

Python 3教程



廖雪峰

目录

前言	0
简介	1
安装	2
解释器	2.1
第一个程序	3
使用文本编辑器	3.1
代码运行助手	3.2
输入和输出	3.3
基础	4
数据类型和变量	4.1
字符串和编码	4.2
使用list和tuple	4.3
条件判断	4.4
循环	4.5
使用dict和set	4.6
函数	5
调用函数	5.1
定义函数	5.2
函数的参数	5.3
递归函数	5.4
高级特性	6
切片	6.1
迭代	6.2
列表生成式	6.3
生成器	6.4
迭代器	6.5
函数式编程	7
高阶函数	7.1
map/reduce	7.1.1
filter	7.1.2
sorted	7.1.3
返回函数	7.2
匿名函数	7.3

装饰器	7.4
偏函数	7.5
模块	8
使用模块	8.1
安装第三方模块	8.2
面向对象编程	9
类和实例	9.1
访问限制	9.2
继承和多态	9.3
获取对象信息	9.4
实例属性和类属性	9.5
面向对象高级编程	10
使用__slots__	10.1
使用@property	10.2
多重继承	10.3
定制类	10.4
使用枚举类	10.5
使用元类	10.6
错误、调试和测试	11
错误处理	11.1
调试	11.2
单元测试	11.3
文档测试	11.4
IO编程	12
文件读写	12.1
StringIO和BytesIO	12.2
操作文件和目录	12.3
序列化	12.4
进程和线程	13
多进程	13.1
多线程	13.2
ThreadLocal	13.3
进程 vs 线程	13.4
分布式进程	13.5
正则表达式	14
常用内建模块	15

datetime	15.1
collections	15.2
base64	15.3
struct	15.4
hashlib	15.5
hmac	15.6
itertools	15.7
contextlib	15.8
urllib	15.9
XML	15.10
HTMLParser	15.11
常用第三方模块	16
Pillow	16.1
requests	16.2
chardet	16.3
psutil	16.4
virtualenv	17
图形界面	18
网络编程	19
TCP/IP简介	19.1
TCP编程	19.2
UDP编程	19.3
电子邮件	20
SMTP发送邮件	20.1
POP3收取邮件	20.2
访问数据库	21
使用SQLite	21.1
使用MySQL	21.2
使用SQLAlchemy	21.3
Web开发	22
HTTP协议简介	22.1
HTML简介	22.2
WSGI接口	22.3
使用Web框架	22.4
使用模板	22.5
异步IO	23

<u>协程</u>	23.1
<u>asyncio</u>	23.2
<u>async/await</u>	23.3
<u>aiohttp</u>	23.4
<u>实战</u>	24
<u>Day 1 - 搭建开发环境</u>	24.1
<u>Day 2 - 编写Web App骨架</u>	24.2
<u>Day 3 - 编写ORM</u>	24.3
<u>Day 4 - 编写Model</u>	24.4
<u>Day 5 - 搭建Web框架</u>	24.5
<u>Day 6 - 编写Web App骨架</u>	24.6
<u>Day 7 - 编写配置文件</u>	24.7
<u>Day 8 - 编写MVC</u>	24.8
<u>Day 9 - 构建前端</u>	24.9
<u>Day 10 - 编写API</u>	24.10
<u>Day 11 - 用户注册和登录</u>	24.11
<u>Day 12 - 编写日志创建页</u>	24.12
<u>Day 13 - 编写日志列表页</u>	24.13
<u>Day 14 - 提升开发效率</u>	24.14
<u>Day 15 - 部署Web App</u>	24.15
<u>Day 16 - 编写移动App</u>	24.16
<u>FAQ</u>	25
<u>期末总结</u>	26

前言

这是小白的Python新手教程，具有如下特点：

中文，免费，零起点，完整示例，基于最新的Python 3版本。

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？

首先，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，代码少好不好？代码少的代价是运行速度慢，C程序运行1秒钟，Java程序可能需要2秒，而Python程序可能就需要10秒。

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

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

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

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

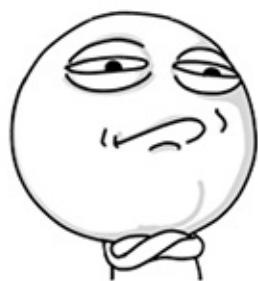
如果你是小白用户，满足以下条件：

- 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？

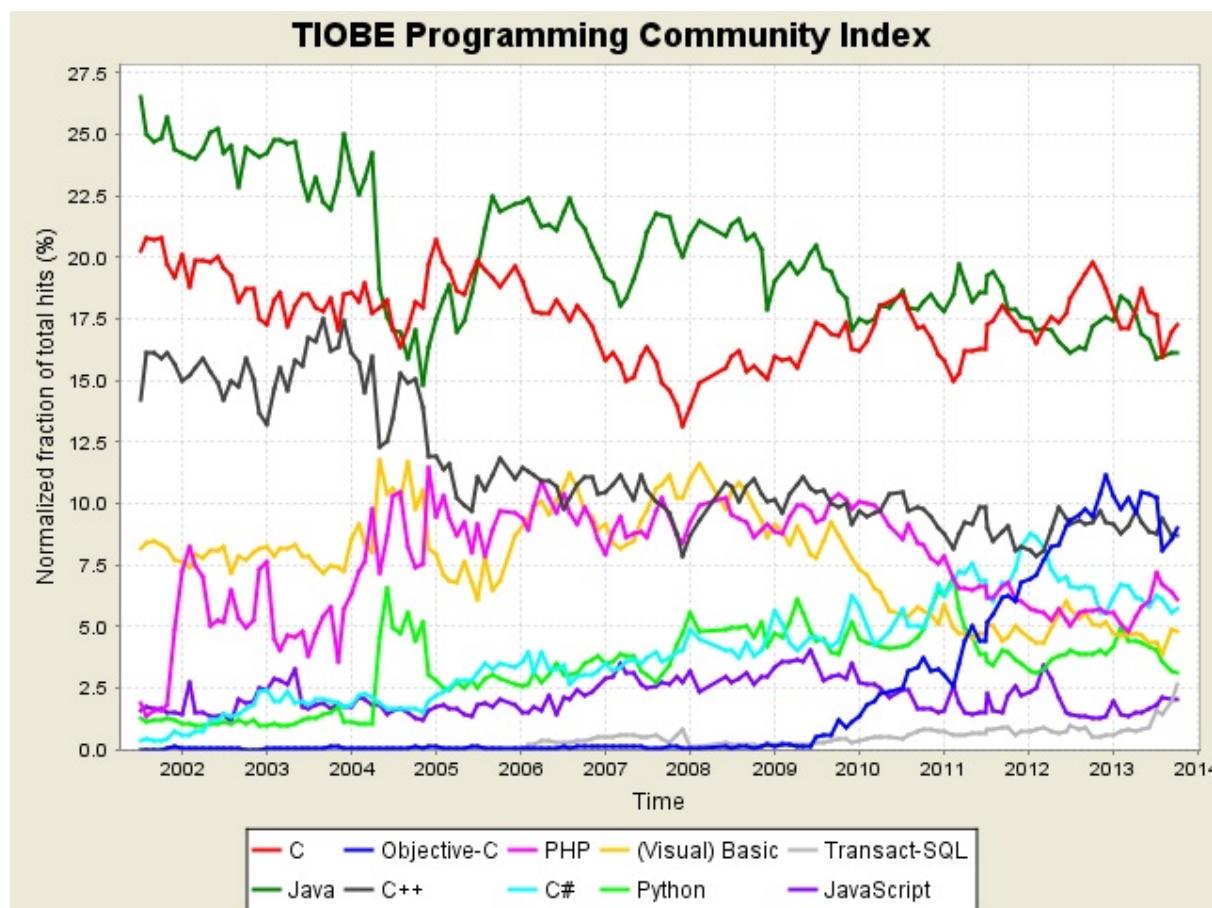
CHALLENGE ACCEPTED !



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；

其次は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

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。

安装Python 3.6

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。由于3.x版越来越普及，我们的教程将以最新的Python 3.6版本为基础。请确保你的电脑上安装的Python版本是最新的3.6.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 10.8~10.10，那么系统自带的Python版本是2.7。要安装最新的Python 3.6，有两个方法：

方法一：从Python官网下载Python 3.6的[安装程序](#)（网速慢的同学请移步[国内镜像](#)），双击运行并安装；

方法二：如果安装了Homebrew，直接通过命令 `brew install python3` 安装即可。

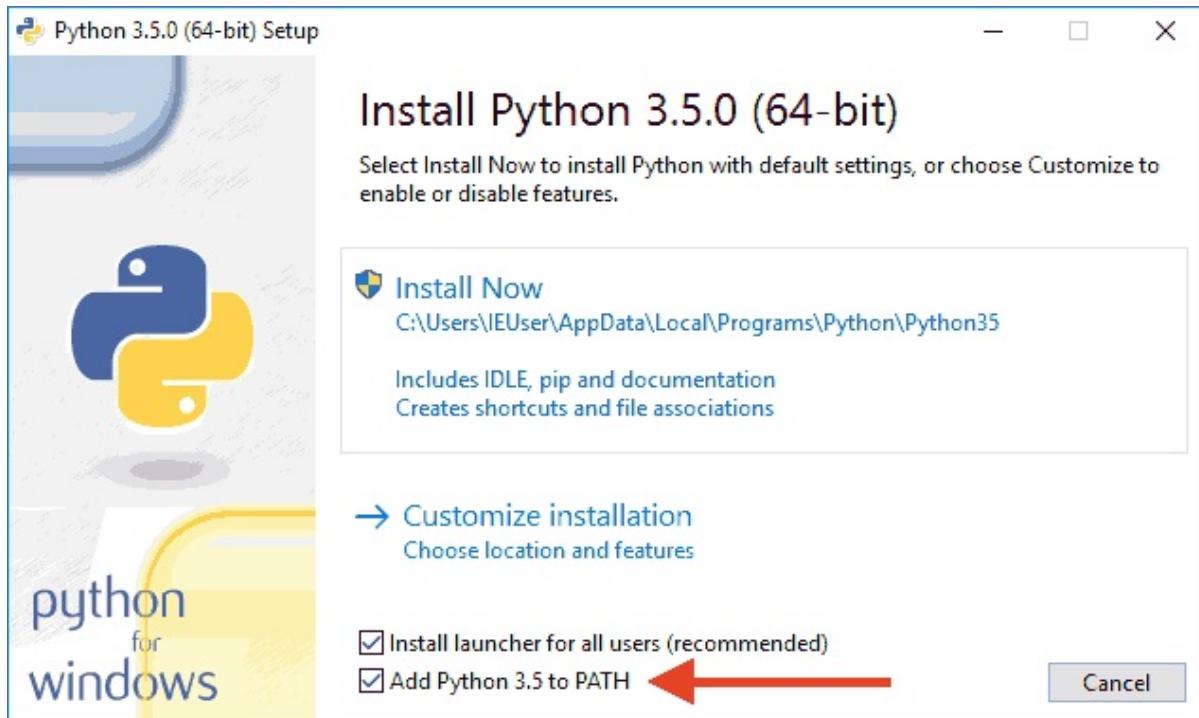
在Linux上安装Python

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 3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，根据你的Windows版本（64位还是32位）从Python的官方网站下载Python 3.6对应的[64位安装程序](#)或[32位安装程序](#)（网速慢的同学请移步[国内镜像](#)），然后，运行下载的EXE安装包：



特别要注意勾上 Add Python 3.6 to PATH , 然后点“Install Now”即可完成安装。

运行Python

安装成功后，打开命令提示符窗口，敲入python后，会出现两种情况：

情况一：

A screenshot of a Windows Command Prompt window titled "Command Prompt - python". The window contains the following text:

```
(c) 2015 Microsoft Corporation. All rights reserved.  
C:\Users\IEUser>python  
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:27:37)  
[MSC v.1900 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license" for more  
information.  
>>> -
```

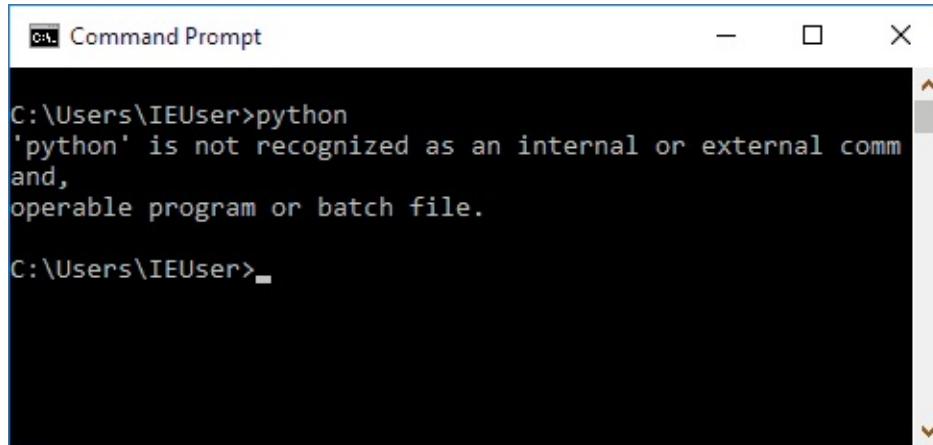
The window title bar also displays "Command Prompt - python".

看到上面的画面，就说明Python安装成功！

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

情况二：得到一个错误：

```
'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。如果在安装时漏掉了勾选 Add Python 3.6 to PATH ，那就要手动把 python.exe 所在的路径添加到Path中。

如果你不知道怎么修改环境变量，建议把Python安装程序重新运行一遍，务必记得勾上 Add Python 3.6 to PATH 。

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。

在Windows上运行Python时，请先启动命令行，然后运行 python 。

在Mac和Linux上运行Python时，请打开终端，然后运行 python3 。

解释器

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

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

C`Python`

当我们从[Python官方网站](#)下载并安装好Python 3.x后，我们就直接获得了一个官方版本的解释器：CPython。这个解释器是用C语言开发的，所以叫CPython。在命令行下运行 `python` 就是启动CPython解释器。

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

I`Python`

I`Python`是基于CPython之上一个交互式解释器，也就是说，I`Python`只是在交互方式上有所增强，但是执行Python代码的功能和CPython是完全一样的。好比很多国产浏览器虽然外观不同，但内核其实都是调用了IE。

CPython用 `>>>` 作为提示符，而I`Python`用 `In [序号]:` 作为提示符。

Py`Py`

Py`Py`是另一个Python解释器，它的目标是执行速度。Py`Py`采用[JIT技术](#)，对Python代码进行动态编译（注意不是解释），所以可以显著提高Python代码的执行速度。

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

J`ython`

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

I`ronPython`

I`ronPython`和J`ython`类似，只不过I`ronPython`是运行在微软.NET平台上的Python解释器，可以直接把Python代码编译成.NET的字节码。

小结

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

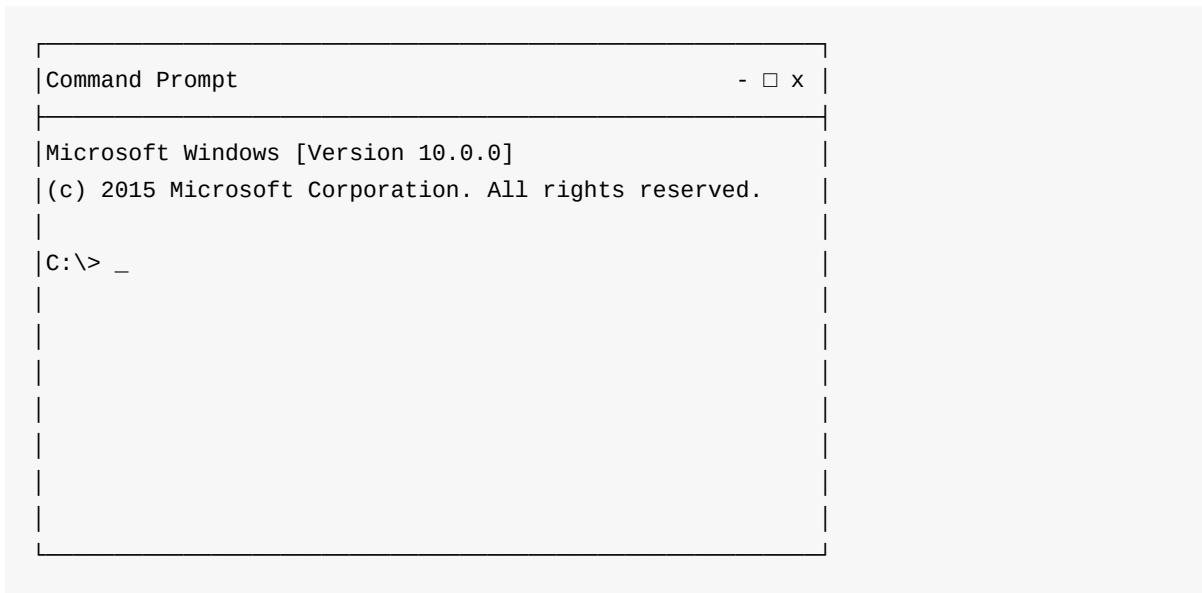
本教程的所有代码只确保在CPython 3.x版本下运行。请务必在本地安装CPython（也就是从Python官方网站下载的安装程序）。

第一个程序

在正式编写第一个Python程序前，我们先复习一下什么是命令行模式和Python交互模式。

命令行模式

在Windows开始菜单选择“命令提示符”，就进入到命令行模式，它的提示符类似 c:\> :



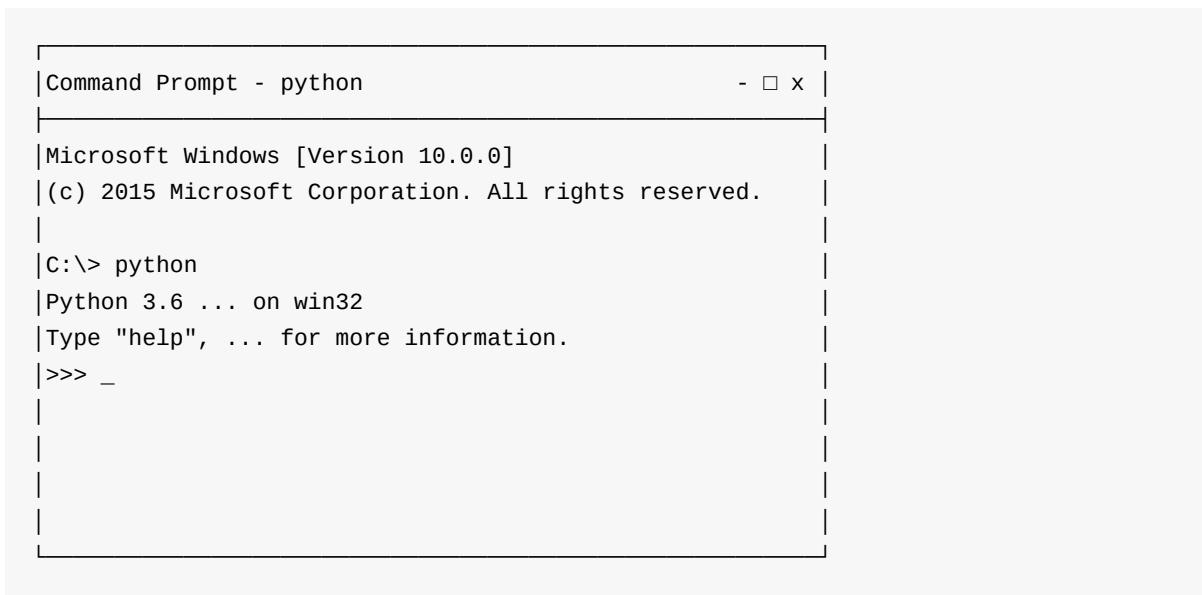
```
Command Prompt - □ X

