room)d"%myinfo
4. qiwsir is in 703
```

看官是否看明白上面的输出了？有点意思。这样的输出算是对前面输出

的扩展了。

出了这个扩展之外，在输出的时候，还可以用一个名曰：format的东西，这里面看不到%，但是多了{}。看实验先：

```
1. >>> print "My name is {0} and I am in {1}".format("qiwsir",703)
    #将format后面的内容以此填充
2. My name is qiwsir and I am in 703
3. >>> "My website is {website}".format(website="qiwsir.github.io")
    #{}里面那个相当于一个变量了吧
4. 'My website is qiwsir.github.io'
```

看到这里，是不是感觉这个format有点意思？一点不输给前面的输出方式。据说，format会逐渐逐渐取代前面的。关于format，我计划后面一讲继续。这里只是来一个引子，后面把用format输出搞得多点。

从格式化表达式到方法

- 从格式化表达式到方法
 - 基本的操作
 - 序列对象的偏移量
 - dictionary的键
 - 模板中添加属性
 - 其它进制

从格式化表达式到方法

上一讲，主要介绍了用%表达的一种输出格式化表达式。在那一讲最后又拓展了一点东西，拓展的那点，名曰：格式化方法。因为它知识上是使用了str的**format**方法。

现在我们就格式化方法做一个详细一点的交代。

基本的操作

所谓格式化方法，就是可以先建立一个输出字符串的模板，然后用format来填充模板的内容。

```
1. >>> #先做一个字符串模板
2. >>> template = "My name is {0}. My website is {1}. I am writing
   {2}."
3.
4. >>> #用format依次对应模板中的序号内容
5. >>> template.format("qiwsir", "qiwsir.github.io", "python")
6. 'My name is qiwsir. My website is qiwsir.github.io. I am writing
   python.'
```

当然，上面的操作如果你要这样做，也是可以的：


```

1. >>> "My name is {0}. My website is {1}. I am writing
      {2}.".format("qiwsir", "qiwsir.github.io", "python")
2. 'My name is qiwsir. My website is qiwsir.github.io. I am writing
   python.'
```

这些，跟用%写的表达式没有什么太大的区别。不过看官别着急，一般小孩子都区别不到，长大了才有区别的。慢慢看，慢慢实验。

除了可以按照对应顺序（类似占位符了）填充模板中的位置之外，还能这样，用关键字来指明所应该填中的内容。

```

1. >>> template = "My name is {name}. My website is {site}"
2. >>> template.format(site='qiwsir.github.io', name='qiwsir')
3. 'My name is qiwsir. My website is qiwsir.github.io'
```

关键词所指定的内容，也不一定非是str，其它的数据类型也可以。此外，关键词和前面的位置编号，还可以混用。比如：

```

1. >>> "{number} is in {all}. {0} are my
      number.".format("seven", number=7, all=[1, 2, 3, 4, 5, 6, 7, 8, 9, 0])
2. '7 is in [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]. seven are my number.'
```

是不是开始感觉有点意思了？看输出结果，就知道，经过format方法得到是一个新的str。

序列对象的偏移量

有这样一个要求：在输出中，显示出一个单词的第一个字母和第三个字母。比如单词python，要告诉看官，第一字母是p，第三个字母是t。

这个问题并不难。实现方法也不少，这里主要是要展示一下偏移量在format中的应用。

```

1. >>> template = "First={0[0]}, Third={0[2]}"
2. >>> template.format(word)
3. 'First=p, Third=t'

```

list也是序列类型的，其偏移量也可。

```

1. >>> word_lst = list(word)
2. >>> word_lst
3. ['p', 'y', 't', 'h', 'o', 'n']
4. >>> template
5. 'First={0[0]}, Third={0[2]}'
6. >>> template.format(word_lst)
7. 'First=p, Third=t'

```

对上面的综合一下，稍微啰嗦一点的实验：

```

1. >>> template = "The word is {0}, Its first is {0[0]}. Another word
    is {1}, Its second is {1[1]}."
2. >>> template.format("python", "learn")
3. 'The word is python, Its first is p. Another word is learn, Its
    second is e.'
4.
5. >>> "{name}\n's first is {name[0]}".format(name="qiwsir")    #指定关
    键词的值的偏移量
6. "qiwsir's first is q"

```

值得注意的是，偏移量在序列类型的数据中，因为可以是负数，即能够从右边开始计数。

```

1. >>> word
2. 'python'
3. >>> word[-1]
4. 'n'
5. >>> word[-2]
6. 'o'

```

但是，在模板中，无法使用负数的偏移量。

```
1. >>> "First={0[0]}, End={0[-1]}".format(word) #报错
2. Traceback (most recent call last):
3.   File "<stdin>", line 1, in <module>
4.   TypeError: string indices must be integers, not str
5.
6. >>> "First={0[0]}, End={0[5]}".format(word) #把-1改为5就可以了。
7. 'First=p, End=n'
```

当然，放到模板外面是完全可行的。这样就好了：

```
1. >>> "First={0}, End={1}".format(word[0],word[-1])
2. 'First=p, End=n'
```

dictionary的键

直接上实验，先观察，再得结论

```
1. >>> myinfo
2. {'website': 'qiwsir.github.io', 'name': 'qiwsir', 'room': 703}
3. >>> template = "I am {0[name]}"
4. >>> template.format(myinfo)
5. 'I am qiwsir'
6. >>> template = "I am {0[name]}. My QQ is {qq}"
7. >>> template.format(myinfo, qq="26066913")
8. 'I am qiwsir. My QQ is 26066913'
```

位置后面跟键，就能得到format的参数中字典的键对应的值。太罗嗦了吧，看例子就明白了。出了根据位置得到，还能够根据关键词得到：

```
1. >>> myinfo
2. {'website': 'qiwsir.github.io', 'name': 'qiwsir', 'room': 703}
3. >>> "my website is {info[website]}, and I like
    {0}".format("python", info=myinfo) #关键词info引用的是一个字典
```

```
4. 'my website is qiwsir.github.io, and I like python'
```

模板中添加属性

看标题不懂在说什么。那就看实验吧。

```
1. >>> import math
2. >>> "PI is {PI.pi}".format(PI=math)
3. 'PI is 3.14159265359'
```

这是用关键词，下面换个稍微复杂点，用位置的。

```
1. >>> import sys, math
2. >>> 'PI is {0.pi}. My laptop runs {1.platform}'.format(math, sys)
3. 'PI is 3.14159265359. My laptop runs linux2'
```

看官理解了吧。

其它进制

在这个世界上的数学领域，除了有我们常常用到的十进制、十二进制（几点了，这是你我常用到的，钟表面就是12进制）、六十进制（这个你也熟悉的）外，还有别的进制，比如二进制、八进制、十六进制等等。此处不谈进制问题，有兴趣详细了解，请各自google。不过，进制的确在计算机上很重要的。因为机器在最底层是用二进制的。

这里只是说明一下输出时候的进制问题。

```
1. >>> "{0:X}, {1:o}, {2:b}".format(255, 255, 255)
2. 'FF, 377, 11111111'
```

- X: 十六进制, Hex
- o: 八进制, octal

- b:二进制, binary

顺便补充，对于数的格式化方法输出和格式化表达式一样，就不赘述了。

在格式化方法中，还能够指定字符宽度，左右对齐等简单排版格式，不过，在我的经验中，这些似乎用的不那么多。如果看官需要，可以google或者到官方文档看看即可。

关于格式化表达式和格式化方法，有的人进行了不少比较，有的人说用这个，有的人倾向用那个。我的建议是，你用哪个顺手就用哪个。切忌门派之见呀。不过，有人传说格式化表达式可能在将来某个版本中废除。那是将来的事情，将来再说好了。现在，你就捡着顺手的用吧。

复习if语句

- 复习if语句
 - 基本语句结构
 - 拉出来溜溜
 - 一个有趣的赋值-三元操作符

复习if语句

看官是否记得，在上一部分的时候，有一讲专门介绍if语句的：[从if开始语句的征程](#)。在学习if语句的时候，对python编程的基础知识了解的还不是很多，或许没有做什么太复杂的东西。本讲，要对它进行一番复习，通过复习提高一下。如果此前有的东西忘记了，建议首先回头，看看前面那讲。

基本语句结构

```
1.  if 判断条件1:
2.     执行语句1.....
3.  elif 判断条件2:
4.     执行语句2.....
5.  elif 判断条件3:
6.     执行语句3.....
7.  else:
8.     执行语句4.....
```

只有当“判断条件”的值是True的时候，才执行下面的执行语句。

那么，在python中，怎么知道一个判断条件是不是真呢？这个问题我们在[眼花缭乱的运算符](#)中已经讲解了一种数据类型：布尔类型。可以通过一个内置函数bool()来判断一个条件的结果True还是False。看看

下面的例子，是不是能够理解bool()的判断规则？

```
1. >>> bool("")
2. False
3. >>> bool(0)
4. False
5. >>> bool('none')
6. True
7. >>> bool(False)
8. False
9. >>> bool("False")
10. True
11. >>> bool(True)
12. True
13. >>> bool("True")
14. True
15. >>> bool(3>4)
16. False
17. >>> bool("b">"a")
18. True
19. >>> bool(not "")
20. True
21. >>> bool(not True)
22. False
```

忘记了怎么办？看下面的语句：

1. if 忘记：
2. 复习-->眼花缭乱的运算符一讲

在执行语句中，其实不一定非要把bool()写上的。如同这样：

```
1. >>> x = 9
2.
3. >>> if bool(x>7):        #条件为True则执行下面的
4. ...        print "%d more than 7"%x
5. ... else:
```

```

6. ...     print "%d not more than 7"%x
7. ...
8. 9 more than 7
9.
10. >>> if x>7:
11. ...     print "%d more than 7"%x
12. ... else:
13. ...     print "%d not more than 7"%x
14. ...
15. 9 more than 7

```

以上两个写法是等效的，但是，在实际的编程中，我们不用if bool(x>7)的格式，而是使用if x>7的样式，还要特别提醒，如果写成if (x>7)，用一个括号把条件表达式括起来，是不是可以呢？可以，但也不是python提倡的。

```

1. >>> if (x>7):           #不提倡这么写，这不是python风格
2. ...     print "%d more than 7"%x
3. ...
4. 9 more than 7

```

拉出来溜溜

平时总有人在不服气的时候说“是骡子是马，拉出来溜溜”，赵本山有一句名言“走两步”。其本质都是说“光说不练是假把式”。今天收到一个朋友的邮件，也询问，在学习python的时候，记不住python的内容。其实不用记，我在前面的课程中已经反复讲过了。但是，在应用中，会越来越熟练。

下面就做一个练习，要求是：

1. 接收任何字符和数字的输入
2. 判断输入的内容，如果不是整数是字符，就告诉给用户；如果是小

数，也告诉用户

3. 如果输入的是整数，判断这个整数是奇数还是偶数，并且告诉给用户

在这个练习中，显然要对输入的内容进行判断，以下几点需要看官注意：

- 通过`raw_input()`得到的输入内容，都是`str`类型
- 要判断一个字符串是否是由纯粹数字组成，可以使用`str.isdigit()`（建议看官查看该内置函数官方文档）

下面的代码是一个参考：

```
1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  print "请输入字符串,然后按下回车键:"
5.
6.  user_input = raw_input()
7.
8.  result = user_input.isdigit()
9.
10. if not result:
11.     print "您输入的不完全是数字"
12.
13. elif int(user_input)%2==0:
14.     print "您输入的是一个偶数"
15. elif int(user_input)%2!=0:
16.     print "您输入的是一个奇数"
17. else:
18.     print "您没有输入什么呢吧"
```

特别提醒列为，这个代码不是非常完善的，还有能够修改的地方，看官能否完善之？

再来一个如何？

已知一个由整数构成的list，从中跳出奇数和偶数，并且各放在一个list中。

请看官在看下面的参考代码之前，自己写一写。

```
1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  import random
5.
6.  numbers = [random.randint(1,100) for i in range(20)] #以list解析的方式得到随机的list
7.
8.  odd = []
9.  even = []
10.
11. for x in numbers:
12.     if x%2==0:
13.         even.append(x)
14.     else:
15.         odd.append(x)
16.
17. print numbers
18. print "odd:",odd
19. print "even:",even
```

用这个例子演示一下if在list解析中的应用。看能不能继续改进一些呢？

可以将循环的那部分用下面的list解析代替

```
1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  import random
```

```

5.
6. numbers = [random.randint(1,100) for i in range(20)] #以list解析的方式得到随机的list
7.
8. odd = [x for x in numbers if x%2!=0]
9. even = [x for x in numbers if x%2==0]
10.
11. print numbers
12. print "odd:", odd
13. print "even:", even

```

一个有趣的赋值—三元操作符

对赋值，看官应该比较熟悉了吧，如果要复习，请看《赋值，简单也不简单》以及《正规地说一句》的相关内容。

这里说的有趣赋值是什么样子的呢？请看：

```

1. >>> name = "qiwsir" if "laoqi" else "github"
2. >>> name
3. 'qiwsir'
4. >>> name = 'qiwsir' if "" else "python"
5. >>> name
6. 'python'
7. >>> name = "qiwsir" if "github" else ""
8. >>> name
9. 'qiwsir'

```

总结一下：A = Y if X else Z

什么意思，结合前面的例子，可以看出：

- 如果X为真，那么就执行A=Y
- 如果X为假，就执行A=Z

||| x = 2

```
y = 8
a = "python" if x>y else "qiwsir"
a
'qiwsir'
b = "python" if x<y else "qiwsir"
b
'python'
```

再看看上面的例子，是不是这样执行呢？

if语句似乎简单，但是在编程时间中常用到。勤加练习吧。

用while来循环

- 用while来循环
 - 再做猜数字游戏
 - break和continue
 - while...else

用while来循环

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 somthing, 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:

1. while 年龄大于60岁 : ----->当年龄大于60岁的时候
2. 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们, 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:



再做猜数字游戏

本教程有一讲, 是跟看官一同[做一个小游戏](#), 在里面做了一个猜数的游戏, 当时遇到了一个问题, 就是只能猜一两次, 如果猜不到, 程序就不

能继续运行了。

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，单元李航同学不要见怪，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。

```

1.  #!/usr/bin/env python
2.  #coding:UTF-8
3.
4.  import random
5.
6.  i=0
7.  while i < 4:
8.      print'*****'
9.      num = input('请您输入0到9任一个数：')      #李同学用的是python3
10.
11.      xnum = random.randint(0,9)
12.
13.      x = 3 - i
14.
15.      if num == xnum:
16.          print'运气真好，您猜对了！'
17.          break
18.      elif num > xnum:
19.          print'''您猜大了！\n哈哈，正确答案是：%s\n您还有%s次机会！''' %
20.              (xnum,x)
21.      elif num < xnum:
22.          print'''您猜小了！\n哈哈，正确答案是：%s\n您还有%s次机会！''' %
23.              (xnum,x)
24.          print'*****'
25.
26.      i += 1

```

我们就用这段程序来分析一下，首先看while `i<4`，这是程序中为猜测限制了次数，最大是三次，请看官注意，在while的循环体中的最后一句：`i +=1`，这就是说每次循环到最后，就给i增加1，当`bool(i<4)=False`的时候，就不再循环了。

当`bool(i<4)=True`的时候，就执行循环体内的语句。在循环体内，让用户输入一个整数，然后程序随机选择一个整数，最后判断随机生成的数和用户输入的数是否相等，并且用if语句判断三种不同情况。

根据上述代码，看官看看是否可以修改？

为了让用户的体验更爽，不妨把输入的整数范围扩大，在1到100之间吧。

```
1. num_input = raw_input("please input one integer that is in 1 to
    100:")      #我用的是python2.7，在输入指令上区别于李同学
```

程序用num_input变量接收了输入的内容。但是，请列位看官一定要注意，看到这里想睡觉的要打起精神了，我要分享一个多年编程经验，请牢记：任何用户输入的内容都是不可靠的。这句话含义深刻，但是，这里不做过多的解释，需要各位在随后的编程生涯中体验了。为此，我们要检验用户输入的是否符合我们的要求，我们要求用户输入的是1到100之间的整数，那么就要做如下检验：

1. 输入的是否是整数
2. 如果是整数，是否在1到100之间。

为此，要做：

```
1. if not num_input.isdigit():      #str.isdigit()是用来判断字符串是否纯粹
    由数字组成
2.     print "Please input interger."
3. elif int(num_input)<0 and int(num_input)>=100:
```

```

4.     print "The number should be in 1 to 100."
5. else:
6.     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。

```

1. import random
2.
3. number = random.randint(1,100)
4.
5. while True:          #不限制用户的次数了
6.     ...

```

观察李同学的程序，还有一点需要向列位显明的，那就是在条件表达式中，两边最好是同种类型数据，上面的程序中有：num>xnum样式的条件表达式，而一边是程序生成的int类型数据，一边是通过输入函数得到的str类型数据。在某些情况下可以运行，为什么？看官能理解吗？都是数字的时候，是可以的。但是，这样不好。

那么，按照这种思路，把这个猜数字程序重写一下：

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  import random
5.
6.  number = random.randint(1,101)
7.
8.  guess = 0
9.
10. while True:

```



```

11.
12.     num_input = raw_input("please input one integer that is in 1 to
    100:")
13.     guess +=1
14.
15.     if not num_input.isdigit():
16.         print "Please input interger."
17.     elif int(num_input)<0 or int(num_input)>=100:
18.         print "The number should be in 1 to 100."
19.     else:
20.         if number==int(num_input):
21.             print "OK, you are good.It is only %d, then you
    succeeded."%guess
22.             break
23.         elif number>int(num_input):
24.             print "your number is more less."
25.         elif number<int(num_input):
26.             print "your number is bigger."
27.         else:
28.             print "There is something bad, I will not work"

```

以上供参考，看官还可改进。

break和continue

break, 在上面的例子中已经出现了，其含义就是要在这个地方中断循环，跳出循环体。下面这个简要的例子更明显：

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  a = 8
5.  while a:
6.      if a%2==0:
7.          break
8.      else:
9.          print "%d is odd number"%a

```

```

10.         a = 0
11. print "%d is even number"%a

```

a=8的时候，执行循环体中的break，跳出循环，执行最后的打印语句，得到结果：

```

1. 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：

```

1. 9 is odd number
2. 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。

```

1. #!/usr/bin/env python
2. #coding:utf-8
3.
4. a = 9
5. while a:
6.     if a%2==0:
7.         a -=1
8.         continue    #如果是偶数，就返回循环的开始
9.     else:
10.        print "%d is odd number"%a #如果是奇数，就打印出来
11.        a -=1

```

其实，对于这两东西，我个人在编程中很少用到。我有一个固执的观念，尽量将条件在循环之前做足，不要在循环中跳来跳去，不仅可读性下降，有时候自己也糊涂了。

while...else

这两个的配合有点类似if ... else，只需要一个例子列出来就理解了，当然，一遇到else了，就意味着已经不在while循环内了。

```
1. #!/usr/bin/env python
2.
3. count = 0
4. while count < 5:
5.     print count, " is less than 5"
6.     count = count + 1
7. else:
8.     print count, " is not less than 5"
```

执行结果：

```
1. 0 is less than 5
2. 1 is less than 5
3. 2 is less than 5
4. 3 is less than 5
5. 4 is less than 5
6. 5 is not less than 5
```

难以想象的for

- 难以想象的for
 - for的基本操作
 - zip

难以想象的for

看这个标题，有点匪夷所思吗？为什么for是难以想象的呢？因为在python中，它的确是很常用而且很强悍，强悍到以至于另外一个被称之为迭代的東西，在python中就有点相形见绌了。在别的语言中，for的地位从来没有如同python中这么高的。

废话少说，上干活。

for的基本操作

for是用来循环的，是从某个对象那里依次将元素读取出来。看下面的例子，将已经学习过的数据对象用for循环一下，看看哪些能够使用，哪些不能使用。同时也是复习一下过往的内容。

```
1. >>> name_str = "qiwsir"
2. >>> for i in name_str: #可以对str使用for循环
3. ...     print i,
4. ...
5. q i w s i r
6.
7. >>> name_list = list(name_str)
8. >>> name_list
9. ['q', 'i', 'w', 's', 'i', 'r']
10. >>> for i in name_list: #对list也能用
11. ...     print i,
12. ...
```

```

13. q i w s i r
14.
15. >>> name_set = set(name_str)    #set还可以用
16. >>> name_set
17. set(['q', 'i', 's', 'r', 'w'])
18. >>> for i in name_set:
19. ...     print i,
20. ...
21. q i s r w
22.
23. >>> name_tuple = tuple(name_str)
24. >>> name_tuple
25. ('q', 'i', 'w', 's', 'i', 'r')
26. >>> for i in name_tuple:        #tuple也能呀
27. ...     print i,
28. ...
29. q i w s i r
30.
31. >>> name_dict=
    {"name":"qiwsir","lang":"python","website":"qiwsir.github.io"}
32. >>> for i in name_dict:          #dict也不例外
33. ...     print i,"-->",name_dict[i]
34. ...
35. lang --> python
36. website --> qiwsir.github.io
37. name --> qiwsir

```

除了上面的数据类型之外，对文件也能够用for，这在前面有专门的《[不要红头文件](#)》两篇文章讲解有关如何用for来读取文件对象的内容。看官若忘记了，可去浏览。

for在list解析中，用途也不可小觑，这在讲解list解析的时候，也已说明，不过，还是再复习一下为好，所谓学而时常复习之，不亦哈哈乎。

```

1. >>> one = range(1,9)

```

```

2. >>> one
3. [1, 2, 3, 4, 5, 6, 7, 8]
4. >>> [ x for x in one if x%2==0 ]
5. [2, 4, 6, 8]

```

什么也不说了，list解析的强悍，在以后的学习中会越来越体会到的，佩服佩服呀。

列位如果用python3，会发现字典解析、元组解析也是奇妙的呀。

要上升一个档次，就得进行概括。将上面所说的for循环，概括一下，就是下图所示：



用一个文字表述：

```

1. for iterating_var in sequence:
2.     statements

```

iterating_var是对象sequence的迭代变量，也就是sequence必须是一个能够有某种序列的对象，特别注意没某种序列，就是说能够按照一定的脚标获取元素。当然，文件对象属于序列，我们没有用脚标去获取每行，如果把它读取出来，因为也是一个str,所以依然可以用脚标读取其内容。

zip

zip是什么东西？在交互模式下用help(zip),得到官方文档是：

```

zip(...)
zip(seq1 [, seq2 [...]]) -> [(seq1[0], seq2[0] ...), (...)]

Return a list of tuples, where each tuple contains the i-th element
from each of the argument sequences. The returned list is truncated in

```

length to the length of the shortest argument sequence.

通过实验来理解上面的文档：

```

1. >>> a = "qiwsir"
2. >>> b = "github"
3. >>> zip(a,b)
4. [('q', 'g'), ('i', 'i'), ('w', 't'), ('s', 'h'), ('i', 'u'), ('r',
    'b')]
5. >>> c = [1,2,3]
6. >>> d = [9,8,7,6]
7. >>> zip(c,d)
8. [(1, 9), (2, 8), (3, 7)]
9. >>> e = (1,2,3)
10. >>> f = (9,8)
11. >>> zip(e,f)
12. [(1, 9), (2, 8)]
13. >>> m = {"name","lang"}
14. >>> n = {"qiwsir","python"}
15. >>> zip(m,n)
16. [('lang', 'python'), ('name', 'qiwsir')]
17. >>> s = {"name":"qiwsir"}
18. >>> t = {"lang":"python"}
19. >>> zip(s,t)
20. [('name', 'lang')]

```

zip是一个内置函数，它的参数必须是某种序列数据类型，如果是字典，那么键视为序列。然后将序列对应的元素依次组成元组，做为一个list的元素。

下面是比较特殊的情况，参数是一个序列数据的时候，生成的结果样子：

```

1. >>> a
2. 'qiwsir'
3. >>> c
4. [1, 2, 3]

```

```

5. >>> zip(c)
6. [(1,), (2,), (3,)]
7. >>> zip(a)
8. [('q',), ('i',), ('w',), ('s',), ('i',), ('r',)]

```

这个函数和for连用，就是实现了：

```

1. >>> c
2. [1, 2, 3]
3. >>> d
4. [9, 8, 7, 6]
5. >>> for x,y in zip(c,d):      #实现一对一对地打印
6. ...     print x,y
7. ...
8. 1 9
9. 2 8
10. 3 7
11. >>> for x,y in zip(c,d):     #把两个list中的对应量上下相加。
12. ...     print x+y
13. ...
14. 10
15. 10
16. 10

```

上面这个相加的功能，如果不用zip，还可以这么写：

```

1. >>> length = len(c) if len(c)<len(d) else len(d)      #判断c,d的长度,
   ...     将短的长度拿出来
2. >>> for i in range(length):
3. ...     print c[i]+d[i]
4. ...
5. 10
6. 10
7. 10

```

以上两种写法那个更好呢？前者？后者？哈哈。我看差不多了。还可以这么做呢：


```
1. >>> [ x+y for x,y in zip(c,d) ]  
2. [10, 10, 10]
```

前面多次说了，list解析强悍呀。当然，还可以这样的：

```
1. >>> [ c[i]+d[i] for i in range(length) ]  
2. [10, 10, 10]
```

for循环语句在后面还会经常用到，其实前面已经用了很多了。所以，看官应该不感到太陌生。

关于循环的小伎俩

- 关于循环的小伎俩
 - `range`
 - `zip`
 - `enumerate`

关于循环的小伎俩

不管是while还是for，所发起的循环，在python编程中是经常被用到的。特别是for，一般认为，它要比while快，而且也容易写（是否容易，可能因人而异，但是，执行时间快，是的确的），因此在实践中，for用的比较多点，不是说while就不用，比如前面所列举而得那个猜数字游戏，在业务逻辑上，用while就更容易理解（当然是限于那个游戏的业务需要而言）。另外，在某些情况下，for也不是简单地把对象中的元素遍历一遍，比如有有隔一个取一个的要求，等等。

在编写代码的实践中，为了对付循环中的某些要求，需要用一些其它的函数，比如前面已经介绍过的range就是一个被看做循环中的计数器的好东西。

range

在《[有容乃大的list\(4\)](#)》中，专门对range()这个内置函数做了详细介绍，看官可以回到那节教程复习一番。这里重点是复习并展示一下它的for循环中，做为计数器的使用。

还记得曾经在教程中有一个问题：[列出100以内被3整除的数](#)。下面引用那个问题的代码和运行结果。

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  aliquot = []
5.
6.  for n in range(1,100):
7.      if n%3 == 0:
8.          aliquot.append(n)
9.
10. print aliquot

```

代码运行结果：

```

1.  [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]

```

这个问题，如果改写一下（也有网友在博客中提出了改写方法）

```

1.  >>> aliquot = [ x for x in range(1,100) if x%3==0 ] #用list解析，本质
    上跟上面无太大差异
2.  >>> aliquot
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]
4.
5.  >>> aliquot = range(3,100,3) #这种方法更简单。这是博客中一网友提供。
6.  >>> aliquot
7.  [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]

```

如果有一个由字母组成的字符串，只想隔一个从字符串中取一个字母。可以这样来实现，这是`range()`的一个重要用途。

```

1.  >>> one = "Ilikepython"
2.  >>> new_list = [ one[i] for i in range(0,len(one),2) ]
3.  >>> new_list
4.  ['I', 'i', 'e', 'y', 'h', 'n']

```

当然，间隔的举例，是可以任意指定的。还是前面那个问题，还可以通过下面的方式，选出所有能够被3整除的数。

```
1. >>> all_int = range(1,100)
2. >>> all_int
3. [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, 32, 33, 34, 35, 36,
    37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53,
    54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
    71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87,
    88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
4. >>> aliquot = [ all_int[i] for i in range(len(all_int)) if
    all_int[i]%3==0 ]
5. >>> aliquot
6. [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]
```

通过上述实例，主要是让看官理解range()在for循环中计数器的作用。

zip

在《难以想象的for》中，已经对zip进行了介绍，此处还要提到这个函数，不仅仅是复习，还能深入一下，更主要是它也会常常被用到循环之中。

zip是用于并行遍历的函数。

比如有两个list，元素是由整数组成，如果计算对应位置元素的和。一种方法是通过循环，分别从两个list中取出元素，然后求和。

```
1. >>> list1 = range(2,10,2)
2. >>> list1
3. [2, 4, 6, 8]
4. >>> list2 = range(11,20,2)
```

```

5. >>> list2
6. [11, 13, 15, 17, 19]
7. >>> result = [ list1[i]+list2[i] for i in range(len(list1)) ]
8. >>> result
9. [13, 17, 21, 25]

```

正如在《难以想象的for》中讲述的那样，上面的方法不是很完美，在上一讲中有比较完美一点的代码，请看官欣赏。

zip完成上面的任务，是这么做的：

```

1. >>> list1
2. [2, 4, 6, 8]
3. >>> list2
4. [11, 13, 15, 17, 19]
5. >>> for a,b in zip(list1,list2):
6. ...     print a+b,
7. ...
8. 13 17 21 25

```

zip()的作用就是把list1和list2两个对象中的对应元素放到一个元组(a,b)中，然后对这两个元素进行操作。

```

1. >>> list1
2. [2, 4, 6, 8]
3. >>> list2
4. [11, 13, 15, 17, 19]
5. >>> zip(list1,list2)
6. [(2, 11), (4, 13), (6, 15), (8, 17)]

```

对这个功能，看官可以理解为，将两个list压缩成为(zip)一个list，只不过找不到配对的就丢掉了。

能够压缩，也能够解压缩，用下面的方式就是反过来了。

```

1. >>> result = zip(list1,list2)

```

```

2. >>> result
3. [(2, 11), (4, 13), (6, 15), (8, 17)]
4. >>> zip(*result)
5. [(2, 4, 6, 8), (11, 13, 15, 17)]

```

列位注意观察，解压缩得到的结果，跟前面压缩前的结果相比，第二项就少了一个元素19，因为在压缩的时候就丢掉了。

这似乎跟for没有什么关系呀。别着急，思考一个问题，看看如何求解：

问题描述：有一个dictionary, myinfor = {"name": "qiwsir", "site": "qiwsir.github.io", "lang": "python"}, 将这个字典变换成：infor = {"qiwsir": "name", "qiwsir.github.io": "site", "python": "lang"}

解法有几个，如果用for循环，可以这样做（当然，看官如果有方法，欢迎贴出来）。

```

1. >>> infor = {}
2. >>> for k,v in myinfor.items():
3. ...     infor[v]=k
4. ...
5. >>> infor
6. {'python': 'lang', 'qiwsir.github.io': 'site', 'qiwsir': 'name'}

```

下面用zip()来试试：

```

1. >>> dict(zip(myinfor.values(),myinfor.keys()))
2. {'python': 'lang', 'qiwsir.github.io': 'site', 'qiwsir': 'name'}

```

呜呼，这是什么情况？原来这个zip()还能这样用。是的，本质上是这么回事。如果将上面这一行分解开来，看官就明白其中的奥妙了。

```

1. >>> myinfor.values()      #得到两个list
2. ['python', 'qiwsir', 'qiwsir.github.io']
3. >>> myinfor.keys()
4. ['lang', 'name', 'site']
5. >>> temp = zip(myinfor.values(),myinfor.keys())      #压缩成一个list,
    每个元素是一个tuple
6. >>> temp
7. [('python', 'lang'), ('qiwsir', 'name'), ('qiwsir.github.io',
    'site')]
8.
9. >>> dict(temp)              #这是函数dict()的功能, 将上述列
    表转化为dictionary
10. {'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。特别是在用python读取数据库的时候（比如mysql），zip()的作用更会显现。

enumerate

enumerate的详细解释，在《再深点，更懂list》中已经有解释，这里姑且复习。

如果要对一个列表，想得到其中每个元素的偏移量（就是那个脚标）和对应的元素，怎么办呢？可以这样：

```

1. >>> mylist = ["qiwsir",703,"python"]
2. >>> new_list = []
3. >>> for i in range(len(mylist)):
4. ...     new_list.append((i,mylist[i]))
5. ...
6. >>> new_list
7. [(0, 'qiwsir'), (1, 703), (2, 'python')]

```

enumerate的作用就是简化上述操作：

```

1. >>> enumerate(mylist)
2. <enumerate object at 0xb74a63c4>    #出现这个结果，用list就能显示内容.类
    似的会在后面课程出现，意味着可迭代。
3. >>> list(enumerate(mylist))
4. [(0, 'qiwsir'), (1, 703), (2, 'python')]

```

对enumerate()的深刻阐述，还得看这个官方文档：

```

class enumerate(object)
| enumerate(iterable[, start]) -> iterator for index, value of
| iterable
|
| Return an enumerate object. iterable must be another object that
| supports
| iteration. The enumerate object yields pairs containing a count
| (from
| start, which defaults to zero) and a value yielded by the iterable
| argument.
| enumerate is useful for obtaining an indexed list:
| (0, seq[0]), (1, seq[1]), (2, seq[2]), ...
|
| Methods defined here:
|
| getattr(...)
| x.getattr('name') <==> x.name
|
| iter(...)
| x.iter() <==> iter(x)
|
| next(...)
| x.next() -> the next value, or raise StopIteration
|
| _____
| Data and other attributes defined here:
|
| new =
| T.new(S, ...) -> a new object with type S, a subtype of T

```

对官方文档，有的朋友可能看起来有点迷糊，不要紧，至少浏览一下，看个大概。因为随着个人实践的越来越多，对文档的含义理解会越来越

深刻。这就好比令狐冲，刚刚学习了独孤九剑的口诀和招式后，理解不是很深刻，只有在不断的打打杀杀实践中，特别跟东方不败等高手过招之后，才能越来越体会到独孤九剑中的奥妙。

让人欢喜让人忧的迭代

- 让人欢喜让人忧的迭代
 - 逐个访问
 - 文件迭代器

让人欢喜让人忧的迭代

跟一些比较牛X的程序员交流，经常听到他们嘴里冒出一个不标准的英文单词，而loop、iterate、traversal和recursion如果不在其内，总觉得他还不够牛X。当让，真正牛X的绝对不会这么说的，他们只是说“循环、迭代、遍历、递归”，然后再问“这个你懂吗？”。哦，这就是真正牛X的程序员。不过，他也仅仅是牛X罢了，还不是大神。大神程序员是什么样儿呢？他是扫地僧，大隐隐于市。

先搞清楚这些名词再说别的：

- 循环（loop），指的是在满足条件的情况下，重复执行同一段代码。比如，while语句。
- 迭代（iterate），指的是按照某种顺序逐个访问列表中的每一项。比如，for语句。
- 递归（recursion），指的是一个函数不断调用自身的行为。比如，以编程方式输出著名的斐波纳契数列。
- 遍历（traversal），指的是按照一定的规则访问树形结构中的每个节点，而且每个节点都只访问一次。

对于这四个听起来高深莫测的词汇，在教程中，已经涉及到了一个——循环（loop），本经主要介绍一下迭代（iterate），看官在网上google，就会发现，对于迭代和循环、递归之间的比较的文章不少，分别从不同角度将它们进行了对比。这里暂不比较，先搞明白python

中的迭代。之后适当时机再比较，如果我不忘记的话，哈哈。

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）

```
1. >>> lst
2. ['q', 'i', 'w', 's', 'i', 'r']
3. >>> for i in lst:
4. ...     print i,
5. ...
6. q i w s i r
```

除了这种方法，还可以这样：

```
1. >>> lst_iter = iter(lst)      #对原来的list实施了一个iter()
2. >>> lst_iter.next()           #要不厌其烦地一个一个手动访问
3. 'q'
4. >>> lst_iter.next()
5. 'i'
6. >>> lst_iter.next()
7. 'w'
8. >>> lst_iter.next()
9. 's'
10. >>> lst_iter.next()
11. 'i'
12. >>> lst_iter.next()
13. 'r'
14. >>> lst_iter.next()
15. Traceback (most recent call last):
16.   File "<stdin>", line 1, in <module>
17. StopIteration
```

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：

```

1. >>> while True:
2. ...     print lst_iter.next()
3. ...
4. Traceback (most recent call last):          #居然报错，而且错误跟前面一样？
    什么原因
5.   File "<stdin>", line 2, in <module>
6. StopIteration
7.
8. >>> lst_iter = iter(lst)                    #那就再写一遍，上面的错误暂且搁
    置，回头在研究
9. >>> while True:
10. ...     print lst_iter.next()
11. ...
12. q                                           #果然自动化地读取了
13. i
14. w
15. s
16. i
17. r
18. Traceback (most recent call last):          #读取到最后一个之后，报错，停止
    循环
19.   File "<stdin>", line 2, in <module>
20. StopIteration
21. >>>

```

首先了解一下上面用到的那个内置函数：`iter()`，官方文档中有这样一段话描述之：

```
iter(o[, sentinel])
```

Return an iterator object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, o must be a collection object which supports the iteration protocol (the `iter()` method), or it must support the sequence protocol (the `getitem()` method with integer arguments starting at 0). If it does not support either of those protocols, `TypeError` is raised. If the second argument, `sentinel`, is given, then o must be a callable object. The iterator created in this case will call o with no arguments for each call to its `next()` method; if the

value returned is equal to sentinel, StopIteration will be raised, otherwise the value will be returned.

