

Python 入门指南

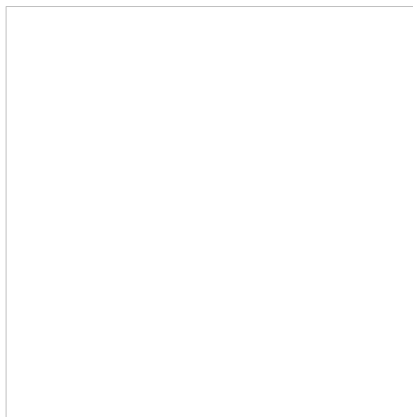
Python 是一门当下极为流行的程序设计语言，在网站服务器开发、数据分析、人工智能方面也有着广泛的应用。

同时，由于 Python 语法简洁、模块丰富，非常适合零基础新手作为入门编程的第一门语言。

本教程从零开始，介绍 Python 的基本语法、常用类型和方法，手把手带你进入编程的世界。无需预备知识，完全没有接触过计算机编程方面的小白也可以无压力地学习本教程。本教程从第一版于 2013 年上线，经过数次更新，见证了数十万人的编程之路。

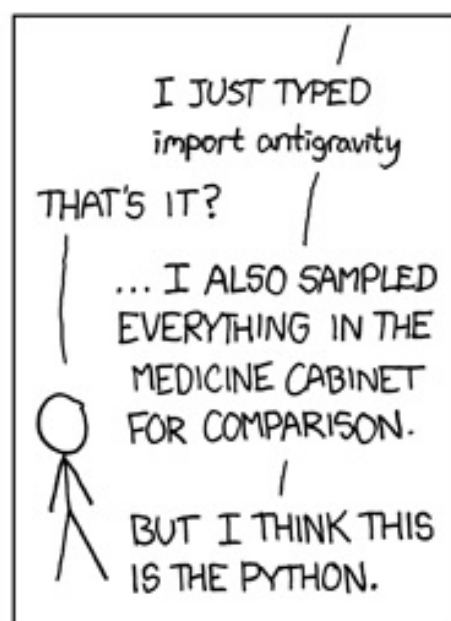
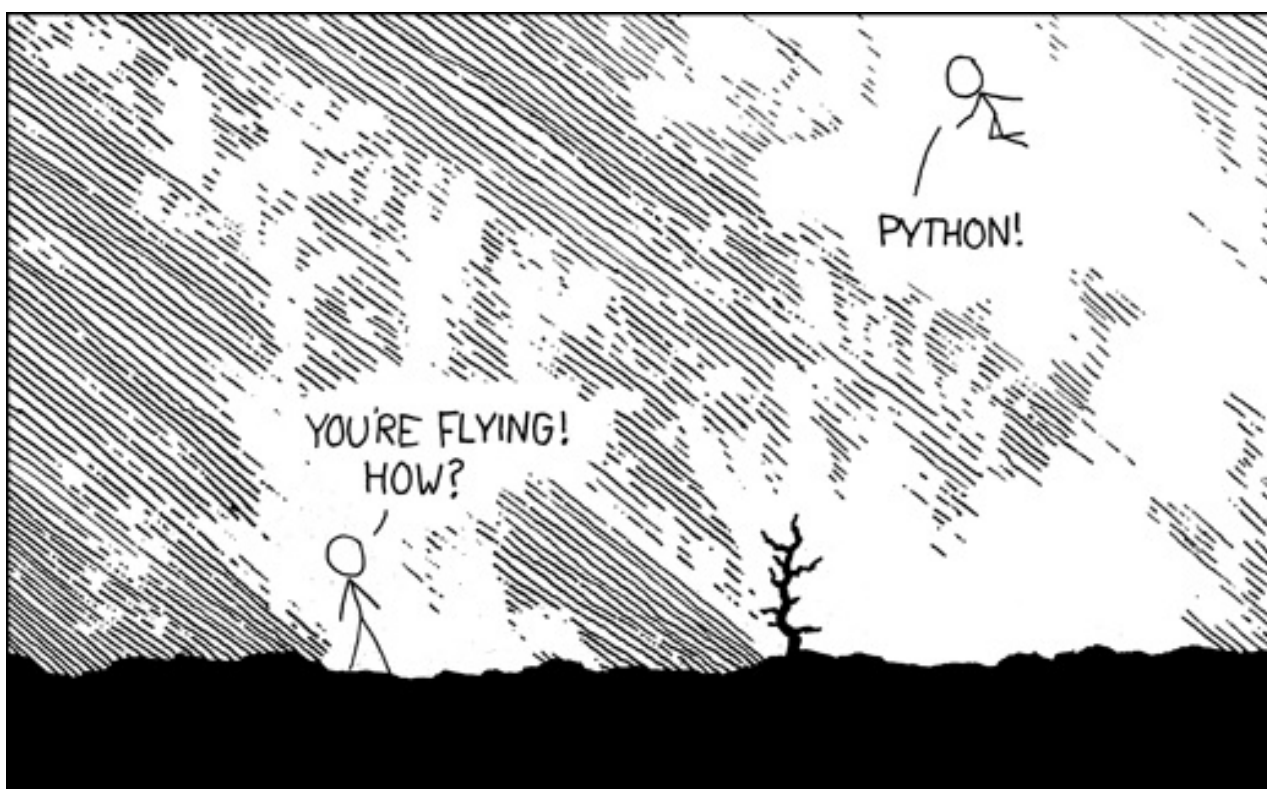
本教程推荐使用最新版的 Python 3 进行学习。

对于本教程以及学习中的任何疑问和想法，可以关注我们的公众号“**Crossin的编程教室**”，或关注同名知乎专栏与我们交流。



- [0.为什么选择 Python 入门? →](#)

【Python 第0课】为什么选择 Python 入门？



Why Python?

为什么用Python作为编程入门语言？

原因，很简单。

嗯。。。原因就是，很简单。。。。

每种语言都会有它的支持者和反对者。去网上搜索一下“python的好处”，你会得到很多结果，诸如应用范围广泛、开源、社区活跃、丰富的库、跨平台等等等等，也可能找到不少对它的批评，格式死板、效率低、版本不兼容之类。不过这些优缺点的权衡都是程序员们的烦恼。作为一个想要学点编程入门的初学者来说，简单才是最重要的。如果对你来说不能上手，后面其他的是空谈。当学C++的同学还在写链表，学Java的同学还在折腾运行环境的时候，学Python的你已经像上图一样飞上天了。

当然，除了简单，我做Python教程还有一个重要的原因：我每天都在写Python代码。我可以更细致地为你讲解其中容易被忽略的细节。Python是很有利于形成良好编程思维的一门语言。每天5分钟，先动起手来再说。

推荐一本我比较喜欢的入门书籍《父与子的编程之旅》（又译作《与孩子一起学编程》），这本书特别适合完全没有接触过编程的人入门（唯一缺点是版本有一点老）。如果你曾接触过其他编程语言，可以考虑另外两本：《简明Python教程》和《Head First Python》。

----- 程序员的分割线 | 课外的话 -----

Why这个公众账号？

事情的直接起因是Sunny同学昨天跟我说，她最近在学Python，如果碰到不懂的地方希望能问问我。我又联想到前阵子Jing同学说想学一门编程语言，于是就有了这么个号。（这中间还有个插曲：我之前申请过两个公众号，结果再次申请的才被系统告知不能再申了，之前的号也不能改名字、不能删除。后来多亏Jing同学帮忙申请才得以把这个账号开起来。在此谢过！）

回想起来，我可能从很早的时候就有一种好为人师的心理。当别人听了半天课又琢磨了很久也没搞懂某个问题，被自己讲解了一番就恍然大悟的时候，总会有一种成就感。

其实就算没这个号，我现在也经常辅导别人学习编程。既然都是教，干脆开个号，给大家一起听听。如果这个号能满足我小小的成就感，又能帮到一点点想学编程的朋友，何乐而不为？只不过最近的确很忙，每天5分钟，先试试看吧。

我觉得，如果真能坚持说下去，又有人能坚持听下去（当然，有人听是前提），那至少听完的人可以对编程有个大概的了解，写点小程序自娱自乐不在话下。至多的话，那就不好说了，编个游戏、弄个网站、甚至以此为业，Impossible is Nothing。真能那样的话，我也算功德一件了。

如果你有任何疑问，没听懂的，觉得我讲得不好的，寂寞了想找人聊天的，都可以直接发消息。反正现在关注的人少，才两位数，收到必回。（更新：现在六位数了）

好了，就这么多。不出意外的话，明天大家就能看到第一个Python程序了。还是那句话，希望我坚持下去的话，请推荐更多的人来关注，虽然只要还有一个人在听，我就尽力坚持，但更多的人听，我就更能坚持啦！

- [返回首页](#)
- [1.安装 →](#)

【Python 第1课】安装

在Windows系统上安装Python的方法还算简单，就比平常装个软件稍稍麻烦一点点。（Mac 也差不多）

1. 下载

进入Python的官方下载页面 <https://www.python.org/downloads/>，你会看到下载按钮和一堆下载链接。我们就直接选“**Download Python 3.7.3**”（选最新的即可），如果没有自动下载，64位系统的同学可以选下面那个“Windows x86-64 executable installer”。如果你不是百分百确定自己是64位系统，请装非64位版本“Windows x86 executable installer”。

由于Python3是今后的主流，不建议安装Python 2。不过想用 py2 版本也可以，请在公众号回复关键字 **2v3**，查看一篇关于2和3之间的一些变化注意事项。

2. 安装

下载之后，就和装其他软件一样，双击，一路Next，想换安装路径的同学可以换个位置。但不管换不换，**请把这个路径复制下来**，比如我的是“**C:\python37**”，后面要用到它。另外有个要注意的是，如果有“**add python.exe to path**”这个选项，请选中它，会让你省不少事。（不同版本这里略有差异）



3. 运行

安装完之后，你应该可以在开始菜单的程序里找到 Python 的文件夹了。里面有一个叫做 **IDLE** 的程序，点击它，就进入了 Python 的开发工具。

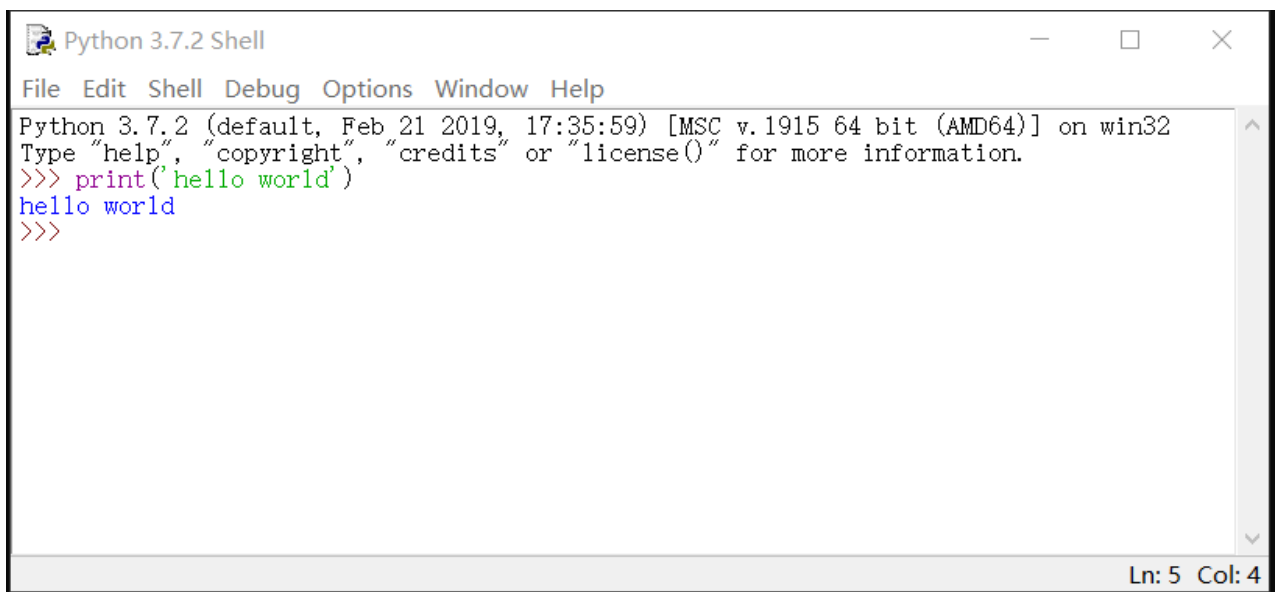


能打开 IDLE，看到里面输出的版本提示信息，就完成 Python 的安装了。

接下来，你就可以写下那句程序员最爱的

```
print('Hello World')
```

向Python的世界里发出第一声啼哭。注意：单引号、双引号都可以，但引号和括号都一定要用英文的标点！

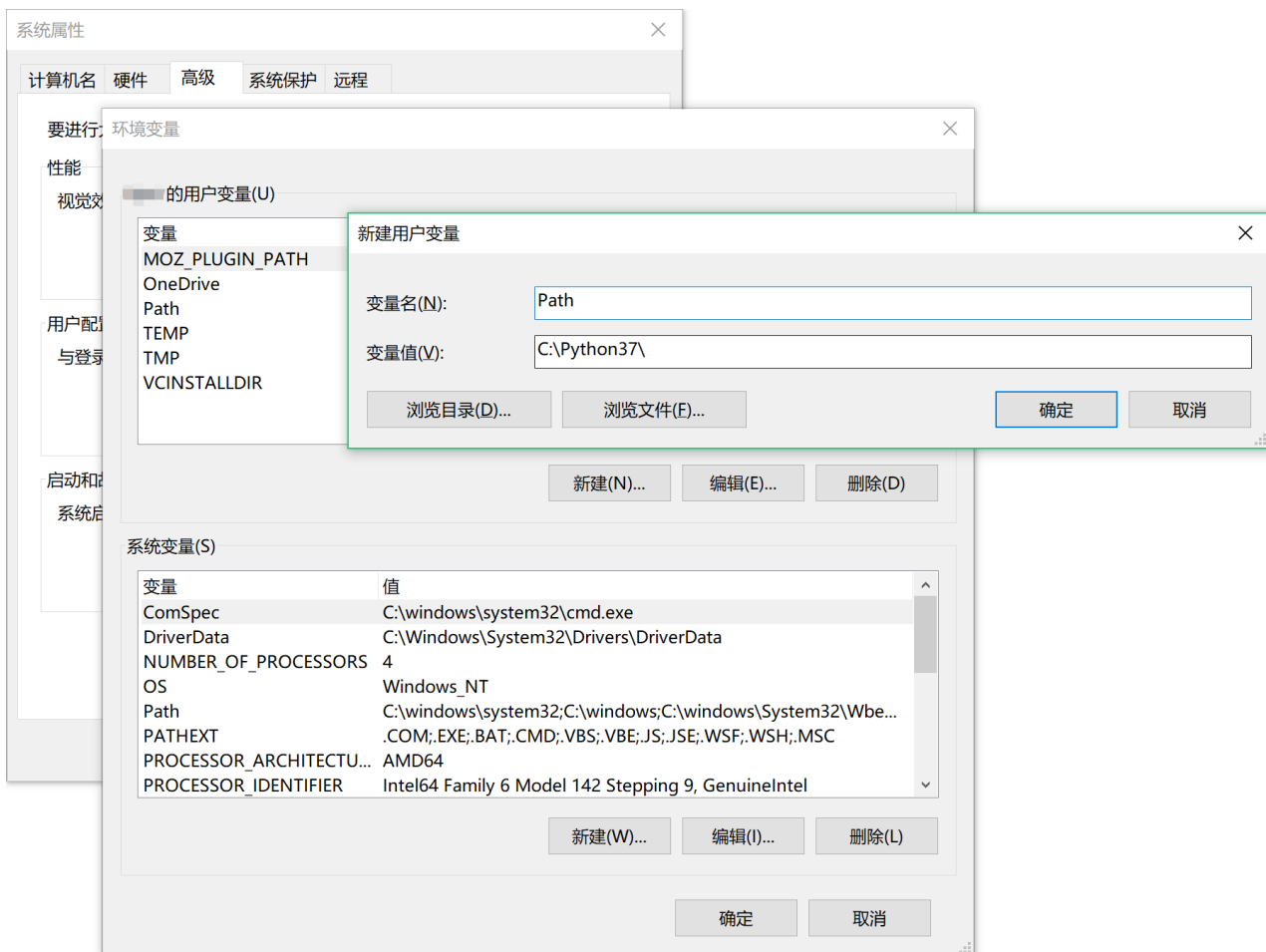


特别说明：有时候，虽然 python 正常安装完毕，但是打开 IDLE 的时候会报错，或者无法正常使用。这时候，首先把报错信息在网上搜索一下，通常都会找到解决方案。如果不行，可以考虑换个早一点的python版本，比如 3.5、3.4 之类的重新安装下（不影响学习）。仍然不行的话，可以考虑再多安装一个 pycharm 软件来写代码（详细说明可在公众号回复关键字 **pycharm**）。你也可以去我们的 [论坛](#) 上寻求帮助。

