

# Python教程

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

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

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

用Python可以做什么？可以做日常任务，比如自动备份你的MP3；可以做网站，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++。

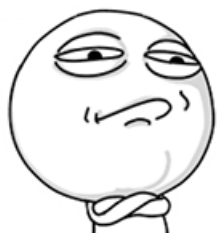
如果你是小白用户，满足以下条件：

- 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 !



## 关于作者

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



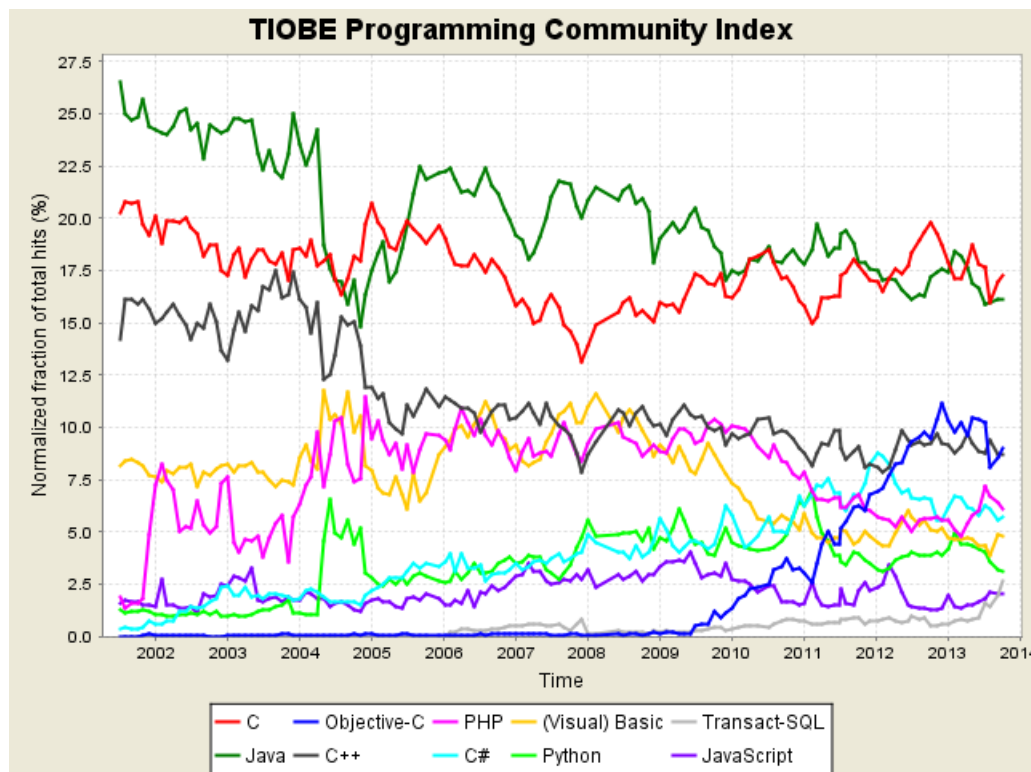
廖雪峰  北京 朝阳区

[+ 加关注](#)

# Python简介

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

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



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

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

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

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

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

龟叔给Python的定位是“优雅”、“明确”、“简单”，所以Python程序看上去总是简单易懂，初学者学Python，不但入门容易，而且将来深入下去，可以编写那些非常非常复杂的程序。

总的来说，Python的哲学就是简单优雅，尽量写容易看明白的代码，尽量写少的代码。如果一个资深程序员向你炫耀他写的晦涩难懂、动不动就几万行的代码，你可以尽情地嘲笑他。