大意是说...(此处故意省略若干字, 因为我相信看此文章的看官英语水平是达到看文档的水平了, 如果没有, 也不用着急, 找个词典什么的帮助一下。)

尽管不翻译了, 但是还要提炼一下主要的东西:

- 返回值是一个迭代器对象
- 参数需要是一个符合迭代协议的对象或者是一个序列对象
- `next()`配合与之使用

什么是“可迭代的对象”呢? 一般, 我们常常将哪些能够用for来一个一个读取元素的对象, 就称之为可迭代的对象。那么for也就被称之为迭代工具。所谓迭代工具, 就是能够按照一定顺序扫描迭代对象的每个元素(按照从左到右的顺序), 显然, 除了for之外, 还有别的可以称作迭代工具, 比如列表解析, in来判断某元素是否属于序列对象等。

那么, 刚才介绍的`iter()`的功能呢? 它与`next()`配合使用, 也是实现上述迭代工具的作用。在python中, 甚至在其它的语言中, 迭代这块的说法比较乱, 主要是名词乱, 刚才我们说, 那些能够实现迭代的东西, 称之为迭代工具, 就是这些迭代工具, 不少程序员都喜欢叫做迭代器。当然, 这都是汉语翻译, 英语就是iterator。

看官看上面的所有例子会发现, 如果用for来迭代, 当到末尾的时候, 就自动结束了, 不会报错。如果用`iter()...next()`迭代, 当最后一个完成之后, 它不会自动结束, 还要向下继续, 但是后面没有元素了, 于是就报一个称之为StopIteration的错误(这个错误的名字叫做: 停止迭代, 这哪里是报错, 分明是警告)。

看官还要关注`iter()...next()`迭代的一个特点。当迭代对象

lst_iter被迭代结束，即每个元素都读取一边之后，指针就移动到了最后一个元素的后面。如果再访问，指针并没有自动返回到首位置，而是仍然停留在末位置，所以报StopIteration，想要再开始，需要重新再入迭代对象。所以，列位就看到，当我在上面重新进行迭代对象赋值之后，又可以继续了。这在for等类型的迭代工具中是没有的。

文件迭代器

现在有一个文件，名称：208.txt，其内容如下：

```
1. Learn python with qiwsir.
2. There is free python course.
3. The website is:
4. http://qiwsir.github.io
5. 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：

```
1. >>> f = open("208.txt")
2. >>> f.readline()          #读第一行
3. 'Learn python with qiwsir.\n'
4. >>> f.readline()          #读第二行
5. 'There is free python course.\n'
6. >>> f.readline()          #读第三行
7. 'The website is:\n'
8. >>> f.readline()          #读第四行
9. 'http://qiwsir.github.io\n'
10. >>> 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
11. 'Its language is Chinese.\n'
12. >>> f.readline()          #无内容了，但是不报错，返回空。
13. ''
```

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：

```
1. >>> for line in f:      #这个操作是紧接着上面的操作进行的，请看官主要观察
2.     ...     print line,  #没有打印出任何东西
3.     ...
```

这段代码之所没有打印出东西来，是因为经过前面的迭代，指针已经移到了最后了。这就是迭代的一个特点，要小心指针的位置。

```
1. >>> f = open("208.txt")    #从头再来
2. >>> for line in f:
3.     ...     print line,
4.     ...
5. Learn python with qiwsir.
6. There is free python course.
7. The website is:
8. http://qiwsir.github.io
9. Its language is Chinese.
```

这种方法是读取文件常用的。另外一个`readlines()`也可以。但是，需要有一些小心的地方，看官如果想不起来小心什么，可以在将关于文件的课程复习一边。

上面过程用`next()`也能够读取。

```
1. >>> f = open("208.txt")
2. >>> f.next()
3. 'Learn python with qiwsir.\n'
4. >>> f.next()
5. 'There is free python course.\n'
6. >>> f.next()
7. 'The website is:\n'
8. >>> f.next()
9. 'http://qiwsir.github.io\n'
```

```

10. >>> f.next()
11. 'Its language is Chinese.\n'
12. >>> f.next()
13. Traceback (most recent call last):
14.   File "<stdin>", line 1, in <module>
15. StopIteration

```

如果用`next()`，就可以直接读取每行的内容。这说明文件是天然的可迭代对象，不需要用`iter()`转换了。

再有，我们用`for`来实现迭代，在本质上，就是自动调用`next()`，只不过这个工作，已经让`for`偷偷地替我们干了，到这里，列位是不是应该给`for`取另外一个名字：它叫雷锋。

前面提到了，列表解析也能够做为迭代工具，在研究列表的时候，看官想必已经清楚了。那么对文件，是否可以用？试一试：

```

1. >>> [ line for line in open('208.txt') ]
2. ['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']

```

至此，看官难道还不为列表解析所折服吗？真的很强大，又强又大呀。

其实，迭代器远远不止上述这么简单，下面我们随便列举一些，在python中还可以这样得到迭代对象中的元素。

```

1. >>> list(open('208.txt'))
2. ['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']
3.
4. >>> tuple(open('208.txt'))
5. ('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')

```



```
6.
7. >>> "$$$".join(open('208.txt'))
8. 'Learn python with qiwsir.\n$$$There is free python course.\n$$$The
   website is:\n$$$http://qiwsir.github.io\n$$$Its language is
   Chinese.\n'
9.
10. >>> a,b,c,d,e = open("208.txt")
11. >>> a
12. 'Learn python with qiwsir.\n'
13. >>> b
14. 'There is free python course.\n'
15. >>> c
16. 'The website is:\n'
17. >>> d
18. 'http://qiwsir.github.io\n'
19. >>> e
20. 'Its language is Chinese.\n'
```

上述方式，在编程实践中不一定用得上，只是向看官展示一下，并且看官要明白，可以这么做，不是非要这么做。

补充一下，字典也可以迭代，看官自己不妨摸索一下（其实前面已经用for迭代过了，这次请摸索一下用iter()...next()手动一步一步迭代）。

大话题小函数(1)

- [大话题小函数\(1\)](#)
 - [lambda](#)
 - [map](#)

大话题小函数(1)

开篇就要提到一个大的话题：编程范型。什么是编程范型？引用[维基百科](#)中的解释：

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。

不管看官是初学者还是老油条，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。

这里推荐一篇文章，这篇文章来自网络：[《主要的编程范型》](#)

扯了不少编程范型，今天本讲要讲什么呢？今天要介绍几个python中的小函数，这几个函数都是从函数式编程借鉴过来的，它们就是：

filter、map、reduce、lambda、yield

有了它们，最大的好处是程序更简洁；没有它们，程序也可以用别的方式实现，只不过麻烦一些罢了。所以，还是能用则用之吧。

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：

```

1. >>> def add(x):      #定义一个函数，将输入的变量增加3,然后返回增加之后的值
2. ...     x +=3
3. ...     return x
4. ...
5. >>> numbers = range(10)
6. >>> numbers
7. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] #有这样一个list, 想让每个数字增加3,然后输出到一个新的list中
8.
9. >>> new_numbers = []
10. >>> for i in numbers:
11. ...     new_numbers.append(add(i)) #调用add()函数, 并append到list中
12. ...
13. >>> new_numbers
14. [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：

```

1. >>> new_numbers = [ i+3 for i in numbers ]
2. >>> new_numbers
3. [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

首先说明，这种列表解析的方式是非常非常好的。

但是，我们偏偏要用`lambda`这个函数替代`add(x)`，如果看官和我一样这么偏执，就可以：

```
1. >>> lam = lambda x:x+3
2. >>> n2 = []
3. >>> for i in numbers:
4. ...     n2.append(lam(i))
5. ...
6. >>> n2
7. [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

这里的`lam`就相当于`add(x)`，请看官对应一下，这一行`lambda x:x+3`就完成`add(x)`的三行（还是两行？），特别是最后返回值。还可以写这样的例子：

```
1. >>> g = lambda x,y:x+y #x+y,并返回结果
2. >>> g(3,4)
3. 7
4. >>> (lambda x:x**2)(4) #返回4的平方
5. 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：

```
1. lambda arg1, arg2, ...argN : expression using arguments
```

要特别提醒看官：虽然`lambda` 函数可以接收任意多个参数（包括可选参数）并且返回单个表达式的值，但是`lambda` 函数不能包含命令，包含的表达式不能超过一个。不要试图向 `lambda` 函数中塞入太

多的东西；如果你需要更复杂的东西，应该定义一个普通函数，然后想让它多长就多长。

就lambda而言，它并没有给程序带来性能上的提升，它带来的是代码的简洁。比如，要打印一个list，里面依次是某个数字的1次方，二次方，三次方，四次方。用lambda可以这样做：

```
1. >>> lamb = [ lambda x:x, lambda x:x**2, lambda x:x**3, lambda x:x**4 ]
2. >>> for i in lamb:
3. ...     print i(3),
4. ...
5. 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。根据我的经验，尽量少用，因为它或许更多地是为减少单行函数的定义而存在的。

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：

```
1. >>> numbers
2. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]          #把列表中每一项都加3
3.
4. >>> map(add, numbers)                 #add(x)是上面讲述的那个函数，但是这里只引用函数名称即可
5. [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
6.
7. >>> map(lambda x: x+3, numbers)       #用lambda当然可以啦
8. [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

map()是python的一个内置函数，它的基本样式是：map(func, seq)，func是一个函数，seq是一个序列对象。在执行的时候，序列

对象中的每个元素，按照从左到右的顺序，依次被取出来，并塞入到func那个函数里面，并将func的返回值依次存到一个list中。

在应用中，map的所能实现的，也可以用别的方式实现。比如：

```

1. >>> items = [1,2,3,4,5]
2. >>> squared = []
3. >>> for i in items:
4. ...     squared.append(i**2)
5. ...
6. >>> squared
7. [1, 4, 9, 16, 25]
8.
9. >>> def sqr(x): return x**2
10. ...
11. >>> map(sqr, items)
12. [1, 4, 9, 16, 25]
13.
14. >>> map(lambda x: x**2, items)
15. [1, 4, 9, 16, 25]
16.
17. >>> [ 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
18. [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，再来浏览有关map()的官方说明，能够更明白一些。

```
map(function, iterable, ...)
```

Apply function to every item of iterable and return a list of the results. If additional iterable arguments are passed, function must take that many arguments and is applied to the items from all iterables in parallel. If one iterable is shorter than another it is assumed to be extended with None items. If function is None, the identity function is assumed; if there are multiple arguments, map()

returns a list consisting of tuples containing the corresponding items from all iterables (a kind of transpose operation). The iterable arguments may be a sequence or any iterable object; the result is always a list.

理解要点：

- 对iterable中的每个元素，依次应用function的方法（函数）（这本质上就是一个for循环）。
- 将所有结果返回一个list。
- 如果参数很多，则对多个参数并行执行function。

例如：

```
1. >>> lst1 = [1,2,3,4,5]
2. >>> lst2 = [6,7,8,9,0]
3. >>> 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
4. [7, 9, 11, 13, 5]
```

请看官注意了，上面这个例子如果用for循环来写，还不是很困难，如果扩展一下，下面的例子用for来改写，就要小心了：

```
1. >>> lst1 = [1,2,3,4,5]
2. >>> lst2 = [6,7,8,9,0]
3. >>> lst3 = [7,8,9,2,1]
4. >>> map(lambda x,y,z: x+y+z, lst1,lst2,lst3)
5. [14, 17, 20, 15, 6]
```

这才显示出map的简洁优雅。

预告：下一讲详解reduce和filter

大话题小函数(2)

- 大话题小函数(2)
 - `reduce`
 - `filter`

大话题小函数(2)

上一讲和本讲的标题是“大话题小函数”，所谓大话题，就是这些函数如果溯源，都会找到听起来更高大上的东西。这种思维方式绝对我坚定地继承了中华民族的优良传统的。自从天朝的臣民看到英国人开始踢足球，一直到现在所谓某国勃起了，都一直在试图论证足球起源于该朝的前前前朝的某国时代，并且还搬出了那时候的一个叫做高俅的球星来论证，当然了，勃起的某国是挡不住该国家队在世界杯征程上的阳痿，只能用高俅来意淫一番了。这种思维方式，我是坚定地继承，因为在我成长过程中，它一直被奉为优良传统。阿Q本来是姓赵的，和赵老爷是本家，比秀才要长三辈，虽然被赵老爷打了嘴。

废话少说，书接前文，已经研究了map，下面来看reduce。

忍不住还得来点废话。不知道看官是不是听说过MapReduce，如果没有，那么Hadoop呢？如果还没有，就google一下。下面是我从[维基百科](#)上抄下来的，共赏之。

MapReduce是Google提出的一个软件架构，用于大规模数据集（大于1TB）的并行运算。概念“Map（映射）”和“Reduce（化简）”，及他们的主要思想，都是从函数式编程语言借来的，还有从矢量编程语言借来的特性。

不用管是不是看懂，总之又可以用开头的思想意淫一下了，原来今天要鼓捣的这个reduce还跟大数据有关呀。不管怎么样，你有梦一般的感觉就行。

reduce

回到现实，清醒一下，继续敲代码：

```
1. >>> reduce(lambda x,y: x+y, [1,2,3,4,5])
2. 15
```

请看官仔细观察，是否能够看出是如何运算的呢？画一个图：



还记得map是怎么运算的吗？忘了？看代码：

```
1. >>> list1 = [1,2,3,4,5,6,7,8,9]
2. >>> list2 = [9,8,7,6,5,4,3,2,1]
3. >>> map(lambda x,y: x+y, list1,list2)
4. [10, 10, 10, 10, 10, 10, 10, 10, 10]
```

看官对比一下，就知道两个的区别了。原来map是上下运算，reduce是横着逐个元素进行运算。

权威的解释来自官网：

```
reduce(function, iterable[, initializer])
```

Apply function of two arguments cumulatively to the items of iterable, from left to right, so as to reduce the iterable to a single value. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates $((((1+2)+3)+4)+5)$. The left argument, *x*, is the accumulated value and the right argument, *y*, is the update value from the iterable. If the optional *initializer* is present, it is placed before the items of the iterable in the calculation, and serves as a default when the iterable is empty. If *initializer* is not given and *iterable* contains only one item, the first item is returned. Roughly equivalent to:

```
1. def reduce(function, iterable, initializer=None):
2.     it = iter(iterable)
3.     if initializer is None:
```

```

4.         try:
5.             initializer = next(it)
6.         except StopIteration:
7.             raise TypeError('reduce() of empty sequence with no
            initial value')
8.     accum_value = initializer
9.     for x in it:
10.        accum_value = function(accum_value, x)
11.     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：

```

1. >>> lst = range(1,6)
2. >>> lst
3. [1, 2, 3, 4, 5]
4. >>> r = 0
5. >>> for i in range(len(lst)):
6. ...     r += lst[i]
7. ...
8. >>> r
9. 15

```

for普世的，reduce是简洁的。

为了锻炼思维，看这么一个问题，有两个list，a = [3,9,8,5,2], b=[1,4,9,2,6], 计算：a[0]b[0]+a[1]b[1]+...的结果。

```

1. >>> a
2. [3, 9, 8, 5, 2]
3. >>> b
4. [1, 4, 9, 2, 6]
5.
6. >>> zip(a,b)           #复习一下zip, 下面的方法中要用到
7. [(3, 1), (9, 4), (8, 9), (5, 2), (2, 6)]
8.
9. >>> sum(x*y for x,y in zip(a,b))    #解析后直接求和

```

```

10. 133
11.
12. >>> new_list = [x*y for x,y in zip(a,b)]      #可以看做是上面方法的分布实
    施
13. >>> #这样解析也可以: new_tuple = (x*y for x,y in zip(a,b))
14. >>> new_list
15. [3, 36, 72, 10, 12]
16. >>> sum(new_list)      #或者: sum(new_tuple)
17. 133
18.
19. >>> reduce(lambda sum, (x,y): sum+x*y, zip(a,b), 0)      #这个方法是在耍酷
    呢吗?
20. 133
21.
22. >>> from operator import add, mul      #耍酷的方法也不止一个
23. >>> reduce(add, map(mul, a, b))
24. 133
25.
26. >>> reduce(lambda x,y: x+y, map(lambda x,y: x*y, a, b))
    #map, reduce, lambda都齐全了, 更酷吗?
27. 133

```

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：

```

1. >>> numbers = range(-5,5)
2. >>> numbers
3. [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
4.
5. >>> filter(lambda x: x>0, numbers)
6. [1, 2, 3, 4]
7.
8. >>> [x for x in numbers if x>0]      #与上面那句等效
9. [1, 2, 3, 4]
10.
11. >>> filter(lambda c: c!='i', 'qiwsir') #能不能对应上面文档说明那句话
    呢？
12. 'qwsr'                               #“If iterable is a string
    or a tuple, the result also has that type;”

```

至此，用两此介绍了几个小函数，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。

python文档

- [Python文档](#)
 - [查看python文档](#)
 - [给自己的程序加上文档](#)

Python文档

文档，这个词语在经常在程序员的嘴里冒出来，有时候他们还经常以文档有没有或者全不全为标准来衡量一个软件项目是否高大上。那么，软件中的文档是什么呢？有什么要求呢？python文档又是什么呢？文档有什么用呢？

文档很重要。独孤九剑的剑诀、易筋经的心法、写着辟邪剑谱的袈裟，这些都是文档。连那些大牛人都要这些文档，更何况我们呢？所以，文档是很重要的。

文档，说白了就是用word（这个最多了）等（注意这里的等，把不常用的工具都等掉了，包括我编辑文本时用的vim工具）文本编写工具写成的包含文本内容但不限于文字的文件。有点啰嗦，啰嗦的目的是为了严谨，呵呵。最好还是来一个更让人信服的定义，当然是来自[维基百科](#)。

软件文档或者源代码文档是指与软件系统及其软件工程过程有关联的文本实体。文档的类型包括软件需求文档，设计文档，测试文档，用户手册等。其中的需求文档，设计文档和测试文档一般是在软件开发过程中由开发者写就的，而用户手册等非过程类文档是由专门的非技术类写作人员写就的。

早期的软件文档主要指的是用户手册，根据Barker的定义，文档是用来对软件系统界面元素的设计、规划和实现过程的记录，以此来增强系统的可用性。而Forward则认为软件文档是被软件工程师之间用作沟通交流的一种方式，沟通的信息主要是有关所开发的软件系统。Parnas则强调文档的权威性，他认为文档应该提供对软件系统的精确描述。

综上，我们可以将软件文档定义为：

1. 文档是一种对软件系统的书面描述；
2. 文档应当精确地描述软件系统；
3. 软件文档是软件工程师之间用作沟通交流的一种方式；
4. 文档的类型有很多种，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。

查看python文档

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。



除了看网站上的文档，还有别的方式吗？

有，而且看官并不陌生，此前已经在本教程中多次用到，那就是`dir()`和`help()`

```
1. >>> dir(list)
2. ['__add__', '__class__', '__contains__', '__delattr__',
    '__delitem__', '__delslice__', '__doc__', '__eq__', '__format__',
    '__ge__', '__getattribute__', '__getitem__', '__getslice__',
    '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
```

```

1.  '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
2.  '__new__', '__reduce__', '__reduce_ex__', '__repr__',
3.  '__reversed__', '__rmul__', '__setattr__', '__setitem__',
4.  '__setslice__', '__sizeof__', '__str__', '__subclasshook__',
5.  'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
6.  'reverse', 'sort']
7.
8.
9. >>> help(list.__mul__)
10.
11. Help on wrapper_descriptor:
12.
13. __mul__(...)
14.     x.__mul__(n) <==> x*n

```

这种查看文档的方式，在交互模式下经常用到，快捷方便，请看官务必牢记并使用。

正如前面已经介绍过的，还有一个文档：**doc**, help调用的其实就是这个函数里面的内容。

```

1. >>> print(list.__mul__.__doc__)      #与help(list.__mul__)显示的内容一
2.     致
3. x.__mul__(n) <==> x*n
4.
5. >>> print(list.index.__doc__)        #查看index的文档
6. L.index(value, [start, [stop]]) -> integer -- return first index of
7.     value.
8.
9. Raises ValueError if the value is not present.

```

给自己的程序加上文档

在自己编写程序的时候，也非常希望能够有类似上面查看python文档的功能，可以通过某种方式查看自己的程序文档，这样显得自己多牛呀。

有一种方法可以实现，就是在你所编写的程序中用三个双引号或者单引号成对地出现，中间写上有关文档内容。

```

1. >>> def qiwsir():
2. ...     """I like python"""
3. ...     print "http://qiwsir.github.io"
4. ...
5. >>> qiwsir()
6. http://qiwsir.github.io
7.
8. >>> print(qiwsir.__doc__)    #用这种方法可以看自己写的函数中的文档
9. I like python
10.
11. >>> help(qiwsir)             #其实就是调用__doc__显示的内容
12.
13. Help on function qiwsir in module __main__:
14.
15. qiwsir()
16.     I like python

```

另外，对于一个文件，可以把有关说明放在文件的前面，不影响该文件代码运行。

例如，有这样一个扩展名是.py的python文件，其内容是：

```

1. #!/usr/bin/env python
2. #coding:utf-8
3.
4. import random
5.
6. number = random.randint(1,100)
7.
8. guess = 0
9.
10. while True:
11.
12.     num_input = raw_input("please input one integer that is in 1 to

```



```

100:")
13.     guess +=1
14.
15.     if not num_input.isdigit():
16.         print "Please input interger."
17.     elif int(num_input)<0 and int(num_input)>=100:
18.         print "The number should be in 1 to 100."
19.     else:
20.         if number==int(num_input):
21.             print "OK, you are good.It is only %d, then you
succeeded."%guess
22.             break
23.         elif number>int(num_input):
24.             print "your number is more less."
25.         elif number<int(num_input):
26.             print "your number is bigger."
27.         else:
28.             print "There is something bad, I will not work"

```

这段程序，就是在《用while来循环》中用到的一个猜数字的游戏，它存储在名为205-2.py的文件中，如果要对这段程序写一个文档，就可以这么做。

```

1.  """
2.     This is a game.
3.     I am Qiwei.
4.     I like python.
5.     I am writing python articles in my website.
6.     My website is http://qiwsir.github.io
7.     You can learn python free in it.
8.  """
9.
10.  #!/usr/bin/env python
11.  #coding:utf-8
12.
13.  import random
14.

```

```
15. number = random.randint(1,100)
16.
17. guess = 0
18.
19. while True:
20.
21.     num_input = raw_input("please input one integer that is in 1 to
    100:")
22.     guess +=1
23.
24.     if not num_input.isdigit():
25.         print "Please input interger."
26.     elif int(num_input)<0 and int(num_input)>=100:
27.         print "The number should be in 1 to 100."
28.     else:
29.         if number==int(num_input):
30.             print "OK, you are good.It is only %d, then you
    succeeded."%guess
31.             break
32.         elif number>int(num_input):
33.             print "your number is more less."
34.         elif number<int(num_input):
35.             print "your number is bigger."
36.         else:
37.             print "There is something bad, I will not work"
```

最后，推荐一篇相当相当好的文章，与列位分享：

[Python 自省指南:如何监视您的 Python 对象](#)

重回函数

- [重回函数](#)
 - [函数的基本结构](#)
 - [调用函数](#)
 - [注意事项](#)

重回函数

在本教程的开始部分，就已经引入了函数的概念：《[永远强大的函数](#)》，之所以那时候就提到函数，是因为我觉得函数之重要，远远超过一般。这里，重回函数，一是复习，二是要在已经学习的基础上，对函数有更深刻的理解。

函数的基本结构

Python中的函数基本结构：

```
1. def 函数名([参数列表]):  
2.  
3.     语句
```

几点说明：

- 函数名的命名规则要符合python中的命名要求。一般用小写字母和单下划线、数字等组合
- def是函数的开始，这个简写来自英文单词define，显然，就是要定义一个什么东西
- 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

- 语句，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：

```

1. >>> def name():           #定义一个无参数的函数，只是通过这个函数打印
2.     ...     print "qiwsir" #缩进4个空格
3.     ...
4. >>> name()                #调用函数，打印结果
5. qiwsir
6.
7. >>> def add(x,y):         #定义一个非常简单的函数
8.     ...     return x+y    #缩进4个空格
9.     ...
10. >>> add(2,3)              #通过函数，计算2+3
11. 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。

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。python中不需要提前声明变量，有的语言就需要声明。只有当用到该变量的时候，才建立变量与对象的对应关系，否则，关系不建立。而对象才有不同的类型。那么，在add(x,y)函数中，x,y在引用对象之前，是完全自由的，也就是它们可以引用任何对象，只要后面的运算许可，如果后面的运算不许可，则会报错。

```

1. >>> add("qiw","sir")      #这里，x="qiw",y="sir",让函数计算x+y,也就是"qiw"+"sir"
2. 'qiwsir'
3.
4. >>> add("qiwsir",4)
5. Traceback (most recent call last):
6.   File "<stdin>", line 1, in <module>

```

```
7. File "<stdin>", line 2, in add
8. 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。这是python和其它的静态语言的重要区别。在python中，代码不关心特定的数据类型。

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。

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：

```
1. >>> result = add(3,4)
2. >>> result
3. 7
```

在这里，其实解释了函数的一个秘密。 $\text{add}(x, y)$ 在被运行之前，计算机内是不存在的，直到代码运行到这里的时候，在计算机中，就建立起来了一个对象，这就如同前面所学习过的字符串、列表等类型的对象一样，运行 $\text{add}(x, y)$ 之后，也建立了一个 $\text{add}(x, y)$ 的对象，这个对象与变量`result`可以建立引用关系，并且 $\text{add}(x, y)$ 将运算结果返回。于是，通过`result`就可以查看运算结果。

如果看官上面一段，感觉有点吃力或者晕乎，也不要紧，那就再读一边。是在搞不明白，就不要搞了。随着学习的深入，它会被明白的。

调用函数

扯了不少函数怎么编写，到底编写函数有什么用？在程序中怎么调用呢？

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一个猜数字的程序，在那么就没有写函数，当然，用python的函数不算了。现在之所以使用函数，主要是：

1. 降低编程的难度，通常将一个复杂的大问题分解成一系列更简单的小问题，然后将小问题继续划分成更小的问题，当问题细化为足够简单时，就可以分而治之。为了实现这种分而治之的设想，就要通过编写函数，将各个小问题逐个击破，再集合起来，解决大的问题。（看官请注意，分而治之的思想是编程的一个重要思想，所谓“分治”方法也。）
2. 代码重(chong, 二声音)用。在编程的过程中，比较忌讳同样一段代码不断的重复，所以，可以定义一个函数，在程序的多个位置使用，也可以用于多个程序。当然，后面我们还会讲到“模块”（此前也涉及到了，就是import导入的那个东西），还可以把函数放到一个模块中供其他程序员使用。也可以使用其他程序员定义的函数（比如import ...,前面已经用到了，就是应用了别人——创造python的人——写好的函数）。这就避免了重复劳动，提供了工作效率。

这样看来，函数还是很必要的了。废话少说，那就看函数怎么调用吧。以add(x,y)为例，前面已经演示了基本调用方式，此外，还可以这样：

```

1. >>> def add(x,y):           #为了能够更明了显示参数赋值特点，重写此函数
2.     ...     print "x=",x     #分别打印参数赋值结果
3.     ...     print "y=",y
4.     ...     return x+y
5.     ...
6. >>> add(10,3)               #x=10,y=3
7. x= 10
8. y= 3
9. 13

```

```

10. >>> add(x=10,y=3)      #同上
11. x= 10
12. y= 3
13. 13
14. >>> add(y=10,x=3)      #x=3,y=10
15. x= 3
16. y= 10
17. 13
18. >>> add(3,10)          #x=3,y=10
19. x= 3
20. y= 10
21. 13

```

在定义函数的时候，参数可以想前面那样，等待被赋值，也可以定义的时候就赋给一个默认值。例如：

```

1. >>> def times(x,y=2):    #y的默认值为2
2. ...     print "x=",x
3. ...     print "y=",y
4. ...     return x*y
5. ...
6. >>> times(3)              #x=3,y=2
7. x= 3
8. y= 2
9. 6
10.
11. >>> times(x=3)           #同上
12. x= 3
13. y= 2
14. 6
15.
16. >>> times(3,4)           #x=3,y=4,y的值不再是2
17. x= 3
18. y= 4
19. 12
20.
21. >>> times("qiwsir")     #再次体现了多态特点

```

```
22. x= qiwsir
23. y= 2
24. 'qiwsirqiwsir'
```

给列位看官提一个思考题，请在闲暇之余用python完成：写两个数的加、减、乘、除的函数，然后用这些函数，完成简单的计算。

注意事项

下面的若干条，是常见编写代码的注意事项：

1. 别忘了冒号。一定要记住符合语句首行末尾输入“:”(if,while,for等的第一行)
2. 从第一行开始。要确定顶层（无嵌套）程序代码从第一行开始。
3. 空白行在交互模式提示符下很重要。模块文件中符合语句内的空白行常被忽视。但是，当你在交互模式提示符下输入代码时，空白行则是会结束语句。
4. 缩进要一致。避免在块缩进中混合制表符和空格。
5. 使用简洁的for循环，而不是while or range.相比，for循环更易写，运行起来也更快
6. 要注意赋值语句中的可变对象。
7. 不要期待在原处修改的函数会返回结果,比如list.append()
8. 一定要之用括号调用函数
9. 不要在导入和重载中使用扩展名或路径。

变量和参数

- 变量和参数
 - 全局变量和局部变量
 - 不确定参数的数量

变量和参数

对于变量和参数，不管是已经敲代码多年的老鸟，还是刚刚接触编程的小白，都会有时候清楚，有时候又有点模糊。因为，在实际应用中，它们之间分分离离，比如，敲代码都知道， $x=3$ 中 x 是变量，它不是参数，但是在函数 $y=3x+4$ 中， x 是变量，也是参数。那么什么这两个到底有什么区别和联系呢？我在网上搜了一下，发现很多说法，虽然大同小异，但是似乎只有下面这一段来自[微软网站](#)的比较高度抽象，而且意义涵盖深远。我摘抄过来，看官读一读，是否理解，虽然是针对VB而言的，一样有启发。

参数和变量之间的差异 (*Visual Basic*)

多数情况下，过程必须包含有关调用环境的一些信息。执行重复或共享任务的过程对每次调用使用不同的信息。此信息包含每次调用过程时传递给它的变量、常量和表达式。

若要将此信息传递给过程，过程先要定义一个形参，然后调用代码将一个实参传递给所定义的形参。您可以将形参当作一个停车位，而将实参当作一辆汽车。就像一个停车位可以在不同时间停放不同的汽车一样，调用代码在每次调用过程时可以将不同的实参传递给同一个形参。

形参表示一个值，过程希望您在调用它时传递该值。

当您定义 *Function* 或 *Sub* 过程时，需要在紧跟过程名称的括号内指定形参列表。对于每个形参，您可以指定名称、数据类型和传入机制 (*ByVal* (*Visual Basic*) 或 *ByRef* (*Visual Basic*))。您还可以指示某个形参是可选的。这意味着调用代码不必传递它的值。

每个形参的名称均可作为过程内的局部变量。形参名称的使用方法与其他任何变量的使用方法相同。

实参表示在您调用过程时传递给过程形参的值。调用代码在调用过程时提供参数。

调用 *Function* 或 *Sub* 过程时，需要在紧跟过程名称的括号内包括实参列表。每个实参与此列表中位于相同位置的那个形参相对应。

与形参定义不同，实参没有名称。每个实参就是一个表达式，它包含零或多个变量、常数和文本。求值的表达式的数据类型通常应与为相应形参定义的数据类型相匹配，并且在任何情况下，该表达式值都必须可转换为此形参类型。

看官如果硬着头皮看完这段引文，发现里面有几个关键词：参数、变量、形参、实参。本来想弄清楚参数和变量，结果又冒出另外两个东东，更混乱了。请稍安勿躁，本来这段引文就是有点多余，但是，之所以引用，就是让列位开阔一下眼界，在编程业界，类似的东西有很多名词。下次听到有人说这些，不用害怕啦，反正自己听过了。

在Python中，没有这么复杂。

看完上面让人晕头转向的引文之后，再看下面的代码，就会豁然开朗了。

```
1. >>> def add(x):      #x是参数
2. ...     a = 10        #a是变量
3. ...     return a+x
4. ...
5. >>> x = 3            #x是变量，只不过在函数之外
6. >>> add(x)           #这里的x是参数，但是它由前面的变量x传递对象3
7. 13
8. >>> add(3)           #把上面的过程合并了
9. 13
```

至此，看官是否清楚了一点点。当然，我所表述不正确之处或者理解错误之处，也请看官不吝赐教，小可作揖感谢。

全局变量和局部变量

下面是一段代码，注意这段代码中有一个函数funcx()，这个函数里面有一个变量x=9，在函数的前面也有一个变量x=2

```

1. x = 2
2.
3. def funcx():
4.     x = 9
5.     print "this x is in the funcx:-->",x
6.
7. funcx()
8. print "-----"
9. print "this x is out of funcx:-->",x

```

那么，这段代码输出的结果是什么呢？看：

```

1. this x is in the funcx:--> 9
2. -----
3. this x is out of funcx:--> 2

```

从输出看出，运行funcx()，输出了funcx()里面的变量x=9；然后执行代码中的最后一行，print "this x is out of funcx:-->",x

特别要关注的是，前一个x输出的是函数内部的变量x；后一个x输出的是函数外面的变量x。两个变量彼此没有互相影响，虽然都是x。从这里看出，两个x各自在各自的领域内起到作用，那么这样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

```

1. x = 2
2. def funcx():
3.     global x
4.     x = 9
5.     print "this x is in the funcx:-->",x
6.
7. funcx()
8. print "-----"

```

```
9.      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。

```
1.  this x is in the funcx:--> 9
2.  -----
3.  this x is out of funcx:--> 9
```

好似全局变量能力很强悍，能够统帅函数内外。但是，要注意，这个东西要慎重使用，因为往往容易带来变量的换乱。内外有别，在程序中一定要注意的。

不确定参数的数量

在设计函数的时候，有时候我们能够确认参数的个数，比如一个用来计算圆面积的函数，它所需要的参数就是半径（ πr^2 ），这个函数的参数是确定的。

然而，这个世界不总是这么简单的，也不总是这么确定的，反而不确定性是这个世界常常存在的。如果看官了解量子力学这个好多人听都没有听过的东西，那就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？

既然有很多不确定性，那么函数的参数的个数，也当然有不确定性，函数怎么解决这个问题呢？python用这样的方式解决参数个数的不确定性：

```
1.  def add(x, *arg):
```

```

2.     print x           #输出参数x的值
3.     result = x
4.     print arg         #输出通过*arg方式得到的值
5.     for i in arg:
6.         result +=i
7.     return result
8.
9.     print add(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.     1                 #这是函数体内的第一个print，参数x得到的值是1
2.     (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
3.     45                #最后的计算结果

```

上面这个输出的结果表现相当不界面友好，如果不对照着原函数，根本不知道每行打印的是什么东西。自责呀。

从上面例子可以看出，如果输入的参数过多，其它参数全部通过 *arg，以元组的形式传给了参数（变量）arg。请看官注意，我这里用了一个模糊的词语：参数（变量），这样的表述意思是，在传入数据的前，arg在函数头部是参数，当在函数语句中，又用到了它，就是变量。也就是在很多时候，函数中的参数和变量是不用那么太区分较真的，只要知道对象是通过什么渠道、那个东西传到了什么目标即可。

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：

```

1.     >>> def foo(*args):
2.         ...     print args         #打印通过这个参数得到的对象
3.         ...
4.     >>> #下面演示分别传入不同的值，通过参数*args得到的结果
5.
6.     >>> foo(1,2,3)
7.     (1, 2, 3)

```

```

8.
9. >>> foo("qiwsir", "qiwsir.github.io", "python")
10. ('qiwsir', 'qiwsir.github.io', 'python')
11.
12. >>> foo("qiwsir", 307, ["qiwsir", 2],
13.         {"name": "qiwsir", "lang": "python"})
14. ('qiwsir', 307, ['qiwsir', 2], {'lang': 'python', 'name':
15.     'qiwsir'})

```

不管是什么，都一股脑地塞进了tuple中。

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：

```

1. >>> def foo(**kargs):
2.     ...     print kargs
3.     ...
4. >>> foo(a=1, b=2, c=3)    #注意观察这次赋值的方式和打印的结果
5. {'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？

```

1. >>> foo(1, 2, 3)
2. Traceback (most recent call last):
3.   File "<stdin>", line 1, in <module>
4. TypeError: foo() takes exactly 0 arguments (3 given)

```

看官到这里可能想了，不是不确定性吗？我也不知道参数到底会可能用什么样的方式传值呀，这好办，把上面的都综合起来。

```

1. >>> def foo(x, y, z, *args, **kargs):
2.     ...     print x
3.     ...     print y
4.     ...     print z
5.     ...     print args
6.     ...     print kargs

```

```
7. ...
8. >>> foo('qiwsir',2,"python")
9. qiwsir
10. 2
11. python
12. ()
13. {}
14. >>> foo(1,2,3,4,5)
15. 1
16. 2
17. 3
18. (4, 5)
19. {}
20. >>> foo(1,2,3,4,5,name="qiwsir")
21. 1
22. 2
23. 3
24. (4, 5)
25. {'name': 'qiwsir'}
```

很good了，这样就能够足以应付各种各样的参数要求了。

总结参数的传递

- 总结参数的传递

- 参数的传递

- `def foo(p1,p2,p3,...)`
 - `def foo(p1=value1,p2=value2,...)`
 - `def foo(*args)`
 - `def foo(**args)`

总结参数的传递

就前面所讲，函数的基本内容已经完毕。但是，函数还有很多值得不断玩味的细节。这里进行阐述。

参数的传递

python中函数的参数通过赋值的方式来传递引用对象。下面总结通过总结常见的函数参数定义方式，来理解参数传递的流程。

```
def foo(p1,p2,p3,...)
```

这种方式最常见了，列出有限个数的参数，并且彼此之间用逗号隔开。在调用函数的时候，按照顺序以此对参数进行赋值，特备注意的是，参数的名字不重要，重要的是位置。而且，必须数量一致，一一对应。第一个对象（可能是数值、字符串等等）对应第一个参数，第二个对应第二个参数，如此对应，不得偏左也不得偏右。

```
1. >>> def foo(p1,p2,p3):
2. ...     print "p1==>",p1
3. ...     print "p2==>",p2
4. ...     print "p3==>",p3
```



```

5. ...
6. >>> foo("python",1,["qiwsir","github","io"])    #一一对应地赋值
7. p1==> python
8. p2==> 1
9. p3==> ['qiwsir', 'github', 'io']
10.
11. >>> foo("python")
12. Traceback (most recent call last):
13.   File "<stdin>", line 1, in <module>
14. TypeError: foo() takes exactly 3 arguments (1 given)    #注意看报错信息
15.
16. >>> foo("python",1,2,3)
17. Traceback (most recent call last):
18.   File "<stdin>", line 1, in <module>
19. TypeError: foo() takes exactly 3 arguments (4 given)    #要求3个参数，实际上放置了4个，报错

```

def foo(p1=value1,p2=value2,...)

这种方式比前面一种更明确某个参数的赋值，貌似这样就不乱子了，很明确呀。颇有一个萝卜对着一个坑的意味。

还是上面那个函数，用下面的方式赋值，就不用担心顺序问题了。

```

1. >>> foo(p3=3,p1=10,p2=222)
2. p1==> 10
3. p2==> 222
4. 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

```

1. >>> def foo(p1,p2=22,p3=33):    #设置了两个参数p2,p3的默认值
2. ...     print "p1==>",p1
3. ...     print "p2==>",p2
4. ...     print "p3==>",p3

```

```

5. ...
6. >>> foo(11)          #p1=11, 其它的参数为默认赋值
7. p1==> 11
8. p2==> 22
9. p3==> 33
10. >>> foo(11,222)      #按照顺序, p2=222, p3依旧维持原默认值
11. p1==> 11
12. p2==> 222
13. p3==> 33
14. >>> foo(11,222,333)  #按顺序赋值
15. p1==> 11
16. p2==> 222
17. p3==> 333
18.
19. >>> foo(11,p2=122)
20. p1==> 11
21. p2==> 122
22. p3==> 33
23.
24. >>> foo(p2=122)      #p1没有默认值, 必须要赋值的, 否则报错
25. Traceback (most recent call last):
26.   File "<stdin>", line 1, in <module>
27. 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。

```

1. >>> def foo(*args):    #接收不确定个数的数据对象
2. ...     print args
3. ...
4. >>> foo("qiwsir.github.io") #以tuple形式接收到, 哪怕是一个
5. ('qiwsir.github.io',)
6. >>> foo("qiwsir.github.io","python")
7. ('qiwsir.github.io', 'python')

```

上一讲中已经有例子说明，可以和前面的混合使用。此处不赘述。

```
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。

```

1. >>> def foo(**args):      #这种方式接收，以dictionary的形式接收数据对象
2. ...     print args
3. ...
4.
5. >>> foo(1,2,3)            #这样就报错了
6. Traceback (most recent call last):
7.   File "<stdin>", line 1, in <module>
8. TypeError: foo() takes exactly 0 arguments (3 given)
9.
10. >>> foo(a=1,b=2,c=3)      #这样就可以了，因为有了键值对
11. {'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

```

1. >>> def foo(x,y=2,*targs,**dargs):
2. ...     print "x==>",x
3. ...     print "y==>",y
4. ...     print "targs_tuple==>",targs
5. ...     print "dargs_dict==>",dargs
6. ...
7.
8. >>> foo("1x")
9. x==> 1x
10. y==> 2
11. targs_tuple==> ()
12. dargs_dict==> {}
13.
14. >>> foo("1x","2y")
15. x==> 1x
16. y==> 2y
17. targs_tuple==> ()
```

```
18. dargs_dict==> {}
19.
20. >>> foo("1x", "2y", "3t1", "3t2")
21. x==> 1x
22. y==> 2y
23. targs_tuple==> ('3t1', '3t2')
24. dargs_dict==> {}
25.
26. >>> foo("1x", "2y", "3t1", "3t2", d1="4d1", d2="4d2")
27. x==> 1x
28. y==> 2y
29. targs_tuple==> ('3t1', '3t2')
30. dargs_dict==> {'d2': '4d2', 'd1': '4d1'}
```

通过上面的例子，看官是否看出什么名堂了呢？

传说中的函数条规

- 传说中的函数编写条规
 - 小试一下递归
 - 铭记：函数是对象

传说中的函数编写条规

关于函数的事情，总是说不完的，下面就罗列一些编写函数的注意事项。特别声明，这些事项不是我总结的，我是从一本名字为《Learning Python》的书里面抄过来的，顺便写成了汉语，当然，是按照自己的视角翻译的，里面也夹杂了一些自己的观点。看官也可以理解为源于《Learning Python》但又有点儿不同。

- 函数具有独立性。也就是常说的不要有太强的耦合性。要让函数能够独立于外部的东西。参数和return语句就是实现这种独立性的最好方法。
- 尽量不要使用全局变量，这也是让函数具有低耦合度的方法。全局变量虽然进行了函数内外通信，但是它强化了函数对外部的依赖，常常让函数的修改和程序调试比较麻烦。
- 如果参数的对象是可变类型的数据，在函数中，不要做对它的修改操作。当然，更多时候，参数传入的最好是不可变的。
- 函数实现的功能和目标要单一化。每个函数的开头，都要有简短的一句话来说明本函数的功能和目标。
- 函数不要太大，能小则小，根据前一条的原则，功能目标单一，则代码条数就小了。如果感觉有点大，看看能不能拆解开，分别为几个函数。
- 不要修改另外一个模块文件中的变量。这跟前面的道理是一样的，目的是降低耦合性。

小试一下递归

对于在python中使用递归，我一项持谨慎态度，能不用就不用，为什么呢？一方面深恐自己学艺不精，另外，递归不仅消耗资源，而且很多时候速度也不如for循环快。

不过，做为程序员，递归还是需要了解的。这里就列举一个简单的例子。

```
1. >>> def newsum(lst):
2. ...     if not lst:
3. ...         return 0
4. ...     else:
5. ...         return lst[0] + newsum(lst[1:])
6. ...
7. >>> newsum([1,2,3])
8. 6
```

这是一个对list进行求和的函数（看官可能想到了，不是在python中有一个sum内置函数来求和么？为什么要自己写呢？是的，在实际的编程中，没有必要自己写，用sum就可以了。这里用这个例子，纯粹是为了说明递归，没有编程实践的意义），当然，我没有判断传给函数的参数是否为完全由数字组成的list，所以，如果输入的list中字母，就会编程这样了：

```
1. >>> newsum([1,2,3,'q'])
2. Traceback (most recent call last):
3.   File "<stdin>", line 1, in <module>
4.   File "<stdin>", line 5, in newsum
5.   File "<stdin>", line 5, in newsum
6.   File "<stdin>", line 5, in newsum
7.   File "<stdin>", line 5, in newsum
8. TypeError: cannot concatenate 'str' and 'int' objects
```

这就是本函数的缺憾了。但是，为了说明递归，我们就顾不了这么多了。暂且忽略这个缺憾。看官注意上面的函数中，有一句：`return lst(0)+newsum(lst[1:])`，在这句话中，又调用了一边函数本身。对了，这就递归，在函数中调用本函数自己。当然，区别在于传入的参数有变化了。为了清除函数的调用流程，我们可以将每次传入的参数打印出来：