4. 配置命令行（可选）

说明：完成前3步，你就已经可以开始写 Python 了，所以如果接下来的这一步让你感到头大，可以暂时忽略，基本不影响初期的学习。而且如果你上一步按照我说的，选上了“**add python.exe to path**”，此步骤就已自动完成。这一步的目的是设置 **环境变量**，它的目的是让你能够在系统的命令行里运行 Python，具体是什么意思我暂时先不说得太复杂，大家照着做就好。

右键单击 **我的电脑**，依次点击“**属性**”->“**高级**”->“**环境变量**”（或者直接通过开始菜单的搜索栏搜索“**环境变量**”进入），在“**系统变量**”表单中点击叫做 **Path** 的变量，然后编辑这个变量，把“**；C:\Python37**”，也就是你刚才复制的安装路径，用 **英文分号** 和前面已有的内容隔开，加到它的结尾。然后点确定，点确定，再点确定。完成。



怎么知道你已经配置成功安装了呢？这时候你需要打开 **命令行**，或者叫命令提示符、控制台。方法是：点击 **开始菜单 -> 程序 -> 附件 -> 命令提示符**；或者直接在桌面按快捷键“**Win+r**”，Win键就是Ctrl和Alt旁边那个有windows图标键，输入 **cmd**，回车。这时候你就看到可爱的黑底白字了。

在命令行里输入 **python**，回车。如果看到诸如：

Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32

的提示文字，恭喜你！

```
命令提示符 - python
Microsoft Windows [版本 10.0.17134.765]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\...>python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

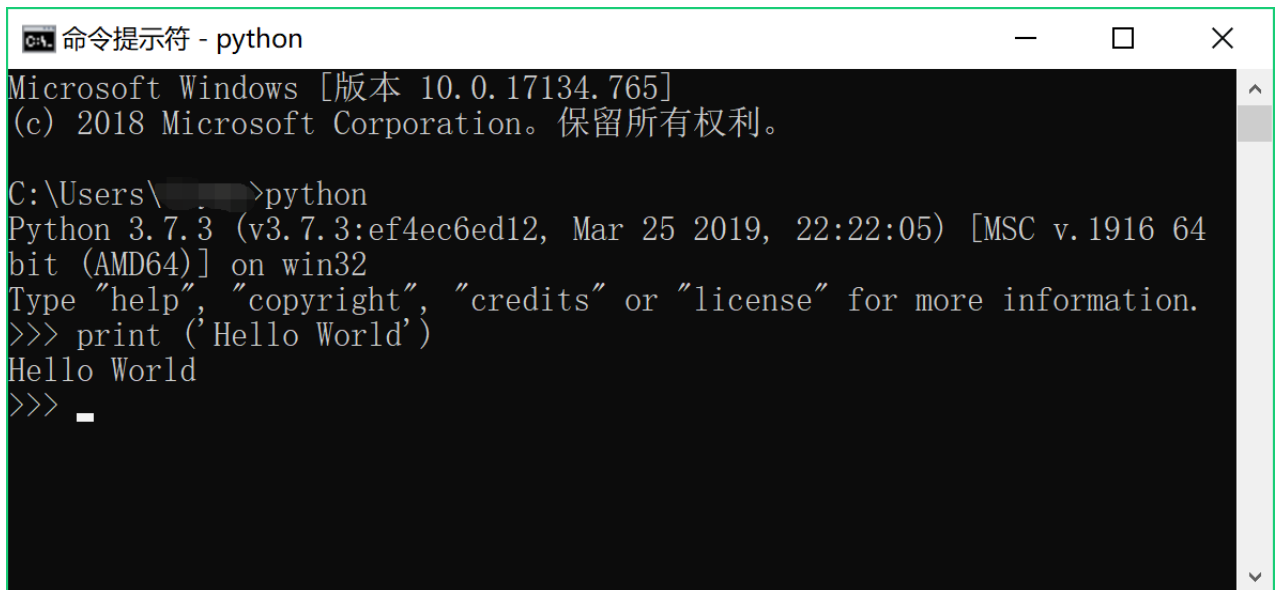
否则，请重新检查你哪里的打开方式不对，或者直接给我留言。

没出现上图效果请务必检查这几项注意：

- 注意1：一定是要用 **英文分号** 和前面已有的内容隔开，记得关闭你的中文输入法
- 注意2： **win7系统** 是右键单击“计算机”，点击“属性”->“高级系统设置”->“环境变量”

- 注意3: 如果不存在Path记录, 就创建一条新的
- 注意4: **win10系统** 的Path不是用分号分隔, 而是需要点击Path后再点击 **新建** 一条记录, 把路径加进去, **无需分号**
- 注意5: 环境变量里会有 **用户变量** 和 **系统变量** 两类, 如果添加后无效, 建议在两类的Path里都加上路径, 并尝试 **重启** 下系统
- 注意6: 设置完要 **重新打开命令行**

命令行里的Python环境一样可以运行代码:



```
命令提示符 - python
Microsoft Windows [版本 10.0.17134.765]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\>python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello World')
Hello World
>>> _
```

嗯。。。如果这么几步你还是被绕晕了, 没关系, 我还留了一手: 从公众号右边菜单里的“**在线编程**”或回复关键字 **code**, 可进入我为你开发的一个在线写python的网页工具, 练习一下python语言。不过在线工具毕竟功能有限, 仅可作为体验, 真正要学习还得在电脑上。

那Mac的同学怎么办? Mac上叫“**终端**”, 英文版叫 **Terminal**, 可以在“应用程序”里找到, 也可以直接在你的Mac上搜索“终端”或者“Terminal”找到。打开之后输入 **idle**, 敲下回车, 就可以进入开发工具; 如果输入 **python**, 就可以打开如上的 python 命令行, 你可以自己亲手试一试。

好了, 今天就这么多, 快去试试你的python, 输出一行“Hello World”吧。完成的同学可以截个屏发给我。欢迎各种建议、讨论和闲聊, 当然更欢迎你在这里分享给更多的朋友。

- [← 0.为什么选择 Python 入门?](#)
- [2.print →](#)

【Python 第2课】print

昨天大家是不是都在自己的电脑上搞定了python环境？或是试用过了在线环境？

今天要讲的东西，昨天课上大家已经见过，就是：**print**（注意：全是小写字母）。

print，中文意思是打印，在python里它不是往纸上打印，而是打印在命令行，或者叫终端、控制台里面。print是python里很基本很常见的一个操作，它的操作对象是一个字符串（什么是字符串，此处按住不表，且待日后慢慢道来）。基本格式是：

```
print(你要打印的东西)
```

这里一定要英文字符的括号，所有程序中出现的符号都必须是英文字符，注意别被你的输入法坑了。

各位同学可以在自己的python环境中试着输出以下内容：

```
>>> print('world')
world
>>> print(1)
1
>>> print(3.14)
3.14
>>> print(3e30)
3e+30
>>> print(1 + 2 * 3)
7
>>> print(2 > 5)
False
```

直接在print后面加一段文字来输出的话，需要给文字加上双引号或者单引号，除此之外的数字、计算式，还有我们以后会提到的变量，都不要加引号。

大家发现，print除了打印文字之外，还能输出各种数字、运算结果、比较结果等。你们试着自己print一些别的东西，看看哪些能成功，哪些会失败，有兴趣的话再猜一猜失败的原因。

其实在python命令行下（每行前面有>>>的地方），print是可以省略的，默认就会输出每一次命令的结果。就像这样：

```
>>> 'Your YiDa!'
'Your YiDa!'
>>> 2+13+250
265
>>> 5<50
True
```

今天内容就这么多。没听出个所以然？没关系，只要成功print出来结果就可以，我们以后还有很多时间来讨论其中的细节。

#===== 课程预告 =====#

昨晚我想了下，如果只是单纯一个个语法、命令讲过去，实在太枯燥了。所以我决定设定一个短期目标，吊一下大家的胃口。

这个短期目标就是一个很简单很弱智的小游戏：

COM: Guess what I think?

5

COM: Your answer is too small.

12

COM: Your answer is too large.

9

COM: Your answer is too small.

10

COM: BINGO!!!

解释一下：首先电脑会在心中掐指一算，默念一个数字，然后叫你猜。你猜了个答案，电脑会厚道地告诉你大了还是小了，直到最终被你果断猜中。

这是我十几年前刚接触编程时候写的第一个程序，当时家里没有电脑，在纸上琢磨了很久之后，熬到第二个星期的电脑课才在学校的486上运行起来。后来我还写过一个windows下的窗口程序版本。现在就让它也成为你们第一个完整的程序吧。照我们每天5分钟的进度，初步估计半个月后大约能完成了。

明天我打算再回到开发环境上，介绍一下编写python的开发工具。工欲善其事，必先利其器嘛。

#===== 课外的话 =====#

今天早上醒来，发现咱们的同学人数一夜之间多了50，后来又陆陆续续来了很多，于是我坚持下去的信心又增加了不少。在这里感谢连客官微的宣传，表示今晚将用加班写代码来表达谢意！

今天新来的同学，可以公众号内回复关键字 **python** 查看已有的课程目录，也可以直接发送数字 **0** 和 **1** 查看前两课的内容。

- [← 1.安装](#)
- [3.IDE →](#)

【Python 第3课】IDE

什么是 IDE？英文叫做 **Integrated Development Environment**，中文就是 **集成开发环境**。嗯，等于没说。

说人话：打个不恰当的比方，如果说写代码是制作一件工艺品，那IDE就是机床。再打个不恰当的比方，PS就是图片的IDE，Word就是doc文档的IDE，PowerPoint就是ppt文件的IDE。python也有自己的IDE，而且还有很多。

python自带了一款IDE，就是我们一开始介绍过的 **IDLE**。Windows上安装 python 了之后，可以在“开始菜单”->“程序”->“Python 3.7”里找到它（或者直接搜索 idle）。

不知道各位同学注意到没有，在这个默认的窗口里，三个箭头 >>> 后面写代码，输一行代码敲回车就会返回结果，没法换行；而且之前 print 了那么多，关掉之后也不知道到哪里去了，重新打开就都没有了。

所以它没法满足我们编写弱智小游戏的大计划。我们需要用新的方法！

如何新建一个文件

点击窗口上方菜单栏的“**File**”->“**New File**”（有些版本是“New Window”），会打一个长得很像的新窗口，但里面什么也没有。这是一个文本编辑器，在这里面就可以写我们的python程序了。在里面写上几行 print 代码，这次可以多 print 一点：

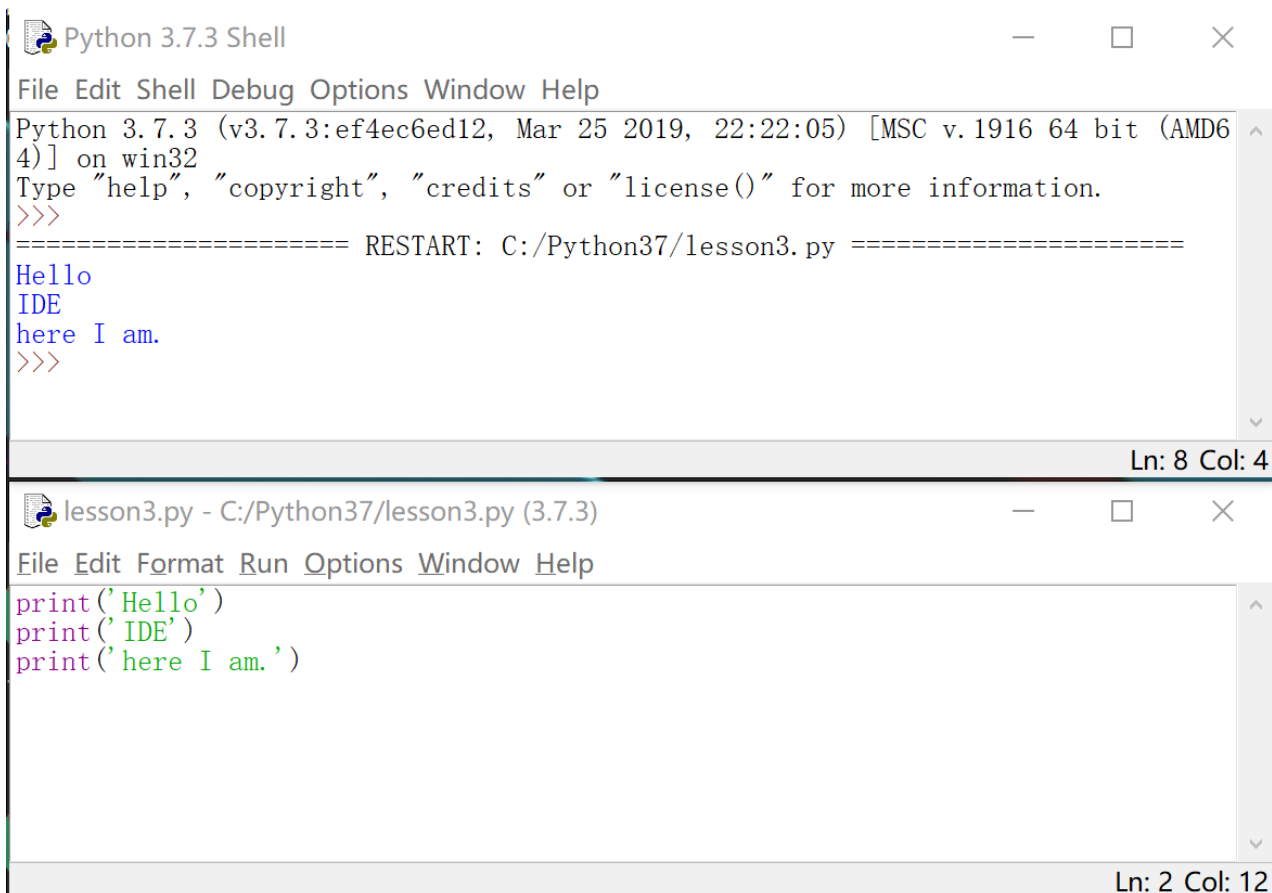
```
print('Hello')

print('IDE')

print('Here I am.')
```

注意1： ①用英文符号②别忘了括号

现在是，见证奇迹的时刻！点击菜单栏的“**Run**”->“**Run Module**”，或者直接按快捷键F5。会提示你保存刚才文件，随便取个名字，比如“lesson3.py”。（.py是python代码文件的类型，虽然不指定.py也是可以的，但建议还按规范来）保存完毕后，之前那个控制台窗口里就会一次性输出你要的结果。



以后想再次编辑或运行刚才的代码，只要在IDLE里选择“**File**”->“**Open...**”，打开刚才保存的.py文件就可以了。

注意2：之后我们写多行代码时，一定要通过新建的代码文件，写好后保存运行。否则直接打开IDLE的环境是无法写多行代码的。

注意3：一开始请不要在代码里写中文，可能会导致无法保存。如果一定要写的话，需要在文件一开始加上一行内容：

