

# Python 教程

这是小白的 Python 新手教程。

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

那 Python 是一种什么语言？

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

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

所以 Python 是一种相当高级的语言。

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

那是不是越低级的程序越难学，越高级的程序越简单？表面上来说，是的，但是，在非常高的抽象计算中，高级的 Python 程序设计也是非常难学的，所以，高级程序语言不等于简单。但是，对于初学者和完成普通任务，Python 语言是非常简单易用的。连 Google 都在大规模使用 Python，你就不用担心学了会没用。

用 Python 可以做什么？可以做日常任务，比如自动备份你的 MP3；可以做网站后台，你現在看到的网站就是 Python 写的；可以做网络游戏的后台，很多在线游戏的后台都是 Python 开发的。总之就是能干很多很多事啦。

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

如果你是小白用户，满足以下条件：

会使用电脑，但从来没写过程序；

还记得初中数学学的方程式和一点点代数知识；

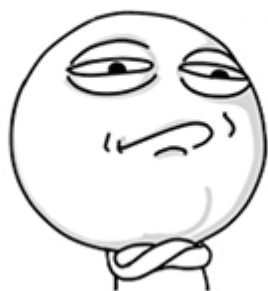
想从编程小白变成专业的软件架构师；

每天能抽出半个小时学习。

不要再犹豫了，这个教程就是为你准备的！

准备好了吗？


## CHALLENGE ACCEPTED!



关于作者

廖雪峰，十年软件开发经验，业余产品经理，精通 Java/Python/Ruby/Visual Basic/Objective C 等，对开源框架有深入研究，著有《Spring 2.0 核心技术与最佳实践》一书，多个业余开源项目托管在 GitHub，欢迎微博交流：



廖雪峰 ：确定从今天的雾霾中看到了春天？[转]ShimonPeres：朋友们，我..[图]

4月10日 17:49 | 微博

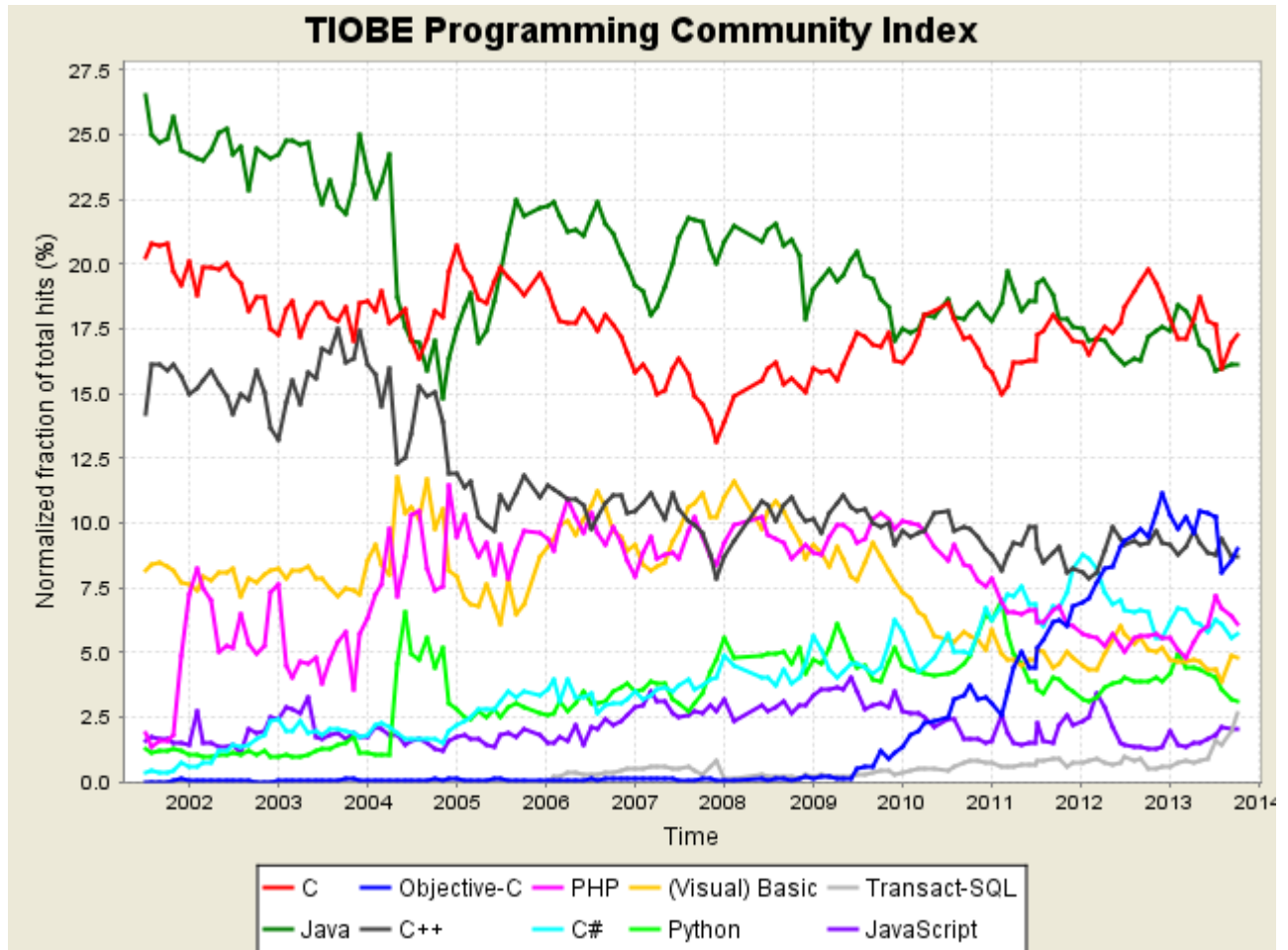


[weibo.com/liaoxuefeng](https://weibo.com/liaoxuefeng)

## Python 简介

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

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



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

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

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

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

许多大型网站就是用 Python 开发的，例如 YouTube、Instagram，还有国内的豆瓣。很多大公司，包括 Google、Yahoo 等，甚至 NASA（美国航空航天局）都大量地使用 Python。

龟叔给 Python 的定位是“优雅”、“明确”、“简单”，所以 Python 程序看上去总是简单易懂，初学者学 Python，不但入门容易，而且将来深入下去，可以编写那些非常非常复杂的程序。总的来说，Python 的哲学就是简单优雅，尽量写容易看明白的代码，尽量写少的代码。如果一个资深程序员向你炫耀他写的晦涩难懂、动不动就几万行的代码，你可以尽情地嘲笑他。

那 Python 适合开发哪些类型的应用呢？

首选是网络应用，包括网站、后台服务等等；

其次是许多日常需要的小工具，包括系统管理员需要的脚本任务等等；

另外就是把其他语言开发的程序再包装起来，方便使用。

最后说说 Python 的缺点。

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

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

而 C 程序是运行前直接编译成 CPU 能执行的机器码，所以非常快。

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



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

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

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



当然，Python 还有其他若干小缺点，请自行忽略，就不一一列举了。

## 安装 Python

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

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

2.x 还是 3.x

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

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

在 Mac 上安装 Python

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

查看系统版本的办法是点击左上角的苹果图标，选择“关于本机”：



在 Linux 上安装 Python

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

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

以下内容。

在 Windows 上安装 Python

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

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

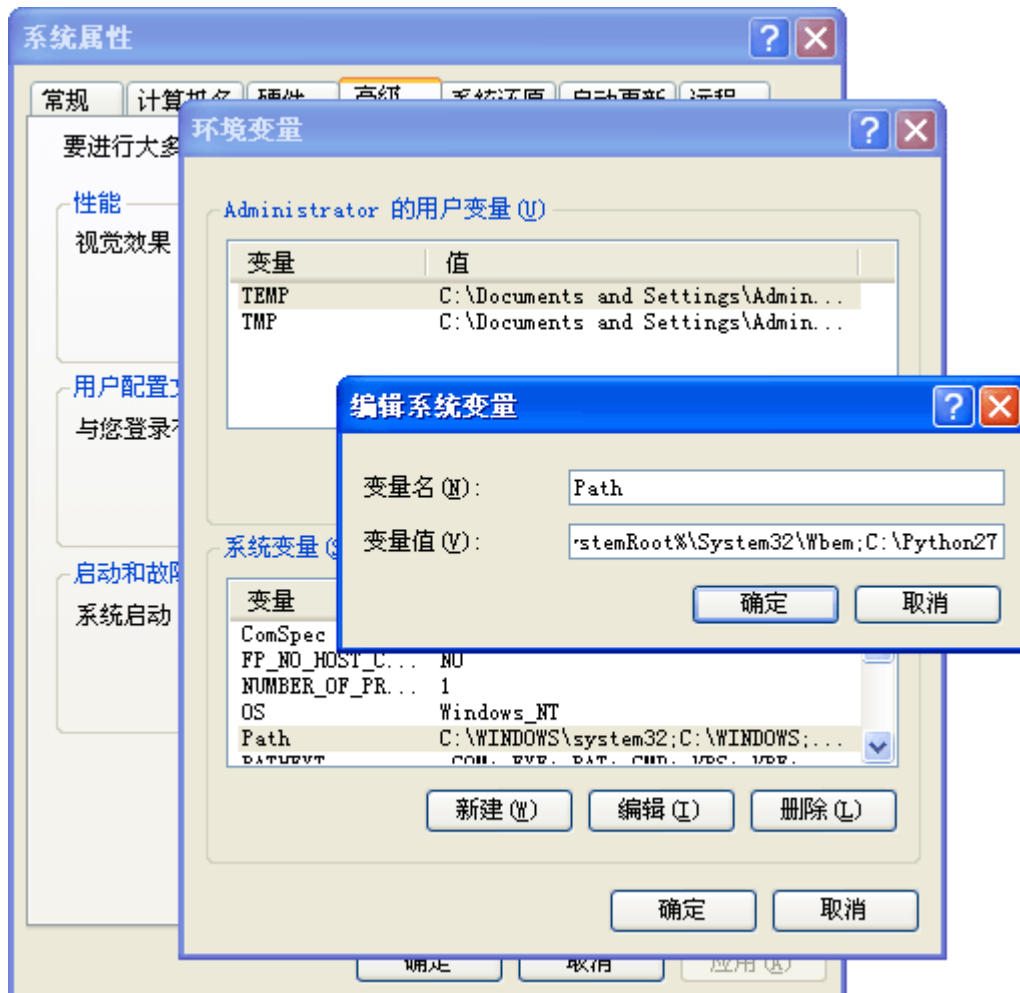
然后，运行下载的 MSI 安装包，不需要更改任何默认设置，直接一路点“Next”即可完成安装：

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

‘python’不是内部或外部命令，也不是可运行的程序或批处理文件。

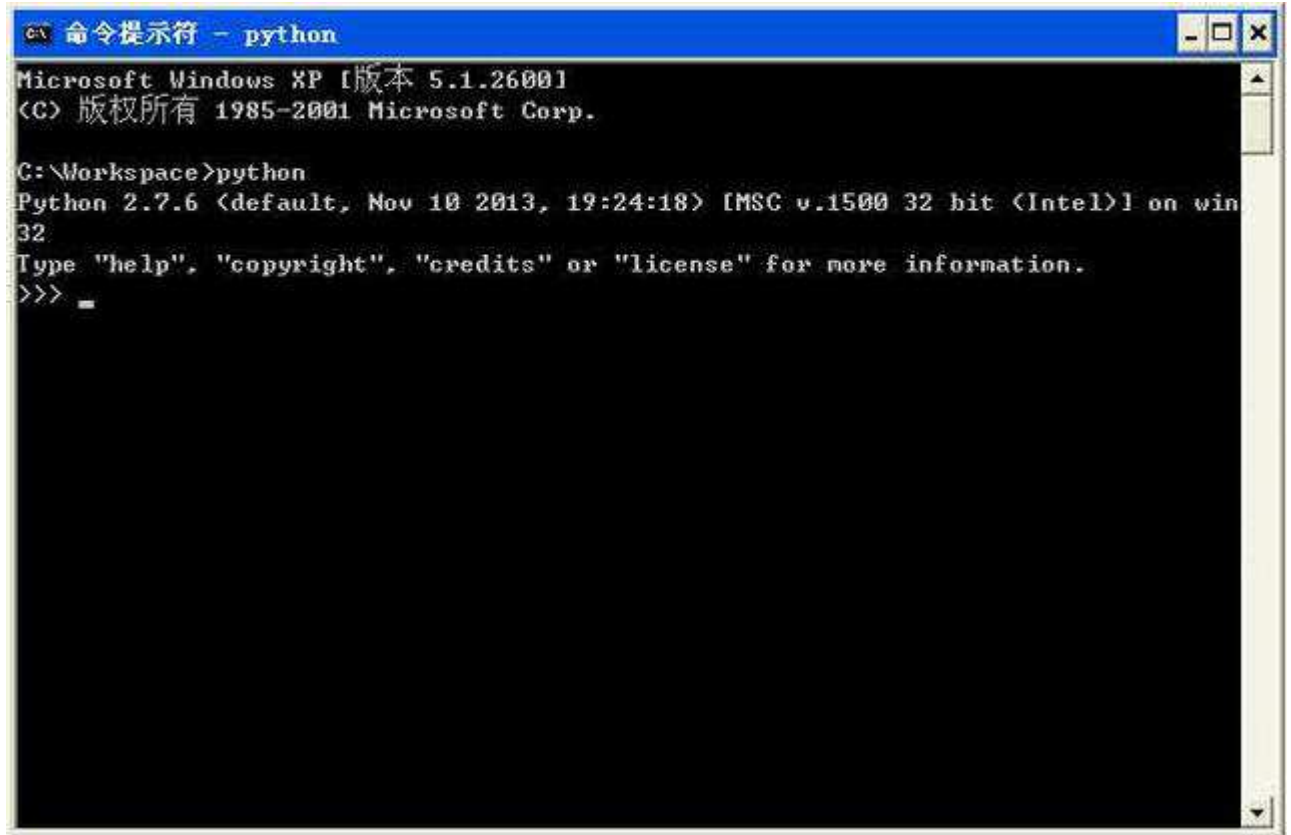
这是因为 Windows 会根据一个 Path 的环境变量设定的路径去查找 python.exe，如果没找到，就会报错。解决办法是把 python.exe 所在的路径 C:\Python27 添加到 Path 中。

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



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

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



```
命令提示符 - python
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Workspace>python
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit <Intel>] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

看到上面的画面，就说明 Python 安装成功！

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

小结

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

## 第一个 Python 程序

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

在写代码之前，请千万不要用“复制”-“粘贴”把代码从页面粘贴到你自己的电脑上。写程序也讲究一个感觉，你需要一个字母一个字母地把代码自己敲进去，在敲代码的过程中，初学者经常会敲错代码，所以，你需要仔细地检查、对照，才能以最快的速度掌握如何写程序。在交互式环境的提示符>>>下，直接输入代码，按回车，就可以立刻得到代码执行结果。现在，试试输入 100+200，看看计算结果是不是 300：

```
>>> 100+200
300
```



很简单吧，任何有效的数学计算都可以算出来。

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

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

这种用单引号或者双引号括起来的文本在程序中叫字符串，今后我们还会经常遇到。

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

## 小结

在 Python 交互式命令行下，可以直接输入代码，然后执行，并立刻得到结果。



## 使用文本编辑器

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

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

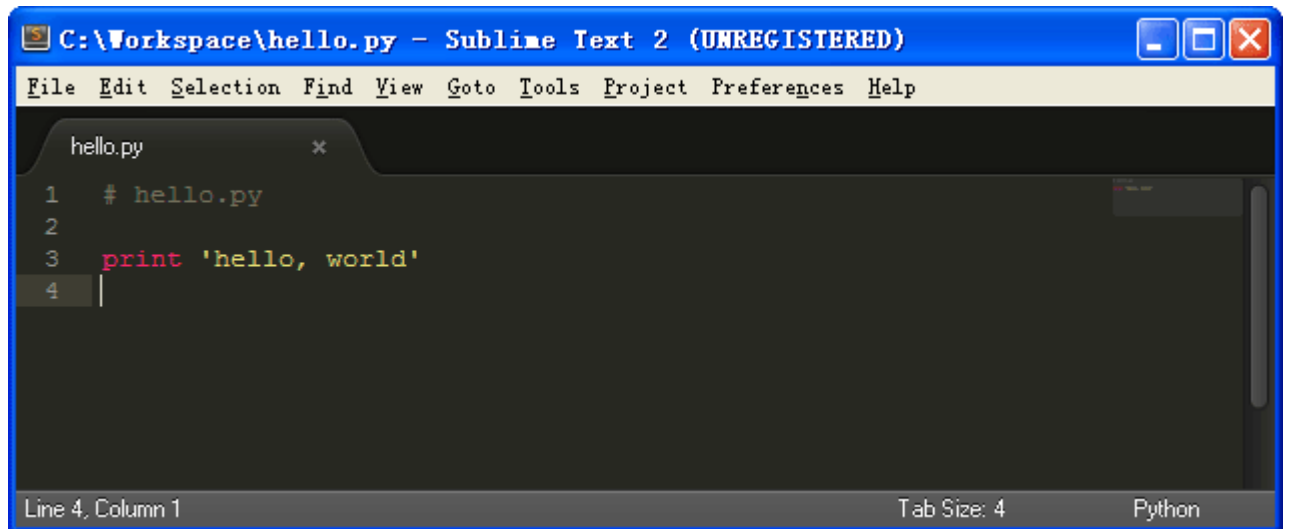
现在，我们就把上次的 'hello, world' 程序用文本编辑器写出来，保存下来。

所以问题又变成了：用什么文本编辑器？

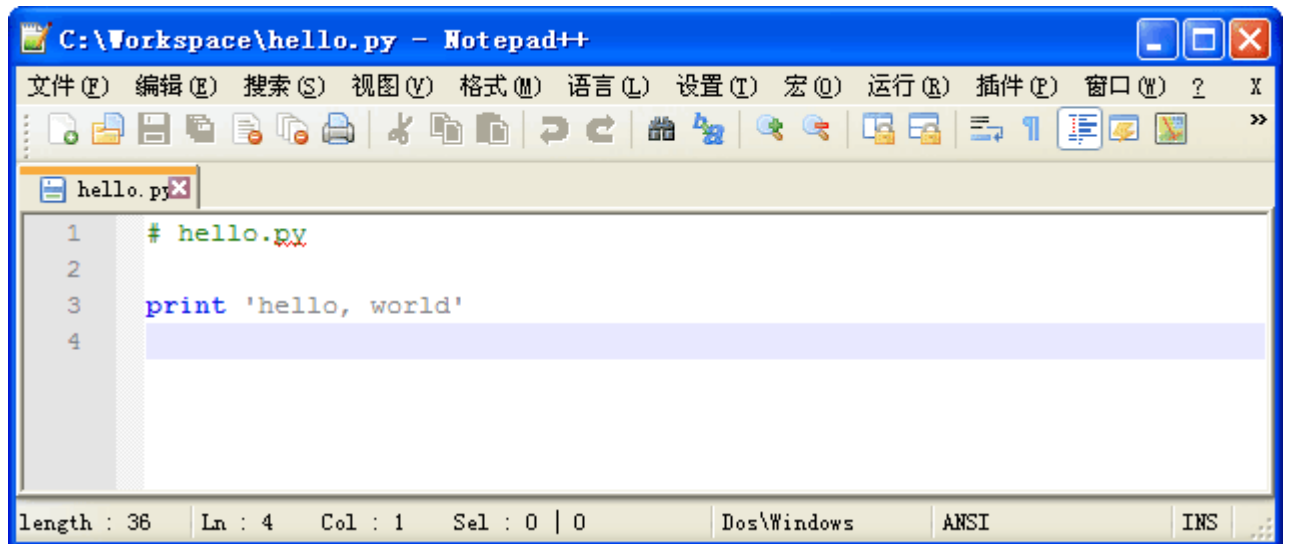
推荐两款文本编辑器：

一个是 Sublime Text，免费使用，但是不付费会弹出提示框：





一个是 Notepad++，免费使用，有中文界面：



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

安装好文本编辑器后，输入以下代码：

```
print 'hello, world'
```

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

```
C:\Workspace>python hello.py
```

```
hello, world
```

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

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

```
python hello.py
```

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

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

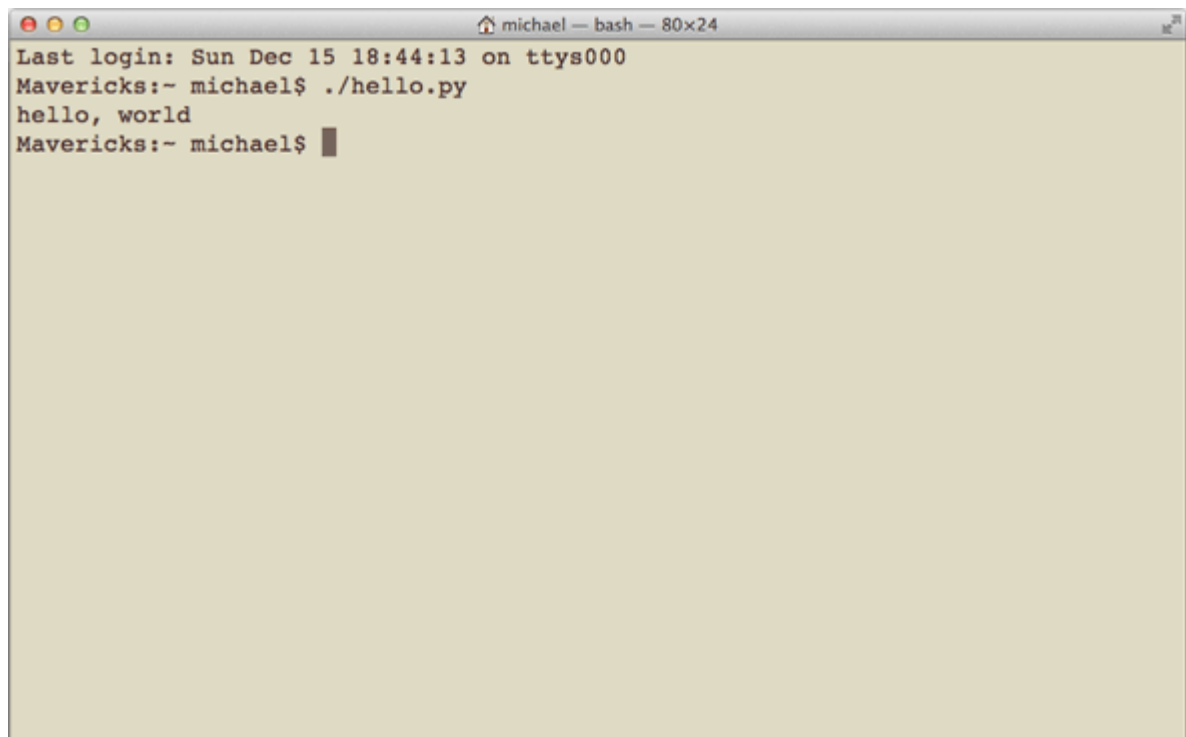
还有同学问，能不能像 `.exe` 文件那样直接运行 `.py` 文件呢？在 Windows 上是不行的，但是，在 Mac 和 Linux 上是可以的，方法是在 `.py` 文件的第一行加上：

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

然后，通过命令：