```
1. >>> def newsum(lst):
2. ...     print lst
3. ...     if not lst:
4. ...         return 0
5. ...     else:
6. ...         return lst[0] + newsum(lst[1:])
7. ...
8. >>> newsum([1,2,3])
9. [1, 2, 3]
10. [2, 3]
11. [3]
12. []
13. 6
```

这就是递归了。

其实，看官或许已经想到了，即使不用sum，也可以用for来事项上述操作。

```
1. >>> lst = [1,2,3]
2. >>> sum_result = 0
3. >>> for x in lst: sum_result += x
4. ...
5. >>> sum_result
6. 6
```

铭记：函数是对象

还记得，在第一部分学习的时候，不断强调的：变量无类型，数据有类型，那时候遇到的数据包括字符串、数值、列表、元组、字典、文件，这些东西，都被视为对象。函数跟它们类似，也是对象。因此就可以像以前的对象一样进行赋值、传递给其它函数、嵌入到数据结构、从一个函数返回给另一个函数等等面向对象的操作。当然，函数这个对象也有特殊性，就是它可以由一个函数表达式后面的括号中的列表参数调用。

```

1. >>> def newsum(lst):           #依然以这个递归的函数为例
2. ...     print lst
3. ...     if not lst:
4. ...         return 0
5. ...     else:
6. ...         return lst[0] + newsum(lst[1:])
7. ...
8.
9. >>> lst = [1,2,3]
10.
11. >>> newsum(lst)               #这是前面已经常用的方法
12. [1, 2, 3]
13. [2, 3]
14. [3]
15. []
16. 6
17. >>> recursion_fun = newsum    #通过赋值语句，让变量recursion_fun也引用了函
    #数newsum(lst)对象
18. >>> recursion_fun(lst)        #从而变量能够实现等同函数调用的操作
19. [1, 2, 3]
20. [2, 3]
21. [3]
22. []
23. 6

```

再看一个例子，在这个例子中，一定要谨记函数是对象。看官曾记否？在list中，可以容纳任何对象，那么，是否能够容纳一个函数中呢？


```

1. >>> fun_list = [(newsum, [1, 2, 3]), (newsum, [1, 2, 3, 4, 5])]
2. >>> for fun, arg in fun_list:
3. ...     fun(arg)
4. ...
5. [1, 2, 3]
6. [2, 3]
7. [3]
8. []
9. 6
10. [1, 2, 3, 4, 5]
11. [2, 3, 4, 5]
12. [3, 4, 5]
13. [4, 5]
14. [5]
15. []
16. 15

```

函数，真的就是对象啊。

既然是对象，就可以用`dir(object)`方式查看有关信息喽：

```

1. >>> dir(newsum)
2. ['__call__', '__class__', '__closure__', '__code__',
   '__defaults__', '__delattr__', '__dict__', '__doc__', '__format__',
   '__get__', '__getattr__', '__globals__', '__hash__',
   '__init__', '__module__', '__name__', '__new__', '__reduce__',
   '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
   '__str__', '__subclasshook__', 'func_closure', 'func_code',
   'func_defaults', 'func_dict', 'func_doc', 'func_globals',
   'func_name']
3. >>> dir(newsum.__code__)
4. ['__class__', '__cmp__', '__delattr__', '__doc__', '__eq__',
   '__format__', '__ge__', '__getattr__', '__gt__', '__hash__',
   '__init__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__',
   '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
   '__str__', '__subclasshook__', 'co_argcount', 'co_cellvars',
   'co_code', 'co_consts', 'co_filename', 'co_firstlineno',

```

```
'co_flags', 'co_freevars', 'co_lnotab', 'co_name', 'co_names',  
'co_nlocals', 'co_stacksize', 'co_varnames']  
5. >>> newsum.__code__.__doc__  
6. 'code(argcount, nlocals, stacksize, flags, codestring, constants,  
   names,\n      varnames, filename, name, firstlineno, lnotab[,  
   freevars[, cellvars]])\n\nCreate a code object.  Not for the faint  
of heart.'  
7. >>> newsum.__code__.co_varnames  
8. ('lst',)  
9. >>> newsum.__code__.co_argcount  
10. 1
```

所以，各位看官，在使用函数的时候，首先要把它放在对象的层面考量，它不是什么特殊的东西，尽管我们使用了不少篇幅讲述它，但它终究还是一个对象。

关于类的基本认识

- 关于类的基本认识
 - 编写类

关于类的基本认识

在开始部分，请看官非常非常耐心地阅读下面几个枯燥的术语解释，本来这不符合本教程的风格，但是，请看官谅解，因为列位将来一定要阅读枯燥的东西的。这些枯燥的属于解释，均来自维基百科。

1、问题空间

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）。

2、对象

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。

对象这个属于，比较抽象。因此，有人认为，将object翻译为“对象”，常常让人迷茫，不如翻译为“物件”更好。因为“物件”让人感到一种具体的东西，而所谓对象，就是指那种具体的东西。

这种看法在某些语言中是非常适合的。但是，在Python中，则无所谓，不管怎样，python中的一切都是对象，不管是字符串、函数、模

块还是类，都是对象。“万物皆对象”。

都是对象有什么优势吗？太有了。这说明python天生就是OOP的。也说明，python中的所有东西，都能够进行拼凑组合应用，因为对象就是可以拼凑组合应用的。

对于对象这个东西，OOP大师Grandy Booch的定义，应该是权威的，相关定义的内容包括：

- 对象：一个对象有自己的状态、行为和唯一的标识；所有相同类型的对象所具有的结构和行为在他们共同的类中被定义。
- 状态（**state**）：包括这个对象已有的属性（通常是类里面已经定义好的）在加上对象具有的当前属性值（这些属性往往是动态的）
- 行为（**behavior**）：是指一个对象如何影响外界及被外界影响，表现为对象自身状态的改变和信息的传递。
- 标识（**identity**）：是指一个对象所具有的区别于所有其它对象的属性。（本质上指内存中所创建的对象地址）

3、面向对象

面向对象程序设计（英语：*Object-oriented programming*，缩写：*OOP*）是一种程序设计范型，同时也是一种程序开发的方法。对象指的是类的实例。它将对象作为程序的基本单元，将程序和数据封装其中，以提高软件的重用性、灵活性和扩展性。

面向对象程序设计可以看作一种在程序中包含各种独立而又互相调用的对象的思想，这与传统的思想刚好相反：传统的程序设计主张将程序看作一系列函数的集合，或者直接就是一系列对电脑下达的指令。面向对象程序设计中的每一个对象都应该能够接受数据、处理数据并将数据传达给其它对象，因此它们都可以被看作一个小型的“机器”，即对象。

目前已经被证实的是，面向对象程序设计推广了程序的灵活性和可维护性，并且在大型项目设计中广为应用。此外，支持者声称面向对象程序设计要比以往的做法更加便于学习，因为它能够让人们更简单地设计并维护程序，使得程序更加便于分析、设计、理解。反对者在某些领域对此予以否认。

当我们提到面向对象的时候，它不仅指一种程序设计方法。它更多意义上是一种程序开发方式。在这一方面，我们必须了解更多关于面向对象系统分析和面向对象设计（*Object Oriented Design*，简称OOD）方面的知识。

下面再引用一段来自维基百科中关于OOP的历史。

面向对象程序设计的雏形，早在1960年的*Simula*语言中即可发现，当时的程序设计领域正面临着一种危机：在软硬件环境逐渐复杂的情况下，软件如何得到良好的维护？面向对象程序设计在某种程度上通过强调可重复性解决了这一问题。20世纪70年代的*Smalltalk*语言在面向对象方面堪称经典——以至于30年后的今天依然将这一语言视为面向对象语言的基础。

计算机科学中对象和实例概念的最早萌芽可以追溯到麻省理工学院的*PDP-1*系统。这一系统大概是最早的基于容量架构(*capability based architecture*)的实际系统。另外1963年*Ivan Sutherland*的*Sketchpad*应用中也蕴含了同样的思想。对象作为编程实体最早是于1960年代由*Simula 67*语言引入思维。*Simula*这一语言是奥利·约翰·达尔和克利斯登·奈加特在挪威奥斯陆计算机中心为模拟环境而设计的。（据说，他们是为了模拟船只而设计的这种语言，并且对不同船只间属性的相互影响感兴趣。他们将不同的船只归纳为不同的类，而每一个对象，基于它的类，可以定义它自己的属性和行为。）这种办法是分析式程序的最早概念体现。在分析式程序中，我们将真实世界的对象映射到抽象的对象，这叫做“模拟”。*Simula*不仅引入了“类”的概念，还应用了实例这一思想——这可能是这些概念的最早应用。

20世纪70年代施乐*PARC*研究所发明的*Smalltalk*语言将面向对象程序设计的概念定义为，在基础运算中，对对象和消息的广泛应用。*Smalltalk*的创建者深受*Simula 67*的主要思想影响，但*Smalltalk*中的对象是完全动态的——它们可以被创建、修改并销毁，这与*Simula*中的静态对象有所区别。此外，*Smalltalk*还引入了继承性的思想，它因此一举超越了不可创建实例的程序设计模型和不具备继承性的*Simula*。此外，*Simula 67*的思想亦被应用在许多不同的语言，如*Lisp*、*Pascal*。

面向对象程序设计的80年代成为了一种主导思想，这主要应归功于C++——C语言的扩充版。在图形用户界面(*GUI*)日渐崛起的情况下，面向对象程序设计很好地适应了潮流。*GUI*和面向对象程序设计的紧密关联在*Mac OS X*中可见一斑。*Mac OS X*是由*Objective-C*语言写成的，这一语言是一个仿*Smalltalk*的C语言扩充版。面向对象程序设计的思想也使事件处理式的程序设计更加广泛被应用（虽然这一概念并非仅存在于面向对象程序设计）。一种说法是，*GUI*的引入极大地推动了面向对象程序设计的发展。

苏黎世联邦理工学院的尼克劳斯·维尔特和他的同事们对抽象数据和模块化程序设计进行了研究。*Modula-2*将这些都包括了进去，而*Oberon*则包括了一种特殊的面向对象方法——不同于*Smalltalk*与C++。

面向对象的特性也被加入了当时较为流行的语言：*Ada*、*BASIC*、*Lisp*、*Fortran*、*Pascal*以及种种。由于这些语言最初并没有面向对象的设计，故而这种糅合常常会导致兼容性和维护性的问题。与之相反的是，“纯正的”面向对象语言却缺乏一些程序员们赖以生存的特性。在这一大环境下，开发新的语言成为了当务之急。作为先行者，*Eiffel*成功地解决了这些问题，并成为了当时较受欢迎的语言。

在过去的几年中，*Java*语言成为了广为应用的语言，除了它与C和C++语法上的近似性。*Java*的可移植性是它的成功中不可磨灭的一步，因为这一特性，已吸引了庞大的程序员群的投入。

在最近的计算机语言发展中，一些既支持面向对象程序设计，又支持面向过程程序设计的语言悄然浮出水面。它们中的佼佼者有Python、Ruby等等。

正如面向过程程序设计使得结构化程序设计的技术得以提升，现代的面向对象程序设计方法使得对设计模式的用途、契约式设计和建模语言（如UML）技术也得到了一定提升。

列位看官，当您阅读到这句话的时候，我就姑且认为您已经对面向对象有了一个模糊的认识了。那么，类和OOP有什么关系呢？

[维基百科](#)中这样定义了类：

在面向对象程序设计，类（*class*）是一种面向对象计算机编程语言的构造，是创建对象的蓝图，描述了所创建的对象共同的属性和方法。

类的更严格的定义是由某种特定的元数据所组成的内聚的包。它描述了一些对象的行为规则，而这些对象就被称为该类的实例。类有接口和结构。接口描述了如何通过方法与类及其实例互操作，而结构描述了一个实例中数据如何划分为多个属性。类是与某个层的对象的最具体的类型。类还可以有运行时表示形式（元对象），它为操作与类相关的元数据提供了运行时支持。

支持类的编程语言在支持与类相关的各种特性方面都多多少少有一些微妙的差异。大多数都支持不同形式的类继承。许多语言还支持提供封装性的特性，比如访问修饰符。类的出现，为面向对象编程的三个最重要的特性（封装性，继承性，多态性），提供了实现的手段。

看到这里，看官或许有一个认识，要OOP编程，就得用到类。可以这么说，虽然不是很严格。但是，反过来就不能说了。不是说用了类就一定是OOP。

编写类

对类的理解，需要看官有一定的抽象思维。因为类（Class）本身所定义的是某事物的抽象特点。例如定义一个类：

```
1. class Human:           #这是定义类的方法，通常类的名称用首字母大写的单词或者单词拼接
    pass
2. 
```

好，现在就从这里开始，编写一个类，不过这次我们暂时不用python，而是用伪代码，当然，这个代码跟python相去甚远。如下：

```
1. class Human:
2.     四肢
3.     性格
4.     爱好
5.     学习()
```

对象（Object）是类的实例。刚才已经定义了一个名字为Human的类，从而定义了世界上所有的Human，但是这是一个抽象的Human，不是具体某个人。而对一个具体的人，他的四肢特点、性格、爱好等都是具体的，这些东西在这里被称之为属性。

下面就找一个具体的人：王二麻子，把上面的类实例化。

```
1. 王二麻子 = Human()
2. 王二麻子.四肢 = 修长
3. 王二麻子.爱好 = 看MM
```

在这里，王二麻子就是Human这个类的一个实例。一个具体对象属性的值被称作它的“状态”。（系统给对象分配内存空间，而不会给类分配内存空间，这很好理解，类是抽象的系统不可能给抽象的东西分配空间，对象是具体的）

行文至此，看官是不是大概对类有了一个模糊的认识了呢？

鉴于类，距离我们的直观感觉似乎有点远。所以，要慢慢道来。本讲内容不多，盼望看官能理解。

编写类之一创建实例

- 编写类之一创建实例
 - 创建类
 - 类和实例
 - self的作用
 - 构造函数的参数

编写类之一创建实例

虽然已经对类有了一点点模糊概念，但是，阅读前面一讲的内容的确感到累呀，都是文字，连代码都没有。

本讲就要简单多了，尝试走一个类的流程。

说明：关于类的这部分，我参考了《Learning Python》一书的讲解。

创建类

创建类的方法比较简单，如下：

```
1. class Person:
```

注意，类的名称一般用大写字母开头，这是惯例。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。既然大家都是靠右走的，你就别非要在路中间睡觉了。

接下来，一般都要编写构造函数，在写这个函数之前，先解释一下什么是构造函数。

```
1. class Person:
```



```
2.     def __init__(self, name, lang, website):
3.         self.name = name
4.         self.lang = lang
5.         self.website = website
```

上面的类中，首先呈现出来的是一个名为：`__init__()` 的函数，注意，这个函数是以两个下划线开始，然后是init，最后以两个下划线结束。这是一个函数，就跟我们此前学习过的函数一样的函数。但是，这个函数又有点奇特，它的命名是用“__”开始和结束。

请看官在这里要明确一个基本概念，类就是一种对象类型，和跟前面学习过的数值、字符串、列表等等类型一样。比如这里构建的类名字叫做Person，那么就是我们要试图建立一种对象类型，这种类型被称之为Person，就如同有一种对象类型是list一样。

在构建Person类的时候，首先要做的就是对这种类型进行初始化，也就是要说明这种类型的基本结构，一旦这个类型的对象被调用了，第一件事情就是要运行这个类型的基本结构，也就是类Person的基本结构。就好比我们每个人，在头脑中都有关于“人”这样一个对象类型（对应着类），一旦遇到张三（张三是一个具体人），我们首先运行“人”这个类的基本结构：一个鼻子两只眼，鼻子下面一张嘴。如果张三符合这个基本机构，我们不会感到惊诧（不报错），如果张三不符合这个基本结构（比如三只眼睛），我们就会感到惊诧（报错了）。

由于类是我们自己构造的，那么基本结构也是我们自己手动构造的。在类中，基本结构是写在 `__init__()` 这个函数里面。故这个函数称为构造函数，担负着对类进行初始化的任务。

还是回到Person这个类，如果按照上面的代码，写好了，是不是 `__init__()` 就运行起来了呢？不是！这时候还没有看到张三呢，必须看到张三才能运行。所谓看到张三，看到张三这样一个具体的实实在

在的人，此动作，在python中有一个术语，叫做实例化。当类Person实例化后立刻运行 `__init__()` 函数。

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  class Person:
5.      def __init__(self, name, lang, website):
6.          self.name = name
7.          self.lang = lang
8.          self.website = website
9.
10. info = Person("qiwsir", "python", "qiwsir.github.io")    #实例化
    Person
11. print "info.name=", info.name
12. print "info.lang=", info.lang
13. print "info.website=", info.website
14.
15. #上面代码的运行结果：
16.
17. info.name= qiwsir
18. info.lang= python
19. info.website= qiwsir.github.io

```

在上面的代码中，建立的类Person，构造函数声明了这个类的基本结构：name, lang, website。

注意观察：

info=Person("qiwsir", "python", "qiwsir.github.io"),
这句话就是将类Person实例化了。也就是在内存中创建了一个对象，这个对象的类型是Person类型，这个Person类型是什么样子的呢？就是 `__init__()` 所构造的那样。在实例化时，必须通过参数传入具体的数据：

name="qiwsir", lang="python", website="qiwsir.github.i

o”。这样在内存中就存在了一个对象，这个对象的类型是Person，然后通过赋值语句，与变量info建立引用关系。请看官回忆以前已经讲述过的变量和对象的引用关系。

看官是不是有点晕乎了？类、实例，这两个概念会一直伴随着后续的学习，并且，在很多的OOP模型中，都会遇到这两个概念。为了让看官不晕乎，这里将它们进行比较（注意：比较的内容，参考了《Learning Python》一书）

类和实例

- “类提供默认行为，是实例的工厂”，我觉得这句原话非常经典，一下道破了类和实例的关系。看上面代码，体会一下，是不是这个理？所谓工厂，就是可以用同一个模子做出很多具体的产品。类就是那个模子，实例就是具体的产品。所以，实例是程序处理的实际对象。
- 类是由一些语句组成，但是实例，是通过调用类生成，每次调用一个类，就得到这个类的新的实例。
- 对于类的：`class Person`，`class`是一个可执行的语句。如果执行，就得到了一个类对象，并且将这个类对象赋值给对象名（比如Person）。

也许上述比较还不足以让看官理解类和实例，没关系，继续学习，在前进中排除疑惑。

self的作用

细心的看官可能注意到了，在构造函数中，第一个参数是self，但是在实例化的时候，似乎没有这个参数什么事儿，那么self是干什么的呢？

self是一个很神奇的参数。

在Person实例化的过程中，数据"qiwsir","python","qiwsir.github.io"通过构造函数（`__init__()`）的参数已经存入到内存中，并且这些数据以Person类型的面貌存在组成一个对象，这个对象和变量info建立的引用关系。这个过程也可说成这些数据附加到一个实例上。这样就能够以`object.attribute`的形式，在程序中任何地方调用某个数据，例如上面的程序中以`info.name`得到"qiwsir"这个数据。这种调用方式，在类和实例中经常使用，点号`"."`后面的称之为类或者实例的属性。

这是在程序中，并且是在类的外面。如果在类的里面，想在某个地方使用传入的数据，怎么办？

随着学习的深入，看官会发现，在类内部，我们会写很多不同功能的函数，这些函数在类里面有另外一个名称，曰：方法。那么，通过类的构造函数中的参数传入的这些数据也想在各个方法中被使用，就需要在类中长久保存并能随时调用这些数据。为了解决这个问题，在类中，所有传入的数据都赋给一个变量，通常这个变量的名字是`self`。注意，这是习惯，而且是共识，所以，看官不要另外取别的名字了。

在构造函数中的第一个参数`self`，就是起到了这个作用——接收实例化过程中传入的所有数据，这些数据是通过构造函数后面的参数导入的。显然，`self`应该就是一个实例（准确说法是应用实例），因为它所对应的就是具体数据。

如果将上面的类增加两句，看看效果：

```
1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
```

```

4. class Person:
5.     def __init__(self, name, lang, website):
6.         self.name = name
7.         self.lang = lang
8.         self.website = website
9.
10.        print self          #打印, 看看什么结果
11.        print type(self)
12.
13. #运行结果
14. <__main__.Person instance at 0xb74a45cc>
15. <type 'instance'>

```

证实了推理。self就是一个实例（准确说是实例的引用变量）。

self这个实例跟前面说的那个info所引用的实例对象一样，也有属性。那么，接下来就规定其属性和属性对应的数据。上面代码中：self.name = name，就是规定了self实例的一个属性，这个属性的名字也叫做name，这个属性的数据等于构造函数的参数name所导入的数据。注意，self.name中的name和构造函数的参数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），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.xxxooo = name，也是可以的。

其实，从效果的角度来理解，可能更简单一些，那就是类的实例info对应着self，info通过self导入实例属性的所有数据。

当然，self的属性数据，也不一定非得是由参数传入的，也可以在构造函数中自己设定。比如：

```

1. #!/usr/bin/env python
2. #coding:utf-8
3.
4. class Person:
5.     def __init__(self, name, lang, website):

```

```

6.         self.name = name
7.         self.lang = lang
8.         self.website = website
9.         self.email = "qiwsir@gmail.com"      #这个属性不是通过参数传入的
10.
11. info = Person("qiwsir","python","qiwsir.github.io")
12. print "info.name=",info.name
13. print "info.lang=",info.lang
14. print "info.website=",info.website
15. print "info.email=",info.email      #info通过self建立实例，并导入实例属性数据
16.
17. #运行结果
18.
19. info.name= qiwsir
20. info.lang= python
21. info.website= qiwsir.github.io
22. info.email= qiwsir@gmail.com      #打印结果

```

通过这个例子，其实让我们拓展了对self的认识，也就是它不仅仅是为了在类内部传递参数导入的数据，还能在构造函数中，通过self.attribute的方式，规定self实例对象的属性，这个属性也是类实例化对象的属性，即做为类通过构造函数初始化后所具有的属性。所以在实例info中，通过info.email同样能够得到该属性的数据。在这里，就可以把self形象地理解为“内外兼修”了。或者按照前面所提到的，将info和self对应起来，self主内，info主外。

其实，self的话题还没有结束，后面的方法中还会出现它。它真的神奇呀。

构造函数的参数

前面已经说过了，构造函数 `__init__` 就是一个函数，只不过长相有点古怪罢了。那么，函数中的操作在构造函数中依然可行。比如：

```

1. def __init__(self, *args):
2.     pass

```

这种类型的参数：*args和前面讲述函数参数一样，就不多说了。忘了的看官，请去复习。但是，self这个参数是必须的，因为它要来建立实例对象。

很多时候，并不是每次都要从外面传入数据，有时候会把构造函数的某些参数设置默认值，如果没有新的数据传入，就应用这些默认值。比如：

```

1. class Person:
2.     def __init__(self, name, lang="golang",
3.         website="www.google.com"):
4.         self.name = name
5.         self.lang = lang
6.         self.website = website
7.         self.email = "qiwsir@gmail.com"
8.
9. laoqi = Person("LaoQi")      #导入一个数据name="LaoQi", 其它默认值
10. info = Person("qiwsir", lang="python", website="qiwsir.github.io")
11.     #全部重新导入数据
12.
13. print "laoqi.name=", laoqi.name
14. print "info.name=", info.name
15. print "-----"
16. print "laoqi.lang=", laoqi.lang
17. print "info.lang=", info.lang
18. print "-----"
19. print "laoqi.website=", laoqi.website
20. print "info.website=", info.website
21.
22. #运行结果
23.
24. laoqi.name= LaoQi
25. info.name= qiwsir

```

```
24.  -----
25.  laoqi.lang= golang
26.  info.lang= python
27.  -----
28.  laoqi.website= www.google.com
29.  info.website= qiwsir.github.io
```

在这段代码中，看官首先要体会一下，“类是实例的工厂”这句话的含义，通过类Person生成了两个实例：laoqi、info

此外，在看函数赋值的情况，允许设置默认参数值。

至此，仅仅是初步构建了一个类的基本结构，完成了类的初始化。

编写类之二方法

- 编写类之二方法
 - 数据流转过程
 - 为什么要用到方法
 - 编写和操作方法

编写类之二方法

[上一讲](#)中创建了类，并且重点讲述了构造函数以及类实例，特别是对那个self，描述了不少。在讲述构造函数的时候特别提到，`init()`是一个函数，只不过在类中有一点特殊的作用罢了，每个类，首先要运行它，它规定了类的基本结构。

数据流转过程

除了在类中可以写这种函数之外，在类中还可以写别的函数，延续上一讲的例子：

```
1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  class Person:
5.      def __init__(self, name, lang="golang",  
      website="www.google.com"):
6.          self.name = name
7.          self.lang = lang
8.          self.website = website
9.          self.email = "qiwsir@gmail.com"
10.
11.     def author(self):
12.         return self.name
13.
```

```

14. laoqi = Person("LaoQi")
15. info = Person("qiwsir", lang="python", website="qiwsir.github.io")
16.
17. print "Author name from laoqi:", laoqi.author()
18. print "Author name from info:", info.author()
19.
20. #运行结果
21.
22. Author name from laoqi: LaoQi
23. Author name from info: qiwsir

```

看官可能已经注意了，这段代码比上一讲多了一个函数 `author(self)`，这个我们先不管，稍后会详细分解。首先看看数据是如何在这个代码中流转的。为了能够清楚，画一张图，所谓一图胜千言万语，有图有真相。



定义类 `Person`，然后创建实例 `laoqi=Person("LaoQi")`，看官注意观察图上的箭头方向。`laoqi` 这个实例和 `Person` 类中的 `self` 对应，它们都是引用了实例对象（很多时候简化说成是实例对象）。“`LaoQi`”是一个具体的数据，通过构造函数中的 `name` 参数，传给实例的属性 `self.name`，在类 `Person` 中的另外一个方法 `author` 的参数列表中第一个就是 `self`，表示要承接 `self` 对象，`return self.name`，就是在类内部通过 `self` 对象，把它的属性 `self.name` 的数据传导如 `author`。

当运行 `laoqi.author()` 的时候，就是告诉上面的代码，调用 `laoqi` 实例对象，并得到 `author()` 方法的结果，`laoqi` 这个实例就自动被告告诉了 `author()`（注意，`self` 参数在这里不用写，这个告诉过程是 `python` 自动完成的，不用我们操心了），`author` 方法就返回 `laoqi` 实例的属性，因为前面已经完成了 `laoqi` 与 `self` 的对应过程，所以这时

候author里面的self就是laoqi，自然self.name=laoqi.name。

看官可以跟随我在做一个实验，那就是在author中，return laoqi.name，看看什么效果。因为既然laoqi和self是同一个实例对象，直接写成laoqi.name是不是也可以呢？

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  class Person:
5.      def __init__(self, name, lang="golang",
6.                  website="www.google.com"):
7.          self.name = name
8.          self.lang = lang
9.          self.website = website
10.         self.email = "qiwsir@gmail.com"
11.
12.     def author(self):
13.         #return self.name
14.         return laoqi.name      #返回
15.
16. laoqi = Person("LaoQi")
17. info = Person("qiwsir", lang="python", website="qiwsir.github.io")
18.
19. print "Author name from laoqi:",laoqi.author()
20. print "Author name from info:",info.author()
21.
22. #输出结果
23. Author name from laoqi: LaoQi    #laoqi实例输出结果
24. Author name from info: LaoQi    #info实例输出结果

```

从结果中可以看出，没有报错。但是，info这个实例输出的结果和laoqi实例输出的结果一样。原来，当调用了info实例之后，运行到author()，返回的是laoqi.name。所以，这里一定要用self实例。在调用不同的实例时，self会自动的进行匹配，当然，匹配过程是

python完成，仍然不用我们操心。

OK，数据流转过程，看官是否理解了呢？下面进入方法编写的环节

为什么要用到方法

在类里面，可以用def语句来编写函数，但是，通常这个函数的样子是这样的：

```
1. class ClassName:
2.     def __init__(self, *args):
3.         ...
4.     def method(self, *args):      #是一个在类里面的函数
5.         ...
```

在类ClassName里面，除了前面那个具有初始化功能的构造函数之外，还有一个函数method，这个函数和以前学习过的函数一样，函数里面要写什么，也没有特别的规定。但是，这个函数的第一个参数必须是self，或者说，可以没有别的参数，但是self是必须写上并且是第一个。这个self参数的作用前面已经说过了。

这样看来，类里面的这个函数还有点跟以前函数不同的地方。

类里面的这个函数，我们就称之为方法。

之所以用方法，也是用类的原因，也是用函数的原因，都是为了减少代码的冗余，提高代码的重用性，这也是OOP的原因。

方法怎样被重用呢？看本最开始的那段代码，里面有一个author方法，不管是laoqi还是info实例，都用这个方法返回实例导入的名字。这就是体现了重用。

编写和操作方法

编写方法的过程和编写一个函数的过程一样，需要注意的就是要在参数列表中第一个写上self，即使没有其它的参数。

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  class Person:
5.      def __init__(self, name, lang="golang",
6.                  website="www.google.com"):
7.          self.name = name
8.          self.lang = lang
9.          self.website = website
10.         self.email = "qiwsir@gmail.com"
11.
12.     def author(self, address):
13.         #return self.name
14.         return self.name+" in "+address
15.
16.     info = Person("qiwsir", lang="python", website="qiwsir.github.io")
17.
18.     print "Author name from laoqi:",laoqi.author("China")
19.     print "Author name from info:",info.author("Suzhou")
20.
21.     #运行结果
22.
23.     Author name from laoqi: LaoQi in China
24.     Author name from info: LaoQi in Suzhou

```

这段代码中，对author方法增加了一个参数address，当调用这个方法的时候：laoqi.author("China")，要对这个参数赋值，看官特别注意，在类中，这个方法显示是有两个参数(self,address)，但是在调用的时候，第一个参数是自动将实例laoqi与之对应起来，不需要显化赋值，可以理解成是隐含完成的（其实，也可以将laoqi看做隐藏的主体，偷偷地跟self勾搭上了）。

通过上面的讲述，看官可以试试类了。提醒，一定要对类的数据流通过程清晰。

编写类之三子类

- 编写类之三子类
 - 子类、父类和继承

编写类之三子类

关于类，看官想必已经有了感觉，看下面的代码，请仔细阅读，并看看是否能够发现点什么问题呢？

```
1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  class Person:
5.      def __init__(self, name, lang, email):
6.          self.name = name
7.          self.lang = lang
8.          self.email = email
9.
10.     def author(self):
11.         return self.name
12.
13.     class Programmer:
14.         def __init__(self, name, lang, email, system, website):
15.             self.name = name
16.             self.lang = lang
17.             self.email = email
18.             self.system = system
19.             self.website = website
20.
21.         def pythoner(self):
22.             pythoner_list = [ self.name, self.lang, self.email,
23.                               self.system, self.website ]
24.             return pythoner_list
25.     if __name__=="__main__":
```

```

26.     writer = Person("qiwsir", "Chinese", "qiwsir@gmail.com")
27.     python =
        Programmer("qiwsir", "Python", "qiwsir@gmail.com", "Ubutun", "qiwsir.github")
28.     print "My name is:%s"%writer.author()
29.     print "I write program by:%s"%python.pythoner()[1]

```

上面这段代码，运行起来没有什么问题，但是，仔细看，发现有两个类，一个名字叫做Person，另外一个叫做Programmer，这还不是问题所在，问题所在是这两个类的构造函数中，存在这相同的地方：
self.name=name, self.lang=lang, self.email=email, 这对于追求代码质量的程序员，一般是不允许的。最好不要有重复代码或者冗余代码。可是，在两个类中都要有这些参数，应该怎么办呢？

子类、父类和继承

看下面的代码，里面有两个类A，B。这段程序能够正确运行，每个类的功能是仅仅打印指定的内容。

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  class A:
5.      def __init__(self):
6.          print "aaa"
7.
8.  class B:
9.      def __init__(self):
10.         print "bbb"
11.
12.  if __name__=="__main__":
13.      a = A()
14.      b = B()
15.
16.  #运行结果

```



```

17.  aaa
18.  bbb

```

上面的两个类彼此之间没有所谓的父子关系。现在稍加改变，将类B改写，注意观察与上面的差异。

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  class A:
5.      def __init__(self):
6.          print "aaa"
7.
8.  class B(A):          #这里和上面程序不同。B继承了A
9.      def __init__(self):
10.         print "bbb"
11.
12.  if __name__=="__main__":
13.      a = A()
14.      b = B()
15.
16.  #运行结果
17.  aaa
18.  bbb

```

这段程序中，类B跟前面的那段有一点不同，`class B(A):`，这样写就表明了B相对A的关系：B是A的子类，B从A继承A的所有东西（子承父业）。

但是，看官发现了没有，运行结果一样。是的，那是以为在B中尽管继承了A，但是没有调用任何A的东西，就好比儿子从老爸那里继承了财富，但是儿子一个子也没动，外界看到的和没有继承一样。

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.

```

```

4. class A:
5.     def __init__(self):
6.         print "aaa"
7.
8. class B(A):
9.     def __init__(self):
10.        #print "bbb"
11.        A.__init__(self)    #运行继承的父类
12.
13. if __name__=="__main__":
14.     a = A()
15.     b = B()
16.
17. #运行结果
18. aaa
19. aaa

```

这回运行结果有了变化，本来**b=B()**是运行类B，但是B继承了A，并且在初始化的构造函数中，引入A的构造函数，所以，就运行A的结果相应结果了。

下面把最开头的那端程序用子类继承的方式重写，可以是这样的：

```

1. #!/usr/bin/env python
2. #coding:utf-8
3.
4. class Person:
5.     def __init__(self, name, lang, email):
6.         self.name = name
7.         self.lang = lang
8.         self.email = email
9.
10.    def author(self):
11.        return self.name
12.    """
13. class Programmer:
14.     def __init__(self, name, lang, email, system, website):

```

```

15.         self.name = name
16.         self.lang = lang
17.         self.email = email
18.         self.system = system
19.         self.website = website
20.
21.     def pythoner(self):
22.         pythoner_list = [ self.name, self.lang, self.email,
23.                             self.system, self.website ]
24.         return pythoner_list
25.
26. class Programmer(Person):          #继承父类Person
27.     def __init__(self, name, lang, email, system, website):
28.         Person.__init__(self, name, lang, email)    #将
29.         #self.name = name                        #这三句是Person中已经
30.         #self.lang = lang                        #通过继承已经实现了这三
31.         #self.email = email
32.         self.system = system                    #子类中不同于Person父
33.         self.website = website
34.
35.     def pythoner(self):
36.         pythoner_list = [ self.name, self.lang, self.email,
37.                             self.system, self.website ]
38.         return pythoner_list
39.
40. if __name__ == "__main__":
41.     writer = Person("qiwsir", "Chinese", "qiwsir@gmail.com")
42.     python =
43.         Programmer("qiwsir", "Python", "qiwsir@gmail.com", "Ubutun", "qiwsir.github")
44.     print "My name is:%s"%writer.author()
45.     print "I write program by:%s"%python.pythoner()[1]

```

代码运行结果与前面一样。

列位是否理解了子类和父类、继承的特点。如果你有一个老爹，是一个高官或者富豪，那么你就官二代或者富二代了，你就从他们那里继承了很多财富，所以生活就不用太劳累了。这就是继承的作用。在代码中，也类似，继承能够让写代码的少劳累一些。

需要提供注意的是，在子类中，如果要继承父类，必须用显明的方式将所继承的父类方法写出来，例如上面的

`Person.init(self, name, lang, email)`，必须这样写，才能算是在子类中进行了继承。如果不写上，是没有继承的。用编程江湖的黑话（比较文雅地称为“行话”）说就是“显式调用父类方法”。

对于为什么要用继承，好友@令狐虫 大侠给了以非常精彩的解释：

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，这是不一定的。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。

我表示完全赞同上述解释。不过看官如果不理解，也没有关系，上述解释中的精神，的确需要在编程实践中感悟才能领会到的。

编写类之四再论继承

- 编写类之四再论继承
 - 多余的B
 - 其它方法的继承

编写类之四再论继承

在上一讲代码的基础上，做进一步修改，成为了如下程序，请看官研习这个程序：

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  class Person:
5.      def __init__(self, name, email):
6.          self.name = name
7.          self.email = email
8.
9.  class Programmer(Person):
10.     def __init__(self, name, email, lang, system, website):
11.         Person.__init__(self, name, email)
12.         self.lang = lang
13.         self.system = system
14.         self.website = website
15.
16.  class Pythoner(Programmer):
17.     def __init__(self, name, email):
18.
19.         Programmer.__init__(self, name, email, "python", "Ubuntu", "qiwsir.github.
20.
21.  if __name__=="__main__":
22.     writer = Pythoner("qiwsir", "qiwsir@gmail.com")
23.     print "name=", writer.name
24.     print "lang=", writer.lang

```

```

24.     print "email=",writer.email
25.     print "system=",writer.system
26.     print "website=",writer.website
27.
28. #运行结果
29.
30. name= qiwsir
31. lang= python
32. email= qiwsir@gmail.com
33. system= Ubuntu
34. website= qiwsir.github.io

```

对结果很满意，再看程序中的继承关系：Pythoner ← Programmer ← Person，从上面的过程中不难看出，继承能够减少代码重复，是代码更简练。另外，在继承的时候，也可以在函数中对参数进行默认赋值。

为了能够突出继承问题的探究，还是用那种简单的类来做实验。

多余的B

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  class A:
5.      def __init__(self):
6.          print "aaa"
7.
8.  class B(A):
9.      pass
10.
11. if __name__=="__main__":
12.     a = A()
13.     b = B()
14.

```

```

15. #运行结果
16.
17. aaa
18. aaa

```

B继承A，没有任何修改地继承，B就可以不用写任何东西了，或者说B本质上就是一个多余。在真实的编程过程中，没有这样写的，这里仅仅是为了向看官展示一下继承的含义罢了。

```

1.  ##首个继承有效
2.
3.  #!/usr/bin/env python
4.  #coding:utf-8
5.
6.  class A:
7.      def __init__(self):
8.          print "aaa"
9.
10. class B:
11.     def __init__(self):
12.         print "bbb"
13.
14. class C1(A,B):
15.     pass
16.
17. class C2(B,A):
18.     pass
19.
20. if __name__=="__main__":
21.     print "A--->",
22.     a = A()
23.     print "B--->",
24.     b = B()
25.     print "C1(A,B)--->",
26.     c1 = C1()
27.     print "C2(B,A)--->",
28.     c2 = C2()

```

```

29.
30.  #运行结果
31.
32.  A---> aaa
33.  B---> bbb
34.  C1(A,B)---> aaa
35.  C2(B,A)---> bbb

```

列位看官是否注意了，类C1继承了两个类A，B；类C2也继承了两个类，只不过书写顺序有点区别(B,A)。从运行结果可以看出，当子类继承多个父类的时候，对于构造函数 `__init__()`，只有第一个能够被继承，第二个就等掉了。所以，一般情况下，不会在程序中做关于构造函数的同时多个继承，不过可以接力继承，就如同前面那个比较真实的代码一样。

其它方法的继承

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  class A:
5.      def __init__(self):
6.          print "aaa"
7.      def amethod(self):
8.          print "method a"
9.
10. class B(A):
11.     def __init__(self):
12.         print "bbb"
13.
14.
15. if __name__=="__main__":
16.     print "A--->"
17.     a = A()
18.     a.amethod()

```



```

19.     print "B--->"
20.     b = B()
21.     b.amethod()
22.
23. #运行结果
24.
25. A--->
26. aaa
27. method a
28. B--->
29. bbb
30. method a

```

为了说明白上面的情况，还是画了一张图，不过，我画完之后，就后悔了，看这张图好像更糊涂了。怎么着也画了，还是贴出来，如果能够协助理解更好了。



A的实例和调用，就不多说了。重点看B，类B继承了A，同时，B在构造函数中自己做了规定，也就是B的构造函数是按照B的意愿执行，不执行A的内容，但是，A还有一个amethod(self)方法，B则继承了这个方法。当通过类B的实例调用这个方法的时候，就能够成功了：

```
b.amethod()
```

这就是方法的继承和调用方法。

所谓继承，就是从下到上一级一级地找相应的继承对象，找到了就继承之。如果有同名的怎么办？按照什么顺序找呢？

应用网上的一段：

在Python中，可以进行多重继承，这个时候要注意搜寻的顺序，是从子类别开始，接著是同一阶层父类别由左至右搜寻，再至更上层同一阶层父类别由左至右搜寻，直到达到顶层为止。

代码举例：

```

1. class A(object):
2.     def method1(self):
3.         print('A.method1')
4.
5.     def method2(self):
6.         print('A.method2')
7.
8. class B(A):
9.     def method3(self):
10.        print('B.method3')
11.
12. class C(A):
13.     def method2(self):
14.         print('C.method2')
15.
16.     def method3(self):
17.         print('C.method3')
18.
19. class D(B, C):
20.     def method4(self):
21.         print('C.method4')
22.
23. d = D()
24. d.method4() # 在 D 找到, C.method4
25. d.method3() # 以 D->B 顺序找到, B.method3
26. d.method2() # 以 D->B->C 顺序找到, C.method2
27. d.method1() # 以 D->B->C->A 顺序找到, A.method1

```

务必请真正的学习者要对照每个类的每个方法，依次找到相应的输出结果。从而理解继承的顺序。学习，就要点滴积累。

命名空间

- 命名空间
 - 什么是命名空间
 - 作用域

命名空间

命名空间，英文名字：namespaces

在研习命名空间以前，请打开在python的交互模式下，输入：import this

```
1. >>> import this
2. The Zen of Python, by Tim Peters
3.
4. Beautiful is better than ugly.
5. Explicit is better than implicit.
6. Simple is better than complex.
7. Complex is better than complicated.
8. Flat is better than nested.
9. Sparse is better than dense.
10. Readability counts.
11. Special cases aren't special enough to break the rules.
12. Although practicality beats purity.
13. Errors should never pass silently.
14. Unless explicitly silenced.
15. In the face of ambiguity, refuse the temptation to guess.
16. There should be one-- and preferably only one --obvious way to do
    it.
17. Although that way may not be obvious at first unless you're Dutch.
18. Now is better than never.
19. Although never is often better than *right* now.
20. If the implementation is hard to explain, it's a bad idea.
21. If the implementation is easy to explain, it may be a good idea.
```

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

这里列位看到的就是所谓《python之禅》，在本教程的[第零部分：唠叨一些关于Python的事情](#)有专门的翻译，这里再次列出，是要引用其中的一句话。请看官看最后一句：Namespaces are one honking great idea – let's do more of those!