Microsoft Windows [Version 10.0.0]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\> _
```

Python交互模式

在命令行模式下敲命令 python，就看到类似如下的一堆文本输出，然后就进入到Python交互模式，它的提示符是 >>> 。



```
Command Prompt - python - □ X

Microsoft Windows [Version 10.0.0]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\> python
Python 3.6 ... on win32
Type "help", ... for more information.
>>> _
```

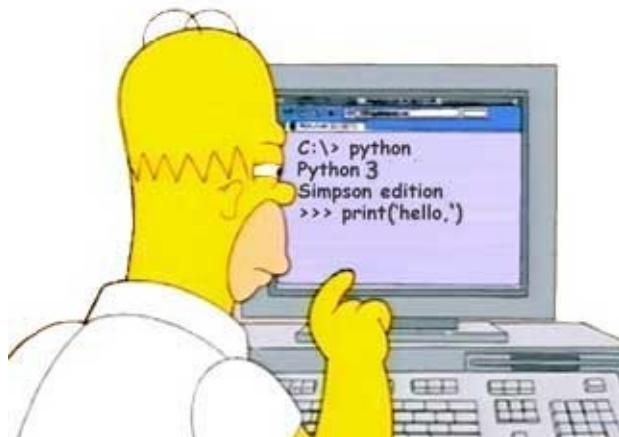
在Python交互模式下输入 `exit()` 并回车，就退出了Python交互模式，并回到命令行模式：

```
| Command Prompt - □ × |
|
| Microsoft Windows [Version 10.0.0]
| (c) 2015 Microsoft Corporation. All rights reserved.
|
| C:\> python
| Python 3.6 ... on win32
| Type "help", ... for more information.
| >>> exit()
|
| C:\> _
```

也可以直接通过开始菜单选择 `Python (command line)` 菜单项，直接进入Python交互模式，但是输入 `exit()` 后窗口会直接关闭，不会回到命令行模式。

了解了如何启动和退出Python的交互模式，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：拼写不对，大小写不对，混用中英文标点，混用空格和Tab键，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。



在交互模式的提示符 `>>>` 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()` 函数，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。

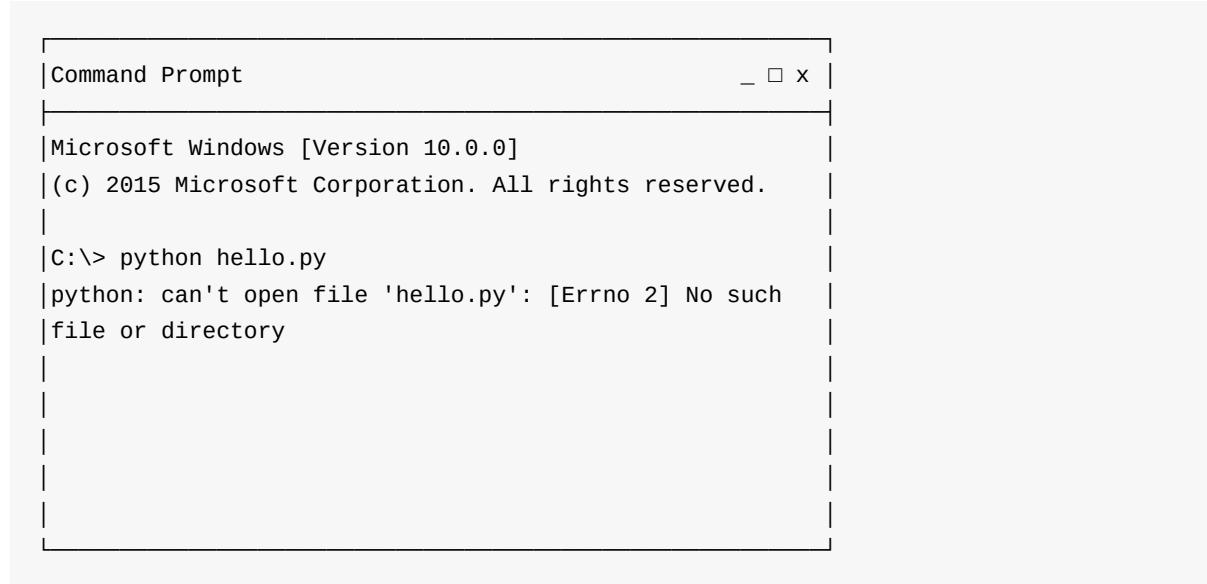
最后，用 `exit()` 退出Python，我们的第一个Python程序完成！唯一的缺憾是没有保存下来，下次运行时还要再输入一遍代码。

命令行模式和Python交互模式

请注意区分命令行模式和Python交互模式。

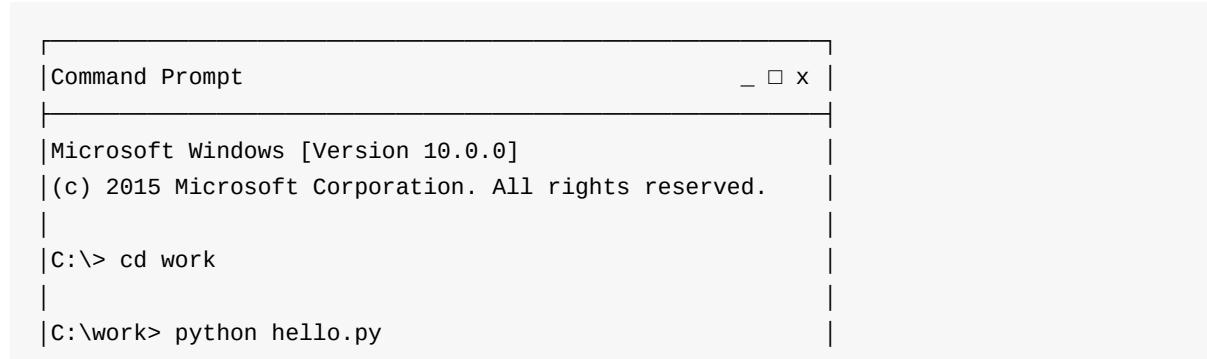
在命令行模式下，可以执行 `python` 进入Python交互式环境，也可以执行 `python hello.py` 运行一个 `.py` 文件。

执行一个 `.py` 文件只能在命令行模式执行。如果敲一个命令 `python hello.py`，看到如下错误：



```
| Command Prompt
|
| Microsoft Windows [Version 10.0.0]
| (c) 2015 Microsoft Corporation. All rights reserved.
|
| C:\> python hello.py
| python: can't open file 'hello.py': [Errno 2] No such
| file or directory
|
```

错误提示 `No such file or directory` 说明这个 `hello.py` 在当前目录找不到，必须先把当前目录切换到 `hello.py` 所在的目录下，才能正常执行：



```
| Command Prompt
|
| Microsoft Windows [Version 10.0.0]
| (c) 2015 Microsoft Corporation. All rights reserved.
|
| C:\> cd work
|
| C:\work> python hello.py
```

```
|Hello, world!  
|  
|  
|  
|
```

此外，在命令行模式运行 .py 文件和在Python交互式环境下直接运行Python代码有所不同。Python 交互式环境会把每一行Python代码的结果自动打印出来，但是，直接运行Python代码却不会。

例如，在Python交互式环境下，输入：

```
>>> 100 + 200 + 300  
600
```

直接可以看到结果 600。

但是，写一个 calc.py 的文件，内容如下：

```
100 + 200 + 300
```

然后在命令行模式下执行：

```
C:\work>python calc.py
```

发现什么输出都没有。

这是正常的。想要输出结果，必须自己用 print() 打印出来。把 calc.py 改造一下：

```
print(100 + 200 + 300)
```

再执行，就可以看到结果：

```
C:\work>python calc.py  
600
```

最后，Python交互模式的代码是输入一行，执行一行，而命令行模式下直接运行 .py 文件是一次性执行该文件内的所有代码。可见，Python交互模式主要是为了调试Python代码用的，也便于初学者学习，它不是正式运行Python代码的环境！

小结

在Python交互式模式下，可以直接输入代码，然后执行，并立刻得到结果。

在命令行模式下，可以直接运行 .py 文件。

使用文本编辑器

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

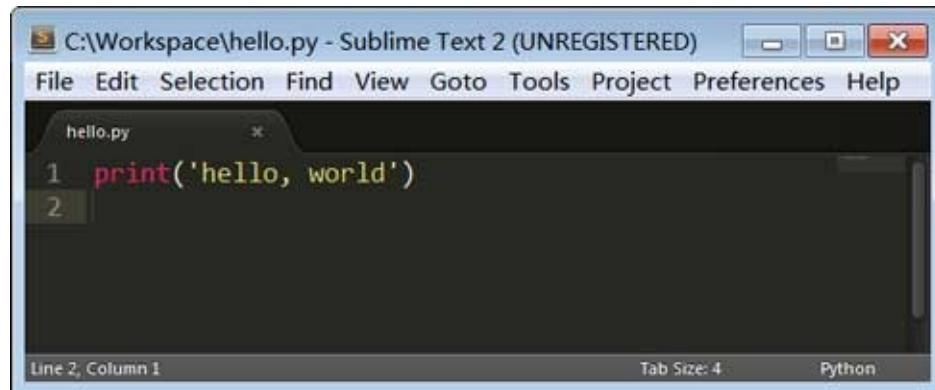
所以，实际开发的时候，我们总是使用一个文本编辑器来写代码，写完了，保存为一个文件，这样，程序就可以反复运行了。

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

那么问题来了：文本编辑器到底哪家强？

推荐两款文本编辑器：

一个是[Sublime Text](#)，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:\work`，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:\work>python hello.py
hello, world
```

也可以保存为别的名字，比如 `first.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：

```
C:\Users\IEUser>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。如果 `hello.py` 存放在另外一个目录下，要首先用 `cd` 命令切换当前目录。

直接运行py文件

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` 文件的第一行加上一个特殊的注释：

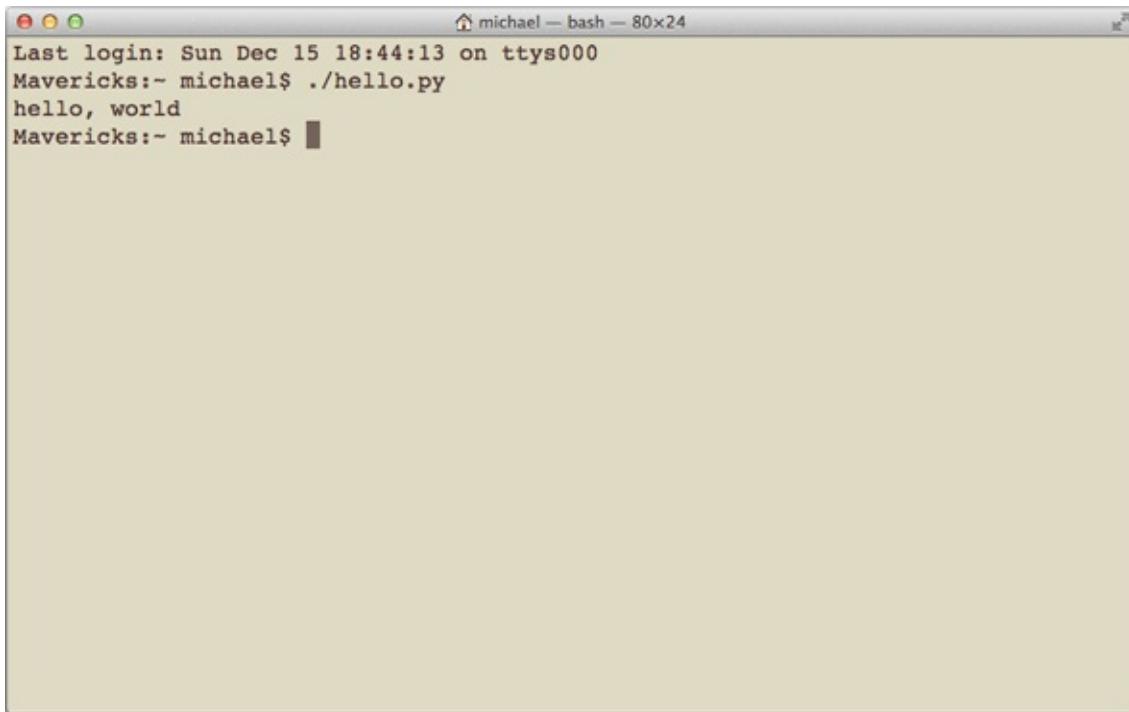
```
#!/usr/bin/env python3

print('hello, world')
```

然后，通过命令给 `hello.py` 以执行权限：

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

就可以直接运行 `hello.py` 了，比如在Mac下运行：



```
Last login: Sun Dec 15 18:44:13 on ttys000
Mavericks:- michael$ ./hello.py
hello, world
Mavericks:- michael$
```

小结

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

Python的交互模式和直接运行 `.py` 文件有什么区别呢？

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

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

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

参考源码

[hello.py](#)

代码运行助手

Python代码运行助手可以让你在线输入Python代码，然后通过本机运行的一个Python脚本来执行代码。原理如下：

- 在网页输入代码：

```
# 测试代码:  
print('Hello, world')
```

▶ Run

- 点击 Run 按钮，代码被发送到本机正在运行的Python代码运行助手；
- Python代码运行助手将代码保存为临时文件，然后调用Python解释器执行代码；
- 网页显示代码执行结果：