```
# coding: gbk
```

Mac上的IDLE是预装好的，在“**终端**”里输入“**IDLE**”就可以启动，使用方法同Windows。也可以在文件夹 `/usr/bin` 里可以找到IDLE。如果是重新下载安装了python，有些版本是可以在“应用程序”里找到IDLE的（一个小火箭图标）。（有同学反馈说他用的 Mac 上默认的 Python 版本里是不带 IDLE 的，需要自行下载安装 Python 后里面才有，那请参考 [安装](#) 课程里的方法下载安装。）

除了官配的 IDLE，还有一些很好用的第三方 IDE，把文件管理、文本编辑器、命令行都整合到了一起，还增加了很多辅助功能比如代码提示、自动补全和跳转等，配置好之后用起来比 IDLE 更爽。这其中首推 **PyCharm**，它之前是收费软件，现在已经推出了免费版本，足够一般的学习和开发使用。有兴趣的同学也可以去找来试试看。在公众号里回复关键字 **pycharm**，可以看到之前写过的相关介绍文章。

今天的内容有点长。配置开发环境这种事最麻烦了，也是自学时候劝退率最高的环节，大家耐心一点，毕竟一次投入，长期受益。以后我们的课程都会在IDE中进行。

最后说下，有一些python程序员不使用任何IDE。至于原因嘛，可能就像优秀的手工艺人是不会用机床来加工艺术品的吧。

#===== 课外的话 =====#

昨天的课发出去之后，有不少同学发来了反馈，有完成截图的，也有遇到问题的。一些问题突然让我意识到，很多地方自己描述得不是很到位，会产生歧义，或者干脆就很难听懂。比如：Mac上有控制台（console）和终端（Terminal）之分。我想说的其实是终端。Mac的同学们请注意。

另外，前面的课程我们是在Python命令行（有>>>的环境下）运行，我在文章里面的例子是在命令行里一行一行的输入得到的效果，有同学误以为全都是输入，一起贴到在线编辑器里，结果就报错得不到结果。

因此在这里，我特别要申明一下：如果你发现照我说的去做，没有得到预期的结果，那多半是我没说清。千万不要觉得为什么编程这么难，搞了半天也不对。导致错误的原因，往往只是一点点小偏差，稍微改一下就好了。（顺便提一句，今天下午我工作的时候就因为一个单词拼错了，折腾了半天代码）

所以嘛，有问题不要一直自己闷着头纠结，多沟通一下就好了。人生中的事情，大抵如此，做人嘛，最重要的是要开心啦……咳咳。

另外，为了让大家更好地回顾讲过的内容，以及有问题时候方便讨论交流，我建了一个论坛：bbs.crossincode.com，你们可以在论坛相关的帖子下讨论课程内容，或者单独发帖提问。

- [← 2.print](#)
- [4.输入 →](#)

【Python 第4课】输入

Hi~我Crossin又来了。

可以用编程语言让计算机按你说的指令做事情之后，大家是不是有些跃跃欲试呢？别着急，先回顾一下我们之前几节课。我们到现在一共提到了三种可以运行print的方式：

1. 打开 IDLE，直接在 `>>>` 后面输入 `print` 语句并回车执行。但是这种方法很难帮我们实现写一个完整小程序的目标。
2. 在 IDLE 里新建一个代码文件，写好 `print` 语句后，保存并运行。以后我们课程里的内容，你都可以用此方法进行。不知道大家是不是都顺利搞定，并且能顺利保存并打开py文件了呢？
3. 也可以直接在命令行里运行 `python` 命令，进入 `python` 命令行的交互环境。命令行，包括Win下的控制台（CMD）和Mac下的终端（Terminal）。

大家是不是都准备好自己的武器了呢？那我们接下来就要正式开战啦！

#===== 进入今天的正题 =====#

之前print了那么多，都是程序在向屏幕“输出”。那有来得有往，有借得有还，有吃。。。咳咳！那啥，我们得有向程序“输入”信息的办法，才能和程序对话，进行所谓的“人机交互”。

python有一个接收命令行下输入的方法：

```
input()
```

注意，和 `print` 一样，我们必须得加上`()`，而且得是英文字符的括号。

好了，终于可以搬出那个弱智小游戏了，耶！

游戏里我们需要跟程序一问一答，所以我们先把话给说上。

打开我们的python编辑器，不管是IDLE，还是其他的IDE。新建一个代码文件，在其中输入下面几句代码：

```
print("Who do you think I am?")

input()

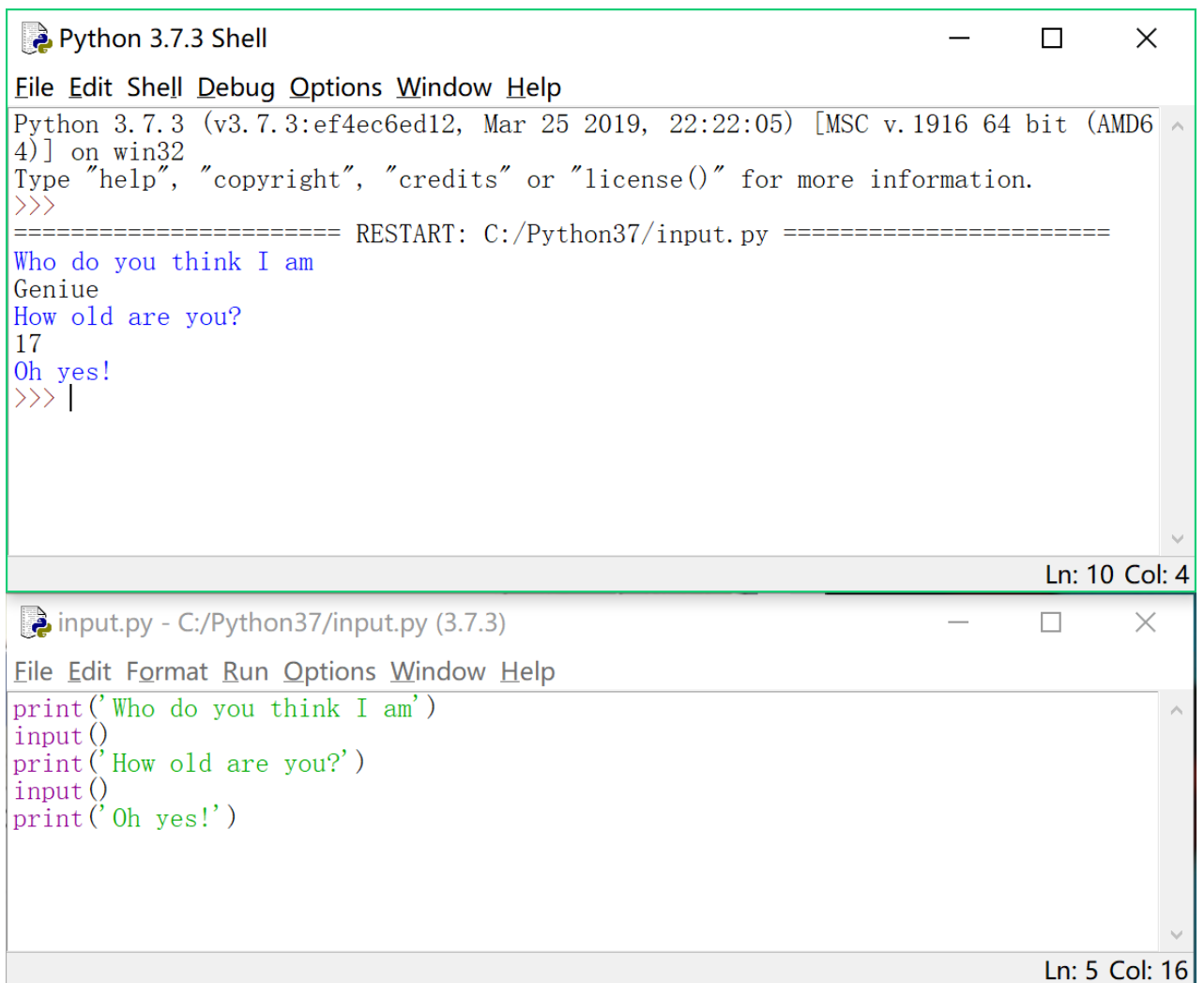
print("Oh, yes!")
```

然后，点击 `Run -> Run module`。你会在命令行中看到，程序输出了第一句之后就停住了，这是 `input` 在等待你的输入。

注意1：这里的代码请新建一个代码文件后，在其中编写并运行，否则无法换行并获取输入（参见之前 [IDE](#) 那课中的说明）

注意2：这里是在控制台里输入内容后回车，而不是把你要输的东西提前写在括号里！切记！括号里可以加文字，但那只是用来作为输入的提示语。`input` 不是提前代码里写死输入内容，而是在程序运行时动态获取用户的输入。

输入你的回答，敲回车。你会看到程序的回答。



The image shows two screenshots of a Python 3.7.3 environment. The top screenshot is a 'Python 3.7.3 Shell' window. It displays the Python version and build information, followed by a prompt 'Type "help", "copyright", "credits" or "license()" for more information.' Below this, there's a separator line '===== RESTART: C:/Python37/input.py ====='. The user has entered 'Who do you think I am' and 'Geniue' (misspelled), and the program has responded with 'Oh yes!'. The bottom screenshot is an IDE window titled 'input.py - C:/Python37/input.py (3.7.3)'. It shows the source code of the script:

```
print('Who do you think I am')
input()
print('How old are you?')
input()
print('Oh yes!')
```

图中的代码额外有增加了一问一答，蓝色字是程序输出，黑色字是我们运行时的输入。

注意3: py3里的input()得到的都是字符串。如果输入了数字，得到的也是含有这个数字的字符串，而非数值。

看上去不错哦，似乎就这么对上话了。是不是觉得离小游戏的完成迈进了一大步？可是大家发现没有，即使你说"Idiot!", 程序仍然会淡定地回答"Oh, yes!"因为它左耳进右耳出，根本就没听进去我们到底说了啥。那怎么才能让它认真听话呢？

啪！且听下回分解。

#===== 课外的话 =====#

PS: 今天心情不错，给大家讲个很冷的程序员笑话。

一位程序员对书法十分感兴趣，退休后决定在这方面有所建树。于是花重金购买了上等的文房四宝。一日，饭后突生雅兴，一番磨墨拟纸，并点上了上好的檀香，颇有王羲之风范，又具颜真卿气势，定神片刻，泼墨挥毫郑重地写下一行字：

hello world

- [← 3.IDE](#)
- [5.变量 →](#)

【Python 第5课】变量

昨天说到，需要让程序理解我们输入的东西。那首先，就需要有东西把我们输入的内容记录下来，好为接下来的操作做准备。

Python之神说，要有变量！于是就有了变量。

变量，望文生义，就是可变化的量。python里创建一个变量的方法很简单，给它起个名字，然后给它一个值。举起几个栗子：

```
name = 'Crossin'
myVar = 123
price = 5.99
visible = True
```

“=”的作用是把右边的值赋予给左边的变量。

这里说一下另外一个概念，叫做“数据类型”，上面4颗栗子分别代表了python中较常见的四种基本类型：

- 字符串：表示一串字符，需要用"单引号或"双引号包围起来
- 整数
- 浮点数：就是小数
- bool（布尔）：这个比较特殊，是用来表示逻辑上的“真”和“假”（或者说“是”和“非”）的一种类型，它只有两个值，**True** 和 **False**。（注意：这里没有引号，有了引号就变成字符串了）

再次用到我们熟悉的 print。这次，我们升级了，要用print输出一个“变量”：

```
name = 'Crossin'
print(name)
```

看到结果了吗？没有输出“**name**”，也没有报错，而是输出了“**Crossin**”。

注意：name不需要加引号，不然它也就成了一个字符串，而不是变量

现在想一想：为什么之前 print 一段文字，如果没加引号就会报错，而 print 一个数字就没有问题呢？

它叫变量，那就是能变的。所以在一次“赋值”操作之后，还可以继续给它赋予新的值，而且可以是不同类型的值。

```
a = 123
print(a)
a = 'hi'
print(a)
```

“=”的右边还可以更复杂一点，比如是一个计算出的值：

```
value = 3 * 4
print(value)
value = 2 < 5
print(value)
```

甚至，也可以是input()：

```
name = input()
print(name)
```

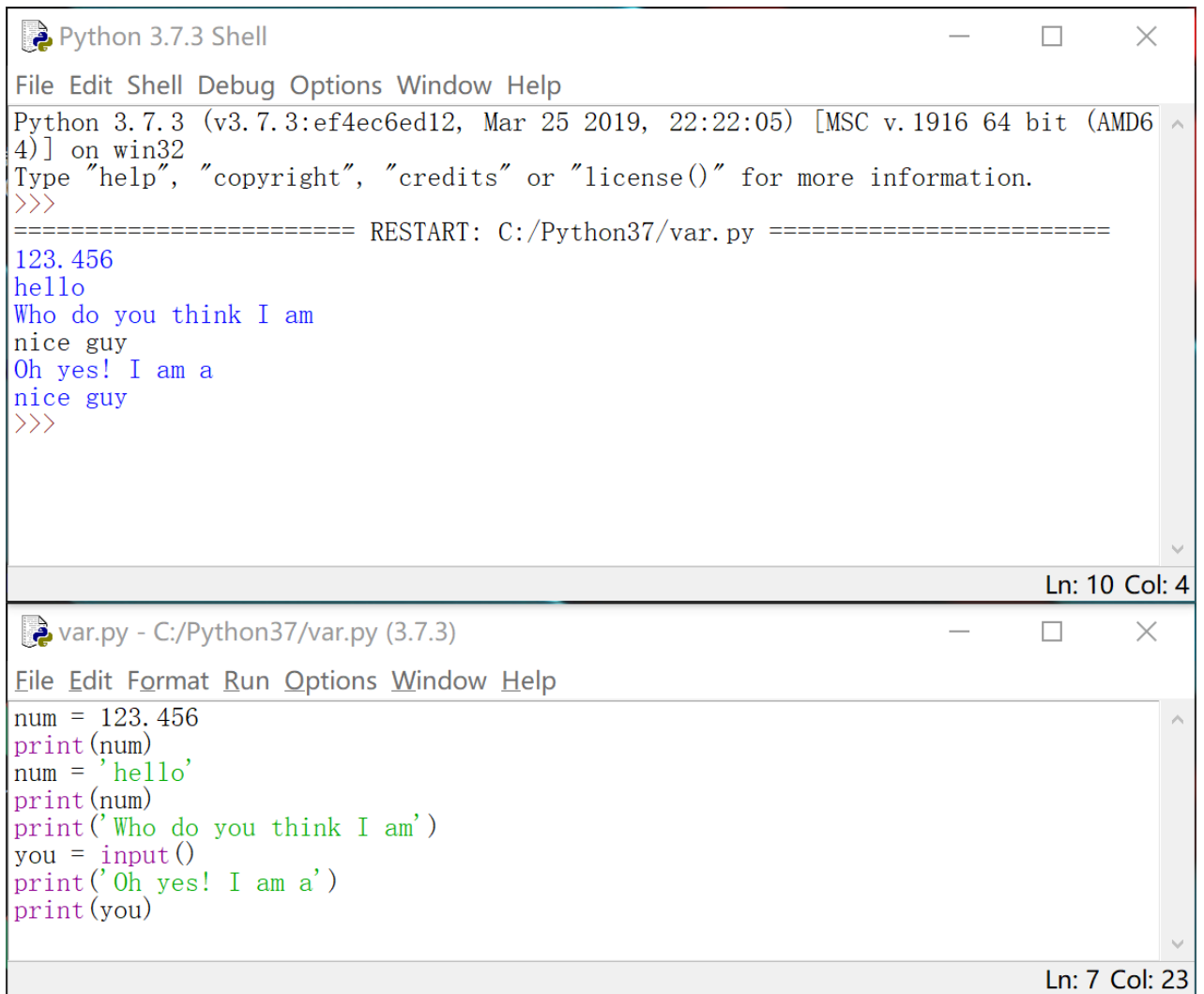
于是，我们又可以进化一下我们的小游戏了。把上次写的内容稍微改一下，加上变量：