```
$ chmod a+x hello.py
```

就可以直接运行 `hello.py` 了，比如在 Mac 下运行：

A screenshot of a macOS terminal window titled 'michael — bash — 80x24'. The terminal shows the following text: 'Last login: Sun Dec 15 18:44:13 on ttys000', 'Mavericks:~ michael\$ ./hello.py', 'hello, world', and 'Mavericks:~ michael\$'. The prompt 'Mavericks:~ michael\$' is followed by a cursor. The terminal has a light beige background and a dark grey title bar with standard macOS window controls.

小结

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

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

## 输入和输出

输出

用 `print` 加上字符串，就可以向屏幕上输出指定的文字。比如输出 `'hello, world'`，用代码实现

如下：

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

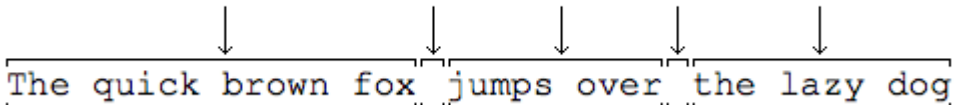
`print` 语句也可以跟上多个字符串，用逗号“,”隔开，就可以连成一串输出：

```
>>> print 'The quick brown fox', 'jumps over', 'the lazy dog'
```

The quick brown fox jumps over the lazy dog

`print` 会依次打印每个字符串，遇到逗号“,”会输出一个空格，因此，输出的字符串是这样拼起来的：

```
print 'The quick brown fox' , 'jumps over' , 'the lazy dog'
```



The quick brown fox jumps over the lazy dog

`print` 也可以打印整数，或者计算结果：

```
>>> print 300
```

300

```
>>> print 100 + 200
```

300

因此，我们可以把计算  $100 + 200$  的结果打印得更漂亮一点：

```
>>> print '100 + 200 =', 100 + 200
```

100 + 200 = 300

注意，对于  $100 + 200$ ，Python 解释器自动计算出结果 300，但是，`'100 + 200 ='` 是字符串而非数学公式，Python 把它视为字符串，请自行解释上述打印结果。

输入

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

```
>>> name = raw_input()
```

Michael

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

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

```
>>> name
```

'Michael'

什么是变量？请回忆初中数学所学的代数基础知识：

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

若  $a=2$ ，则面积为  $a \times a = 2 \times 2 = 4$ ；

若  $a=3.5$ ，则面积为  $a \times a = 3.5 \times 3.5 = 12.25$ 。

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

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

```
>>> print name
Michael
```

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

```
name = raw_input()
print 'hello,', name
```

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

```
C:\Workspace> python hello.py
Michael
hello, Michael
```

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

```
name = raw_input('please enter your name: ')
print 'hello,', name
```

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

```
C:\Workspace> python hello.py
please enter your name: Michael
hello, Michael
```

每次运行该程序，根据用户输入的不同，输出结果也会不同。

在命令行下，输入和输出就是这么简单。

小结

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

输入是 `Input`，输出是 `Output`，因此，我们把输入输出统称为 `Input/Output`，或者简写为 `IO`。`raw_input` 和 `print` 是在命令行下面最基本的输入和输出，但是，用户也可以通过其他更高级的图形界面完成输入和输出，比如，在网页上的一个文本框输入自己的名字，点击“确定”后在网页上看到输出信息。

## Python 基础

Python 是一种计算机编程语言。计算机编程语言和我们日常使用的自然语言有所不同，最大的区别就是，自然语言在不同的语境下有不同的理解，而计算机要根据编程语言执行任务，就必须保证编程语言写出的程序决不能有歧义，所以，任何一种编程语言都有自己的一套语

法，编译器或者解释器就是负责把符合语法的程序代码转换成 CPU 能够执行的机器码，然后执行。Python 也不例外。

Python 的语法比较简单，采用缩进方式，写出来的代码就像下面的样子：

```
# print absolute value of an integer:
```

```
a = 100
```

```
if a >= 0:
```

```
    print a
```

```
else:
```

```
    print -a
```

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

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

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

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

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

## 数据类型和变量

### 数据类型

计算机顾名思义就是可以做数学计算的机器，因此，计算机程序理所当然地可以处理各种数值。但是，计算机能处理的远不止数值，还可以处理文本、图形、音频、视频、网页等各种各样的数据，不同的数据，需要定义不同的数据类型。在 Python 中，能够直接处理的数据类型有以下几种：

#### 整数

Python 可以处理任意大小的整数，当然包括负整数，在程序中的表示方法和数学上的写法一模一样，例如：1，100，-8080，0，等等。

计算机由于使用二进制，所以，有时候用十六进制表示整数比较方便，十六进制用 0x 前缀和 0-9，a-f 表示，例如：0xff00，0xa5b4c3d2，等等。

#### 浮点数

浮点数也就是小数，之所以称为浮点数，是因为按照科学记数法表示时，一个浮点数的小数点位置是可变的，比如，1.23x10<sup>9</sup> 和 12.3x10<sup>8</sup> 是相等的。浮点数可以用数学写法，如 1.23，3.14，-9.01，等等。但是对于很大或很小的浮点数，就必须用科学计数法表示，把 10 用 e 替代，1.23x10<sup>9</sup> 就是 1.23e9，或者 12.3e8，0.000012 可以写成 1.2e-5，等等。

整数和浮点数在计算机内部存储的方式是不同的，整数运算永远是精确的（除法难道也是精确的？是的！），而浮点数运算则可能会有四舍五入的误差。

#### 字符串

字符串是以"或"'括起来的任意文本，比如'abc'，"xyz"等等。请注意，"或"'本身只是一种表示方式，不是字符串的一部分，因此，字符串'abc'只有 a，b，c 这 3 个字符。如果'本身也是

一个字符，那就可以用""括起来，比如"I'm OK"包含的字符是 I, ', m, 空格, O, K 这 6 个字符。

如果字符串内部既包含'又包含"怎么办？可以用转义字符\来标识，比如：

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

表示的字符串内容是：

```
I'm "OK"!
```

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

```
>>> print 'I\'m ok.'
I'm ok.
>>> print 'I\'m learning\nPython.'
I'm learning
Python.
>>> print '\\n\\'
\
\
```

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

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

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

```
>>> print """line1
... line2
... line3"""
line1
line2
line3
```

上面是在交互式命令行内输入，如果写成程序，就是：

```
print """line1
line2
line3"""
```

多行字符串"""..."""还可以在前面加上 r 使用，请自行测试。

布尔值

布尔值和布尔代数的表示完全一致，一个布尔值只有 True、False 两种值，要么是 True，要么是 False，在 Python 中，可以直接用 True、False 表示布尔值（请注意大小写），也可以通

过布尔运算计算出来：

```
>>> True
True
>>> False
False
>>> 3 > 2
True
>>> 3 > 5
False
```

布尔值可以用 `and`、`or` 和 `not` 运算。

`and` 运算是与运算，只有所有都为 `True`，`and` 运算结果才是 `True`：

```
>>> True and True
True
>>> True and False
False
>>> False and False
False
```

`or` 运算是或运算，只要其中有一个为 `True`，`or` 运算结果就是 `True`：

```
>>> True or True
True
>>> True or False
True
>>> False or False
False
```

`not` 运算是非运算，它是一个单目运算符，把 `True` 变成 `False`，`False` 变成 `True`：

```
>>> not True
False
>>> not False
True
```

布尔值经常用在条件判断中，比如：

```
if age >= 18:
    print 'adult'
else:
    print 'teenager'
```

空值

空值是 Python 里一个特殊的值，用 `None` 表示。`None` 不能理解为 0，因为 0 是有意义的，而 `None` 是一个特殊的空值。

此外，Python 还提供了列表、字典等多种数据类型，还允许创建自定义数据类型，我们后面会继续讲到。



## 变量

变量的概念基本上和初中代数的方程变量是一致的，只是在计算机程序中，变量不仅可以是数字，还可以是任意数据类型。

变量在程序中就是用一个变量名表示了，变量名必须是大小写英文、数字和\_的组合，且不能用数字开头，比如：

```
a = 1
```

变量 `a` 是一个整数。

```
t_007 = 'T007'
```

变量 `t_007` 是一个字符串。

```
Answer = True
```

变量 `Answer` 是一个布尔值 `True`。

在 Python 中，等号=是赋值语句，可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量，例如：

```
a = 123 # a 是整数
print a
a = 'ABC' # a 变为字符串
print a
```

这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。例如 Java 是静态语言，赋值语句如下（// 表示注释）：

```
int a = 123; // a 是整数类型变量
a = "ABC"; // 错误：不能把字符串赋给整型变量
```

和静态语言相比，动态语言更灵活，就是这个原因。

请不要把赋值语句的等号等同于数学的等号。比如下面的代码：

```
x = 10
x = x + 2
```

如果从数学上理解  $x = x + 2$  那无论如何是不成立的，在程序中，赋值语句先计算右侧的表达式  $x + 2$ ，得到结果 12，再赋给变量 `x`。由于 `x` 之前的值是 10，重新赋值后，`x` 的值变成 12。

最后，理解变量在计算机内存中的表示也非常重要。当我们写：

```
a = 'ABC'
```

时，Python 解释器干了两件事情：

在内存中创建了一个'ABC'的字符串；

在内存中创建了一个名为 `a` 的变量，并把它指向'ABC'。

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

```
a = 'ABC'
```

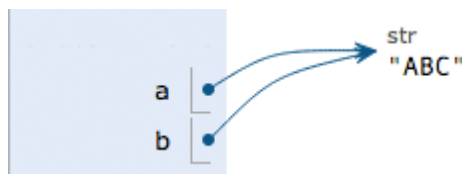
```
b = a
a = 'XYZ'
print b
```

最后一行打印出变量 `b` 的内容到底是'ABC'呢还是'XYZ'? 如果从数学意义上理解, 就会错误地得出 `b` 和 `a` 相同, 也应该是'XYZ', 但实际上 `b` 的值是'ABC', 让我们一行一行地执行代码, 就可以看到到底发生了什么事:

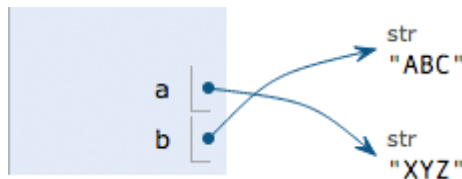
执行 `a = 'ABC'`, 解释器创建了字符串'ABC'和变量 `a`, 并把 `a` 指向'ABC':



执行 `b = a`, 解释器创建了变量 `b`, 并把 `b` 指向 `a` 指向的字符串'ABC':



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



所以, 最后打印变量 `b` 的结果自然是'ABC'了。

常量

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

```
PI = 3.14159265359
```

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

最后解释一下整数的除法为什么也是精确的, 可以试试:

```
>>> 10 / 3
3
```

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

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

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

```
>>> 10 % 3
```

无论整数做除法还是取余数，结果永远是整数，所以，整数运算结果永远是精确的。

小结

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

## 字符串和编码

字符编码

我们已经讲过了，字符串也是一种数据类型，但是，字符串比较特殊的是还有一个编码问题。因为计算机只能处理数字，如果要处理文本，就必须先把文本转换为数字才能处理。最早的计算机在设计时采用 8 个比特（bit）作为一个字节（byte），所以，一个字节能表示的最大的整数就是 255（二进制 11111111=十进制 255），如果要表示更大的整数，就必须用更多的字节。比如两个字节可以表示的最大整数是 65535，4 个字节可以表示的最大整数是 4294967295。

由于计算机是美国人发明的，因此，最早只有 127 个字母被编码到计算机里，也就是大小写英文字母、数字和一些符号，这个编码表被称为 ASCII 编码，比如大写字母 A 的编码是 65，小写字母 z 的编码是 122。

但是要处理中文显然一个字节是不够的，至少需要两个字节，而且还不能和 ASCII 编码冲突，所以，中国制定了 GB2312 编码，用来把中文编进去。

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



因此，Unicode 应运而生。Unicode 把所有语言都统一到一套编码里，这样就不会再有乱码问题了。

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

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

字母 A 用 ASCII 编码是十进制的 65，二进制的 01000001；

字符 0 用 ASCII 编码是十进制的 48，二进制的 00110000，注意字符'0'和整数 0 是不同的；汉字中已经超出了 ASCII 编码的范围，用 Unicode 编码是十进制的 20013，二进制的 01001110 00101101。

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

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

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

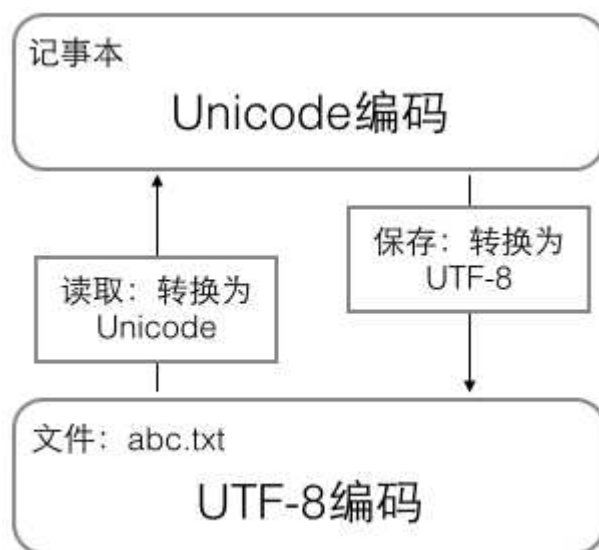
字符	ASCII	Unicode	UTF-8
A	01000001	00000000 01000001	01000001
中	x	01001110 00101101	11100100 10101101

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

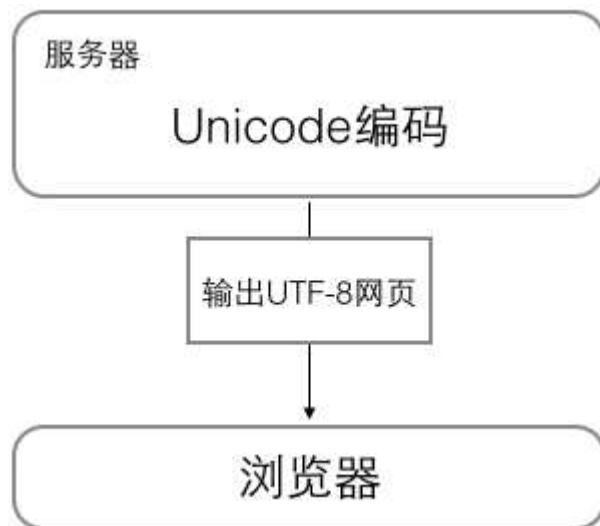
搞清楚了 ASCII、Unicode 和 UTF-8 的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

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

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



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



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

Python 的字符串

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

因为 Python 的诞生比 Unicode 标准发布的时间还要早，所以最早的 Python 只支持 ASCII 编码，普通的字符串'ABC'在 Python 内部都是 ASCII 编码的。Python 提供了 ord()和 chr()函数，可以把字母和对应的数字相互转换：

```
>>> ord('A')
65
>>> chr(65)
'A'
```

Python 在后来添加了对 Unicode 的支持，以 Unicode 表示的字符串用 u'...'表示，比如：

```
>>> print u'中文'
中文
>>> u'中'
u'\u4e2d'
```

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

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

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

```
>>> u'ABC'.encode('utf-8')
'ABC'
>>> u'中文'.encode('utf-8')
'\xe4\xb8\xad\xe6\x96\x87'
```

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

以返回字符串的长度：

```
>>> len(u'ABC')
3
>>> len('ABC')
3
>>> len(u'中文')
2
>>> len('\xe4\xb8\xad\xe6\x96\x87')
6
```

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

```
>>> 'abc'.decode('utf-8')
u'abc'
>>> '\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
u'\u4e2d\u6587'
>>> print '\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
中文
```

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

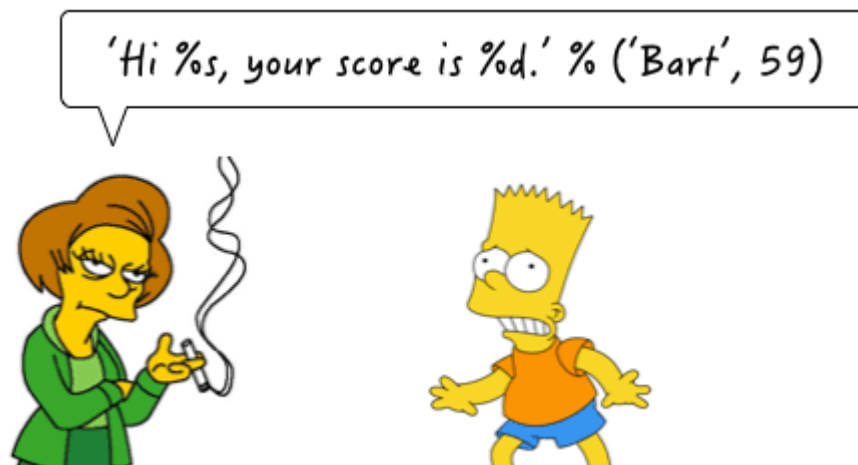
```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

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

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

格式化

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



在 Python 中，采用的格式化方式和 C 语言是一致的，用%实现，举例如下：

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

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

常见的占位符有：

%d	整数
%f	浮点数
%s	字符串
%x	十六进制整数

其中，格式化整数和浮点数还可以指定是否补 0 和整数与小数的位数：

```
>>> '%2d-%02d' % (3, 1)
'3-01'
>>> '%.2f' % 3.1415926
'3.14'
```

如果你不太确定应该用什么，%s 永远起作用，它会把任何数据类型转换为字符串：

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

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

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

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

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

小结

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

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

```
>>> u'中文'.encode('gb2312')
'\xd6\xd0\xce\xca'
```

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



种编码方式。

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

格式化字符串的时候，可以用 Python 的交互式命令行测试，方便快捷。

## 使用 list 和 tuple

list

Python 内置的一种数据类型是列表：list。list 是一种有序的集合，可以随时添加和删除其中的元素。

比如，列出班里所有同学的名字，就可以用一个 list 表示：

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

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

```
>>> len(classmates)
3
```

用索引来访问 list 中每一个位置的元素，记得索引是从 0 开始的：

```
>>> classmates[0]
'Michael'
>>> classmates[1]
'Bob'
>>> classmates[2]
'Tracy'
>>> classmates[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

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

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

```
>>> classmates[-1]
'Tracy'
```

以此类推，可以获取倒数第 2 个、倒数第 3 个：

```
>>> classmates[-2]
'Bob'
>>> classmates[-3]
'Michael'
```

```
>>> classmates[-4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

当然，倒数第 4 个就越界了。

list 是一个可变的有序表，所以，可以往 list 中追加元素到末尾：

```
>>> classmates.append('Adam')
>>> classmates
['Michael', 'Bob', 'Tracy', 'Adam']
```

也可以把元素插入到指定的位置，比如索引号为 1 的位置：

```
>>> classmates.insert(1, 'Jack')
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']
```

要删除 list 末尾的元素，用 pop()方法：

```
>>> classmates.pop()
'Adam'
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy']
```

要删除指定位置的元素，用 pop(i)方法，其中 i 是索引位置：

```
>>> classmates.pop(1)
'Jack'
>>> classmates
['Michael', 'Bob', 'Tracy']
```

要把某个元素替换成别的元素，可以直接赋值给对应的索引位置：

```
>>> classmates[1] = 'Sarah'
>>> classmates
['Michael', 'Sarah', 'Tracy']
```

list 里面的元素的数据类型也可以不同，比如：

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

list 元素也可以是另一个 list，比如：

```
>>> s = ['python', 'java', ['asp', 'php'], 'scheme']
>>> len(s)
4
```

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

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

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

如果一个 `list` 中一个元素也没有，就是一个空的 `list`，它的长度为 0：

```
>>> L = []
>>> len(L)
0
```

## tuple

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

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

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

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

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

```
>>> t = (1, 2)
>>> t
(1, 2)
```