这是为了向看官说明Namespaces、命名空间值重要性。

什么是命名空间

从“一切皆为对象”开始说起吧。对象，很多时候我们直接使用它并不方便，因此要给它取一个名字。打个比方，有这样一个物种，它是哺乳纲灵长目人科人属智人种，这就是所谓的对象，但是，在平时提及这个对象的时候，总是要说“哺乳纲灵长目人科人属智人种”，是不是太麻烦了？于是聪明的这个物种就为这个世界上的各种对象命名，例如将“哺乳纲灵长目人科人属智人种”这个对象命名为“人”。

在编程中也是如此，前面在讲述变量相关知识的时候已经说明了变量和引用对象的关系。

```
1. >>> a = 7
2. >>> id(7)
3. 137589400
4. >>> id(a)
5. 137589400
6. >>> id(7)==id(a)
7. True
```

看这个例子。7就是一个计算机内存中存在的对象，用id()这个内置函数可以查看7在内存（在RAM）中的地址。a 就是为这个对象预备的名字，如前面所讲的，它与内存中的一个编号为137589400的对象关

联，或者说引用了这个对象，这个对象就是7。

如果做了下面的操作：

```
1. >>> a = a+1
2. >>> id(a)
3. 137589388
4. >>> a
5. 8
6. >>> id(8)
7. 137589388
```

其实，上面操作中的a+1完成的是a引用的对象7+1，只不过是顺着对象7的命名a导入了对象7罢了，这样就在内存中建立了一个新的对象8，同样通过id()函数查看到内存中的地址，通过地址可以看到，这时候的a又自动引用对象8了。

```
1. >>> id(7)    #对象7在内存中的地址没变
2. 137589400
3. >>> b = 7    #b引用此对象
4. >>> id(b)
5. 137589400
```

上面a转换引用对象的过程，是自动完成的。而当b=7的时候，并不是在内存中从新建立一个对象7,而是b引用了已有的对象。这就是python的所谓动态语言的特点。



当然，可以给任何对象取名字，或者说为任何对象都可以建立一个所引用的变量。比如函数、类都可以，此处不赘述，前面已经多次用到了。

现在已经又一次明确了，每个名称（命名）——英文中的NAME有动词和名字两种，所以，由于中文的特点，似乎怎么说都可以，只要明白所

指，因为中文是强调语境的语言——都与某个对象有对应关系。那么所谓的命名空间，就是这些命名（名称）的集合，它们分别与相应的对象有对应关系。

用一句比较学术化的语言说：

命名空间是从所定义的命名到对象的映射集合。

不同的命名空间，可以同时存在，当彼此相互独立互不干扰。

命名空间因为对象的不同，也有所区别，可以分为如下几种：

- 内置命名空间(Built-in Namespaces)：Python运行起来，它们就存在了。内置函数的命名空间都属于内置命名空间，所以，我们可以在任何程序中直接运行它们，比如前面的`id()`，不需要做什么操作，拿过来就直接使用了。
- 全局命名空间(Module:Global Namespaces)：每个模块创建它自己所拥有的全局命名空间，不同模块的全局命名空间彼此独立，不同模块中相同名称的命名空间，也会因为模块的不同而不相互干扰。
- 本地命名空间(Function&Class: Local Namespaces)：模块中有函数或者类，每个函数或者类所定义的命名空间就是本地命名空间。如果函数返回了结果或者抛出异常，则本地命名空间也结束了。

从网上盗取了一张图，展示一下上述三种命名空间的关系



那么程序在查询上述三种命名空间的时候，就按照从里到外的顺序，即：Local Namespaces → Global Namespaces → Built-in Namespaces

还要补充说一下，既然命名空间中存在着命名和对象的映射，不知道看官看到这句话能想到什么？启发一下，回忆以往学过的那种类型数据也存在对应关系呢？字典，就是那个dictionary，是“键值”对应的，例如：`{"name": "qiwsir", "lang": "python"}`

```
1. >>> def foo(num, str):
2. ...     name = "qiwsir"
3. ...     print locals()
4. ...
5. >>> foo(221, "qiwsir.github.io")
6. {'num': 221, 'name': 'qiwsir', 'str': 'qiwsir.github.io'}
7. >>>
```

这是一个访问本地命名空间的方法，用`print locals()`完成，从这个结果中不难看出，所谓的命名空间中的数据存储结构和dictionary是一样的。

根据习惯，看官估计已经猜测到了，如果访问全局命名空间，可以使用`print globals()`。

作用域

作用域是指 Python 程序可以直接访问到的命名空间。“直接访问”在这里意味着访问命名空间中的命名时无需加入附加的修饰符。（这句话是从网上抄来的）

程序也是按照搜索命名空间的顺序，搜索相应空间的能够访问到的作用域。

```
1. def outer_foo():
2.     b = 20
3.     def inner_foo():
4.         c = 30
```

```
5. a = 10
```

加入我现在位于`inner_foo()`函数内，那么`c`对我来讲就在本地作用域，而`b`和`a`就不是。如果我在`inner_foo()`内再做：`b=50`，这其实是在本地命名空间内新创建了对象，和上一层中的`b=20`毫不相干。可以看下面的例子：

```
1. #!/usr/bin/env python
2. #coding:utf-8
3.
4. def outer_foo():
5.     a = 10
6.     def inner_foo():
7.         a = 20
8.         print "inner_foo,a=", a      #a=20
9.
10.    inner_foo()
11.    print "outer_foo,a=", a          #a=10
12.
13. a = 30
14. outer_foo()
15. print "a=", a                      #a=30
16.
17. #运行结果
18.
19. inner_foo,a= 20
20. outer_foo,a= 10
21. a= 30
```

如果要将某个变量在任何地方都使用，且能够关联，那么在函数内就使用`global` 声明，其实就是曾经讲过的全局变量。请参考《[变量和参数](#)》

类的细节

- 类的细节
 - 绑定和无绑定方法
 - 调用绑定实例方法对象
 - 调用无绑定类方法对象
 - 文档字符串

类的细节

前面对类的有关内容已经描述不少了，其实话题远远没有结束，不过对于初学者，掌握这些已经算是入门，在以后的实践中，还需要进行体会和感悟。

这几天和几个朋友以各种途径讨论过OOP的相关问题，他们是：令狐虫、Frank、晋剑、小冯

大家对OOP有不同看法，所谓工程派和学院派看法不一致。从应用的角度看，工程派的观点是值得推荐的，那就是：不用太在意内部是怎么工作的，只要能够解决眼下的问题即可。但是，对于学习者而言，如果仅仅停留在工程派的层面（特别提醒，上述几位朋友都是工程派的大侠，他们可不是简单地能够使用，其实是更高层次的“无招胜有招”），学习者可能感觉有点不透彻。所以，学习者，特别是初学者，要知道一些内部原因，但是也别为了钻研内部原因而忘记了应用的目的。看来两者协调还是一个难办的事情。不用着急，随着实践的深入，就逐渐有体会了。

下面我根据MARK Lutz的《Learning Python》中的“大师眼中的OOP”，列一些使用OOP的常见原因。

- 代码重用。这是很简单（并且是使用OOP的最主要原因）。通过支

持继承，类允许通过定制来编程，而不是每次都从头开始一个项目。

- 封装。在对象接口后包装其实现的细节，从而隔离了代码的修改对用户产生的影响。
- 结构。类提供了一个新的本地作用域，最小化了变量名冲突。他们还提供了一种编写和查找实现代码，以及去管理对象状态的自然场所。
- 维护性。类自然而然地促进了代码的分解，这让我们减少了冗余。对亏支持类的结构以及代码重用，这样每次只需要修改代码中一个拷贝就可以了。
- 一致性。类和继承可以实现通用的接口。这样代码不仅有了统一的外表和观感，还简化了代码的调试、理解以及维护。
- 多态。多态让代码更灵活和有了广泛的适用性。（这似乎是OOP的属性，不是使用它的理由）

不管怎么样，类是一个非常重要的东西，看官在学习的时候，一定要多加运用。

此外，对于python2来说，还有一个叫做“新式类”(new-style)的东西，这个对应于前面讲过的类，那么前面讲过的类就称为“经典”(classic)类。但是，对于Python3来讲，没有这种区别，二者融合。只是在Python2中，两个是有区别的。本教程在基础部分，依然不讲授新式类的问题，如果看官有兴趣，可以自己在GOOGLE中查找有关资料，也可以随着本课程深入，到下一个阶段来学习。

绑定和无绑定方法

看官是否还记得，在学习类的方法的时候，提到过，类的方法就是函数，只不过这个函数的表现有点跟前面学过的函数不一样，比如有个self。当然，也不是必须要有的，下面看官就会看到没有self的。既

然方法和函数一样，本质上都是函数，那么，函数那部分学习的时候已经明确了：函数是对象，所以，类方法也是对象。正如刚才说的，类的方法中，有的可以有self，有的可以没有。为了进行区别，进一步做了这样的定义：

- 无绑定类方法对象：无self
- 绑定实例方法对象：有self

调用绑定实例方法对象

```
1. >>> class MyClass:
2. ...     def foo(self,text):
3. ...         print text
4. ...
```

可以用下面的方式调用实例方法

```
1. >>> a = MyClass()           #创建类实例
2. >>> a.foo('qiwsir.github.io') #调用实例方法
3. qiwsir.github.io
4. >>> a.foo
5. <bound method MyClass.foo of <__main__.MyClass instance at
   0xb74495ac>>
```

在这个实例方法调用的时候，其数据传递流程，在《编写类之二方法》中有一张图，图中显示了，上述的调用方法中，其实已经将实例名称a传给了self，这就是调用绑定实例方法对象，有self。

上面的调用过程，还可以这样来实现：

```
1. >>> a = MyClass()
2. >>> x = a.foo           #把实例a和方法函数foo绑定在一起
3. >>> x
4. <bound method MyClass.foo of <__main__.MyClass instance at
```

```
0xb74495ac>>
5. >>> x("qiwsir.github.io")
6. qiwsir.github.io
```

在上面的调用中，其实相当于前面的调用过程的分解动作。即先将实例a和方法函数foo绑定在一起，然后赋值给x，这时候x就相当于一个简单函数一样，可以通过上述方式传入参数。这里将实例和方法函数绑定的方式就是运用点号运算（object.method_function）

调用无绑定类方法对象

所谓类方法对象，就是不通过实例，而是用类进行点号运算来获得方法函数（ClassName.method_function）

```
1. >>> a = MyClass()
2. >>> y = MyClass.foo      #这里没有用类调用
3. >>> y
4. <unbound method MyClass.foo>
```

这样的调用，就得到了无绑定方法对象，但是，调用的时候必须传入实例做为第一参数，如下

```
1. >>> y(a, "qiwsir.github.io")
2. qiwsir.github.io
```

否则，就报错。请看官特别注意报错信息

```
1. >>> y("qiwsir.github.io")
2. Traceback (most recent call last):
3.   File "<stdin>", line 1, in <module>
4. TypeError: unbound method foo() must be called with MyClass
   instance as first argument (got str instance instead)
5. >>>
```

在编程实践中，似乎用实例方法调用更多一下。

文档字符串

在写程序的时候，必须要写必要的文字说明，没别的原因，除非你的代码写的非常容易理解，特别是各种变量、函数和类等的命名任何人都能够很容易理解，否则，文字说明是不可缺少的。

在函数、类或者文件开头的部分写文档字符串说明，一般采用三重引号。这样写的最大好处是能够用`help()`函数看。

```
1.  """This is python lesson"""
2.
3.  def start_func(arg):
4.      """This is a function."""
5.      pass
6.
7.  class MyClass:
8.      """Thi is my class."""
9.      def my_method(self,arg):
10.         """This is my method."""
11.         pass
```

这样的文档是必须的。

当然，在编程中，有不少地方要用“#”符号来做注释。一般用这个来注释局部。

类其实并没有结束，不过本讲座到此对类暂告一段。看官要多实践。

Import 模块

- Import 模块
 - 认识模块
 - 标准库
 - 自己编写模块

Import 模块

认识模块

对于模块，在前面的一些举例中，已经涉及到了，比如曾经有过：
`import random`（获取随机数模块）。为了能够对模块有一个清晰的了解，首先要看看什么模块，这里选取[官方文档](#)中对它的定义：

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`.

都是洋码子，翻译一下不？不！还是只说要点：

- 模块就是一个含有python语句的文件
- 模块名就是文件名（不要扩展名.py）

那么，那个`import random`的文件在哪里呢？

用曾经讲过的那个法宝：`help()`函数看看：

```
1. >>> help(random)
```

然后就出现：

```

1.  NAME
2.      random - Random variable generators.
3.
4.  FILE
5.      /usr/local/lib/python2.7/random.py
6.
7.  MODULE DOCS
8.      http://docs.python.org/library/random
9.
10. DESCRIPTION
11.     ...

```

这里非常明显的告诉我们，random模块的文件就是：

/usr/local/lib/python2.7/random.py（注意：这个地址是我的计算机中的地址，可能跟看官的不一样，特别是如果看官用的是windows，肯定跟我这个不一样了。）

看官这时候可能有疑问了，import是怎么找到那个文件的？类似文件怎么写？不用着急，这些我都会一一道来。

标准库

看了前面的random这个例子，看官可能立刻想到一个问题：是不是已经有人把很多常用的功能都写成模块了？然后使用者只需要用类似方法调用即可。的确是，比如上面显示的，就不是某个程序员在使用的时候自己编写的，而是在安装python的时候，就被安装在了计算机里面。观察那个文件存储地址，就知道了。

我根据上面得到的地址，列出/usr/local/lib/python2.7/里面的文件，这些文件就是类似random的模块，由于是python安装就有的，算是标配吧，给它们一个名字“标准模块库”，简称“标准库”。



这张图列出了很少一部分存在这个目录中的模块文件。

Python的标准库(standard library)是Python的一个组成部分，也是Python为的利器，可以让编程事半功倍。

如果看官有时间，请经常访

问：<https://docs.python.org/2/library/>，这里列出了所有标准库的使用方法。

有一点，请看官特别注意，对于标准库而言，由于内容太多，恐怕是记不住的。也不用可以的去记忆，只需要知道有这么一个东西。如果在编写程序的时候，一定要想到，对于某个东西，是不是会有标准库支持呢？然后就到google或者上面给出的地址上搜索。

举例：

```
1. >>> import sys #导入了标准库sys
2. >>> dir(sys)    #如果不到网页上看，用这种方法可以查看这个标准库提供的各种方法
                    (函数)
3. ['__displayhook__', '__doc__', '__egginsert', '__excepthook__',
    '__name__', '__package__', '__plen', '__stderr__', '__stdin__',
    '__stdout__', '_clear_type_cache', '_current_frames', '_getframe',
    '_mercurial', 'api_version', 'argv', 'builtin_module_names',
    'byteorder', 'call_tracing', 'callstats', 'copyright',
    'displayhook', 'dont_write_bytecode', 'exc_clear', 'exc_info',
    'exc_type', 'excepthook', 'exec_prefix', 'executable', 'exit',
    'flags', 'float_info', 'float_repr_style', 'getcheckinterval',
    'getdefaultencoding', 'getdlopenflags', 'getfilesystemencoding',
    'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof',
    'gettrace', 'hexversion', 'last_traceback', 'last_type',
    'last_value', 'long_info', 'maxint', 'maxsize', 'maxunicode',
    'meta_path', 'modules', 'path', 'path_hooks',
    'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2',
    'py3kwarning', 'setcheckinterval', 'setdlopenflags', 'setprofile',
    'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
    'subversion', 'version', 'version_info', 'warnoptions']
```

```

4. >>> sys.platform      #比如这个
5. 'linux2'
6. >>> sys.version       #还有这个
7. '2.7.6 (default, Nov 13 2013, 19:24:16) \n[GCC 4.6.3]'
8.
9. >>> help(sys.stdin)   #这是查看某个模块方法具体内容的方式

```

标准库，在编程中经常用到。这里不赘述。只要看官能够知道在哪里找、如何找所需要的标准库即可。

自己编写模块

看官可能比较喜欢“自己动手，丰衣足食”（虽然真的不一定是丰衣足食），在某些必要的时候，还真得自己动手写一些模块。那么怎么编写模块呢？

前面已经交代，模块就是.py文件，所以，只要将某些语句写到一个.py文件中，它就是一个模块了。没有什么太多的秘密。

在某个目录下面建立了一个文件，名称是：meee.py，如下图所示，然后编辑这个文件内容。编辑好后保存。

代码是文件内容：

```

1. #!/usr/bin/env python
2. #coding:utf-8
3.
4. web = "https://qiwsir.github.io"
5.
6. def my_name(name):
7.     print name
8.
9. class pythoner:
10.     def __init__(self, lang):
11.         self.lang = lang

```

```

12.         def programmer(self):
13.             print "python programmer language is: ",self.lang

```

图示是文件所在目录，并且在该目录下打开了python的交互模式（我这是在ubuntu下，看官是别的操作系统的化，注意路径，如果遇到问题，可以暂时搁置，看下文）。



从图中可以看出，当前目录中有这个文件：mmmm.py

在交互模式下，仿照对标准库模块的操作方式：

```

1. >>> import mmmm
2. >>> dir(mmmm)
3. ['__builtins__', '__doc__', '__file__', '__name__', '__package__',
   'my_name', 'pythoner', 'web']
4. >>> mmmm.__doc__      #这个是空的，正是，因为我未曾写过任何文档说明
5. >>> mmmm.__name__     #名字
6. 'mmmm'
7. >>> mmmm.__file__     #文件
8. 'mmmm.py'

```

再看后面的：my_name, pythoner, web，都是我在内容中自己写的。

```

1. >>> mmmm.web
2. 'https://qiwsir.github.io'

```

web是模块mmmm中的一个通过赋值语句建立的变量，在这里，它编程了mmmm的属性，能够通过点号运算访问，其实不仅仅是这类型的赋值，其它通过def, class等，都能做为mmmm模块的属性。

```

1. >>> mmmm.my_name
2. <function my_name at 0xb74ceb54>
3. >>> mmmm.pythoner

```

```
4. <class mmmm.pythoner at 0xb73e6bcc>
```

当然，跟操作标准库一样，一样能够使用`help()`来看看这些属性的具体内容：

```
1. >>> help(mmmm.my_name)
2.
3. Help on function my_name in module mmmm:
4.
5. my_name(name)
6.
7. >>> help(mmmm.pythoner)
8.
9. Help on class pythoner in module mmmm:
10.
11. class pythoner
12. |   Methods defined here:
13. |
14. |   __init__(self, lang)
15. |
16. |   programmer(self)
```

怎么调用呢？这样即可：

```
1. >>> mmmm.my_name("qiwsir")
2. qiwsir
```

当调用模块中的函数的时候，用模块的名称（`import mmmm`）+点号+函数（注意，函数后面要有括号，如果有参数，括号里面跟参数），即 `module_name.funciton(*args)`

```
1. >>> py = mmmm.pythoner("c++")
2. >>> py.programmer()
3. python programmer language is: c++
```

上面两行，则是演示用绑定的方法调用模块中的类以及类的实例方法。跟以往的相比较，似乎都是在前面多了一个mmmm。

如果感觉这个mmmm比较麻烦，可以用from，具体是这样的：

```
1. >>> from mmmm import *
2. >>> my_name('qiwsir')
3. qiwsir
4. >>> web
5. 'https://qiwsir.github.io'
6. >>> py = pythoner("c++")
7. >>> py.programmer()
8. python programmer language is: c++
```

这次不用总写那么mmmm了。两种方式，哪个更好呢？没有定论。看官在以后的实践中体会，什么时候用什么方式。

上面用from mmmm import ，其中符号，表示将所有的都import进来，用这个方法，也可以只import一部分，如同：

```
1. >>> from mmmm import my_name      #如果看官前面运行了上述操作，需要关闭交互模式，
2.                                     #再重启，才能看到下面过程
3. >>> my_name("qiwsir")
4. qiwsir
5. >>> web                           #没有import这个，所以报错。
6. Traceback (most recent call last):
7.   File "<stdin>", line 1, in <module>
8. NameError: name 'web' is not defined
```

这就是基本的import模块方法。看官的疑问，还要存着。且听下回分解。

模块的加载

- 模块的加载
 - [import的工作流程](#)
 - [搜索模块](#)
 - [重载模块](#)

模块的加载

不管是用import还是用from mmmm import *的方式导入模块，当程序运行之后，回头在看那个存储着mmmm.py文件的目录中（关于[mmmm.py文件可以看上一讲](#)），多了一个文件：

```
1. qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ ls  
   mmm*  
2. mmmm.py  mmmm.pyc
```

在这个目录下面，除了原来的那个mmmm.py之外，又多了一个mmmm.pyc文件，这个文件不是我写的，是哪里来的呢？

要破开此迷，需要用import的过程说起。

import的工作流程

import mmmm，并不是仅仅将mmmm.py这个文件装载到当前位置（文件内），其实是首先进行了一次运算。当mmmm.py被第一次导入的时候，python首先要对其进行编译，生成扩展名为.pyc的同名文件，然后才执行mmmm模块的代码，创建相应的对象等。就如同把大象装进冰箱，有三步要执行：

1. 搜索。就是python要能够找到import的模块。怎么找到，后面讲

述。

2. 编译。找到模块文件之后，将其编译成字节码，就是那个`.pyc`文件里面的（关于字节码，下面会介绍，请继续阅读）。注意，不是什么时候都编译的，只有第一次运行时候才编译，如果`mmmm.py`文件改变了，相当于又一个新文件，也会从新编译。其实就是`.pyc`文件中有一个时间戳，python会自动检查这个时间戳，如果它比同名的`.py`文件时间戳旧，就会从新编译。否则跳过。当然，如果根本就没有找到同名的`.py`源文件，只有字节码文件`.pyc`，那么就只能运行这个了。
3. 运行。这就没什么好说的了，生米已经淘干净了，并且放到锅里，开始加热了，最后就只能熟饭了。执行就是前面已经编译的模块字节码文件，顺理成章要执行了。

搜索模块

一般情况下，python会自动的完成模块搜索过程。但是，在某些情况下，或许会要求程序员来设定搜索路径。当`import`一个模块后，python会按照下面的顺序来找那个将要导入的模块文件

1. 程序的主目录。上一讲中，在`codes`这个目录中运行交互模式，这时候的主目录就是`codes`，当在那个交互模式中运行`import mmmm`的时候，就首先在`codes`这个目录中搜索相应的文件（找到`.py`之后编译成为`.pyc`）。当然，后面在网页编程中，看官会看到，所谓主目录是可以通过顶层文件设置的目录。
2. `PYTHONPATH`目录。这是一个环境变量设置，如果没有设置则滤去。如何进行环境变量设置，请看官google啦。
3. 标准库目录。已经随着Python的安装进入到计算机中的那个。
4. 任何`.pth`文件的内容。如果有这类文件，最后要在这类文件中搜索一下。这是一个简单的方法，在`.pth`文件中，加入有效目录，

使之成为搜索路径。下图就是我的计算机上，存放.pth文件的位置以及里面放着的.pth文件



看官也可以自己编写.pth文件，里面是有关搜索目录，保存到这里。比如，打开目录中的easy-install.pth文件，发现的内容：



搜索就是这么一个过程。这里建议看官了解即可，不一定非要进行什么设置，在很多情况下，python都是会自动完成的。特别是初学者，暂且不要轻举妄动。

重载模块

以mmmm模块为例（在这里要特别提醒看官：我这样命名是相当不好滴，只不过是为了恶搞才这样命名的）。

在一个shell里面，运行了python，并且做了如下操作：

```
1. >>> import mmmm
2. >>> mmmm.web
3. 'https://qiwsir.github.io'
```

下面我再打开一个shell，编辑mmmm.py这个文件，进行适当修改：



保存之后，切换到原来的那个导入了模块的交互模式：

```
1. >>> mmmm.web
2. 'https://qiwsir.github.io'
```

输出的跟前面的一样，没有任何变化，这是为什么呢？

原来，当导入模块的时候，只会在第一次导入时加载和执行模块代码，之后就不会重新加载或重新执行了，如果模块代码修改了，但是这里执行的还是修改之前的。

怎么实现代码修改之后，执行新的呢？一种方式就是退出原来的交互模式，再重新进入，再import mmmm。呵呵，这种方法有点麻烦。

Python提供了另外一个函数—reload函数，能够实现模块的重新加载（简称重载），重载后模块代码重新执行。如下继续：

```
1. >>> reload(mmmm)
2. <module 'mmm' from 'mmm.py'>
3. >>> mmmm.web
4. 'https://qiwsir.github.io, I am writing a python book on line.'
```

这下就显示修改之后的内容了。

特别提醒注意：

- reload是内置函数
- reload(module), module是一个已经存在的模块，不是变量名。

私有和专有

- 私有和专有
 - 私有函数
 - 专有方法

私有和专有

在任何语言中，都会规定某些对象(属性、方法、函数、类等)只能够在某个范围内访问，出了这个范围就不能访问了。这是“公”、“私”之分。此外，还会专门为某些特殊的東西指定一些特殊表示，比如类的名字就不能用class，def等，这就是保留字。除了保留字，python中还为类的名字做了某些特殊准备，就是“专有”的范畴。

私有函数

在某些时候，会看到有一种方法命名比较特别，是以“__”双划线开头的，将这类命名的函数/方法称之为“私有函数”。

所谓私有函数，就是：

- 私有函数不可以从它们的模块外面被调用
- 私有类方法不能够从它们的类外面被调用
- 私有属性不能够从它们的类外面被访问

跟私有对应的，就是所谓的公有啦。有的编程语言用特殊的关键词来说明某函数或方法或类是私有还是公有。但是python仅仅用名字来说明，因为python深刻理解了2k年前孔先生丘所说的“名不正言不顺”的含义。

如果一个 Python 函数, 类方法, 或属性的名字以两个下划线开始 (但

不是结束),它是私有的;其它所有的都是公有的。类方法或者是私有(只能在它们自己的类中使用)或者是公有(任何地方都可使用)。例如:

```
1. class Person:
2.     def __init__(self,name):
3.         self.name = name
4.
5.     def __work(self,salary):
6.         print "%s salary is:%d"%(self.name,salary)
```

这里边定义的方法'__work()'就是一个私有方法。

下面把上面的类进行完善,然后运行,通过实例来调用这个私有方法

```
1. #!/usr/bin/env python
2. #coding:utf-8
3.
4. class Person:
5.     def __init__(self,name):
6.         self.name = name
7.         print self.name
8.
9.     def __work(self,salary):
10.        print "%s salary is: %d"%(self.name,salary)
11.
12. if __name__=="__main__":
13.     officer = Person("Tom")
14.     officer.__work(1000)
15.
16. #运行结果
17.
18. Tom
19. Traceback (most recent call last):
20.   File "225.py", line 14, in <module>
21.     officer.__work(1000)
22. AttributeError: Person instance has no attribute '__work'
```

从运行结果中可以看出，当运行到`officer.__work(1000)`的时候，报错了。并且从报错信息中说，没有该方法。这说明，这个私有方法，无法在类意外调用（其实类意外可以调用私有方法，就是太麻烦，况且也不提倡，故本教程滤去）。

下面将上述代码进行修改，成为：

```
1. #!/usr/bin/env python
2. #coding:utf-8
3.
4. class Person:
5.     def __init__(self,name):
6.         self.name = name
7.         print self.name
8.
9.     def __work(self,salary):
10.        print "%s salary is: %d"%(self.name,salary)
11.
12.    def worker(self):
13.        self.__work(500)           #在类内部调用私有方法
14.
15. if __name__=="__main__":
16.     officer = Person("Tom")
17.     #officer.__work(1000)
18.     officer.worker()
19.
20. #运行结果
21.
22. Tom
23. Tom salary is: 500
```

结果正是要得到的。看官是否理解私有方法的使用了呢？

专有方法

如果是双下划线开头，但不是以双下划线结尾，所命名的方法是私有方法；

如果以双下划线开头，并且以双下划线结尾，所命名的方法就是专有方法。

这是python规定的。所以在写程序的时候要执行，不执行就是跟python过不去，过不去就报错了。

比如前面反复提到的'`_init_()`'，就是一个典型的专有方法。那么自己在写别的方法时，就不要用双下划线开头和结尾了。虽然用了也大概没有什么影响，但是在可读性上就差很多了，一段程序如果可读性不好，用不了多长时间自己就看不懂了，更何况别人呢？

关于专有方法，出了'`_init_()`'之外，还有诸如：'`_str_`'，'`__setitem__`'等等，要查看，可以利用`dir()`函数在交互模式下看看某个函数里面的专有东西。当然，也可以自己定义啦。

因为'`_init_`'用的比较多，所以前面很多例子都是它。

折腾一下目录

- 折腾一下目录
 - 文件的绝对路径
 - 分开目录和文件名
 - 判断
 - 组合路径

折腾一下目录

python在安装的时候，就自带了很多模块，我们把这些模块称之为标准库，其中，有一个是使用频率比较高的，就是 `os` 。这个库中方法和属性众多，有兴趣的看官可以参考官方文档：

档：<https://docs.python.org/2/library/os.html>，或者在交互模式中，用 `dir(os)` 看一看。

```
1. >>> import os    #这个动作很重要，不能缺少
2. >>> dir(os)
3. ['EX_CANTCREAT', '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',
    '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',
'_pickle_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', 'execlp', 'execlpe', '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', 'spawnlp', 'spawnlpe', 'spawnv', 'spawnve', 'spawnvp',
'spawnvpe', 'stat', 'stat_float_times', 'stat_result', 'statvfs',
'statvfs_result', 'strerror', 'symlink', 'sys', 'sysconf',
'sysconf_names', 'system', 'tcgetpgrp', 'tcsetpgrp', 'tempnam',
'times', 'tmpfile', 'tmpnam', 'ttyname', 'umask', 'uname',
'unlink', 'unsetenv', 'urandom', 'utime', 'wait', 'wait3', 'wait4',
'waitpid', 'walk', 'write']
```

在这么多的东西中，本讲只关注 `os.path`，真所谓“弱水三千，只取一瓢”，为什么这么偏爱它呢？因为它和前面已经讲过的文件操作进行配合，就能够随心所欲操作各个地方的文件了（关于文件，请参考：[不要红头文件\(1\)](#)、[不要红头文件\(2\)](#)）

关于 `os.path` 的属性也不少，依然可以用 `dir(os.path)` 查看：

```
1. >>> dir(os.path)
2. ['__all__', '__builtins__', '__doc__', '__file__', '__name__',
```



```
'__package__', '_joinrealpath', '_unicode', '_varprog', 'abspath',
'altsep', 'basename', 'commonprefix', 'curdir', 'defpath',
'devnull', 'dirname', 'exists', 'expanduser', 'expandvars',
'extsep', 'genericpath', 'getatime', 'getctime', 'getmtime',
'getsize', 'isabs', 'isdir', 'isfile', 'islink', 'ismount', 'join',
'lexists', 'normcase', 'normpath', 'os', 'pardir', 'pathsep',
'realpath', 'relpath', 'samefile', 'sameopenfile', 'samestat',
'sep', 'split', 'splitdrive', 'splittext', 'stat',
'supports_unicode_filenames', 'sys', 'walk', 'warnings']
```

这么多属性，看官可以用 `help()` 逐个查看有关信息，并了解其使用方法。下面列出常见的几个使用方法，为看官减轻一点阅读英文的障碍，不过，如果看官英语足够好，请直接看原文档。就像这样：

```
1. >>> help(os.path.split)
2.
3. split(p)
4.     Split a pathname. Returns tuple "(head, tail)" where "tail" is
5.     everything after the final slash. Either part may be empty.
```

以下将一些典型举例说明：

特别说明，下面的所有操作，均是进入到如下的目录中进行的。

```
1. qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$ pwd
2. /home/qw/Documents/ITArticles/BasicPython/codes          #当前目录
3.
4. qw@qw-Latitude-E4300:~/Documents/ITArticles/BasicPython/codes$
  python
5.
6. Python 2.7.6 (default, Nov 13 2013, 19:24:16)
7. [GCC 4.6.3] on linux2
8. Type "help", "copyright", "credits" or "license" for more
  information.
9. >>>
```

文件的绝对路径

```
1. >>> import os.path
2. >>> os.path.abspath("225.py")
3. '/home/qw/Documents/ITArticles/BasicPython/codes/225.py'
```

文件 225.py 是真实存在上述路径中的，得到了该文件的绝对路径。但是，如果随便提供一个不在这个目录中的文件，又如何？

```
1. >>> os.path.isfile("225.py")
2. True
3.
4. >>> os.path.isfile("2222.py")
5. False
6. >>> os.path.abspath("2222.py")
7. '/home/qw/Documents/ITArticles/BasicPython/codes/2222.py'
```

`os.path.isfile(path)`，可以判断path中是否是文件，其实是判断在该路径中，是否存在那个文件，如果存在则返回True，否则False。上面的操作发现 2222.py 这个文件在当前目录下是不存在的，但是，用 `os.path.abspath("2222.py")` 能够返回一个绝对路径并带有这个不存在的文件的文件名。这里不妨理解为，如果要建立这个文件，它即将被放在那个位置。

按照这样理解，还可：

```
1. >>> os.path.abspath("/home/qw/kkkkkkkk.kk")
2. '/home/qw/kkkkkkkk.kk'
```

分开目录和文件名

```
1. >>> pn = os.path.abspath("225.py")
2. >>> pn
```

```

3.  '/home/qw/Documents/ITArticles/BasicPython/codes/225.py'
4.
5.  >>> os.path.split(pn)
6.  ('/home/qw/Documents/ITArticles/BasicPython/codes', '225.py')
7.  >>> path, filename = os.path.split(pn)[0], os.path.split(pn)[1]
8.  >>> path
9.  '/home/qw/Documents/ITArticles/BasicPython/codes'
10. >>> filename
11. '225.py'

```

`os.paht.split()` , 参数是目录加文件名, 就可以将路径和文件名分开。其实, 我看这个功能不是很智能, 你看这样

```

1.  >>> os.path.split("/home/qw")
2.  ('/home', 'qw')
3.
4.  >>> os.path.split("/home/qw/")
5.  ('/home/qw', '')

```

它就是将最后一组认为是文件名了, 即最后一个 `/` 后面的就是文件名, 所以第二个实验中, 文件名是空了。是不是有点傻呢?