```
# 测试代码:  
print('Hello, world')
```

▶ Run

Hello, world

下载

点击右键，目标另存为：[learning.py](#)

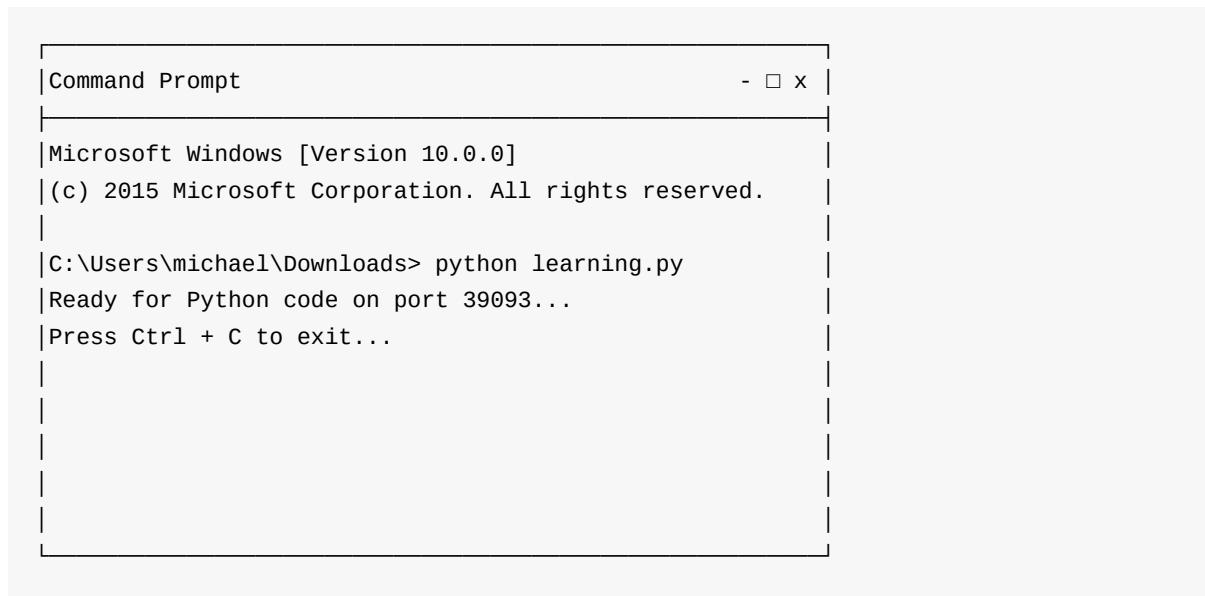
备用下载地址：[learning.py](#)

运行

在存放 `learning.py` 的目录下运行命令：

```
C:\Users\michael\Downloads> python learning.py
```

如果看到 Ready for Python code on port 39093... 表示运行成功，不要关闭命令行窗口，最小化放到后台运行即可：



The screenshot shows a Windows Command Prompt window. The title bar says "Command Prompt". The window content is as follows:

```
Microsoft Windows [Version 10.0.0]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\michael\Downloads> python learning.py
Ready for Python code on port 39093...
Press Ctrl + C to exit...
```

试试效果

需要支持HTML5的浏览器：

- IE >= 9
- Firefox
- Chrome
- Sarafi

输入和输出

输出

用 `print()` 在括号中加上字符串，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()` 函数也可以接受多个字符串，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，如果要让用户从电脑输入一些字符怎么办？Python提供了一个 `input()`，可以让用户输入字符串，并存放到一个变量里。比如输入用户名：

```
>>> name = input()
Michael
```

当你输入 `name = 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 × 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 × a = 2 × 2 = 4`；

若`a=3.5`，则面积为`a × a = 3.5 × 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()` 函数：

```
>>> 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 = input()
print('hello,', name)
```

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

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

但是程序运行的时候，没有任何提示信息告诉用户：“嘿，赶紧输入你的名字”，这样显得很不友好。幸好，`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 = 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。

`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。

练习

请利用 `print()` 输出 `1024 * 768 = xxx`：

```
# -*- coding: utf-8 -*-
print("1024 * 768 =", 1024*768)
```

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。

小结

Python使用缩进来组织代码块，请务必遵守约定俗成的习惯，坚持使用4个空格的缩进。

在文本编辑器中，需要设置把Tab自动转换为4个空格，确保不混用Tab和空格。

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， 1.23×10^9 和 12.3×10^8 是完全相等的。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， 1.23×10^9 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

字符串是以单引号'或双引号"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，注意在输入多行内容时，提示符由 >>> 变为 ...，提示你可以接着上一行输入，注意 ... 是提示符，不是代码的一部分：

```
|Command Prompt - python
|_
|>>> print('''line1
|... line2
|... line3''')
|line1
|line2
|line3
|
|>>> _
|
```

当输入完结束符 ‘‘’ 和括号) 后，执行该语句并打印结果。

如果写成程序并存为 .py 文件，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：

```
# -*- coding: utf-8 -*-
print(r'''hello,\nworld''')
```

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，要是是 True ，要是是 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
>>> 5 > 3 and 3 > 1
True
```

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
>>> 5 > 3 or 1 > 3
True
```

`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
>>> not 1 > 2
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：

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

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

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

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

和静态语言相比，动态语言更灵活，就是这个原因。

请不要把赋值语句的等号等同于数学的等号。比如下面的代码：

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

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

最后，理解变量在计算机内存中的表示也非常重要。当我们写：

```
a = 'ABC'
```

时，Python解释器干了两件事情：

1. 在内存中创建了一个 '`ABC`' 的字符串；
2. 在内存中创建了一个名为 `a` 的变量，并把它指向 '`ABC`'。

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

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

a = 'ABC'
```

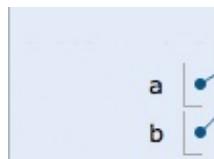
```
b = a
a = 'XYZ'
print(b)
```

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：

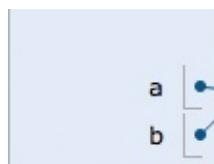
执行 `a = 'ABC'`，解释器创建了字符串 '`ABC`' 和变量 `a`，并把 `a` 指向 '`ABC`'：



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 `π` 就是一个常量。在 Python 中，通常用全部大写的变量名表示常量：

```
PI = 3.14159265359
```

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

最后解释一下整数的除法为什么也是精确的。在 Python 中，有两种除法，一种除法是 `/`：

```
>>> 10 / 3
3.3333333333333335
```

`/` 除法计算结果是浮点数，即使是两个整数恰好整除，结果也是浮点数：

```
>>> 9 / 3
```

3.0

还有一种除法是 `//`，称为地板除，两个整数的除法仍然是整数：

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

你没有看错，整数的地板除 `//` 永远是整数，即使除不尽。要做精确的除法，使用 `/` 就可以。

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

```
>>> 10 % 3
1
```

无论整数做 `//` 除法还是取余数，结果永远是整数，所以，整数运算结果永远是精确的。

练习

请打印出以下变量的值：

```
# -*- coding: utf-8 -*-
n = 123
f = 456.789
s1 = 'Hello, world'
s2 = 'Hello, \'Adam\''
s3 = r'Hello, "Bart"'
s4 = r'''Hello,
Lisa!''''

print(n)
print(f)
print(s1)
print(s2)
print(s3)
print(s4)
```

运行结果

```
123
456.789
Hello, world
Hello, 'Adam'
Hello, "Bart"
Hello,
Lisa!
```

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。

对变量赋值 `x = y` 是把变量 `x` 指向真正的对象，该对象是变量 `y` 所指向的。随后对变量 `y` 的赋值不影响变量 `x` 的指向。

注意：Python的整数没有大小限制，而某些语言的整数根据其存储长度是有大小限制的，例如Java对32位整数的范围限制在 `-2147483648 - 2147483647`。

Python的浮点数也没有大小限制，但是超出一定范围就直接表示为 `inf`（无限大）。

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个字符被编码到计算机里，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，而且不能和ASCII编码冲突，所以，中国制定了 GB2312 编码，用来把中文编进去。

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



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

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

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

字母 A 用ASCII编码是十进制的 65，二进制的 01000001；

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

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

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

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

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

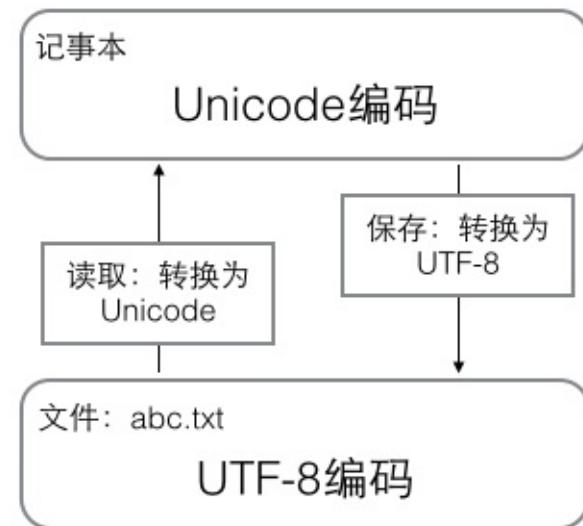
字符	ASCII	Unicode	UTF-8
A	01000001	00000000 01000001	01000001
中	x	01001110 00101101	11100100 10111000 10101101

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

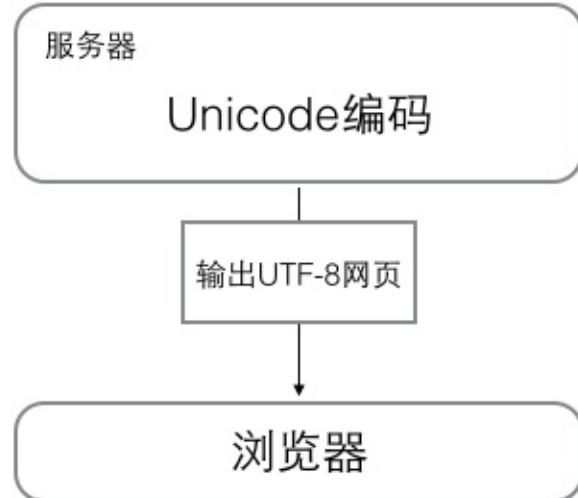
搞清楚了ASCII、Unicode和UTF-8的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

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

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



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



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

Python的字符串

搞清楚了令人头疼的字符编码问题后，我们再来研究 Python 的字符串。

在最新的 Python 3 版本中，字符串是以 Unicode 编码的，也就是说，Python 的字符串支持多语言，例如：

```

>>> print('包含中文的str')
包含中文的str
  
```

对于单个字符的编码，Python 提供了 `ord()` 函数获取字符的整数表示，`chr()` 函数把编码转换为对应的字符：

```

>>> ord('A')
65
>>> ord('中')
20013
>>> chr(66)
'B'
>>> chr(25991)
'文'
  
```

如果知道字符的整数编码，还可以用十六进制这么写 `str`：

```

>>> '\u4e2d\u6587'
'中文'
  
```

两种写法完全是等价的。

由于Python的字符串类型是 `str`，在内存中以Unicode表示，一个字符对应若干个字节。如果要在网络上传输，或者保存到磁盘上，就需要把 `str` 变为以字节为单位的 `bytes`。

Python对 `bytes` 类型的数据用带 `b` 前缀的单引号或双引号表示：

```
x = b'ABC'
```

要注意区分 `'ABC'` 和 `b'ABC'`，前者是 `str`，后者虽然内容显示得和前者一样，但 `bytes` 的每个字符都只占用一个字节。

以Unicode表示的 `str` 通过 `encode()` 方法可以编码为指定的 `bytes`，例如：

```
>>> 'ABC'.encode('ascii')
b'ABC'
>>> '中文'.encode('utf-8')
b'\xe4\xb8\xad\xe6\x96\x87'
>>> '中文'.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1: ordinal not
|
```

纯英文的 `str` 可以用 ASCII 编码为 `bytes`，内容是一样的，含有中文的 `str` 可以用 UTF-8 编码为 `bytes`。含有中文的 `str` 无法用 ASCII 编码，因为中文编码的范围超过了 ASCII 编码的范围，Python会报错。

在 `bytes` 中，无法显示为ASCII字符的字节，用 `\x##` 显示。

反过来，如果我们从网络或磁盘上读取了字节流，那么读到的数据就是 `bytes`。要把 `bytes` 变为 `str`，就需要用 `decode()` 方法：

```
>>> b'ABC'.decode('ascii')
'ABC'
>>> b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
'中文'
```

如果 `bytes` 中包含无法解码的字节，`decode()` 方法会报错：

```
>>> b'\xe4\xb8\xad\xff'.decode('utf-8')
Traceback (most recent call last):
...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 3: invalid start
|
```

如果 `bytes` 中只有一小部分无效的字节，可以传入 `errors='ignore'` 忽略错误的字节：

```
>>> b'\xe4\xb8\xad\xff'.decode('utf-8', errors='ignore')
```

'中'

要计算 `str` 包含多少个字符，可以用 `len()` 函数：

```
>>> len('ABC')
3
>>> len('中文')
2
```

`len()` 函数计算的是 `str` 的字符数，如果换成 `bytes`，`len()` 函数就计算字节数：

```
>>> len(b'ABC')
3
>>> len(b'\xe4\xb8\xad\xe6\x96\x87')
6
>>> len('中文'.encode('utf-8'))
6
```

可见，1个中文字符经过UTF-8编码后通常会占用3个字节，而1个英文字母只占用1个字节。

在操作字符串时，我们经常遇到 `str` 和 `bytes` 的互相转换。为了避免乱码问题，应当始终坚持使用 UTF-8 编码对 `str` 和 `bytes` 进行转换。

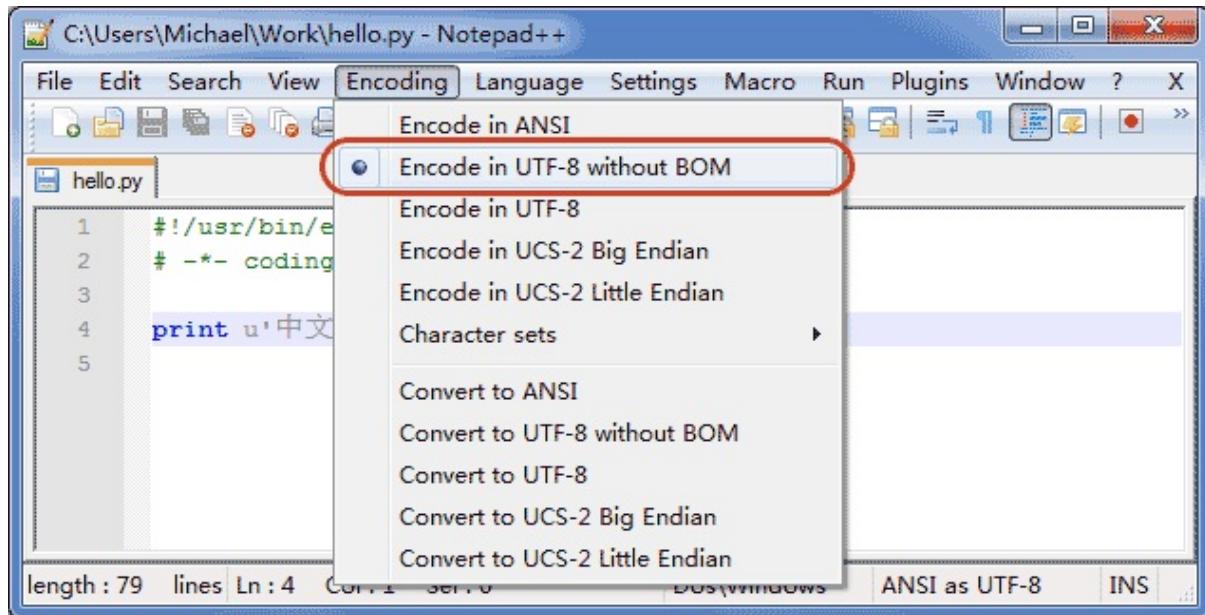
由于Python源代码也是一个文本文件，所以，当你的源代码中包含中文的时候，在保存源代码时，就需要务必指定保存为UTF-8编码。当Python解释器读取源代码时，为了让它按UTF-8编码读取，我们通常在文件开头写上这两行：

```
#!/usr/bin/env python3
# -*- 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。

申明了UTF-8编码并不意味着你的 `.py` 文件就是UTF-8编码的，必须并且要确保文本编辑器正在使用 UTF-8 without BOM 编码：



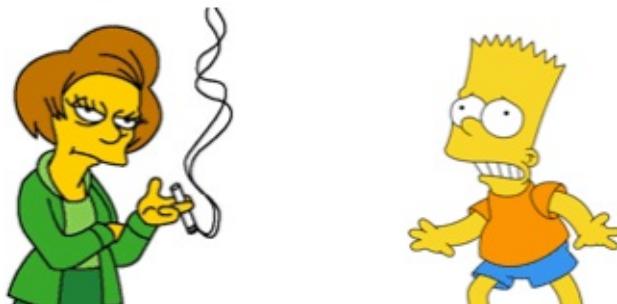
如果 .py 文件本身使用UTF-8编码，并且也申明了 `# -*- coding: utf-8 -*-`，打开命令提示符测试就可以正常显示中文：



格式化

最后一个常见的问题是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。

'Hi %s, your score is %d.' % ('Bart', 59)



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：

占位符	替换内容
<code>%d</code>	整数
<code>%f</code>	浮点数
<code>%s</code>	字符串
<code>%x</code>	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：

```
# -*- coding: utf-8 -*-
print('%2d-%02d' % (3, 1))
print('%.2f' % 3.1415926)
```

运行结果

```
3-01
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'
```

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 %'
```

format()

另一种格式化字符串的方法是使用字符串的 `format()` 方法，它会用传入的参数依次替换字符串内的占位符 `{0}`、`{1}`……，不过这种方式写起来比`%`要麻烦得多：

```
>>> 'Hello, {0}, 成绩提升了 {1:.1f}%' .format('小明', 17.125)
'Hello, 小明, 成绩提升了 17.1%'
```

练习

小明的成绩从去年的72分提升到了今年的85分，请计算小明成绩提升的百分点，并用字符串格式化显示出`'xx.x%'`，只保留小数点后1位：

```
# -*- coding: utf-8 -*-
s1 = 72
s2 = 85

r = (s2 - s1) / s1 * 100
print('%.1f%%' % r)
```