如果要定义一个空的 `tuple`，可以写成`()`：

```
>>> t = ()
>>> t
()
```

但是，要定义一个只有 1 个元素的 `tuple`，如果你这么定义：

```
>>> t = (1)
>>> t
1
```

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

所以，只有 1 个元素的 `tuple` 定义时必须加一个逗号，来消除歧义：

```
>>> t = (1,)
>>> t
(1,)
```

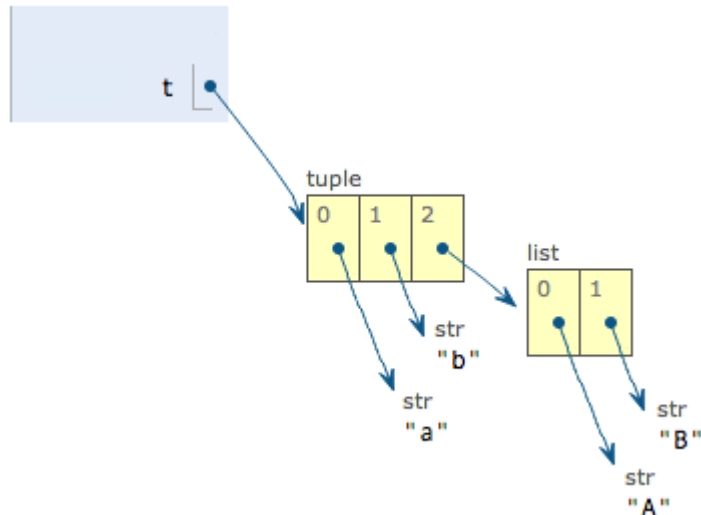
Python 在显示只有 1 个元素的 `tuple` 时，也会加一个逗号，以免你误解成数学计算意义上的括号。

最后来看一个“可变的”`tuple`：

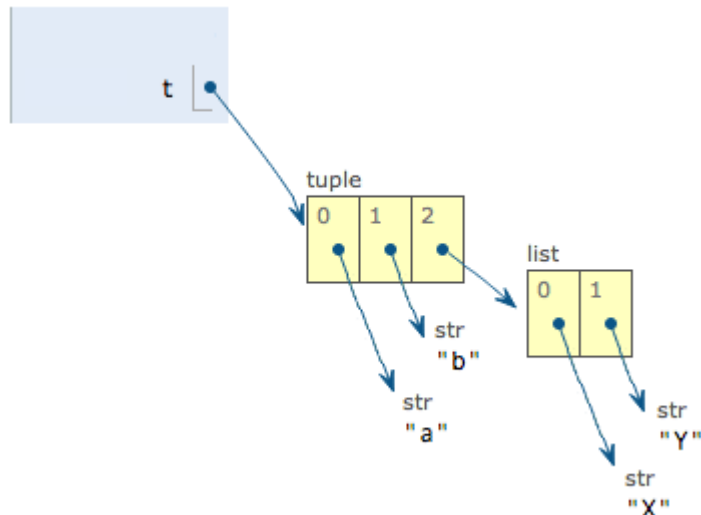
```
>>> t = ('a', 'b', ['A', 'B'])
>>> t[2][0] = 'X'
>>> t[2][1] = 'Y'
>>> t
('a', 'b', ['X', 'Y'])
```

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

别急，我们先看看定义的时候 tuple 包含的 3 个元素：



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



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

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

小结

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

## 条件判断和循环

### 条件判断

计算机之所以能做很多自动化的任务，因为它可以自己做条件判断。

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

```
age = 20
if age >= 18:
    print 'your age is', age
    print 'adult'
```

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

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

```
age = 3
if age >= 18:
    print 'your age is', age
    print 'adult'
else:
    print 'your age is', age
    print 'teenager'
```

注意不要少写了冒号。

当然上面的判断是很粗略的，完全可以用 elif 做更细致的判断：

```
age = 3
if age >= 18:
    print 'adult'
elif age >= 6:
    print 'teenager'
else:
    print 'kid'
```

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

```
if <条件判断 1>:
    <执行 1>
elif <条件判断 2>:
    <执行 2>
elif <条件判断 3>:
    <执行 3>
else:
    <执行 4>
```

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

```
age = 20
if age >= 6:
    print 'teenager'
elif age >= 18:
    print 'adult'
else:
    print 'kid'
```

if 判断条件还可以简写，比如写：

```
if x:
    print 'True'
```

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

循环

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

```
names = ['Michael', 'Bob', 'Tracy']
for name in names:
    print name
```

执行这段代码，会依次打印 `names` 的每一个元素：

```
Michael
Bob
Tracy
```

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

再比如我们想计算 1-10 的整数之和，可以用一个 `sum` 变量做累加：

```
sum = 0
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    sum = sum + x
print sum
```

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

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

`range(101)` 就可以生成 0-100 的整数序列，计算如下：

```
sum = 0
for x in range(101):
    sum = sum + x
```

```
print sum
```

请自行运行上述代码，看看结果是不是当年高斯同学心算出的 5050。

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

```
sum = 0
n = 99
while n > 0:
    sum = sum + n
    n = n - 2
print sum
```

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

小结

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

```
if salary >= 10000:
```

```
    print
```



```
elif salary >= 5000:
```

```
    print
```



```
else:
```

```
    print
```



循环是让计算机做重复任务的有效的方法，有些时候，如果代码写得有问题，会让程序陷入“死循环”，也就是永远循环下去。这时可以用 `Ctrl+C` 退出程序，或者强制结束 Python 进程。请试写一个死循环程序。

## 使用 dict 和 set

dict

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

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

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

```
scores = [95, 75, 85]
```



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

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

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

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

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

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

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

把数据放入 `dict` 的方法，除了初始化时指定外，还可以通过 `key` 放入：

```
>>> d['Adam'] = 67
>>> d['Adam']
67
```

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

```
>>> d['Jack'] = 90
>>> d['Jack']
90
>>> d['Jack'] = 88
>>> d['Jack']
88
```

如果 `key` 不存在，`dict` 就会报错：

```
>>> d['Thomas']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Thomas'
```

要避免 `key` 不存在的错误，有两种办法，一是通过 `in` 判断 `key` 是否存在：

```
>>> 'Thomas' in d
False
```

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

```
>>> d.get('Thomas')
```

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

注意：返回 None 的时候 Python 的交互式命令行不显示结果。

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

```
>>> d.pop('Bob')
75
>>> d
{'Michael': 95, 'Tracy': 85}
```

请务必注意，dict 内部存放的顺序和 key 放入的顺序是没有关系的。

和 list 比较，dict 有以下几个特点：

查找和插入的速度极快，不会随着 key 的增加而增加；

需要占用大量的内存，内存浪费多。

而 list 相反：

查找和插入的时间随着元素的增加而增加；

占用空间小，浪费内存很少。

所以，dict 是用空间来换取时间的一种方法。

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

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

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

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

### set

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

要创建一个 set，需要提供一个 list 作为输入集合：

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

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

重复元素在 set 中自动被过滤：

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

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

```
>>> s.add(4)
>>> s
set([1, 2, 3, 4])
>>> s.add(4)
>>> s
set([1, 2, 3, 4])
```

通过 `remove(key)`方法可以删除元素：

```
>>> s.remove(4)
>>> s
set([1, 2, 3])
```

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

```
>>> s1 = set([1, 2, 3])
>>> s2 = set([2, 3, 4])
>>> s1 & s2
set([2, 3])
>>> s1 | s2
set([1, 2, 3, 4])
```

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

再议不可变对象

上面我们讲了，`str` 是不变对象，而 `list` 是可变对象。

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

```
>>> a = ['c', 'b', 'a']
>>> a.sort()
>>> a
['a', 'b', 'c']
```

而对于不可变对象，比如 `str`，对 `str` 进行操作呢：

```
>>> a = 'abc'
>>> a.replace('a', 'A')
'Abc'
>>> a
'abc'
```

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

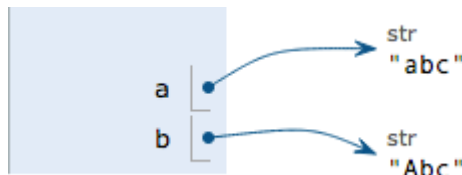
我们先把代码改成下面这样：

```
>>> a = 'abc'
>>> b = a.replace('a', 'A')
>>> b
'Abc'
>>> a
'abc'
```

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



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



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

小结

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

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

## 函数

我们知道圆的面积计算公式为：

$$S = \pi r^2$$

当我们知道半径 `r` 的值时，就可以根据公式计算出面积。假设我们需要计算 3 个不同大小的圆的面积：

```
r1 = 12.34
r2 = 9.08
r3 = 73.1
s1 = 3.14 * r1 * r1
s2 = 3.14 * r2 * r2
s3 = 3.14 * r3 * r3
```

当代码出现有规律的重复的时候，你就需要当心了，每次写 `3.14 * x * x` 不仅很麻烦，而且，如果要把 `3.14` 改成 `3.14159265359` 的时候，得全部替换。

有了函数，我们就不再每次写  $s = 3.14 * x * x$ ，而是写成更有意义的函数调用  $s = \text{area\_of\_circle}(x)$ ，而函数 `area_of_circle` 本身只需要写一次，就可以多次调用。

基本上所有的高级语言都支持函数，Python 也不例外。Python 不但能非常灵活地定义函数，而且本身内置了很多有用的函数，可以直接调用。

抽象

抽象是数学中非常常见的概念。举个例子：

计算数列的和，比如： $1 + 2 + 3 + \dots + 100$ ，写起来十分不方便，于是数学家发明了求和符号  $\Sigma$ ，可以把  $1 + 2 + 3 + \dots + 100$  记作：

100

$\Sigma n$

$n=1$

这种抽象记法非常强大，因为我们看到  $\Sigma$  就可以理解成求和，而不是还原成低级的加法运算。

而且，这种抽象记法是可扩展的，比如：

100

$\Sigma(n^2+1)$

$n=1$

还原成加法运算就变成了：

$(1 \times 1 + 1) + (2 \times 2 + 1) + (3 \times 3 + 1) + \dots + (100 \times 100 + 1)$

可见，借助抽象，我们才能不关心底层的具体计算过程，而直接在更高的层次上思考问题。

写计算机程序也是一样，函数就是最基本的一种代码抽象的方式。

## 调用函数

Python 内置了很多有用的函数，我们可以直接调用。

要调用一个函数，需要知道函数的名称和参数，比如求绝对值的函数 `abs`，只有一个参数。

可以直接从 Python 的官方网站查看文档：

<http://docs.python.org/2/library/functions.html#abs>

也可以在交互式命令行通过 `help(abs)` 查看 `abs` 函数的帮助信息。

调用 `abs` 函数：

```
>>> abs(100)
```

```
100
```

```
>>> abs(-20)
```

```
20
```

```
>>> abs(12.34)
```

```
12.34
```

调用函数的时候，如果传入的参数数量不对，会报 `TypeError` 的错误，并且 Python 会明确地告诉你：`abs()` 有且仅有 1 个参数，但给出了两个：

```
>>> abs(1, 2)
```

```
Traceback (most recent call last):
```

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

```
TypeError: abs() takes exactly one argument (2 given)
```

如果传入的参数数量是对的，但参数类型不能被函数所接受，也会报 `TypeError` 的错误，并且给出错误信息：`str` 是错误的参数类型：

```
>>> abs('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```

而比较函数 `cmp(x, y)` 就需要两个参数，如果  $x < y$ ，返回 -1，如果  $x == y$ ，返回 0，如果  $x > y$ ，返回 1：

```
>>> cmp(1, 2)
-1
>>> cmp(2, 1)
1
>>> cmp(3, 3)
0
```

#### 数据类型转换

Python 内置的常用函数还包括数据类型转换函数，比如 `int()` 函数可以把其他数据类型转换为整数：

```
>>> int('123')
123
>>> int(12.34)
12
>>> float('12.34')
12.34
>>> str(1.23)
'1.23'
>>> unicode(100)
u'100'
>>> bool(1)
True
>>> bool("")
False
```

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

```
>>> a = abs # 变量 a 指向 abs 函数
>>> a(-1) # 所以也可以通过 a 调用 abs 函数
1
```

#### 小结

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

## 定义函数

在 Python 中，定义一个函数要使用 `def` 语句，依次写出函数名、括号、括号中的参数和冒号`:`，然后，在缩进块中编写函数体，函数的返回值用 `return` 语句返回。

我们以自定义一个求绝对值的 `my_abs` 函数为例：

```
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

请自行测试并调用 `my_abs` 看看返回结果是否正确。

请注意，函数体内部的语句在执行时，一旦执行到 `return` 时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

如果没有 `return` 语句，函数执行完毕后也会返回结果，只是结果为 `None`。

`return None` 可以简写为 `return`。

### 空函数

如果想定义一个什么事也不做的空函数，可以用 `pass` 语句：

```
def nop():
    pass
```

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

`pass` 还可以用在其他语句里，比如：

```
if age >= 18:
    pass
```

缺少了 `pass`，代码运行就会有语法错误。

### 参数检查

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

```
>>> my_abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: my_abs() takes exactly 1 argument (2 given)
```

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

```
>>> my_abs('A')
'A'
>>> abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```



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

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

```
def my_abs(x):
    if not isinstance(x, (int, float)):
        raise TypeError('bad operand type')
    if x >= 0:
        return x
    else:
        return -x
```

添加了参数检查后，如果传入错误的参数类型，函数就可以抛出一个错误：

```
>>> my_abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in my_abs
TypeError: bad operand type
```

错误和异常处理将在后续讲到。

返回多个值

函数可以返回多个值吗？答案是肯定的。

比如在游戏中经常需要从一个点移动到另一个点，给出坐标、位移和角度，就可以计算出新的新的坐标：

```
import math
```

```
def move(x, y, step, angle=0):
    nx = x + step * math.cos(angle)
    ny = y - step * math.sin(angle)
    return nx, ny
```

这样我们就可以同时获得返回值：

```
>>> x, y = move(100, 100, 60, math.pi / 6)
>>> print x, y
151.961524227 70.0
```

但其实这只是一种假象，Python 函数返回的仍然是单一值：

```
>>> r = move(100, 100, 60, math.pi / 6)
>>> print r
(151.96152422706632, 70.0)
```

原来返回值是一个 `tuple`！但是，在语法上，返回一个 `tuple` 可以省略括号，而多个变量可以同时接收一个 `tuple`，按位置赋给对应的值，所以，Python 的函数返回多值其实就是返回一个 `tuple`，但写起来更方便。

小结

定义函数时，需要确定函数名和参数个数；

如果有必要，可以先对参数的数据类型做检查；

函数体内部可以用 `return` 随时返回函数结果；

函数执行完毕也没有 `return` 语句时，自动 `return None`。

函数可以同时返回多个值，但其实就是一个 `tuple`。

## 函数的参数

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

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

默认参数

我们仍以具体的例子来说明如何定义函数的默认参数。先写一个计算  $x^2$  的函数：

```
def power(x):  
    return x * x
```

当我们调用 `power` 函数时，必须传入有且仅有的一个参数  $x$ ：

```
>>> power(5)  
25  
>>> power(15)  
225
```

现在，如果我们要计算  $x^3$  怎么办？可以再定义一个 `power3` 函数，但是如果我们要计算  $x^4$ 、 $x^5$ ……怎么办？我们不可能定义无限多个函数。

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

```
def power(x, n):  
    s = 1  
    while n > 0:  
        n = n - 1  
        s = s * x  
    return s
```

对于这个修改后的 `power` 函数，可以计算任意  $n$  次方：

```
>>> power(5, 2)  
25  
>>> power(5, 3)  
125
```

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

```
>>> power(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() takes exactly 2 arguments (1 given)
```

这个时候，默认参数就排上用场了。由于我们经常计算  $x^2$ ，所以，完全可以把第二个参数  $n$  的默认值设定为 2：

```
def power(x, n=2):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

这样，当我们调用 `power(5)` 时，相当于调用 `power(5, 2)`：

```
>>> power(5)
25
>>> power(5, 2)
25
```

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

从上面的例子可以看出，默认参数可以简化函数的调用。设置默认参数时，有几点要注意：一是必选参数在前，默认参数在后，否则 Python 的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；

二是如何设置默认参数。

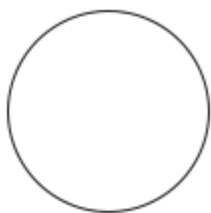
当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

使用默认参数有什么好处？最大的好处是能降低调用函数的难度。

举个例子，假设要在屏幕上画一个圆，由于画圆只需要确定圆心坐标  $(x, y)$  和半径  $r$  就可以了，所以，我们假设可以这样调用该函数：

```
>>> x, y = 0, 0
>>> r = 20
>>> draw_circle(x, y, r)
```

画出来的图形是这样：



但是，如果想画一个红色的圆怎么办？幸好，编写 `draw_circle` 函数的是一位资深 Python 开发者，除了  $x, y, r$  这 3 个必选参数外，该函数还提供默认参数 `linecolor=0x000000`，`fillcolor=0xffffffff`，`penwidth=1`，如果改变默认参数，我们不但能画出红色的圆，还可以控制

圆的线条粗细和填充颜色:

```
>>> draw_circle(0, 0, 20, linecolor=0xff0000)
>>> draw_circle(0, 0, 20, linecolor=0xff0000, penwidth=5)
>>> draw_circle(0, 0, 20, linecolor=0xff0000, fillcolor=0xffff00, penwidth=5)
```

结果如下:



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

有多个默认参数时, 调用的时候, 既可以按顺序提供默认参数, 比如调用 `draw_circle(0, 0, 20, 0xff0000, 0xffff00)`, 意思是, 除了 `0, 0, 20` 这 3 个必选参数外, 后两个参数应用在参数 `linecolor` 和 `fillcolor` 上, `penwidth` 参数由于没有提供, 仍然使用默认值。

也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时, 需要把参数名写上。比如调用 `draw_circle(0, 0, 20, fillcolor=0x00ff00)`, 意思是, `fillcolor` 参数用传进去的值 `0x00ff00`, 其他默认参数继续使用默认值。

默认参数很有用, 但使用不当, 也会掉坑里。默认参数有个最大的坑, 演示如下:

先定义一个函数, 传入一个 `list`, 添加一个 `END` 再返回:

```
def add_end(L=[]):
    L.append('END')
    return L
```

当你正常调用时, 结果似乎不错:

```
>>> add_end([1, 2, 3])
[1, 2, 3, 'END']
>>> add_end(['x', 'y', 'z'])
['x', 'y', 'z', 'END']
```

当你使用默认参数调用时, 一开始结果也是对的:

```
>>> add_end()
['END']
```

但是, 再次调用 `add_end()` 时, 结果就不对了:

```
>>> add_end()
['END', 'END']
>>> add_end()
['END', 'END', 'END']
```

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

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

所以，定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，我们可以用 None 这个不变对象来实现：

```
def add_end(L=None):
    if L is None:
        L = []
    L.append('END')
    return L
```

现在，无论调用多少次，都不会有问题：

```
>>> add_end()
['END']
>>> add_end()
['END']
```

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

可变参数

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

我们以数学题为例，给定一组数字 a, b, c.....，请计算  $a^2 + b^2 + c^2 + \dots$ 。

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

```
def calc(numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

但是调用的时候，需要先组装出一个 list 或 tuple：

```
>>> calc([1, 2, 3])
14
>>> calc((1, 3, 5, 7))
84
```

如果利用可变参数，调用函数的方式可以简化成这样：

```
>>> calc(1, 2, 3)
14
>>> calc(1, 3, 5, 7)
84
```

所以，我们把函数的参数改为可变参数：

```
def calc(*numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

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

```
>>> calc(1, 2)
5
>>> calc()
0
```

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

```
>>> nums = [1, 2, 3]
>>> calc(nums[0], nums[1], nums[2])
14
```

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

```
>>> nums = [1, 2, 3]
>>> calc(*nums)
14
```

这种写法相当有用，而且很常见。

关键字参数

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

```
def person(name, age, **kw):
    print 'name:', name, 'age:', age, 'other:', kw
```

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

```
>>> person('Michael', 30)
name: Michael age: 30 other: {}
```

也可以传入任意个数的关键字参数：

```
>>> person('Bob', 35, city='Beijing')
name: Bob age: 35 other: {'city': 'Beijing'}
>>> person('Adam', 45, gender='M', job='Engineer')
```

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

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

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

```
>>> kw = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, city=kw['city'], job=kw['job'])
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

当然，上面复杂的调用可以用简化的写法：

```
>>> kw = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, **kw)
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

### 参数组合

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

比如定义一个函数，包含上述 4 种参数：

```
def func(a, b, c=0, *args, **kw):
    print 'a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw
```

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

```
>>> func(1, 2)
a = 1 b = 2 c = 0 args = () kw = {}
>>> func(1, 2, c=3)
a = 1 b = 2 c = 3 args = () kw = {}
>>> func(1, 2, 3, 'a', 'b')
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
>>> func(1, 2, 3, 'a', 'b', x=99)
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
```

最神奇的是通过一个 `tuple` 和 `dict`，你也可以调用该函数：

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

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

### 小结

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

参数。

默认参数一定要用不可变对象，如果是可变对象，运行会有逻辑错误！

要注意定义可变参数和关键字参数的语法：

**\*args** 是可变参数，**args** 接收的是一个 **tuple**；

**\*\*kw** 是关键字参数，**kw** 接收的是一个 **dict**。

以及调用函数时如何传入可变参数和关键字参数的语法：

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

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

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

## 递归函数

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

举个例子，我们来计算阶乘  $n! = 1 \times 2 \times 3 \times \dots \times n$ ，用函数 **fact(n)** 表示，可以看出：

$\text{fact}(n) = n! = 1 \times 2 \times 3 \times \dots \times (n-1) \times n = (n-1)! \times n = \text{fact}(n-1) \times n$

所以，**fact(n)** 可以表示为  $n \times \text{fact}(n-1)$ ，只有  $n=1$  时需要特殊处理。

于是，**fact(n)** 用递归的方式写出来就是：

```
def fact(n):
    if n==1:
        return 1
    return n * fact(n - 1)
```

上面就是一个递归函数。可以试试：

```
>>> fact(1)
1
>>> fact(5)
120
>>> fact(100)
9332621544394415268169923885626670049071596826438162146859296389521759999322991
560894146397615651828625369792082722375825118521091686400000000000000000000000
L
```