同样, 参数中的文件或者目录, 不一定是你的电脑中真实存在的, 请看:

```

1.  >>> os.path.split("/foo/python/qiwsir/git.git")
2.  ('/foo/python/qiwsir', 'git.git')

```

只要符合目录书写结构, 就可以分解了。

有另外两个属性, 是 `os.path.split()` 的分别执行, 即可以分别获得路径和文件名, 这样让操作更简单了。

```

1.  >>> os.path.dirname("/foo/python/qiwsir/git.git")
2.  '/foo/python/qiwsir'

```

```
3. >>> os.path.basename("foo/python/qiwsir/git.git")
4. 'git.git'
```

判断

前面稍微提到了 `os.path.isfile()` 可以用来判断一个文件是否存在，那么判断目录路径是否存在，可否？可：

```
1. >>> os.path.exists("/foo/python/qiwsir")
2. False
3. >>> os.path.exists("/home/qw/Documents")
4. True
```

判断相关的属性还有：

- `os.path.isabs(path)`：判断path是否为绝对路径
- `os.paht.isdir(path)`：判断path是否为存在的目录

组合路径

将两个或多个对象组合起来，是常见的事情，那么如何将多个路径组合呢？如下：

```
1. >>> os.path.join("/home/python", "/BasicsPython", "226.md")
2. '/BasicsPython/226.md'
```

特别提醒，这个属性的返回值中，将第一个绝对路径忽略。

```
1. >>> os.path.join("/", "/home/qw", "learnpython.md")
2. '/home/qw/learnpython.md'
```

[返回首页](#) | [上一讲：私有和专有](#)

第三部分 昨夜西风，亭台谁登

链接

- [网站的结构](#)
- [通过Python连接数据库](#)
- [用Python操作数据库\(1\)](#)
- [用Python操作数据库\(2\)](#)
- [用Python操作数据库\(3\)](#)
- [python开发框架](#)
- [Hello, 第一个网页分析](#)
- [实例分析get和post](#)
- [问候世界](#)
- [使用表单和模板](#)
- [模板中的语法](#)
- [静态文件以及一个项目框架](#)
- [模板转义](#)

网站的结构

- 网站的结构
 - 网站组成
 - 从数据库开始
 - 安装MySQL
 - 运行mysql

网站的结构

很早很早的时候，computer这个东西习惯于被称之为计算机，因为它的主要功能是完成一些科学计算的东西，我记得自己鼓捣它的时候，就是计算，根本就没有想到它有早一日还可以用来做别的。后来另外一个名字“电脑”逐渐被人们接收了，特别是网络发展起来之后，computer这个东西，如果不上网，简直就不知道干什么。而且，现在似乎还有一个趋势，越来越强化网络的作用，而本机的功能虽然硬件在提升，可以做的事情感觉不多了。

不管怎么，网络是离不开了。上网，连上网之后干什么呢？就是要登录某某网站。不是联网之后自动的网上内容就涌进自己的计算机，而是要操作一下那个浏览器，输入网址，打开某个网站的页面，才能得到我们要看的内容。所以，网络上，必须有网站，才能让别人来看。上网——看网页，这是发生频率非常高的动作。

那么这里就涉及到网站。网站是谁做的呢？这是废话，人做的。只不过这里的人可能是给某个公司打工的，也可能是类似个体户的。

网站怎么做呢？做法很多啦。有直接用html网页写的，有用别的什么开源系统做的，等等。

从本讲开始，我和列位看官就来看看，用python怎么做一个网站。

维基百科对网站有如下描述：

网站（英文：*Website*）是指在互联网上，根据一定的规则，使用HTML等工具制作的用于展示特定内容的相关网页的集合。简单地说，网站是一种通信工具，就像布告栏一样，人们可以通过网站来发布自己想要公开的信息，或者利用网站来提供相关的网络服务。人们可以通过网页浏览器来访问网站，获取自己需要的信息或者享受网络服务。世界上第一个网站由蒂姆·伯纳斯-李创建于1991年8月6日。

网站组成

网站是由两大部分组成，一是服务器，二是程序。

服务器，是硬件部分。如果看官有条件，可以自己购买服务器，然后自己建立机房或者托管到什么信的机房等等，这样拥有了自己的服务器啦。当然，要不少银两的。如果银两不足，就可以用省钱的方法，购买某公司所提供的服务器空间，因为市场经济带来的好处，总有人会想到不是人人都自己买得起服务器的，也不是人人都有必要自己买服务器的。但是，如果还要做网站，自己又不拥有服务器怎么办？所以，有人就做这个生意，出租他自己的服务器的一部分空间给我们这些穷人，这样就双赢了，穷人只要有技术，就可以很低的代价在网上拥有自己的网站，富人（出租服务器的）也能够通过出租收租金啦。就好比租房的人和房东的关系一样。当然，这样做的结果就是必须要跟别人共同租用一个服务器，如果自己单独租一个，价格就又贵了。

如果，我是说如果，如果你做的网站不打算放到网上让别人随时看（有这样的吗？那不是白做了吗？有！而且很多，比如我的网站还没有做好，我就不让别人看），这时候还可以将自己的电脑当做一个服务器，在自己的电脑上发布自己的网站，自我欣赏，必要时把把旁边人拉到显示器前面看看吧。很自恋啦。（在自己的电脑发布的网站，其实也能够通过互联网被人看到，就是需要一点小小的技术来发布了，这个不是重点，本教程不讲，需要者可以google或者联系我。）

看官和我在后续的学习中，用的服务器就是自己的电脑啦。我们都是喜欢自恋的。

另外一部分就是服务器里面装的软件部分，通常所说的网站，更多的是指这个部分。一般来讲，这个部分是比较复杂的，因为网站不同，而有很多不同的程序。但是，不管什么网站，都得有一个让别人看的界面，这就是一个网页，或者说，只要有一个网页了，它就可以做为一个网站发布出去。

那么就出现了一种比较简单的网站，就是由一些网页组成，而且，这些网页仅仅是用html代码写成的（或者用html网页编辑工具，有图文形式的，就可以编辑网页），回想我最早做的那个网页，就是纯粹用html代码写的。这样写出来的网页，用行话说是“静态的”。意思就是指它不允许用户和网站有什么交互，只是让别人看。比如看客手欠，非要搜索什么东西，对不起，网站不提供此功能。这种网站现在比较少了。

如果要增加交互功能，怎么办？那就要有处理用户向网站提交的信息的程序了。这样，网站就多了一部分，行话常说是“后端”，对应前面说的那个直接展示给看客的叫做“前端”。“后端”所做的事情就是处理“前端”用户提交的信息，然后给用户一个反馈。这样就交互起来了。

此外，为了将网站上的数据保存起来，通常会用到一个叫做“数据库”的东西（这个不是必须的，有的网站就没有数据库，有的网站用别的方式存储数据，比如文本等），数据库主要是存储某些数据，让网站的后端和前端从这里将某些数据读出来，显示给看官，或者将看官提交的某些数据存进去，以便以后使用。

数据库是计算机行业中的一个专业门来，看官有兴趣，可以在这个行业中深入，公司里面有个职位：DBA，就是干这个的。

数据库，简单来说本身可视为电子化的文件柜——存储电子文件的处所，使用者可以对文件中

的数据运行新增、截取、更新、删除等操作。

数据库管理员（英语：*Database administrator*，简称DBA），是负责管理数据库的人。数据库管理员负责在系统上运行数据库，执行备份，执行安全策略和保持数据库的完整性。因为管理数据库是个很庞大的职务，每个公司或组织的数据库管理员的需要也是很不同。一个大公司可能有很多数据库管理员，但是一个小公司可能也没有数据库管理员，而让系统管理员管理数据库。

综合以下，一般来讲，网站应该是这样的：



为了写一个漂亮的前端，一般都要用CSS和JavaScript，但是，本教程中，因为不是专门讲授这些，所以，涉及到前端的时候，就不用CSS和JavaScript了，这样的一个恶果就是界面相当丑陋。请看官忍受吧。

在控制端，就是前面说的后端，仅适用一种语言：Python。这是本教程的终极目的，如何用Python做网站。

数据库，我选用MySQL，关于这个数据库有很多传说。例如[维基百科上这么说](#)：

MySQL（官方发音为英语发音：*/maɪˌɛskjuːˈɛl/* “My S-Q-L”，^[1]，但也经常读作英语发音：*/maɪˈsiːkwəl/* “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依然是一个不错的数据库选择，足够支持看官完成

一个相当不小的网站。

至于服务器空间，就放在自己的电脑上吧。

从数据库开始

数据库是我们要做的网站的一个基础，我在这里不演示不用数据库的情况，因为那种玩具网站，虽然讲授简单，但是看官总是有点晕乎，距离真实的环境差距太大了，既然学，就学点真的。

从现在开始，就进入网站建设的进程。

安装MySQL

你的电脑不会天生就有MySQL，它本质上也是一个程序，需要安装到电脑中。

如果看官跟我一样，用的是ubuntu操作系统，可以用下面的方法（我相信，用ubuntu的一定很少，不过，如果看官要成为一个优秀的程序员，我还是推荐使用这个操作系统，或者别的LINUX发行版。哈哈）。

第一步，在shell端运行如下命令：

```
1. sudo apt-get install mysql-server
```

这样，看官的电脑上就已经安装好了这个数据库。当然，当然，还要进行配置。

第二步，配置MySQL

安装之后，运行：

```
1. service mysqld start
```

启动mysql数据库。然后进行下面的操作，对其进行配置。（启动数据库这步是后来补充的，网友[王孝先](#)告诉我，这个不能丢掉。谢谢王先生。）

默认的MySQL安装之后根用户是没有密码的，看官注意，这里有一个名词“根用户”，其用户名是：root。运行：

```
1. $mysql -u root
```

在这里之所以用-u root是因为我现在是一般用户（firehare），如果不加-u root的话，mysql会以为是firehare在登录。注意，我在这里没有进入根用户模式，因为没必要。一般来说，对mysql中的数据库进行操作，根本没必要进入根用户模式，只有在设置时才有这种可能。

进入mysql之后，会看到>符号开头，这就是mysql的命令操作界面了。

下面设置Mysql中的root用户密码了，否则，Mysql服务无安全可言了。

```
1. mysql> GRANT ALL PRIVILEGES ON *.* TO root@localhost IDENTIFIED BY "123456";
```

注意，我这儿用的是123456做为root用户的密码，但是该密码是不安全的，请大家最好使用大小写字母与数字混合的密码，且不少于8位。

以后如果在登录数据库，就可以用刚才设置的密码了。

除了上面的安装过程，看官如果用的是别的操作系统，可以在google上搜索相应的安装方法，恕我不在这里演示，因为我只能演示在ubuntu上的安装流程。不过，google会帮你解决安装遇到的问题。

运行mysql

安装之后，就要运行它，并操作这个数据库，建立一个做网站的基础。
我这样来运行数据库：

```
1. qw@qw-Latitude-E4300:~$ mysql -u root -p
2. Enter password:
```

输入数据库的密码，之后出现：

```
1. Welcome to the MySQL monitor.  Commands end with ; or \g.
2. Your MySQL connection id is 373
3. Server version: 5.5.38-0ubuntu0.14.04.1 (Ubuntu)
4.
5. Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights
   reserved.
6.
7. Oracle is a registered trademark of Oracle Corporation and/or its
8. affiliates. Other names may be trademarks of their respective
9. owners.
10.
11. Type 'help;' or '\h' for help. Type '\c' to clear the current input
    statement.
12.
13. mysql>
```

看到这个界面内容，就说明你已经进入到数据里面了。接下来就可以对这个数据进行操作。例如：

```
1. mysql> show databases;
2. +-----+
3. | Database |
4. +-----+
5. | information_schema |
6. | carstore |
7. | cutvideo |
```

```
8. | itddiffer |
9. | mysql |
10. | performance_schema |
11. | phpcms |
12. | phpcms2 |
13. | pushsystem |
14. | sipras |
15. | test |
16. +-----+
```

用这个命令，就列出了当前mysql已经有的数据库。

除了这种用命令行形式对数据库进行操作之外，还有不少可视化方式操作数据库的工具。这里也不作介绍，有兴趣的请google。不过，我喜欢命令行。

通过Python连接数据库

- 通过Python连接数据库
 - 安装python-MySQLdb
 - 交互模式下操作数据库之连接数据库

通过Python连接数据库

用Python来编写网站，必须要能够通过python操作数据库，所谓操作数据库，就是通过python实现对数据的连接，以及对记录、字段的各种操作。上一讲提到的那种操作方式，是看官直接通过交互模式来操作数据库。

安装python-MySQLdb

要想通过python来操作数据库，还需要在已经安装了mysql的基础上安装一个称之为mysqldb的库，它是一个接口程序，python通过它对mysql数据实现各种操作。

在编程中，会遇到很多类似的接口程序，通过接口程序对另外一个对象进行操作，比较简单。接口程序就好比钥匙，如果要开锁，人直接用手指去捅，肯定是不行的，那么必须借助工具，插入到锁孔中，把所打开，打开所之后，门开了，就可以操作门里面的东西了。那么打开所的工具就是接口程序。而打开所的工具会有便利与否之分，如果用这锁的钥匙，就便利，如果用别的工具，或许不便利（其实还分人，也就是人开锁的水平，如果是江洋大盗或者小毛贼什么的，擅长开锁，用别的工具也便利了），也就是接口程序不同，编码水平不同，都是考虑因素。

这里下载python-

mysqldb:<https://pypi.python.org/pypi/MySQL-python/>

下载之后就可以安装了。

我这里只能演示ubuntu下安装的过程。

```
1. sudo apt-get install python-MySQLdb
```

在shell中输入上面的命令行，就安装了。看看，多么简洁的安装，请快快用ubuntu吧。我愿意做ubuntu的免费代言。哈哈。

不管什么系统，安装不是难题。安装之后，怎么知道安装的结果呢？

```
1. >>> import MySQLdb
```

在python的交互模式中，输入上面的指令，如果不报错，恭喜你，已经安装好了。如果报错，恭喜你，可以借着错误信息提高自己的计算机水平了，请求助于google大神。

交互模式下操作数据库之连接数据库

操作数据库的前提是先有数据库。

先建立一个数据库。

```
1. qw@qw-Latitude-E4300:~$ mysql -u root -p
2. Enter password:
```

打开数据库，正确输入密码之后，呈现下面的结果

```
1. Welcome to the MySQL monitor. Commands end with ; or \g.
2. Your MySQL connection id is 373
3. Server version: 5.5.38-0ubuntu0.14.04.1 (Ubuntu)
4.
5. Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.
```



```

6.
7. Oracle is a registered trademark of Oracle Corporation and/or its
8. affiliates. Other names may be trademarks of their respective
9. owners.
10.
11. Type 'help;' or '\h' for help. Type '\c' to clear the current input
    statement.
12.
13. mysql>

```

在这个状态下，输入如下命令，建立一个数据库：

```

1. mysql> create database qiwsirtest character set utf8;
2. Query OK, 1 row affected (0.00 sec)

```

注意上面的指令，如果仅仅输入：create database qiwsirtest，也可以，但是，我在后面增加了character set utf8，意思是所建立的数据库qiwsirtest，编码是utf-8的，这样存入汉字就不是乱码了。

看到那一行提示：Query OK, 1 row affected (0.00 sec)，就说明这个数据库已经建立好了，名字叫做：qiwsirtest

数据库建立之后，就可以用python通过已经安装的mysqldb来连接这个名字叫做qiwsirtest的库了。进入到python交互模式（现在这个实验室做实验）。

```

1. >>> import MySQLdb
2. >>> conn =
    MySQLdb.connect(host="localhost", user="root", passwd="123123", db="qiws

```

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，因为就数据库就

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，如果是比较大型的服务，数据库会提供不同的用户，那时候可以更改为相应用户。但是，不同用户的权限可能不同，所以，在程序中，如果要操作数据库，还要注意所拥有的权限。在这里用root，就放心了，什么权限都有啦。不过，这样做，在大型系统中是应该避免的。
- passwd:上述user账户对应的登录mysql的密码。我在上面的例子中用的密码是"123123"。不要忘记引号。
- db:就是刚刚通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，mysql的默认端口是3306，当mysql被安装到服务器之后，为了能够允许网络访问，服务器（计算机）要提供一个访问端口给它。
- charset:这个设置，在很多教程中都不写，结果在真正进行数据存储的时候，发现有乱码。这里我将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mysql设置中，utf-8写成utf8,没有中间的横线。但是在python文件开头和其它地方设置编码格式的时候，要写成utf-8。切记！

注：connect中的host、user、passwd等可以不写，只有在写的时候按照host、user、passwd、db(可以不写)、port顺序写就可以，注意端口号port=3306还是不要省略的为好，如果没有db在port前面，直接写3306会报错。

其实，关于connect的参数还不少，下面摘抄来自[mysqlldb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.

我已经完成了数据库的连接，虽然是在交互模式下，看官你是否也实现了呢？下一讲，将开始讲述如何操作数据库。

用Python操作数据库(1)

- 用Python操作数据库（1）
 - 建数据库表并插入数据
 - python操作数据库
 - cursor执行命令的方法：

“So do not worry about tomorrow, for tomorrow will bring worries of its own. Today’s trouble is enough for today.” (MATTHEW 7:34)

用Python操作数据库（1）

在[上一讲](#)中已经连接了数据库。就数据库而言，连接之后就要对其操作。但是，目前那个名字叫做qiwsirtest的数据仅仅是空架子，没有什么可操作的，要操作它，就必须在里面建立“表”，什么是数据库的表呢？下面摘抄自维基百科对数据库表的简要解释，要想详细了解，需要看官在找一些有关数据库的教程和书籍来看看。

在关系数据库中，数据库表是一系列二维数组的集合，用来代表和储存数据对象之间的关系。它由纵向的列和横向的行组成，例如一个有关作者信息的名为 *authors* 的表中，每个列包含的是所有作者的某个特定类型的信息，比如“姓氏”，而每行则包含了某个特定作者的所有信息：姓、名、住址等等。

对于特定的数据库表，列的数目一般事先固定，各列之间可以由列名来识别。而行的数目可以随时、动态变化，每行通常都可以根据某个（或某几个）列中的数据来识别，称为候选键。

我打算在qiwsirtest中建立一个存储用户名、用户密码、用户邮箱的表，其结构用二维表格表现如下：

username	password	email
qiwsir	123123	qiwsir@gmail.com

特别说明，这里为了简化细节，突出重点，对密码不加密，直接明文保存，虽然这种方式是很不安全的。但是，有不少网站还都这么做的，这

么做的目的是比较可恶的。就让我在这里，仅仅在这里可恶一次。

建数据库表并插入数据

为了在数据库中建立这个表，需要进入到 `mysql>` 交互模式中操作。道理在于，如果qiwsirtest这个屋子里面没有类似家具的各种数据库表，即使进了屋子也没有什么好操作的东西，因此需要先到 `mysql>` 模式下在屋子里面摆家具。

进入数据库交互模式：

1. `qw@qw-Latitude-E4300:~$ mysql -u root -p`
2. `Enter password:`

调用已经建立的数据库：qiwsirtest

1. `mysql> use qiwsirtest;`
2. `Database changed`
3. `mysql> show tables;`
4. `Empty set (0.00 sec)`

用 `show tables` 命令显示这个数据库中是否有数据表了。查询结果显示为空。

下面就用如下命令建立一个数据表，这个数据表的内容就是上面所说明的。

1. `mysql> create table users(id int(2) not null primary key auto_increment,username varchar(40),password text,email text)default charset=utf8;`
2. `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。

```
1. mysql> show tables;
2. +-----+
3. | Tables_in_qiwsirtest |
4. +-----+
5. | users                |
6. +-----+
7. 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。

```
1. mysql> desc users;
2. +-----+-----+-----+-----+-----+-----+
3. | Field      | Type          | Null | Key | Default | Extra          |
4. +-----+-----+-----+-----+-----+-----+
5. | id         | int(2)        | NO   | PRI | NULL    | auto_increment |
6. | username  | varchar(40)   | YES  |     | NULL    |                |
7. | password  | text          | YES  |     | NULL    |                |
8. | email     | text          | YES  |     | NULL    |                |
9. +-----+-----+-----+-----+-----+-----+
10. 4 rows in set (0.00 sec)
```

显示表users的结构：

- id：每增加一个用户，id号自动增加一个。
- username：存储用户名，类型是varchar(40)
- password：存储用户密码，类型是text
- email：存储用户的邮箱，类型是text

特别提醒：在这里，我没有对每个字段做注入不得为空等设置，在真正的开发中，或许必须让username和password不得为空。

这个结构和上面所期望的结构是一样的，只不过这个表中还没有任何数

据，是一个空表。可以查询一下看看：

```
1. mysql> select * from users;
2. Empty set (0.01 sec)
```

目前表是空的，为了能够在后面用python操作这个数据表，需要向里面插入点信息，就只插入一条吧。

```
1. mysql> insert into users(username,password,email)
   values("qiwsir","123123","qiwsir@gmail.com");
2. Query OK, 1 row affected (0.05 sec)
3.
4. mysql> select * from users;
5. +----+-----+-----+-----+
6. | id | username | password | email |
7. +----+-----+-----+-----+
8. | 1 | qiwsir | 123123 | qiwsir@gmail.com |
9. +----+-----+-----+-----+
10. 1 row in set (0.00 sec)
```

到目前为止，在 `mysql>` 中的工作已经完成了，接下来就是用python操作了。

python操作数据库

要对数据库进行操作，需要先连接它。上一讲看官连接过了，但是，随后你关闭了python的交互模式，所以还要从新连接。这也是交互模式的缺点。不过在这里操作直观，所以暂且忍受一下，后面就会讲解如何在程序中自动完成了。

```
1. >>> import MySQLdb
2. >>> conn =
   MySQLdb.connect(host="localhost",user="root",passwd="123123",db="qiwsir")
```

完成连接的过程，其实是建立了一个 `MySQLdb.connect()` 的实例对象 `conn`，那么这个对象有哪些属性呢？

- `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])`: 游标指针。下面详解。

连接成功之后，开始操作。注意：MySQLdb用游标（指针）`cursor`的方式操作数据库，就是这样：

```
1. >>> cur = conn.cursor()
```

因该模块底层其实是调用C API的，所以，需要先得到当前指向数据库的指针。这也就提醒我们，在操作数据库的时候，指针会移动，如果移动到数据库最后一条了，再查，就查不出什么来了。看后面的例子就明白了。

下面用`cursor()`提供的方法来进行操作，方法主要是：

1. 执行命令
2. 接收结果

cursor执行命令的方法：

- `execute(query, args)`: 执行单条sql语句。query为sql语句本身，args为参数值的列表。执行后返回值为受影响的行数。
- `executemany(query, args)`: 执行单条sql语句，但是重复执行参数列表里的参数，返回值为受影响的行数

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：

```
1. >>> cur.execute("insert into users (username,password,email) values
    (%s,%s,%s)", ("python", "123456", "python@gmail.com"))
2. 1L
```

没有报错，并且返回一个“1L”结果，说明有一行记录操作成功。不妨用“mysql>”交互方式查看一下：

```
1. mysql> select * from users;
2. +----+-----+-----+-----+
3. | id | username | password | email |
4. +----+-----+-----+-----+
5. | 1 | qiwsir | 123123 | qiwsir@gmail.com |
6. +----+-----+-----+-----+
7. 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。

在这里，特别请列位看官注意，通过“cur.execute()”对数据库进行操作之后，没有报错，完全正确，但是不等于数据就已经提交到数据库中了，还必须要用到“MySQLdb.connect”的一个属性：commit()，将数据提交上去，也就是进行了“cur.execute()”操作，要将数据提交，必须执行：

```
1. >>> conn.commit()
```

在到“mysql>”中运行“select * from users”试一试：

```
1. mysql> select * from users;
2. +----+-----+-----+-----+
3. | id | username | password | email |
4. +----+-----+-----+-----+
```

```

5. | 1 | qiwsir | 123123 | qiwsir@gmail.com |
6. | 2 | python | 123456 | python@gmail.com |
7. +---+-----+-----+-----+
8. 2 rows in set (0.00 sec)

```

good, very good。果然如此。这就如同编写一个文本一样，将文字写到文本上，并不等于文字已经保留在文本文件中了，必须执行“CTRL-S”才能保存。也就是在通过python操作数据库的时候，以“execute()”执行各种sql语句之后，要让已经执行的效果保存，必须运行“commit()”，还要提醒，这个属性是“MySQLdb.connect()”实例的。

再尝试一下插入多条的那个命令“executemany(query, args)”。

```

1. >>> 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")))
2. 4L
3. >>> conn.commit()

```

到“mysql>”里面看结果：

```

1. mysql> select * from users;
2. +---+-----+-----+-----+
3. | id | username | password | email |
4. +---+-----+-----+-----+
5. | 1 | qiwsir | 123123 | qiwsir@gmail.com |
6. | 2 | python | 123456 | python@gmail.com |
7. | 3 | google | 111222 | g@gmail.com |
8. | 4 | facebook | 222333 | f@face.book |
9. | 5 | github | 333444 | git@hub.com |
10. | 6 | docker | 444555 | doc@ker.com |
11. +---+-----+-----+-----+
12. 6 rows in set (0.00 sec)

```

成功插入了多条记录。特别请列位注意的是，在“`executemany(query, args)`”中，`query`还是一条sql语句，但是`args`这时候是一个tuple，这个tuple里面的元素也是tuple，每个tuple分别对应sql语句中的字段列表。这句话其实被执行多次。只不过执行过程不显示给我们看罢了。

已经会插入了，然后就可以有更多动作。且看下一讲吧。

[首页](#) | [上一讲：通过Python连接数据库](#) | [下一讲：用Python操作数据库（2）](#)

用Python操作数据库 (2)

- 用Python操作数据库 (2)
 - 查询数据
 - 更新数据

用Python操作数据库 (2)

回顾一下已有的战果：（1）连接数据库；（2）建立指针；（3）通过指针插入记录；（4）提交将插入结果保存到数据库。在交互模式中，先温故，再知新。

```
1. >>> #导入模块
2. >>> import MySQLdb
3.
4. >>> #连接数据库
5. >>> conn =
    MySQLdb.connect(host="localhost",user="root",passwd="123123",db="qiws
6.
7. >>> #建立指针
8. >>> cur = conn.cursor()
9.
10. >>> #插入记录
11. >>> cur.execute("insert into users (username,password,email) values
    (%s,%s,%s)",("老齐","9988","qiwsir@gmail.com"))
12. 1L
13.
14. >>> #提交保存
15. >>> conn.commit()
```

如果看官跟我似的，有点强迫症，总是想我得看到数据中有了，才放芳心呀。那就在进入到数据库，看看。

```

1. mysql> select * from users;
2.      +-----+-----+-----+-----+
3.      | id | username | password | email |
4.      +-----+-----+-----+-----+
5.      |  1 | qiwsir  | 123123   | qiwsir@gmail.com |
6.      |  2 | python  | 123456   | python@gmail.com  |
7.      |  3 | google  | 111222   | g@gmail.com       |
8.      |  4 | facebook| 222333   | f@face.book       |
9.      |  5 | github  | 333444   | git@hub.com       |
10.     |  6 | docker  | 444555   | doc@ker.com       |
11.     |  7 | 老齐    | 9988     | qiwsir@gmail.com  |
12.      +-----+-----+-----+-----+
13.     7 rows in set (0.00 sec)

```

刚才温故的时候，插入的那条记录也赫然在目。不过这里特别提醒看官，我在前面建立这个数据库和数据表的时候，就已经设定好了字符编码为utf8，所以，在现在看到的查询结果中，可以显示汉字。否则，就看到的是一堆你不懂的码子了。如果看官遇到，请不要慌张，只需要修改字符编码即可。怎么改？请google。网上很多。

温故结束，开始知新。

查询数据

在前面操作的基础上，如果要从数据库中查询数据，当然也可以用指针来操作了。

```

1. >>> cur.execute("select * from users")
2. 7L

```

这说明从users表汇总查询出来了7条记录。但是，这似乎有点不友好，告诉我7条记录查出来了，但是在哪里呢，看前面在'mysql>'下操作查询命令的时候，一下就把7条记录列出来了。怎么显示python在

这里的查询结果呢？

原来，在指针实例中，还要用这样的方法，才能实现上述想法：

- `fetchall(self)`:接收全部的返回结果行。
- `fetchmany(size=None)`:接收size条返回结果行。如果size的值大于返回的结果行的数量,则会返回`cursor.arraysize`条数据。
- `fetchone()`:返回一条结果行。
- `scroll(value, mode='relative')`:移动指针到某一行。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。

按照这些规则，尝试：

```
1. >>> cur.execute("select * from users")
2. 7L
3. >>> lines = cur.fetchall()
```

到这里，还没有看到什么，其实已经将查询到的记录（把他们看做对象）赋值给变量`lines`了。如果要把它们显示出来，就要用到曾经学习过的循环语句了。

```
1. >>> for line in lines:
2. ...     print line
3. ...
4. (1L, u'qiwsir', u'123123', u'qiwsir@gmail.com')
5. (2L, u'python', u'123456', u'python@gmail.com')
6. (3L, u'google', u'111222', u'g@gmail.com')
7. (4L, u'facebook', u'222333', u'f@face.book')
8. (5L, u'github', u'333444', u'git@hub.com')
9. (6L, u'docker', u'444555', u'doc@ker.com')
10. (7L, u'\u8001\u9f50', u'9988', u'qiwsir@gmail.com')
```

很好。果然是逐条显示出来了。列位注意，第七条中的 `u'\u8001\u95f5'`，这里是汉字，只不过由于我的shell不能显示罢了，不必惊慌，不必搭理它。

只想查出第一条，可以吗？当然可以！看下面的：

```
1. >>> cur.execute("select * from users where id=1")
2. 1L
3. >>> line_first = cur.fetchone()      #只返回一条
4. >>> print line_first
5. (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：

```
1. >>> cur.execute("select * from users")
2. 7L
3. >>> print cur.fetchall()
4. ((1L, u'qiwsir', u'123123', u'qiwsir@gmail.com'), (2L, u'python',
u'123456', u'python@gmail.com'), (3L, u'google', u'111222',
u'g@gmail.com'), (4L, u'facebook', u'222333', u'f@face.book'), (5L,
u'github', u'333444', u'git@hub.com'), (6L, u'docker', u'444555',
u'doc@ker.com'), (7L, u'\u8001\u95f5', u'9988',
u'qiwsir@gmail.com'))
```

原来，用 `cur.execute()` 从数据库查询出来的东西，被“保存在了 `cur` 所能找到的某个地方”，要找出这些被保存的东西，需要用 `cur.fetchall()`（或者 `fetchone` 等），并且找出来之后，做为对象存在。从上面的实验探讨发现，被保存的对象是一个 `tuple` 中，里面的每个元素，都是一个一个的 `tuple`。因此，用 `for` 循环就可以一个一个拿出来了。

看官是否理解其内涵了？

接着看，还有神奇的呢。

接着上面的操作，再打印一遍

```
1. >>> print cur.fetchall()
2. ()
```

晕了！怎么什么是空？不是说做为对象已经存在了内存中了吗？难道这个内存中的对象是一次有效吗？

不要着急。

通过指针找出来的对象，在读取的时候有一个特点，就是那个指针会移动。在第一次操作了`print cur.fetchall()`后，因为是将所有的都打印出来，指针就要从第一条移动到最后一条。当`print`结束之后，指针已经在最后一条的后面了。接下来如果再次打印，就空了，最后一条后面没有东西了。

下面还要实验，检验上面所说：

```
1. >>> cur.execute('select * from users')
2. 7L
3. >>> print cur.fetchone()
4. (1L, u'qiwsir', u'123123', u'qiwsir@gmail.com')
5. >>> print cur.fetchone()
6. (2L, u'python', u'123456', u'python@gmail.com')
7. >>> print cur.fetchone()
8. (3L, u'google', u'111222', u'g@gmail.com')
```

这次我不一次全部打印出来了，而是一次打印一条，看官可以从结果中看出来，果然那个指针在一条一条向下移动呢。注意，我在这次实验中，是重新运行了查询语句。

那么，既然在操作存储在内存中的对象时候，指针会移动，能不能让指针向上移动，或者移动到指定位置呢？这就是那个`scroll()`

```

1. >>> cur.scroll(1)
2. >>> print cur.fetchone()
3. (5L, u'github', u'333444', u'git@hub.com')
4. >>> cur.scroll(-2)
5. >>> print cur.fetchone()
6. (4L, u'facebook', u'222333', u'f@face.book')

```

果然，这个函数能够移动指针，不过请仔细观察，上面的方式是让指针相对与当前位置向上或者向下移动。即：

`cur.scroll(n)`，或者，`cur.scroll(n, "relative")`：意思是相对当前位置向上或者向下移动，`n`为正数，表示向下（向前），`n`为负数，表示向上（向后）

还有一种方式，可以实现“绝对”移动，不是“相对”移动：增加一个参数“absolute”

特别提醒看官注意的是，在python中，序列对象是的顺序是从0开始的。

```

1. >>> cur.scroll(2, "absolute")    #回到序号是2,但指向第三条
2. >>> print cur.fetchone()        #打印，果然是
3. (3L, u'google', u'111222', u'g@gmail.com')
4.
5. >>> cur.scroll(1, "absolute")
6. >>> print cur.fetchone()
7. (2L, u'python', u'123456', u'python@gmail.com')
8.
9. >>> cur.scroll(0, "absolute")    #回到序号是0,即指向tuple的第一条
10. >>> print cur.fetchone()
11. (1L, u'qiwsir', u'123123', u'qiwsir@gmail.com')

```

至此，已经熟悉了`cur.fetchall()`和`cur.fetchone()`以及`cur.scroll()`几个方法，还有另外一个，接这上边的操作，也就是指

针在序号是1的位置，指向了tuple的第二条

```
1. >>> cur.fetchmany(3)
2. ((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（指针指向tuple的序号为1的位置，即第二条记录）开始，含当前位置，向下列出3条记录。

读取数据，好像有点啰嗦呀。细细琢磨，还是有道理的。你觉得呢？

不过，python总是能够为我们着想的，它的指针提供了一个参数，可以实现将读取到的数据变成字典形式，这样就提供了另外一种读取方式了。

```
1. >>> cur = conn.cursor(cursorclass=MySQLdb.cursors.DictCursor)
2. >>> cur.execute("select * from users")
3. 7L
4. >>> cur.fetchall()
5. ({'username': u'qiwsir', 'password': u'123123', 'id': 1L, 'email':
    u'qiwsir@gmail.com'}, {'username': u'mypythn', '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。可以这样来操作这个对象：

```
1. >>> cur.scroll(0, "absolute")
```

```

2. >>> for line in cur.fetchall():
3. ...     print line["username"]
4. ...
5. qiwsir
6. mypython
7. google
8. facebook
9. github
10. docker
11. 老齐

```

根据字典对象的特点来读取了“键-值”。

更新数据

经过前面的操作，这个就比较简单了，不过需要提醒的是，如果更新完毕，和插入数据一样，都需要`commit()`来提交保存。

```

1. >>> cur.execute("update users set username=%s where id=2",
2. ...     ("mypython"))
3. >>> cur.execute("select * from users where id=2")
4. >>> cur.fetchone()
5. >>> cur.fetchone()
6. (2L, u'mypython', u'123456', u'python@gmail.com')

```

从操作中看出来了，已经将数据库中第二条的用户名修改为mypython了，用的就是update语句。

不过，要真的实现在数据库中更新，还要运行：

```

1. >>> conn.commit()

```

这就大事完吉了。

用Python操作数据库 (3)

- 用Python操作数据库 (3)
 - 建立数据库
 - 关闭一切
 - 关于乱码问题

用Python操作数据库 (3)

通过python操作数据库的行为，除了能够完成前面两讲中的操作之外（当然，那是比较常用的），其实任何对数据库进行的操作，都能够通过python-mysqldb来实现。

建立数据库

在《用python操作数据库 (1)》中，我是通过 `mysql>` 写SQL语句，建立了一个名字叫做qiwsirtest的数据库，然后用下面的方式跟这个数据库连接

```
1. >>> import MySQLdb
2. >>> conn =
    MySQLdb.connect(host="localhost",user="root",passwd="123123",db="qiws
```

在上面的连接中，参数 `db="qiwsirtest"` 其实可以省略，如果省略，就是没有跟任何具体的数据库连接，只是连接了mysql。

```
1. >>> import MySQLdb
2. >>> conn =
    MySQLdb.connect("localhost","root","123123",port=3306,charset="utf8")
```


这种连接没有指定具体数据库，接下来就可以用类似 `mysql>` 交互模式下的方式进行操作。

```
1. >>> conn.select_db("qiwsirtest")
2. >>> cur = conn.cursor()
3. >>> cur.execute("select * from users")
4. 7L
5. >>> cur.fetchall()
6. ((1L, u'qiwsir', u'123123', u'qiwsir@gmail.com'), (2L, u'mypythons',
    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'))
```

用 `conn.select_db()` 选择要操作的数据库，然后通过指针就可以操作这个数据库了。其它的操作跟前两讲一样了。

如果不选数据库，而是要新建一个数据库，如何操作？

```
1. >>> cur = conn.cursor()
2. >>> cur.execute("create database newtest")
3. 1L
```

建立数据库之后，就可以选择这个数据库，然后在这个数据库中建立一个数据表。

```
1. >>> cur.execute("create table newusers (id int(2) primary key
    auto_increment, username varchar(20), age int(2), email text)")
2. 0L
```

括号里面是引号，引号里面就是创建数据表的语句，看官一定是熟悉的。这样就在newtest这个数据库中创建了一个名为newusers的表

```
1. >>> cur.execute("show tables")
```

```

2. 1L
3. >>> cur.fetchall()
4. ((u'newusers',),)

```

这是查看表的方式。当然，看官可以在 `mysql>` 交互模式下查看是不是存在这个表。如下：

```

1. mysql> use newtest;
2. Reading table information for completion of table and column names
3. You can turn off this feature to get a quicker startup with -A
4.
5. Database changed
6. mysql> show tables;
7. +-----+
8. | Tables_in_newtest |
9. +-----+
10. | newusers           |
11. +-----+
12. 1 row in set (0.00 sec)
13.
14. mysql> desc newusers;
15. +-----+-----+-----+-----+-----+-----+
16. | Field      | Type          | Null | Key | Default | Extra          |
17. +-----+-----+-----+-----+-----+-----+
18. | id         | int(2)        | NO   | PRI | NULL    | auto_increment |
19. | username  | varchar(20)   | YES  |     | NULL    |                |
20. | age       | int(2)        | YES  |     | NULL    |                |
21. | email     | text         | YES  |     | NULL    |                |
22. +-----+-----+-----+-----+-----+-----+
23. 4 rows in set (0.00 sec)

```

以上就通过python-mysqldb实现了对数据库和表的建立。

当然，能建就能删除。看官可以自行尝试，在这里就不赘述，原理就是在 `cur.execute()` 中写SQL语句。

关闭一切

当进行完有关数据操作之后，最后要做的就是关闭游标（指针）和连接。用如下命令实现：

```
1. >>> cur.close()  
2. >>> conn.close()
```

注意关闭顺序，和打开的顺序相反。

为什么要关闭？这个问题有点那个了。你把房子里面都收拾好了，如果离开房子，不关门吗？不要以为自己生活在那个理想社会。树欲静而风不止，小偷在行动。更何况，如果不关闭，服务器的内容总塞着那些东西而没有释放，早晚就满了。所以，必须关闭。必须的。

关于乱码问题

这个问题是编写web时常常困扰程序员的问题，乱码的本质来自于编码格式的设置混乱。所以，要特别提醒诸位注意。在用python-mysqldb的时候，为了放置乱码，可以做如下统一设置：

1. Python文件设置编码 utf-8（文件前面加上 #encoding=utf-8）
2. MySQL数据库charset=utf8（数据库的设置方法，可以网上搜索）
3. Python连接MySQL是加上参数 charset=utf8（在前面教程中都这么演示了，很重要）
4. 设置Python的默认编码为 utf-8
(sys.setdefaultencoding(utf-8)，这个后面会讲述)

代码示例：

```
1. #encoding=utf-8
2.
3. import sys
4. import MySQLdb
5.
6. reload(sys)
7. sys.setdefaultencoding('utf-8')
8.
9. db=MySQLdb.connect(user='root',charset='utf8')
```

MySQL的配置文件设置也必须配置成utf8 设置 MySQL 的 my.cnf 文件，在 [client]/[mysqld]部分都设置默认的字符集（通常在/etc/mysql/my.cnf）：

```
1. [client] default-character-set = utf8
2. [mysqld] default-character-set = utf8
```

windows操作系统请看官自己google。

python开发框架

- python开发框架
 - 框架的基本概念
 - python框架
 - Tornado
 - 安装Tornado

"One does not live by bread alone, but by every word that comes from the mouth of God"
—(MATTHEW4:4)

python开发框架

不管是python，还是php，亦或别的做web项目的语言，乃至做其它非web项目的开发，一般都要用到一个称之为什么什么框架的东西。

框架的基本概念

开发这对框架的认识，由于工作习惯和工作内容的不同，有很大差异，这里姑且截取[维基百科中的一种定义](#)，之所以要给出一个定义，无非是让看官有所了解，但是是否知道这个定义，丝毫不影响后面的工作。

软件框架 (Software framework)，通常指的是为了实现某个业界标准或完成特定基本任务的软件组件规范，也指为了实现某个软件组件规范时，提供规范所要求之基础功能的软件产品。

框架的功能类似于基础设施，与具体的软件应用无关，但是提供并实现最为基础的软件架构和体系。软件开发者通常依据特定的框架实现更为复杂的商业运用和业务逻辑。这样的软件应用可以在支持同一种框架的软件系统中运行。

简而言之，框架就是制定一套规范或者规则（思想），大家（程序员）在该规范或者规则（思想）下工作。或者说就是使用别人搭好的舞台，你来做表演。

我比较喜欢最后一句的解释，别人搭好舞台，我来表演。这也就是说，如果我在做web项目的时候，能够省却很多开发工作。的确是。所有，

做web开发，要用一个框架。

有高级工程师鄙视框架，认为自己编写的才是王道。这方面不争论，框架是开发中很流行的东西，我还是固执地认为用框架来开发，更划算。

python框架

有人说php(什么是php，严肃的说法，这是另外一种语言，更高雅的说法，是某个活动的汉语拼音简称)框架多，我不否认，php的开发框架的确很多很多。不过，python的web开发框架，也足够使用了，列举几种常见的web框架：

- Django:这是一个被广泛应用的框架，如果看官在网上搜索，会发现很多公司在招聘的时候就说要会这个，其实这种招聘就暴露了该公司的开发水平要求不高。框架只是辅助，真正的程序员，用什么框架，都应该是根据需要而来。当然不同框架有不同的特点，需要学习一段时间。
- Flask: 一个用Python编写的轻量级Web应用框架。基于 Werkzeug WSGI工具箱和Jinja2模板引擎。
- Web2py: 是一个为Python语言提供的全功能Web应用框架，旨在敏捷快速的开发Web应用，具有快速、安全以及可移植的数据库驱动的应用，兼容Google App Engine (这是google的元计算引擎，后面我会单独介绍)。
- Bottle: 微型Python Web框架，遵循WSGI，说微型，是因为它只有一个文件，除Python标准库外，它不依赖于任何第三方模块。
- Tornado: 全称是Torado Web Server，从名字上看就可知道它可以用作Web服务器，但同时它也是一个Python Web的开发框架。最初是在FriendFeed公司的网站上使用，FaceBook收购了之后便开源了出来。

- webpy：轻量级的Python Web框架。webpy的设计理念力求精简（Keep it simple and powerful），源码很简短，只提供一个框架所必须的东西，不依赖大量的第三方模块，它没有URL路由、没有模板也没有数据库的访问。

说明：以上信息选自：<http://blog.jobbole.com/72306/>，这篇文章中还有别的框架，由于不是web框架，我没有选摘，有兴趣的去阅读。

Tornado

一看到这个标题就知道，本教程中将选择使用这个框架。此前有朋友建议我用Django，首先它是一个好东西。但是，我更愿意用Tornado，为什么呢？因为.....，看下边或许是理由，也或许不是。

Tornado全称Tornado Web Server，是一个用Python语言写成的Web服务器兼Web应用框架，由FriendFeed公司在自己的网站FriendFeed中使用，被Facebook收购以后框架以开源软件形式开放给大众。看来Tornado的出身高贵呀，对了，如果是在天朝的看官，可能对Facebook有风闻，但是要一睹其芳容，还要努力。或者有人是不是怀疑这个地球上就没有这个网站呢？哈哈。按照某个地方的网络，它是存在的。废话不说，还是看Tornado的性能，因为选框架，一定要选好性能的，没准儿什么时候你也开发高大上的东西了。

Tornado的性能是相当优异的，因为它试图解决一个被称之为“C10k”问题，就是处理大于或等于一万的并发。一万呀，这可是不小的量。（关于C10K问题，看官可以浏览：[C10k problem](http://c10k.com/)）

下表是和一些其他Web框架与服务器的对比，供看官参考（数据来源：<https://developers.facebook.com/blog/post/301>）

条件：处理器为 AMD Opteron, 主频2.4GHz, 4核

服务	部署	请求/每秒
Tornado	nginx, 4进程	8213
Tornado	1个单线程进程	3353
Django	Apache/mod_wsgi	2223
web.py	Apache/mod_wsgi	2066
CherryPy	独立	785

看了这个对比表格，还有什么理由不选择Tornado呢？

就是它了——**Tornado**

安装Tornado

Tornado的官方网站：<http://www.tornadoweb.org>

在官网上，有安装方法，其实，看官也可以直接在官方上学习。另外，有一个中文镜像网站，看官也可以访问：

<http://www.tornadoweb.cn/>

我在自己电脑中 (ubuntu12.04), 用下面方法安装，只需要一句话即可：

```
1. pip install tornado
```

这是因为Tornado已经列入PyPI，因此可以通过 `pip` 或者 `easy_install` 来安装。

如果你没有安装 `libcurl` 的话，你需要将其单独安装到系统中。请参见下面的安装依赖一节。

如果不用这种方式安装，下面的页面中有可供看官下载的最新源码版本和安装方式：

<https://pypi.python.org/pypi/tornado/>

此外，在github上也有托管，看官可以通过上述页面进入到github看源码。

最后要补充一个要点，就是上述下载的Tornado无法直接安装在windows上，如果要在windows上安装，建议使用pypm（这是一个什么东西，关于这个东西，可以访问官方文档：<http://docs.activestate.com/activepython/2.6/pypm.html>，说实话，我也没有用过它，只是看了看文档罢了。看官如果有使用的，可以写一个教程共享之。），如下安装：