那Python适合开发哪些类型的应用呢？

首选是网络应用，包括网站、后台服务等；

其次是许多日常需要的小工具，包括系统管理员需要的脚本任务等等；

另外就是把其他语言开发的程序再包装起来，方便使用。

最后说说Python的缺点。

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

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

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



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

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

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



当然，Python还有其他若干小缺点，请自行忽略，就不一一列举了。

# 安装Python

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

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

## 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。

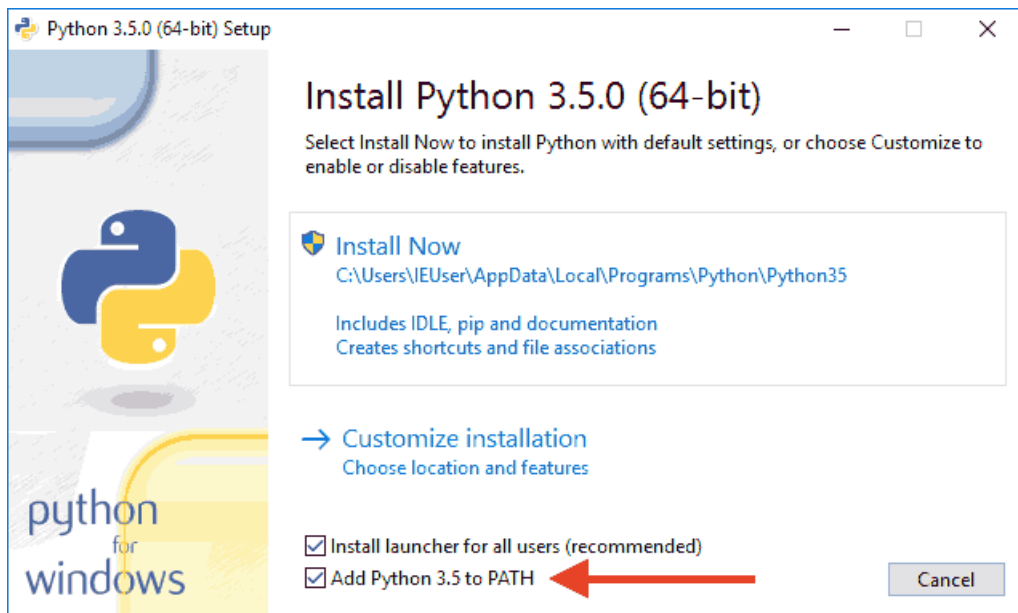
## 在Linux上安装Python

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：



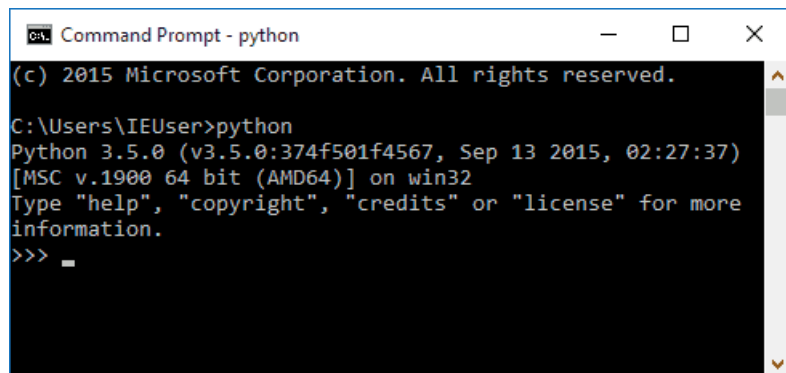
特别注意勾选上Add Python 3.6 to PATH，然后点“Install Now”即可完成安装。

视频演示：

## 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：



```
Command Prompt - python
(c) 2015 Microsoft Corporation. All rights reserved.

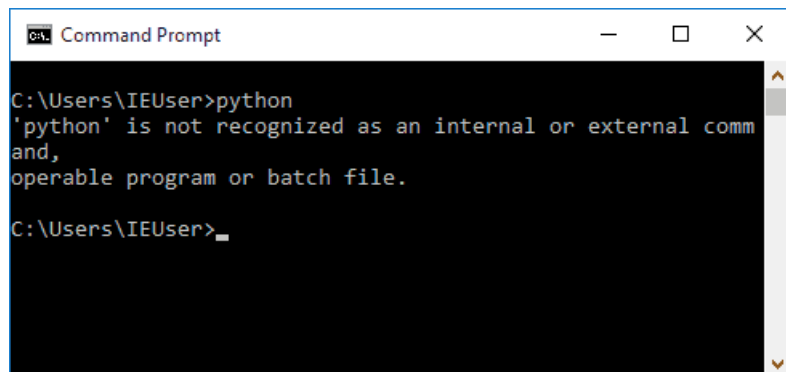
C:\Users\IEUser>python
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:27:37)
[MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> _
```