如果我们计算 **fact(5)**，可以根据函数定义看到计算过程如下：

```
==> fact(5)
==> 5 * fact(4)
==> 5 * (4 * fact(3))
==> 5 * (4 * (3 * fact(2)))
==> 5 * (4 * (3 * (2 * fact(1))))
==> 5 * (4 * (3 * (2 * 1)))
==> 5 * (4 * (3 * 2))
```



```
====> 5 * (4 * 6)
====> 5 * 24
====> 120
```

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（**stack**）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。可以试试 `fact(1000)`:

```
>>> fact(1000)
```

```
Traceback (most recent call last):
```

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

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

```
...
```

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

```
RuntimeError: maximum recursion depth exceeded
```

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

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

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

```
def fact(n):
    return fact_iter(1, 1, n)

def fact_iter(product, count, max):
    if count > max:
        return product
    return fact_iter(product * count, count + 1, max)
```

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

`fact(5)` 对应的 `fact_iter(1, 1, 5)` 的调用如下：

```
====> fact_iter(1, 1, 5)
====> fact_iter(1, 2, 5)
====> fact_iter(2, 3, 5)
====> fact_iter(6, 4, 5)
====> fact_iter(24, 5, 5)
====> fact_iter(120, 6, 5)
====> 120
```

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



环语句的编程语言只能通过尾递归实现循环。

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

## 高级特性

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

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

```
L = []
n = 1
while n <= 99:
    L.append(n)
    n = n + 2
```

取 list 的前一半的元素，也可以通过循环实现。

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

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

## 切片

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

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

取前 3 个元素，应该怎么做？

笨办法：

```
>>> [L[0], L[1], L[2]]
['Michael', 'Sarah', 'Tracy']
```

之所以是笨办法是因为扩展一下，取前 N 个元素就没辙了。

取前 N 个元素，也就是索引为 0-(N-1)的元素，可以用循环：

```
>>> r = []
>>> n = 3
>>> for i in range(n):
...     r.append(L[i])
...
>>> r
['Michael', 'Sarah', 'Tracy']
```

对这种经常取指定索引范围的操作，用循环十分繁琐，因此，Python 提供了切片（Slice）操作符，能大大简化这种操作。

对应上面的问题，取前 3 个元素，用一行代码就可以完成切片：

```
>>> L[0:3]
['Michael', 'Sarah', 'Tracy']
```

L[0:3]表示，从索引 0 开始取，直到索引 3 为止，但不包括索引 3。即索引 0，1，2，正好是 3 个元素。

如果第一个索引是 0，还可以省略：

```
>>> L[:3]
['Michael', 'Sarah', 'Tracy']
```

也可以从索引 1 开始，取出 2 个元素出来：

```
>>> L[1:3]
['Sarah', 'Tracy']
```

类似的，既然 Python 支持 L[-1]取倒数第一个元素，那么它同样支持倒数切片，试试：

```
>>> L[-2:]
['Bob', 'Jack']
>>> L[-2:-1]
['Bob']
```

记住倒数最后一个元素的索引是-1。

切片操作十分有用。我们先创建一个 0-99 的数列：

```
>>> L = range(100)
>>> L
[0, 1, 2, 3, ..., 99]
```

可以通过切片轻松取出某一段数列。比如前 10 个数：

```
>>> L[:10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

后 10 个数：

```
>>> L[-10:]
[90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

前 11-20 个数：

```
>>> L[10:20]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

前 10 个数，每两个取一个：

```
>>> L[:10:2]
[0, 2, 4, 6, 8]
```

所有数，每 5 个取一个：

```
>>> L[::5]
```

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

甚至什么都不写，只写[:]就可以原样复制一个 list:

```
>>> L[:]
[0, 1, 2, 3, ..., 99]
```

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

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

字符串'xxx'或 Unicode 字符串 u'xxx'也可以看成是一种 list，每个元素就是一个字符。因此，字符串也可以用切片操作，只是操作结果仍是字符串:

```
>>> 'ABCDEFGH'[:3]
'ABC'
>>> 'ABCDEFGH'[:2]
'ACEG'
```

在很多编程语言中，针对字符串提供了很多各种截取函数，其实目的就是对字符串切片。Python 没有针对字符串的截取函数，只需要切片一个操作就可以完成，非常简单。

小结

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

## 迭代

### 重点:

1. 因为 dict 的存储不是按照 list 的方式顺序排列，所以，迭代出的结果顺序很可能不一样。
2. 默认情况下，dict 迭代的是 key。如果要迭代 value，可以用 for value in d.itervalues()，如果要同时迭代 key 和 value，可以用 for k, v in d.iteritems()。
3. Python 内置的 enumerate 函数可以把一个 list 变成索引-元素对，这样就可以在 for 循环中同时迭代索引和元素本身

如果给定一个 list 或 tuple，我们可以通过 for 循环来遍历这个 list 或 tuple，这种遍历我们成为迭代 (Iteration)。

在 Python 中，迭代是通过 for ... in 来完成的，而很多语言比如 C 或者 Java，迭代 list 是通过下标完成的，比如 Java 代码:

```
for (i=0; i<list.length; i++) {
    n = list[i];
}
```

可以看出，Python 的 for 循环抽象程度要高于 Java 的 for 循环，因为 Python 的 for 循环不仅可以作用在 list 或 tuple 上，还可以作用在其他可迭代对象上。

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

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> for key in d:
...     print key
...
a
c
b
```

因为 dict 的存储不是按照 list 的方式顺序排列，所以，迭代出的结果顺序很可能不一样。默认情况下，dict 迭代的是 key。如果要迭代 value，可以用 `for value in d.itervalues()`，如果要同时迭代 key 和 value，可以用 `for k, v in d.iteritems()`。

由于字符串也是可迭代对象，因此，也可以作用于 for 循环：

```
>>> for ch in 'ABC':
...     print ch
...
A
B
C
```

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

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

```
>>> from collections import Iterable
>>> isinstance('abc', Iterable) # str 是否可迭代
True
>>> isinstance([1,2,3], Iterable) # list 是否可迭代
True
>>> isinstance(123, Iterable) # 整数是否可迭代
False
```

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

```
>>> for i, value in enumerate(['A', 'B', 'C']):
...     print i, value
...
0 A
1 B
2 C
```

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

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

```
...     print x, y
...
1 1
2 4
3 9
```

### 小结

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

## 列表生成式

全都是重点！

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

举个例子，要生成 list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] 可以用 range(1, 11):

```
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

但如果要生成 [1x1, 2x2, 3x3, ..., 10x10] 怎么做？方法一是循环：

```
>>> L = []
>>> for x in range(1, 11):
...     L.append(x * x)
...
>>> L
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

但是循环太繁琐，而列表生成式则可以用一行语句代替循环生成上面的 list：

```
>>> [x * x for x in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

写列表生成式时，把要生成的元素  $x * x$  放到前面，后面跟 for 循环，就可以把 list 创建出来，十分有用，多写几次，很快就可以熟悉这种语法。

for 循环后面还可以加上 if 判断，这样我们就可以筛选出仅偶数的平方：

```
>>> [x * x for x in range(1, 11) if x % 2 == 0]
[4, 16, 36, 64, 100]
```

还可以使用两层循环，可以生成全排列：

```
>>> [m + n for m in 'ABC' for n in 'XYZ']
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

三层和三层以上的循环就很少用到了。

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

```
>>> import os # 导入 os 模块，模块的概念后面讲到
>>> [d for d in os.listdir('.')] # os.listdir 可以列出文件和目录
['.emacs.d', '.ssh', '.Trash', 'Adlm', 'Applications', 'Desktop', 'Documents', 'Downloads', 'Library',
'Movies', 'Music', 'Pictures', 'Public', 'VirtualBox VMs', 'Workspace', 'XCode']
```

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

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }
>>> for k, v in d.iteritems():
...     print k, '=', v
...
y = B
x = A
z = C
```

因此，列表生成式也可以使用两个变量来生成 list：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }
>>> [k + '=' + v for k, v in d.iteritems()]
['y=B', 'x=A', 'z=C']
```

最后把一个 list 中所有的字符串变成小写：

```
>>> L = ['Hello', 'World', 'IBM', 'Apple']
>>> [s.lower() for s in L]
['hello', 'world', 'ibm', 'apple']
```

小结

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

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

```
>>> L = ['Hello', 'World', 18, 'Apple', None]
>>> [s.lower() for s in L]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'lower'
```

使用内建的 isinstance 函数可以判断一个变量是不是字符串：

```
>>> x = 'abc'
>>> y = 123
>>> isinstance(x, str)
True
>>> isinstance(y, str)
```



False

请修改列表生成式，通过添加 if 语句保证列表生成式能正确地执行。

## 生成器

**这个功能很高端！都需要看！**

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

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

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

```
>>> L = [x * x for x in range(10)]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> g = (x * x for x in range(10))
>>> g
<generator object <genexpr> at 0x104feab40>
```

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

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

如果要一个一个打印出来，可以通过 generator 的 next() 方法：

```
>>> g.next()
0
>>> g.next()
1
>>> g.next()
4
>>> g.next()
9
>>> g.next()
16
>>> g.next()
25
>>> g.next()
36
>>> g.next()
49
>>> g.next()
64
```

```
>>> g.next()
81
>>> g.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

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

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

```
>>> g = (x * x for x in range(10))
>>> for n in g:
...     print n
...
0
1
4
9
16
25
36
49
64
81
```

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

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

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

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

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

```
def fab(max):
    n, a, b = 0, 0, 1
    while n < max:
        print b
        a, b = b, a + b
        n = n + 1
```

上面的函数可以输出斐波那契数列的前 N 个数：

```
>>> fab(6)
1
```

1  
2  
3  
5  
8

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

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

```
def fab(max):  
    n, a, b = 0, 0, 1  
    while n < max:  
        yield b  
        a, b = b, a + b  
        n = n + 1
```

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

```
>>> fab(6)  
<generator object fab at 0x104feaaa0>
```

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

举个简单的例子，定义一个 generator，依次返回数字 1，3，5：

```
>>> def odd():  
...     print 'step 1'  
...     yield 1  
...     print 'step 2'  
...     yield 3  
...     print 'step 3'  
...     yield 5  
...  
>>> o = odd()  
>>> o.next()  
step 1  
1  
>>> o.next()  
step 2  
3  
>>> o.next()  
step 3  
5
```

```
>>> o.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

可以看到，odd 不是普通函数，而是 generator，在执行过程中，遇到 yield 就中断，下次又继续执行。执行 3 次 yield 后，已经没有 yield 可以执行了，所以，第 4 次调用 next()就报错。回到 fab 的例子，我们在循环过程中不断调用 yield，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

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

```
>>> for n in fab(6):
...     print n
...
1
1
2
3
5
8
```

## 小结

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

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

## 函数式编程

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

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

我们首先要搞明白计算机（Computer）和计算（Compute）的概念。

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

而计算则指数学意义上的计算，越是抽象的计算，离计算机硬件越远。

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

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没

有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

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

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

## 高阶函数

重点：

1 map reduce 函数，高端应用

2 把 map reduce 整合到一起的 `str2int` 函数也是高大上，引出 `lambda` 函数，牛 B

3 利用 `sort()` 高阶函数来自定义排序，核心代码非常简洁

4 高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。这个代码好像不怎么好懂。

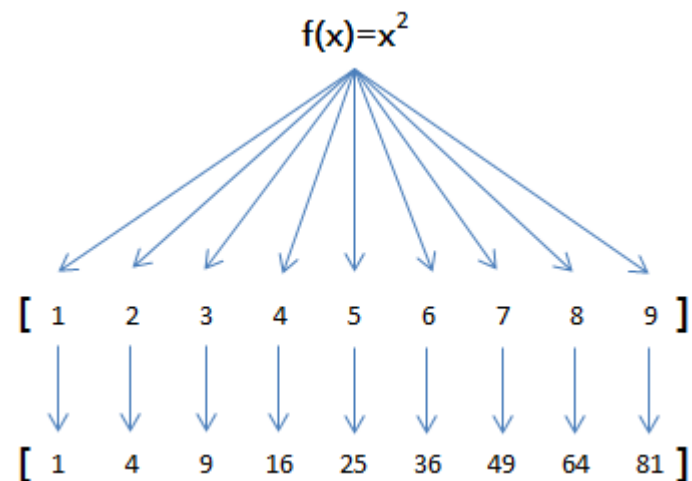
传入函数

要理解“函数本身也可以作为参数传入”，可以从 Python 内建的 map/reduce 函数入手。

如果你读过 Google 的那篇大名鼎鼎的论文“MapReduce: Simplified Data Processing on Large Clusters”，你就能大概明白 map/reduce 的概念。

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

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



现在，我们用 Python 代码实现：

```
>>> def f(x):
...     return x * x
...
>>> map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

请注意我们定义的函数 `f`。当我们写 `f` 时，指的是函数对象本身，当我们写 `f(1)` 时，指的是调用 `f` 函数，并传入参数 `1`，期待返回结果 `1`。

因此，`map()` 传入的第一个参数是 `f`，即函数对象本身。

像 `map()` 函数这种能够接收函数作为参数的函数，称之为高阶函数 (Higher-order function)。

你可能会想，不需要 `map()` 函数，写一个循环，也可以计算出结果：

```
L = []
for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
    L.append(f(n))
print L
```

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

所以，`map()` 作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的 `f(x)=x2`，还可以计算任意复杂的函数。

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

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

比方说对一个序列求和，就可以用 `reduce` 实现：

```
>>> def add(x, y):
...     return x + y
...
>>> reduce(add, [1, 3, 5, 7, 9])
25
```

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

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

```
>>> def fn(x, y):
...     return x * 10 + y
...
>>> reduce(fn, [1, 3, 5, 7, 9])
13579
```

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

```
>>> def fn(x, y):
...     return x * 10 + y
...
>>> reduce(fn, map(int, '13579'))
13579
```

整理成一个 `str2int` 的函数就是：

```
def str2int(s):
    def fn(x, y):
```

```
        return x * 10 + y
    return reduce(fn, map(int, s))
```

还可以用 lambda 函数进一步简化成：

```
def str2int(s):
    return reduce(lambda x,y: x*10+y, map(int, s))
```

也就是说，你可以自己写一个把字符串转化为整数的函数，而且只需要一行代码！  
lambda 函数的用法在下一节介绍。

## 排序算法

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

Python 内置的 sorted() 函数就可以对 list 进行排序：

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

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

```
def reversed_cmp(x, y):
    if x > y:
        return -1
    if x < y:
        return 1
    return 0
```

传入自定义的比较函数 reversed\_cmp，就可以实现倒序排序：

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

我们再看一个字符串排序的例子：

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

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

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

```
def cmp_ignore_case(s1, s2):
    u1 = s1.upper()
    u2 = s2.upper()
```

```
if u1 < u2:
    return -1
if u1 > u2:
    return 1
return 0
```

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

这样，我们给 `sorted` 传入上述比较函数，即可实现忽略大小写的排序：

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

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

## 函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

```
def calc_sum(*args):
    ax = 0
    for n in args:
        ax = ax + n
    return ax
```

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

```
def lazy_sum(*args):
    def sum():
        ax = 0
        for n in args:
            ax = ax + n
        return ax
    return sum
```

当我们调用 `lazy_sum()` 时，返回的并不是求和结果，而是求和函数：

```
>>> f = lazy_sum(1, 3, 5, 7, 9)
>>> f
<function sum at 0x10452f668>
```

调用函数 `f` 时，才真正计算求和的结果：

```
>>> f()
25
```



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

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

```
>>> f1 = lazy_sum(1, 3, 5, 7, 9)
>>> f2 = lazy_sum(1, 3, 5, 7, 9)
>>> f1==f2
False
```

`f1()` 和 `f2()` 的调用结果互不影响。

小结

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

假设 Python 没有提供 `map()` 函数，请自行编写一个 `my_map()` 函数实现与 `map()` 相同的功能。

## 匿名函数

**重点：**

1 匿名函数 `lambda` 有个限制，就是只能有一个表达式，不用写 `return`，返回值就是该表达式的结果。

当我们在传入函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。

在 Python 中，对匿名函数提供了有限支持。还是以 `map()` 函数为例，计算  $f(x)=x^2$  时，除了定义一个  $f(x)$  的函数外，还可以直接传入匿名函数：

```
>>> map(lambda x: x * x, [1, 2, 3, 4, 5, 6, 7, 8, 9])
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

通过对比可以看出，匿名函数 `lambda x: x * x` 实际上就是：

```
def f(x):
    return x * x
```

关键字 `lambda` 表示匿名函数，冒号前面的 `x` 表示函数参数。

匿名函数有个限制，就是只能有一个表达式，不用写 `return`，返回值就是该表达式的结果。

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

```
>>> f = lambda x: x * x
>>> f
<function <lambda> at 0x10453d7d0>
>>> f(5)
25
```

同样，也可以把匿名函数作为返回值返回，比如：

```
def build(x, y):  
    return lambda: x * x + y * y
```

小结

Python 对匿名函数的支持有限，只有一些简单的情况下可以使用匿名函数。

## 装饰器

这个有点高端！需要好好看一下。

但目测这个可以用类来实现这种装饰器模式。

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

```
>>> def now():  
...     print '2013-12-25'  
...  
>>> f = now  
>>> f()  
2013-12-25
```

函数对象有一个 `__name__` 属性，可以拿到函数的名字：

```
>>> now.__name__  
'now'  
>>> f.__name__  
'now'
```

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

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

```
def log(func):  
    def wrapper(*args, **kw):  
        print 'call %s():' % func.__name__  
        return func(*args, **kw)  
    return wrapper
```

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

```
@log  
def now():  
    print '2013-12-25'
```

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

```
>>> now()
```

```
call now():
2013-12-25
```

把@log 放到 now()函数的定义处，相当于执行了语句：

```
now = log(now)
```

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

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

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

```
def log(text):
    def decorator(func):
        def wrapper(*args, **kw):
            print '%s %s():' % (text, func.__name__)
            return func(*args, **kw)
        return wrapper
    return decorator
```

这个 3 层嵌套的 decorator 用法如下：

```
@log('execute')
def now():
    print '2013-12-25'
```

执行结果如下：

```
>>> now()
execute now():
2013-12-25
```

和两层嵌套的 decorator 相比，3 层嵌套的效果是这样的：

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

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

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

```
>>> now.__name__
'wrapper'
```

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

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

```
import functools
```

```
def log(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        print 'call %s():' % func.__name__
        return func(*args, **kw)
    return wrapper
```

或者针对带参数的 decorator：

```
import functools
```

```
def log(text):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            print '%s %s():' % (text, func.__name__)
            return func(*args, **kw)
        return wrapper
    return decorator
```

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

小结

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

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

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

再思考一下能否写出一个 `@log` 的 decorator，使它既支持：

```
@log
def f():
    pass
```

又支持：

```
@log('execute')
def f():
    pass
```

## 偏函数

重点:

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

这个用处多么，总觉的是有点自寻麻烦嘞！

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

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

`int()` 函数可以把字符串转换为整数，当仅传入字符串时，`int()` 函数默认按十进制转换：

```
>>> int('12345')
12345
```

但 `int()` 函数还提供额外的 `base` 参数，默认值为 10。如果传入 `base` 参数，就可以做 N 进制的转换：

```
>>> int('12345', base=8)
5349
>>> int('12345', 16)
74565
```

假设要转换大量的二进制字符串，每次都传入 `int(x, base=2)` 非常麻烦，于是，我们想到，可以定义一个 `int2()` 的函数，默认把 `base=2` 传进去：

```
def int2(x, base=2):
    return int(x, base)
```

这样，我们转换二进制就非常方便了：

```
>>> int2('1000000')
64
>>> int2('1010101')
85
```

`functools.partial` 就是帮助我们创建一个偏函数的，不需要我们自己定义 `int2()`，可以直接使用下面的代码创建一个新的函数 `int2`：

```
>>> import functools
>>> int2 = functools.partial(int, base=2)
>>> int2('1000000')
64
>>> int2('1010101')
85
```

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

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

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

最后，创建偏函数时，要从右到左固定参数，就是说，对于函数 `f(a1, a2, a3)`，可以固定 `a3`，也可以固定 `a3` 和 `a2`，也可以固定 `a3`，`a2` 和 `a1`，但不要跳着固定，比如只固定 `a1` 和 `a3`，把 `a2` 漏下了。如果这样做，调用新的函数会更复杂，可以自己试试。

小结

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

## 模块

重点：

用顶层文件名+`__init__.py` 文件来定义包

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

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

使用模块有什么好处？

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

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

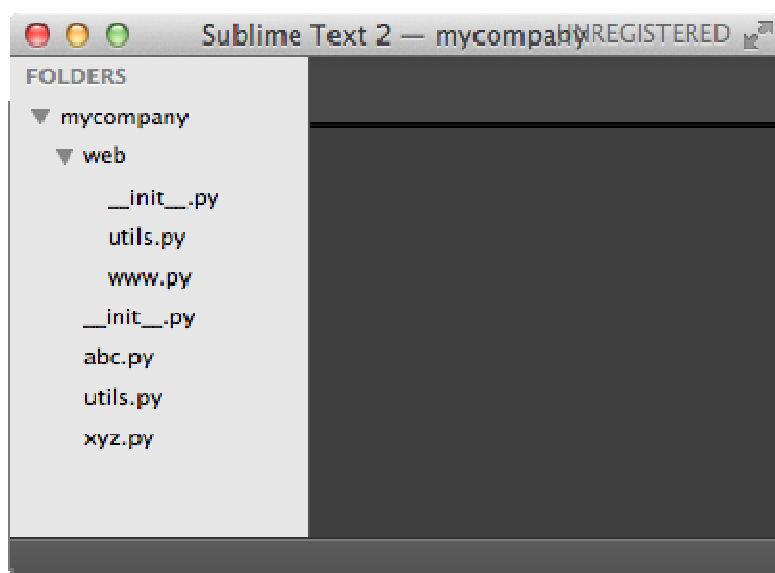
你也许还想到，如果不同的人编写的模块名相同怎么办？为了避免模块名冲突，Python 又引入了按目录来组织模块的方法，称为包（Package）。