```
1. C:\> pypm install tornado
```

[首页](#) | [上一讲：用python操作数据库 3](#)

Hello,第一个网页分析

- Hello, 第一个网页分析
 - WEB服务器工作流程
 - 引入模块
 - 定义请求-处理程序类
 - main()方法
 - Application类
 - HTTPServer类
 - IOLoop类

As he walked by the sea of Galilee, he saw two brothers, Simon, who is called Peter, and Andrew his brother, casting a net into the sea—for they were fishermen. And he said to them, “Follow me, and I will make you fish for people.” Immediately they left their nets and followed him. (MATTHEW 5:18-20)

Hello, 第一个网页分析

打开文本编辑器。这里要说一下啦，理论上讲，所有的文本编辑器都可以做为编写程序的工具。前面已经提到的那个python IDE，是一个很好的工具，再有别的也行，比如我就用vim（好像我的计算机只能用vim了，上次运行Libre Office都很慢，敲一个键之后喝口水，才看到那个字母出来，等有人资助我了，也搞一个苹果的什么机器玩玩。）。用什么编辑工具，全是自己的喜欢罢了，不用争论那个好，这个差，只要自己顺手即可。

把下面的代码原封不动地复制过去，并且保存为文件名是hello.py的文件，存到那个目录中，自己选好了。

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

```

3.
4. import tornado.httpserver
5. import tornado.ioloop
6. import tornado.options
7. import tornado.web
8.
9. from tornado.options import define, options
10. define("port", default=8000, help="run on the given port",
        type=int)
11.
12. class IndexHandler(tornado.web.RequestHandler):
13.     def get(self):
14.         greeting = self.get_argument('greeting', 'Hello')
15.         self.write(greeting + ', welcome you to read:
        www.itdiffer.com')
16.
17. if __name__ == "__main__":
18.     tornado.options.parse_command_line()
19.     app = tornado.web.Application(handlers=[(r"/", IndexHandler)])
20.     http_server = tornado.httpserver.HTTPServer(app)
21.     http_server.listen(options.port)
22.     tornado.ioloop.IOLoop.instance().start()

```

进入到保存hello.py文件的目录，在shell或者命令输入框（windows可以用cmd）中，输入：

```
1. qw@qw-Latitude-E4300:~/codes$ python hello.py
```

用python运行这个文件，其实就已经发布了一个网站，只不过这个网站太简单了。

接下来，打开浏览器，在浏览器中输入：

<http://localhost:8000>，得到如下界面：



当然，如果还可以在shell中用下面方式运行：

```
1. qw@qw-Latitude-E4300:~$ curl http://localhost:8000/
2. Hello, welcome you to read: www.itdiffer.com
3.
4. qw@qw-Latitude-E4300:~$ curl http://localhost:8000/?greeting=Qiwsir
5. Qiwsir, welcome you to read: www.itdiffer.com
```

如果你的所有操作都正确，一定能够看到上面的结果。

恭喜你，迈出了决定性一步，已经可以用Tornado发布网站了。在这里似乎没有做什么部署，只是安装了Tornado。是的，不需要如同部署Nginx或者Apache那样，做各种设置了，因为Tornado就是一个很好的server，也是一个开发框架。

上面代码虽然跑起来了，但是每行都什么意思呢？下面就逐行解释，也就理解了Tornado这个框架的基本结构和用法。

WEB服务器工作流程

任何一个网站都离不开Web服务器，这里所说的不是指那个更计算机一样的硬件设备，是指里面安装的软件，有时候初次接触的看官容易搞混。就来伟大的[维基百科都这么说](#)：

有时，这两种定义会引起混淆，如Web服务器。它可能是指用于网站的计算机，也可能是指像Apache这样的软件，运行在这样的计算机上以管理网页组件和回应网页浏览器的请求。

在具体的语境中，看官要注意分析，到底指的是什么。

关于Web服务器比较好的解释，推荐看看百度百科的内容，我这里就不复制粘贴了，具体可以点击连接查阅：[WEB服务器](#)

在WEB上，用的最多的就是输入网址，访问某个网站。全世界那么多网站网页，如果去访问他们，怎么能够做到彼此互通互联呢。为了协调彼

此，就制定了很多通用的协议，其中http协议，就是网络协议中的一种。关于这个协议的介绍，网上随处就能找到，请看官自己google.

网上偷来的一张图（从哪里偷来的，我都告诉你了，多实在呀。哈哈。），显示在下面，简要说明web服务器的工作过程



偷个彻底，把原文中的说明也贴上：

1. 创建listen socket，在指定的监听端口，等待客户端请求的到来
2. listen socket接受客户端的请求，得到client socket，接下来通过client socket与客户端通信
3. 处理客户端的请求，首先从client socket读取http请求的协议头，如果是post协议，还可能要读取客户端上传的数据，然后处理请求，准备好客户端需要的数据，通过client socket写给客户端

剽窃就此结束，下面就自己写了。

引入模块

```
1. import tornado.httpserver
2. import tornado.ioloop
3. import tornado.options
4. import tornado.web
```

这四个都是Tornado的模块，在本例中都是必须的。它们四个在一般的网站开发中，都要用到，基本作用分别是：

- tornado.httpserver：这个模块就是用来解决web服务器的http协议问题，它提供了不少属性方法，实现客户端和服务端

的互通。Tornado的非阻塞、单线程的特点在这个模块中体现。

- `tornado.ioloop`: 这个也非常重要，能够实现非阻塞socket循环，不能互通一次就结束呀。
- `tornado.options`: 这是命令行解析模块，也常用到。
- `tornado.web`: 这是必不可少的模块，它提供了一个简单的Web框架与异步功能，从而使其扩展到大量打开的连接，使其成为理想的长轮询。

还有一个模块引入，是用`from...import`完成的

```
1. from tornado.options import define, options
2. define("port", default=8000, help="run on the given port",
    type=int)
```

这两句就显示了所谓“命令行解析模块”的用途了。在这里通

过 `tornado.options.define()` 定义了访问本服务器的端口，就是当在浏

览器地址栏中输入 `http 8000` 的时候，才能访问本网站，因为http协议默认的端口是80，为了区分，我在这里设置为8000，为什么要区分呢？因为我的计算机或许你的也是，已经部署了别的注入Nginx服务器了，它的端口是80，所以要区分开，并且，后面我们还会将tornado和Nginx联合起来工作，这样两个服务器在同一台计算机上，就要分开喽。

定义请求-处理程序类

```
1. class IndexHandler(tornado.web.RequestHandler):
2.     def get(self):
3.         greeting = self.get_argument('greeting', 'Hello')
4.         self.write(greeting + ', welcome you to read:
    www.itdiffer.com')
```

所谓“请求处理”程序类，就是要定义一个类，专门应付客户端向服务器提出请求（这个请求也许是要读取某个网页，也许是要将某些信息存到服务器上），服务器要有相应的程序来接收并处理这个请求，并且反馈某些信息（或者是针对请求反馈所要的信息，或者返回其它的错误信息等）。

于是，就定义了一个类，名字是IndexHandler，当然，名字可以随便取了，但是，按照习惯，类的名字中的单词首字母都是大写的，并且如果这个类是请求处理程序类，那么就最好用Handler结尾，这样在名称上很明确，是干什么的。

类IndexHandler的参数是 `tornado.web.RequestHandler`，这个参数很重要，是专门用于完成请求处理程序的，通过它定义 `get()` 和 `post()` 两个在web中应用最多的方法的内容（关于这两个方法的详细解释，可以参考：[HTTP GET POST的本质区别详解](#)，作者在这篇文章中，阐述了两个方法的本质）。

在本例中，只定义了一个 `get()` 方法。请看官注意，类中的方法可以没有别的参数，但是必须有 `self` 这个参数，关于这点请参与前面几篇关于类的讲授内容（返回[首页](#)找相关文章）。

在 `greeting = self.get_argument('greeting', 'Hello')` 这句中，当实例化之后，`self` 对应的就是 `tornado.web.RequestHandler`，而 `get_argument` 则是 `tornado.web.RequestHandler` 的一个方法。官方文档对这个方法的描述如下：

```
RequestHandler.get_argument(name, default=, []strip=True)
```

```
Returns the value of the argument with the given name.
```

```
If default is not provided, the argument is considered to be required, and we raise a MissingArgumentError if it is missing.
```

```
If the argument appears in the url more than once, we return the
```

last value.

The returned value is always unicode.

这段描述已经很清晰了，此外，看完这段说明，看官是否明白我在前面运行的：

1. `qw@qw-Latitude-E4300:~$ curl http://localhost:8000/?greeting=Qiwsir`
2. `Qiwsir, welcome you to read: www.itdiffer.com`

为什么通过 `http://localhost:8000/?greeting=Qiwsir`，就可以实现对greeting的赋值。

接下来的那句 `self.write(greeting + ',welcome you to read: www.itdiffer.com)'` 中，write也是 `tornado.web.RequestHandler` 的一个方法，这发方法主要功能是向客户端反馈参数中的信息。也浏览一下官方文档信息，对以后正确理解使用有帮助：

`RequestHandler.write(chunk)[source]`

Writes the given chunk to the output buffer.

To write the output to the network, use the `flush()` method below.

If the given chunk is a dictionary, we write it as JSON and set the Content-Type of the response to be application/json. (if you want to send JSON as a different Content-Type, call `set_header` after calling `write()`).

main()方法

`if __name__ == "__main__"`，从这句话开始执行编写的程序，前面相当于预备工作吧。这个方法跟以往执行python程序是一样的。

`tornado.options.parse_command_line()`，这是在执行tornado的解析命令行。在tornado的程序中，只要import模块之后，就会在运行的时候自动加载，不需要了解细节，但是，在main()方法中如果有命令行

解析，必须要提前将模块引入。

Application类

下面这句是重点：

```
1. app = tornado.web.Application(handlers=[(r"/", IndexHandler)])
```

将tornado.web.Application实例化。这个实例化，本质上是建立了整个网站程序的请求处理集合，然后它可以被HTTPServer做为参数调用，实现http协议服务器访问。Application类的 `__init__` 方法参数形式：

```
1. def __init__(self, handlers=None, default_host="",
    transforms=None, **settings):
2.     pass
```

在一般情况下，handlers是不能为空的，因为Application类通过这个参数的值处理所得到的请求。例如在本例中， `handlers=[(r"/", IndexHandler)]`，就意味着如果通过浏览器的地址栏输入根路径（ `http://localhost:8000` 就是根路径，如果是 `http://localhost:8000/qiwsir`，就不属于根，而是一个子路径或目录了），对应这就是让名字为IndexHandler类处理这个请求。

通过handlers传入的数值格式，一定要注意，在后面做复杂结构的网站是，这里就显得重要了。它是一个list，list里面的参数是列表，列表的组成包括两部分，一部分是请求路径，另外一部分是处理程序的类名称。注意请求路径可以用正则表达式书写。举例说明：

```
1. handlers = [
2.     (r"/", IndexHandlers),           #来自根路径的请求用
    IndesHandlers处理
```

```

3.     (r"/qiwsir/(.*)", QiwsirHandlers), #来自/qiwsir/以及其下任何请求
      (正则表达式表示任何字符) 都由QiwsirHandlers处理
4. ]

```

注意

在这里我使用了 `r"/"` 的样式，意味着就不需要使用转义符，`r`后面的都表示该符号本来的含义。例如，`\n`，如果单纯这么来使用，就以为着换行，因为符号“`\`”具有转义功能（关于转义详细阅读《玩转字符串（1）》），当写成 `r"\n"` 的形式是，就不再表示换行了，而是两个字符，`\`和`n`，不会转意。一般情况下，由于正则表达式和 `\` 会有冲突，因此，当一个字符串使用了正则表达式后，最好在前面加上‘`r`’。（关于正则表达式，看官姑且网上搜索，在后面的课程中，我也会介绍）

关于Application类的介绍，告一段落，但是并未完全讲述了，因为还有别的参数设置没有讲，看官有兴趣可以阅读官方的文档资料，地址是：http://tornado.readthedocs.org/en/latest/_modules/tornado/web.html#Application

HTTPServer类

实例化之后，Application对象（用app做为标签的）就可以被另外一个类HTTPServer引用，形式为：

```

1. http_server = tornado.httpserver.HTTPServer(app)

```

HTTPServer是tornado.httpserver里面定义的类。HTTPServer是一个单线程非阻塞HTTP服务器，执行HTTPServer一般要回调Application对象，并提供发送响应的接口，也就是下面的内容是跟随上面语句的（options.port的值在IndexHandler类前面通过from...import..设置的）。

```
1. http_server.listen(options.port)
```

这种方法，就建立了单进程的http服务。

请看官牢记，如果在以后编码中，遇到需要多进程，请参考官方文档说明：<http://tornado.readthedocs.org/en/latest/httpserver.html#http-server>

IOLoop类

剩下最后一句了：

```
1. tornado.ioloop.IOLoop.instance().start()
```

这句话，总是在 `__main__` 的最后一句。表示可以接收来自HTTP的请求了。

以上把一个简单的hello.py剖析。想必读者对Tornado编写网站的基本概念已经有了。

[首页](#) | [上一讲：PYthon框架](#)

实例分析get和post

- 探析get和post方法
 - `get()`
 - `post()`方法
 - 简要总结RequestHandler

"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 and rust consumes and where thieves do not break in and steal. For where your treasure is, there your heart will be also." (MATTHEW6:19-21)

探析get和post方法

在开发网站的过程中，post和get是常见常用的两个方法，关于这两个方法的详细解释，请列为阅读这篇文章：[《HTTP POST GET 本质区别详解》](#)，这篇文章前面已经推荐阅读了，可以这么说，如果看官没有搞明白get和post，也可以写出web程序，但是，只要遇到“但是”，就说明另有乾坤，但是如果看官要对这方面有深入理解，并且将来能上一个档次，是必须了解的。这就如同你要练习辟邪剑谱中的剑法，不自宫，也可以练，但是无法突破某个极限，岳不群也不傻，最终他要成为超一流，就不惜按照剑谱中开篇所说“欲练神功，挥刀自宫”，“神功”是需要“自宫”为前提，否则，练出来的不是“神功”，无法问鼎江湖。

特别提醒，看官不要自宫，因为本教程不是辟邪剑谱，也不是葵花宝典，撰写本课程的人更是生理健全者。若看官自宫了，责任自负，与本作者无关。直到目前，科学上尚未有证实或证伪自宫和写程序之间是否存在某种因果关系。所以提醒看官慎重行事。

还是扯回来，看下面的代码先：

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  import textwrap
5.
6.  import tornado.httpserver
7.  import tornado.ioloop
8.  import tornado.options
9.  import tornado.web
10.
11. from tornado.options import define, options
12. define("port", default=8000, help="Please send email to me",
13.       type=int)
14.
15. class ReverseHandler(tornado.web.RequestHandler):
16.     def get(self, input_word):
17.         self.write(input_word[::-1])
18.
19. class WrapHandler(tornado.web.RequestHandler):
20.     def post(self):
21.         text = self.get_argument("text")
22.         width = self.get_argument("width", 40)
23.         self.write(textwrap.fill(text, width))
24.
25. if __name__ == "__main__":
26.     tornado.options.parse_command_line()
27.     app = tornado.web.Application(
28.         handlers = [
29.             (r"/reverse/(\w+)", ReverseHandler),
30.             (r"/wrap", WrapHandler)
31.         ]
32.     )
33.     http_server = tornado.httpserver.HTTPServer(app)
34.     http_server.listen(options.port)
35.     tornado.ioloop.IOLoop.instance().start()

```

这段代码跟上一讲的代码相比，基本结构是一样的，但是在程序主体

中，这次写了两个类 `ReverseHandler` 和 `WrapHandler`，这两个类中分别有两个方法 `get()` 和 `post()`。在 `tornado.web.Application()` 实例化中，`handlers` 的参数值分别设置了不同路径对应这两个类。

其它方面跟上一讲的代码一样。

把上述代码的文件，存到某个目录下，我给他取名：`request_url.py`，名字看官也可以自己定。然后进入该目录，运行：`python request_url.py`，就将这个tornado框架的网站以端口8000发布了。

打开网页，在浏览器中输入：

`http://localhost:8000/reverse/qiwsirpython`

界面上输出什么结果？



还可以在命令终端，用下面方式调试，跟在网页上输出是等同效果。

```
1. qw@qw-Latitude-E4300:~$ curl
   http://localhost:8000/reverse/qiwsirpython
2. nohtypriswiq
```

再看另外一个路径，看官运行的是否是下面的结果呢？

```
1. qw@qw-Latitude-E4300:~$ curl http://localhost:8000/wrap -d
   text=I+love+Python+programming+and+I+am+writing+python+lessons+on+lin
2. I love Python programming and I am
3. writing python lessons on line
```

调试通过，就开始分析其中的奥妙。

get()

在ReverseHandler类中，定义了这个方法。

```
1. class ReverseHandler(tornado.web.RequestHandler):
2.     def get(self, input_word):
3.         self.write(input_word[::-1])
```

这个get()方法要和下面Application实例化中的路径：

```
1. (r"/reverse/(\w+)", ReverseHandler),
```

关联起来看。

首先看路径设置：`r"/reverse/(\w+)"`，这个路径的意思就是可以在浏览器的url中输入：<http://localhost:8000/reverse/dddd>，这个样子的地址，注意路径中的`(\w+)`，是正则表达式，在reverse/的后面可以输入一个或者多个包括下划线的任何单词字符。也就是dddd可以更换为任何其它字母或者下划线，一个或者多个均可以。

在URL中输入的这个地址，就被ReverseHandler类中的get()方法接收，这就是`(r"/reverse/(\w+)", ReverseHandler)`之含义了。那么，ReverseHandler中的get()方法如何接收url中传递过来的信息呢？

前文已经说过，在`def get(self, input_word)`中，self参数在类实例化后对应的是tornado.web.RequestHandler，另外一个参数input_word用来接收来自url的信息，但是它只接收所设置的路径尾部数据，也就是路径`r"/reverse/(\w+)"`中reverse后面的第一个分割符号“/”之后的内容，都被input_word接收过来，即正则表达式的内容。

input_word接收过来的对象，是什么类型呢？猜测一下，从前面程序的运行结果看，肯定是某种序列类型的对象。具体是哪种呢？可以实

验。

将get方法修改为：

```
1. def get(self, input_word):
2.     input_type = type(input_word)
3.     self.write("%s"%input_type)
```

再运行程序，打印出来的是：

```
1. qw@qw-Latitude-E4300:~$ curl http://localhost:8000/reverse/qiwei
2. <type 'unicode'>
```

这说明，get()方法通过URL接收到的数据类型是unicode编码的字符，即字符串。

原来类方法中的 `self.write(input_word[::-1])` 含义是，将原来的字符串倒置，并返回该数据到客户端（如页面）。

```
1. >>> a = "python,laoqi"
2. >>> a[::-1]
3. 'iqoal,nohtyp'
4. >>> b = [1,2,3,4]
5. >>> b[::-1]
6. [4, 3, 2, 1]
7. >>> c = ("a","b","c")
8. >>> c[::-1]
9. ('c', 'b', 'a')
```

这是一种将序列类型对象倒置的一种方法。

总结一下：get()通过第二个参数，获得已经设置的显示路径中最后一个/后面的数据，并且为unincode编码的字符。

这种方式通过URL得到有关数据，也就是说在URL中，只需要

以 `http://localhost/aaa/bbb/ccc` 的形式来显示路径即可。看官是否注意到，有的网站是这么显示的：`http://localhost/aaa?name=Tom&&?age=25`，这其实是两种不同的规范也好、方法也罢的东西，前者更接近时下流行的REST规范，可能看官听说过MVC吧，我听不少的公司都强调网站要符合MVC规范，殊不知，更流行的是REST了。那么到底那个好呢？我的观点：It depends.如果有兴趣，请阅读：[《理解本真的REST架构风格》](#)，对REST了解一二。

post()方法

post()也是web上常用的方法，在本例中，该方法写在了WrapHandler类中：

```
1. class WrapHandler(tornado.web.RequestHandler):
2.     def post(self):
3.         text = self.get_argument("text")
4.         width = self.get_argument("width", 40)
5.         self.write(textwrap.fill(text, width))
```

对应的Application类路径：

```
1. (r"/wrap", WrapHandler)
```

但是，看官要注意，post()无法从URL中获得数据。这是跟get()方法非常不一样的。关于get和post之间的区别，请看官点击[《HTTP POST GET 本质区别详解》](#)阅读。

客户端的数据通过post方法发送到服务器，这个内在过程就是由所谓HTTP协议完成，不用去管它，因为现在我们只是研究应用层，不去深入网络协议的层面。看官可以有这样的以为：怎么传的数据，但是我也可以不讲，就算我也不会吧。不过，如果看官非要了解，请问google

大神。

我要解释的是，`post()`方法怎么接收到客户端传过来的数据。

因为post不能从URL中得到数据，所以就不能用类似的方式在网页的url中输入要传给它的数据了，只能这样来测试：

```
1. qw@qw-Latitude-E4300:~$ curl http://localhost:8000/wrap -d
   text=I+love+Python+programming+and+I+am+writing+python+lessons+on+line
2. I love Python programming and I am
3. writing python lessons on line
```

请看官注意，URL依然是 `http://localhost:8000/wrap`，后面的部分 `-d text=...`，就是向这个地址对应的类`WrapHandler`中的`post`方法传送相应的数据，这个数据被 `tornado.web.RequestHandler` 中的 `get_argument()`方法获得，也就是通过 `text=self.get_argument("text")` 得到传过来的对象，并赋值给 `text`。

这里需要提醒看官注意，`self.get_argument("text")` 的参数中，是 `"text"`，就意味着，传入数据的时候，需要用`text`这个变量，即必须写成 `text=...`。如果 `self.get_argument("word")`，那么就应该是 `word=...` 方式传入数据了。

看官此时是否已经晓得，`get_argument()`在`post`方法中，能够获得客户端传过来的数据，当然是unicode编码的。得到这个数据之后，就可以按照自己的需要进行操作了。

下一句 `width = self.get_argument("width", 40)` 是要返回一个对象，这个对象约定变量为40，并将它用在下面的 `textwrap.fill(text, width)` 中。这里并没有什么特别支出，也可以写成 `width = 40`，其实就是给`textwrap.fill()`提供参数罢了。关于`textwrap`模块中的

fill方法，可以用help命令来看看。

```

1. >>> import textwrap
2. >>> help(textwrap.fill)
3.
4. Help on function fill in module textwrap:
5.
6. fill(text, width=70, **kwargs)
7.     Fill a single paragraph of text, returning a new string.
8.
9.     Reformat the single paragraph in 'text' to fit in lines of no
    more
10.    than 'width' columns, and return a new string containing the
    entire
11.    wrapped paragraph. As with wrap(), tabs are expanded and other
12.    whitespace characters converted to space. See TextWrapper
    class for
13.    available keyword args to customize wrapping behaviour.

```

简要总结RequestHandler

RequestHandler就是请求处理程序的方法，从上面的流程中，可以简要地初步地认为（深奥的东西还不少，这里只能是简要地初步地肤浅地，随着学习的深入会一点点深入地）：

- 通过 `self.write()` 向客户端返回数据
- `get()` 中，以一个参数从URL路径末尾获取数据，特别提醒看官，这是在本讲的例子中，`get()`方法中，用第二个参数获得url数据。在上一讲中，同样是`get()`方法，用到了 `greeting = self.get_argument('greeting', 'Hello')`，于是不需要在`get()`中另外写参数，只需要通过“greeting”就可以得到URL中的数据，不过这时候的url应该写成 `http://localhost:8000/?greeting=PYTHON` 的样式，于是字符传‘PYTHON’就能够让`get()`中

的 `self.get_argument('greeting', 'Hello')` 获得，如果没有，就是 'Hello'。

- `post()` 中，以 `self.argument("text")` 的形式得到 `text` 为标签提交的数据。

`get` 和 `post` 是 `http` 中用的最多的方法啦。此外，Tornado 也还支持其它的 HTTP 请求，如：PUT、DELETE、HEAD、OPTIONS。在具体编程的时候，如果看官用到，可以搜索，一般用的不多。

最后交代一句，`get` 和 `post` 方法，由于一个是通过 URL 得到数据，另外一个不是，所以，他们可以写到同一个类中，彼此互不干扰。

还要说明，我在这部分参考了一本书的讲授内容，特别是其中的代码例子，这本书就是《[Introduction to Tornado](#)》

[首页](#) | [上一讲: Hello, 第一个网页分析](#)

问候世界

- [问候世界](#)
 - [官方网站](#)
 - [注册](#)
 - [下载SDK](#)
 - [在本地建立项目](#)
 - [编写main.py](#)

“Do not judge, so that you may not be judged. For with the judgement you make you will be judged, and the measure you give will be the measure you get.” (MATTHEW 7:1-2)

问候世界

按照作家韩寒的说法，这个世界存在两种逻辑，一种是逻辑，另外一种是中国逻辑。中国由于“特色”，跟世界是不同的。在网络上，也是如此，世界的网络和中国网络有很大不同，不用多说，看官应该体会到了。那么，我这里的标题是“问候世界”，就当然不包括中国了。

已经用Tornado可以在自己的计算机上发布网站了，也就是能够在自己计算机的浏览器地址栏中输入 `http://localhost:8000`，显示那个Hello，这仅仅是向自己问候了。如果要向世界问候，就需要把这个那个程序放到连接到互联网的服务器上。

在网上，有很多提供服务器租赁服务的公司，可以购买虚拟空间、VPS，现在比较时髦的就是购买云服务主机等，当然，有钱的就自己买服务器硬件设备，然后自己建立机房了。我这里演示给诸位的，不是上面这些，是按照穷人的思路来解决问题。

Google是伟大的，谁要不认同，我就跟谁急。除了因为它是一个好的搜索引擎之外，还因为它给我提供了免费的GAE。什么是GAE？GAE就

是Google App Engine，维基百科上这样说的：

*Google App Engine*是一个开发、托管网络应用程序的平台，使用*Google*管理的数据中心。它在2008年4月发布了第一个*beta*版本。

*Google App Engine*使用了云计算技术。它跨越多个服务器和数据中心来虚拟化应用程序。
[1] 其他基于云的平台还有*Amazon Web Services*和微软的*Azure*服务平台等。

*Google App Engine*在用户使用一定的资源时是免费的。支付额外的费用可以获得应用程序所需的更多的存储空间、带宽或是*CPU*负载。

看官注意了，上面那诱人的“免费”，尽管有所限制，但是，已经足够一般用户使用了，下面是免费内容，看看是不是足够慷慨了呀。

项目	配额
每天的Email数量	100封
每天的输入数据	无限
每天的输出数据	1 GB
每天可使用CPU	28小时
每天调用Datastore API次数	50000次
数据存储	1 GB
每天调用URLFetch API次数	657000次

如果你做一个网站，超过了上面的免费配额，说明你的网站已经不小了。也能够挣钱或者找风险投资了。中国的互联网上，尚未见到如此慷慨的，虽然也有自吹自擂的公司。

请看官注意，这个服务只能在世界范围内使用，由于你知道和不知道的原因，在中国不能使用。当然，要立志做一个优秀程序员的，一定要能够进入世界范围。

欲进入世界，必科学上网

否则，今天这一讲看官就无法学习。

官方网站

在进入学习之前，请看官登录GAE官方网站浏览：

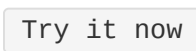
<https://cloud.google.com/appengine/docs?hl=zh-cn>

目前GAE支持使用的语言有：Java, Python, PHP, Go

我在这里当然是使用python了。虽然，我也喜欢PHP。

如果看官自己有能力阅读文档，直接在该网站上阅读，就能够学会如何使用GAE了，不必看我下面的啰嗦。我认为GAE网站上讲的很好了。不过，就是有一点不足，缺少趣味性。如果要享受胡扯的东西，就看我的课程啦。

注册

在上面的GAE官方文档中，看到下面图片中的  按钮，点它，进入下一个界面。



进入的新页面中，开头有这样一段。

Try Google App Engine Now

Creating an App Engine app is easy, and it's free to start. Upload your app and share it with users right away, at no charge and with no commitment required.

看下面的图



首先给自己的网站取一个名字，当然这个名字以后还可以修改呢。

然后，在四种语言中选择一种，一定要选python，选别的不跟你玩了。

选好之后，看第三步，如下图。其实是一个例子罢了。不过，不是用tornado框架的。



下载SDK

下载SDK，看操作系统，不同的操作系统有不同的下载安装方法。如下两图所示中，都说明了安装流程。



我的操作系统是ubuntu，就选择按照上面那张图的方式安装。特别提醒，一定要让你的计算机用VPN科学上网，才能实现上面的安装流程。

从新打开一个shell，按照上面要求，输入 `curl`

`https://sdk.cloud.google.com/ | bash`，剩下的事情就是根据shell中提示的信息向下进行了。它要询问是否需要帮助的时候，就输入y，然后选择语言（2，是python and php，也就是按照这种方法安装的SDK是python和php两者都可以使用的），还要输入一个放置SDK文件的目录名称。之后，系统就自动将google-cloud-sdk中的很多东西，放到指定目录中了。在安装过程中，还会询问工作环境，不修改用自动设置的，简单地回车即可。

如果网络差点，需要等待一段时间。最终在你指定的目录中会有一个google-cloud-sdk目录，所要安装的内容全在里面了。

还可以

到：<https://cloud.google.com/appengine/downloads?hl=zh-cn>

下载，这个页面也有安装帮助。安装方法可以是：

1. 将下载的文件解压，并存储到本地计算机中的某个目录内，比如我在前面安装的时候，就个放置google-cloud-sdk的目录命名为GAEPython，这里所说的目录，就跟前面的效能一样。
2. 设置本用户的工作环境，就是要在当前用户主目录下的.bashrc中设置路径。方法是：用户主目录下的.bashrc文件（有的linux是.profile文件），用vim打开该文件，并进入编辑模式，在文件最后加入PATH设置，参考格式 `export PATH="$PATH:your path1/:your path2/ ..."`，然后保存，并退出该shell，则变量生效，本添加的变量只对当前用户有效。

这样两种方式折腾，就搞定了SDK

在上图中，官方给的安装流程中，接下来就是要用一个账号登录google云。关闭前面安装时使用的shell，然后新打开一个，输入命令 `gcloud auth login`，会打开浏览器，出现下图



我也不用别的账号，就直接用“老齐”那个账号，点击之，进入一个授权界面，也不用怎么仔细看，要想用，就得点击网页最下面的“接受”，否则别用。网站都是这么忽悠人的。我不截图了，看官除了要科学上网，这里不会有问题的吧。然后看到下面的界面，就成功了。



在这个页面中，可以了解很多关于google云平台的内容以及操作方法。

因为我以前已经建立了一下项目，所以，我点击这个页面上“转至我的控制台”之后，会列出我已经存在的项目名称。当然，在这里可以新建

项目。

不过，这里暂且搁置。先回到自己的计算机上。已经将gdk做好了。

在本地建立项目

在自己的计算机上，任何认为合适的地方，建立一个目录，这个目录就是你要发布到GAE上的项目名称，例如我的操作：

```
1. qw@qw-Latitude-E4300:~/Documents$ mkdir mypy
```

我是在这里建立了一个名字是mypy的目录，也就意味着我要发布的项目名称是mypy，然后，要把tornado的东西放进这个目录。就是这么做：

先建立一个目录，叫做tornado_source

```
1. qw@qw-Latitude-E4300:~/Documents$ cd tornado_source
2. qw@qw-Latitude-E4300:~/Documents/tornado_source$ git clone
   https://github.com/facebook/tornado.git
3. 正克隆到 'tornado'...
4. remote: Counting objects: 13952, done.
5. remote: Total 13952 (delta 0), reused 0 (delta 0)
6. 接收对象中: 100% (13952/13952), 7.41 MiB | 107.00 KiB/s, done.
7. 处理 delta 中: 100% (8877/8877), done.
8. 检查连接... 完成。
```

然后将tornado_source中的tornado目录，移动到mypy里面去。

接下来在mypy目录里面新建一个文件，名称必须是：app.yaml，该文件的内容是：

```
1. application: mypy
2. version: 1
3. runtime: python27
```

```

4.  api_version: 1
5.  threadsafe: no
6.
7.  handlers:
8.
9.  - url: /favicon\.ico
10.    static_files: favicon.ico
11.    upload: favicon\.ico
12.
13.  - url: .*
14.    script: main.py

```

关于yaml，这里也引用[维基百科的内容](#)，补充一下：

YAML (IPA: /'jæməɪl/, 尾音类似camel骆驼) 是一个可读性高，用来表达资料序列的格式。YAML参考了其他多种语言，包括：XML、C语言、Python、Perl以及电子邮件格式RFC2822。Clark Evans在2001年首次发表了这种语言[1]，另外Ingy döt Net与Oren Ben-Kiki也是这语言的共同设计者。目前已经有数种编程语言或脚本语言支援（或者说解析）这种语言。

YAML是“YAML Ain't a Markup Language” (YAML不是一种置标语言) 的递归缩写。在开发的这种语言时，YAML 的意思其实是：“Yet Another Markup Language” (仍是一种置标语言)，但为了强调这种语言以数据做为中心，而不是以置标语言为重点，而用返璞词重新命名。

YAML的官方网站：www.yaml.org

编写main.py

本来根据我以前的经验，到这里就可以运行 `dev_appserver.py mypy` 了。可是发现错误了，说没有找到sqlite3模块。不知道为什么，python升级之后，这个模块没有装上，本来这个模块是标准库的。看官如果在操作过程中，也遇到类似情况。解决方法就是到该模块的官方网站下载源码，然后安装，一般流程是：

```

1.  tar -zxvf ***.gz
2.  cd ***      #进入解压缩后得到的目录

```

```

3. ./configure      #如果不使用默认的, 可以指定目录: --prefix=**/**/
4. make
5. make install    #若权限不足, 可以前面加sudo, 得到临时root权限。

```

然后, 这个很重要, 再将python源码中setup.py文件中该模块的路径从新设置, 设置为该模块被安装的路径。一般情况可能是这里的模块路径错了。

接下来就从新安装python

搞定sqlite3模块, 再尝试, 我回到含有mypy项目的
(/home/qw/Documents/mypy, 即回
到/home/qw/Documents) :

```

1. dev_appserver.py mypy

```

结果显示:

```

1. ....
2. google.appengine.tools.devappserver2.wsgi_server.BindError: Unable
   to bind localhost:8080

```

原来, 我科学上网, 用的是8080端口, 已经被占用了,
google.appengine也打算用这个端口。那好, 因为是在本地, 就不用科学了, 将科学上网端口停了, 再试试。

```

1. qw@qw-Latitude-E4300:~/Documents$ dev_appserver.py mypy
2. INFO      2014-10-07 13:57:14,275 devappserver2.py:733] Skipping SDK
   update check.
3. WARNING   2014-10-07 13:57:14,279 api_server.py:383] Could not
   initialize images API; you are likely missing the Python "PIL"
   module.
4. INFO      2014-10-07 13:57:14,283 api_server.py:171] Starting API
   server at: http://localhost:40433
5. INFO      2014-10-07 13:57:14,295 dispatcher.py:186] Starting module

```

```

"default" running at: http://localhost:8080
6. INFO      2014-10-07 13:57:14,298 admin_server.py:117] Starting
   admin server at: http://localhost:8000
7.
8. /usr/local/lib/python2.7/site-packages/setuptools-2.2-
   py2.7.egg/pkg_resources.py:991: UserWarning: /home/qw/.python-eggs
   is writable by group/others and vulnerable to attack when used with
   get_resource_filename. Consider a more secure location (set with
   .set_extraction_path or the PYTHON_EGG_CACHE environment variable).