小结

Python 3的字符串使用Unicode，直接支持多语言。

当 `str` 和 `bytes` 互相转换时，需要指定编码。最常用的编码是 `UTF-8`。Python当然也支持其他编码方式，比如把Unicode编码成 `GB2312`：

```
>>> '中文'.encode('gb2312')
b'\xd6\xd0\xce\xc4'
```

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

格式化字符串的时候，可以用Python的交互式环境测试，方便快捷。

参考源码

the_string.py

使用list和tuple

list

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

比如，列出班里所有同学的名字，就可以用一个list表示：

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

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

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

用索引来访问list中每一个位置的元素，记得索引是从 0 开始的：

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

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

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

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

以此类推，可以获取倒数第2个、倒数第3个：

```
>>> classmates[-2]
'Bob'
>>> classmates[-3]
'Michael'
>>> classmates[-4]
```

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

当然，倒数第4个就越界了。

list是一个可变的有序表，所以，可以往list中追加元素到末尾：

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

也可以把元素插入到指定的位置，比如索引号为 1 的位置：

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

要删除list末尾的元素，用 `pop()` 方法：

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

要删除指定位置的元素，用 `pop(i)` 方法，其中 `i` 是索引位置：

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

要把某个元素替换成别的元素，可以直接赋值给对应的索引位置：

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

list里面的元素的数据类型也可以不同，比如：

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

list元素也可以是另一个list，比如：

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

```
>>> len(s)
4
```

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

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

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

如果一个list中一个元素也没有，就是一个空的list，它的长度为0：

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

tuple

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

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

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

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

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

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

如果要定义一个空的tuple，可以写成 ()：

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

但是，要定义一个只有1个元素的tuple，如果你这么定义：

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

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

所以，只有1个元素的tuple定义时必须加一个逗号 , ，来消除歧义：

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

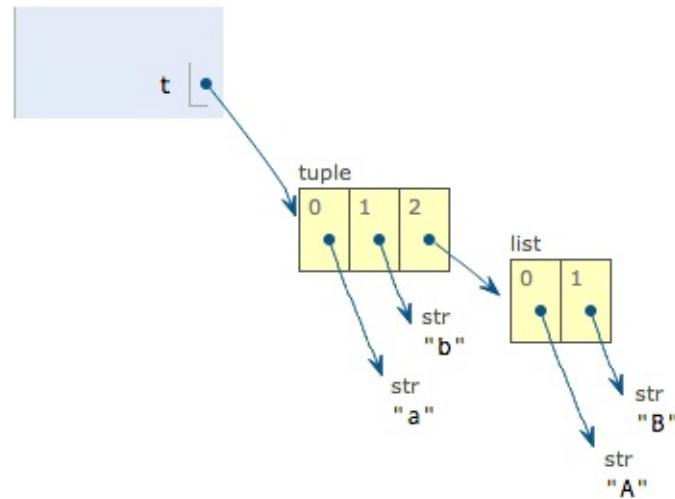
Python在显示只有1个元素的tuple时，也会加一个逗号 , ，以免你误解成数学计算意义上的括号。

最后来看一个“可变的”tuple：

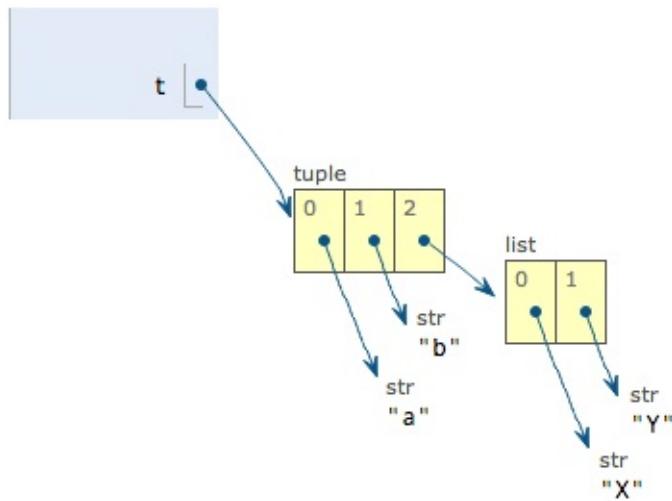
```
>>> t = ('a', 'b', ['A', 'B'])
>>> t[2][0] = 'X'
>>> t[2][1] = 'Y'
>>> t
('a', 'b', ['X', 'Y'])
```

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

别急，我们先看看定义的时候tuple包含的3个元素：



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



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

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

练习

请用索引取出下面list的指定元素：

```

# -*- coding: utf-8 -*-
L = [
    ['Apple', 'Google', 'Microsoft'],
    ['Java', 'Python', 'Ruby', 'PHP'],
    ['Adam', 'Bart', 'Lisa']
]

# 打印Apple:
print(L[0][0])
# 打印Python:
print(L[1][1])
# 打印Lisa:
print(L[2][2])
  
```

运行结果

```

Apple
Python
Lisa
  
```

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。

参考源码

[the_list.py](#)

[the_tuple.py](#)

条件判断

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`。

再议 `input`

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

```
birth = input('birth: ')
if birth < 2000:
    print('00前')
else:
    print('00后')
```

输入 `1982`，结果报错：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() > int()
```

这是因为 `input()` 返回的数据类型是 `str`，`str` 不能直接和整数比较，必须先把 `str` 转换成整数。Python提供了 `int()` 函数来完成这件事情：

```
s = input('birth: ')
birth = int(s)
if birth < 2000:
```

```

    print('00前')
else:
    print('00后')

```

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

```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'abc'

```

原来 `int()` 函数发现一个字符串并不是合法的数字时就会报错，程序就退出了。

如何检查并捕获程序运行期的错误呢？后面的错误和调试会讲到。

练习

小明身高1.75，体重80.5kg。请根据BMI公式（体重除以身高的平方）帮小明计算他的BMI指数，并根据BMI指数：

- 低于18.5：过轻
- 18.5-25：正常
- 25-28：过重
- 28-32：肥胖
- 高于32：严重肥胖

用 `if-elif` 判断并打印结果：

```

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

height = 1.75
weight = 80.5

bmi = weight / (height ** 2)
if bmi < 18.5:
    print('过轻')
elif bmi < 25:
    print('正常')
elif bmi < 28:
    print('过重')
elif bmi < 32:
    print('肥胖')
else:
    print('严重肥胖')

```

运行结果

过重

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。

条件判断从上向下匹配，当满足条件时执行对应的块内语句，后续的elif和else都不再执行。

`if salary >= 10000:`

`print` 

`elif salary >=5000:`

`print` 

`else:`

`print` 

参考源码

[do_if.py](#)

循环

循环

要计算 $1+2+3$ ，我们可以直接写表达式：

```
>>> 1 + 2 + 3
6
```

要计算 $1+2+3+\dots+10$ ，勉强也能写出来。

但是，要计算 $1+2+3+\dots+10000$ ，直接写表达式就不可能了。

为了让计算机能计算成千上万次的重复运算，我们就需要循环语句。

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，再通过 `list()` 函数可以转换为list。比如 `range(5)` 生成的序列是从 0 开始小于 5 的整数：

```
>>> list(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：

```
# -*- coding: utf-8 -*-
sum = 0
for x in range(101):
    sum = sum + x
print(sum)
```

运行结果

```
5050
```

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。

练习

请利用循环依次对list中的每个名字打印出 Hello, xxx! :

```
# -*- coding: utf-8 -*-
L = ['Bart', 'Lisa', 'Adam']
for name in L:
    print('Hello, %s!' % name)
```

运行结果

```
Hello, Bart!
Hello, Lisa!
Hello, Adam!
```

break

在循环中， break 语句可以提前退出循环。例如，本来要循环打印1~100的数字：

```
n = 1
while n <= 100:
    print(n)
    n = n + 1
print('END')
```

上面的代码可以打印出1~100。

如果要提前结束循环，可以用 `break` 语句：

```
n = 1
while n <= 100:
    if n > 10: # 当n = 11时，条件满足，执行break语句
        break # break语句会结束当前循环
    print(n)
    n = n + 1
print('END')
```

执行上面的代码可以看到，打印出1~10后，紧接着打印 `END`，程序结束。

可见 `break` 的作用是提前结束循环。

continue

在循环过程中，也可以通过 `continue` 语句，跳过当前的这次循环，直接开始下一次循环。

```
n = 0
while n < 10:
    n = n + 1
    print(n)
```

上面的程序可以打印出1~10。但是，如果我们想只打印奇数，可以用 `continue` 语句跳过某些循环：

```
n = 0
while n < 10:
    n = n + 1
    if n % 2 == 0: # 如果n是偶数，执行continue语句
        continue # continue语句会直接继续下一轮循环，后续的print()语句不会执行
    print(n)
```

执行上面的代码可以看到，打印的不再是1~10，而是1, 3, 5, 7, 9。

可见 `continue` 的作用是提前结束本轮循环，并直接开始下一轮循环。

小结

循环是让计算机做重复任务的有效方法。

`break` 语句可以在循环过程中直接退出循环，而 `continue` 语句可以提前结束本轮循环，并直接开始下一轮循环。这两个语句通常都必须配合 `if` 语句使用。

要特别注意，不要滥用 `break` 和 `continue` 语句。`break` 和 `continue` 会造成代码执行逻辑分叉过多，容易出错。大多数循环并不需要用到 `break` 和 `continue` 语句，上面的两个例子，都可以通过改写循环条件或者修改循环逻辑，去掉 `break` 和 `continue` 语句。

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。

参考源码

[do_for.py](#)

[do_while.py](#)

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的交互环境不显示结果。

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

```

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

```

请务必注意， dict内部存放的顺序和key放入的顺序是没有关系的。

和list比较， dict有以下几个特点：

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

而list相反：

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

所以， dict是用空间来换取时间的一种方法。

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

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

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

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

set

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

要创建一个set，需要提供一个list作为输入集合：

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

注意，传入的参数 [1, 2, 3] 是一个list，而显示的 {1, 2, 3} 只是告诉你这个set内部有1, 2, 3这三个元素，显示的顺序也不表示set是有序的。。

重复元素在set中自动被过滤：

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

通过 remove(key) 方法可以删除元素：

```
>>> s.remove(4)
>>> s
{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
{2, 3}
>>> s1 | s2
{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：

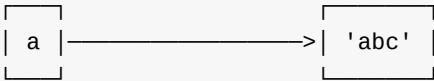
```
>>> a = 'abc'
>>> a.replace('a', 'A')
'Abc'
>>> a
'abc'
```

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？

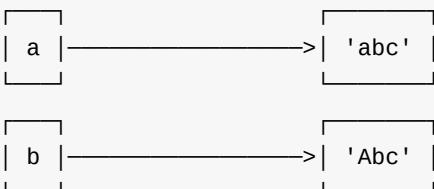
我们先把代码改成下面这样：

```
>>> 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。

参考源码

[the_dict.py](#)

[the_set.py](#)

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 = area_of_circle(x)`，而函数 `area_of_circle` 本身只需要写一次，就可以多次调用。

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

抽象

抽象是数学中非常常见的概念。举个例子：

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

$$\sum_{n=1}^{100} n$$

这种抽象记法非常强大，因为我们看到 Σ 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：

$$\sum_{n=1}^{100} (n^2 + 1)$$

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/3/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 `max` 函数 `max()` 可以接收任意多个参数，并返回最大的那个：

```
>>> max(1, 2)
2
>>> max(2, 3, 1, -5)
3
```

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'
>>> str(100)
'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
```

练习

请利用Python内置的 `hex()` 函数把一个整数转换成十六进制表示的字符串：

```
# -*- coding: utf-8 -*-
n1 = 255
n2 = 1000

print(hex(n1))
print(hex(n2))
```

运行结果

```
0xff
0x3e8
```

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！

参考源码

[call_func.py](#)

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：

```
# -*- coding: utf-8 -*-
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x

print(my_abs(-99))
```

运行结果

```
99
```

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`。

在Python交互环境中定义函数时，注意Python会出现 `...` 的提示。函数定义结束后需要按两次回车重新回到 `>>>` 提示符下：

```
| Command Prompt - python
|>>> def my_abs(x):
|...     if x >= 0:
|...         return x
|...     else:
|...         return -x
|...
|>>> my_abs(-9)
|9
|>>> _
```

如果你已经把 `my_abs()` 的函数定义保存为 `abstest.py` 文件了，那么，可以在该文件的当前目录下启动Python解释器，用 `from abstest import my_abs` 来导入 `my_abs()` 函数，注意 `abstest` 是文件名（不含 `.py` 扩展名）：

```
| Command Prompt - python
| - □ x
|>>> from abstest import my_abs
|>>> my_abs(-9)
| 9
|>>> _
```

`import` 的用法在后续[模块](#)一节中会详细介绍。

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 1 positional argument but 2 were 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')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 2, in my_abs
      TypeError: unorderable types: str() >= int()
>>> 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，会导致 `if` 语句出错，出错信息和 `abs` 不一样。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
```

`import math` 语句表示导入 `math` 包，并允许后续代码引用 `math` 包里的 `sin`、`cos` 等函数。

然后，我们就可以同时获得返回值：

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

但其实这只是一个假象，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。

练习

请定义一个函数 `quadratic(a, b, c)`，接收3个参数，返回一元二次方程：

$$ax^2 + bx + c = 0$$

的两个解。

提示：计算平方根可以调用 `math.sqrt()` 函数：

```
>>> import math
>>> math.sqrt(2)
```

```
1.4142135623730951
```

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

import math

def quadratic(a, b, c):
    if a == 0:
        return ('非一元二次方程式')
    else:
        r = b*b - 4*a*c
        if r < 0:
            return ('无实根')
        else:
            x1 = (-b + math.sqrt(r))/(2*a)
            x2 = (-b - math.sqrt(r))/(2*a)
            return x1, x2

# 测试:
print('quadratic(2, 3, 1) =', quadratic(2, 3, 1))
print('quadratic(1, 3, -4) =', quadratic(1, 3, -4))

if quadratic(2, 3, 1) != (-0.5, -1.0):
    print('测试失败')
elif quadratic(1, 3, -4) != (1.0, -4.0):
    print('测试失败')
else:
    print('测试成功')
```

参考源码

[def_func.py](#)

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。

位置参数

我们先写一个计算 x^2 的函数：

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

对于 `power(x)` 函数，参数 `x` 就是一个位置参数。

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(x, n)` 函数，可以计算任意n次方：

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

修改后的 `power(x, n)` 函数有两个参数：`x` 和 `n`，这两个参数都是位置参数，调用函数时，传入的两个值按照位置顺序依次赋给参数 `x` 和 `n`。

默认参数

新的 `power(x, n)` 函数定义没有问题，但是，旧的调用代码失败了，原因是我们在增加了一个参数，导致旧的代码因为缺少一个参数而无法正常调用：

```
>>> power(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() missing 1 required positional argument: 'n'
```

Python的错误信息很明确：调用函数 `power()` 缺少了一个位置参数 `n`。

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

```
def enroll(name, gender):
    print('name:', name)
    print('gender:', gender)
```

这样，调用 `enroll()` 函数只需要传入两个参数：

```
>>> enroll('Sarah', 'F')
name: Sarah
gender: F
```

如果要继续传入年龄、城市等信息怎么办？这样会使得调用函数的复杂度大大增加。

我们可以把年龄和城市设为默认参数：

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

这样，大多数学生注册时不需要提供年龄和城市，只提供必须的两个参数：

```
>>> enroll('Sarah', 'F')
name: Sarah
gender: F
age: 6
city: Beijing
```

只有与默认参数不符的学生才需要提供额外的信息：

```
enroll('Bob', 'M', 7)
enroll('Adam', 'M', city='Tianjin')
```

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

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

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

默认参数很有用，但使用不当，也会掉坑里。默认参数有个最大的坑，演示如下：

先定义一个函数，传入一个list，添加一个END再返回：

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

当你正常调用时，结果似乎不错：

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

当你使用默认参数调用时，一开始结果也是对的：

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

但是，再次调用 `add_end()` 时，结果就不对了：

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

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

原因解释如下：

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

定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，我们可以用 `None` 这个不变对象来实现：

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

现在，无论调用多少次，都不会有问题：

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

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

可变参数

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

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

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一个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
```

`*nums` 表示把`nums`这个list的所有元素作为可变参数传进去。这种写法相当有用，而且很常见。

关键字参数

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

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

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

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

也可以传入任意个数的关键字参数：

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

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

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

```
>>> extra = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, city=extra['city'], job=extra['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：

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

`**extra` 表示把 `extra` 这个dict的所有key-value用关键字参数传入到函数的 `**kw` 参数，`kw` 将获得一个dict，注意 `kw` 获得的dict是 `extra` 的一份拷贝，对 `kw` 的改动不会影响到函数外的 `extra`。

命名关键字参数

对于关键字参数，函数的调用者可以传入任意不受限制的关键字参数。至于到底传入了哪些，就需要在函数内部通过 `kw` 检查。

仍以 `person()` 函数为例，我们希望检查是否有 `city` 和 `job` 参数：

```
def person(name, age, **kw):
    if 'city' in kw:
        # 有city参数
        pass
    if 'job' in kw:
        # 有job参数
        pass
    print('name:', name, 'age:', age, 'other:', kw)
```

但是调用者仍可以传入不受限制的关键字参数：