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

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



引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。现在，`abc.py` 模块的名字就变成了 `mycompany.abc`，类似的，`xyz.py` 的模块名变成了 `mycompany.xyz`。请注意，每一个包目录下面都会有一个 `__init__.py` 的文件，这个文件是必须存在的，否则，Python 就把这个目录当成普通目录，而不是一个包。`__init__.py` 可以是空文件，也可以有 Python 代码，因为 `__init__.py` 本身就是一个模块，而它的模块名就是 `mycompany`。类似的，可以有多个目录，组成多级层次的包结构。比如如下的目录结构：



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

mycompany.web 也是一个模块，请指出该模块对应的.py 文件。

## 使用模块

重点：

- 1 导入某些包时为防出错，可使用 `try...except ImportError` 来捕获 # 导入失败会捕获到 `ImportError`
- 2 使用类似 `_xxx` 和 `__xxx` 来定义私有(private) 函数或变量

Python 本身就内置了很多非常有用的模块，只要安装完毕，这些模块就可以立刻使用。  
我们以内建的 sys 模块为例，编写一个 hello 的模块：

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

'a test module '

__author__ = 'Michael Liao'

import sys

def test():
    args = sys.argv
    if len(args)==1:
        print 'Hello, world!'
    elif len(args)==2:
        print 'Hello, %s!' % args[1]
    else:
        print 'Too many arguments!'

if __name__=='__main__':
    test()
```

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

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

第 6 行使用 `__author__` 变量把作者写进去，这样当你公开源代码后别人就可以瞻仰你的大名；  
以上就是 Python 模块的标准文件模板，当然也可以全部删掉不写，但是，按标准办事肯定没错。

后面开始就是真正的代码部分。

你可能注意到了，使用 sys 模块的第一步，就是导入该模块：

```
import sys
```



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

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

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

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

最后，注意到这两行代码：

```
if __name__=='__main__':  
    test()
```

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

我们可以用命令行运行 hello.py 看看效果：

```
$ python hello.py  
Hello, world!  
$ python hello.py Michael  
Hello, Michael!
```

如果启动 Python 交互环境，再导入 hello 模块：

```
$ python  
Python 2.7.5 (default, Aug 25 2013, 00:04:04)  
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import hello  
>>>
```

导入时，没有打印 Hello, word!，因为没有执行 test()函数。

调用 hello.test()时，才能打印出 Hello, word!：

```
>>> hello.test()  
Hello, world!
```

别名

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

try:

```
    import cStringIO as StringIO  
except ImportError: # 导入失败会捕获到 ImportError  
    import StringIO
```

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

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

```
try: import json # python >= 2.6 except ImportError: import simplejson as json # python <= 2.5
```

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

作用域

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

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

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

类似 `__xxx` 和 `__xxx__` 这样的函数或变量就是非公开的 (**private**)，不应该被直接引用，比如 `_abc`，`__abc` 等；

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

**private** 函数或变量不应该被别人引用，那它们有什么用呢？请看例子：

```
def _private_1(name):
    return 'Hello, %s' % name

def _private_2(name):
    return 'Hi, %s' % name

def greeting(name):
    if len(name) > 3:
        return _private_1(name)
    else:
        return _private_2(name)
```

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

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

## 安装第三方模块

**重点：**

1 Mac Linux 使用 `setuptools`，windows 使用 `ez_setup.py`

2 添加第三方模块时，添加搜索目录

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

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

如果你正在使用 Windows，请首先从这个地址下载 ez\_setup.py：  
<https://pypi.python.org/pypi/setuptools#windows>

下载后，随便放到一个目录下，然后运行以下命令来安装 setuptools：  
`python ez_setup.py`

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



重新打开命令提示符窗口，就可以运行 `easy_install` 了：

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

耐心等待下载并安装后，就可以使用 PIL 了。

有了 PIL，处理图片易如反掌。随便找个图片生成缩略图：

```
>>> import Image
>>> im = Image.open('test.png')
>>> print im.format, im.size, im.mode
PNG (400, 300) RGB
>>> im.thumbnail((200, 100))
>>> im.save('thumb.jpg', 'JPEG')
```

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

### 模块搜索路径

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

```
>>> import mymodule
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named mymodule
```

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

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

如果我们要添加自己的搜索目录，有两种方法：

一是直接修改 sys.path，添加要搜索的目录：

```
>>> import sys
>>> sys.path.append('/Users/michael/my_py_scripts')
```

这种方法是在运行时修改，运行结束后失效。

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

## 使用\_\_future\_\_

重点：

1 Python 新旧版本不兼容，使用\_\_future\_\_模块，把下一个新版本的特性导入到当前版本

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

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

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

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

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

```
# still running on Python 2.7
```

```
from __future__ import unicode_literals
```

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

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

```
$ python task.py
```

```
'xxx' is unicode? True
```

```
u'xxx' is unicode? True
```

```
'xxx' is str? False
```

```
b'xxx' is str? True
```

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

```
>>> 10 / 3
```

```
3
```

要做精确除法，必须把其中一个数变成浮点数：

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

```
3.3333333333333335
```

而在 Python 3.x 中，所有的除法都是精确除法，地板除用 // 表示：

```
$ python3
```

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

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

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

```
>>> 10 / 3
```

```
3.3333333333333335
```

```
>>> 10 // 3
```

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

```
from __future__ import division
```

```
print '10 / 3 =', 10 / 3  
print '10.0 / 3 =', 10.0 / 3  
print '10 // 3 =', 10 // 3
```

结果如下：

```
10 / 3 = 3.333333333333  
10.0 / 3 = 3.333333333333  
10 // 3 = 3
```

小结

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

## 面向对象编程

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

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

而面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。在 Python 中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

我们以一个例子来说明面向过程和面向对象在程序流程上的不同之处。

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

```
std1 = { 'name': 'Michael', 'score': 98 }  
std2 = { 'name': 'Bob', 'score': 81 }
```

而处理学生成绩可以通过函数实现，比如打印学生的成绩：

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

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是 Student 这种数据类型应该被视为一个对象，这个对象拥有 name 和 score 这两个属性（Property）。如

果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后，给对象发一个 `print_score` 消息，让对象自己把自己的数据打印出来。

```
class Student(object):
```

```
    def __init__(self, name, score):
        self.name = name
        self.score = score

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

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

```
bart = Student('Bart Simpson', 98)
lisa = Student('Lisa Simpson', 77)
bart.print_score()
lisa.print_score()
```

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

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

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

小结

数据封装、继承和多态是面向对象的三大特点，我们后面会详细讲解。

## 类和实例

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

仍以 Student 类为例，在 Python 中，定义类是通过 `class` 关键字：

```
class Student(object):
    pass
```

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

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

```
>>> bart = Student()
>>> bart
```

```
<__main__.Student object at 0x10a67a590>
>>> Student
<class '__main__.Student'>
```

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

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

```
>>> bart.name = 'Bart Simpson'
>>> bart.name
'Bart Simpson'
```

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

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score
```

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

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

```
>>> bart = Student('Bart Simpson', 98)
>>> bart.name
'Bart Simpson'
>>> bart.score
98
```

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

数据封装

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

```
>>> def print_score(std):
...     print '%s: %s' % (std.name, std.score)
...
>>> print_score(bart)
Bart Simpson: 98
```

但是，既然 `Student` 实例本身就拥有这些数据，要访问这些数据，就没有必要从外面的函数去访问，可以直接在 `Student` 类的内部定义访问数据的函数，这样，就把“数据”给封装起来



了。这些封装数据的函数是和 `Student` 类本身是关联起来的，我们称之为类的方法：

```
class Student(object):
```

```
    def __init__(self, name, score):
        self.name = name
        self.score = score

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

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

```
>>> bart.print_score()
```

```
Bart Simpson: 98
```

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

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

```
class Student(object):
```

```
    ...
```

```
    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'
```

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

```
>>> bart.get_grade()
```

```
'A'
```

## 小结

类是创建实例的模板，而实例则是一个一个具体的对象，各个实例拥有的数据都不相同；通过在实例变量上调用方法，我们就直接操作了对象内部的数据，但无需知道方法内部的实现细节。

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

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

```
>>> lisa = Student('Lisa Simpson', 77)
```

```
>>> bart.age = 8
```

```
>>> bart.age
```

8

```
>>> lisa.age
```

Traceback (most recent call last):

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

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

## 访问限制

### 重点：

- 1 如果要想内部属性不被外部访问，可以把属性的名称前加上两个下划线\_\_
- 2 使用 get set 方法而不是直接用 直接用 public 的原因： bart.score = 59 也可以修改啊，为什么要定义一个方法大费周折？因为在方法中，可以对参数做检查，避免传入无效的参数：
- 3 变量名类似\_\_xxx\_\_的是特殊变量，特殊变量是可以直接访问的，不是 private 变量，所以，不能用\_\_name\_\_、\_\_score\_\_这样的变量名。
- 4 使用\_\_name\_\_的变量含义是：虽然我可以被访问，但是，请把我视为私有变量，不要随意访问（约定俗成）
- 5 \_\_name\_\_变量也可以访问，通过\_\_Student\_\_name，但请不要这么干!!!

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

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

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

```
>>> bart.score
```

```
98
```

```
>>> bart.score = 59
```

```
>>> bart.score
```

```
59
```

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

```
class Student(object):
```

```
    def __init__(self, name, score):
```

```
        self.__name = name
```

```
        self.__score = score
```

```
    def print_score(self):
```

```
        print '%s: %s' % (self.__name, self.__score)
```

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

例变量.\_\_score 了：

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

```
>>> bart.__name
```

Traceback (most recent call last):

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

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

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

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

```
class Student(object):
```

```
...
```

```
    def get_name(self):
```

```
        return self.__name
```

```
    def get_score(self):
```

```
        return self.__score
```

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

```
class Student(object):
```

```
...
```

```
    def set_score(self, score):
```

```
        self.__score = score
```

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

```
class Student(object):
```

```
...
```

```
    def set_score(self, score):
```

```
        if 0 <= score <= 100:
```

```
            self.__score = score
```

```
        else:
```

```
            raise ValueError('bad score')
```

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

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

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

```
>>> bart._Student__name
'Bart Simpson'
```

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

总的来说就是，Python 本身没有任何机制阻止你干坏事，一切全靠自觉。

## 继承和多态

### 重点：

1 面向对象的开闭原则，

对扩展开放：允许新增 `Animal` 子类；

对修改封闭：不需要修改依赖 `Animal` 类型的 `run_twice()` 等函数。

在 OOP 程序设计中，当我们定义一个 `class` 的时候，可以从某个现有的 `class` 继承，新的 `class` 称为子类（Subclass），而被继承的 `class` 称为基类、父类或超类（Base class、Super class）。比如，我们已经编写了一个名为 `Animal` 的 `class`，有一个 `run()` 方法可以直接打印：

```
class Animal(object):
    def run(self):
        print 'Animal is running...'
```

当我们需要编写 `Dog` 和 `Cat` 类时，就可以直接从 `Animal` 类继承：

```
class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

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

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

```
dog = Dog()
dog.run()
```

```
cat = Cat()
cat.run()
```

运行结果如下：

```
Animal is running...
```

Animal is running...

当然，也可以对子类增加一些方法，比如 Dog 类：

```
class Dog(Animal):
    def run(self):
        print 'Dog is running...'
    def eat(self):
        print 'Eating meat...'
```

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

```
class Dog(Animal):
    def run(self):
        print 'Dog is running...'

class Cat(Animal):
    def run(self):
        print 'Cat is running...'
```

再次运行，结果如下：

```
Dog is running...
Cat is running...
```

当子类 and 父类都存在相同的 run() 方法时，我们说，子类的 run() 覆盖了父类的 run()，在代码运行的时候，总是会调用子类的 run()。这样，我们就获得了继承的另一个好处：多态。要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个 class 的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和 Python 自带的数据类型，比如 str、list、dict 没什么两样：

```
a = list() # a 是 list 类型
b = Animal() # b 是 Animal 类型
c = Dog() # c 是 Dog 类型
```

判断一个变量是否是某个类型可以用 isinstance() 判断：

```
>>> isinstance(a, list)
True
>>> isinstance(b, Animal)
True
>>> isinstance(c, Dog)
True
```

看来 a、b、c 确实对应着 list、Animal、Dog 这 3 种类型。

但是等等，试试：

```
>>> isinstance(c, Animal)
```

True

看来 c 不仅仅是 Dog, c 还是 Animal!

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

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

```
>>> b = Animal()
>>> isinstance(b, Dog)
False
```

Dog 可以看成 Animal, 但 Animal 不可以看成 Dog。

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

```
def run_twice(animal):
    animal.run()
    animal.run()
```

当我们传入 Animal 的实例时, run\_twice()就打印出:

```
>>> run_twice(Animal())
Animal is running...
Animal is running...
```

当我们传入 Dog 的实例时, run\_twice()就打印出:

```
>>> run_twice(Dog())
Dog is running...
Dog is running...
```

当我们传入 Cat 的实例时, run\_twice()就打印出:

```
>>> run_twice(Cat())
Cat is running...
Cat is running...
```

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

```
class Tortoise(Animal):
    def run(self):
        print 'Tortoise is running slowly...'
```

当我们调用 run\_twice()时, 传入 Tortoise 的实例:

```
>>> run_twice(Tortoise())
Tortoise is running slowly...
Tortoise is running slowly...
```

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

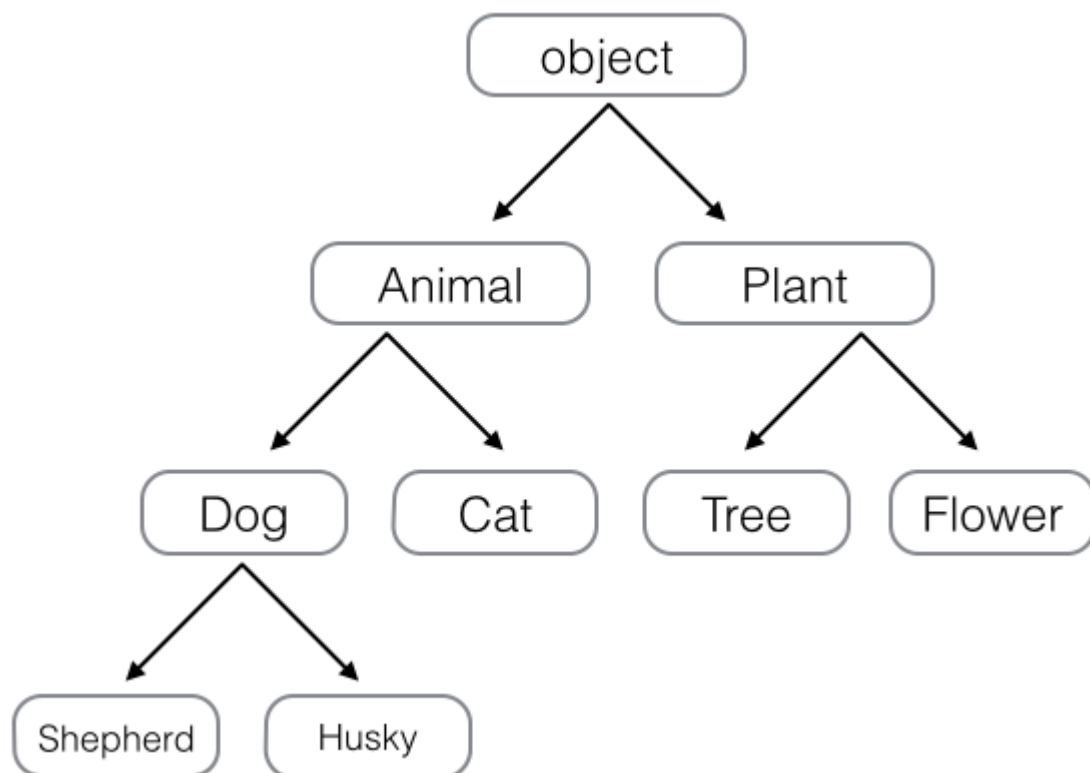
多态的好处就是,当我们需要传入 `Dog`、`Cat`、`Tortoise`.....时,我们只需要接收 `Animal` 类型就可以了,因为 `Dog`、`Cat`、`Tortoise`.....都是 `Animal` 类型,然后,按照 `Animal` 类型进行操作即可。由于 `Animal` 类型有 `run()`方法,因此,传入的任意类型,只要是 `Animal` 类或者子类,就会自动调用实际类型的 `run()`方法,这就是多态的意思:

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

对扩展开放:允许新增 `Animal` 子类;

对修改封闭:不需要修改依赖 `Animal` 类型的 `run_twice()`等函数。

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



#### 小结

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

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

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

## 获取对象信息

重点:

- 1 首先，我们来判断对象类型，使用 `type()` 函数:
- 2 对于 `class` 的继承关系来说，使用 `type()` 就很不方便。我们要判断 `class` 的类型，可以使用 `isinstance()` 函数
- 3 如果要获得一个对象的所有属性和方法，可以使用 `dir()` 函数，它返回一个包含字符串的 `list`，比如，获得一个 `str` 对象的所有属性和方法:
- 4 还有其他的内置函数

当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

使用 `type()`

首先，我们来判断对象类型，使用 `type()` 函数:

基本类型都可以用 `type()` 判断:

```
>>> type(123)
<type 'int'>
>>> type('str')
<type 'str'>
>>> type(None)
<type 'NoneType'>
```

如果一个变量指向函数或者类，也可以用 `type()` 判断:

```
>>> type(abs)
<type 'builtin_function_or_method'>
>>> type(a)
<class '__main__.Animal'>
```

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

```
>>> type(123)==type(456)
True
>>> type('abc')==type('123')
True
>>> type('abc')==type(123)
False
```

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

```
>>> import types
>>> type('abc')==types.StringType
True
>>> type(u'abc')==types.UnicodeType
True
```



```
>>> type([])==types.ListType
True
>>> type(str)==types.TypeType
True
```

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

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

使用 `isinstance()`

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

我们回顾上次的例子，如果继承关系是：

`object -> Animal -> Dog -> Husky`

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

```
>>> a = Animal()
>>> d = Dog()
>>> h = Husky()
```

然后，判断：

```
>>> isinstance(h, Husky)
True
```

没有问题，因为 `h` 变量指向的就是 `Husky` 对象。

再判断：

```
>>> isinstance(h, Dog)
True
```

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

```
>>> isinstance(h, Animal)
True
```

同理，实际类型是 `Dog` 的 `d` 也是 `Animal` 类型：

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

但是，`d` 不是 `Husky` 类型：

```
>>> isinstance(d, Husky)
False
```

能用 `type()` 判断的基本类型也可以用 `isinstance()` 判断：

```
>>> isinstance('a', str)
True
>>> isinstance(u'a', unicode)
True
>>> isinstance('a', unicode)
False
```

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

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

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

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

使用 `dir()`

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

```
>>> dir('ABC')
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__',
 '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '_formatter_field_name_split', '_formatter_parser', 'capitalize', 'center', 'count',
 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit',
 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex',
 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
 'upper', 'zfill']
```

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

```
>>> len('ABC')
3
>>> 'ABC'.__len__()
3
```

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

```
>>> class MyObject(object):
...     def __len__(self):
...         return 100
```

```
...
>>> obj = MyObject()
>>> len(obj)
100
```

剩下的都是普通属性或方法，比如 `lower()` 返回小写的字符串：

```
>>> 'ABC'.lower()
'abc'
```

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

```
>>> class MyObject(object):
...     def __init__(self):
...         self.x = 9
...     def power(self):
...         return self.x * self.x
...
>>> obj = MyObject()
```

紧接着，可以测试该对象的属性：

```
>>> hasattr(obj, 'x') # 有属性'x'吗？
True
>>> obj.x
9
>>> hasattr(obj, 'y') # 有属性'y'吗？
False
>>> setattr(obj, 'y', 19) # 设置一个属性'y'
>>> hasattr(obj, 'y') # 有属性'y'吗？
True
>>> getattr(obj, 'y') # 获取属性'y'
19
>>> obj.y # 获取属性'y'
19
```

如果试图获取不存在的属性，会抛出 `AttributeError` 的错误：

```
>>> getattr(obj, 'z') # 获取属性'z'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyObject' object has no attribute 'z'
```

可以传入一个 `default` 参数，如果属性不存在，就返回默认值：

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

也可以获得对象的方法：

```
>>> hasattr(obj, 'power') # 有属性'power'吗?
True
>>> getattr(obj, 'power') # 获取属性'power'
<bound method MyObject.power of <__main__.MyObject object at 0x108ca35d0>>
>>> fn = getattr(obj, 'power') # 获取属性'power'并赋值到变量 fn
>>> fn # fn 指向 obj.power
<bound method MyObject.power of <__main__.MyObject object at 0x108ca35d0>>
>>> fn() # 调用 fn()与调用 obj.power()是一样的
81
```

小结

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

就不要写：

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

一个正确的用法的例子如下：

```
def readImage(fp):
    if hasattr(fp, 'read'):
        return readData(fp)
    return None
```

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

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

## 面向对象高级编程

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

我们会讨论多重继承、定制类、元类等概念。

### 使用\_\_slots\_\_

**重点：**

1 动态地为类绑定属性和方法

## 2 使用\_\_slots\_\_限制绑定的属性和方法

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

```
>>> class Student(object):
...     pass
...
```

然后，尝试给实例绑定一个属性：

```
>>> s = Student()
>>> s.name = 'Michael' # 动态给实例绑定一个属性
>>> print s.name
Michael
```

还可以尝试给实例绑定一个方法：