```

看下图，如果在URL输入 `http://localhost:8000`，打开的是本地App Engine



这个就是本地的server。

如果在URL输入 `http://localhost:8080`，网页空白，什么提示没有。这就对了，因为我还没有编写那个最重要的文件，就是app.yaml里面设定的main.py

特别提醒看官：我在上面操作的时候，出现了警告，但是当时没有引起我的注意。于是就编写main.py文件。结果，运行 `http://localhost:8080`，不成功。而且告诉我无法找到tornado.wsgi模块。

本讲有悬念了。

使用表单和模板

- 使用表单和模板
 - 一个表单
 - 后端处理程序
 - 显示结果
 - 运行结果

But when he heard this, he said: "Those who are well have no need of a physician, but those who are sick. Go and learn what this means, 'I desire mercy, not sacrifice' For I have come to call not the righteous but sinners." (MATTHEW 9:12)

使用表单和模板

如果像前面那么做网站，也太丑陋了。并且功能也不多。

在实际做网站中，现在都要使用一个模板，并且在用户直接看到的页面，用html语言来写页面（关于HTML，本教程默认为看官已经熟悉，如果不熟悉，可以到找有关教程来学习）。

在做网站的行业里面，常常将HTML+CSS+JS组成的网页，称作“前端”。它主要负责展示，或者让用户填写一些表格，通过JS提交给用python写的程序，让python程序来处理数据，那些处理数据的python程序称之为“后端”。我常常提醒做“后端”的，不要轻视“前端”。如果看官立志成为全栈工程师，就要从前到后都通。

在本讲中，为了突出模板和后端程序，我略去CSS+JS，虽然这样一来界面难看，而且用户的友好度也不怎么样（JS，javascript是使网页变得更友好的重要工具，如不用更换地址就能刷新页面等等，特别提醒看官，一定要学好javascript，虽然这个可能没有几个大学教的。请看维基百科对javascript简介：）。

JavaScript，一种直译式脚本语言，是一种动态类型、弱类型、基于原型的语言，内置支持类。它的解释器被称为*JavaScript*引擎，为浏览器的一部分，广泛用于客户端的脚本语言，最早是在*HTML*网页上使用，用来给*HTML*网页增加动态功能。然而现在*JavaScript*也可被用于网络服务器，如*Node.js*。

在1995年时，由网景公司的布兰登·艾克，在网景导航者浏览器上首次设计实作而成。因为网景公司与升阳公司合作，网景公司管理层次结构希望它外观看起来像*Java*，因此取名为*JavaScript*。但实际上它的语法风格与*Self*及*Scheme*较为接近。

为了取得技术优势，微软推出了*JScript*，*CEnv*推出*ScriptEase*，与*JavaScript*同样可在浏览器上运行。为了统一规格，1997年，在*ECMA*（欧洲计算机制造商协会）的协调下，由*Netscape*、*Sun*、微软、*Borland*组成的工作组确定统一标准：*ECMA-262*。因为*JavaScript*兼容于*ECMA*标准，因此也称为*ECMAScript*。

上面这段引文里面提到了一个非常著名的公司：网景，可能年青一代都忘却了。也建议有兴趣的看官到维基百科中了解这个传奇公司，它曾经有一个传奇产品，虽然名字不复存在，但是*Firefox*是秉承它衣钵的。

话题再转回来，还是关于本讲，主要是要演示一个用模板（*HTML*）写一个表单，然后提交给后端的*python*程序，再转到另外一个显示的前端页面显示。为了简化流程，这个过程中没有数据处理和*CSS+Javascript*的工作，所有界面会丑陋。但是，“我很丑，可是我很温柔”。

一个表单

要做一个前端的页面，显示的内容就如同下图样式



相应代码是，并命名为*index.html*，存在一个名称是*template*的目录中。

```
1. <!DOCTYPE html>
2. <html>
3.     <head>
```

```

4.         <title>sign in your name</title>
5.     </head>
6.     <body>
7.         <h2>Please sing in.</h2>
8.         <form method="post" action="/user">
9.             <p>Name:<br><input type="text" name="username"></p>
10.            <p>Email:<br><input type="text" name="email"></p>
11.            <p>Website:<br><input type="text" name="website"></p>
12.            <p>Language:<br><input type="text" name="language"></p>
13.            <input type="submit" value="ok,submit my information">
14.        </form>
15.    </body>
16. </html>

```

上面的代码是比较简单，如果看官熟悉html的话，不熟悉也不要紧，网上搜索就能理解。注意，没有CSS+JS，所以简单。如果在真正开发中，这两个是不能少的。

有了这个表单之后，如果用户把相关信息都填写好了。点击下面的按钮，就应该提交给后端的python程序来处理。

后端处理程序

做为tornado驱动的网站，首先要能够把前面的index.html显示出来，这个一般用get方法，显示的样式就按照上面的样子显示。

用户填写信息之后，点击按钮提交。注意观察上面的代码表单中，设定了 `post` 方法，所以，在python程序中，应该有一个post方法专门来接收所提交的数据，然后把提交的数据在另外一个网页显示出来。

在表单中还要注意，有一个 `action=/user`，表示的是要将表单的内容提交给 `/user` 路径所对应的程序来处理。这里需要说明的是，在网站中，数据提交和显示，路径是非常重要的。

按照以上意图，编写如下代码，并命名为`usercontroller.py`，保存在`template`目录中

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  import os.path
5.
6.  import tornado.httpserver
7.  import tornado.ioloop
8.  import tornado.options
9.  import tornado.web
10.
11.  from tornado.options import define, options
12.  define("port", default=8000, help="run on the given port",
13.        type=int)
14.
15.  class IndexHandler(tornado.web.RequestHandler):
16.      def get(self):
17.          self.render("index.html")
18.
19.  class UserHandler(tornado.web.RequestHandler):
20.      def post(self):
21.          user_name = self.get_argument("username")
22.          user_email = self.get_argument("email")
23.          user_website = self.get_argument("website")
24.          user_language = self.get_argument("language")
25.
26.          self.render("user.html", username=user_name, email=user_email, website=u
27.
28.  handlers = [
29.      (r"/", IndexHandler),
30.      (r"/user", UserHandler)
31.  ]
32.
33.  template_path = os.path.join(os.path.dirname(__file__), "template")
34.
35.  if __name__ == "__main__":

```

```

34.     tornado.options.parse_command_line()
35.     app = tornado.web.Application(handlers, template_path)
36.     http_server = tornado.httpserver.HTTPServer(app)
37.     http_server.listen(options.port)
38.     tornado.ioloop.IOLoop.instance().start()

```

这次代码量多一些。但是多数在前面讲述tornado基本结构的时候已经说过了，跟前面一样，这里仅仅把重点的和新出现的进行讲述，如果看官对某些内容还有疑问，可以参考前面的相关章节。

在引入的模块上，多了一个 `import os.path`，这个模块主要用在：

```
1. template_path = os.path.join(os.path.dirname(__file__), "template")
```

这是要获取存放程序的目录 `template` 的路径。

重点看两个类中都有的 `self.render()`，用这个方法引入相应的模板。

```
1. self.render("index.html")
```

显示index.html模板，但是此时并没有向模板网页传递任何数据，仅仅显示罢了。下面一个：

```
1. self.render("user.html", username=user_name, email=user_email, website=u
```

与前面的不同在于，不仅仅是要引用模板网页user.html，还要向这个网页传递一些数据，例如username=user_name，含义就是，在模板中，某个地方是用username来标示得到的数据，而user_name是此方法中的一个变量，也就是对应一个数据，那么模板中的username也就对应了user_name的数据，这是通过username=user_name完成的（说的有点像外语了）。后面的变量同理。

那么，`user_name`的数据是哪里来的呢？就是在`index.html`页面的表单中提交上来的。注意观察路径的设置，`r"/user", UserHandler`，也就是在form中的 `action='/user'`，就是要将数据提交给`UserHandler`处理，并且是通过post方法。所以，在`UserHandler`类中，有`post()`方法来处理这个问题。通过`self.get_argument()`来接收前端提交过来的数据，接收方法就是，`self.get_argument()`的参数与`index.html`表单form中的各项的name值相同，就会得到相应的数据。例如 `user_name = self.get_argument("username")`，就能够得到`index.html`表单中name为“username”的元素的值，并赋给`user_name`变量。

还差一个网页。

显示结果

在上面的代码中，又多了一个模板：`index.html`，对这个模板，跟前面那个模板有一点儿不一样的地方，就是要引入一些变量。它的代码是这样的：

```

1.  <!DOCTYPE html>
2.  <html>
3.      <head>
4.          <title>sign in your name</title>
5.      </head>
6.      <body>
7.          <h2>Your Information</h2>
8.          <p>Your name is {{username}}</p>
9.          <p>Your email is {{email}}</p>
10.         <p>Your website is {{website}}, it is very good. This
            website is make by {{language}}</p>
11.     </body>
12. </html>

```

请将上面的代码和这句话对照：

```
1. self.render("user.html", username=user_name, email=user_email, website=u
```

上面的模板代码存储为名为 `user.html` 的文件，并且和前面已经保存的在同一个目录中。

看HTML模板代码中，有类似 `{{username}}` 的变量，模板中用 `{{}}` 引入变量，这个变量就是在 `self.render()` 中规定的，两者变量名称一致，对应将相应的值对象引入到模板中。

运行结果

进入到template目录，执行：

```
1. qw@qw-Latitude-E4300:~/template$ python userscontroller.py
```

然后在浏览器的地址栏中输入

```
1. http://localhost:8000
```

出现如下图的表单，并填写表单内容

点击“按钮”之后：

这样就实现了一个简单表单提交的网站。

模板中的语法

- 模板中的语法
 - 模板中循环的例子
 - 模板中的判断语句
 - 模板中设置变量

“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 12:28-30)

模板中的语法

在[上一讲](#)的练习中，列位已经晓得，模板中 `{{placeholder}}` 可以接收来自python文件（.py）中通过 `self.render()` 传过来的参数值，这样模板中就显示相应的结果。在这里，可以将 `{{placeholder}}` 理解为占位符，就如同变量一样啦。

这是一种最基本的模板显示方式了。但如果仅仅如此，模板的功能有点单调，无法完成比较复杂的数据传递。不仅仅是tornado，其它框架如Django等，模板都有比较“高级”的功能。在tornado的模板中，功能还是不少的，本讲介绍模板语法先。

模板中循环的例子

在模板中，也能像在python中一样，可以使用某些语法，比如常用的if、for、while等语句，使用方法如下：

先看例子

先写一个python文件（命名为index.py），这个文件中有一个列

表 `["python", "www.itdiffer.com", "qiwsir@gmail.com"]`，要求是将这个列表通过 `self.render()` 传给模板。

然后在模板中，利用for语句，依次显示得到的列表中的元素。

```

1.  #!/usr/bin/env python
2.  #!/usr/bin/env python
3.
4.  import os.path
5.  import tornado.httpserver
6.  import tornado.ioloop
7.  import tornado.web
8.  import tornado.options
9.
10. from tornado.options import define, options
11. define("port", default=8000, help="run on the given port",  

   type=int)
12.
13. class IndexHandler(tornado.web.RequestHandler):
14.     def get(self):
15.         lst = ["python", "www.itdiffer.com", "qiwsir@gmail.com"]  #定义一个list
16.         self.render("index.html", info=lst)  #将  

   上述定义的list传给模板
17.
18. handlers = [(r"/", IndexHandler),]
19.
20. template_path = os.path.join(os.path.dirname(__file__), "temploop")  

   #模板路径
21.
22. if __name__ == "__main__":
23.     tornado.options.parse_command_line()
24.     app = tornado.web.Application(handlers, template_path)
25.     http_server = tornado.httpserver.HTTPServer(app)
26.     http_server.listen(options.port)
27.     tornado.ioloop.IOLoop.instance().start()

```

模板文件，名称是index.html，在目录templeop中。代码如下：

```

1. <DOCTYPE html>
2. <html>
3.     <head>
4.         <title>Loop in template</title>
5.     </head>
6.     <body>
7.         <p>There is a list, it is <b>{{info}}</b></p>
8.         <p>I will print the elements of this list in order.</p>
9.         {% for element in info %}      <!-- 循环开始，注意写法类似python中的
for，但是最后没有冒号 -->
10.             <p>{{element}}</p>        <!-- 显示element的内容 -->
11.         {% end %}                      <!-- 结束标志 -->
12.         <br>
13.         {% for index,element in enumerate(info) %}
14.             <p>info[{{index}}] is {{element}}
15.         {% end %}
16.     </body>
17. </html>

```

运行上面的程序：

```
1. >>> python index.py
```

然后在浏览器地址栏中输入：`http://localhost:8000`，显示的页面如下
图：



在上面的例子中，用如下样式，实现了模板中的for循环，这是在模板中常用到的，当然，从python程序中传过来的不一定是list类型数据，也可能是其它类型的序列数据。

```

1. {% for index,element in enumerate(info) %}
2.     <p>info[{{index}}] is {{element}}

```


3. `{% end %}`

特别提醒注意的是，语句要用 `{% end %}` 来结尾。在循环体中，用 `{{ element }}` 方式使用序列的元素。

模板中的判断语句

除了循环之外，在模板中也可以有判断，在上面代码的基础上，改写一下，直接上代码，看官想必也能理解了。

`index.py`的代码不变，只修改模板`index.html`的代码，重点理解模板中的语句写法。

```

1. <DOCTYPE html>
2. <html>
3.     <head>
4.         <title>Loop in template</title>
5.     </head>
6.     <body>
7.         <p>There is a list, it is <b>{{info}}</b></p>
8.         <p>I will print the elements of this list in order.</p>
9.         {% for element in info %}
10.            <p>{{element}}</p>
11.        {% end %}
12.        <br>
13.        {% for index,element in enumerate(info) %}
14.            <p>info[{{index}}] is {{element}}
15.            {% if element == "python" %}                                <!-- 增加了一个判断语
句 -->
16.                <p> <b>I love this language--{{element}}</b></p>
17.            {% end %}
18.        {% end %}
19.
20.        {% if "qiwsir@gmail.com" in info %}                            <!-- 还是判断一下 -->
21.            <p><b>A Ha, this the python lesson of LaoQi, It is good!
                His email is {{info[2]}}</b></p>

```

```

22.     {% end %}
23.     </body>
24. </html>

```

上面的模板运行结果是下图样子，看官对比一下，是否能够理解呢？



模板中设置变量

废话不说，直接上例子，因为例子是非常直观的：

```

1. <DOCTYPE html>
2. <html>
3.     <head>
4.         <title>Loop in template</title>
5.     </head>
6.     <body>
7.         <p>There is a list, it is <b>{{info}}</b></p>
8.         <p>I will print the elements of this list in order.</p>
9.         {% for element in info %}
10.            <p>{{element}}</p>
11.        {% end %}
12.        <br>
13.        {% for index,element in enumerate(info) %}
14.            <p>info[{{index}}] is {{element}}
15.            {% if element == "python" %}
16.                <p> <b>I love this language--{{element}}</b></p>
17.            {% end %}
18.        {% end %}
19.
20.        {% if "qiwsir@gmail.com" in info %}
21.            <p><b>A Ha, this the python lesson of LaoQi, It is good!
22.            His email is {{info[2]}}</b></p>
23.        {% end %}
24.        <h2>Next, I set "python-tornado"(a string) to a variable(var)
25.    </h2>

```

```
24.     {% set var="python-tornado" %}  
25.     <p>Would you like {{var}}?</p>  
26.     </body>  
27. </html>
```

显示结果如下：



看官发现了吗？我用 `{% set var="python-tornado" %}` 的方式，将一个字符串赋给了变量 `var`，在下面的代码中，就直接引用这个变量了。这样就是实现了模板中变量的使用。

Tornado的模板真的功能不少呢。不过远非这些，后面还有。敬请等待。

[首页](#) | [上一讲：使用表单和模板](#)

静态文件以及一个项目框架

- 静态文件以及一个项目框架
 - 静态路径
 - 编写模板文件
 - 一个项目框架

"just as the Son of Man came not to be served but to serve, and to give his life a ransom for many." (MATTHEW 28)

静态文件以及一个项目框架

在网上浏览网页，由于现在网速也快了，大概你很少注意网页中那些所谓的静态文件。怎么找出来静态文件呢？

如果使用firefox(我特别向列位推荐这个浏览器，它是我认为的最好的浏览器，没有之一。哈哈。“你信不信？反正我信了。”)，可以通过firebug组件，来研究网页的代码，当然，你直接看源码也行。



上图中，我打开了一个对天朝很多人来说不存在的网站，并且通过Firebug查看其源码，打开 `<head>`，发现里面有不少 `<script` 和 `<link` 开始引入的文件，这些文件一部分是javascript文件，一部分是css文件。在一个网站中，这类文件一般是不会发生变化的，也就是它的内容稳定，直到下次文件管理员或者有权限的人修改时。而网站程序本身一般不会修改它们。因此将他们称之为静态文件。

此外，网站中的静态文件还包括一些图片，比如logo，以及做为边框的图片等。

在tornado中，有专门方法处理这些静态文件。

静态路径

看官是否还记得在前面写过这个模样的代

码：`template_path=os.path.join(os.path.dirname(__file__), "templates")`，这里是设置了模板的路径，放置模板的目录名称是 `templates`。类似的方法，我们也可以设置好静态文件的路径。

```
1. static_path=os.path.join(os.path.dirname(__file__), "static")
```

这里的设置，就是将所有的静态文件，放在了 `static` 目录中。

这样就设置了静态路径。

下面的代码是将[上一节](#)的代码进行了改写，注意变化的地方

```
1.  #!/usr/bin/env python
2.  #-*- coding:utf-8 -*-
3.
4.  import os.path
5.  import tornado.httpserver
6.  import tornado.ioloop
7.  import tornado.web
8.  import tornado.options
9.
10. from tornado.options import define, options
11. define("port", default=8000, help="run on the given port",
12.       type=int)
13.
14. class IndexHandler(tornado.web.RequestHandler):
15.     def get(self):
16.         lst = ["python", "www.itdiffer.com", "qiwsir@gmail.com"]
17.         self.render("index.html", info=lst)
```

```

18. handlers = [(r"/", IndexHandler),]
19.
20. template_path = os.path.join(os.path.dirname(__file__), "temploop")
21. static_path = os.path.join(os.paht.dirname(__file__), "static")
    #这里增加设置了静态路径
22.
23. if __name__ == "__main__":
24.     tornado.options.parse_command_line()
25.     app = tornado.web.Application(handlers, template_path,
        static_path, debug=True)    #这里也多了点
26.     http_server = tornado.httpserver.HTTPServer(app)
27.     http_server.listen(options.port)
28.     tornado.ioloop.IOLoop.instance().start()

```

上面的代码比前一讲中，多了两处，一处是定义了静态文件路径 `static_path`，在这里将存储静态文件的目录命名为 `static`。另外一个修改就是在实例化 `tornado.web.Application()` 的时候，在参数中，出了有静态路径参数 `static_path`，还有一个参数设置 `debug=True`，这个参数设置的含义是当前的tornado服务器可以不用重启，就能够体现已经修改的代码功能。回想一下，在前面进行调试的时候，是不是每次如果修改了代码，就得重启tornado服务器，才能看到修改效果。用了这个参数就不用这么麻烦了。

特别说着，`debug=True` 仅仅用于开发的调试过程中，如果在生产部署的时候，就不要这么使用了。

编写模板文件

我们设置静态路径的目的就是要在模板中引入css和js等类型的文件以及图片等等。那么如何引入呢，下面以css为例说明。

在一般网页的 `<head>...</head>` 部分，都会引入CSS，例如下面的写法不少网站都愿意引用google的字体库，样式如下：

```
1. <link href='http://fonts.googleapis.com/css?
    family=Open+Sans:300,400,600&subset=latin,latin-ext'
    rel='stylesheet'>
```

这就是CSS的引入。

但是，如果看官在墙内也这么引入字体库，希望自己的网页上能使用，那就有点麻烦了，因为google的这个项目已经不行被墙，如果在网页中写了上面代码，会导致网页打开速度很慢，有的甚至出错。

怎么办？那就不用啦。不过，国内有好心网站做了整个谷歌字体的代理，可以用下面方式，墙里面就不怕了。

```
1. <link href='http://fonts.useso.com/css?
    family=Open+Sans:300,400,600&subset=latin,latin-ext'
    rel='stylesheet'>
```

顺便提供一个墙内的常用前端库地址：<http://libs.useso.com/>，供看官参考使用。

那么如果我自己写CSS呢？并且按照前面的设定，已经将该CSS文件放在了static目录里面，命名为style.css，就可以这样引入

```
1. <link href="/static/style.css" rel="stylesheet">
```

但是，这里往往会有一个不方便的地方，如果我手闲着无聊，或者因为别的充足理由，将存储静态文件的目录static换成了sta，并且假设我已经在好几个模板中都已经写了上面的代码。接下来我就要把这些文件全打开，一个一个更改 `<link>` 里面的地址。

请牢记，凡是遇到重复的事情，一定要找一个函数方法解决。tornado就提供了这么一个：`static_url()`，把上面的代码写成：

```
1. <link href="{{ static_url("style.css") }}" rel="stylesheet" >
```

这样，就不管你如何修改静态路径，模板中的设置可以不用变。

按照这个引入再修改相应的模板文件。

一个项目框架

以上以及此前，我们所有写过的，都是简单的技术方法演示，如果要真的写一个基于tornado框架的网站，一般是不用这样的直接把代码都写到一个文件index.py中的，一个重要原因，就是这样做不容易以后维护，也不便于多人协作写一个项目。

所以在真实的项目中，常常要将不同部件写在不同文件中。下面的例子就是一个项目的基本框架。当然，这还是一个比较小的项目，但是“麻雀虽小，五脏俱全”。

创建一个文件夹，我给它命名为project，在这个文件里面分别创建下面的文件和目录，文件和目录里面的内容可以先不用管，“把式把式，先看架势”，先搭起项目结构来。

- 文件 `application.py`：这个文件的核心任务是完成 `tornado.web.Application()` 的实例化
- 文件 `url.py`：在这个文件中记录项目中所有URL和映射的类，即完成前面代码中 `handlers=[...]` 的功能
- 文件 `server.py`：这是项目的入口文件，里面包含 `if __name__ == "__main__"`，从这里启动项目和服务
- 目录handler：存放 `.py` 文件，即所谓各种请求处理类（当然，如果更大一些的项目，可能还要分配给别的目录来存储这种东西）
- 目录optsql：存放操作数据库的文件，比如各种读取或者写入数据库的类或函数，都放在这里面的某些文件中

- 目录static: 存放静态文件, 就是上文说的比如CSS, JS, 图片等, 为了更清晰, 在这个目录里面, 还可建立子目录, 比如我就建立了: css, js, img三个子目录, 分别对应上面的三种。
- 目录template: 存放 `.html` 的模板 (在更大型的项目中, 可能会设计多个目录来存放不同的模板, 或者在里面再有子目录进行区分)

以上就是我规划的一个项目的基本框架了。不同开发者根据自己的习惯, 有不同的规划, 或者有不同的命名, 这没有关系。不过需要说明的, 尽量使用名词 (英文)。我看到过有人做过单复数之争论。我个人认为, 这个无所谓, 只要在一个项目中一贯就好了。我还是用单数吧, 因为总忘记那个复数后面的s

下面分别把不同部分文件的内容列出来, 因为都是前面代码的重复, 不过是做了一点从新部署, 所以, 就不解释了。个别地方有一点点说明。

文件application.py的代码如下:

```
1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  from url import url
5.
6.  import tornado.web
7.  import os
8.
9.  setting = dict(
10.
11.      template_path=os.path.join(os.path.dirname(__file__), "template"),
12.      static_path=os.path.join(os.path.dirname(__file__), "static"),
13.  )
14.  application = tornado.web.Application(
15.      handlers=url,
```

```
16.         **setting
17.     )
```

`from url import url` 是从文件`url.py`引入内容

下面看看`url.py`文件内容：

```
1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  import sys
5.  reload(sys)
6.  sys.setdefaultencoding('utf-8')
7.
8.  from handler.index import IndexHandler
9.
10. url=[
11.     (r'/', IndexHandler),
12.
13. ]
```

在这个文件中，从 `import sys` 开始的三行，主要是为了解决如果文件里面有汉字，避免出现乱码。现在这个文件很简单，里面只有 `(r'/', IndexHandler)` 一条URL，如果多条了，就要说明每条是什么用途，如果用中文写注释，需要避免乱码。

以上两个预备好了，就开始写`server.py`，内容如下：

```
1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  import tornado.ioloop
5.  import tornado.options
6.  import tornado.httpserver
7.
8.  import sys
```

```

9.
10. from application import application
11.
12. from tornado.options import define, options
13. define("port", default=8888, help="run on the given port", type=int)
14.
15. def main():
16.     tornado.options.parse_command_line()
17.     http_server = tornado.httpserver.HTTPServer(application)
18.     http_server.listen(options.port)
19.     print 'Development server is running at http://127.0.0.1:%s/' %
options.port
20.     print 'Quit the server with Control-C'
21.     tornado.ioloop.IOLoop.instance().start()
22.
23. if __name__ == "__main__":
24.     main()

```

这个就不需要解释了。接下来就看目录，首先在 `static/css/` 里面建立一个 `style.css` 的文件，并写样式表。我只写了下面的样式，这个样式的目的主要是去除浏览器默认的样式，在实际的工程项目中，这个步骤是非常必要的，一定要去除所有默认的样式，然后重新定义，才能便于控制。

```

1. html, body, div, span, applet, object, iframe, h1, h2, h3, h4, h5,
h6, p, blockquote, pre, a, abbr, acronym, address, big, cite,
code, del, dfn, em, img, ins, kbd, q, s, samp, small, strike, strong,
sub, sup, tt, var, b, u, i, center, dl, dt, dd, ol, ul, li, fieldset,
form, label, legend, table, caption, tbody, tfoot, thead, tr, th,
td, article, aside, canvas, details, embed, figure, figcaption,
footer, header, hgroup, menu, nav, output, ruby, section,
summary, time, mark, audio, video {
2.     margin: 0;
3.     padding: 0;
4.     border: 0;
5.     font-size: 100%;

```

```

6.     font: inherit;
7.     vertical-align: baseline;
8. }
9. /* HTML5 display-role reset for older browsers */
10. article, aside, details, figcaption, figure, footer, header,
    hgroup, menu, nav, section {
11.     display: block;
12. }
13.
14.
15. body {
16.     /* standard body */
17.     margin: 0 auto;
18.     width: 960px;
19.     font: 14px/20px "Trebuchet MS", Verdana, Helvetica, Arial,
        sans-serif;
20. }

```

为了能够在演示的时候看出样式控制的变化，多写了一个对body的控制，居中且宽度为960px。

样式表已经定义好，就要看 `template/index.html` 了，这个文件就是本项目中的唯一一个模板。

```

1. <DOCTYPE html>
2. <html>
3.     <head>
4.         <title>Loop in template</title>
5.         <link rel="stylesheet" type="text/css" href="{{
            static_url('css/style.css')}}">
6.     </head>
7.     <body>
8.         <h1>aaaAAA</h1>
9.         <p>There is a list, it is <b>{{info}}</b></p>
10.        <p>I will print the elements of this list in order.</p>
11.        {% for element in info %}
12.            <p>{{element}}</p>

```

```

13.     {% end %}
14.     <br>
15.     {% for index,element in enumerate(info) %}
16.         <p>info[{{index}}] is {{element}}
17.         {% if element == "python" %}
18.             <p> <b>I love this language--{{element}}</b></p>
19.         {% end %}
20.     {% end %}
21.
22.     {% if "qiwsir@gmail.com" in info %}
23.         <p><b>A Ha, this the python lesson of LaoQi, It is good!
His email is {{info[2]}}</b></p>
24.     {% end %}
25.     <h2>Next, I set "python-tornado"(a string) to a variable(var)
</h2>
26.     {% set var="python-tornado" %}
27.     <p>Would you like {{var}}?</p>
28.     </body>
29. </html>

```

在这个文件中，特别注意就是 `<link rel="stylesheet" type='text/css' href="{{ static_url('css/style.css')}}"`，这里引入了前面定义的样式表文件。引入方式就是前文讲述的方式，不过由于是在css这个子目录里面，所以写了相对路径。

行文到此，我原以为已经完成了。一检查，发现一个重要的目录 `handler` 里面还空着呢，那里面放 `index.py` 文件，这个文件里面是请求响应的类 `IndexHandler`

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  import tornado.web
5.
6.  import sys
7.  reload(sys)

```

```
8. sys.setdefaultencoding('utf-8')
9.
10. class IndexHandler(tornado.web.RequestHandler):
11.     def get(self):
12.         lst = ["python", "www.itdiffer.com", "qiwsir@gmail.com"]
13.         self.render("index.html", info=lst)
```

这个文件的代码没有什么增加的内容，只是多了三行设置为utf-8的配置，目的是避免汉字乱码。另外，很需要说明的是，由于这个文件在handler目录里面，要在上一层的url.py中引用（看url.py内容），必须要在本目录中建立一个名称是 `__init__.py` 的空文。

好了，基本结构已经完成。跑起来。效果就是这样的：



[首页](#) | [上一讲：模板中的语法](#)

模板转义

- 模板转义
 - 模板自动转义
 - 不转义的办法
 - url转义

“Pay attention to what you hear; the measure you give will be the measure you get, and still more will be given you. For to those who have, more will be given; and from those who have nothing, even what they have will be taken away.”(MARK 4:24-25)

模板转义

列位看官曾记否？在《玩转字符串（1）》中，有专门讲到了有关“转义”问题，就是在python的字符串中，有的符号要想表达其本意，需要在前面加上 `\` 符号，例如单引号，如果要在字符串中表现它，必须写成 `\'单引号里面\'` 样式，才能实现一对单引号以及里面的内容，否则，它就表示字符串了。

在HTML代码中，也有类似的问题，比如 `>` 等，就是代码的一部分，如果直接写，就不会显示在网页里，要向显示，同样需要转义。另外，如果在网页中有表单，总会有别有用心的人向表单中写点包含 `>` 等字符的东西，目的就是要攻击你的网站，为了防治邪恶之辈，也需要将用户输入的字符进行转义，转化为字符实体，让它不具有HTML代码的含义。

转义字符串 (Escape Sequence) 也称字符实体 (Character Entity)。在HTML中，定义转义字符串的原因有两个：第一个原因是像“<”和“>”这类符号已经用来表示HTML标签，因此就不能直接当作文本中的符号来使用。为了在HTML文档中使用这些符号，就需要定义它的转义字符串。当解释程序遇到这类字符串时就把它解释为真实的字符。在输入转义字符串时，要严格遵守字母大小写的规则。第二个原因是，有些字符在ASCII字符集中没有定义，因此需要使用转义字符串来表示。

模板自动转义

Tornado 2 开始的模板具有自动转义的功能，这让开发者省却了不少事情。看一个例子。就利用上一讲中建立的开发框架。要在首页模板中增加一个表单提交功能。

修改template/index.html文件，内容如下：

```

1. <DOCTYPE html>
2. <html>
3.     <head>
4.         <title>Loop in template</title>
5.         <link rel="stylesheet" type="text/css" href="{{
static_url('css/style.css')}}">
6.     </head>
7.     <body>
8.         <h1>aaaAAA</h1>
9.         <p>There is a list, it is <b>{{info}}</b></p>
10.        <p>I will print the elements of this list in order.</p>
11.        {% for element in info %}
12.            <p>{{element}}</p>
13.        {% end %}
14.        <br>
15.        {% for index,element in enumerate(info) %}
16.            <p>info[{{index}}] is {{element}}
17.            {% if element == "python" %}
18.                <p> <b>I love this language--{{element}}</b></p>
19.            {% end %}
20.        {% end %}
21.
22.        {% if "qiwsir@gmail.com" in info %}
23.            <p><b>A Ha, this the python lesson of LaoQi, It is good!
His email is {{info[2]}}</b></p>
24.        {% end %}
25.        <h2>Next, I set "python-tornado"(a string) to a variable(var)
</h2>
26.        {% set var="python-tornado" %}

```



```

27.     <p>Would you like {{var}}?</p>
28.     <!--增加表单-->
29.     <form method="post" action="/option">
30.         <p>WebSite:<input id="website" name="website" type="text">
31.     </p>
32.         <p><input type="submit" value="ok,submit"></p>
33.     </form>
34. </body>
35. </html>

```

在增加的表单中，要将内容以 `post` 方法提交到 `"/option"`，所以，要在 `url.py` 中设置路径，并且要建立相应的类。

然后就在 `handler` 目录中建立一个新的文件，命名为 `optform.py`，其内容就是一个类，用来接收 `index.html` 中 `post` 过来的表单内容。

```

1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  import tornado.web
5.
6.  import sys
7.  reload(sys)
8.  sys.setdefaultencoding('utf-8')
9.
10. class OptionForm(tornado.web.RequestHandler):
11.     def post(self):
12.         website = self.get_argument("website")      #接收名称
13.         #为'website'的表单内容
14.         self.render("info.html",web=website)

```

为了达到接收表单 `post` 到上述类中内容的目的，还需要对 `url.py` 进行如下改写：

```

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

```

```

3.
4. import sys
5. reload(sys)
6. sys.setdefaultencoding('utf-8')
7.
8. from handler.index import IndexHandler
9. from handler.optform import OptionForm
10. url=[
11.     (r'/', IndexHandler),
12.     (r'/option', OptionForm),
13.
14. ]

```

看官还要注意，我在新建立的optform.py中，当接收到来自表单内容之后，就用另外一个模板 `info.html` 显示所接收到的内容。这个文件放在template目录中，代码是：

```

<DOCTYPE html>
<html>
  <head>
    <title>Loop in template</title>
    <link rel="stylesheet" type="text/css" href="{{ static_u
  </head>
  <body>
    <h1>My Website is:</h1>
    <p>{{web}}</p>
  </body>
</html>

```

这样我们就完成表单内容的提交和显示过程。

从上面的流程中，看官是否体验到这个框架的优势了？不用重复敲代码，只需要在框架内的不同地方增加内容，即可完成网站。

演示运行效果：

我在表单中输入了 `<script>alert('bad script')</script>`，这是多么阴险毒辣呀。



然而我们的tornado是不惧怕这种攻击的，因为它的模板自动转义了。当点击按钮提交内容的时候，就将那些阴险的符号实体化，成为转义之后的符号了。于是就这样了：



输入什么，就显示什么，不会因为输入的内容含有阴险毒辣的符号而网站无法正常工作。这就是转义的功劳。

不转义的办法

在tornado中，模板实现了自动转义，省却了开发者很多事，但是，事情往往没有十全十美的，这里省事了，一定要在别的地方费事。例如在上面那个info.html文件中，我打算在里面加入我的电子信箱，但是要像下面代码这样，设置一个变量，主要是为了以后修改方便和在其它地方也可以随意使用。

```
<DOCTYPE html>
<html>
  ... (省略)
  <body>
    <h1>My Website is:</h1>
    <p>{{web}}</p>
    {% set email="<a href='mailto:qiwsir@gmail.com'>Connect to m
    <p>{{email}}</p>
  </body>
</html>
```

本来希望在页面中出现的是 `Connect to me`，点击它之后，就直接连接到发送电子邮件。结果，由于转义，出现的是下面的显示结果：



实现电子邮件超链接未遂。

这时候，就需要不让模板转义。tornado提供的方法是：

- 在Application函数实例化的时候，设置参数：
`autoescape=None`。这种方法不推荐适应，因为这样就让全站模板都不转意了，看官愿意尝试，不妨进行修改试一试，我这里就不展示了。
- 在每个页面中设置`{% autoescape None %}`，表示这个页面不转义。也不推荐。理由，自己琢磨。
- 以上都不推荐，我推荐的是：`{% raw email %}`，想让哪里不转义，就在那里用这种方式，比如要在email超级链接那里不转移，就写成这样好了。于是修改上面的代码，看结果为：



如此，实现了不转义。

以上都实现了模板的转义和不转义。

url转义

本来模板转义和不转义问题已经交代清楚了。怎奈上周六一个朋友问了一个问题，那个问题涉及到url转义问题，于是在这里再补上一段，专门谈谈url转义的问题。

有些符号在URL中是不能直接传递的，如果要在URL中传递这些特殊符号，那么就要使用它们的编码了。编码的格式为：`%`加字符的ASCII

码，即一个百分号%，后面跟对应字符的ASCII（16进制）码值。例如空格的编码值是"%20"。

在python中，如果用utf-8写了一段地址，如何转义成url能够接收的字符呢？

在python中有一个urllib模块：

```
>>> import urllib

>>> #假设下面的url, 是utf-8编码
>>> url_mail='http://www.itdiffer.com/email?=qiwsir@gmail.com'

>>> #转义为url能够接受的
>>> urllib.quote(url_mail)
'http%3A//www.itdiffer.com/email%3F%3Dqiwsir%40gmail.com'
```

反过来，一个url也能转移为utf-8编码格式，请用urllib.unquote()

下面抄录帮助文档中的内容，供用到的朋友参考：

```
quote(s, safe='/')
quote('abc def') -> 'abc%20def'
```

Each part of a URL, e.g. the path info, the query, etc., has different set of reserved characters that must be quoted.

RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax the following reserved characters.

```
reserved    = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+"
              "$" | ","
```

Each of these characters is reserved in some component of a but not necessarily in all of them.

By default, the quote function is intended for quoting the path section of a URL. Thus, it will not encode '/'. This character is reserved, but in typical usage the quote function is being called on a path where the existing slash characters are used for reserved characters.

```
unquote(s)
```

```
unquote('abc%20def') -> 'abc def'.
```

```
quote_plus(s, safe='')
```

```
Quote the query fragment of a URL; replacing ' ' with '+'
```

```
unquote_plus(s)
```

```
unquote('%7e/abc+def') -> '~/abc def'
```

转义是网站开发中要特别注意的地方，不小心或者忘记了，就会纠结。

[首页](#) | [上一讲：静态文件以及一个项目框架](#)

第四部分 暮然回首，灯火阑珊处

链接

- [requests库](#)
- [比较json/dictionary的库](#)
- [defaultdict 模块和 namedtuple 模块](#)

requests库

- [requests库](#)
 - [安装](#)
- [get请求](#)
- [post请求](#)
 - [http头部](#)
 - [老齐备注](#)

requests库

作者：**1world0x00**（说明：在入选本教程的时候，我进行了适当从新编辑）

requests是一个用于在程序中进行http协议下的get和post请求的库。

安装

```
1. easy_install requests
```

或者用

```
1. pip install requests
```

安装好之后，在交互模式下运行：

```
1. >>> import requests
2. >>> dir(requests)
3. ['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

```
1. >>> r = requests.get("http://www.itdiffer.com")
```

得到一个请求的实例，然后：

```
1. >>> r.cookies
2. <<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：

```
1. >>> r = requests.get("http://www.1world0x00.com")
2. >>> r.cookies
3. <<class 'requests.cookies.RequestsCookieJar'>[Cookie(version=0,
name='PHPSESSID', value='buqj70k7f9rrg51emsvatveda2', 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)]>
```

原来这样呀。继续，还有别的属性可以看看。

```
1. >>> r.headers
2. {'x-powered-by': 'PHP/5.3.3', 'transfer-encoding': 'chunked', 'set-
```

```

1. cookie': 'PHPSESSID=buqj70k7f9rrg51emsvatveda2; path=/', 'expires':
2. 'Thu, 19 Nov 1981 08:52:00 GMT', 'keep-alive': 'timeout=15,
3. max=500', 'server': 'Apache/2.2.15 (CentOS)', 'connection': 'Keep-
4. Alive', 'pragma': 'no-cache', 'cache-control': 'no-store, no-cache,
5. must-revalidate, post-check=0, pre-check=0', 'date': 'Mon, 10 Nov
6. 2014 01:39:03 GMT', 'content-type': 'text/html; charset=UTF-8', 'x-
7. pingback': 'http://www.1world0x00.com/index.php/action/xmlrpc'}
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
27.
28.
29.
30.
31.
32.
33.
34.
35.
36.
37.
38.
39.
40.
41.
42.
43.
44.
45.
46.
47.
48.
49.
50.
51.
52.
53.
54.
55.
56.
57.
58.
59.
60.
61.
62.
63.
64.
65.
66.
67.
68.
69.
70.
71.
72.
73.
74.
75.
76.
77.
78.
79.
80.
81.
82.
83.
84.
85.
86.
87.
88.
89.
90.
91.
92.
93.
94.
95.
96.
97.
98.
99.
100.
101.
102.
103.
104.
105.
106.
107.
108.
109.
110.
111.
112.
113.
114.
115.
116.
117.
118.
119.
120.
121.
122.
123.
124.
125.
126.
127.
128.
129.
130.
131.
132.
133.
134.
135.
136.
137.
138.
139.
140.
141.
142.
143.
144.
145.
146.
147.
148.
149.
150.
151.
152.
153.
154.
155.
156.
157.
158.
159.
160.
161.
162.
163.
164.
165.
166.
167.
168.
169.
170.
171.
172.
173.
174.
175.
176.
177.
178.
179.
180.
181.
182.
183.
184.
185.
186.
187.
188.
189.
190.
191.
192.
193.
194.
195.
196.
197.
198.
199.
200.
201.
202.
203.
204.
205.
206.
207.
208.
209.
210.
211.
212.
213.
214.
215.
216.
217.
218.
219.
220.
221.
222.
223.
224.
225.
226.
227.
228.
229.
230.
231.
232.
233.
234.
235.
236.
237.
238.
239.
240.
241.
242.
243.
244.
245.
246.
247.
248.
249.
250.
251.
252.
253.
254.
255.
256.
257.
258.
259.
260.
261.
262.
263.
264.
265.
266.
267.
268.
269.
270.
271.
272.
273.
274.
275.
276.
277.
278.
279.
280.
281.
282.
283.
284.
285.
286.
287.
288.
289.
290.
291.
292.
293.
294.
295.
296.
297.
298.
299.
300.
301.
302.
303.
304.
305.
306.
307.
308.
309.
310.
311.
312.
313.
314.
315.
316.
317.
318.
319.
320.
321.
322.
323.
324.
325.
326.
327.
328.
329.
330.
331.
332.
333.
334.
335.
336.
337.
338.
339.
340.
341.
342.
343.
344.
345.
346.
347.
348.
349.
350.
351.
352.
353.
354.
355.
356.
357.
358.
359.
360.
361.
362.
363.
364.
365.
366.
367.
368.
369.
370.
371.
372.
373.
374.
375.
376.
377.
378.
379.
380.
381.
382.
383.
384.
385.
386.
387.
388.
389.
390.
391.
392.
393.
394.
395.
396.
397.
398.
399.
400.
401.
402.
403.
404.
405.
406.
407.
408.
409.
410.
411.
412.
413.
414.
415.
416.
417.
418.
419.
420.
421.
422.
423.
424.
425.
426.
427.
428.
429.
430.
431.
432.
433.
434.
435.
436.
437.
438.
439.
440.
441.
442.
443.
444.
445.
446.
447.
448.
449.
450.
451.
452.
453.
454.
455.
456.
457.
458.
459.
460.
461.
462.
463.
464.
465.
466.
467.
468.
469.
470.
471.
472.
473.
474.
475.
476.
477.
478.
479.
480.
481.
482.
483.
484.
485.
486.
487.
488.
489.
490.
491.
492.
493.
494.
495.
496.
497.
498.
499.
500.
501.
502.
503.
504.
505.
506.
507.
508.
509.
510.
511.
512.
513.
514.
515.
516.
517.
518.
519.
520.
521.
522.
523.
524.
525.
526.
527.
528.
529.
530.
531.
532.
533.
534.
535.
536.
537.
538.
539.
540.
541.
542.
543.
544.
545.
546.
547.
548.
549.
550.
551.
552.
553.
554.
555.
556.
557.
558.
559.
560.
561.
562.
563.
564.
565.
566.
567.
568.
569.
570.
571.
572.
573.
574.
575.
576.
577.
578.
579.
580.
581.
582.
583.
584.
585.
586.
587.
588.
589.
590.
591.
592.
593.
594.
595.
596.
597.
598.
599.
600.
601.
602.
603.
604.
605.
606.
607.
608.
609.
610.
611.
612.
613.
614.
615.
616.
617.
618.
619.
620.
621.
622.
623.
624.
625.
626.
627.
628.
629.
630.
631.
632.
633.
634.
635.
636.
637.
638.
639.
640.
641.
642.
643.
644.
645.
646.
647.
648.
649.
650.
651.
652.
653.
654.
655.
656.
657.
658.
659.
660.
661.
662.
663.
664.
665.
666.
667.
668.
669.
670.
671.
672.
673.
674.
675.
676.
677.
678.
679.
680.
681.
682.
683.
684.
685.
686.
687.
688.
689.
690.
691.
692.
693.
694.
695.
696.
697.
698.
699.
700.
701.
702.
703.
704.
705.
706.
707.
708.
709.
710.
711.
712.
713.
714.
715.
716.
717.
718.
719.
720.
721.
722.
723.
724.
725.
726.
727.
728.
729.
730.
731.
732.
733.
734.
735.
736.
737.
738.
739.
740.
741.
742.
743.
744.
745.
746.
747.
748.
749.
750.
751.
752.
753.
754.
755.
756.
757.
758.
759.
760.
761.
762.
763.
764.
765.
766.
767.
768.
769.
770.
771.
772.
773.
774.
775.
776.
777.
778.
779.
780.
781.
782.
783.
784.
785.
786.
787.
788.
789.
790.
791.
792.
793.
794.
795.
796.
797.
798.
799.
800.
801.
802.
803.
804.
805.
806.
807.
808.
809.
810.
811.
812.
813.
814.
815.
816.
817.
818.
819.
820.
821.
822.
823.
824.
825.
826.
827.
828.
829.
830.
831.
832.
833.
834.
835.
836.
837.
838.
839.
840.
841.
842.
843.
844.
845.
846.
847.
848.
849.
850.
851.
852.
853.
854.
855.
856.
857.
858.
859.
860.
861.
862.
863.
864.
865.
866.
867.
868.
869.
870.
871.
872.
873.
874.
875.
876.
877.
878.
879.
880.
881.
882.
883.
884.
885.
886.
887.
888.
889.
890.
891.
892.
893.
894.
895.
896.
897.
898.
899.
900.
901.
902.
903.
904.
905.
906.
907.
908.
909.
910.
911.
912.
913.
914.
915.
916.
917.
918.
919.
920.
921.
922.
923.
924.
925.
926.
927.
928.
929.
930.
931.
932.
933.
934.
935.
936.
937.
938.
939.
940.
941.
942.
943.
944.
945.
946.
947.
948.
949.
950.
951.
952.
953.
954.
955.
956.
957.
958.
959.
960.
961.
962.
963.
964.
965.
966.
967.
968.
969.
970.
971.
972.
973.
974.
975.
976.
977.
978.
979.
980.
981.
982.
983.
984.
985.
986.
987.
988.
989.
990.
991.
992.
993.
994.
995.
996.
997.
998.
999.
1000.