```
>>> person('Jack', 24, city='Beijing', addr='Chaoyang', zipcode=123456)
```

如果要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收 `city` 和 `job` 作为关键字参数。这种方式定义的函数如下：

```
def person(name, age, *, city, job):
```

```
print(name, age, city, job)
```

和关键字参数 `**kw` 不同，命名关键字参数需要一个特殊分隔符 `*`，`*` 后面的参数被视为命名关键字参数。

调用方式如下：

```
>>> person('Jack', 24, city='Beijing', job='Engineer')
Jack 24 Beijing Engineer
```

如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符 `*` 了：

```
def person(name, age, *args, city, job):
    print(name, age, args, city, job)
```

命名关键字参数必须传入参数名，这和位置参数不同。如果没有传入参数名，调用将报错：

```
>>> person('Jack', 24, 'Beijing', 'Engineer')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: person() takes 2 positional arguments but 4 were given
```

由于调用时缺少参数名 `city` 和 `job`，Python 解释器把这 4 个参数均视为位置参数，但 `person()` 函数仅接受 2 个位置参数。

命名关键字参数可以有缺省值，从而简化调用：

```
def person(name, age, *, city='Beijing', job):
    print(name, age, city, job)
```

由于命名关键字参数 `city` 具有默认值，调用时，可不传入 `city` 参数：

```
>>> person('Jack', 24, job='Engineer')
Jack 24 Beijing Engineer
```

使用命名关键字参数时，要特别注意，如果没有可变参数，就必须加一个 `*` 作为特殊分隔符。如果缺少 `*`，Python 解释器将无法识别位置参数和命名关键字参数：

```
def person(name, age, city, job):
    # 缺少 *, city 和 job 被视为位置参数
    pass
```

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、关键字参数和命名关键字参数，这5种参数都可以组合使用。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、命名关键字参数和关键字参数。

比如定义一个函数，包含上述若干种参数：

```
def f1(a, b, c=0, *args, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)

def f2(a, b, c=0, *, d, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'd =', d, '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。

```
>>> f1(1, 2)
a = 1 b = 2 c = 0 args = () kw = {}
>>> f1(1, 2, c=3)
a = 1 b = 2 c = 3 args = () kw = {}
>>> f1(1, 2, 3, 'a', 'b')
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
>>> f1(1, 2, 3, 'a', 'b', x=99)
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
>>> f2(1, 2, d=99, ext=None)
a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}
```

最神奇的是通过一个tuple和dict，你也可以调用上述函数：

```
>>> args = (1, 2, 3, 4)
>>> kw = {'d': 99, 'x': '#'}
>>> f1(*args, **kw)
a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x': '#'}
>>> args = (1, 2, 3)
>>> kw = {'d': 88, 'x': '#'}
>>> f2(*args, **kw)
a = 1 b = 2 c = 3 d = 88 kw = {'x': '#'}
```

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。

虽然可以组合多达5种参数，但不要同时使用太多的组合，否则函数接口的可理解性很差。

练习

以下函数允许计算两个数的乘积，请稍加改造，变成可接收一个或多个数并计算乘积：

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

```

```

# 原函数
# def product(x, y):
#     return x * y

# 改造后
def product(x, *numbers):
    y = 1
    for y in numbers:
        x = x * y
    return x

# 测试
print('product(5) =', product(5))
print('product(5, 6) =', product(5, 6))
print('product(5, 6, 7) =', product(5, 6, 7))
print('product(5, 6, 7, 9) =', product(5, 6, 7, 9))
if product(5) != 5:
    print('测试失败!')
elif product(5, 6) != 30:
    print('测试失败!')
elif product(5, 6, 7) != 210:
    print('测试失败!')
elif product(5, 6, 7, 9) != 1890:
    print('测试失败!')
else:
    try:
        product()
        print('测试失败!')
    except TypeError:
        print('测试成功!')

```

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，程序运行时会有逻辑错误！

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。

命名的关键字参数是为了限制调用者可以传入的参数名，同时可以提供默认值。

定义命名的关键字参数在没有可变参数的情况下不要忘了写分隔符 `*`，否则定义的将是位置参数。

参考源码

[var_args.py](#)

[kw_args.py](#)

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 * 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)
9332621544394415268169923885626670049071\
5968264381621468592963895217599993229915\
6089414639761565182862536979208272237582\
5118521091686400000000000000000000000000000000
```

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 in comparison
```

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(n, 1)

def fact_iter(num, product):
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
```

可以看到，`return fact_iter(num - 1, num * product)` 仅返回递归函数本身，`num - 1` 和 `num * product` 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(5, 1)` 的调用如下：

```
====> fact_iter(5, 1)
====> fact_iter(4, 5)
====> fact_iter(3, 20)
====> fact_iter(2, 60)
====> fact_iter(1, 120)
====> 120
```

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

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

小结

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。

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

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

练习

[汉诺塔](#)的移动可以用递归函数非常简单地实现。

请编写 `move(n, a, b, c)` 函数，它接收参数 `n`，表示3个柱子A、B、C中第1个柱子A的盘子数量，然后打印出把所有盘子从A借助B移动到C的方法，例如：

```
# -*- coding: utf-8 -*-
def move(n, a, b, c):

    if n == 1:
        print(a, '-->', c)
    else:
        move(n-1, a, c, b)
        move(1, a, b, c)
        move(n-1, b, a, c)

    # 期待输出:
    # 期待输出:
    # A --> C
    # A --> B
    # C --> B
    # A --> C
    # B --> A
    # B --> C
    # A --> C
    move(3, 'A', 'B', 'C')
```

参考源码

[recur.py](#)

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，1行代码能实现的功能，决不写5行代码。
请始终牢记，代码越少，开发效率越高。

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']
```

记住倒数第一个元素的索引是 `-1`。

切片操作十分有用。我们先创建一个0-99的数列：

```
>>> L = list(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[::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' 也可以看成是一种list，每个元素就是一个字符。因此，字符串也可以用切片操作，只是操作结果仍是字符串：

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

在很多编程语言中，针对字符串提供了很多各种截取函数（例如，substring），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。

练习

利用切片操作，实现一个trim()函数，去除字符串首尾的空格，注意不要调用str的 strip() 方法：

```
# -*- coding: utf-8 -*-
def trim(s):
    while s[:1] == ' ':
        s = s[1:]
    while s[-1:] == ' ':
        s = s[:-1]
    return s

# 测试:
if trim('hello ') != 'hello':
    print('测试失败!')
elif trim(' hello') != 'hello':
    print('测试失败!')
elif trim(' hello ') != 'hello':
    print('测试失败!')
elif trim(' hello world ') != 'hello world':
    print('测试失败!')
elif trim('') != '':
    print('测试失败!')
elif trim('    ') != '':
    print('测试失败!')
else:
    print('测试成功!')
```

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。

参考源码

[do_slice.py](#)

迭代

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

在Python中， 迭代是通过 `for ... in` 来完成的， 而很多语言比如C语言， 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C的 `for` 循环， 因为Python的 `for` 循环不仅可以用在list或tuple上， 还可以作用在其他可迭代对象上。

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

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

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

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

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
```

练习

请使用迭代查找一个list中最小和最大值，并返回一个tuple：

```
# -*- coding: utf-8 -*-
def findMinAndMax(L):
    if L == []:
        return (None, None)
    min = max = L[0]
    for x in L:
        if max <= x:
            max = x
        if min >= x:
            min = x
    return (min, max)

# 测试
if findMinAndMax([]) != (None, None):
    print('测试失败!')
elif findMinAndMax([7]) != (7, 7):
    print('测试失败!')
```

```
elif findMinAndMax([7, 1]) != (1, 7):
    print('测试失败!')
elif findMinAndMax([7, 1, 3, 9, 5]) != (1, 9):
    print('测试失败!')
else:
    print('测试成功!')
```

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。

参考源码

[do_iter.py](#)

列表生成式

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

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

```
>>> list(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 items() 可以同时迭代key和value：

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

练习

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>
    File "<stdin>", line 1, in <listcomp>
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：

```
# -*- coding: utf-8 -*-
L1 = ['Hello', 'World', 18, 'Apple', None]

L2 = [s.lower() for s in L1 if isinstance(s,str)]

# 测试:
print(L2)
if L2 == ['hello', 'world', 'apple']:
    print('测试通过!')
else:
    print('测试失败!')
```

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。

参考源码

[do_listcompr.py](#)

生成器

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有有限的。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 0x1022ef630>
```

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`next()`函数获得generator的下一个返回值：

```
>>> next(g)
0
>>> next(g)
1
>>> next(g)
4
>>> next(g)
9
>>> next(g)
16
>>> next(g)
25
>>> next(g)
36
>>> next(g)
49
>>> next(g)
64
>>> next(g)
81
>>> next(g)
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(g)`，就计算出 `g` 的下一个元素的值，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(g)` 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()`，而是通过 `for` 循环来迭代它，并且不需要关心 `StopIteration` 的错误。

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

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

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

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        print(b)
        a, b = b, a + b
        n = n + 1
    return 'done'
```

注意，赋值语句

```
a, b = b, a + b
```

相当于：

```
t = (b, a + b) # t是一个tuple
a = t[0]
b = t[1]
```

但不必显式写出临时变量t就可以赋值。

上面的函数可以输出斐波那契数列的前N个数：

```
>>> fib(6)
1
1
2
3
5
8
'done'
```

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

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

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
    return 'done'
```

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：

```
>>> f = fib(6)
>>> f
<generator object fib at 0x104fea0a0>
```

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)

```

调用该generator时，首先要生成一个generator对象，然后用 `next()` 函数不断获得下一个返回值：

```

>>> o = odd()
>>> next(o)
step 1
1
>>> next(o)
step 2
3
>>> next(o)
step 3
5
>>> next(o)
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(o)` 就报错。

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

同样的，把函数改成generator后，我们基本上从来不会用 `next()` 来获取下一个返回值，而是直接使用 `for` 循环来迭代：

```

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

```

但是用 `for` 循环调用generator时，发现拿不到generator的 `return` 语句的返回值。如果想要拿到返回值，必须捕获 `StopIteration` 错误，返回值包含在 `StopIteration` 的 `value` 中：

```

>>> g = fib(6)
>>> while True:

```

```

...
    try:
...
        x = next(g)
        print('g:', x)
    except StopIteration as e:
        print('Generator return value:', e.value)
...
g: 1
g: 1
g: 2
g: 3
g: 5
g: 8
Generator return value: done

```

关于如何捕获错误，后面的错误处理还会详细讲解。

练习

[杨辉三角](#)定义如下：

```

      1
     / \
    1   1
   / \ / \
  1   2   1
 / \ / \ / \
1   3   3   1
 / \ / \ / \ / \
1   4   6   4   1
 / \ / \ / \ / \ / \
1   5   10  10  5   1

```

把每一行看做一个list，试写一个generator，不断输出下一行的list：

```

# -*- coding: utf-8 -*-
def triangles():
    num = [1]
    while True:
        yield num
        num = [num[i] + num[i-1] for i in range(len(num)) if i != 0]
        num.append(1)
        num.insert(0, 1)
# 期待输出：
# [1]
# [1, 1]
# [1, 2, 1]
# [1, 3, 3, 1]

```

```

# [1, 4, 6, 4, 1]
# [1, 5, 10, 10, 5, 1]
# [1, 6, 15, 20, 15, 6, 1]
# [1, 7, 21, 35, 35, 21, 7, 1]
# [1, 8, 28, 56, 70, 56, 28, 8, 1]
# [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
n = 0
results = []
for t in triangles():
    print(t)
    results.append(t)
    n = n + 1
    if n == 10:
        break
if results == [
    [1],
    [1, 1],
    [1, 2, 1],
    [1, 3, 3, 1],
    [1, 4, 6, 4, 1],
    [1, 5, 10, 10, 5, 1],
    [1, 6, 15, 20, 15, 6, 1],
    [1, 7, 21, 35, 35, 21, 7, 1],
    [1, 8, 28, 56, 70, 56, 28, 8, 1],
    [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
]:
    print('测试通过!')
else:
    print('测试失败!')

```

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。

请注意区分普通函数和generator函数，普通函数调用直接返回结果：

```

>>> r = abs(6)
>>> r
6

```

generator函数的“调用”实际返回一个generator对象：

```

>>> g = fib(6)

```

```
>>> g
<generator object fib at 0x1022ef948>
```

参考源码

[do_generator.py](#)

迭代器

我们已经知道，可以直接作用于 `for` 循环的数据类型有以下几种：

一类是集合数据类型，如 `list`、`tuple`、`dict`、`set`、`str` 等；

一类是 `generator`，包括生成器和带 `yield` 的 generator function。

这些可以直接作用于 `for` 循环的对象统称为可迭代对象： `Iterable`。

可以使用 `isinstance()` 判断一个对象是否是 `Iterable` 对象：

```
>>> from collections import Iterable
>>> isinstance([], Iterable)
True
>>> isinstance({}, Iterable)
True
>>> isinstance('abc', Iterable)
True
>>> isinstance((x for x in range(10)), Iterable)
True
>>> isinstance(100, Iterable)
False
```

而生成器不但可以作用于 `for` 循环，还可以被 `next()` 函数不断调用并返回下一个值，直到最后抛出 `StopIteration` 错误表示无法继续返回下一个值了。

可以被 `next()` 函数调用并不断返回下一个值的对象称为迭代器：`Iterator`。

可以使用 `isinstance()` 判断一个对象是否是 `Iterator` 对象：

```
>>> from collections import Iterator
>>> isinstance((x for x in range(10)), Iterator)
True
>>> isinstance([], Iterator)
False
>>> isinstance({}, Iterator)
False
>>> isinstance('abc', Iterator)
False
```

生成器都是 `Iterator` 对象，但 `list`、`dict`、`str` 虽然是 `Iterable`，却不是 `Iterator`。

把 `list`、`dict`、`str` 等 `Iterable` 变成 `Iterator` 可以使用 `iter()` 函数：

```
>>> isinstance(iter([]), Iterator)
True
>>> isinstance(iter('abc'), Iterator)
```

```
True
```

你可能会问，为什么 `list`、`dict`、`str` 等数据类型不是 `Iterator`？

这是因为Python的 `Iterator` 对象表示的是一个数据流，`Iterator`对象可以被 `next()` 函数调用并不断返回下一个数据，直到没有数据时抛出 `StopIteration` 错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过 `next()` 函数实现按需计算下一个数据，所以 `Iterator` 的计算是惰性的，只有在需要返回下一个数据时它才会计算。

`Iterator` 甚至可以表示一个无限大的数据流，例如全体自然数。而使用`list`是永远不可能存储全体自然数的。

小结

凡是可作用于 `for` 循环的对象都是 `Iterable` 类型；

凡是可作用于 `next()` 函数的对象都是 `Iterator` 类型，它们表示一个惰性计算的序列；

集合数据类型如 `list`、`dict`、`str` 等是 `Iterable` 但不是 `Iterator`，不过可以通过 `iter()` 函数获得一个 `Iterator` 对象。

Python的 `for` 循环本质上就是通过不断调用 `next()` 函数实现的，例如：

```
for x in [1, 2, 3, 4, 5]:
    pass
```

实际上完全等价于：

```
# 首先获得Iterator对象:
it = iter([1, 2, 3, 4, 5])
# 循环:
while True:
    try:
        # 获得下一个值:
        x = next(it)
    except StopIteration:
        # 遇到StopIteration就退出循环
        break
```

参考源码

[do_iter.py](#)

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

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

变量可以指向函数

以Python内置的求绝对值的函数 `abs()` 为例，调用该函数用以下代码：

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

但是，如果只写 `abs` 呢？

```
>>> abs
<built-in function abs>
```

可见，`abs(-10)` 是函数调用，而 `abs` 是函数本身。

要获得函数调用结果，我们可以把结果赋值给变量：

```
>>> x = abs(-10)
>>> x
10
```

但是，如果把函数本身赋值给变量呢？

```
>>> f = abs
>>> f
<built-in function abs>
```

结论：函数本身也可以赋值给变量，即：变量可以指向函数。

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

```
>>> f = abs
>>> f(-10)
10
```

成功！说明变量 `f` 现在已经指向了 `abs` 函数本身。直接调用 `abs()` 函数和调用变量 `f()` 完全相同。

函数名也是变量

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

如果把 `abs` 指向其他对象，会有什么情况发生？

```
>>> abs = 10
>>> abs(-10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

把 `abs` 指向 `10` 后，就无法通过 `abs(-10)` 调用该函数了！因为 `abs` 这个变量已经不指向求绝对值函数而是指向一个整数 `10`！

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

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

传入函数

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

一个最简单的高阶函数：

```
def add(x, y, f):
    return f(x) + f(y)
```

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

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

用代码验证一下：