```
>>> def set_age(self, age): # 定义一个函数作为实例方法
...     self.age = age
...
>>> from types import MethodType
>>> s.set_age = MethodType(set_age, s, Student) # 给实例绑定一个方法
>>> s.set_age(25) # 调用实例方法
>>> s.age # 测试结果
25
```

但是，给一个实例绑定的方法，对另一个实例是不起作用的：

```
>>> s2 = Student() # 创建新的实例
>>> s2.set_age(25) # 尝试调用方法
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'set_age'
```

为了给所有实例都绑定方法，可以给 class 绑定方法：

```
>>> def set_score(self, score):
...     self.score = score
...
>>> Student.set_score = MethodType(set_score, None, Student)
```

给 class 绑定方法后，所有实例均可调用：

```
>>> s.set_score(100)
>>> s.score
100
>>> s2.set_score(99)
>>> s2.score
99
```

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

### 使用 `__slots__`

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

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

```
>>> class Student(object):
...     __slots__ = ('name', 'age') # 用 tuple 定义允许绑定的属性名称
...
```

然后，我们试试：

```
>>> s = Student() # 创建新的实例
>>> s.name = 'Michael' # 绑定属性'name'
>>> s.age = 25 # 绑定属性'age'
>>> s.score = 99 # 绑定属性'score'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'score'
```

由于 `'score'` 没有被放到 `__slots__` 中，所以不能绑定 `score` 属性，试图绑定 `score` 将得到 `AttributeError` 的错误。

使用 `__slots__` 要注意，`__slots__` 定义的属性仅对当前类起作用，对继承的子类是不起作用的：

```
>>> class GraduateStudent(Student):
...     pass
...
>>> g = GraduateStudent()
>>> g.score = 9999
```

除非在子类中也定义 `__slots__`，这样，子类允许定义的属性就是自身的 `__slots__` 加上父类的 `__slots__`。

## 使用 `@property`

### 重点：

1 `@property` 广泛应用在类的定义中，可以让调用者写出简短的代码，同时保证对参数进行必要的检查，这样，程序运行时就减少了出错的可能性。这个好像很高端，但不知道是不是经常使用，待检验

在绑定属性时，如果我们直接把属性暴露出去，虽然写起来很简单，但是，没办法检查参数，导致可以把成绩随便改：

```
s = Student()
s.score = 9999
```

这显然不合逻辑。为了限制 `score` 的范围，可以通过一个 `set_score()` 方法来设置成绩，再通过一个 `get_score()` 来获取成绩，这样，在 `set_score()` 方法里，就可以检查参数：

```
class Student(object):
```

```
    def get_score(self):
        return self._score

    def set_score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value
```

现在，对任意的 `Student` 实例进行操作，就不能随心所欲地设置 `score` 了：

```
>>> s = Student()
>>> s.set_score(60) # ok!
>>> s.get_score()
60
>>> s.set_score(9999)
Traceback (most recent call last):
...
ValueError: score must between 0 ~ 100!
```

但是，上面的调用方法又略显复杂，没有直接用属性这么直接简单。

有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量呢？对于追求完美的 Python 程序员来说，这是必须要做到的！

还记得装饰器（decorator）可以给函数动态加上功能吗？对于类的方法，装饰器一样起作用。

Python 内置的 `@property` 装饰器就是负责把一个方法变成属性调用的：

```
class Student(object):
```

```
    @property
    def score(self):
        return self._score

    @score.setter
    def score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
```

```
self._score = value
```

@property 的实现比较复杂，我们先考察如何使用。把一个 getter 方法变成属性，只需要加上@property 就可以了，此时，@property 本身又创建了一个装饰器@score.setter，负责把一个 setter 方法变成属性赋值，于是，我们就拥有一个可控的属性操作：

```
>>> s = Student()
>>> s.score = 60 # OK, 实际转化为 s.set_score(60)
>>> s.score # OK, 实际转化为 s.get_score()
60
>>> s.score = 9999
```

Traceback (most recent call last):

...

ValueError: score must between 0 ~ 100!

注意到这个神奇的@property，我们在对实例属性操作的时候，就知道该属性很可能不是直接暴露的，而是通过 getter 和 setter 方法来实现的。

还可以定义只读属性，只定义 getter 方法，不定义 setter 方法就是一个只读属性：

class Student(object):

```
    @property
    def birth(self):
        return self._birth

    @birth.setter
    def birth(self, value):
        self._birth = value

    @property
    def age(self):
        return 2014 - self._birth
```

上面的 birth 是可读写属性，而 age 就是一个只读属性，因为 age 可以根据 birth 和当前时间计算出来。

小结

@property 广泛应用在类的定义中，可以让调用者写出简短的代码，同时保证对参数进行必要的检查，这样，程序运行时就减少了出错的可能性。

## 多重继承

重点：

### 1 一般的面向对象继承

继承是面向对象编程的一个重要的方式，因为通过继承，子类就可以扩展父类的功能。



回忆一下 `Animal` 类层次的设计，假设我们要实现以下 4 种动物：

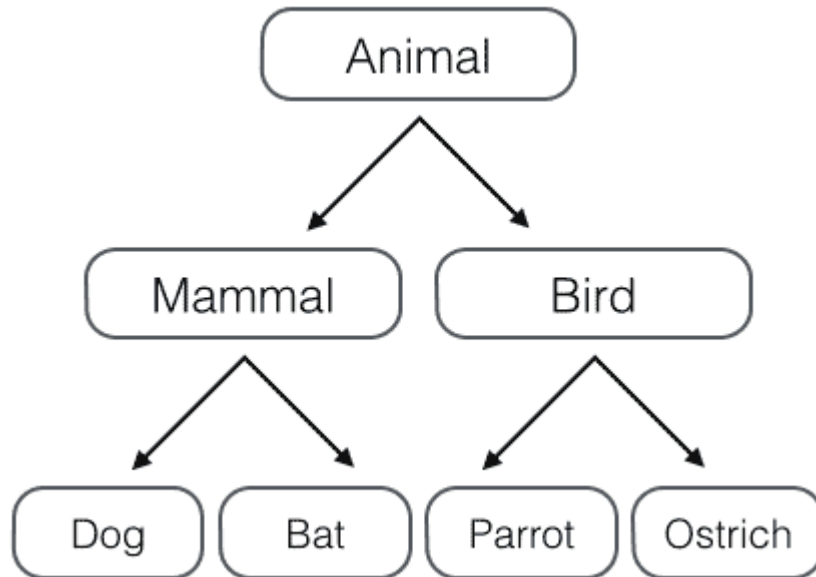
`Dog` - 狗狗；

`Bat` - 蝙蝠；

`Parrot` - 鹦鹉；

`Ostrich` - 鸵鸟。

如果按照哺乳动物和鸟类归类，我们可以设计出这样的类的层次：



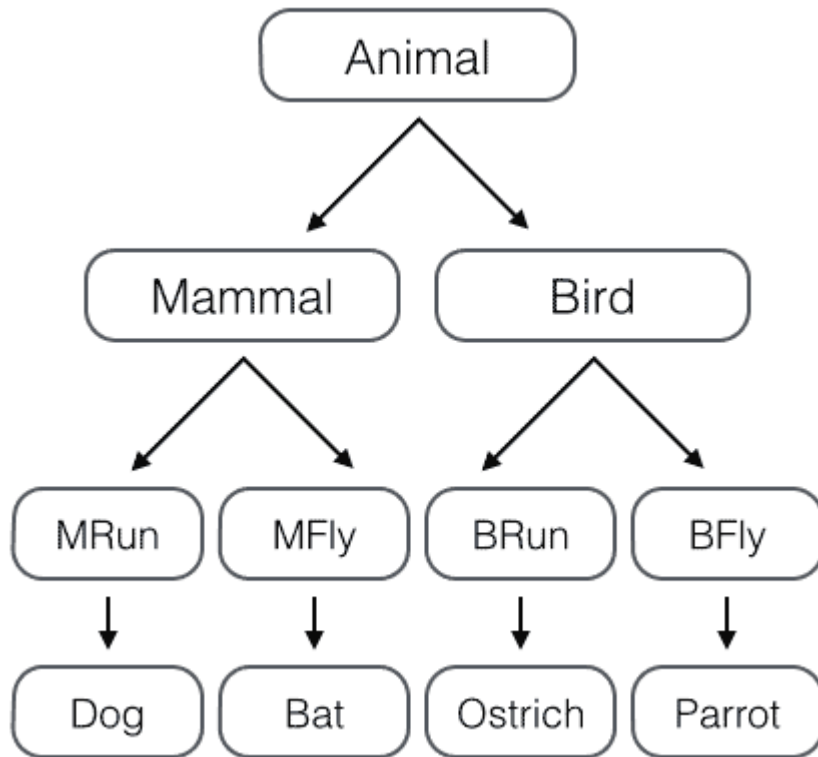
但是如果按照“能跑”和“能飞”来归类，我们就应该设计出这样的类的层次：

如果要把上面的两种分类都包含进来，我们就得设计更多的层次：

哺乳类：能跑的哺乳类，能飞的哺乳类；

鸟类：能跑的鸟类，能飞的鸟类。

这么一来，类的层次就复杂了：



如果要再增加“宠物类”和“非宠物类”，这么搞下去，类的数量会呈指数增长，很明显这样设计是不行的。

正确的做法是采用多重继承。首先，主要的类层次仍按照哺乳类和鸟类设计：

```
class Animal(object):
```

```
    pass
```

```
# 大类:
```

```
class Mammal(Animal):
```

```
    pass
```

```
class Bird(Animal):
```

```
    pass
```

```
# 各种动物:
```

```
class Dog(Mammal):
```

```
    pass
```

```
class Bat(Mammal):
```

```
    pass
```

```
class Parrot(Bird):
```

```
    pass
```

```
class Ostrich(Bird):
```

```
    pass
```

现在,我们要给动物再加上 Runnable 和 Flyable 的功能,只需要先定义好 Runnable 和 Flyable 的类:

```
class Runnable(object):
    def run(self):
        print('Running...')
```

```
class Flyable(object):
    def fly(self):
        print('Flying...')
```

对于需要 Runnable 功能的动物,就多继承一个 Runnable,例如 Dog:

```
class Dog(Mammal, Runnable):
    pass
```

对于需要 Flyable 功能的动物,就多继承一个 Flyable,例如 Bat:

```
class Bat(Mammal, Flyable):
    pass
```

通过多重继承,一个子类就可以同时获得多个父类的所有功能。

## Mixin

在设计类的继承关系时,通常,主线都是单一继承下来的,例如, Ostrich 继承自 Bird。但是,如果需要“混入”额外的功能,通过多重继承就可以实现,比如,让 Ostrich 除了继承自 Bird 外,再同时继承 Runnable。这种设计通常称之为 Mixin。

为了更好地看出继承关系,我们把 Runnable 和 Flyable 改为 RunnableMixin 和 FlyableMixin。类似的,你还可以定义出肉食动物 CarnivorousMixin 和植食动物 HerbivoresMixin,让某个动物同时拥有好几个 Mixin:

```
class Dog(Mammal, RunnableMixin, CarnivorousMixin):
    pass
```

Mixin 的目的就是给一个类增加多个功能,这样,在设计类的时候,我们优先考虑通过多重继承来组合多个 Mixin 的功能,而不是设计多层次的复杂的继承关系。

Python 自带的很多库也使用了 Mixin。举个例子,Python 自带了 TCPServer 和 UDPServer 这两类网络服务,而要同时服务多个用户就必须使用多进程或多线程模型,这两种模型由 ForkingMixin 和 ThreadingMixin 提供。通过组合,我们就可以创造出合适的服务来。

比如,编写一个多进程模式的 TCP 服务,定义如下:

```
class MyTCPServer(TCPServer, ForkingMixin):
    pass
```

编写一个多线程模式的 UDP 服务,定义如下:

```
class MyUDPServer(UDPServer, ThreadingMixin):
    pass
```

如果你打算搞一个更先进的协程模型，可以编写一个 CoroutineMixin:

```
class MyTCPServer(TCPServer, CoroutineMixin):
    pass
```

这样一来，我们不需要复杂而庞大的继承链，只要选择组合不同的类的功能，就可以快速构造出所需的子类。

小结

由于 Python 允许使用多重继承，因此，Mixin 就是一种常见的设计。

只允许单一继承的语言（如 Java）不能使用 Mixin 的设计。

## 定制类

看到类似 `__slots__` 这种形如 `__xxx__` 的变量或者函数名就要注意，这些在 Python 中是有特殊用途的。

`__slots__` 我们已经知道怎么用了，`__len__()` 方法我们也知道是为了能让 class 作用于 `len()` 函数。

除此之外，Python 的 class 中还有许多这样有特殊用途的函数，可以帮助我们定制类。

`__str__`

我们先定义一个 Student 类，打印一个实例：

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...
>>> print Student('Michael')
<__main__.Student object at 0x109afb190>
```

打印出一堆 `<__main__.Student object at 0x109afb190>`，不好看。

怎么才能打印得好看呢？只需要定义好 `__str__()` 方法，返回一个好看的字符串就可以了：

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...     def __str__(self):
...         return 'Student object (name: %s)' % self.name
...
>>> print Student('Michael')
Student object (name: Michael)
```

这样打印出来的实例，不但好看，而且容易看出实例内部重要的数据。

但是细心的朋友会发现直接敲变量不用 `print`，打印出来的实例还是不好看：

```
>>> s = Student('Michael')
>>> s
<__main__.Student object at 0x109afb310>
```

这是因为直接显示变量调用的不是\_\_str\_\_(), 而是\_\_repr\_\_(), 两者的区别是\_\_str\_\_()返回用户看到的字符串, 而\_\_repr\_\_()返回程序开发者看到的字符串, 也就是说, \_\_repr\_\_()是为调试服务的。

解决办法是再定义一个\_\_repr\_\_()。但是通常\_\_str\_\_()和\_\_repr\_\_()代码都是一样的, 所以, 有个偷懒的写法:

```
class Student(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Student object (name=%s)' % self.name
    __repr__ = __str__
```

\_\_iter\_\_

如果一个类想被用于 for ... in 循环, 类似 list 或 tuple 那样, 就必须实现一个\_\_iter\_\_()方法, 该方法返回一个迭代对象, 然后, Python 的 for 循环就会不断调用该迭代对象的 next()方法拿到循环的下一个值, 直到遇到 StopIteration 错误时退出循环。

我们以斐波那契数列为例, 写一个 Fib 类, 可以作用于 for 循环:

```
class Fib(object):
    def __init__(self):
        self.a, self.b = 0, 1 # 初始化两个计数器 a, b

    def __iter__(self):
        return self # 实例本身就是迭代对象, 故返回自己

    def next(self):
        self.a, self.b = self.b, self.a + self.b # 计算下一个值
        if self.a > 100000: # 退出循环的条件
            raise StopIteration();
        return self.a # 返回下一个值
```

现在, 试试把 Fib 实例作用于 for 循环:

```
>>> for n in Fib():
```

```
...     print n
```

```
...
```

```
1
```

```
1
```

```
2
```

```
3
```

```
5
```

```
...
```

```
46368
```

```
75025
```

\_\_getitem\_\_

Fib 实例虽然能作用于 for 循环，看起来和 list 有点像，但是，把它当成 list 来使用还是不行，比如，取第 5 个元素：

```
>>> Fib()[5]
```

Traceback (most recent call last):

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

TypeError: 'Fib' object does not support indexing

要表现得像 list 那样按照下标取出元素，需要实现\_\_getitem\_\_()方法：

```
class Fib(object):
```

```
    def __getitem__(self, n):
```

```
        a, b = 1, 1
```

```
        for x in range(n):
```

```
            a, b = b, a + b
```

```
        return a
```

现在，就可以按下标访问数列的任意一项了：

```
>>> f = Fib()
```

```
>>> f[0]
```

```
1
```

```
>>> f[1]
```

```
1
```

```
>>> f[2]
```

```
2
```

```
>>> f[3]
```

```
3
```

```
>>> f[10]
```

```
89
```

```
>>> f[100]
```

```
573147844013817084101
```

但是 list 有个神奇的切片方法：

```
>>> range(100)[5:10]
```

```
[5, 6, 7, 8, 9]
```

对于 Fib 却报错。原因是\_\_getitem\_\_()传入的参数可能是一个 int，也可能是一个切片对象 slice，所以要做判断：

```
class Fib(object):
```

```
    def __getitem__(self, n):
```

```
        if isinstance(n, int):
```

```
            a, b = 1, 1
```

```
            for x in range(n):
```

```
                a, b = b, a + b
```

```
            return a
```

```
        if isinstance(n, slice):
```

```

start = n.start
stop = n.stop
a, b = 1, 1
L = []
for x in range(stop + 1):
    if x >= start:
        L.append(a)
        a, b = b, a + b
return L

```

现在试试 Fib 的切片：

```

>>> f = Fib()
>>> f[0:5]
[1, 1, 2, 3, 5, 8]
>>> f[:10]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

```

但是没有对 step 参数作处理：

```

>>> f[:10:2]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

```

也没有对负数作处理，所以，要正确实现一个 `__getitem__()` 还是有很多工作要做的。

此外，如果把对象看成 dict，`__getitem__()` 的参数也可能是一个可以作 key 的 object，例如 str。

与之对应的是 `__setitem__()` 方法，把对象视作 list 或 dict 来对集合赋值。最后，还有一个 `__delitem__()` 方法，用于删除某个元素。

总之，通过上面的方法，我们自己定义类表现得和 Python 自带的 list、tuple、dict 没什么区别，这完全归功于动态语言的“鸭子类型”，不需要强制继承某个接口。

`__getattr__`

正常情况下，当我们调用类的方法或属性时，如果不存在，就会报错。比如定义 Student 类：

```
class Student(object):
```

```

    def __init__(self):
        self.name = 'Michael'

```

调用 name 属性，没问题，但是，调用不存在的 score 属性，就有问题了：

```

>>> s = Student()
>>> print s.name
Michael
>>> print s.score
Traceback (most recent call last):
...
AttributeError: 'Student' object has no attribute 'score'

```

错误信息很清楚地告诉我们，没有找到 `score` 这个 `attribute`。

要避免这个错误，除了可以加上一个 `score` 属性外，Python 还有另一个机制，那就是写一个 `__getattr__()` 方法，动态返回一个属性。修改如下：

```
class Student(object):
```

```
    def __init__(self):
        self.name = 'Michael'

    def __getattr__(self, attr):
        if attr=='score':
            return 99
```

当调用不存在的属性时，比如 `score`，Python 解释器会试图调用 `__getattr__(self, 'score')` 来尝试获得属性，这样，我们就有机会返回 `score` 的值：

```
>>> s = Student()
>>> s.name
'Michael'
>>> s.score
99
```

返回函数也是完全可以的：

```
class Student(object):
```

```
    def __getattr__(self, attr):
        if attr=='age':
            return lambda: 25
```

只是调用方式要变为：

```
>>> s.age()
25
```

注意，只有在没有找到属性的情况下，才调用 `__getattr__`，已有的属性，比如 `name`，不会在 `__getattr__` 中查找。

此外，注意到任意调用如 `s.abc` 都会返回 `None`，这是因为我们定义的 `__getattr__` 默认返回就是 `None`。要让 `class` 只响应特定的几个属性，我们就要按照约定，抛出 `AttributeError` 的错误：

```
class Student(object):
```

```
    def __getattr__(self, attr):
        if attr=='age':
            return lambda: 25
        raise AttributeError("'Student' object has no attribute '%s'" % attr)
```

这实际上可以把一个类的所有属性和方法调用全部动态化处理了，不需要任何特殊手段。



这种完全动态调用的特性有什么实际作用呢？作用就是，可以针对完全动态的情况作调用。举个例子：

现在很多网站都搞 REST API，比如新浪微博、豆瓣啥的，调用 API 的 URL 类似：

`http://api.server/user/friends`

`http://api.server/user/timeline/list`

如果要写 SDK，给每个 URL 对应的 API 都写一个方法，那得累死，而且，API 一旦改动，SDK 也要改。

利用完全动态的 `__getattr__`，我们可以写出一个链式调用：

`class Chain(object):`

```
def __init__(self, path=""):
    self._path = path

def __getattr__(self, path):
    return Chain('%s/%s' % (self._path, path))

def __str__(self):
    return self._path
```

试试：

```
>>> Chain().status.user.timeline.list
'/status/user/timeline/list'
```

这样，无论 API 怎么变，SDK 都可以根据 URL 实现完全动态的调用，而且，不随 API 的增加而改变！

还有些 REST API 会把参数放到 URL 中，比如 GitHub 的 API：

`GET /users/:user/repos`

调用时，需要把 `:user` 替换为实际用户名。如果我们能写出这样的链式调用：

`Chain().users('michael').repos`

就可以非常方便地调用 API 了。有兴趣的童鞋可以试试写出来。

`__call__`

一个对象实例可以有自己的属性和方法，当我们调用实例方法时，我们用 `instance.method()` 来调用。能不能直接在实例本身上调用呢？类似 `instance()`？在 Python 中，答案是肯定的。任何类，只需要定义一个 `__call__()` 方法，就可以直接对实例进行调用。请看示例：

`class Student(object):`

```
def __init__(self, name):
    self.name = name

def __call__(self):
    print('My name is %s.' % self.name)
```

调用方式如下：

```
>>> s = Student('Michael')
>>> s()
My name is Michael.
```

`__call__()`还可以定义参数。对实例进行直接调用就好比对一个函数进行调用一样，所以你可以完全可以把对象看成函数，把函数看成对象，因为这两者之间本来就没啥根本的区别。

如果你把对象看成函数，那么函数本身其实也可以在运行期动态创建出来，因为类的实例都是运行期创建出来的，这么一来，我们就模糊了对象和函数的界限。

那么，怎么判断一个变量是对象还是函数呢？其实，更多的时候，我们需要判断一个对象是否能被调用，能被调用的对象就是一个 `Callable` 对象，比如函数和我们上面定义的带有 `__call__()` 的类实例：