```
print("Who do you think I am?")
```

```
you = input()

print("Oh, yes! I am a")

print(you)
```



Python 3.7.3 Shell

File Edit Shell Debug Options Window Help

Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Python37/var.py =====
123.456
hello
Who do you think I am
nice guy
Oh yes! I am a
nice guy
>>>

Ln: 10 Col: 4

var.py - C:/Python37/var.py (3.7.3)

File Edit Format Run Options Window Help

```
num = 123.456
print(num)
num = 'hello'
print(num)
print('Who do you think I am')
you = input()
print('Oh yes! I am a')
print(you)
```

Ln: 7 Col: 23

看来程序已经知道我们的输入了。接下来，就要让它学会对不同的答案做出判断。这个我们留到下次再说。

#===== 课外的话 =====#

有些同学在用我们的 [在线编辑器](#) 或者其他类似的网页工具在写代码。但有个不幸的消息是：绝大多数在线编辑器没办法完全模拟控制台输入输出的操作（不支持 `input`）。所以，你可以通过它来简单的体验和学习，但是想跟着我一起完成那个弱智小游戏，还是必须给你的电脑装上python。

今天是周五。我觉得吧，到周末了，大家应该远离一下电脑，多陪陪家人朋友，吃吃饭，出去走走。祝大家周末愉快！

- [←4.输入](#)
- [6.bool→](#)

【Python 第6课】bool

昨天说到了python中的几个基本类型，字符串、整数、浮点数都还算好理解，关于剩下的那个 **bool**（布尔值）我要稍微多说几句。

逻辑判断 在编程中是非常重要的。大量的复杂程序在根本上都是建立在“真”与“假”的基本逻辑之上。而 bool 所表示的就是这种最单纯最本质的 **True/False**，真与假，是与非。

来看下面的例子：

```
a = 1 < 3

print(a)

b = 1

c = 3

print(b > c)
```

通过用“>”“<”来比较两个数值，我们就得到了一个bool值。这个bool值的真假取决于比较的结果。

“>”“<”在编程语言中被称为 **比较运算符**（或叫 **关系运算符**），常用的比较运算符包括：

>	大于
<	小于
>=	大于等于
<=	小于等于
=	等于（比较两个值是否相等。之所以用两个等号，是为了和变量赋值区分开来）
!=	不等于

还有一种 **逻辑运算符**：

not	逻辑“非”	如果 x 为 True，则 not x 为 False
and	逻辑“与”	如果 x 为 True，且 y 为 True，则 x and y 为 True
or	逻辑“或”	如果 x、y 中至少有一个为 True，则 x or y 为 True

比较运算符和逻辑运算符的结果都是 bool 类型的值。

关于bool值和逻辑运算其实远不止这些，但现在我们暂时不去考虑那么多，以免被绕得找不到北。最基本的 **大于、小于、等于** 已经够我们先用一用的了。

试试把bool加到我们的小游戏里：

```
num = 10

print('Guess what I think?')

answer = int(input())

result = answer<num

print('too small?')

print(result)

result = answer>num

print('too big?')

print(result)

result = answer==num
```

```
print('equal?')

print(result)
```

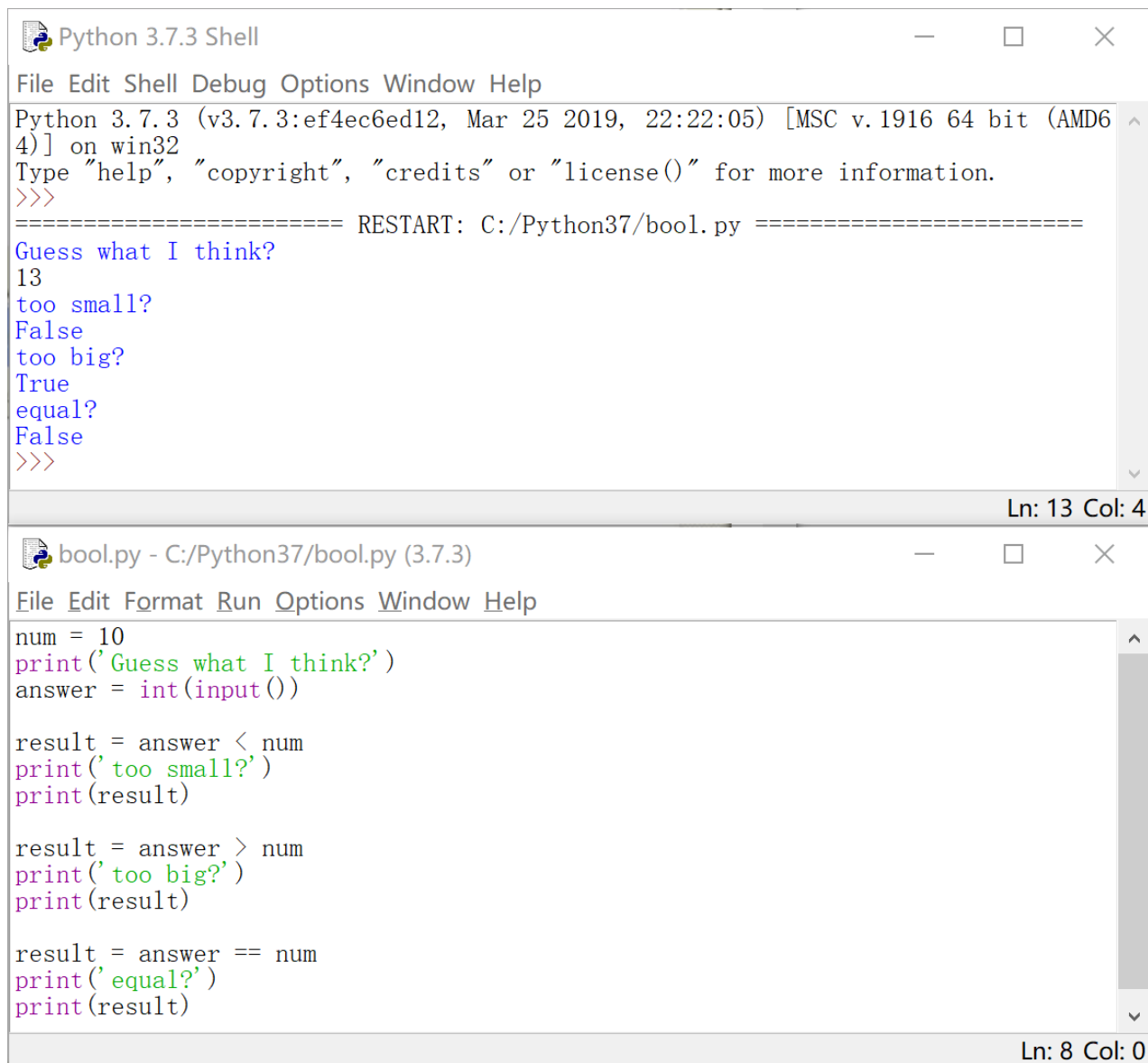
代码比之前稍微多了一点，解释一下。

第一段代码：先创建一个值为10的变量 `num`，输出一句提示，然后再输入一个值给变量 `answer`。（因为`input`拿到的值是字符串而不是数字，这里我们需要将`input`的结果强行转成整数类型`int`）

第二段代码：计算 `answer < num` 的结果，记录在 `result` 里，输出提示，再输出结果。

第三段、第四段都与第二段类似，只是比较的内容不一样。

注意：当你自己写这段代码的时候，要确保不要有笔误，比如拼错单词，漏掉等号、引号、括号……。这种情况在新手身上屡有发生）



```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Python37/bool.py =====
Guess what I think?
13
too small?
False
too big?
True
equal?
False
>>>
Ln: 13 Col: 4

bool.py - C:/Python37/bool.py (3.7.3)
File Edit Format Run Options Window Help
num = 10
print('Guess what I think?')
answer = int(input())

result = answer < num
print('too small?')
print(result)

result = answer > num
print('too big?')
print(result)

result = answer == num
print('equal?')
print(result)
Ln: 8 Col: 0
```

看看结果是不是跟你预期的一致？虽然看上去还是有点傻，但是离目标又进了一步。（这里不管正确与否，三个答案都会被输出。如果你只要它输出正确的那一个结果，学到后面课程就知道了）

现在数数你手上的工具：**输入**、**输出**，用来记录数值的**变量**，还有可以比较数值大小的**逻辑运算**。用它们在你的python里折腾一番吧。

#————— 课外的话 —————#

闲扯还是要的。有同学问，为什么这个语言要叫python。这个嘛，它肯定不是我起的。python，读作“派森”（差不多啦），中文意思“巨蟒”。其实是一个喜剧团体用了“Monty Python”这个名字，而python的创造者（Guido van Rossum 老爷子）又是他们的电视节目《Monty Python and the Flying Circus》（巨蟒飞行马戏团）的粉丝。当他还在自娱自乐地折腾python的雏形时，就拿来了名。所以，你要是发明了一种语言，也可以命名个 GoT、TBBT、zhenhuan之类的。

- \leftarrow 5.变量
- 7.if \rightarrow

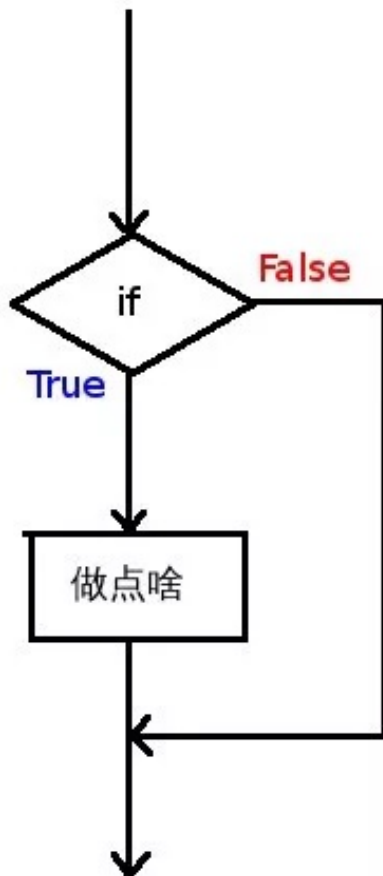
【Python 第7课】if

继续上次的程序。我们已经能让程序判断我们输入的值了，但这程序还是有点呆，不管怎样都要把话说三遍。因为到目前为止，我们的程序都是按照顺序从上到下一行接一行地执行。有同学发来问题了：怎么能让它根据我们输入的结果来选择执行呢？

答案就是：**if**

if，英文翻译过来就是**如果**。

来看一张图（纯手绘，渣画质）



解释一下：程序顺序往下执行遇到**if语句**的时候，会去判断它所带**条件的真假**。

“如果”条件为True，就会去执行接下来的内容。“如果”条件为False，就跳过。

语法为：

if 条件：

选择执行的语句

特别说明：条件后面的**冒号不能少**，同样必须是**英文标点**。

特别特别说明：if内部的语句需要有一个统一的**缩进**（就是指每一行开头的空格），一般用4个空格。缩进表示这些代码属于这个if条件内部，是一个“代码块”。python用这种方法替代了其他很多编程语言中的大括号{}。

你也可以选择1/2/3...个空格或者按一下tab键，但必须整个文件中都统一起来。**千万不可以tab和空格混用**，不然就会出现各种莫名其妙的错误。所以建议都直接用4个空格。

如果if中的代码块又再包含一个if，那就需要再进一步缩进一次。一个代码中的缩进需要统一，比如每次都是增

加4个空格。

上栗子：

```
# coding: gbk

age = int(input())

if age >= 18:

    print("你是个成年人了！")
```

注意1：这里的代码请新建一个代码文件后，在其中编写并运行，否则无法换行及缩进（参见之前 [IDE](#) 课中的说明）

注意2：由于代码中有中文，所以我们在开头加上了一行编码申明（同样参见之前 [IDE](#) 课中的说明）

试试看：当你输入一个大于等于18的数字时就会有输出，否则什么也没有。想想是为什么？

所以现在，我们的游戏可以这样改写：

```
num = 10

print ('Guess what I think?')

answer = int(input())

if answer<num:

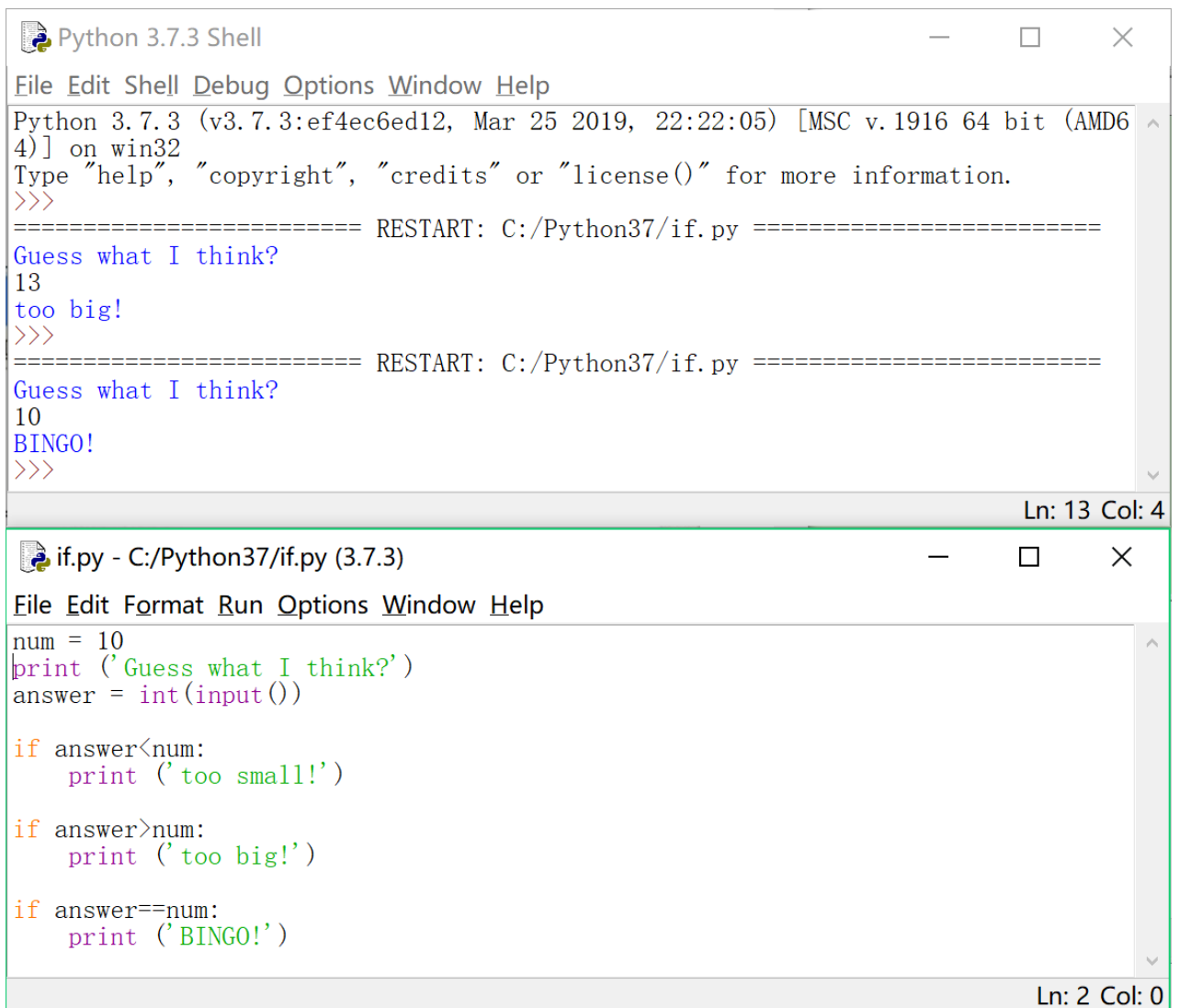
    print ('too small!')