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。



```
Command Prompt

C:\Users\IEUser>python
'python' is not recognized as an internal or external command,
and,
operable program or batch file.

C:\Users\IEUser>_
```

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。

视频演示：

## 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`。



# Python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。

## CPython

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。

## IPython

IPython是基于CPython之上的一个交互式解释器，也就是说，IPython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IPython用In [序号]:作为提示符。

## PyPy

PyPy是另一个Python解释器，它的目标是执行速度。PyPy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PyPy下运行，但是PyPy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PyPy下执行，就需要了解[PyPy和CPython的不同点](#)。

## Jython

Jython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。

## IronPython

IronPython和Jython类似，只不过IronPython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Python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 - □ 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.
>>> exit()

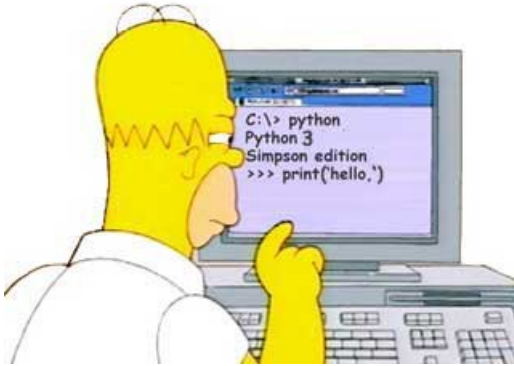
C:\> _
```

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。

视频演示：

## 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 _ □ x
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。

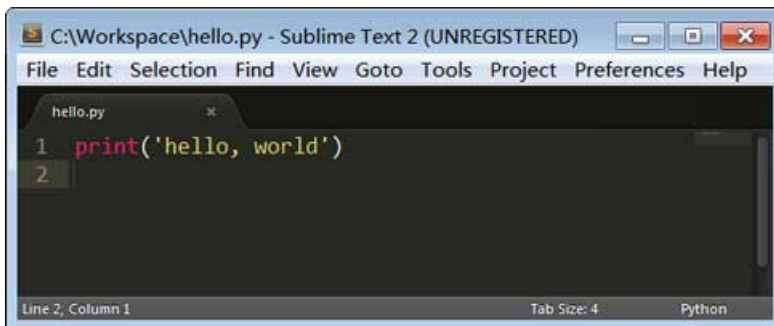
所以，实际开发的时候，我们总是使用一个文本编辑器来写代码，写完了，保存为一个文件，这样，程序就可以反复运行了。

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。

视频演示：

## 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：

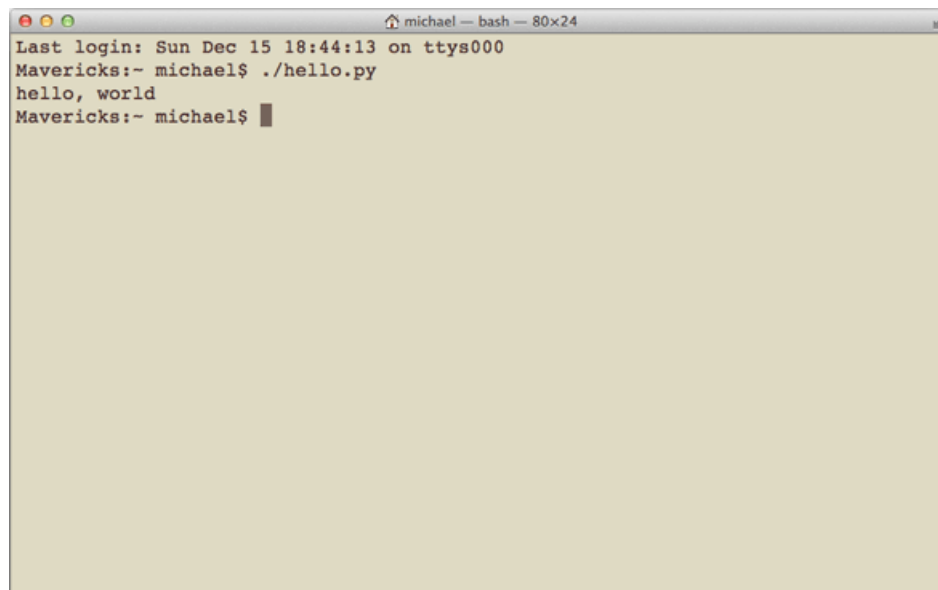
```
#!/usr/bin/env python3