```
>>> callable(Student())
True
>>> callable(max)
True
>>> callable([1, 2, 3])
False
>>> callable(None)
False
>>> callable('string')
False
```

通过 `callable()` 函数，我们就可以判断一个对象是否是“可调用”对象。

小结

Python 的 `class` 允许定义许多定制方法，可以让我们非常方便地生成特定的类。

本节介绍的是最常用的几个定制方法，还有很多可定制的方法，请参考 Python 的官方文档。

## 使用元类

`type()`

动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时定义的，而是运行时动态创建的。

比方说我们要定义一个 Hello 的 `class`，就写一个 `hello.py` 模块：

```
class Hello(object):
    def hello(self, name='world'):
        print('Hello, %s.' % name)
```

当 Python 解释器载入 `hello` 模块时，就会依次执行该模块的所有语句，执行结果就是动态创建出一个 Hello 的 `class` 对象，测试如下：

```
>>> from hello import Hello
>>> h = Hello()
>>> h.hello()
```

```
Hello, world.  
>>> print(type(Hello))  
<type 'type'>  
>>> print(type(h))  
<class 'hello.Hello'>
```

`type()`函数可以查看一个类型或变量的类型，`Hello` 是一个 `class`，它的类型就是 `type`，而 `h` 是一个实例，它的类型就是 `class Hello`。

我们说 `class` 的定义是运行时动态创建的，而创建 `class` 的方法就是使用 `type()`函数。

`type()`函数既可以返回一个对象的类型，又可以创建出新的类型，比如，我们可以通过 `type()`函数创建出 `Hello` 类，而无需通过 `class Hello(object)...`的定义：

```
>>> def fn(self, name='world'): # 先定义函数  
...     print('Hello, %s.' % name)  
...  
>>> Hello = type('Hello', (object,), dict(hello=fn)) # 创建 Hello class  
>>> h = Hello()  
>>> h.hello()  
Hello, world.  
>>> print(type(Hello))  
<type 'type'>  
>>> print(type(h))  
<class '__main__.Hello'>
```

要创建一个 `class` 对象，`type()`函数依次传入 3 个参数：

`class` 的名称；

继承的父类集合，注意 Python 支持多重继承，如果只有一个父类，别忘了 `tuple` 的单元素写法；

`class` 的方法名称与函数绑定，这里我们把函数 `fn` 绑定到方法名 `hello` 上。

通过 `type()`函数创建的类和直接写 `class` 是完全一样的，因为 Python 解释器遇到 `class` 定义时，仅仅是扫描一下 `class` 定义的语法，然后调用 `type()`函数创建出 `class`。

正常情况下，我们都用 `class Xxx...`来定义类，但是，`type()`函数也允许我们动态创建出类来，也就是说，动态语言本身支持运行期动态创建类，这和静态语言有非常大的不同，要在静态语言运行期创建类，必须构造源代码字符串再调用编译器，或者借助一些工具生成字节码实现，本质上都是动态编译，会非常复杂。

## metaclass

除了使用 `type()`动态创建类以外，要控制类的创建行为，还可以使用 `metaclass`。

`metaclass`，直译为元类，简单的解释就是：

当我们定义了类以后，就可以根据这个类创建出实例，所以：先定义类，然后创建实例。

但是如果 we 想创建出类呢？那就必须根据 `metaclass` 创建出类，所以：先定义 `metaclass`，然后创建类。

连接起来就是：先定义 `metaclass`，就可以创建类，最后创建实例。

所以，`metaclass` 允许你创建类或者修改类。换句话说，你可以把类看成是 `metaclass` 创建出来的“实例”。

`metaclass` 是 Python 面向对象里最难理解，也是最难使用的魔术代码。正常情况下，你不会

碰到需要使用 metaclass 的情况，所以，以下内容看不懂也没关系，因为基本上你不会用到。我们先看一个简单的例子，这个 metaclass 可以给我们自定义的 MyList 增加一个 add 方法：定义 ListMetaclass，按照默认习惯，metaclass 的类名总是以 Metaclass 结尾，以便清楚地表示这是一个 metaclass：

# metaclass 是创建类，所以必须从`type`类型派生：

```
class ListMetaclass(type):
    def __new__(cls, name, bases, attrs):
        attrs['add'] = lambda self, value: self.append(value)
        return type.__new__(cls, name, bases, attrs)

class MyList(list):
    __metaclass__ = ListMetaclass # 指示使用 ListMetaclass 来定制类
```

当我们写下\_\_metaclass\_\_ = ListMetaclass 语句时，魔术就生效了，它指示 Python 解释器在创建 MyList 时，要通过 ListMetaclass.\_\_new\_\_()来创建，在此，我们可以修改类的定义，比如，加上新的方法，然后，返回修改后的定义。

\_\_new\_\_()方法接收到的参数依次是：

当前准备创建的类的对象；

类的名字；

类继承的父类集合；

类的方法集合。

测试一下 MyList 是否可以调用 add()方法：

```
>>> L = MyList()
>>> L.add(1)
>>> L
[1]
```

而普通的 list 没有 add()方法：

```
>>> l = list()
>>> l.add(1)
```

Traceback (most recent call last):

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

AttributeError: 'list' object has no attribute 'add'

动态修改有什么意义？直接在 MyList 定义中写上 add()方法不是更简单吗？正常情况下，确实应该直接写，通过 metaclass 修改纯属变态。

但是，总会遇到需要通过 metaclass 修改类定义的。ORM 就是一个典型的例子。

ORM 全称“Object Relational Mapping”，即对象-关系映射，就是把关系数据库的一行映射为一个对象，也就是一个类对应一个表，这样，写代码更简单，不用直接操作 SQL 语句。

要编写一个 ORM 框架，所有的类都只能动态定义，因为只有使用者才能根据表的结构定义出对应的类来。

让我们来尝试编写一个 ORM 框架。

编写底层模块的第一步，就是先把调用接口写出来。比如，使用者如果使用这个 ORM 框架，想定义一个 User 类来操作对应的数据库表 User，我们期待他写出这样的代码：

```

class User(Model):
    # 定义类的属性到列的映射:
    id = IntegerField('id')
    name = StringField('username')
    email = StringField('email')
    password = StringField('password')

# 创建一个实例:
u = User(id=12345, name='Michael', email='test@orm.org', password='my-pwd')
# 保存到数据库:
u.save()

```

其中，父类 `Model` 和属性类型 `StringField`、`IntegerField` 是由 ORM 框架提供的，剩下的魔术方法比如 `save()` 全部由 `metaclass` 自动完成。虽然 `metaclass` 的编写会比较复杂，但 ORM 的使用者用起来却异常简单。

现在，我们就按上面的接口来实现该 ORM。

首先来定义 `Field` 类，它负责保存数据库表的字段名和字段类型：

```

class Field(object):
    def __init__(self, name, column_type):
        self.name = name
        self.column_type = column_type
    def __str__(self):
        return '<%s:%s>' % (self.__class__.__name__, self.name)

```

在 `Field` 的基础上，进一步定义各种类型的 `Field`，比如 `StringField`，`IntegerField` 等等：

```

class StringField(Field):
    def __init__(self, name):
        super(StringField, self).__init__(name, 'varchar(100)')

class IntegerField(Field):
    def __init__(self, name):
        super(IntegerField, self).__init__(name, 'bigint')

```

下一步，就是编写最复杂的 `ModelMetaclass` 了：

```

class ModelMetaclass(type):
    def __new__(cls, name, bases, attrs):
        if name=='Model':
            return type.__new__(cls, name, bases, attrs)
        mappings = dict()
        for k, v in attrs.iteritems():
            if isinstance(v, Field):
                print('Found mapping: %s==>%s' % (k, v))
                mappings[k] = v
        for k in mappings.iterkeys():

```

```

        attrs.pop(k)
    attrs['__table__'] = name # 假设表名和类名一致
    attrs['__mappings__'] = mappings # 保存属性和列的映射关系
    return type.__new__(cls, name, bases, attrs)

```

以及基类 Model:

```

class Model(dict):
    __metaclass__ = ModelMetaclass

    def __init__(self, **kw):
        super(Model, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"Model' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value

    def save(self):
        fields = []
        params = []
        args = []
        for k, v in self.__mappings__.iteritems():
            fields.append(v.name)
            params.append('?')
            args.append(getattr(self, k, None))
        sql = 'insert into %s (%s) values (%s)' % (self.__table__, ','.join(fields), ','.join(params))
        print('SQL: %s' % sql)
        print('ARGS: %s' % str(args))

```

当用户定义一个 `class User(Model)` 时，Python 解释器首先在当前类 `User` 的定义中查找 `__metaclass__`，如果没有找到，就继续在父类 `Model` 中查找 `__metaclass__`，找到了，就使用 `Model` 中定义的 `__metaclass__` 的 `ModelMetaclass` 来创建 `User` 类，也就是说，`metaclass` 可以隐式地继承到子类，但子类自己却感觉不到。

在 `ModelMetaclass` 中，一共做了几件事情：

排除掉对 `Model` 类的修改；

在当前类（比如 `User`）中查找定义的类的所有属性，如果找到一个 `Field` 属性，就把它保存到一个 `__mappings__` 的 `dict` 中，同时从类属性中删除该 `Field` 属性，否则，容易造成运行时错误；

把表名保存到 `__table__` 中，这里简化为表名默认为类名。

在 `Model` 类中，就可以定义各种操作数据库的方法，比如 `save()`，`delete()`，`find()`，`update`

等等。

我们实现了 `save()` 方法，把一个实例保存到数据库中。因为有表名，属性到字段的映射和属性值的集合，就可以构造出 `INSERT` 语句。

编写代码试试：

```
u = User(id=12345, name='Michael', email='test@orm.org', password='my-pwd')
u.save()
```

输出如下：

```
Found model: User
Found mapping: email ==> <StringField:email>
Found mapping: password ==> <StringField:password>
Found mapping: id ==> <IntegerField:uid>
Found mapping: name ==> <StringField:username>
SQL: insert into User (password,email,username,uid) values (?, ?, ?, ?)
ARGS: ['my-pwd', 'test@orm.org', 'Michael', 12345]
```

可以看到，`save()` 方法已经打印出了可执行的 SQL 语句，以及参数列表，只需要真正连接到数据库，执行该 SQL 语句，就可以完成真正的功能。

不到 100 行代码，我们就通过 `metaclass` 实现了一个精简的 ORM 框架，完整的代码从这里下载：

[https://github.com/michaelliao/learn-python/blob/master/metaclass/simple\\_orm.py](https://github.com/michaelliao/learn-python/blob/master/metaclass/simple_orm.py)

最后解释一下类属性和实例属性。直接在 `class` 中定义的是类属性：

```
class Student(object):
    name = 'Student'
```

实例属性必须通过实例来绑定，比如 `self.name = 'xxx'`。来测试一下：

```
>>> # 创建实例 s:
>>> s = Student()
>>> # 打印 name 属性，因为实例并没有 name 属性，所以会继续查找 class 的 name 属性:
>>> print(s.name)
Student
>>> # 这和调用 Student.name 是一样的:
>>> print(Student.name)
Student
>>> # 给实例绑定 name 属性:
>>> s.name = 'Michael'
>>> # 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的 name 属性:
>>> print(s.name)
Michael
>>> # 但是类属性并未消失，用 Student.name 仍然可以访问:
>>> print(Student.name)
Student
>>> # 如果删除实例的 name 属性:
>>> del s.name
```

```
>>> # 再次调用 s.name, 由于实例的 name 属性没有找到, 类的 name 属性就显示出来了:
>>> print(s.name)
Student
```

因此, 在编写程序的时候, 千万不要把实例属性和类属性使用相同的名字。  
在我们编写的 ORM 中, ModelMetaclass 会删除掉 User 类的所有类属性, 目的就是避免造成混淆。

## 错误和调试

在程序运行过程中, 总会遇到各种各样的错误。

有的错误是程序编写有问题造成的, 比如本来应该输出整数结果输出了字符串, 这种错误我们通常称之为 bug, bug 是必须修复的。

有的错误是用户输入造成的, 比如让用户输入 email 地址, 结果得到一个空字符串, 这种错误可以通过检查用户输入来做相应的处理。

还有一类错误是完全无法在程序运行过程中预测的, 比如写入文件的时候, 磁盘满了, 写不进去了, 或者从网络抓取数据, 网络突然断掉了。这类错误也称为异常, 在程序中通常是必须处理的, 否则, 程序会因为各种问题终止并退出。

Python 内置了一套异常处理机制, 来帮助我们进行错误处理。

## 错误处理

在程序运行的过程中, 如果发生了错误, 可以事先约定返回一个错误代码, 这样, 就可以知道是否有错, 以及出错的原因。在操作系统提供的调用中, 返回错误码非常常见。比如打开文件的函数 open(), 成功时返回文件描述符 (就是一个整数), 出错时返回 -1。

用错误码来表示是否出错十分不便, 因为函数本身应该返回的正常结果和错误码混在一起, 造成调用者必须用大量的代码来判断是否出错:

```
def foo():
    r = some_function()
    if r == (-1):
        return (-1)
    # do something
    return r
```

```
def bar():
    r = foo()
    if r == (-1):
        print 'Error'
    else:
        pass
```



一旦出错，还要一级一级上报，直到某个函数可以处理该错误（比如，给用户输出一个错误信息）。

所以高级语言通常都内置了一套 `try...except...finally...` 的错误处理机制，Python 也不例外。

`try`

让我们用一个例子来看看 `try` 的机制：

`try:`

```
    print 'try...'
```

```
    r = 10 / 0
```

```
    print 'result:', r
```

`except ZeroDivisionError, e:`

```
    print 'except:', e
```

`finally:`

```
    print 'finally...'
```

`print 'END'`

当我们认为某些代码可能会出错时，就可以用 `try` 来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即 `except` 语句块，执行完 `except` 后，如果有 `finally` 语句块，则执行 `finally` 语句块，至此，执行完毕。

上面的代码在计算 `10 / 0` 时会产生一个除法运算错误：

`try...`

`except: integer division or modulo by zero`

`finally...`

`END`

从输出可以看到，当错误发生时，后续语句 `print 'result:', r` 不会被执行，`except` 由于捕获到 `ZeroDivisionError`，因此被执行。最后，`finally` 语句被执行。然后，程序继续按照流程往下走。

如果把除数 `0` 改成 `2`，则执行结果如下：

`try...`

`result: 5`

`finally...`

`END`

由于没有错误发生，所以 `except` 语句块不会被执行，但是 `finally` 如果有，则一定会被执行（可以没有 `finally` 语句）。

你还可以猜测，错误应该有很多种类，如果发生了不同类型的错误，应该由不同的 `except` 语句块处理。没错，可以有多个 `except` 来捕获不同类型的错误：

`try:`

```
    print 'try...'
```

```
    r = 10 / int('a')
```

```
    print 'result:', r
```

`except ValueError, e:`

```
    print 'ValueError:', e
```

`except ZeroDivisionError, e:`

```

        print 'ZeroDivisionError:', e
finally:
    print 'finally...'
print 'END'

```

int()函数可能会抛出 ValueError，所以我们用一个 except 捕获 ValueError，用另一个 except 捕获 ZeroDivisionError。

此外，如果没有错误发生，可以在 except 语句块后面加一个 else，当没有错误发生时，会自动执行 else 语句：

```

try:
    print 'try...'
    r = 10 / int('a')
    print 'result:', r
except ValueError, e:
    print 'ValueError:', e
except ZeroDivisionError, e:
    print 'ZeroDivisionError:', e
else:
    print 'no error!'
finally:
    print 'finally...'
print 'END'

```

Python 的错误其实也是 class，所有的错误类型都继承自 BaseException，所以在使用 except 时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。比如：

```

try:
    foo()
except StandardError, e:
    print 'StandardError'
except ValueError, e:
    print 'ValueError'

```

第二个 except 永远也捕获不到 ValueError，因为 ValueError 是 StandardError 的子类，如果有，也被第一个 except 给捕获了。

Python 所有的错误都是从 BaseException 类派生的，常见的错误类型和继承关系看这里：

<https://docs.python.org/2/library/exceptions.html#exception-hierarchy>

使用 try...except 捕获错误还有一个巨大的好处，就是可以跨越多层调用，比如函数 main() 调用 foo()，foo()调用 bar()，结果 bar()出错了，这时，只要 main()捕获到了，就可以处理：

```

def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

```

```
def main():
    try:
        bar('0')
    except StandardError, e:
        print 'Error!'
    finally:
        print 'finally...'
```

也就是说，不需要在每个可能出错的地方去捕获错误，只要在合适的层次去捕获错误就可以了。这样一来，就大大减少了写 try...except...finally 的麻烦。

调用堆栈

如果错误没有被捕获，它就会一直往上抛，最后被 Python 解释器捕获，打印一个错误信息，然后程序退出。来看看 err.py:

```
# err.py:
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    bar('0')
```

main()

执行，结果如下：

```
$ python err.py
```

Traceback (most recent call last):

```
File "err.py", line 11, in <module>
    main()
File "err.py", line 9, in main
    bar('0')
File "err.py", line 6, in bar
    return foo(s) * 2
File "err.py", line 3, in foo
    return 10 / int(s)
```

ZeroDivisionError: integer division or modulo by zero

出错并不可怕，可怕的是不知道哪里出错了。解读错误信息是定位错误的关键。我们从上往下可以看到整个错误的调用函数链：

错误信息第 1 行：

Traceback (most recent call last):

告诉我们这是错误的跟踪信息。

第 2 行:

```
File "err.py", line 11, in <module>
    main()
```

调用 `main()` 出错了, 在代码文件 `err.py` 的第 11 行代码, 但原因是第 4 行:

```
File "err.py", line 9, in main
    bar('0')
```

调用 `bar('0')` 出错了, 在代码文件 `err.py` 的第 9 行代码, 但原因是第 6 行:

```
File "err.py", line 6, in bar
    return foo(s) * 2
```

原因是 `return foo(s) * 2` 这个语句出错了, 但这还不是最终原因, 继续往下看:

```
File "err.py", line 3, in foo
    return 10 / int(s)
```

原因是 `return 10 / int(s)` 这个语句出错了, 这是错误产生的源头, 因为下面打印了:

`ZeroDivisionError: integer division or modulo by zero`

根据错误类型 `ZeroDivisionError`, 我们判断, `int(s)` 本身并没有出错, 但是 `int(s)` 返回 0, 在计算 `10 / 0` 时出错, 至此, 找到错误源头。

记录错误

如果不捕获错误, 自然可以让 Python 解释器来打印出错误堆栈, 但程序也被结束了。既然我们能捕获错误, 就可以把错误堆栈打印出来, 然后分析错误原因, 同时, 让程序继续执行下去。

Python 内置的 `logging` 模块可以非常容易地记录错误信息:

```
# err.py
import logging

def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except StandardError, e:
        logging.exception(e)

main()
print 'END'
```

同样是出错，但程序打印完错误信息后会继续执行，并正常退出：

```
$ python err.py
ERROR:root:integer division or modulo by zero
Traceback (most recent call last):
  File "err.py", line 12, in main
    bar('0')
  File "err.py", line 8, in bar
    return foo(s) * 2
  File "err.py", line 5, in foo
    return 10 / int(s)
ZeroDivisionError: integer division or modulo by zero
END
```

通过配置，**logging** 还可以把错误记录到日志文件里，方便事后排查。

抛出错误

因为错误是 **class**，捕获一个错误就是捕获到该 **class** 的一个实例。因此，错误并不是凭空产生的，而是有意创建并抛出的。Python 的内置函数会抛出很多类型的错误，我们自己编写的函数也可以抛出错误。

如果要抛出错误，首先根据需要，可以定义一个错误的 **class**，选择好继承关系，然后，用 **raise** 语句抛出一个错误的实例：

```
# err.py
class FooError(StandardError):
    pass

def foo(s):
    n = int(s)
    if n==0:
        raise FooError('invalid value: %s' % s)
    return 10 / n
```

执行，可以最后跟踪到我们自己定义的错误：

```
$ python err.py
Traceback (most recent call last):
...
__main__.FooError: invalid value: 0
```

只有在必要的时候才定义我们自己的错误类型。如果可以选择 Python 已有的内置的错误类型（比如 **ValueError**，**TypeError**），尽量使用 Python 内置的错误类型。

最后，我们来看另一种错误处理的方式：

```
# err.py
def foo(s):
    n = int(s)
    return 10 / n
```

```
def bar(s):
    try:
        return foo(s) * 2
    except StandardError, e:
        print 'Error!'
        raise

def main():
    bar('0')

main()
```

在 `bar()` 函数中，我们明明已经捕获了错误，但是，打印一个 `Error!` 后，又把错误通过 `raise` 语句抛出去了，这不有病么？

其实这种错误处理方式不但没病，而且相当常见。捕获错误目的只是记录一下，便于后续追踪。但是，由于当前函数不知道应该怎么处理该错误，所以，最恰当的方式是继续往上抛，让顶层调用者去处理。

`raise` 语句如果不带参数，就会把当前错误原样抛出。此外，在 `except` 中 `raise` 一个 `Error`，还可以把一种类型的错误转化成另一种类型：

```
try:
    10 / 0
except ZeroDivisionError:
    raise ValueError('input error!')
```

只要是合理的转换逻辑就可以，但是，决不应该把一个 `IOError` 转换成毫不相干的 `ValueError`。

小结

Python 内置的 `try...except...finally` 用来处理错误十分方便。出错时，会分析错误信息并定位错误发生的代码位置才是最关键的。

程序也可以主动抛出错误，让调用者来处理相应的错误。但是，应该在文档中写清楚可能会抛出哪些错误，以及错误产生的原因。

## 调试

程序能一次写完并正常运行的概率很小，基本不超过 1%。总会有各种各样的 `bug` 需要修正。有的 `bug` 很简单，看看错误信息就知道，有的 `bug` 很复杂，我们需要知道出错时，哪些变量的值是正确的，哪些变量的值是错误的，因此，需要一整套调试程序的手段来修复 `bug`。

第一种方法简单直接粗暴有效，就是用 `print` 把可能有问题的变量打印出来看看：