if answer>num:

    print ('too big!')

if answer==num:

    print ('BINGO!')
```



```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Python37/if.py =====
Guess what I think?
13
too big!
>>>
===== RESTART: C:/Python37/if.py =====
Guess what I think?
10
BINGO!
>>>
Ln: 13 Col: 4

if.py - C:/Python37/if.py (3.7.3)
File Edit Format Run Options Window Help
num = 10
print ('Guess what I think?')
answer = int(input())

if answer<num:
    print ('too small!')

if answer>num:
    print ('too big!')

if answer==num:
    print ('BINGO!')
Ln: 2 Col: 0
```

if在编程语言中被称为“**控制流语句**”，用来控制程序的执行顺序。还有其他的控制流语句，后面的课程中我们会陆续介绍。

#===== 课后作业 =====#

有不少同学强烈要求布置作业。好吧，满足你们。还记得之前那个“*你觉得我是什么人？(Who do you think I am)*”的程序吧？（不记得的请戳[4.输入](#)）

改写一下，只有你回答某些好话的时候，程序才会回复“**Oh yes**”，其他都不理你。甚至说某些词的时候，它还要反驳你。

至于用哪些词哪些条件，你们自己设定吧。

#===== 课外的话 =====#

学会了if，有一个好处，就是你能听懂下面这个笑话了：

老婆给当程序员的老公打电话：“下班顺路买一斤包子带回来，如果看到卖西瓜的，就买一个。”

当晚，程序员老公手捧一个包子进了家门.....

老婆怒道：“你怎么就买了一个包子？！”

老公答曰：“因为看到了卖西瓜的。”

- [← 6.bool](#)
- [8.while →](#)

【Python 第8课】while

先介绍一个新东西：注释

python里，以“#”开头的文字会被忽略，不会被认为是可执行的代码。

```
print ("hello world")
```

和

```
print ("hello world")    #输出一行字
```

是同样的效果。但后者可以帮助开发者更好地理解代码。

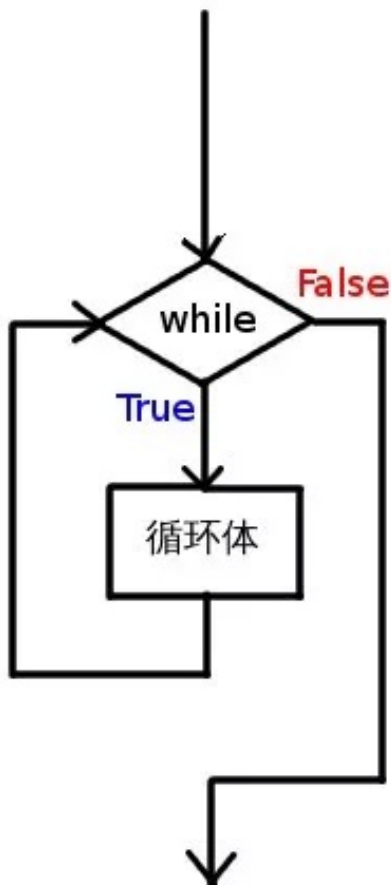
在接下来的课程中，我会经常用注释来解释代码。

#————— 进入今天的正题 —————#

用改进完我们的小游戏后，功能已经基本实现了。很多同学做完后纷纷表示，每次只能猜一次，完了之后又得重新run，感觉好麻烦。能不能有办法让玩家一直猜，直到猜中为止？答案很显然：如果这种小问题都解决不了，那python可就弱爆了。

最简单的解决方法就是：**while**

同 if 一样，while 也是一种控制流语句，另外它也被称作循环语句。继续来看渣画质手绘流程图：



while，英文翻译过来就是“当...的时候”。

程序执行到 **while** 处，“当”条件为 True 时，就去执行 while 内部的代码；“当”条件为 False 时，就跳过。

语法为：

while 条件:

循环执行的语句

同 if 一样，注意冒号，注意缩进。

今天的栗子:

```
# coding: gbk

a = 1          # 为了后面的条件能满足，先把a设为1

while a != 0:   # 如果a不等于0就循环（1不等于0）

    print("please input")

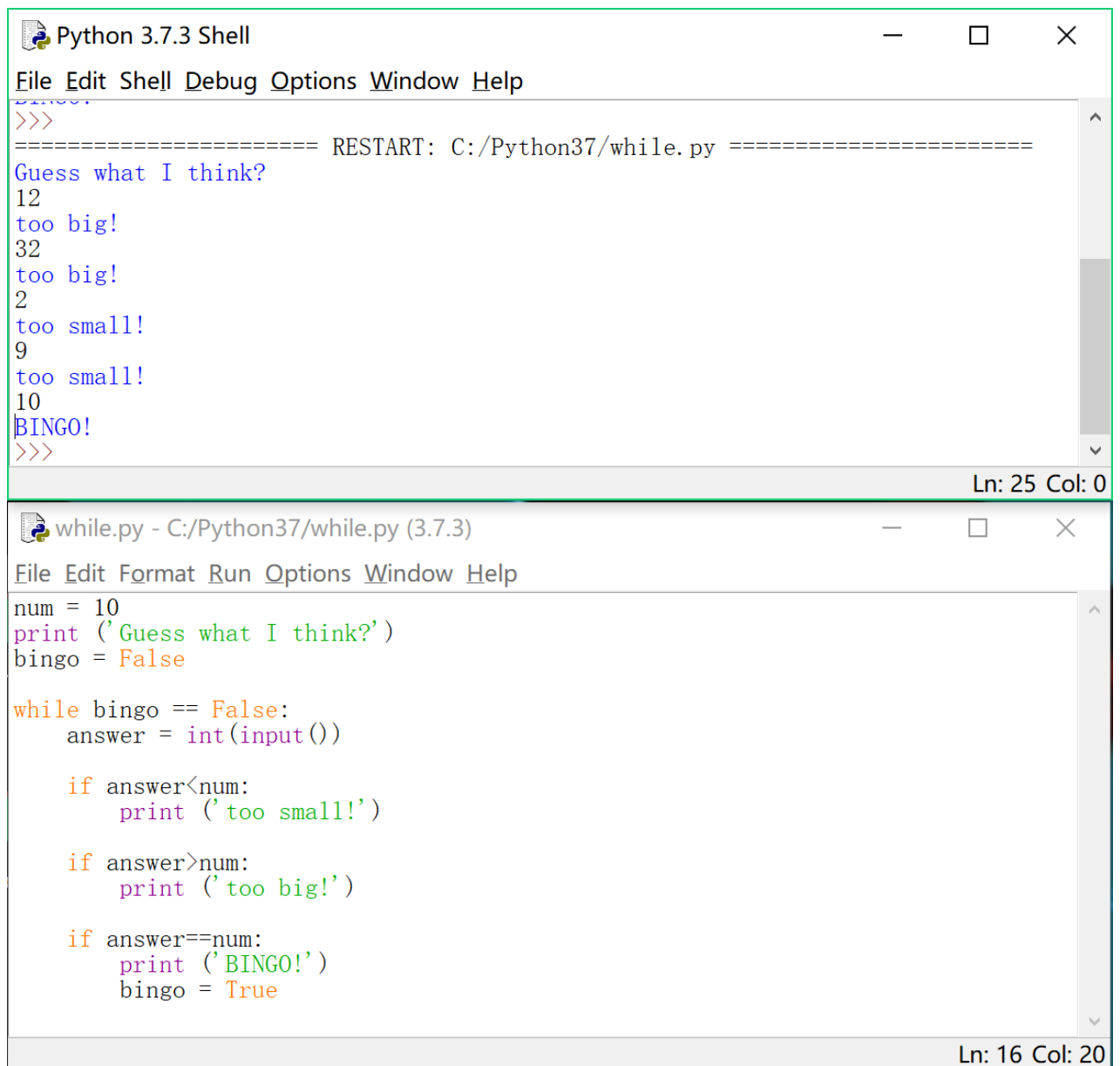
    a = int(input())    # 在循环内部获取输入，改变a的值（想想看不改会怎样？）

print("over")
```

程序执行后，会不断向你询问输入，直到你输入0，条件 **a!=0** 不满足，循环结束。

现在，想想怎么用while改进小游戏？有多种写法，大家自己思考下，我不多做说明了。

下图给出一种方法。



The image shows two screenshots of a Python 3.7.3 environment. The top screenshot is a terminal window titled 'Python 3.7.3 Shell' showing the execution of a script. It prompts 'Guess what I think?' and shows user inputs 12, 32, 2, 9, and 10, with corresponding outputs 'too big!', 'too big!', 'too small!', 'too small!', and 'BINGO!'. The bottom screenshot is a code editor window titled 'while.py - C:/Python37/while.py (3.7.3)' showing the source code for the game. The code uses a 'bingo' variable to track if the user has guessed correctly.

```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
>>>
===== RESTART: C:/Python37/while.py =====
Guess what I think?
12
too big!
32
too big!
2
too small!
9
too small!
10
BINGO!
>>>
Ln: 25 Col: 0

while.py - C:/Python37/while.py (3.7.3)
File Edit Format Run Options Window Help
num = 10
print ('Guess what I think?')
bingo = False

while bingo == False:
    answer = int(input())

    if answer<num:
        print ('too small!')

    if answer>num:
        print ('too big!')

    if answer==num:
        print ('BINGO!')
        bingo = True
Ln: 16 Col: 20
```

解释下:

1. 代码中，我们用了一个叫做 bingo 的变量，来记录是否猜中了结果，猜中了就是 True，没猜中就是 False (bingo 本身就是有“赢了”的意思，你也可以命名为 caizhong，这个随意)。

2. 一开始，我们给 bingo 赋值为 False，使程序可以进入循环。
3. 每次循环中，我们都会输入一遍 answer，然后判断是大是小还是相等。如果相等了，再此条件的代码块里面增加一句 bingo = True，修改 bingo 的值，使得程序在下一次循环判断的时候，发现 bingo == False 这个循环条件不再满足，于是程序结束。

注意：这里出现了两层缩进，要保持每层缩进的空格数相同。

到此为止，小游戏已经基本成型了。不过好像还差一点：每次自己都知道答案，这玩起来有神马意思。

明天来讲，怎么让你不知道电脑的答案。

PS：如果你对本课中的 bingo = False、bingo == False 和 bingo = True 感到困惑，先别着急挠头，我们会在后面 [11.逻辑判断](#) 课程中进一步详细解释。

#===== 作业练习 =====#

你可以在公众号 **Crossin的编程教室** 中的 **课外辅导** 栏目里找到一些 [练习题](#)，学了 while 之后，你可以试一试完成 3~6 题。

- [← 7.if](#)
- [9.random →](#)

【Python 第9课】random

之前我们用了很多次的 `print` 和 `input` 方法，它们的作用是实现控制台的 **输入** 和 **输出**。除此之外，python还提供了很多 **模块**，用来实现各种常见的功能，比如时间处理、科学计算、网络请求、随机数等等等等。

今天我就来说说，如何用python自带的 **随机数模块**，给我们的小游戏增加不确定性。

引入模块的方法：

```
from 模块名 import 方法名
```

看不懂没关系，这东西以后我们会反复用到。今天你只要记住，你想要产生一个随机的整数，就在程序的最开头写上：

```
from random import randint
```

之后你就可以用 **randint** 来产生随机数了。

还记得input后面的()`吗`，我们使用randint的时候后面也要有()`。而且`，还要在括号中提供两个数字，先后分别是产生 **随机整数范围的下限和上限**。例如：

```
randint(5, 10)
```

这样将会产生一个5到10之间（包括5和10）的随机整数。

放到我们的小游戏里，用

```
num = randint(1, 100)
```

替代

```
num = 10
```

程序在运行时候，会产生一个1到100的随机整数，存在num里，每次运行都不一样，我们也不知道是多少，真的全靠猜了。

Python 3.7.3 Sh...

File Edit Shell Debug Options Window Help

Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python37\guess_random.py =====
=====
Guess what I think?
50
too small!
70
too small!
90
too big!
80
too big!
77
too big!
76
too big!
71
BINGO!
>>>

Ln: 20 Col: 4

guess_random.py - ...

File Edit Format Run Options Window Help

from random import randint
num = randint(1, 100)

print('Guess what I think?')
bingo = False

while bingo == False:
 answer = int(input())

 if answer < num:
 print('too small!')

 if answer > num:
 print('too big!')

 if answer == num:
 print('BINGO!')
 bingo = True

Ln: 1 Col: 0

好了，觉得还有点意思么？

我们终于一步步把这个弱智小游戏给做出来了，有没有一丁点的成就感呢？

如果你对其中的某些细节还不是很理解，恭喜你，你已经开始入门了。相信你会带着一颗追求真相的心，在编程这条路上不断走下去。

我们的课程，也才刚刚开始。

- [← 8.while](#)
- [10.变量2 →](#)

【Python 第10课】变量2

变量这东西，我们已经用过。有了变量，就可以存储和计算数据。今天来讲点变量的细节。

#—— 变量命名规则 ——#

变量名不是你想起就能起的：

1. 第一个字符必须是 字母 或者 下划线
2. 剩下的部分可以是 字母、下划线 或 数字0~9
3. 变量名称是对 大小写敏感 的，myname 和 myName 不是同一个变量。

几个有效的栗子：

```
i
__my_name
name_23
a1b2_c3
```

几个坏掉的栗子（想一下为什么不对）：

```
2things
this is spaced out
my-name
```

#—— 变量的运算 ——#

我们前面有用到变量来存储数据：

```
num = 10
answer = input()
```

也有用到变量来比较大小：

```
answer < num
```

除此之外，变量还可以进行数学运算：

```
a = 5
b = a + 3
c = a + b
```

python中运算的顺序是，先把“=”右边的结果算出了，再赋值给左边的变量，相当于是两个步骤。

计算规则本身的顺序和数学中一样，先乘除后加减，有括号先算括号里。（对于拿不准顺序的计算，尽量加上括号）

下面这个例子：

```
a = 5
a = a + 3
print(a)
```

你会看到，输出了8，因为先计算出了右边的值为8，再把8赋给左边的a。通过这种方法，可以实现 累加求和 的效果。

它还有个简化的写法：