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：

A screenshot of a macOS terminal window. The title bar shows 'michael — bash — 80x24'. The terminal content shows the last login time as 'Sun Dec 15 18:44:13 on ttys000'. The user 'Mavericks:- michael\$' runs the command './hello.py', and the terminal outputs 'hello, world'. The prompt returns to 'Mavericks:- michael\$' with a cursor.

```
michael — bash — 80x24
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](#)

# Python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
```

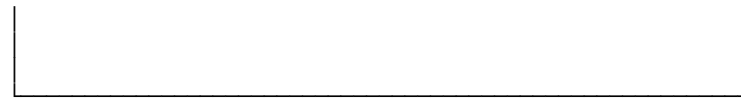
如果看到Ready for Python code on port 39093...表示运行成功，不要关闭命令行窗口，最小化放到后台运行即可：

Command Prompt

- □ x

```
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  $\geq$  9
- Firefox
- Chrome
- Safari

# 测试代码:

```
----  
print('Hello, world')
```



# 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，如果要从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。比如输入用户的名字：

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

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

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

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

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

## 参考源码

[do\\_input.py](#)

# Python基础

Python是一种计算机编程语言。计算机编程语言和我们日常使用的自然语言有所不同，最大的区别就是，自然语言在不同的语境下有不同的理解，而计算机要根据编程语言执行任务，就必须保证编程语言写出的程序决不能有歧义，所以，任何一种编程语言都有自己的一套语法，编译器或者解释器就是负责把符合语法的程序代码转换成CPU能够执行的机器码，然后执行。Python也不例外。

Python的语法比较简单，采用缩进方式，写出来的代码就像下面的样子：

```
# print absolute value of an integer:
a = 100
if a >= 0:
    print(a)
else:
    print(-a)
```

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

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

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

缩进的坏处就是“复制一粘贴”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， $1.23 \times 10^9$ 和 $12.3 \times 10^8$ 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， $1.23 \times 10^9$ 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\\')
\
\
```

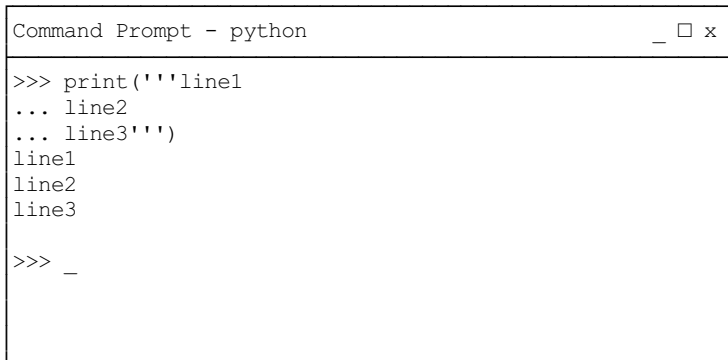
如果字符串里面有很多字符都需要转义，就需要加很多\，为了简化，Python还允许用r''表示''内部的字符串默认不转义，可以自己试试：

```
>>> 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：



当输入完结束符'''和括号)后，执行该语句并打印结果。

如果写成程序并保存为.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,\n
world''')
```

## 布尔值

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

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

布尔值可以用and、or和not运算。

and运算是与运算，只有所有都为True，and运算结果才是True：

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

```
False
>>> 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：

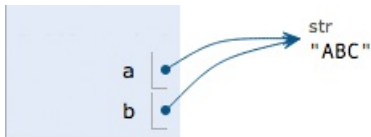
```
# -*- coding: utf-8 -*-
----
a = 'ABC'
b = a
a = 'XYZ'
print(b)
```

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：

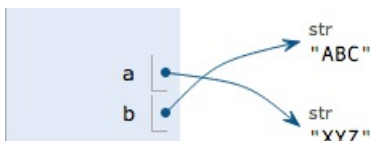
执行`a = 'ABC'`，解释器创建了字符串'ABC'和变量a，并把a指向'ABC'：



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

```
PI = 3.14159265359
```

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

最后解释一下整数的除法为什么也是精确的。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(???)
```