```

下面这个比较长，是网页的内容，仅仅截取显示部分：

```

1. >>> print r.text
2.
3. <!DOCTYPE html>
4. <html lang="zh-CN">
5.   <head>
6.     <meta charset="utf-8">
7.     <meta name="viewport" content="width=device-width, initial-
8.       scale=1.0">
9.     <title>1world0x00sec</title>
10.    <link rel="stylesheet"
11.      href="http://www.1world0x00.com/usr/themes/default/style.min.css">
12.    <link rel="canonical" href="http://www.1world0x00.com/" />
13.    <link rel="stylesheet" type="text/css"
14.      href="http://www.1world0x00.com/usr/plugins/CodeBox/css/codebox.css"
15.      />
16.    <meta name="description" content="爱生活，爱拉芳。不装逼还能做朋友。"
17.      />
18.    <meta name="keywords" content="php" />
19.    <link rel="pingback"
20.      href="http://www.1world0x00.com/index.php/action/xmlrpc" />
21.
22.
23.
24.
25.
26.
27.
28.
29.
30.
31.
32.
33.
34.
35.
36.
37.
38.
39.
40.
41.
42.
43.
44.
45.
46.
47.
48.
49.
50.
51.
52.
53.
54.
55.
56.
57.
58.
59.
60.
61.
62.
63.
64.
65.
66.
67.
68.
69.
70.
71.
72.
73.
74.
75.
76.
77.
78.
79.
80.
81.
82.
83.
84.
85.
86.
87.
88.
89.
90.
91.
92.
93.
94.
95.
96.
97.
98.
99.
100.
101.
102.
103.
104.
105.
106.
107.
108.
109.
110.
111.
112.
113.
114.
115.
116.
117.
118.
119.
120.
121.
122.
123.
124.
125.
126.
127.
128.
129.
130.
131.
132.
133.
134.
135.
136.
137.
138.
139.
140.
141.
142.
143.
144.
145.
146.
147.
148.
149.
150.
151.
152.
153.
154.
155.
156.
157.
158.
159.
160.
161.
162.
163.
164.
165.
166.
167.
168.
169.
170.
171.
172.
173.
174.
175.
176.
177.
178.
179.
180.
181.
182.
183.
184.
185.
186.
187.
188.
189.
190.
191.
192.
193.
194.
195.
196.
197.
198.
199.
200.
201.
202.
203.
204.
205.
206.
207.
208.
209.
210.
211.
212.
213.
214.
215.
216.
217.
218.
219.
220.
221.
222.
223.
224.
225.
226.
227.
228.
229.
230.
231.
232.
233.
234.
235.
236.
237.
238.
239.
240.
241.
242.
243.
244.
245.
246.
247.
248.
249.
250.
251.
252.
253.
254.
255.
256.
257.
258.
259.
260.
261.
262.
263.
264.
265.
266.
267.
268.
269.
270.
271.
272.
273.
274.
275.
276.
277.
278.
279.
280.
281.
282.
283.
284.
285.
286.
287.
288.
289.
290.
291.
292.
293.
294.
295.
296.
297.
298.
299.
300.
301.
302.
303.
304.
305.
306.
307.
308.
309.
310.
311.
312.
313.
314.
315.
316.
317.
318.
319.
320.
321.
322.
323.
324.
325.
326.
327.
328.
329.
330.
331.
332.
333.
334.
335.
336.
337.
338.
339.
340.
341.
342.
343.
344.
345.
346.
347.
348.
349.
350.
351.
352.
353.
354.
355.
356.
357.
358.
359.
360.
361.
362.
363.
364.
365.
366.
367.
368.
369.
370.
371.
372.
373.
374.
375.
376.
377.
378.
379.
380.
381.
382.
383.
384.
385.
386.
387.
388.
389.
390.
391.
392.
393.
394.
395.
396.
397.
398.
399.
400.
401.
402.
403.
404.
405.
406.
407.
408.
409.
410.
411.
412.
413.
414.
415.
416.
417.
418.
419.
420.
421.
422.
423.
424.
425.
426.
427.
428.
429.
430.
431.
432.
433.
434.
435.
436.
437.
438.
439.
440.
441.
442.
443.
444.
445.
446.
447.
448.
449.
450.
451.
452.
453.
454.
455.
456.
457.
458.
459.
460.
461.
462.
463.
464.
465.
466.
467.
468.
469.
470.
471.
472.
473.
474.
475.
476.
477.
478.
479.
480.
481.
482.
483.
484.
485.
486.
487.
488.
489.
490.
491.
492.
493.
494.
495.
496.
497.
498.
499.
500.
501.
502.
503.
504.
505.
506.
507.
508.
509.
510.
511.
512.
513.
514.
515.
516.
517.
518.
519.
520.
521.
522.
523.
524.
525.
526.
527.
528.
529.
530.
531.
532.
533.
534.
535.
536.
537.
538.
539.
540.
541.
542.
543.
544.
545.
546.
547.
548.
549.
550.
551.
552.
553.
554.
555.
556.
557.
558.
559.
560.
561.
562.
563.
564.
565.
566.
567.
568.
569.
570.
571.
572.
573.
574.
575.
576.
577.
578.
579.
580.
581.
582.
583.
584.
585.
586.
587.
588.
589.
590.
591.
592.
593.
594.
595.
596.
597.
598.
599.
600.
601.
602.
603.
604.
605.
606.
607.
608.
609.
610.
611.
612.
613.
614.
615.
616.
617.
618.
619.
620.
621.
622.
623.
624.
625.
626.
627.
628.
629.
630.
631.
632.
633.
634.
635.
636.
637.
638.
639.
640.
641.
642.
643.
644.
645.
646.
647.
648.
649.
650.
651.
652.
653.
654.
655.
656.
657.
658.
659.
660.
661.
662.
663.
664.
665.
666.
667.
668.
669.
670.
671.
672.
673.
674.
675.
676.
677.
678.
679.
680.
681.
682.
683.
684.
685.
686.
687.
688.
689.
690.
691.
692.
693.
694.
695.
696.
697.
698.
699.
700.
701.
702.
703.
704.
705.
706.
707.
708.
709.
710.
711.
712.
713.
714.
715.
716.
717.
718.
719.
720.
721.
722.
723.
724.
725.
726.
727.
728.
729.
730.
731.
732.
733.
734.
735.
736.
737.
738.
739.
740.
741.
742.
743.
744.
745.
746.
747.
748.
749.
750.
751.
752.
753.
754.
755.
756.
757.
758.
759.
760.
761.
762.
763.
764.
765.
766.
767.
768.
769.
770.
771.
772.
773.
774.
775.
776.
777.
778.
779.
780.
781.
782.
783.
784.
785.
786.
787.
788.
789.
790.
791.
792.
793.
794.
795.
796.
797.
798.
799.
800.
801.
802.
803.
804.
805.
806.
807.
808.
809.
810.
811.
812.
813.
814.
815.
816.
817.
818.
819.
820.
821.
822.
823.
824.
825.
826.
827.
828.
829.
830.
831.
832.
833.
834.
835.
836.
837.
838.
839.
840.
841.
842.
843.
844.
845.
846.
847.
848.
849.
850.
851.
852.
853.
854.
855.
856.
857.
858.
859.
860.
861.
862.
863.
864.
865.
866.
867.
868.
869.
870.
871.
872.
873.
874.
875.
876.
877.
878.
879.
880.
881.
882.
883.
884.
885.
886.
887.
888.
889.
890.
891.
892.
893.
894.
895.
896.
897.
898.
899.
900.
901.
902.
903.
904.
905.
906.
907.
908.
909.
910.
911.
912.
913.
914.
915.
916.
917.
918.
919.
920.
921.
922.
923.
924.
925.
926.
927.
928.
929.
930.
931.
932.
933.
934.
935.
936.
937.
938.
939.
940.
941.
942.
943.
944.
945.
946.
947.
948.
949.
950.
951.
952.
953.
954.
955.
956.
957.
958.
959.
960.
961.
962.
963.
964.
965.
966.
967.
968.
969.
970.
971.
972.
973.
974.
975.
976.
977.
978.
979.
980.
981.
982.
983.
984.
985.
986.
987.
988.
989.
990.
991.
992.
993.
994.
995.
996.
997.
998.
999.
1000.

```

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。

```
1. >>> r.content
2. '\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 .....
3.
4. 以二进制的方式打开服务器并返回数据。
```

post请求

requests发送post请求，通常你会想要发送一些编码为表单的数据——非常像一个html表单。要实现这个，只需要简单地传递一个字典给`data`参数。你的数据字典在发出请求时会自动编码为表单形式。

```
1. >>> import requests
2. >>> payload = {"key1": "value1", "key2": "value2"}
3. >>> r = requests.post("http://httpbin.org/post")
4. >>> r1 = requests.post("http://httpbin.org/post", data=payload)
```

r没有加data的请求，看看效果：



r1是加了data的请求，看效果：



多了form项。喵。

http头部

```
1. >>> r.headers['content-type']  
2. '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：

```
1. >>> r.headers['content-type'] = 'adad'  
2. >>> r.headers['content-type']  
3. 'adad'
```

注意，当定制头部的时候，如果需要定制的项目有很多，需要用到数据类型为字典。

老齐备注

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

比较json/dictionary的库

- 比较json/dictionary的库
 - 安装
 - 比较json
 - 将字符串组装成json格式

He called the crowd with his disciples, and said to them, "If any want to become my followers, let them deny themselves and take up their cross and follow me. For those who want to save their life will lose it, and those who lose their life for my sake, and for the sake of the gospel, will save it. For what will it profit them to gain the whole world and forfeit their life? Indeed, what can they give in return for their life? Those who are ashamed of me and of my words in this adulterous and sinful generation, of them the Son of Man will also be ashamed when he comes in the glory of his Father with the holy angels." (MARK 9:34-38)

比较json/dictionary的库

在某些情况下，比较两个json/dictionary，或许这样就可以实现：

```
1. >>> a
2. {'a': 1, 'b': 2}
3. >>> b
4. {'a': 2, 'c': 2}
5. >>> cmp(a, b)          #-1或者1, 代表两个dict不一样
6. -1
7. >>> c=a.copy()
8. >>> c
9. {'a': 1, 'b': 2}
10. >>> cmp(a, c)         #两者相同
11. 0
```

但是，这只能比较两个是不是一样，不能深入各处哪里不一样的比较结果。

有这样一个库，就能解决这个问题，它就是**json_tools**

安装

方法1:

```
1. >>> pip install json_tools
```

或者

```
1. >>> easy_install json_tools
```

方法2:到这里下载源码:

https://pypi.python.org/pypi/json_tools, 然后进行安装

比较json

首先看看都有哪些属性或者方法，用万能的实验室来看:

```
1. >>> import json_tools
2. >>> dir(json_tools)
```

```
['builtins', 'doc', 'file', 'loader', 'name',
'package', 'path', '_patch_main', '_printer_main',
'diff', 'patch', 'path', 'print_function',
'print_json', 'print_style', 'printer']
```

从上面的结果中，可以看到 `json_tools` 的各种属性和方法。

我在一个项目中使用了diff，下面演示一下使用过程

```
1. >>> a
2. {'a': 1, 'b': 2}
```

```

3. >>> b
4. {'a': 2, 'c': 2}
5. >>> json_tools.diff(a,b)
6. [{'prev': 1, 'value': 2, 'replace': '/a'}, {'prev': 2, 'remove':
    '/b'}, {'add': '/c', 'value': 2}]

```

上面这个比较是比较简单的，显示的是b相对于a的变化，特别注意，如果是b相对a，就要这样写：`json_tools.diff(a,b)`，如果是`json_tools.diff(b,a)`，会跟上面有所不同，请看结果：

```

1. >>> json_tools.diff(b,a)
2. [{'prev': 2, 'value': 1, 'replace': '/a'}, {'prev': 2, 'remove':
    '/c'}, {'add': '/b', 'value': 2}]

```

以 `json_tools(a,b)`，即b相对a发生的变化为例进行说明。

- b和a都有键 `'a'`，但是b相对a，键 `'a'` 的值发生了变化，由原来的 `1`，变为了 `2`。所以在比较结果的list中，有一个元素反应了这个结果 `{'prev': 1, 'value': 2, 'replace': '/a'}`，其中，`replace`表示发生变化的键，`value`表示变化后即当前该键的值，`prev`表示该键此前的值。
- b中的 `'c'` 相对与a，是新增的键。于是比较结果中这样反应出来：`{'add': '/c', 'value': 2}`
- b相对于a没有 `'b'` 这个键，也就是在b中将其删除了，于是比较结果中这样来显示：`{'prev': 2, 'remove': '/c'}`

通过上述结果，就显示出来的详细的比较结果，不仅如此，还能对多层嵌套的json进行比较。例如：

```

1. >>> a={"a":{"aa":{"aaa":333,"aaa2":3332},"b":22}}
2. >>> b={"a":{"aa":{"aaa":334,"bbb":339},"b":22}}
3. >>> json_tools.diff(a,b)
4. [{'prev': 3332, 'remove': '/a/aa/aaa2'}, {'prev': 333, 'value':

```

```
334, 'replace': '/a/aa/aaa'}, {'add': '/a/aa/bbb', 'value': 339}]
```

这里就显明了发生变化的key的嵌套关系。比如 `'/a/aa/aaa2'` , 就表示 `{"a":{"aa":{"aaa2":...}}}` 的值发生了变化。

这里有了一个key的嵌套字符串, 在真实的使用中, 有时候需要将字符串转为json的格式, 即 `{'prev': 3332, 'remove': '/a/aa/aaa2'}` 转化为 `{"a":{"aa":{"aaa2":3332}}}` 。

将字符串组装成json格式

首先, 回答前面的问题, 可以自己写一个函数, 实现那种组装。

但是, 我是懒惰地程序员, 我更喜欢python的原因就是它允许我懒惰。

```
1. from itertools import izip
```

具体这个模块如何使用, 请看官到我做过的小项目中看看: <https://github.com/qiwsir/json-diff>

defaultdict 模块和 namedtuple 模块

- defaultdict 模块和 namedtuple 模块
 - 一、defaultdict
 - 1. 简介
 - 2. 示例
 - 3. 原理
 - 4. 版本
 - 二、namedtuple

defaultdict 模块和 namedtuple 模块

author: Wuxiaolong

在Python中有一些内置的数据类型，比如int, str, list, tuple, dict等。Python的collections模块在这些内置数据类型的基础上，提供了几个额外的数据类型：namedtuple, defaultdict, deque, Counter, OrderedDict等，其中defaultdict和namedtuple是两个很实用的扩展类型。defaultdict继承自dict, namedtuple继承自tuple。

一、defaultdict

1. 简介

在使用Python原生的数据结构dict的时候，如果用d[key]这样的方式访问，当指定的key不存在时，是会抛出KeyError异常的。但是，如果使用defaultdict，只要你传入一个默认的工厂方法，那么请求一个不存在的key时，便会调用这个工厂方法使用其结果来作为这个key的默认值。

defaultdict在使用的時候需要傳一個工廠函數 (function_factory), defaultdict(function_factory)會構建一個類似dict的對象, 該對象具有默認值, 默認值通過調用工廠函數生成。

2. 示例

下面給一個defaultdict的使用示例:

```

1. >>> from collections import defaultdict
2. >>> s = [('xiaoming', 99), ('wu', 69), ('zhangsan', 80), ('lisi',
    96), ('wu', 100), ('wu', 100), ('yuan', 98), ('xiaoming', 89)]
3. >>> d = defaultdict(list)
4. >>> for k, v in s:
5. ...     d[k].append(v)
6. ...
7. >>> d
8. defaultdict(<type 'list'>, {'lisi': [96], 'xiaoming': [99, 89],
    'yuan': [98], 'zhangsan': [80], 'wu': [69, 100, 100]})
9. >>> for k,v in d.items():
10. ...     print '%s: %s' % (k, v)
11. ...
12. lisi: [96]
13. xiaoming: [99, 89]
14. yuan: [98]
15. zhangsan: [80]
16. wu: [69, 100, 100]
17. >>>

```

對Python比較熟悉的同学可以發現defaultdict(list)的用法和dict.setdefault(key, [])比較類似, 上述代碼使用setdefault實現如下:

```

1. >>> s
2. [('xiaoming', 99), ('wu', 69), ('zhangsan', 80), ('lisi', 96),

```

```

    ('wu', 100), ('wu', 100), ('yuan', 98), ('xiaoming', 89)]
3. >>> d = {}
4. >>> for k,v in s:
5. ...     d.setdefault(k, []).append(v)
6. ...
7. >>> d
8. {'lisi': [96], 'xiaoming': [99, 89], 'yuan': [98], 'zhangsan':
    [80], 'wu': [69, 100, 100]}

```

3. 原理

从以上的例子中，我们可以基本了defaultdict的用法，下面我们可以通过`help(defaultdict)`了解一下defaultdict的原理。通过Python console打印出的help信息来看，我们可以发现defaultdict具有默认值主要是通过missing方法实现的，如果工厂函数不为None，则通过工厂方法返回默认值，具体如下：

```

1. | __missing__(...)
2. |     __missing__(key) # Called by __getitem__ for missing key;
   | pseudo-code:
3. |         if self.default_factory is None: raise KeyError((key,))
4. |         self[key] = value = self.default_factory()
5. |         return value

```

从上面的说明中，我们可以发现一下几个需要注意的地方：

1. missing方法是在调用getitem方法发现KEY不存在时才调用的，所以，defaultdict也只会在使用`d[key]`或者`d.getitem(key)`的时候才会生成默认值；如果使用`d.get(key)`是不会返回默认值的，会出现KeyError；
2. defaultdict主要是通过missing方法实现，所以，我们也可以通过实现该方法来生成自己的defaultdict，代码入下

4. 版本

defaultdict是在Python 2.5之后才加入的功能，在旧版本的Python中是不支持这个功能的，不过，知道了它的原理，我们可以自己实现一个defaultdict。

```

1.  try:
2.      from collections import defaultdict
3.  except:
4.      class defaultdict(dict):
5.
6.          def __init__(self, default_factory=None, *a, **kw):
7.              if (default_factory is not None and not
hasattr(default_factory, '__call__')):
8.                  raise TypeError('first argument must be callable')
9.              dict.__init__(self, *a, **kw)
10.             self.default_factory = default_factory
11.
12.
13.         def __getitem__(self, key):
14.             try:
15.                 return dict.__getitem__(self, key)
16.             except KeyError:
17.                 return self.__missing__(key)
18.
19.         def __missing__(self, key):
20.             if self.default_factory is None:
21.                 raise KeyError(key)
22.             self[key] = value = self.default_factory()
23.             return value
24.
25.         def __reduce__(self):
26.             if self.default_factory is None:
27.                 args = tuple()
28.             else:
29.                 args = self.default_factory,
30.             return type(self), args, None, None, self.items()

```

```

31.
32.         def copy(self):
33.             return self.__copy__()
34.
35.         def __copy__(self):
36.             return type(self)(self.default_factory, self)
37.
38.         def __deepcopy__(self, memo):
39.             import copy
40.             return type(self)(self.default_factory,
copy.deepcopy(self.items()))
41.
42.         def __repr__(self):
43.             return 'defaultdict(%s, %s)' % (self.default_factory,
dict.__repr__(self))

```

二、namedtuple

namedtuple主要用来产生可以使用名称来访问元素的数据对象，通常用来增强代码的可读性，在访问一些tuple类型的数据时尤其好用。其实，在大部分时候你应该使用namedtuple替代tuple，这样可以让你的代码更容易读懂，更加pythonic。举个例子：

```

1. from collections import namedtuple
2.
3. # 变量名和namedtuple中的第一个参数一般保持一致，但也可以不一样
4. Student = namedtuple('Student', 'id name score')
5. # 或者 Student = namedtuple('Student', ['id', 'name', 'score'])
6.
7. students = [(1, 'Wu', 90), (2, 'Xing', 89), (3, 'Yuan', 98), (4,
'Wang', 95)]
8.
9. for s in students:
10.     stu = Student._make(s)
11.     print stu
12.

```

```
13. # Output:
14. # Student(id=1, name='Wu', score=90)
15. # Student(id=2, name='Xing', score=89)
16. # Student(id=3, name='Yuan', score=98)
17. # Student(id=4, name='Wang', score=95)
```

在上面的例子中，Student就是一个namedtuple，它和tuple的使用方法一样，可以通过index直接取，而且是只读的。这种方式比tuple容易理解多了，可以很清楚的知道每个值代表的含义。

Over！

var:http://segmentfault.com/blog/wuxianglong/1190000002399119?_ea=78694

[首页](#) | [上一讲](#)

第五部分 Python备忘录

链接

- [基本的（字面量）值](#)
- [运算符](#)
- [常用的内建函数](#)

基本的（字面量）值

- [基本的（字面量）值](#)

基本的（字面量）值

类型	描述	语法示例
整型	无小数部分的数	42
长整型	大整数	42L
浮点型	有小数部分的数	42.5, 42.5e-2
复合型	实数（整数或浮点数）和虚数的和	38+4j, 42j
字符串	不可变的字符序列	"foo", 'bar', ""baz"", r'\n'
Unicode	不可变的Unicode字符序列	u'foo', u"bar", u""baz""

运算符

- [运算符](#)

运算符

运算符	描述	优先级	
lambda	lambda表达式	1	
or	逻辑或	2	
and	逻辑与	3	
not	逻辑非	4	
in	成员资格测试	5	
not in	非成员资格测试	5	
is	一致性测试	6	
is not	非一致性测试	6	
<	小于	7	
>	大于	7	
<=	小于或等于	7	
>=	大于或等于	7	
==	等于	7	
!=	不等于	7	
\		按位或	8
^	按位异或	9	
&	按位与	10	
<<	左移	11	
>>	右移	11	
+	加法	12	
-	减法	12	
*	乘法	13	
/	除法	13	
%	求余	13	
+	一元一致性	14	
-	一元不一致性	14	

<code>~</code>	按位补码	15
<code>**</code>	幂	16
<code>x.attribute</code>	特性引用	17
<code>x[index]</code>	项目访问	18
<code>x[index1:index2[:index3]]</code>	切片	19
<code>f(arg...)</code>	函数调用	20
<code>(...)</code>	将表达式加圆括号或元组显示	21
<code>[...]</code>	列表显示	22
<code>{key:value, ...}</code>	字典显示	23
<code>'expressions...'</code>	字符串转化	24



常用的内建函数

- 一些重要的内建函数

一些重要的内建函数

函数	描述
<code>abs(number)</code>	返回一个数的绝对值
<code>apply(function[, args[, kwds]])</code>	调用给定函数，可选择提供参数
<code>all(iterable)</code>	如果所有iterable的元素均为真则返回True，否则返回False
<code>any(iterable)</code>	如果有任一iterable的元素为真则返回True，否则返回False
<code>basestring()</code>	str和unicode抽象超类，用于检查类型
<code>bool(object)</code>	返回True或False，取决于Object的布尔值
<code>callable(object)</code>	检查对象是否可调用
<code>chr(number)</code>	返回ASCII码为给定数字的字符
<code>classmethod(func)</code>	通过一个实例方法创建类的方法
<code>cmp(x, y)</code>	比较x和y—如果xy则返回证书；如果x==y，返回0
<code>complex(real[, imag])</code>	返回给定实部（以及可选的虚部）的复数
<code>delattr(object, name)</code>	从给定的对象中删除给定的属性
<code>dict([mapping-or-sequence])</code>	构造一个字典，可选择从映射或（键、值）对组成的列表构造。 也可以使用关键字参数调用。
<code>dir([object])</code>	当前可见作用于域的（大多数）名称的列表，或者是选择性地列出给定对象的（大多数）特性
<code>divmod(a, b)</code>	返回(a//b, a%b)（float类型有特殊规则）
<code>enumerate(iterable)</code>	对iterable中的所有项迭代（索引，项目）对
<code>eval(string[, globals[, locals]])</code>	对包含表达式的字符串进行计算。 可选择在给定的全局作用域或者局部作用域中进行
<code>execfile(file[, globals[, locals]])</code>	执行一个python文件， 可选在给定全局作用域或者局部作用域中进行
<code>file(filename[, mode[, bufsize]])</code>	创建给定文件名的文件， 可选择使用给定的模式和缓冲区大小
<code>filter(function, sequence)</code>	返回给定序列中函数返回值的元素的列表
<code>float(object)</code>	将字符串或者数值转换为float类型

<code>frozenset([iterable])</code>	创建一个不可变集合，这意味着不能将添加到其它集合中
<code>getattr(object, name[, default])</code>	返回给定对象中所指定的特性的值，可选择给定默认值
<code>globals()</code>	返回表示当前作用域的字典
<code>hasattr(object, name)</code>	检查给定的对象是否有指定的属性
<code>help([object])</code>	调用内建的帮助系统，或者打印给定对象的帮助信息
<code>id(number)</code>	返回给定对象的唯一ID
<code>input([prompt])</code>	等同于 <code>eval(raw_input(prompt))</code>
<code>int(object[, radix])</code>	将字符串或者数字（可以提供基数）转换为整数
<code>isinstance(object, classinfo)</code>	检查给定的对象 <code>object</code> 是否是给定的 <code>classinfo</code> 值的实例， <code>classinfo</code> 可以是类对象、类型对象或者类对象和类型对象的元组
<code>issubclass(class1, class2)</code>	检查 <code>class1</code> 是否是 <code>class2</code> 的子类（每个类都是自身的子类）
<code>iter(object[, sentinel])</code>	返回一个迭代器对象，可以是用于迭代序列的 <code>objectiter()</code> 迭代器（如果 <code>object</code> 支持 <code>_getitem</code> 方法的话），或者提供一个 <code>sentinel</code> ，迭代器会在每次迭代中调用 <code>object</code> ，直到返回 <code>sentinel</code>
<code>len(object)</code>	返回给定对象的长度（项的个数）
<code>list([sequence])</code>	构造一个列表，可选择使用与所提供序列 <code>sequence</code> 相同的项
<code>locals()</code>	返回表示当前局部作用域的字典（不要修改这个字典）
<code>long(object[, radix])</code>	将字符串（可选择使用给定的基数 <code>radix</code> ）或者数字转化为长整型
<code>map(function, sequence, ...)</code>	创建由给定函数 <code>function</code> 应用到所提供列表 <code>sequence</code> 每个项目时返回的值组成的列表
<code>max(object1, [object2, ...])</code>	如果 <code>object1</code> 是非空序列，那么就返回最大的元素。否则返回所提供参数（ <code>object1, object2...</code> ）的最大值
<code>min(object1, [object2, ...])</code>	如果 <code>object1</code> 是非空序列，那么就返回最小的元素。否则返回所提供参数（ <code>object1, object2...</code> ）的最小值
<code>object()</code>	返回所有新式类的技术Object的实例
<code>oct(number)</code>	将整型数转换为八进制表示的字符串
<code>open(filename[, mode[, bufsize]])</code>	<code>file</code> 的别名（在打开文件的时候使用 <code>open</code> 而不是 <code>file</code> ）
<code>ord(char)</code>	返回给定单字符（长度为1的字符串或者Unicode字符串）的ASCII值
<code>pow(x, y[, z])</code>	返回 <code>x</code> 的 <code>y</code> 次方，可选择模除 <code>z</code>
<code>property([fget[, fset[,</code>	通过一组访问器创建属性

<code>fdel[, doc]]])</code>	通过一组访问器创建属性
<code>range([start,]stop[, step])</code>	使用给定的起始值（包括起始值，默认为0）和结束值（不包括）以及步长（默认为1）返回数值范围（以列表形式）
<code>raw_input([prompt])</code>	将用户输入的数据作为字符串返回，可选择使用给定的提示符prompt
<code>reduce(function, sequence[, initializer])</code>	对序列的所有项渐增地应用于给定的函数，使用累积的结果作为第一个参数，所有的项作为第二个参数，可选择给定的起始值（initializer）
<code>reload(module)</code>	重载一个已经载入的模块并将其返回
<code>repr(object)</code>	返回表示对象的字符串，一般作为eval的参数使用
<code>reversed(sequence)</code>	返回序列的反向迭代器
<code>round(float[, n])</code>	将给定的浮点数四舍五入，小数点后保留n位（默认为0）
<code>set([iterable])</code>	返回从iterable（如果给出）生成的元素集合
<code>setattr(object, name, value)</code>	设定给定对象的指定属性的值为给定的值
<code>sorted(iterable[, cmp][, key][, reverse])</code>	从iterable的项目中返回一个新的排序后的列表。可选的参数和列表方法与sort中的相同
<code>staticmethod(func)</code>	从一个实例方法创建静态（类）方法
<code>str(object)</code>	返回表示给定对象object的格式化好的字符串
<code>sum(seq[, start])</code>	返回添加到可选参数start（默认为0）中的一系列数字的和
<code>super(type[, obj/type])</code>	返回给定类型（可选为实例化的）的超类
<code>tuple([sequence])</code>	构造一个元组，可选择使用同提供的序列sequence一样的项
<code>type(object)</code>	返回给定对象的类型
<code>type(name, base, dict)</code>	使用给定的名称、基类和作用域返回一个新的类型对象
<code>unichr(number)</code>	chr的Unicode版本
<code>unicode(object[, encoding[, errors]])</code>	返回给定对象的Unicode编码版本，可以给定编码方式和处理错误的模式（'strict'、'replace'或者'ignore'，'strict'为默认模式）
<code>vars([object])</code>	返回表示局部作用域的字典，或者对应给定对象特性的字典
<code>xrange([start,]stop[, step])</code>	类似于range，但是返回的对象使用内存较少，而且只用于迭代
<code>zip(sequence1, ...)</code>	返回元组的列表，每个元组包括一个给定序列中的项。返回的列表的长度和所提供的序列的最短长度相同

人生苦短，我用Python

- 人生苦短，我用Python
 - 一门编程语言的发展简史
- 起源
- 一门语言的诞生
- 时势造英雄
- 启示录

人生苦短，我用Python

一门编程语言的发展简史

Python是我喜欢的语言，简洁，优美，容易使用。前两天，我很激昂的向朋友宣传Python的好处。

“好吧，我承认Python不错，但它为什么叫Python呢？”

“呃，似乎是一个电视剧的名字。”

“那你说的Guido是美国人么？”

“他从Google换到Dropbox工作，但他的名字像是荷兰人的。”

“你确定你很熟悉Python吗？”

所以为了雪耻，我花时间调查了Python的历史。我看到了Python中许多功能的来源和Python的设计理念，看到了一门编程语言的演化历史，看到了Python与开源运动的奇妙联系。从Python的历史中，我们可以一窥开源开发的理念和成就。

这也可以作为我写的Python快速教程的序篇。

起源

Python的作者，Guido von Rossum，确实是荷兰人。1982年，Guido从阿姆斯特丹大学获得了数学和计算机硕士学位。然而，尽管他算得上是一位数学家，但他更加享受计算机带来的乐趣。用他的话说，尽管拥有数学和计算机双料资质，他总趋向于做计算机相关的工作，并热衷于做任何和编程相关的活儿。

在那个时候，Guido接触并使用过诸如Pascal、C、Fortran等语言。这些语言的基本设计原则是让机器能更快运行。在80年代，虽然IBM和苹果已经掀起了个人电脑浪潮，但这些个人电脑的配置很低。比如早期的Macintosh，只有8MHz的CPU主频和128KB的RAM，一个大的数组就能占满内存。所有的编译器的核心是做优化，以便让程序能够运行。为了增进效率，语言也迫使程序员像计算机一样思考，以便能写出更符合机器口味的程序。在那个时代，程序员恨不得用手榨取计算机每一寸的能力。有人甚至认为C语言的指针是在浪费内存。至于动态类型，内存自动管理，面向对象..... 别想了，那会让你的电脑陷入瘫痪。

这种编程方式让Guido感到苦恼。Guido知道如何用C语言写出一个功能，但整个编写过程需要耗费大量的时间，即使他已经准确的知道了如何实现。他的另一个选择是shell。Bourne Shell作为UNIX系统的解释器已经长期存在。UNIX的管理员们常常用shell去写一些简单的脚本，以进行一些系统维护的工作，比如定期备份、文件系统管理等等。shell可以像胶水一样，将UNIX下的许多功能连接在一起。许多C语言下上百行的程序，在shell下只用几行就可以完成。然而，shell的本质是调用命令。它并不是一个真正的语言。比如说，shell没有数值型的数据类型，加法运算都很复杂。总之，shell不能全面的调动计算机的功能。

Guido希望有一种语言，这种语言能够像C语言那样，能够全面调用计算机的功能接口，又可以像shell那样，可以轻松的编程。ABC语言让Guido看到希望。ABC是由荷兰的数学和计算机研究所开发的。Guido

在该研究所工作，并参与到ABC语言的开发。ABC语言以教学为目的。与当时的大部分语言不同，ABC语言的目标是“让用户感觉更好”。ABC语言希望让语言变得容易阅读，容易使用，容易记忆，容易学习，并以此来激发人们学习编程的兴趣。比如下面是一段来自Wikipedia的ABC程序，这个程序用于统计文本中出现的词的总数：

```
1.  HOW TO RETURN words document:
2.
3.      PUT {} IN collection
4.
5.      FOR line IN document:
6.
7.          FOR word IN split line:
8.
9.              IF word not.in collection:
10.
11.                  INSERT word IN collection
12.
13.      RETURN collection
```

HOW TO用于定义一个函数。一个Python程序员应该很容易理解这段程序。ABC语言使用冒号和缩进来表示程序块。行尾没有分号。for和if结构中也没有括号()。赋值采用的是PUT，而不是更常见的等号。这些改动让ABC程序读起来像一段文字。

尽管已经具备了良好的可读性和易用性，ABC语言最终没有流行起来。在当时，ABC语言编译器需要比较高配置的电脑才能运行。而这些电脑的使用者通常精通计算机，他们更多考虑程序的效率，而非它的学习难度。除了硬件上的困难外，ABC语言的设计也存在一些致命的问题：

- 可拓展性差。ABC语言不是模块化语言。如果想在ABC语言中增加功能，比如对图形化的支持，就必须改动很多地方。
- 不能直接进行IO。ABC语言不能直接操作文件系统。尽管你可以通

过诸如文本流的方式导入数据，但ABC无法直接读写文件。输入输出的困难对于计算机语言来说是致命的。你能想像一个打不开车门的跑车么？

- 过度革新。ABC用自然语言的方式来表达程序的意义，比如上面程序中的HOW TO 。然而对于程序员来说，他们更习惯用function或者define来定义一个函数。同样，程序员更习惯用等号来分配变量。尽管ABC语言很特别，但学习难度也很大。
- 传播困难。ABC编译器很大，必须被保存在磁带上。当时Guido在访问的时候，就必须有一个大磁带来给别人安装ABC编译器。 这样，ABC语言就很难快速传播。

1989年，为了打发圣诞节假期，Guido开始写Python语言的编译器。Python这个名字，来自Guido所挚爱的电视剧Monty Python's Flying Circus。他希望这个新的叫做Python的语言，能符合他的理想：创造一种C和shell之间，功能全面，易学易用，可拓展的语言。Guido作为一个语言设计爱好者，已经有过设计语言的尝试。这一次，也不过是一次纯粹的hacking行为。

一门语言的诞生

1991年，第一个Python编译器诞生。它是用C语言实现的，并能够调用C语言的库文件。从一出生，Python已经具有了：类，函数，异常处理，包含表和词典在内的核心数据类型，以及模块为基础的拓展系统。

Python语法很多来自C，但又受到ABC语言的强烈影响。来自ABC语言的一些规定直到今天还富有争议，比如强制缩进。但这些语法规则让Python容易读。另一方面，Python聪明的选择服从一些惯例，特别是C语言的惯例。比如使用等号赋值，使用def来定义函数。Guido认为，如果“常识”上确立的东西，没有必要过度纠结。

Python从一开始就特别在意可拓展性。Python可以在多个层次上拓展。从高层上，你可以直接引入.py文件。在底层，你可以引用C语言的库。Python程序员可以快速的使用Python写.py文件作为拓展模块。但当性能是考虑的重要因素时，Python程序员可以深入底层，写C程序，编译为.so文件引入到Python中使用。Python就好像是使用钢构建房一样，先规定好大的框架。而程序员可以在此框架下相当自由的拓展或更改。

最初的Python完全由Guido本人开发。Python得到Guido同事的欢迎。他们迅速的反馈使用意见，并参与到Python的改进。Guido和一些同事构成Python的核心团队。他们将自己大部分的业余时间用于hack Python。随后，Python拓展到研究所之外。Python将许多机器层面上的细节隐藏，交给编译器处理，并凸显出逻辑层面的编程思考。Python程序员可以花更多的时间用于思考程序的逻辑，而不是具体的实现细节。这一特征吸引了广大的程序员。Python开始流行。

人生苦短，我用python



时势造英雄

我们不得不暂停我们的Python时间，转而看一看瞬息万变的计算机行业。1990年代初，个人计算机开始进入普通家庭。Intel发布了486处理器，windows发布window 3.0开始的一系列视窗系统。计算机的性能大大提高。程序员开始关注计算机的易用性，比如图形化界面。



由于计算机性能的提高，软件的世界也开始随之改变。硬件足以满足许多个人电脑的需要。硬件厂商甚至渴望高需求软件的出现，以带动硬件

的更新换代。C++和Java相继流行。C++和Java提供了面向对象的编程范式，以及丰富的对象库。在牺牲了一定的性能的代价下，C++和Java大大提高了程序的产量。语言的易用性被提到一个新的高度。我们还记得，ABC失败的一个重要原因是硬件的性能限制。从这方面说，Python要比ABC幸运许多。

另一个悄然发生的改变是Internet。1990年代还是个人电脑的时代，windows和Intel挟PC以令天下，盛极一时。尽管Internet为主体的信息革命尚未到来，但许多程序员以及资深计算机用户已经在频繁使用Internet进行交流，比如使用email和newsgroup。Internet让信息交流成本大大下降。一种新的软件开发模式开始流行：开源。程序员利用业余时间进行软件开发，并开放源代码。1991年，Linus在comp.os.minix新闻组上发布了Linux内核源代码，吸引大批hacker的加入。Linux和GNU相互合作，最终构成了一个充满活力的开源平台。

硬件性能不是瓶颈，Python又容易使用，所以许多人开始转向Python。Guido维护了一个maillist，Python用户就通过邮件进行交流。Python用户来自许多领域，有不同的背景，对Python也有不同的需求。Python相当的开放，又容易拓展，所以当用户不满足于现有功能，很容易对Python进行拓展或改造。随后，这些用户将改动发给Guido，并由Guido决定是否将新的特征加入到Python或者标准库中。如果代码能被纳入Python自身或者标准库，这将极大的荣誉。由于Guido至高无上的决定权，他因此被称为“终身的仁慈独裁者”。

Python被称为“Battery Included”，是说它以及其标准库的功能强大。这些是整个社区的贡献。Python的开发者来自不同领域，他们将不同领域的优点带给Python。比如Python标准库中的正则表达是参考Perl，而lambda, map, filter, reduce等函数参考了Lisp。Python本身的一些功能以及大部分的标准库来自于社区。Python的社

区不断扩大，进而拥有了自己的newsgroup，网站，以及基金。从Python 2.0开始，Python也从maillist的开发方式，转为完全开源的开发方式。社区气氛已经形成，工作被整个社区分担，Python也获得了更加高速的发展。

到今天，Python的框架已经确立。Python语言以对象为核心组织代码，支持多种编程范式，采用动态类型，自动进行内存回收。Python支持解释运行，并能调用C库进行拓展。Python有强大的标准库。由于标准库的体系已经稳定，所以Python的生态系统开始拓展到第三方包。这些包，如Django、web.py、wxpython、numpy、matplotlib、PIL，将Python升级成了物种丰富的热带雨林。

启示录

Python崇尚优美、清晰、简单，是一个优秀并广泛使用的语言。Python在TIOBE排行榜中排行第八，它是Google的第三大开发语言，Dropbox的基础语言，豆瓣的服务器语言。Python的发展史可以作为一个代表，带给我许多启示。

在Python的开发过程中，社区起到了重要的作用。Guido自认为自己不是全能型的程序员，所以他只负责制订框架。如果问题太复杂，他会选择绕过去，也就是cut the corner。这些问题最终由社区中的其他人解决。社区中的人才异常丰富的，就连创建网站，筹集基金这样与开发稍远的事情，也有人乐意于处理。如今的项目开发越来越复杂，越来越庞大，合作以及开放的心态成为项目最终成功的关键。

Python从其他语言中学到了很多，无论是已经进入历史的ABC，还是依然在使用的C和Perl，以及许多没有列出的其他语言。可以说，Python的成功代表了它所有借鉴的语言的成功。同样，Ruby借鉴了Python，它的成功也代表了Python某些方面的成功。每个语言都是混

合体，都有它优秀的地方，但也有各种各样的缺陷。同时，一个语言“好与不好”的评判，往往受制于平台、硬件、时代等等外部原因。程序员经历过许多语言之争。其实，以开放的心态来接受各个语言，说不定哪一天，程序员也可以如Guido那样，混合出自己的语言。

无论Python未来的命运如何，Python的历史已经是本很有趣的小说。

var: [Python快速教程](#) - [Vamei](#) - [博客园](#)