```
a += 3
```

这个和

```
a = a + 3
```

是一样的。

我们之前的猜数字游戏，就可以用此方法加上一个记录猜了多少次的功能：

```
from random import randint

num = randint(1, 100)

print('Guess what I think?')

bingo = False

count = 0

while bingo == False:

    count += 1

    answer = int(input())

    if answer < num:

        print('too small!')

    if answer > num:

        print('too big!')

    if answer == num:

        print('BINGO!')

        bingo = True

print(count)
```

于是，利用变量、循环、累加，可以写一个程序，来完成传说中高斯大牛在小时候做过的题：

$1+2+3+\dots+100=?$

从1加到100等于多少？

提示：你可以用一个变量a记录现在加到几了，再用一个变量b记录加出来的结果，通过while来判断是不是加到100了。

- [← 9.random](#)
- [11.逻辑判断 →](#)

【Python 第11课】逻辑判断

之前粗略地提到 **bool** 类型的变量，又说到 if 和 while 的判断条件。有些同学反馈说没怎么理解，为什么一会儿是 bingo=False，一会又是 bingo==False，一会儿是 while 在条件为 True 的时候执行，一会儿又是 while 在 bingo==False 的时候执行。别急，你听我说。

首先，要理解，一个逻辑表达式，其实最终是代表了一个bool类型的结果，比如：

```
1 < 3
```

这个就像当于是一个True的值

```
2 == 3
```

这个就是False

把它们作为判断条件放到 if 或者 while 的后面，就是根据他们的值来决定要不要执行。

同样的栗子再来几颗：

```
a = 1
```

```
print(a>3) #False
```

```
print(a==2-1) #True
```

```
b = 3
```

```
print(a+b==2+2) #True
```

比较容易搞混的，是bool变量的值和一个逻辑表达式的值，比如：

```
a = False
```

```
print(a) #False
```

```
print(a==False) #True
```

虽然 a 本身的值是 False，但是 a==False 这个表达式的值是True。（说人话！）“a”是错的，但“a是错的”这句话是对的。

回到上面那几个概念：

```
bingo=False
```

把bingo设为一个值为False的变量

```
bingo==False
```

判断bingo的值是不是False，如果是，那么这句话就是True

while 在判断条件条件为 True 时执行循环，所以当 bingo==False 时，条件为 True，循环是要执行的。

晕了没？谁刚学谁都晕。不晕的属于骨骼惊奇百年一遇的编程奇才，还不赶紧转行做程序员！

逻辑这东西是初学编程的一大坑，我们后面还要在这个坑里挣扎很久。

留个习题：

```
a = True
```

```
b = not a # 不记得not请回顾 6.bool
```

想想下面这些逻辑运算的结果，然后用 print 输出看看你想的对不对：

```
b
```

```
not b
```

```
a == b
```


a != b

a and b

a or b

1<2 and b==True

- [← 10.变量2](#)
- [12.for循环 →](#)

【Python 第12课】for循环

大家对 while 循环已经有点熟悉了吧？今天我们来讲另一种循环语句：

for ... in ...

同 while 一样，for 循环可以用来重复做一件事情。在某些场景下，它比 while 更好用。

比如之前的一道习题：**输出1到100**（公众号中的练习题3）。

我们用while来做，需要有一个值来记录已经做了多少次，还需要在 while 后面判断是不是到了100。

如果用for循环，则可以这么写：

```
for i in range(1, 101):  
    print(i)
```

解释一下，**range(1, 101)** 表示从1开始，到101为止（不包括101，注意这里和 randint 不一样），取其中所有的整数。

for i in range(1, 101) 就是说，把这些数，依次赋值给变量i。

相当于一个一个循环过去，第一次 i=1，第二次 i=2，.....，直到 i=100。当 i=101 时跳出循环。

所以，当你需要一个循环10次的循环，你就只需要写：

```
for i in range(1, 11)
```

或者

```
for i in range(0, 10)
```

区别在于前者 i 是从1到10，后者 i 是从0到9。当然，你也可以不用 i 这个变量名，换成其他合法的变量名也可以。

比如一个循环n次的循环：

```
for count in range(0, n)
```

for 循环的本质是对一个序列中的元素进行遍历。什么是序列，以后再说。先记住这个最简单的形式：

```
for i in range(a, b)
```

就是从 a 循环至 b-1

现在，你可以用 for 循环来改写 [习题3~习题6](#)了。（小程序无法跳转的话可从公众号内菜单进入习题）

- [← 11.逻辑判断](#)
- [13.字符串 →](#)

【Python 第13课】字符串

字符串就是一组字符的序列（序列！又见序列！还记得我在说 for 循环的时候就提到过的序列吗？今天仍然不去细说它。），它一向是编程中的常见问题。之前我们用过它，以后我们还要不停地用它。

python中最常用的字符串表示方式是单引号（'）和双引号（"）。我还是要再说：一定得是英文标点！

'string' 和 "string" 对于 Python 来说效果是一样的。

可以直接输出一个字符串

```
print ('good')
```

也可以用一个变量来保存字符串，然后输出

```
str = "bad"
```

```
print(str)
```

如果你想表示一段带有英文单引号或者双引号的文字，那么表示这个字符串的引号就要与内容区别开。

内容带有单引号，就用双引号表示

```
"It's good"
```

反之亦然

```
'You are a "BAD" man'
```

还有一种在字符串中表示引号的方法，就是用 \，可以不受引号的限制

\ 表示单引号，\" 表示双引号

```
'I\'m a \"good\" teacher'
```

\ 被称作 **转译字符**，除了用来表示引号，还有比如用

- \n 表示字符串中的换行（相当于按一下回车键的效果）
- \t 表示字符串中的制表符（相当于按一下tab键的效果）
- \\ 表示字符串中的 \（因为单个斜杠被用来做转义了，所以真的要表示 \ 字符，就要两个斜杠）

\ 还有个用处，就是用来在代码中换行，而不影响输出的结果：

```
"this is the \
same line"
```

这个字符串仍然只有一行，和

```
"this is the same line"
```

是一样的，只是在代码中换了行。当你要写一行很长的代码时，这个会派上用场。

python中还有一种表示字符串的方法：

三个引号（'''）或者（"""）

在三个引号中，你可以方便地使用单引号和双引号，并且可以直接换行

```
'''
What's your name?" I asked.
I'm Han Meimei.
'''
```

#===== 课后作业 =====#

用print输出以下文字：

#1

He said, "I'm yours!"

#2

v//

#3

Stay hungry,

stay foolish.

-- Steve Jobs

#4

*

*

- [← 12.for循环](#)
- [14.字符串格式化 →](#)

【Python 第14课】字符串格式化

我们在输出字符串的时候，如果想对输出的内容进行一些整理，比如把几段字符拼接起来，或者把一段字符插入到另一段字符中间，就需要用到 **字符串的格式化输出**。

先从简单的开始，如果你想把两段字符连起来输出

```
str1 = 'good'
str2 = 'bye'
```

你可以

```
print(str1 + str2)
```

或者还可以把字符变量一个字符串相加

```
print('very' + str1)
print(str1 + ' and ' + str2)
```

但如果你想要把一个数字加到文字后面输出，比如这样

```
num = 18
print('My age is' + num)
```

程序就会报错。因为字符和数字不能直接用 + 相加。

一种解决方法是，用 `str()` 把数字转换成字符串

```
print('My age is' + str(18))
```

或

```
num = 18
print('My age is' + str(num))
```

还有一种方法，就是用 **%** 对字符串进行 **格式化**

```
num = 18
print('My age is %d' % num)
```

输出的时候，原始字符串中的 **%d** 会被 **%** 后面的值替换。输出

My age is 18

这里，**%d** 只能用来替换整数。如果你想格式化的数值是小数，要用 **%f**

```
print('Price is %f' % 4.99)
```

输出

Price is 4.990000

如果你想保留两位小数，需要在前面加上条件：**%.2f**

```
print('Price is %.2f' % 4.99)
```

输出

Price is 4.99

另外，可以用 **%s** 来替换一段字符串

```
name = 'Crossin'
print('%s is a good teacher.' % name)
```

输出

Crossin is a good teacher.

或者

```
print('Today is %s.' % 'Friday' )
```

输出

Today is Friday.

注意区分：有引号的表示一段字符，没有引号的就是一个变量，这个变量可能是字符，也可能是数字，但一定要和%所表示的格式相一致。

#===== 课后作业 =====#

现在，试试看用字符串格式化改进一下之前你写的小游戏。

比如你输入了一个数字72，程序不仅只是告诉你大了还是小了，而是会回答你

72 is too small.

或者

Bingo, 72 is the right answer!

- [← 13.字符串](#)
- [15.循环的嵌套 →](#)

【Python 第15课】循环的嵌套

设想一样，如果我们要输出5个*，用 for 循环可以这么写：

```
for i in range(0, 5):  
    print('*')
```

如果想让这5个*在同一行，需要加上 end 参数，使得 print 之后不换行：

```
for i in range(0, 5):  
    print('*', end=' ')
```

end 参数的作用是指定 print 结束之后的字符，默认是回车。你可以试试设置成不同字符的效果。

现在，如果我想要这样一个图形，怎么办？

```
* * * * *  
  
* * * * *  
  
* * * * *  
  
* * * * *  
  
* * * * *
```

当然，你可以循环5次，每次输出一行“*****”。那如果再进一步，这样呢？

```
*  
  
**  
  
***  
  
****  
  
*****
```

除了你自己动手打好一个多行字符串外，也可以让程序帮我们解决这种问题，我们需要的是两个嵌套在一起的循环：

```
for i in range(0, 3):  
    for j in range(0, 3):  
        print(i, j)
```

第二个 for 循环在第一个 for 循环的内部，表示每一次外层的循环中，都要进行一整遍内层的循环。

print 里面用逗号分割，可以连续输出多个不同的值。

看一下输出的结果：

```
0 0  
0 1  
0 2  
1 0  
1 1  
1 2  
2 0  
2 1  
2 2
```


内层循环中的 `print` 语句一共被执行了 9 次。

`i` 从 0 到 2 循环了 3 次。对应于每一个 `i` 的值，`j` 又做了从 0 到 2 三次循环。所以 3×3 一共 9 次。

所以如果要输出一个 5×5 的方阵图案，我们可以

```
for i in range(0, 5):  
    for j in range(0, 5):  
        print('*', end=' ')  
  
    print()
```

注意：第二个 `print` 的缩进和内层的 `for` 是一样的，这表明它是不是内层 `for` 循环中的语句，每次 `i` 的循环中，它只会在内循环执行完之后执行一次。

`print` 的括号里没有写任何东西，是起到换行的作用，这样，每输出 5 个*，就会换行。

要输出第二个三角图案时，我们需要根据当前外层循环的序数，设置内层循环应当执行的次数。

```
for i in range(0, 5):  
    for j in range(0, i+1):  
        print('*', end=' ')  
  
    print()
```

内层的 `j` 每次从 0 到 `i+1` 进行循环。

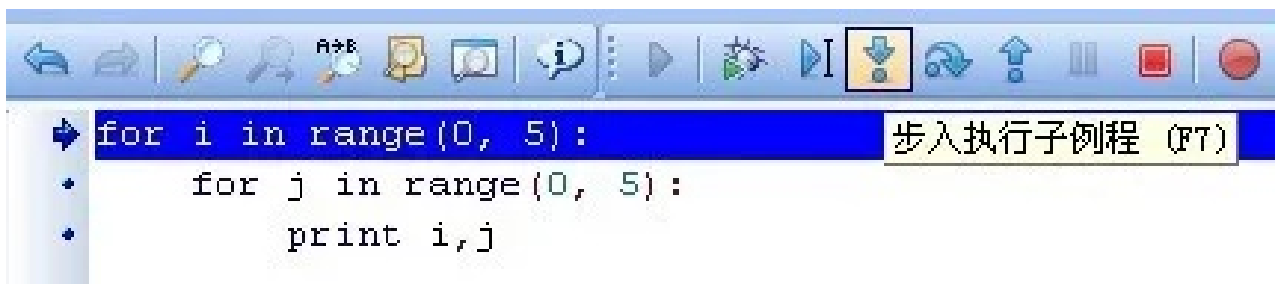
这样，当第一次 `i=0` 时，`j` 就是 `range(0,1)`，只输出 1 个*。

而当最后一次 `i=4` 时，`j` 就是 `range(0,5)`，输出 5 个*。

#===== 扩展阅读 =====#

最后顺便说下，如果有同学用的是 PyCharm，或者其他第三方 IDE，可以通过 debug 中的 `step` 按钮，查看程序是怎样一行一行运行的。IDLE 在这方面做得不太好，虽然也可以步进调试，但是很麻烦且不直观，所以就不推荐去用了。

想了解这种执行方式的，可以查看 [如何在 Python 中使用断点调试](#)



- [← 14.字符串格式化](#)
- [16.字符串格式化2 →](#)

【Python 第16课】字符串格式化2

#—— 继续字符串格式化 ——#

之前我们说到，可以用%来构造一个字符串，比如

```
print ('%s is easy to learn' % 'Python')
```

有时候，仅仅代入一个值不能满足我们构造字符串的需要。假设你现在有一组学生成绩的数据，你要输出这些数据。在一行中，既要输出学生的姓名，又要输出他的成绩。例如

Mike's score is 87.

Lily's score is 95.

在python中，你可以这样实现：

```
print("%s's score is %d" % ('Mike', 87))
```

或者

```
name = 'Lily'
```

```
score = 95
```

```
print("%s's score is %d" % (name, score))
```

无论你有多个值需要代入字符串中进行格式化，只需要在字符串中的合适位置用对应格式的%表示，然后在后面的括号中按顺序提供代入的值就可以了。占位的%和括号中的值在数量上必须相等，类型也要匹配。

('Mike', 87) 这种用()表示的一组数据在python中被称为元组（tuple），是python的一种基本数据结构，以后我们还会用到。

这里有点需要留心的就是，括号比较多，一定注意括号该加到哪里。外层print的括号，要包含整个输出内容的语句，包括前面的字符串和后面的变量。内层的括号包含代入的值。如果 print 的括号加的不对，加在了 % 前面，那只能输出前半部分，导致报错了。

- [← 15.循环的嵌套](#)
- [17.类型转换 →](#)

【Python 第17课】类型转换

#—— 类型转换 ——#

python的几种最基本的数据类型，我们已经见过：

- 字符串
- 整数
- 小数（浮点数）
- bool类型

python在定义一个变量时不需要给它限定类型。变量会根据赋给它的值，自动决定它的类型。你也可以在程序中，改变它的值，于是也就改变了它的类型。例如：

```
a = 1
print(a, type(a))

a = 'hello'
print(a, type(a))

a = True
print(a, type(a))
```

输出：

```
1 <class 'int'>
hello <class 'str'>
True <class 'bool'>
```

变量 a 先后成为了整数int、字符串str、bool类型。

虽然类型可以随意改变，但当你对一个特定类型的变量进行操作时，如果这个操作与它的数据类型不匹配，就会产生错误。比如以下几行代码

```
print('Hello'+1)
print('hello%d' % '123')
```

程序运行时会报错。因为第一句里，字符串和整数不能相加；第二句里，%d需要的是一个整数，而'123'是字符串。

这种情况下，python提供了一些方法对数值进行类型转换：

- int(x) #把x转换成整数
- float(x) #把x转换成浮点数
- str(x) #把x转换成字符串
- bool(x) #把x转换成bool值

上述两个例子就可以写成：