## 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，而且还不能和ASCII编码冲突，所以，中国制定了GB2312编码，用来把中文编进去。

你可以想得到的是，全世界有上百种语言，日本把日文编到Shift\_JIS里，韩国把韩文编到Euc-kr里，各国有各国的标准，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：

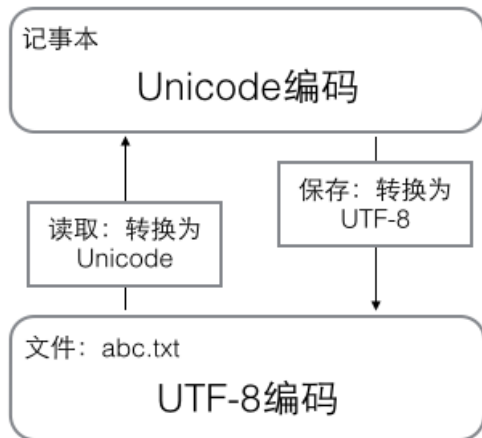
字符	ASCII	Unicode	UTF-8
A	01000001	00000000 01000001	01000001
中	x	01001110 00101101	11100100 10111000 10101101

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。

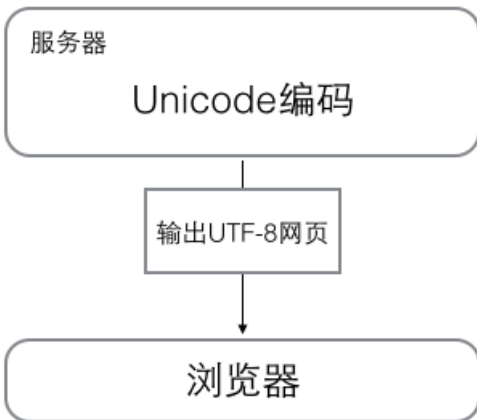
搞清楚了ASCII、Unicode和UTF-8的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

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\xba\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 in range(128)
```

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

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\xba\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个英文字符只占用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。

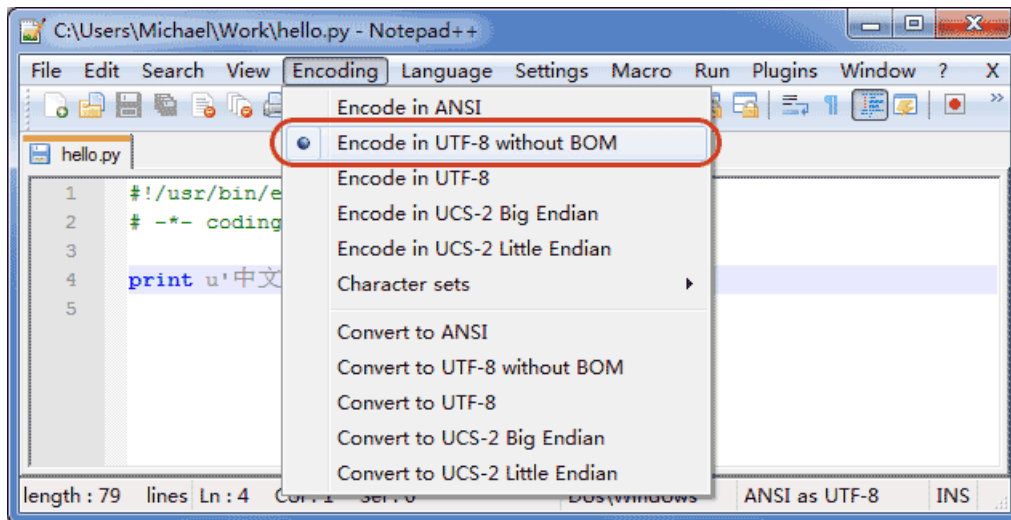
由于Python源代码也是一个文本文件，所以，当你的源代码中包含中文的时候，在保存源代码时，就需要务必指定保存为UTF-8编码。当Python解释器读取源代码时，为了让它按UTF-8编码读取，我们通常在文件开头写上这两行：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

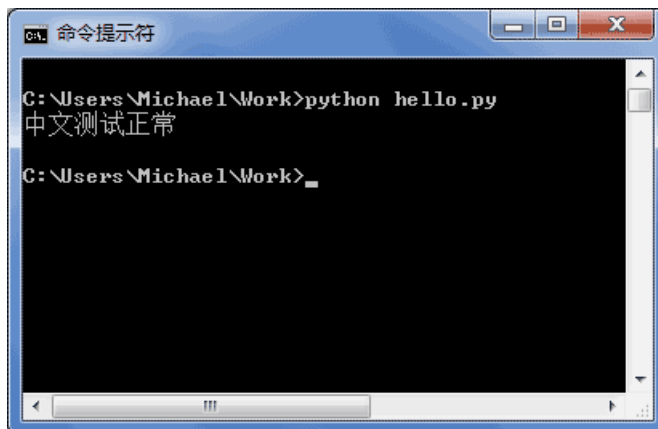
第一行注释是为了告诉Linux/OS X系统，这是一个Python可执行程序，Windows系统会忽略这个注释；

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：

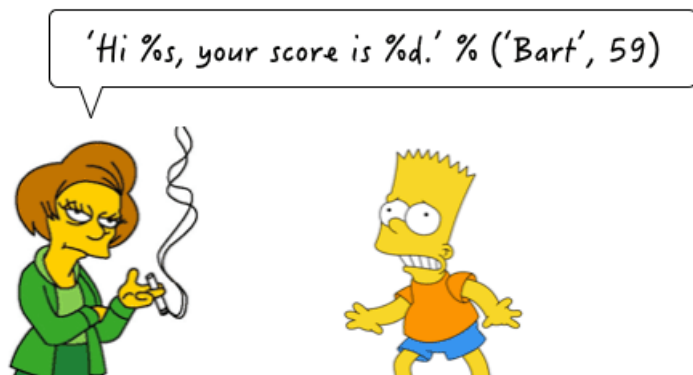


如果.py文件本身使用UTF-8编码，并且也申明了# -\*- coding: utf-8 -\*-，打开命令提示符测试就可以正常显示中文：



## 格式化

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



在Python中，采用的格式化方式和C语言是一致的，用%实现，举例如下：

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

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

常见的占位符有：

#### 占位符 替换内容

%d	整数
%f	浮点数
%s	字符串
%x	十六进制整数

其中，格式化整数和浮点数还可以指定是否补0和整数与小数的位数：