```
# -*- coding: utf-8 -*-
def add(x, y, f):
    return f(x) + f(y)

print(add(-5, 6, abs))
```

编写高阶函数，就是让函数的参数能够接收别的函数。

小结

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

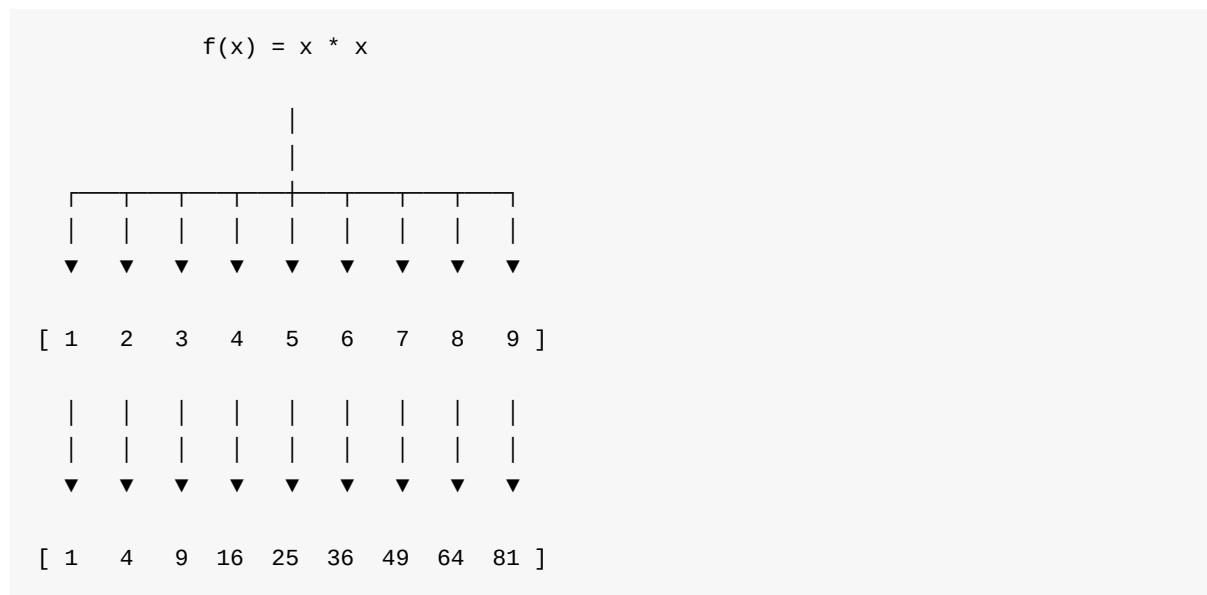
map/reduce

Python内建了 `map()` 和 `reduce()` 函数。

如果你读过Google的那篇大名鼎鼎的论文“[MapReduce: Simplified Data Processing on Large Clusters](#)”，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 `Iterable`，`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 `Iterator` 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
...
>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> list(r)
[1, 4, 9, 16, 25, 36, 49, 64, 81]

```

`map()` 传入的第一个参数是 `f`，即函数对象本身。由于结果 `r` 是一个 `Iterator`，`Iterator` 是惰性序列，因此通过 `list()` 函数让它把整个序列都计算出来并返回一个list。

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 $f(x) = x^2$ ，还可以计算任意复杂的函数，比如，把这个list所有数字转为字符串：

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

只需要一行代码。

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

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

比方说对一个序列求和，就可以用`reduce`实现：

```
>>> from functools import reduce
>>> 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：

```
>>> from functools import reduce
>>> 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：

```
>>> from functools import reduce
>>> def fn(x, y):
...     return x * 10 + y
...
>>> def char2num(s):
```

```

...     digits = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8':
...     return digits[s]
...
>>> reduce(fn, map(char2num, '13579'))
13579

```

整理成一个 `str2int` 的函数就是：

```

from functools import reduce

DIGITS = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9':

def str2int(s):
    def fn(x, y):
        return x * 10 + y
    def char2num(s):
        return DIGITS[s]
    return reduce(fn, map(char2num, s))

```

还可以用`lambda`函数进一步简化成：

```

from functools import reduce

DIGITS = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9':

def char2num(s):
    return DIGITS[s]

def str2int(s):
    return reduce(lambda x, y: x * 10 + y, map(char2num, s))

```

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

`lambda`函数的用法在后面介绍。

练习

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

```

# -*- coding: utf-8 -*-
def normalize(name):
    return name[0].upper() + name[1:].lower()

```

```
# 测试:
L1 = ['adam', 'LISA', 'barT']
L2 = list(map(normalize, L1))
print(L2)
```

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

```
# -*- coding: utf-8 -*-
from functools import reduce

def prod(L):
    return reduce(lambda x, y: x * y, L)

print('3 * 5 * 7 * 9 =', prod([3, 5, 7, 9]))
if prod([3, 5, 7, 9]) == 945:
    print('测试成功!')
else:
    print('测试失败!')
```

利用 `map` 和 `reduce` 编写一个 `str2float` 函数, 把字符串 '123.456' 转换成浮点数 123.456 :

```
# -*- coding: utf-8 -*-
from functools import reduce

def str2float(s):
    list = s.split('.')
    intNum = reduce(lambda x, y: x * 10 + y,
                   map(lambda c:int(c), list[0]))
    floatNum = reduce(lambda x, y: x * 10 + y,
                   map(lambda c: int(c), list[1]))
    return intNum + floatNum * 10 ** -len(list[1])

print('str2float(\'123.456\') =', str2float('123.456'))
if abs(str2float('123.456') - 123.456) < 0.00001:
    print('测试成功!')
else:
    print('测试失败!')
```

参考代码

[do_map.py](#)

[do_reduce.py](#)

filter

Python内建的 `filter()` 函数用于过滤序列。

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

例如，在一个list中，删掉偶数，只保留奇数，可以这么写：

```
def is_odd(n):
    return n % 2 == 1

list(filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15]))
# 结果: [1, 5, 9, 15]
```

把一个序列中的空字符串删掉，可以这么写：

```
def not_empty(s):
    return s and s.strip()

list(filter(not_empty, ['A', '', 'B', None, 'C', ' ']))
# 结果: ['A', 'B', 'C']
```

可见用 `filter()` 这个高阶函数，关键在于正确实现一个“筛选”函数。

注意到 `filter()` 函数返回的是一个 `Iterator`，也就是一个惰性序列，所以要强迫 `filter()` 完成计算结果，需要用 `list()` 函数获得所有结果并返回list。

用filter求素数

计算素数的一个方法是[埃氏筛法](#)，它的算法理解起来非常简单：

首先，列出从 `2` 开始的所有自然数，构造一个序列：

`2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...`

取序列的第一个数 `2`，它一定是素数，然后用 `2` 把序列的 `2` 的倍数筛掉：

`3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...`

取新序列的第一个数 `3`，它一定是素数，然后用 `3` 把序列的 `3` 的倍数筛掉：

`5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...`

取新序列的第一个数 `5`，然后用 `5` 把序列的 `5` 的倍数筛掉：

`7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...`

不断筛下去，就可以得到所有的素数。

用Python来实现这个算法，可以先构造一个从 3 开始的奇数序列：

```
def _odd_iter():
    n = 1
    while True:
        n = n + 2
        yield n
```

注意这是一个生成器，并且是一个无限序列。

然后定义一个筛选函数：

```
def _not_divisible(n):
    return lambda x: x % n > 0
```

最后，定义一个生成器，不断返回下一个素数：

```
def primes():
    yield 2
    it = _odd_iter() # 初始序列
    while True:
        n = next(it) # 返回序列的第一个数
        yield n
        it = filter(_not_divisible(n), it) # 构造新序列
```

这个生成器先返回第一个素数 2，然后，利用 `filter()` 不断产生筛选后的新的序列。

由于 `primes()` 也是一个无限序列，所以调用时需要设置一个退出循环的条件：

```
# 打印1000以内的素数：
for n in primes():
    if n < 1000:
        print(n)
    else:
        break
```

注意到 `Iterator` 是惰性计算的序列，所以我们可以用Python表示“全体自然数”，“全体素数”这样的序列，而代码非常简洁。

练习

回数是指从左向右读和从右向左读都是一样的数，例如 12321，909。请利用 `filter()` 筛选出回数：

```
# -*- coding: utf-8 -*-
def is_palindrome(n):
```

```
return str(n) == str(n)[::-1]

# 测试:
output = filter(is_palindrome, range(1, 1000))
print('1~1000:', list(output))
if list(filter(is_palindrome, range(1, 200))) == [1, 2, 3, 4, 5, 6, 7, 8, 9,
    11, 22, 33, 44, 55, 66, 77, 88, 99,
    101, 111, 121, 131, 141, 151, 161, 171, 181, 191]:
    print('测试成功!')
else:
    print('测试失败!')
```

小结

`filter()` 的作用是从一个序列中筛选出符合条件的元素。由于 `filter()` 使用了惰性计算，所以只有在取 `filter()` 结果的时候，才会真正筛选并每次返回下一个筛选出的元素。

参考源码

[do_filter.py](#)

[prime_numbers.py](#)

sorted

排序算法

排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，我们可以直接比较，但如果是字符串或者两个dict呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。

Python内置的 `sorted()` 函数就可以对list进行排序：

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

此外，`sorted()` 函数也是一个高阶函数，它还可以接收一个 `key` 函数来实现自定义的排序，例如按绝对值大小排序：

```
>>> sorted([36, 5, -12, 9, -21], key=abs)
[5, 9, -12, -21, 36]
```

`key`指定的函数将作用于list的每一个元素上，并根据`key`函数返回的结果进行排序。对比原始的list和经过 `key=abs` 处理过的list：

```
list = [36, 5, -12, 9, -21]

keys = [36, 5, 12, 9, 21]
```

然后 `sorted()` 函数按照`keys`进行排序，并按照对应关系返回list相应的元素：

```
keys排序结果 => [5, 9, 12, 21, 36]
                  | |   |   |
最终结果      => [5, 9, -12, -21, 36]
```

我们再看一个字符串排序的例子：

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

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

现在，我们提出排序应该忽略大小写，按照字母序排序。要实现这个算法，不必对现有代码大加改动，只要我们能用一个`key`函数把字符串映射为忽略大小写排序即可。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 `key` 函数，即可实现忽略大小写的排序：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower)
['about', 'bob', 'Credit', 'Zoo']
```

要进行反向排序，不必改动 `key` 函数，可以传入第三个参数 `reverse=True`：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower, reverse=True)
['Zoo', 'Credit', 'bob', 'about']
```

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。

小结

`sorted()` 也是一个高阶函数。用 `sorted()` 排序的关键在于实现一个映射函数。

练习

假设我们用一组tuple表示学生名字和成绩：

```
L = [('Bob', 75), ('Adam', 92), ('Bart', 66), ('Lisa', 88)]
```

请用 `sorted()` 对上述列表分别按名字排序：

```
# -*- coding: utf-8 -*-
L = [('Bob', 75), ('Adam', 92), ('Bart', 66), ('Lisa', 88)]
def by_name(t):
    return t[0].lower()
L2 = sorted(L, key=by_name)
print(L2)
```

再按成绩从高到低排序：

```
# -*- coding: utf-8 -*-
L = [('Bob', 75), ('Adam', 92), ('Bart', 66), ('Lisa', 88)]
def by_score(t):
    return -t[1]
L2 = sorted(L, key=by_score)
print(L2)
```

参考源码

[do_sorted.py](#)

返回函数

函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

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

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

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

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

```
>>> f = lazy_sum(1, 3, 5, 7, 9)
>>> f
<function lazy_sum.<locals>.sum at 0x101c6ed90>
```

调用函数 `f` 时，才真正计算求和的结果：

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

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

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

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

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

`f1()` 和 `f2()` 的调用结果互不影响。

闭包

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

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

```
def count():
    fs = []
    for i in range(1, 4):
        def f():
            return i*i
        fs.append(f)
    return fs

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

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

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

```
>>> f1()
9
>>> f2()
9
>>> f3()
9
```

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

返回闭包时牢记一点：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

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

```
def count():
    def f(j):
        def g():
            return j*j
        return g
```

```

fs = []
for i in range(1, 4):
    fs.append(f(i)) # f(i)立刻被执行，因此i的当前值被传入f()
return fs

```

再看看结果：

```

>>> f1, f2, f3 = count()
>>> f1()
1
>>> f2()
4
>>> f3()
9

```

缺点是代码较长，可利用lambda函数缩短代码。

练习

利用闭包返回一个计数器函数，每次调用它返回递增整数：

```

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

def createCounter():
    def f():
        n = 0
        while True:
            n = n + 1
            yield n # 先创造一个生成器
    sun = f() # 生成器是惰性的，先执行一次
    def counter():
        return next(sun) # 用一个函数来调用生成器
    return counter

# 测试：
counterA = createCounter()
print(counterA(), counterA(), counterA(), counterA(), counterA()) # 1 2 3 4 5
counterB = createCounter()
if [counterB(), counterB(), counterB(), counterB()] == [1, 2, 3, 4]:
    print('测试通过!')
else:
    print('测试失败!')

```

小结

一个函数可以返回一个计算结果，也可以返回一个函数。

返回一个函数时，牢记该函数并未执行，返回函数中不要引用任何可能会变化的变量。

参考源码

[return_func.py](#)

匿名函数

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

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

```
>>> list(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 0x101c6ef28>
>>> 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
```

练习

请用匿名函数改造下面的代码：

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

# 改造前
def is_odd(n):
    return n % 2 == 1

L = list(filter(is_odd, range(1, 20)))
```

```
# 改造后
# L = list(filter(lambda x: x % 2 == 1, range(1, 20)))
print(L)
```

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

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('2015-3-25')
...
>>> f = now
>>> f()
2015-3-25
```

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('2015-3-25')
```

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():
2015-3-25
```

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('2015-3-25')
```

执行结果如下：

```
>>> now()
execute now():
2015-3-25
```

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。

练习

请设计一个decorator，它可作用于任何函数上，并打印该函数的执行时间：

```
# -*- coding: utf-8 -*-
import time, functools

def metric(fn):
    def wrapper(*args, **kw):
        t1 = time.time()
        rs = fn(*args, **kw)
        print('%s executed in %s ms' % (fn.__name__,
                                         1000*(time.time()-t1)))
        return rs
    return wrapper

# 测试
```

```

@metric
def fast(x, y):
    time.sleep(0.0012)
    return x + y;

@metric
def slow(x, y, z):
    time.sleep(0.1234)
    return x * y * z;

f = fast(11, 22)
s = slow(11, 22, 33)
if f != 33:
    print('测试失败!')
elif s != 7986:
    print('测试失败!')

```

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

```

参考源码

[decorator.py](#)

偏函数

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

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

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

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

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

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

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

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

这样，我们转换二进制就非常方便了：

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

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

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

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

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

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

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

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

实际上固定了 `int()` 函数的关键字参数 `base`，也就是：

```
int2('10010')
```

相当于：

```
kw = {'base': 2}
int('10010', **kw)
```

当传入：

```
max2 = functools.partial(max, 10)
```

实际上会把 `10` 作为 `*args` 的一部分自动加到左边，也就是：

```
max2(5, 6, 7)
```

相当于：

```
args = (10, 5, 6, 7)
max(*args)
```

结果为 `10`。

小结

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

参考源码

[do_partial.py](#)

模块

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

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

使用模块有什么好处？

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

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

```
mycompany
├── __init__.py
├── abc.py
└── xyz.py
```

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

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

类似的，可以有多级目录，组成多层次的包结构。比如如下的目录结构：

```
mycompany
├── web
│   ├── __init__.py
│   ├── utils.py
│   └── www.py
```

```
└─ __init__.py  
└─ abc.py  
└─ xyz.py
```

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`。

自己创建模块时要注意命名，不能和Python自带的模块名称冲突。例如，系统自带了`sys`模块，自己的模块就不可命名为`sys.py`，否则将无法导入系统自带的`sys`模块。

`mycompany.web` 也是一个模块，请指出该模块对应的.py文件。

总结

模块是一组Python代码的集合，可以使用其他模块，也可以被其他模块使用。

创建自己的模块时，要注意：

- 模块名要遵循Python变量命名规范，不要使用中文、特殊字符；
- 模块名不要和系统模块名冲突，最好先查看系统是否已存在该模块，检查方法是在Python交互环境执行 `import abc`，若成功则说明系统存在此模块。

使用模块

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

我们以内建的 `sys` 模块为例，编写一个 `hello` 的模块：

```
#!/usr/bin/env python3
# -*- 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 `python3 hello.py` 获得的 `sys.argv` 就是 `['hello.py']`；

运行 `python3 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：

```
$ python3 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：

```
$ python3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] 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!
```

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

在Python中，安装第三方模块，是通过包管理工具pip完成的。

如果你正在使用Mac或Linux，安装pip本身这个步骤就可以跳过了。

如果你正在使用Windows，请参考[安装Python](#)一节的内容，确保安装时勾选了 `pip` 和 `Add python.exe to Path`。

在命令提示符窗口下尝试运行 `pip`，如果Windows提示未找到命令，可以重新运行安装程序添加 `pip`。

注意：Mac或Linux上有可能并存Python 3.x和Python 2.x，因此对应的pip命令是 `pip3`。

例如，我们要安装一个第三方库——Python Imaging Library，这是Python下非常强大的处理图像的工具库。不过，PIL目前只支持到Python 2.7，并且有年头没有更新了，因此，基于PIL的Pillow项目开发非常活跃，并且支持最新的Python 3。

一般来说，第三方库都会在Python官方的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Pillow的名称叫[Pillow](#)，因此，安装Pillow的命令就是：