```
# err.py
def foo(s):
    n = int(s)
    print '>>> n = %d' % n
    return 10 / n
```

```
def main():
    foo('0')
```

```
main()
```

执行后在输出中查找打印的变量值：

```
$ python err.py
```

```
>>> n = 0
```

Traceback (most recent call last):

```
...
```

ZeroDivisionError: integer division or modulo by zero

用 `print` 最大的坏处是将来还得删掉它，想想程序里到处都是 `print`，运行结果也会包含很多垃圾信息。所以，我们又有第二种方法。

断言

凡是用 `print` 来辅助查看的地方，都可以用断言（`assert`）来替代：

```
# err.py
```

```
def foo(s):
    n = int(s)
    assert n != 0, 'n is zero!'
    return 10 / n
```

```
def main():
    foo('0')
```

`assert` 的意思是，表达式 `n != 0` 应该是 `True`，否则，后面的代码就会出错。

如果断言失败，`assert` 语句本身就会抛出 `AssertionError`：

```
$ python err.py
```

Traceback (most recent call last):

```
...
```

AssertionError: n is zero!

程序中如果到处充斥着 `assert`，和 `print` 相比也好不到哪去。不过，启动 Python 解释器时可以用 `-O` 参数来关闭 `assert`：

```
$ python -O err.py
```

Traceback (most recent call last):

```
...
```

ZeroDivisionError: integer division or modulo by zero

关闭后，你可以把所有的 `assert` 语句当成 `pass` 来看。

logging

把 `print` 替换为 `logging` 是第 3 种方式，和 `assert` 比，`logging` 不会抛出错误，而且可以输出到文件：

```
# err.py
```

```
import logging

s = '0'
n = int(s)
logging.info('n = %d' % n)
print 10 / n
```

logging.info()就可以输出一段文本。运行，发现除了 ZeroDivisionError，没有任何信息。怎么回事？

别急，在 import logging 之后添加一行配置再试试：

```
import logging
logging.basicConfig(level=logging.INFO)
```

看到输出了：

```
$ python err.py
INFO:root:n = 0
Traceback (most recent call last):
  File "err.py", line 8, in <module>
    print 10 / n
ZeroDivisionError: integer division or modulo by zero
```

这就是 logging 的好处，它允许你指定记录信息的级别，有 debug, info, warning, error 等几个级别，当我们指定 level=INFO 时，logging.debug 就不起作用了。同理，指定 level=WARNING 后，debug 和 info 就不起作用了。这样一来，你可以放心地输出不同级别的信息，也不用删除，最后统一控制输出哪个级别的信息。

logging 的另一个好处是通过简单的配置，一条语句可以同时输出到不同的地方，比如 console 和文件。

pdb

第 4 种方式是启动 Python 的调试器 pdb，让程序以单步方式运行，可以随时查看运行状态。我们先准备好程序：

```
# err.py
s = '0'
n = int(s)
print 10 / n
```

然后启动：

```
$ python -m pdb err.py
> /Users/michael/Github/sicp/err.py(2)<module>()
-> s = '0'
```

以参数 -m pdb 启动后，pdb 定位到下一步要执行的代码 -> s = '0'。输入命令 l 来查看代码：

```
(Pdb) l
1      # err.py
2  -> s = '0'
```



```
3     n = int(s)
4     print 10 / n
[EOF]
```

输入命令 `n` 可以单步执行代码：

```
(Pdb) n
> /Users/michael/Github/sicp/err.py(3)<module>()
-> n = int(s)
(Pdb) n
> /Users/michael/Github/sicp/err.py(4)<module>()
-> print 10 / n
```

任何时候都可以输入命令 `p` 变量名来查看变量：

```
(Pdb) p s
'0'
(Pdb) p n
0
```

输入命令 `q` 结束调试，退出程序：

```
(Pdb) n
ZeroDivisionError: 'integer division or modulo by zero'
> /Users/michael/Github/sicp/err.py(4)<module>()
-> print 10 / n
(Pdb) q
```

这种通过 `pdb` 在命令行调试的方法理论上是万能的，但实在是太麻烦了，如果有一千行代码，要运行到第 999 行得敲多少命令啊。还好，我们还有另一种调试方法。

`pdb.set_trace()`

这个方法也是用 `pdb`，但是不需要单步执行，我们只需要 `import pdb`，然后，在可能出错的地方放一个 `pdb.set_trace()`，就可以设置一个断点：

```
# err.py
import pdb

s = '0'
n = int(s)
pdb.set_trace() # 运行到这里会自动暂停
print 10 / n
```

运行代码，程序会自动在 `pdb.set_trace()` 暂停并进入 `pdb` 调试环境，可以用命令 `p` 查看变量，或者用命令 `c` 继续运行：

```
$ python err.py
> /Users/michael/Github/sicp/err.py(7)<module>()
-> print 10 / n
(Pdb) p n
```

0

(Pdb) c

Traceback (most recent call last):

File "err.py", line 7, in <module>

print 10 / n

ZeroDivisionError: integer division or modulo by zero

这个方式比直接启动 pdb 单步调试效率要高很多，但也高不到哪去。

IDE

如果要比比较爽地设置断点、单步执行，就需要一个支持调试功能的 IDE。目前比较好的 Python IDE 有 PyCharm:

<http://www.jetbrains.com/pycharm/>

另外，Eclipse 加上 pydev 插件也可以调试 Python 程序。

小结

写程序最痛苦的事情莫过于调试，程序往往会以你意想不到的流程来运行，你期待执行的语句其实根本没有执行，这时候，就需要调试了。

虽然用 IDE 调试起来比较方便，但是最后你会发现，logging 才是终极武器。

## IO 编程

IO 在计算机中指 Input/Output，也就是输入和输出。由于程序和运行时数据是在内存中驻留，由 CPU 这个超快的计算核心来执行，涉及到数据交换的地方，通常是磁盘、网络等，就需要 IO 接口。

比如你打开浏览器，访问新浪首页，浏览器这个程序就需要通过网络 IO 获取新浪的网页。浏览器首先会发送数据给新浪服务器，告诉它我想要首页的 HTML，这个动作是往外发数据，叫 Output，随后新浪服务器把网页发过来，这个动作是从外面接收数据，叫 Input。所以，通常，程序完成 IO 操作会有 Input 和 Output 两个数据流。当然也有只用一个的情况，比如，从磁盘读取文件到内存，就只有 Input 操作，反过来，把数据写到磁盘文件里，就只是一个 Output 操作。

IO 编程中，Stream（流）是一个很重要的概念，可以把流想象成一个水管，数据就是水管里的水，但是只能单向流动。Input Stream 就是数据从内存流到外面（磁盘、网络）去，Output Stream 就是数据从外面流进来。对于浏览网页来说，浏览器和新浪服务器之间至少需要建立两根水管，才可以既能发数据，又能收数据。

由于 CPU 和内存的速度远远高于外设的速度，所以，在 IO 编程中，就存在速度严重不匹配的问题。举个例子来说，比如要把 100M 的数据写入磁盘，CPU 输出 100M 的数据只需要 0.01 秒，可是磁盘要接收这 100M 数据可能需要 10 秒，怎么办呢？有两种办法：

第一种是 CPU 等着，也就是程序暂停执行后续代码，等 100M 的数据在 10 秒后写入磁盘，再接着往下执行，这种模式称为同步 IO；

另一种方法是 CPU 不等待，只是告诉磁盘，“您老慢慢写，不着急，我接着干别的事去了”，于是，后续代码可以立刻接着执行，这种模式称为异步 IO。

同步和异步的区别就在于是否等待 IO 执行的结果。好比你去麦当劳点餐，你说“来个汉堡”，服务员告诉你，对不起，汉堡要现做，需要等 5 分钟，于是你站在收银台前面等了 5 分钟，拿到汉堡再去逛商场，这是同步 IO。

你说“来个汉堡”，服务员告诉你，汉堡需要等 5 分钟，你可以先去逛商场，等做好了，我们再通知你，这样你可以立刻去干别的事情（逛商场），这是异步 IO。

很明显，使用异步 IO 来编写程序性能会远远高于同步 IO，但是异步 IO 的缺点是编程模型复杂。想想看，你得知道什么时候通知你“汉堡做好了”，而通知你的方法也各不相同。如果是服务员跑过来找到你，这是回调模式，如果服务员发短信通知你，你就得不停地检查手机，这是轮询模式。总之，异步 IO 的复杂度远远高于同步 IO。

操作 IO 的能力都是由操作系统提供的，每一种编程语言都会把操作系统提供的低级 C 接口封装起来方便使用，Python 也不例外。我们后面会详细讨论 Python 的 IO 编程接口。

注意，本章的 IO 编程都是同步模式，异步 IO 由于复杂度太高，后续涉及到服务器端程序开发时我们再讨论。

## 文件读写

读写文件是最常见的 IO 操作。Python 内置了读写文件的函数，用法和 C 是兼容的。

读写文件前，我们先必须了解一下，在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）。

读文件

要以读文件的模式打开一个文件对象，使用 Python 内置的 `open()` 函数，传入文件名和标示符：

```
>>> f = open('/Users/michael/test.txt', 'r')
```

标示符 'r' 表示读，这样，我们就成功地打开了一个文件。

如果文件不存在，`open()` 函数就会抛出一个 `IOError` 的错误，并且给出错误码和详细的信息告诉你文件不存在：

```
>>> f=open('/Users/michael/notfound.txt', 'r')
```

Traceback (most recent call last):

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

IOError: [Errno 2] No such file or directory: '/Users/michael/notfound.txt'

如果文件打开成功，接下来，调用 `read()` 方法可以一次读取文件的全部内容，Python 把内容读到内存，用一个 `str` 对象表示：

```
>>> f.read()
'Hello, world!'
```

最后一步是调用 `close()` 方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的：

```
>>> f.close()
```

由于文件读写时都有可能产生 `IOError`，一旦出错，后面的 `f.close()` 就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，我们可以使用 `try ... finally` 来实现：

try:

```

f = open('/path/to/file', 'r')
print f.read()
finally:
    if f:
        f.close()

```

但是每次都这么写实在太繁琐，所以，Python 引入了 `with` 语句来自动帮我们调用 `close()` 方法：

```

with open('/path/to/file', 'r') as f:
    print f.read()

```

这和前面的 `try ... finally` 是一样的，但是代码更佳简洁，并且不必调用 `f.close()` 方法。

调用 `read()` 会一次性读取文件的全部内容，如果文件有 10G，内存就爆了，所以，要保险起见，可以反复调用 `read(size)` 方法，每次最多读取 `size` 个字节的内容。另外，调用 `readline()` 可以每次读取一行内容，调用 `readlines()` 一次读取所有内容并按行返回 `list`。因此，要根据需要决定怎么调用。

如果文件很小，`read()` 一次性读取最方便；如果不能确定文件大小，反复调用 `read(size)` 比较保险；如果是配置文件，调用 `readlines()` 最方便：

```

for line in f.readlines():
    print(line.strip()) # 把末尾的'\n'删掉

```

## file-like Object

像 `open()` 函数返回的这种有个 `read()` 方法的对象，在 Python 中统称为 `file-like Object`。除了 `file` 外，还可以是内存的字节流，网络流，自定义流等等。`file-like Object` 不要求从特定类继承，只要写个 `read()` 方法就行。

`StringIO` 就是在内存中创建的 `file-like Object`，常用作临时缓冲。

## 二进制文件

前面讲的默认都是读取文本文件，并且是 `ASCII` 编码的文本文件。要读取二进制文件，比如图片、视频等等，用 `'rb'` 模式打开文件即可：

```

>>> f = open('/Users/michael/test.jpg', 'rb')
>>> f.read()
'\xff\xd8\xff\xe1\x00\x18Exif\x00\x00...' # 十六进制表示的字节

```

## 字符编码

要读取非 `ASCII` 编码的文本文件，就必须以二进制模式打开，再解码。比如 `GBK` 编码的文件：

```

>>> f = open('/Users/michael/gbk.txt', 'rb')
>>> u = f.read().decode('gbk')
>>> u
u'\u6d4b\u8bd5'
>>> print u

```

测试

如果每次都这么手动转换编码嫌麻烦（写程序怕麻烦是好事，不怕麻烦就会写出又长又难懂

又没法维护的代码), Python 还提供了一个 `codecs` 模块帮我们在读文件时自动转换编码, 直接读出 `unicode`:

```
import codecs
with codecs.open('/Users/michael/gbk.txt', 'r', 'gbk') as f:
    f.read() # u'\u6d4b\u8bd5'
```

写文件

写文件和读文件是一样的, 唯一区别是调用 `open()` 函数时, 传入标识符 `'w'` 或者 `'wb'` 表示写文本文件或写二进制文件:

```
>>> f = open('/Users/michael/test.txt', 'w')
>>> f.write('Hello, world!')
>>> f.close()
```

你可以反复调用 `write()` 来写入文件, 但是务必要调用 `f.close()` 来关闭文件。当我们写文件时, 操作系统往往不会立刻把数据写入磁盘, 而是放到内存缓存起来, 空闲的时候再慢慢写入。只有调用 `close()` 方法时, 操作系统才保证把没有写入的数据全部写入磁盘。忘记调用 `close()` 的后果是数据可能只写了一部分到磁盘, 剩下的丢失了。所以, 还是用 `with` 语句来得保险:

```
with open('/Users/michael/test.txt', 'w') as f:
    f.write('Hello, world!')
```

要写入特定编码的文本文件, 请效仿 `codecs` 的示例, 写入 `unicode`, 由 `codecs` 自动转换成指定编码。

小结

在 Python 中, 文件读写是通过 `open()` 函数打开的文件对象完成的。使用 `with` 语句操作文件 IO 是个好习惯。

## 操作系统

正在编写中。。。

## 序列化

在程序运行的过程中, 所有的变量都是在内存中, 比如, 定义一个 `dict`:

```
d = dict(name='Bob', age=20, score=88)
```

可以随时修改变量, 比如把 `name` 改成 `'Bill'`, 但是一旦程序结束, 变量所占用的内存就被操作系统全部回收。如果没有把修改后的 `'Bill'` 存储到磁盘上, 下次重新运行程序, 变量又被初始化为 `'Bob'`。

我们把变量存储到磁盘的过程称之为序列化, 在 Python 中叫 `pickling`, 在其他语言中也被称之为 `serialization`, `marshalling`, `flattening` 等等, 都是一个意思。

反过来, 从磁盘把变量内容重新读到内存里称之为反序列化, 即 `unpickling`。

Python 提供两个模块来实现序列化: `cPickle` 和 `pickle`。这两个模块功能是一样的, 区别在于

cPickle 是 C 语言写的，速度快，pickle 是纯 Python 写的，速度慢，跟 cStringIO 和 StringIO 一个道理。用的时候，先尝试导入 cPickle，如果失败，再导入 pickle：

try:

```
    import cPickle as pickle
except ImportError:
    import pickle
```

首先，我们尝试把一个对象序列化并写入文件：

```
>>> d = dict(name='Bob', age=20, score=88)
>>> pickle.dumps(d)
"(dp0\nS'age'\np1\nI20\nsS'score'\np2\nI88\nsS'name'\np3\nS'Bob'\np4\ns."
```

pickle.dumps()方法把任意对象序列化成一个 str，然后，就可以把这个 str 写入文件。或者用另一个方法 pickle.dump()直接把对象序列化后写入一个 file-like Object：

```
>>> f = open('dump.txt', 'wb')
>>> pickle.dump(d, f)
>>> f.close()
```

看看写入的 dump.txt 文件，一堆乱七八糟的内容，这些都是 Python 保存的对象内部信息。当我们要把对象从磁盘读到内存时，可以先把内容读到一个 str，然后用 pickle.loads()方法反序列化出对象，也可以直接用 pickle.load()方法从一个 file-like Object 中直接反序列化出对象。我们打开另一个 Python 命令行来反序列化刚才保存的对象：

```
>>> f = open('dump.txt', 'rb')
>>> d = pickle.load(f)
>>> f.close()
>>> d
{'age': 20, 'score': 88, 'name': 'Bob'}
```

变量的内容又回来了！

当然，这个变量和原来的变量是完全不相干的对象，它们只是内容相同而已。

Pickle 的问题和所有其他编程语言特有的序列化问题一样，就是它只能用于 Python，并且可能不同版本的 Python 彼此都不兼容，因此，只能用 Pickle 保存那些不重要的数据，不能成功地反序列化也没关系。

## JSON

如果我们要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式，比如 XML，但更好的方法是序列化为 JSON，因为 JSON 表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。JSON 不仅是标准格式，并且比 XML 更快，而且可以直接在 Web 页面中读取，非常方便。

JSON 表示的对象就是标准的 JavaScript 语言的对象，JSON 和 Python 内置的数据类型对应如下：

JSON 类型	Python 类型
{}	dict

<code>[]</code>	list
<code>"string"</code>	'str'或 u'unicode'
1234.56	int 或 float
true/false	True/False
null	None

Python 内置的 json 模块提供了非常完善的 Python 对象到 JSON 格式的转换。我们先看看如何把 Python 对象变成一个 JSON:

```
>>> import json
>>> d = dict(name='Bob', age=20, score=88)
>>> json.dumps(d)
'{"age": 20, "score": 88, "name": "Bob"}'
```

dumps()方法返回一个 str，内容就是标准的 JSON。类似的，dump()方法可以直接把 JSON 写入一个 file-like Object。

要把 JSON 反序列化为 Python 对象，用 loads()或者对应的 load()方法，前者把 JSON 的字符串反序列化，后者从 file-like Object 中读取字符串并反序列化:

```
>>> json_str = '{"age": 20, "score": 88, "name": "Bob"}'
>>> json.loads(json_str)
{'u'age': 20, u'score': 88, u'name': u'Bob'}
```

有一点需要注意，就是反序列化得到的所有字符串对象默认都是 unicode 而不是 str。由于 JSON 标准规定 JSON 编码是 UTF-8，所以我们总是能正确地在 Python 的 str 或 unicode 与 JSON 的字符串之间转换。

### JSON 进阶

Python 的 dict 对象可以直接序列化为 JSON 的 {}，不过，很多时候，我们更喜欢用 class 表示对象，比如定义 Student 类，然后序列化:

```
import json

class Student(object):
    def __init__(self, name, age, score):
        self.name = name
        self.age = age
        self.score = score

s = Student('Bob', 20, 88)
print(json.dumps(s))
```

运行代码，毫不留情地得到一个 TypeError:

Traceback (most recent call last):

...

`TypeError: <__main__.Student object at 0x10aabef50> is not JSON serializable`

错误的原因是 `Student` 对象不是一个可序列化为 JSON 的对象。

如果连 `class` 的实例对象都无法序列化为 JSON，这肯定不合理！

别急，我们仔细看看 `dump()` 方法的参数列表，可以发现，除了第一个必须的 `obj` 参数外，`dump()` 方法还提供了一大堆的可选参数：

<https://docs.python.org/2/library/json.html#json.dump>

这些可选参数就是让我们来定制 JSON 序列化。前面的代码之所以无法把 `Student` 类实例序列化为 JSON，是因为默认情况下，`dump()` 方法不知道如何将 `Student` 实例变为一个 JSON 的 {} 对象。

可选参数 `default` 就是把任意一个对象变成一个可序列为 JSON 的对象，我们只需要为 `Student` 专门写一个转换函数，再把函数传进去即可：

```
def student2dict(std):
    return {
        'name': std.name,
        'age': std.age,
        'score': std.score
    }

print(json.dump(s, default=student2dict))
```

这样，`Student` 实例首先被 `student2dict()` 函数转换成 `dict`，然后再被顺利序列化为 JSON。

不过，下次如果遇到一个 `Teacher` 类的实例，照样无法序列化为 JSON。我们可以偷个懒，把任意 `class` 的实例变为 `dict`：

```
print(json.dump(s, default=lambda obj: obj.__dict__))
```

因为通常 `class` 的实例都有一个 `__dict__` 属性，它就是一个 `dict`，用来存储实例变量。也有少数例外，比如定义了 `__slots__` 的 `class`。

同样的道理，如果我们要把 JSON 反序列化为一个 `Student` 对象实例，`loads()` 方法首先转换出一个 `dict` 对象，然后，我们传入的 `object_hook` 函数负责把 `dict` 转换为 `Student` 实例：

```
def dict2student(d):
    return Student(d['name'], d['age'], d['score'])

json_str = '{"age": 20, "score": 88, "name": "Bob"}'
print(json.loads(json_str, object_hook=dict2student))
```

运行结果如下：

```
<__main__.Student object at 0x10cd3c190>
```

打印出的是反序列化的 `Student` 实例对象。

小结

Python 语言特定的序列化模块是 `pickle`，但如果要把序列化搞得更通用、更符合 Web 标准，就可以使用 `json` 模块。

`json` 模块的 `dump()` 和 `loads()` 函数是定义得非常好的接口的典范。当我们使用时，只需要传



入一个必须的参数。但是，当默认的序列化或反序列化机制不满足我们的要求时，我们又可以传入更多的参数来定制序列化或反序列化的规则，既做到了接口简单易用，又做到了充分的扩展性和灵活性。

这部分还在编写中。。。。。。。。。。。。。。。。。。。。。。。。

## 进程和线程

## 进程

## 线程

## 正则表达式

## 常用内建模块

## 常用第三方模块

## 图形界面

## 网络编程

## TCP/IP 简介

## Socket 编程

## UDP 编程

电子邮件

访问数据库

## 使用 sqlite

## 使用 MySQL

## Web 开发

## HTML 与 HTTP 协议简介

## 发起 HTTP 请求

## WSGI 接口

## 处理 HTTP 请求

## 动态响应

## 使用模板

## 常用 Web 框架

部署与运维  
分布式进程  
协程