```
# -*- coding: utf-8 -*-
----
print('%2d-%02d' % (3, 1))
print('%0.2f' % 3.1415926)
```

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 = ???
print('??? ' % 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\xca'
```

但这种方式纯属自找麻烦，如果没有特殊业务要求，请牢记仅使用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,)
```

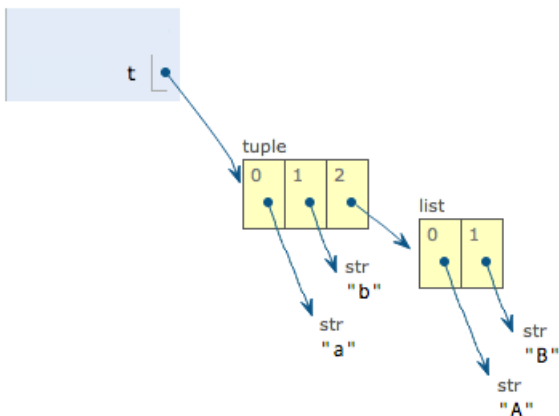
Python在显示只有1个元素的tuple时，也会加一个逗号,，以免你误解成数学计算意义上的括号。

最后来看一个“可变的”tuple：

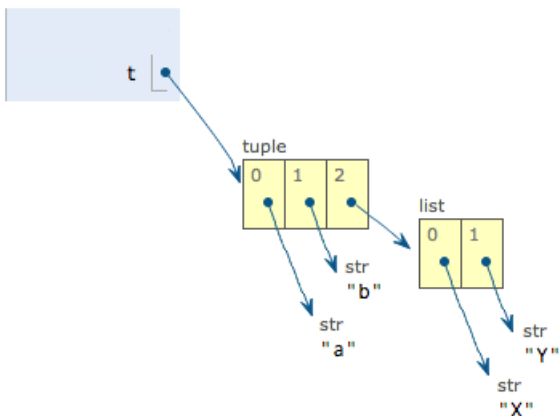
```
>>> t = ('a', 'b', ['A', 'B'])
>>> t[2][0] = 'X'
>>> t[2][1] = 'Y'
>>> t
('a', 'b', ['X', 'Y'])
```

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(?)
# 打印Python:
print(?)
# 打印Lisa:
print(?)
```