```
pip install Pillow
```

耐心等待下载并安装后，就可以使用Pillow了。

且慢！有更好的方法！



安装常用模块

在使用Python时，我们经常需要用到很多第三方库，例如，上面提到的Pillow，以及MySQL驱动程序，Web框架Flask，科学计算Numpy等。用pip一个一个安装费时费力，还需要考虑兼容性。我们推荐直接使用[Anaconda](#)，这是一个基于Python的数据处理和科学计算平台，它已经内置了许多非常有用第三方库，我们装上Anaconda，就相当于把数十个第三方模块自动安装好了，非常简单易用。

可以从[Anaconda官网](#)下载GUI安装包，安装包有500~600M，所以需要耐心等待下载。网速慢的同学请移步[国内镜像](#)。下载后直接安装，Anaconda会把系统Path中的python指向自己自带的Python，并且，Anaconda安装的第三方模块会安装在Anaconda自己的路径下，不影响系统已安装的Python目录。

安装好Anaconda后，重新打开命令行窗口，输入python，可以看到Anaconda的信息：

```
| Command Prompt - python
|
| Microsoft Windows [Version 10.0.0]
| (c) 2015 Microsoft Corporation. All rights reserved.
|
| C:\> python
| Python 3.6.3 |Anaconda, Inc.| ... on win32
|Type "help", ... for more information.
|>>> import numpy
|>>> _
```

可以尝试直接 `import numpy` 等已安装的第三方模块。

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/Frameworks/Python.framework/Versions/3.6/lib/python36.zip',
  '/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6',
  ...,
  '/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages']
```

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。

面向对象编程

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

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

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

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

我们以一个例子来说明面向过程和面向对象在程序流程上的不同之处。

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

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

而处理学生成绩可以通过函数实现，比如打印学生的成绩：

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

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

```
class Student(object):

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

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

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

```
bart = Student('Bart Simpson', 59)
lisa = Student('Lisa Simpson', 87)
bart.print_score()
```

```
lisa.print_score()
```

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

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

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

小结

数据封装、继承和多态是面向对象的三大特点，我们后面会详细讲解。

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` 的实例，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：特殊方法“`__init__`”前后分别有两个下划线！！！

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

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

```
>>> bart = Student('Bart Simpson', 59)
>>> bart.name
'Bart Simpson'
>>> bart.score
59
```

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

数据封装

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

```
>>> def print_score(std):
...     print('%s: %s' % (std.name, std.score))
...
>>> print_score(bart)
Bart Simpson: 59
```

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

```
class Student(object):

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

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

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

```
>>> bart.print_score()
Bart Simpson: 59
```

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

封装的另一个好处是可以给 `student` 类增加新的方法，比如 `get_grade`：

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

    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'
```

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

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

class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score

    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'

lisa = Student('Lisa', 99)
bart = Student('Bart', 59)
print(lisa.name, lisa.get_grade())
print(bart.name, bart.get_grade())
```

小结

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

方法就是与实例绑定的函数，和普通函数不同，方法可以直接访问实例的数据；

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

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

```
>>> bart = Student('Bart Simpson', 59)
>>> lisa = Student('Lisa Simpson', 87)
>>> bart.age = 8
>>> bart.age
8
>>> lisa.age
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'age'
```

参考源码

[student.py](#)

访问限制

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

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

```
>>> bart = Student('Bart Simpson', 59)
>>> bart.score
59
>>> bart.score = 99
>>> bart.score
99
```

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

```
class Student(object):

    def __init__(self, name, score):
        self.__name = name
        self.__score = score

    def print_score(self):
        print('%s: %s' % (self.__name, self.__score))
```

改完后，对于外部代码来说，没什么变动，但是已经无法从外部访问 实例变量.__name 和 实例变量.__score 了：

```
>>> bart = Student('Bart Simpson', 59)
>>> 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？可以再给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 = 99 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。

最后注意下面的这种错误写法：

```
>>> bart = Student('Bart Simpson', 59)
>>> bart.get_name()
'Bart Simpson'
>>> bart.__name = 'New Name' # 设置__name变量!
>>> bart.__name
'New Name'
```

表面上看，外部代码“成功”地设置了 `__name` 变量，但实际上这个 `__name` 变量和 `class` 内部的 `__name` 变量不是一个变量！内部的 `__name` 变量已经被 Python 解释器自动改成 `_Student__name`，而外部代码给 `bart` 新增了一个 `__name` 变量。不信试试：

```
>>> bart.get_name() # get_name() 内部返回 self.__name
'Bart Simpson'
```

练习

请把下面的 `Student` 对象的 `gender` 字段对外隐藏起来，用 `get_gender()` 和 `set_gender()` 代替，并检查参数有效性：

```
# -*- coding: utf-8 -*-
class Student(object):
    def __init__(self, name, gender):
        self.__name = name
        self.__gender = gender

    def get_gender(self):
        return self.__gender

    def set_gender(self, gender):
        if gender == 'male' or 'female':
            self.__gender = gender
        else:
            print('bad gender')

# 测试:
bart = Student('Bart', 'male')
if bart.get_gender() != 'male':
    print('测试失败!')
else:
    bart.set_gender('female')
    if bart.get_gender() != 'female':
        print('测试失败!')
    else:
        print('测试成功!')
```

参考源码

[protected_student.py](#)

继承和多态

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

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

```
class Animal(object):
    def run(self):
        print('Animal is running...')
```

当我们需要编写 Dog 和 cat 类时，就可以直接从 Animal 类继承：

```
class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

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

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

```
dog = Dog()
dog.run()

cat = Cat()
cat.run()
```

运行结果如下：

```
Animal is running...
Animal is running...
```

当然，也可以对子类增加一些方法，比如Dog类：

```
class Dog(Animal):

    def run(self):
        print('Dog is running...')

    def eat(self):
        print('Eating meat...')
```

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

```
class Dog(Animal):
    def run(self):
        print('Dog is running...')

class Cat(Animal):
    def run(self):
        print('Cat is running...')
```

再次运行，结果如下：

```
Dog is running...
Cat is running...
```

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

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

```
a = list() # a是list类型
b = Animal() # b是Animal类型
c = Dog() # c是Dog类型
```

判断一个变量是否是某个类型可以用 `isinstance()` 判断：

```
>>> isinstance(a, list)
True
>>> isinstance(b, Animal)
True
>>> isinstance(c, Dog)
True
```

看来 a、b、c 确实对应着 list、Animal、Dog 这 3 种类型。

但是等等，试试：

```
>>> isinstance(c, Animal)
True
```

看来 `c` 不仅仅是 `Dog`，`c` 还是 `Animal`！

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

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

```
>>> b = Animal()
>>> isinstance(b, Dog)
False
```

`Dog` 可以看成 `Animal`，但 `Animal` 不可以看成 `Dog`。

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

```
def run_twice(animal):
    animal.run()
    animal.run()
```

当我们传入 `Animal` 的实例时，`run_twice()` 就打印出：

```
>>> run_twice(Animal())
Animal is running...
Animal is running...
```

当我们传入 `Dog` 的实例时，`run_twice()` 就打印出：

```
>>> run_twice(Dog())
Dog is running...
Dog is running...
```

当我们传入 `Cat` 的实例时，`run_twice()` 就打印出：

```
>>> run_twice(Cat())
Cat is running...
Cat is running...
```

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

```
class Tortoise(Animal):
    def run(self):
        print('Tortoise is running slowly...')
```

当我们调用 `run_twice()` 时，传入 `Tortoise` 的实例：

```
>>> run_twice(Tortoise())
Tortoise is running slowly...
Tortoise is running slowly...
```

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

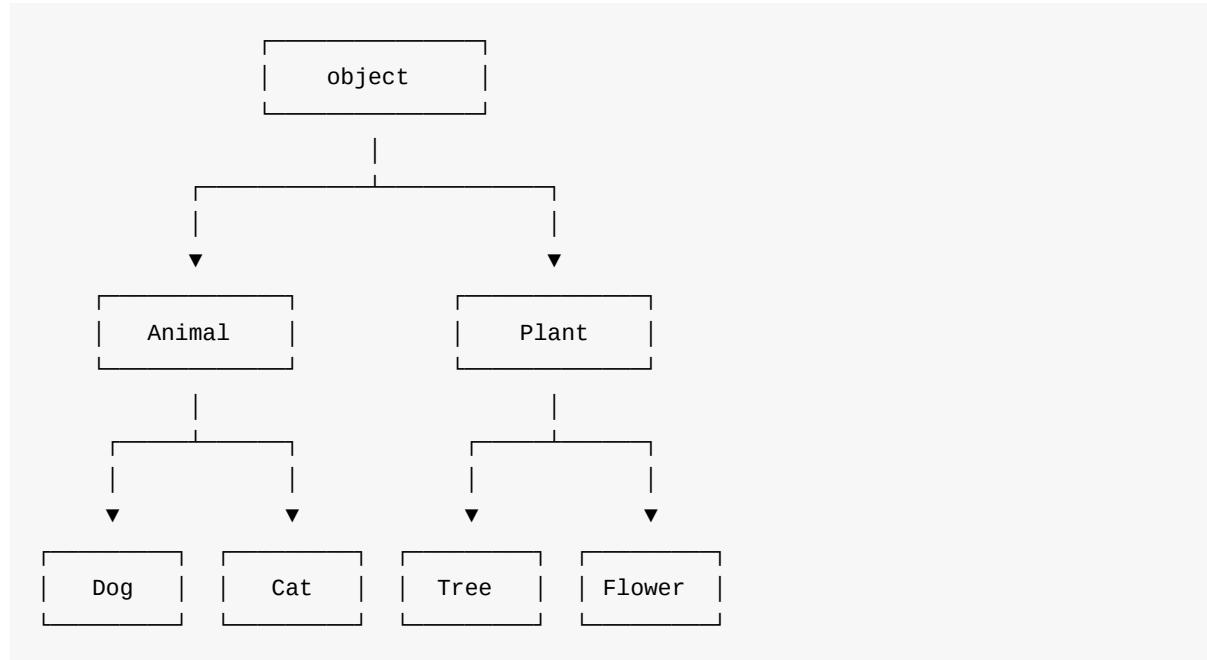
多态的好处就是，当我们需要传入 `Dog`、`Cat`、`Tortoise`时，我们只需要接收 `Animal` 类型就可以了，因为 `Dog`、`Cat`、`Tortoise`都是 `Animal` 类型，然后，按照 `Animal` 类型进行操作即可。由于 `Animal` 类型有 `run()` 方法，因此，传入的任意类型，只要是 `Animal` 类或者子类，就会自动调用实际类型的 `run()` 方法，这就是多态的意思：

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

对扩展开放：允许新增 `Animal` 子类；

对修改封闭：不需要修改依赖 `Animal` 类型的 `run_twice()` 等函数。

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



静态语言 vs 动态语言

对于静态语言（例如Java）来说，如果需要传入 `Animal` 类型，则传入的对象必须是 `Animal` 类型或者它的子类，否则，将无法调用 `run()` 方法。

对于Python这样的动态语言来说，则不一定需要传入 `Animal` 类型。我们只需要保证传入的对象有一个 `run()` 方法就可以了：

```
class Timer(object):
    def run(self):
        print('Start...')
```

这就是动态语言的“鸭子类型”，它并不要求严格的继承体系，一个对象只要“看起来像鸭子，走起路来像鸭子”，那它就可以被看做是鸭子。

Python的“file-like object”就是一种鸭子类型。对真正的文件对象，它有一个 `read()` 方法，返回其内容。但是，许多对象，只要有 `read()` 方法，都被视为“file-like object”。许多函数接收的参数就是“file-like object”，你不一定非要传入真正的文件对象，完全可以传入任何实现了 `read()` 方法的对象。

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。

动态语言的鸭子类型特点决定了继承不像静态语言那样是必须的。

参考源码

[animals.py](#)

获取对象信息

当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

使用type()

首先，我们来判断对象类型，使用 `type()` 函数：

基本类型都可以用 `type()` 判断：

```
>>> type(123)
<class 'int'>
>>> type('str')
<class 'str'>
>>> type(None)
<type(None) 'NoneType'>
```

如果一个变量指向函数或者类，也可以用 `type()` 判断：

```
>>> type(abs)
<class 'builtin_function_or_method'>
>>> type(a)
<class '__main__.Animal'>
```

但是 `type()` 函数返回的是什么类型呢？它返回对应的Class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(123)==int
True
>>> type('abc')==type('123')
True
>>> type('abc')==str
True
>>> type('abc')==type(123)
False
```

判断基本数据类型可以直接写 `int`，`str` 等，但如果要判断一个对象是否是函数怎么办？可以使用 `types` 模块中定义的常量：

```
>>> import types
>>> def fn():
...     pass
...
```

```
>>> type(fn)==types.FunctionType
True
>>> type(abs)==types.BuiltinFunctionType
True
>>> type(lambda x: x)==types.LambdaType
True
>>> type((x for x in range(10)))==types.GeneratorType
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(123, int)
True
>>> isinstance(b'a', bytes)
True
```

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list或者tuple：

```
>>> isinstance([1, 2, 3], (list, tuple))
True
>>> isinstance((1, 2, 3), (list, tuple))
True
```

总是优先使用`isinstance()`判断类型，可以将指定类型及其子类“一网打尽”。

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__', ..., '__subclasshook__', 'capitalize', 'casefold', ..., '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 MyDog(object):
...     def __len__(self):
...         return 100
...
>>> dog = MyDog()
>>> len(dog)
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 0x10077a6a0>>
>>> 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 0x10077a6a0>>
>>> 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：

```
sum = obj.x + obj.y
```

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，根据鸭子类型，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。

参考源码

[get_type.py](#)

[attrs.py](#)

实例属性和类属性

由于Python是动态语言，根据类创建的实例可以任意绑定属性。

给实例绑定属性的方法是通过实例变量，或者通过 `self` 变量：

```
class Student(object):
    def __init__(self, name):
        self.name = name

s = Student('Bob')
s.score = 90
```

但是，如果 `Student` 类本身需要绑定一个属性呢？可以直接在class中定义属性，这种属性是类属性，归 `Student` 类所有：

```
class Student(object):
    name = 'Student'
```

当我们定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到。来测试一下：

```
>>> class Student(object):
...     name = 'Student'
...
>>> s = Student() # 创建实例s
>>> print(s.name) # 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
Student
>>> print(Student.name) # 打印类的name属性
Student
>>> s.name = 'Michael' # 给实例绑定name属性
>>> print(s.name) # 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
Michael
>>> print(Student.name) # 但是类属性并未消失，用Student.name仍然可以访问
Student
>>> del s.name # 如果删除实例的name属性
>>> print(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
Student
```

从上面的例子可以看出，在编写程序的时候，千万不要对实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性，但是当你删除实例属性后，再使用相同的名称，访问到的将是类属性。

练习

为了统计学生人数，可以给`Student`类增加一个类属性，每创建一个实例，该属性自动增加：

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

class Student(object):
    count = 0
    def __init__(self, name):
        self.name = name
        Student.count = Student.count + 1

    # 测试:
    if Student.count != 0:
        print('测试失败!')
    else:
        bart = Student('Bart')
        if Student.count != 1:
            print('测试失败!')
        else:
            lisa = Student('Lisa')
            if Student.count != 2:
                print('测试失败!')
            else:
                print('Students:', Student.count)
                print('测试通过!')
```

小结

实例属性属于各个实例所有，互不干扰；

类属性属于类所有，所有实例共享一个属性；

不要对实例属性和类属性使用相同的名字，否则将产生难以发现的错误。

面向对象高级编程

错误、调试和测试

IO编程

进程和线程

正则表达式

字符串是编程时涉及到的最多的一种数据结构，对字符串进行操作的需求几乎无处不在。比如判断一个字符串是否是合法的Email地址，虽然可以编程提取 @ 前后的子串，再分别判断是否是单词和域名，但这样做不但麻烦，而且代码难以复用。

正则表达式是一种用来匹配字符串的强有力的武器。它的设计思想是用一种描述性的语言来给字符串定义一个规则，凡是符合规则的字符串，我们就认为它“匹配”了，否则，该字符串就是不合法的。

所以我们判断一个字符串是否是合法的Email的方法是：

1. 创建一个匹配Email的正则表达式；
2. 用该正则表达式去匹配用户的输入来判断是否合法。

因为正则表达式也是用字符串表示的，所以，我们要首先了解如何用字符来描述字符。

在正则表达式中，如果直接给出字符，就是精确匹配。用 \d 可以匹配一个数字，\w 可以匹配一个字母或数字，所以：

- '\00\d' 可以匹配 '007'，但无法匹配 '00A'；
- '\d\d\d' 可以匹配 '010'；
- '\w\w\d' 可以匹配 'py3'；
- . 可以匹配任意字符，所以：
 - 'py.' 可以匹配 'pyc'、'pyo'、'py！' 等等。

要匹配变长的字符，在正则表达式中，用 * 表示任意个字符（包括0个），用 + 表示至少一个字符，用 ? 表示0个或1个字符，用 {n} 表示n个字符，用 {n, m} 表示n-m个字符：