```
print ('Hello'+str(1))
print ('hello%d' % int('123'))
```

以下等式的结果均为真：

```
int('123') == 123
float('3.3') == 3.3
str(111) == '111'
bool(0) == False
```

并不是所有的值都能做类型转换，比如 int('abc') 同样会报错，因为 python 没办法把它转成一个整数。

另外关于bool类型的转换，我们会专门再详细说明。大家可以先试试以下结果的值，自己摸索一下转换成bool类型的规律：

```
bool(-123)
```

```
bool(0)
```

```
bool('abc')
```

```
bool('False')
```

```
bool('')
```

- [← 16.字符串格式化2](#)
- [18.bool类型转换 →](#)

【Python 第18课】bool类型转换

昨天最后留的几句关于bool类型的转换，其中有一行：

```
bool('False')
```

print 一下结果，会发现是 True。这是什么原因？

因为在python中，其他类型转成 bool 类型时，以下数值会被认为是False:

- 为0的数字，包括0, 0.0
- 空字符串，包括'', '''
- 表示空值的 None
- 空集合，包括(), [], {}

其他的值都认为是True。

None 是 python 中的一个特殊值，表示什么都没有，它和 0、空字符串、False、空集合 都不一样。关于集合，我们后面的课程再说。

所以，'False' 是一个包含5个字符的字符串，不是空字符串，当被转换成bool类型之后，就得到 True。

同样 bool(' ') 的结果是 True，一个空格也不能算作空字符串。

bool("") 才是False。

在 if、while 等条件判断语句里，判断条件会自动进行一次bool的转换。比如

```
a = '123'

if a:

    print ('this is not a blank string')
```

这在编程中是很常见的一种写法。效果等同于

```
if bool(a) == True:
```

或者

```
if a != '':
```

- [← 17.类型转换](#)
- [19.函数 →](#)

【Python 第19课】函数

数学上的函数，是指给定一个输入，就会有唯一输出的一种对应关系。编程语言里的函数跟这个意思差不多，但也有不同。

编程中所说的函数，就是一堆语句组成的语句块，这个语句块有个名字，你可以在需要时反复地使用这块语句。它有可能需要输入，有可能会返回输出。

举一个现实中的场景：我们去餐厅吃饭，跟服务员点了菜，过了一会儿，服务员把做好的菜端上来。

1. 餐厅的厨房就可以看作是一个函数
2. 我们点的菜单，就是给这个函数的参数（对函数来说就是输入）
3. 厨师在厨房里做菜的过程就是这个函数的执行过程
4. 做好的菜是返回结果，返回到我们的餐桌上（函数的返回值）

我们之前已经用到过 python 里内建的函数，比如 `input` 和 `range`。

以 `range(1,10)` 为例，`range` 是这个函数的名称，后面括号里的1和10是 `range` 需要的参数。它有返回结果，就是一个从1到9的序列生成器（暂时你可以理解为，就是1~9九个数字）。

再来看 `input()`，括号里面什么都没有，表示我们没有给参数。函数执行过程中，需要我们从控制台输入一个值。函数的返回结果就是我们输入的内容。

PS: `range` 还可以接受1个或3个参数，`input`也可以接受1个字符串参数。可以等我之后几课来讲，或者尝试在网上搜索下“*python range 参数*”。

如果我们要自己写一个函数，就需要去定义它。python里的关键字叫 `def`（define的缩写），格式如下：

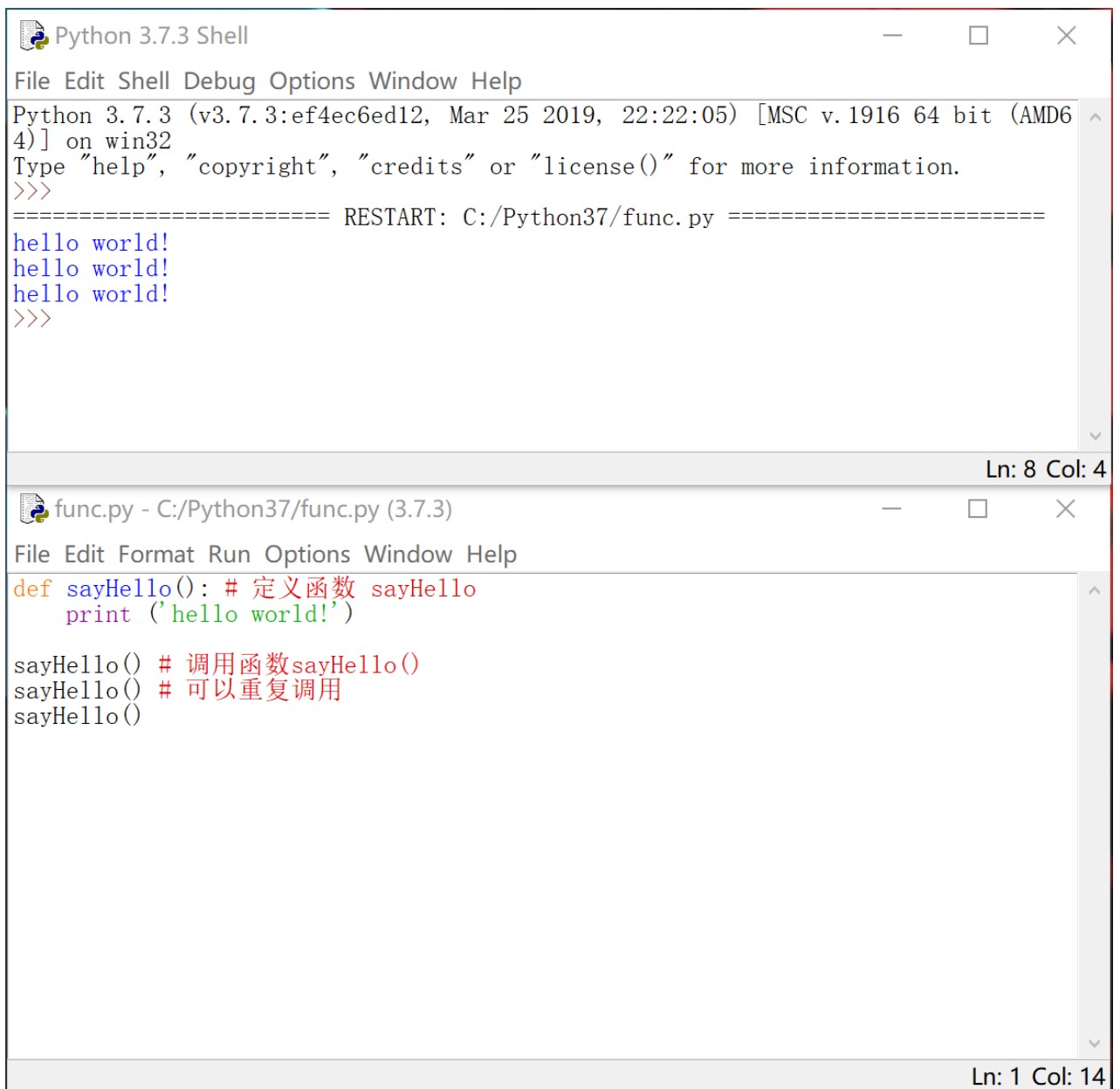
```
def sayHello():  
    print ('hello world!')
```

`sayHello` 是这个函数的名字，后面的括号里是参数，这里没有，表示不需要参数。但括号和后面的冒号都不能少。下面缩进的代码块就是整个函数的内容，称作函数体。

然后我们去调用这个函数，就是用函数名加上括号，有必要的话，括号里放参数：

```
sayHello()
```

得到和直接执行`print ('hello world!')`一样的结果。



Python 3.7.3 Shell

File Edit Shell Debug Options Window Help

Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Python37/func.py =====
hello world!
hello world!
hello world!
>>>

Ln: 8 Col: 4

func.py - C:/Python37/func.py (3.7.3)

File Edit Format Run Options Window Help

```
def sayHello(): # 定义函数 sayHello
    print ('hello world!')

sayHello() # 调用函数sayHello()
sayHello() # 可以重复调用
sayHello()
```

Ln: 1 Col: 14

- [← 18.bool类型转换](#)
- [20.命令行常用命令→](#)

【Python 第20课】命令行常用命令

今天茬开话题，说一下命令行（Windows下叫“命令提示符”，Mac下叫“终端”）里的常用命令。已经熟悉同学可略过。

打开命令行，我们会看到每行前面都有诸如

```
C:\Documents and Settings\Crossin>
```

或者

```
MyMacBook:~ crossin$
```

之类的。

这个提示符表示了当前命令行所在目录。

在这里，我们输入 `python` 并敲下回车，就可以进入python环境了。但今天我们暂时不这么做。

第一个常用的命令是：

dir（windows环境下）

ls（mac环境下）

`dir` 和 `ls` 的作用差不多，都是显示出当前目录下的文件和文件夹。

具体效果可参见文末的附图。

第二个常用命令是：

`cd 目录名`

通过 `dir` 或 `ls` 了解当前目录的结构之后，可以通过“`cd 目录名`”的方式，进入到当前目录下的子目录里。

如果要跳回到上级目录，可以用命令：

```
cd ..
```

另外，Windows下如果要写换盘符，需要输入

盘符：

比如从c盘切换到d盘

```
C:\Documents and Settings\Crossin> d:
```

有了以上两个命令，就可以在文件目录的迷宫里游荡了。虽然没可视化的目录下的操作那么直观，但是会显得你更像个程序员。。。

于是乎，再说个高阶玩法：现在你可以不用 `idle` 那套东西了，随便找个顺手的文本软件，把你的代码写好，保存好，最好是保存成 `py` 文件。

然后在命令行下，通过使用 `cd` 命令进入到 `py` 文件保存的目录，再执行命令：

```
python 代码的文件名
```

就可以运行你写的程序了。

演示如下：


```
命令提示符
C:\Users>cd ..
C:\>dir
驱动器 C 中的卷是 Local Disk
卷的序列号是 6ADA-15B4

C:\ 的目录

2019/02/22  16:22    <DIR>          Intel
2019/06/05  15:19    <DIR>          Program Files
2019/06/11  09:51    <DIR>          Program Files (x86)
2019/06/11  13:09    <DIR>          Python37
2019/04/08  23:29    <DIR>          QMDownload
2019/06/11  13:20    <DIR>          qxdata
2019/06/11  10:38    <DIR>          Users
2019/06/11  11:53    <DIR>          Windows
                0 个文件                0 字节
                8 个目录 148,565,544,960 可用字节

C:\>cd python37
C:\Python37>python lesson3.py
Hello
IDE
here I am.
C:\Python37>_
```

嗯，这才像个python程序员的样！

注意：如果执行 python 命令时提示“不是内部或外部命令”，那么说明你的环境变量没有设置好。请参考 [1.安装](#) 课中最后一点 **配置命令行** 里的几点注意进行检查。

其他常用命令，诸如 **拷贝文件**、**删除文件**、**新建文件夹** 之类的，请你自行在网上搜索相关资料。很容易的，比如你搜“mac 终端 常用命令”，就可以找到很多了。

#===== 扩展阅读 =====#

使用搜索引擎也是程序员的重要技能之一，所以现在你可以练习起来，推荐阅读下 [编程初学者如何使用搜索引擎](#)。

学到这里，很多同学都已经写出来可以完整运行的代码，有了不小的成就感，希望能把程序发给别人运行。然而对方电脑上如果没装python就运行不了了。

这里，我提供两篇关于如果把你的 python 程序打包成别人电脑上也可执行的 exe 文件的教程，稍稍有点复杂，想挑战的可以去试试：

- [利用cx_Freeze将py文件打包成exe文件（图文全解）](#)
- [将打飞机游戏打包成 exe](#)
- [← 19.函数](#)
- [21.函数的参数 →](#)

【Python 第21课】函数的参数

在19课里，我们讲了怎样定义一个自己的函数，但我们没有给他提供输入 **参数** 的功能。不能指定参数的函数就好比你去餐厅吃饭，服务员告诉你，不能点菜，有啥吃啥。这显然不能满足很多情况。

所以，如果我们希望自己定义的函数里允许调用者提供一些参数，就把这些参数写在括号里，如果有多个参数，用逗号隔开，如：

```
def sayHello(someone):
    print(someone + ' says Hello!')
```

或者

```
def plus(num1, num2):
    print(num1+num2)
```

参数在函数中相当于一个变量，而这个变量的值是在调用函数的时候被赋予的。在函数内部，你可以像过去使用变量一样使用它。

调用带参数的函数时，同样把需要传入的参数值放在括号中，用逗号隔开。要注意提供的参数值的数量和类型需要跟函数定义中的一致。如果这个函数不是你自己写的，你需要先了解它的参数类型，才能顺利调用它。

比如上面两个函数，我们可以直接传入值：

```
sayHello('Crossin')
```

还是注意，字符串类型的值不能少了引号。

或者也可以传入变量：

```
x = 3
y = 4
plus(x, y)
```

在这个函数被调用时，相当于做了num1=x, num2=y这么一件事。所以结果是输出了7。

当函数结束后，这里的 num1、num2 就不再可以使用了，它们只存在于函数内部。关于这一点，后面我们会专门来细说。

#===== 课外的话 =====#

今天发现了一个iPad上的游戏，叫Cargo-Bot。这个游戏需要你指令控制一个机械臂去搬箱子。游戏里蕴含了很多编程的思想，包括循环、函数调用、条件判断、寄存器、递归等等，挺有意思的。更厉害的是，这个游戏是用一个叫Codea的app直接在iPad上编写出来的。有iPad的同学不妨玩玩看，挑战一下你的“程商”。

有些家长想让小孩子学编程，其实我觉得，比较小的孩子，玩一玩这种“编程思维”的游戏就足够了。

- [← 20.命令行常用命令](#)
- [22.函数应用示例 →](#)

【Python 第22课】函数应用示例

前两课稍稍介绍了一下函数，但光说概念还是有些抽象了，今天就来把之前那个小游戏用函数改写一下。

我希望有这样一个函数，它比较两个数的大小。

如果第一个数小了，就输出“too small”

如果第一个数大了，就输出“too big”

如果相等，就输出“bingo!”（意为：猜中了！）

函数还有个 **返回值**，当两数相等的时候返回True，不等就返回False。

于是我们来定义这个函数：

```
def isEqual(num1, num2):  
    if num1<num2:  
        print ('too small')  
        return False  
    if num1>num2:  
        print ('too big')  
        return False  
    if num1==num2:  
        print ('bingo!')  
        return True
```

这里说一下，**return** 是函数的结束语句，return后面的值被作为这个函数的 **返回值**。函数中任何地方的 return 被执行到的时候，这个函数就会立刻结束并跳出。

注意：函数的 **返回值** 和我们前面说的 **输出** 是两回事。print 输出是将结果显示在控制台中，最终一定是转成字符类型；而 **返回值**，是将结果返回到调用函数的地方，可以是任何类型。

然后在我们的游戏里使用这个函数：

```
from random import randint  
num = randint(1, 100)  
print('Guess what I think?')  
bingo = False  
while bingo == False:  
    answer = int(input())  
    bingo = isEqual(answer, num)
```

在 isEqual 函数内部，会输出 answer 和 num 的比较结果，如果相等的话，bingo 会得到返回值 True，否则 bingo 得到 False，循环继续。