## 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 = ???
if ???:
    pass
```

## 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 $1+2+3+...+10$ ，勉强也能写出来。

但是，要计算 $1+2+3+...+10000$ ，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)
```

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']
```

## 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的方法。

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3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：

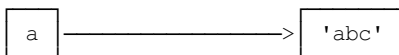
```
>>> a = 'abc'
>>> a.replace('a', 'A')
'Abc'
>>> a
'abc'
```

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？

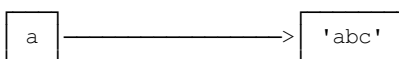
我们先把代码改成下面这样：

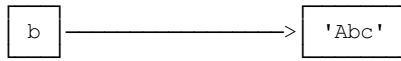
```
>>> a = 'abc'
>>> b = a.replace('a', 'A')
>>> b
'Abc'
>>> a
'abc'
```

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 $s = \text{area\_of\_circle}(x)$ ，而函数 $\text{area\_of\_circle}$ 本身只需要写一次，就可以多次调用。

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

## 抽象

抽象是数学中非常常见的概念。举个例子：

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

100

$$\sum n$$

$n=1$

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

而且，这种抽象记法是可扩展的，比如：

100

$$\sum (n^2 + 1)$$

$n=1$

还原成加法运算就变成了：

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

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

写计算机程序也是一样，函数就是最基本的一种代码抽象的方式。



# 调用函数

Python内置了很多有用的函数，我们可以直接调用。

要调用一个函数，需要知道函数的名称和参数，比如求绝对值的函数`abs`，只有一个参数。可以直接从Python的官方网站查看文档：

<http://docs.python.org/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(???)
```

## 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：

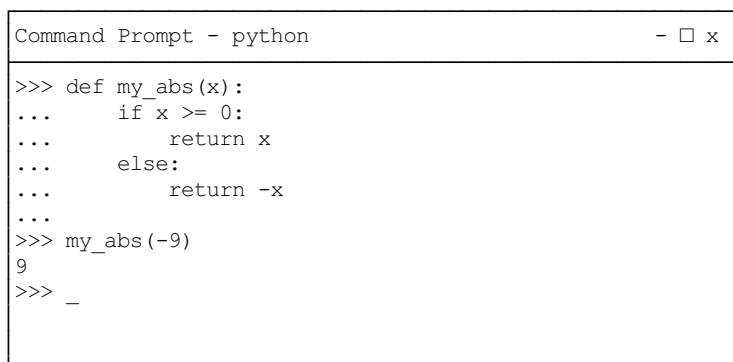
```
# -*- coding: utf-8 -*-
----
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x
----
print(my_abs(-99))
```

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。

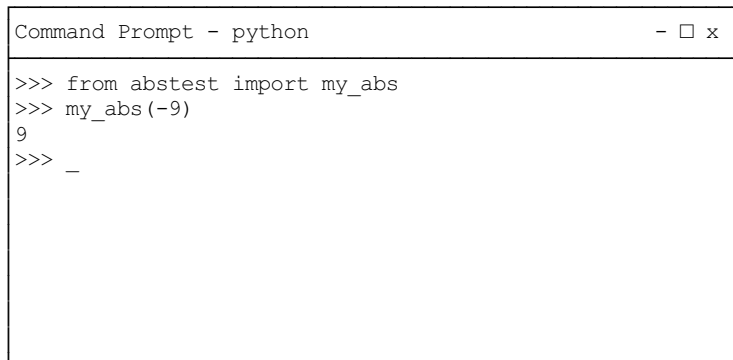
如果没有`return`语句，函数执行完毕后也会返回结果，只是结果为`None`。`return None`可以简写为`return`。

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
>>> 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
```

但其实这只是一种假象，Python函数返回的仍然是单一值：

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

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

## 小结

定义函数时，需要确定函数名和参数个数；

如果有必要，可以先对参数的数据类型做检查；

函数体内部可以用`return`随时返回函数结果；

函数执行完毕也没有`return`语句时，自动`return None`。

函数可以同时返回多个值，但其实就是一个`tuple`。

## 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):
    ----
    pass
    ----
# 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](#)