来看一个复杂的例子：\d{3}\s+\d{3,8}。

我们来从左到右解读一下：

1. \d{3} 表示匹配3个数字，例如 '010'；
2. \s 可以匹配一个空格（也包括Tab等空白符），所以 \s+ 表示至少有一个空格，例如匹配 ' '，' ' 等；
3. \d{3,8} 表示3-8个数字，例如 '1234567'。

综合起来，上面的正则表达式可以匹配以任意个空格隔开的带区号的电话号码。

如果要匹配 '010-12345' 这样的号码呢？由于 '-' 是特殊字符，在正则表达式中，要用 '\\' 转义，所以，上面的正则是 \d{3}-\d{3,8}。

但是，仍然无法匹配 '010 - 12345'，因为带有空格。所以我们需要更复杂的匹配方式。

进阶

要做更精确地匹配，可以用 `[]` 表示范围，比如：

- `[0-9a-zA-Z_]` 可以匹配一个数字、字母或者下划线；
- `[0-9a-zA-Z_]+` 可以匹配至少由一个数字、字母或者下划线组成的字符串，比如 `'a100'`，`'0_Z'`，`'Py3000'` 等等；
- `[a-zA-Z_][0-9a-zA-Z_]*` 可以匹配由字母或下划线开头，后接任意个由一个数字、字母或者下划线组成的字符串，也就是Python合法的变量；
- `[a-zA-Z_][0-9a-zA-Z_]{0, 19}` 更精确地限制了变量的长度是1-20个字符（前面1个字符+后面最多19个字符）。

`A|B` 可以匹配A或B，所以 `(P|p)ython` 可以匹配 `'Python'` 或者 `'python'`。

`^` 表示行的开头，`^\d` 表示必须以数字开头。

`$` 表示行的结束，`\d$` 表示必须以数字结束。

你可能注意到了，`py` 也可以匹配 `'python'`，但是加上 `^py$` 就变成了整行匹配，就只能匹配 `'py'` 了。

re模块

有了准备知识，我们就可以在Python中使用正则表达式了。Python提供 `re` 模块，包含所有正则表达式的功能。由于Python的字符串本身也用 \ 转义，所以要特别注意：

```
s = 'ABC\\-001' # Python的字符串
# 对应的正则表达式字符串变成：
# 'ABC\~-001'
```

因此我们强烈建议使用Python的 `r` 前缀，就不用考虑转义的问题了：

```
s = r'ABC\~-001' # Python的字符串
# 对应的正则表达式字符串不变：
# 'ABC\~-001'
```

先看看如何判断正则表达式是否匹配：

```
>>> import re
>>> re.match(r'^\d{3}~\d{3,8}$', '010-12345')
<sre.SRE_Match object; span=(0, 9), match='010-12345'>
>>> re.match(r'^\d{3}~\d{3,8}$', '010 12345')
>>>
```

`match()` 方法判断是否匹配，如果匹配成功，返回一个 `Match` 对象，否则返回 `None`。常见的判断方法就是：

```
test = '用户输入的字符串'
if re.match(r'正则表达式', test):
    print('ok')
else:
    print('failed')
```

切分字符串

用正则表达式切分字符串比用固定的字符更灵活，请看正常的切分代码：

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

嗯，无法识别连续的空格，用正则表达式试试：

```
>>> re.split(r'\s+', 'a b c')
['a', 'b', 'c']
```

无论多少个空格都可以正常分割。加入`,`，试试：

```
>>> re.split(r'[\s\,]+', 'a,b, c d')
['a', 'b', 'c', 'd']
```

再加入`;`，试试：

```
>>> re.split(r'[\s\,\;]+', 'a,b;; c d')
['a', 'b', 'c', 'd']
```

如果用户输入了一组标签，下次记得用正则表达式来把不规范的输入转化成正确的数组。

分组

除了简单地判断是否匹配之外，正则表达式还有提取子串的强大功能。用`()`表示的就是要提取的分组（Group）。比如：

`^\(\d{3}\)-(\d{3,8})$` 分别定义了两个组，可以直接从匹配的字符串中提取出区号和本地号码：

```
>>> m = re.match(r'^(\d{3})-(\d{3,8})$', '010-12345')
>>> m
<_sre.SRE_Match object; span=(0, 9), match='010-12345'>
>>> m.group(0)
'010-12345'
```

```
>>> m.group(1)
'010'
>>> m.group(2)
'12345'
```

如果正则表达式中定义了组，就可以在 `Match` 对象上用 `group()` 方法提取出子串来。

注意到 `group(0)` 永远是原始字符串，`group(1)`、`group(2)`表示第1、2、.....个子串。

提取子串非常有用。来看一个更凶残的例子：

```
>>> t = '19:05:30'
>>> m = re.match(r'^(\d{2}|\d{1})\:(\d{2}|\d{1})\:(\d{2}|\d{1})', t)
>>> m.groups()
('19', '05', '30')
```

这个正则表达式可以直接识别合法的时间。但是有些时候，用正则表达式也无法做到完全验证，比如识别日期：

```
'^(\d{1}|2|3)\-(\d{1}|2|3|4)\-(\d{1}|2|3|4|5|6|7|8|9)'
```

对于 `'2-30'`，`'4-31'` 这样的非法日期，用正则还是识别不了，或者说写出来非常困难，这时就需要程序配合作识别了。

贪婪匹配

最后需要特别指出的是，正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。举例如下，匹配出数字后面的 `0`：

```
>>> re.match(r'^(\d+)(0*)$', '102300').groups()
('102300', '')
```

由于 `\d+` 采用贪婪匹配，直接把后面的 `0` 全部匹配了，结果 `0*` 只能匹配空字符串了。

必须让 `\d+` 采用非贪婪匹配（也就是尽可能少匹配），才能把后面的 `0` 匹配出来，加个 `?` 就可以让 `\d+` 采用非贪婪匹配：

```
>>> re.match(r'^(\d+?)(0*)$', '102300').groups()
('1023', '00')
```

编译

当我们在Python中使用正则表达式时，`re`模块内部会干两件事情：

1. 编译正则表达式，如果正则表达式的字符串本身不合法，会报错；

2. 用编译后的正则表达式去匹配字符串。

如果一个正则表达式要重复使用几千次，出于效率的考虑，我们可以预编译该正则表达式，接下来重复使用时就不需要编译这个步骤了，直接匹配：

```
>>> import re
# 编译:
>>> re_telephone = re.compile(r'^(\d{3})-(\d{3,8})$')
# 使用:
>>> re_telephone.match('010-12345').groups()
('010', '12345')
>>> re_telephone.match('010-8086').groups()
('010', '8086')
```

编译后生成Regular Expression对象，由于该对象自己包含了正则表达式，所以调用对应的方法时不用给出正则字符串。

小结

正则表达式非常强大，要在短短的一节里讲完是不可能的。要讲清楚正则的所有内容，可以写一本厚厚的书了。如果你经常遇到正则表达式的问题，你可能需要一本正则表达式的参考书。

练习

请尝试写一个验证Email地址的正则表达式。版本一应该可以验证出类似的Email：

- someone@gmail.com
- bill.gates@microsoft.com

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

em=re.compile(r'[\w\.]*(\@\w*\.\com)')
if em.match(addr):
    return True
else:
    return False

# 测试:
assert is_valid_email('someone@gmail.com')
assert is_valid_email('bill.gates@microsoft.com')
assert not is_valid_email('bob@example.com')
assert not is_valid_email('mr-bob@example.com')
print('ok')
```

版本二可以提取出带名字的Email地址：

- <Tom Paris>tom@voyager.org => Tom Paris

- bob@example.com => bob

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

# 测试:
assert name_of_email('<Tom Paris> tom@voyager.org') == 'Tom Paris'
assert name_of_email('tom@voyager.org') == 'tom'
print('ok')
```

参考源码

[regex.py](#)

常用内建模块

常用第三方模块

virtualenv

在开发Python应用程序的时候，系统安装的Python3只有一个版本：3.4。所有第三方的包都会被 pip 安装到Python3的 site-packages 目录下。

如果我们要同时开发多个应用程序，那这些应用程序都会共用一个Python，就是安装在系统的Python 3。如果应用A需要jinja 2.7，而应用B需要jinja 2.6怎么办？

这种情况下，每个应用可能需要各自拥有一套“独立”的Python运行环境。virtualenv就是用来为一个应用创建一套“隔离”的Python运行环境。

首先，我们用 pip 安装virtualenv：

```
$ pip3 install virtualenv
```

然后，假定我们要开发一个新的项目，需要一套独立的Python运行环境，可以这么做：

第一步，创建目录：

```
Mac:~ michael$ mkdir myproject
Mac:~ michael$ cd myproject/
Mac:myproject michael$
```

第二步，创建一个独立的Python运行环境，命名为 venv：

```
Mac:myproject michael$ virtualenv --no-site-packages venv
Using base prefix '/usr/local/.../Python.framework/Versions/3.4'
New python executable in venv/bin/python3.4
Also creating executable in venv/bin/python
Installing setuptools, pip, wheel...done.
```

命令 virtualenv 就可以创建一个独立的Python运行环境，我们还加上了参数 --no-site-packages，这样，已经安装到系统Python环境中的所有第三方包都不会复制过来，这样，我们就得到了一个不带任何第三方包的“干净”的Python运行环境。

新建的Python环境被放到当前目录下的 venv 目录。有了 venv 这个Python环境，可以用 source 进入该环境：

```
Mac:myproject michael$ source venv/bin/activate
(venv)Mac:myproject michael$
```

注意到命令提示符变了，有个 (venv) 前缀，表示当前环境是一个名为 venv 的Python环境。

下面正常安装各种第三方包，并运行 python 命令：

```
(venv)Mac:myproject michael$ pip install jinja2
...
Successfully installed jinja2-2.7.3 markupsafe-0.23
(venv)Mac:myproject michael$ python myapp.py
...
```

在 `venv` 环境下，用 `pip` 安装的包都被安装到 `venv` 这个环境下，系统Python环境不受任何影响。也就是说，`venv` 环境是专门针对 `myproject` 这个应用创建的。

退出当前的 `venv` 环境，使用 `deactivate` 命令：

```
(venv)Mac:myproject michael$ deactivate
Mac:myproject michael$
```

此时就回到了正常的环境，现在 `pip` 或 `python` 均是在系统Python环境下执行。

完全可以针对每个应用创建独立的Python运行环境，这样就可以对每个应用的Python环境进行隔离。

`virtualenv`是如何创建“独立”的Python运行环境的呢？原理很简单，就是把系统Python复制一份到 `virtualenv` 的环境，用命令 `source venv/bin/activate` 进入一个 `virtualenv` 环境时，`virtualenv` 会修改相关环境变量，让命令 `python` 和 `pip` 均指向当前的 `virtualenv` 环境。

小结

`virtualenv` 为应用提供了隔离的Python运行环境，解决了不同应用间多版本的冲突问题。

图形界面

Python支持多种图形界面的第三方库，包括：

- Tk
- wxWidgets
- Qt
- GTK

等等。

但是Python自带的库是支持Tk的Tkinter，使用Tkinter，无需安装任何包，就可以直接使用。本章简单介绍如何使用Tkinter进行GUI编程。

Tkinter

我们来梳理一下概念：

我们编写的Python代码会调用内置的Tkinter，Tkinter封装了访问Tk的接口；

Tk是一个图形库，支持多个操作系统，使用Tcl语言开发；

Tk会调用操作系统提供的本地GUI接口，完成最终的GUI。

所以，我们的代码只需要调用Tkinter提供的接口就可以了。

第一个GUI程序

使用Tkinter十分简单，我们来编写一个GUI版本的“Hello, world!”。

第一步是导入Tkinter包的所有内容：

```
from tkinter import *
```

第二步是从 `Frame` 派生一个 `Application` 类，这是所有Widget的父容器：

```
class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

    def createWidgets(self):
        self.helloLabel = Label(self, text='Hello, world!')
        self.helloLabel.pack()
        self.quitButton = Button(self, text='Quit', command=self.quit)
```

```
    self.quitButton.pack()
```

在GUI中，每个Button、Label、输入框等，都是一个Widget。Frame则是可以容纳其他Widget的Widget，所有的Widget组合起来就是一棵树。

`pack()`方法把Widget加入到父容器中，并实现布局。`pack()`是最简单的布局，`grid()`可以实现更复杂的布局。

在`createWidgets()`方法中，我们创建一个`Label`和一个`Button`，当`Button`被点击时，触发`self.quit()`使程序退出。

第三步，实例化`Application`，并启动消息循环：

```
app = Application()
# 设置窗口标题:
app.master.title('Hello World')
# 主消息循环:
app.mainloop()
```

GUI程序的主线程负责监听来自操作系统的消息，并依次处理每一条消息。因此，如果消息处理非常耗时，就需要在新线程中处理。

运行这个GUI程序，可以看到下面的窗口：



点击“Quit”按钮或者窗口的“x”结束程序。

输入文本

我们再对这个GUI程序改进一下，加入一个文本框，让用户可以输入文本，然后点按钮后，弹出消息对话框。

```
from tkinter import *
import tkinter.messagebox as messagebox

class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

    def createWidgets(self):
        self.nameInput = Entry(self)
        self.nameInput.pack()
```

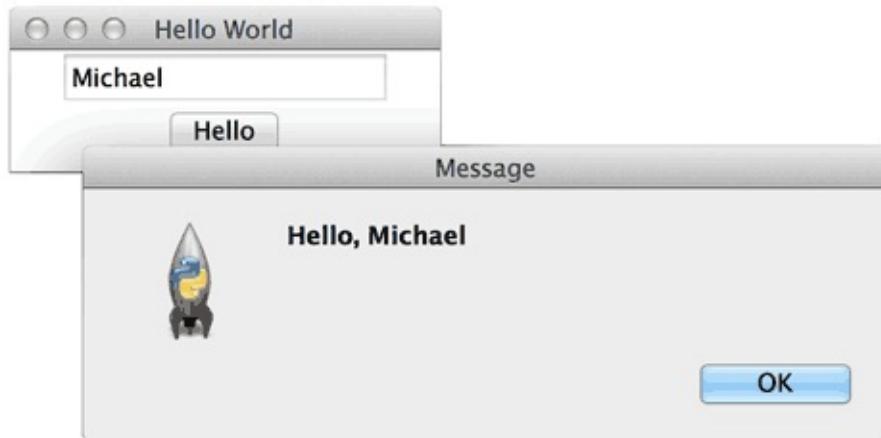
```
self.alertButton = Button(self, text='Hello', command=self.hello)
self.alertButton.pack()

def hello(self):
    name = self.nameInput.get() or 'world'
    messagebox.showinfo('Message', 'Hello, %s' % name)

app = Application()
# 设置窗口标题:
app.master.title('Hello World')
# 主消息循环:
app.mainloop()
```

当用户点击按钮时，触发 `hello()`，通过 `self.nameInput.get()` 获得用户输入的文本后，使用 `tkMessageBox.showinfo()` 可以弹出消息对话框。

程序运行结果如下：



小结

Python内置的Tkinter可以满足基本的GUI程序的要求，如果是非常复杂的GUI程序，建议用操作系统原生支持的语言和库来编写。

参考源码

[hello_gui.py](#)

网络编程

电子邮件

访问数据库

Web开发

异步IO

实战

FAQ

常见问题

本节列出常见的一些问题。

如何获取当前路径

当前路径可以用 `..` 表示，再用 `os.path.abspath()` 将其转换为绝对路径：

```
# -*- coding:utf-8 -*-
# test.py

import os

print(os.path.abspath('.'))
```

运行结果：

```
$ python3 test.py
/Users/michael/workspace/testing
```

如何获取当前模块的文件名

可以通过特殊变量 `__file__` 获取：

```
# -*- coding:utf-8 -*-
# test.py

print(__file__)
```

输出：

```
$ python3 test.py
test.py
```

如何获取命令行参数

可以通过 `sys` 模块的 `argv` 获取：

```
# -*- coding:utf-8 -*-
# test.py
```

```
import sys  
  
print(sys.argv)
```

输出：

```
$ python3 test.py -a -s "Hello world"  
['test.py', '-a', '-s', 'Hello world']
```

`argv` 的第一个元素永远是命令行执行的 `.py` 文件名。

如何获取当前Python命令的可执行文件路径

`sys` 模块的 `executable` 变量就是Python命令可执行文件的路径：

```
# -*- coding:utf-8 -*-  
# test.py  
  
import sys  
  
print(sys.executable)
```

在Mac下的结果：

```
$ python3 test.py  
/usr/local/opt/python3/bin/python3.4
```

期末总结

终于到了期末总结的时刻了！

经过一段时间的学习，相信你对Python已经初步掌握。一开始，可能觉得Python上手很容易，可是越往后学，会越困难，有的时候，发现理解不了代码，这时，不妨停下来思考一下，先把概念搞清楚，代码自然就明白了。

Python非常适合初学者用来进入计算机编程领域。Python属于非常高级的语言，掌握了这门高级语言，就对计算机编程的核心思想——抽象有了初步理解。如果希望继续深入学习计算机编程，可以学习Java、C、JavaScript、Lisp等不同类型的语言，只有多掌握不同领域的语言，有比较才更有收获。