```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
>>>
===== RESTART: C:/Python37/funcgame.py =====
Guess what I think?
50
too big
25
too big
12
too small
18
too small
22
too small
23
bingo
>>> |
Ln: 32 Col: 4

funcgame.py - C:/Python37/funcgame.py (3.7.3)
File Edit Format Run Options Window Help
def isEqual(num1, num2):
    if num1<num2:
        print ('too small')
        return False
    if num1>num2:
        print ('too big')
        return False
    if num1==num2:
        print ('bingo')
        return True

from random import randint
num = randint(1, 100)
print ('Guess what I think?')
bingo = False
while bingo == False:
    answer = int(input())
    bingo = isEqual(answer, num)
Ln: 18 Col: 31
```

函数可以把某个功能的代码分离出来，在需要的时候重复使用，就像拼装积木一样，这会让程序结构更清晰。

- [← 21.函数的参数](#)
- [23.if, elif, else →](#)

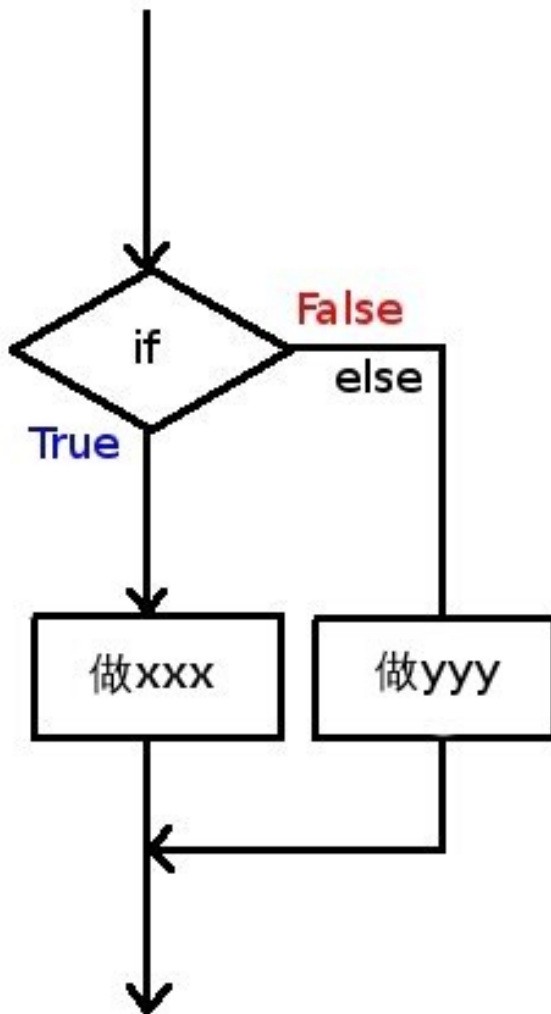
【Python 第23课】if, elif, else

今天补充之前讲过的一个语句：**if**。为什么我跳要着讲，因为我的想法是先讲下最基本的概念，让你能用起来，之后你熟悉了，再说些细节。

关于 **if**，可以回顾 [7.if](#)。它除了我们之前讲的用法外，还可以配合 **elif** 和 **else** 使用，使程序的运行顺序更灵活。

之前说的 **if** 是：“如果”条件满足，就做xxx，否则就不做。

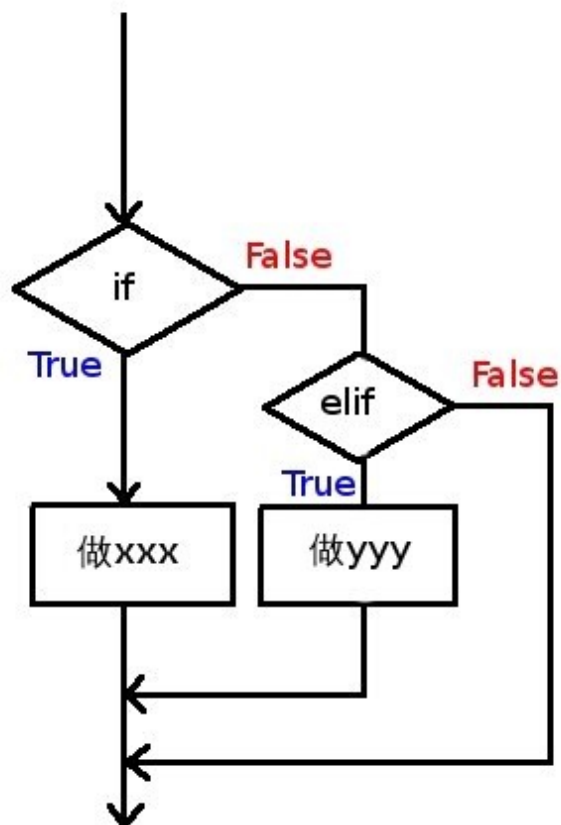
else 顾名思义，就是：“否则”就做yyy。



当**if**后面的条件语句不满足时，与之相对应的 **else** 中的代码块将被执行。

```
if a == 1:
    print('right')
else:
    print('wrong')
```

elif 意为 **else if**，含义就是：“否则如果”条件满足，就做yyy。**elif** 后面需要有一个逻辑判断语句。



当if条件不满足时，再去判断 elif的条件，如果满足则执行其中的代码块：

```
if a == 1:
    print ('one')
elif a == 2:
    print ('two')
```

if, elif, else 可组成一个整体的条件语句。

1. if是 必须有 的；
2. elif可以没有，也可以有很多个，每个elif条件不满足时会进入下一个elif判断；一旦满足，执行完就结束整个条件语句；
3. else可以没有，如果有的话只能有一个，必须在条件语句的最后。

```
if a == 1:
    print ('one')
elif a == 2:
    print ('two')
elif a == 3:
    print ('three')
else:
    print ('too many')
```

我们昨天刚改写的小游戏中的函数 isEqual，用了三个条件判断，我们可以再改写成一个包含 if...elif...else 的结构：

```
def isEqual(num1, num2):
    if num1 < num2:
        print ('too small')
```

```
        return False;

    elif num1>num2:

        print ('too big')

        return False;

    else:

        print ('bingo')

        return True
```



```
def isEqual(num1, num2):
    if num1<num2:
        print (' too small')
        return False
    elif num1>num2:
        print (' too big')
        return False
    else:
        print (' bingo')
        return True

from random import randint
num = randint(1, 100)
print ('Guess what I think?')
bingo = False
while bingo == False:
    answer = int(input())
    bingo = isEqual(answer, num)
```

Ln: 6 Col: 0

运行效果和之前是一样的，不过代码逻辑更合理一点。

- [← 22.函数应用示例](#)
- [24.if的嵌套 →](#)

【Python 第24课】if的嵌套

和 for 循环一样，if也可以嵌套使用，即在一个 if/elif/else 的内部，再使用 if。这有点类似于电路的串联。

```
if 条件1:
    if 条件2:
        语句1
    else:
        语句2
else:
    if 条件2:
        语句3
    else:
        语句4
```

在上面这个两层if的结构中，当：

条件1为True，条件2为True时，

执行语句1；

条件1为True，条件2为False时，

执行语句2；

条件1为False，条件2为True时，

执行语句3；

条件1为False，条件2为False时，

执行语句4。

假设需要这样一个程序：

我们先向程序输入一个值x，再输入一个值y。(x,y)表示一个点的坐标。程序要告诉我们这个点处在坐标系的哪一个象限。

$x \geq 0, y \geq 0$ ，则输出1；

$x < 0, y \geq 0$ ，则输出2；

$x < 0, y < 0$ ，则输出3；

$x \geq 0, y < 0$ ，则输出4。

你可以分别写4个if，也可以用if的嵌套：

```
if y >= 0:
    if x >= 0:
        print (1)
    else:
        print (2)
else:
    if x < 0:
```

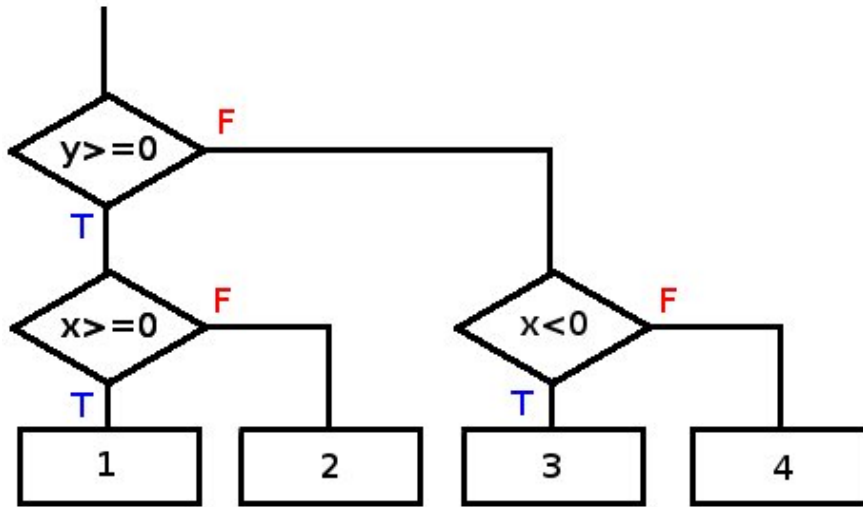


```
print (3)

else:

    print (4)
```

从流程图上来看，应该是这样。



是否从图中看出了两层 if 结构呢？

- [← 23.if, elif, else](#)
- [25.初探list →](#)

【Python 第25课】初探list

今天要说一个新概念：list，中文可以翻译成列表，是用来处理一组有序项目的数据结构。想象一下你的购物清单、待办工作、手机通讯录等等，它们都可以看作是一个列表。说它是新概念也不算确切，因为我们之前已经用过它，就在这个语句里：

```
for i in range(1, 10):

    print(i)

    #此处略过数行代码
```

看出来list在哪里了吗？你试一下：

```
print(list(range(1,10)))
```

得到的结果是：

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

这样一个中括号中的序列就是 list（列表）。虽然在py3，range的结果并不完全等同于list（这个我们以后再讨论），需要额外转换一下，但你暂时可把它当做list来理解。

把上面那个for循环语句写成：

```
l = range(1, 10)

for i in l:

    print(i)

    #此处略过数行代码
```

效果是一样的。

于是可以看出，for循环做的事情其实就是遍历一个列表中的每一项，每次循环都把当前项赋值给一个变量（这里是i），直到列表结束。

我们也可以定义自己的列表，格式就是用 **中括号** 包围、**逗号** 隔开的一组数值：

```
l = [1, 1, 2, 3, 5, 8, 13]
```

可以用print输出这个列表：

```
print(l)
```

同样也可以用for...in遍历这个列表，依次输出了列表中的每一项：

```
for i in l:

    print(i)
```

列表中的元素也可以是别的类型，比如：

```
l = ['meat', 'egg', 'fish', 'milk']
```

甚至是不同类型的混合：

```
l = [365, 'everyday', 0.618, True]
```

l身为一个列表，有一些特有的功能，这个我们下回再说。

注意：我们这里的例子里用的变量名是小写字母l，不是大写字母I。你也可以用其他的名字命名变量，但请不要用list本身，因为它已经用来表示列表类型，如果再给它赋值，会导致原本的功能被覆盖，很可能带来问题。

#—— 点球小游戏 ——#

待会儿要去现场看场中超比赛。于是想到，以我们现在所学的，可以做一个罚点球的游戏。大概流程是这样：

每一轮，你先输入一个方向射门，然后电脑随机判断一个方向扑救。方向不同则算进球得分，方向相同算扑

救成功，不得分。

之后攻守轮换，你选择一个方向扑救，电脑随机方向射门。

第5轮结束之后，如果得分不同，比赛结束。

5轮之内，如果一方即使踢进剩下所有球，也无法达到另一方当前得分，比赛结束。

5论之后平分，比赛继续进行，直到某一轮分出胜负。

你先想想怎么写，能不能自己搞定。过两天我再来说我的做法。

- [← 24.iif的嵌套](#)
- [26.操作list →](#)

【Python 第26课】操作list

上周给list开了个头，知道了什么是list。假设我们现在有一个list:

```
l = [365, 'everyday', 0.618, True]
```

除了用 for...in 遍历 l 中的元素，我们还能做点啥？

1. 访问 list 中的元素

list 中的每个元素都对应一个递增的序号。与现实中习惯的序号不同在于，计算机中的计数通常都是 **从0开始**，python也不例外。（如果你记不清这个而导致了错误，请去听一下孙燕姿的《爱从零开始》:）

要访问 l 中的第 1 个元素 365，只要用 l[0] 就可以了。依次类推，

```
print (l[1])
```

就会输出 'everyday'

注意，你不能访问一个不存在的元素，比如 l[10]，程序就会报错，提示你 index 越界了。

2. 修改list中的元素

修改list中的某一个元素，只需要直接给那个元素赋值就可以了：

```
l[0] = 123
```

输出 l，得到 [123, 'everyday', 0.618, True]，第1个元素已经从365被改成了123。

3. 向list中添加元素

list 有一个 **append** 方法，可以增加元素。以这个列表为例，调用的方法是：

```
l.append(1024)
```

输出 l，你会看到 [123, 'everyday', 0.618, True, 1024]，1024被添加到了 l，成为最后一个元素。（第一个元素在上一步被改成了123）

然后同样可以用 l[4] 得到 1024。

4. 删除list中的元素

删除list中的某一个元素，要用到 del:

```
del l[0]
```

输出 l，得到 ['everyday', 0.618, True, 1024]。这时候再调用 l[0]，会得到 'everyday'，其他元素的序号也相应提前。

以上这些命令，你可以直接在 python shell 中尝试。

```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
>>>
>>> 1 = [365, 'everyday', 0.618, True]
>>> 1
[365, 'everyday', 0.618, True]
>>> 1[0]
365
>>> 1[2]
0.618
>>> 1[3]
True
>>> 1[0] = 123
>>> 1
[123, 'everyday', 0.618, True]
>>> 1.append(1024)
>>> 1
[123, 'everyday', 0.618, True, 1024]
>>> del 1[0]
>>> 1
['everyday', 0.618, True, 1024]
>>> 1[0]
'everyday'
>>>
Ln: 85 Col: 8
```

#——点球小游戏——#

我打算从今天开始，每天说一点这个小游戏的做法。方法有很多种，我只是提供一种参考。你可以按照自己喜欢的方式去做，那样她才是属于你的游戏。

先说一下方向的设定。我的想法比较简单，就是左、中、右三个方向，用字符串来表示。射门或者扑救的时候，直接输入方向。所以这里我准备用input。有同学是用1-9的数字来表示八个方向和原地不动，每次输入一个数字，这也是可以的。不过这样守门员要扑住的概率可就小多了。

至于电脑随机挑选方向，如果你是用数字表示，就用我们之前讲过的randint来随机就行。不过我这次打算用random的另一个方法：**choice**。它的作用是从一个list中随机挑选一个元素。

于是，罚球的过程可以这样写：

```
from random import choice

print ('Choose one side to shoot:')

print ('left, center, right')

you = input()

print ('You kicked ' + you)

direction = ['left', 'center', 'right']

com = choice(direction)

print ('Computer saved ' + com)

if you != com:

    print ('Goal!')

else:

    print ('Oops...')
```

反之亦然，不赘述。

- [← 25.初探list](#)
- [27.list切片 →](#)

【Python 第27课】list切片

list有两类常用操作：**索引(index)**和**切片(slice)**。

昨天我们说的用[]加序号访问的方法就是索引操作。

除了指定位置进行索引外，list还可以处理负数的索引。继续用昨天的例子：

```
l = [365, 'everyday', 0.618, True]
```

`l[-1]`表示中的最后一个元素。

`l[-3]`表示倒数第3个元素。

切片操作符 是在[]内提供一对可选数字，用:分割。冒号前的数表示切片的开始位置，冒号后的数字表示切片到哪里结束。同样，计数从0开始。

注意：开始位置包含在切片中，而结束位置不包括。

```
l[1:3]
```

得到的结果是['everyday', 0.618]。

如果不指定第一个数，切片就从列表第一个元素开始。

如果不指定第二个数，就一直到最后元素结束。

都不指定，则返回整个列表。

```
l[:3]
```

```
l[1:]
```

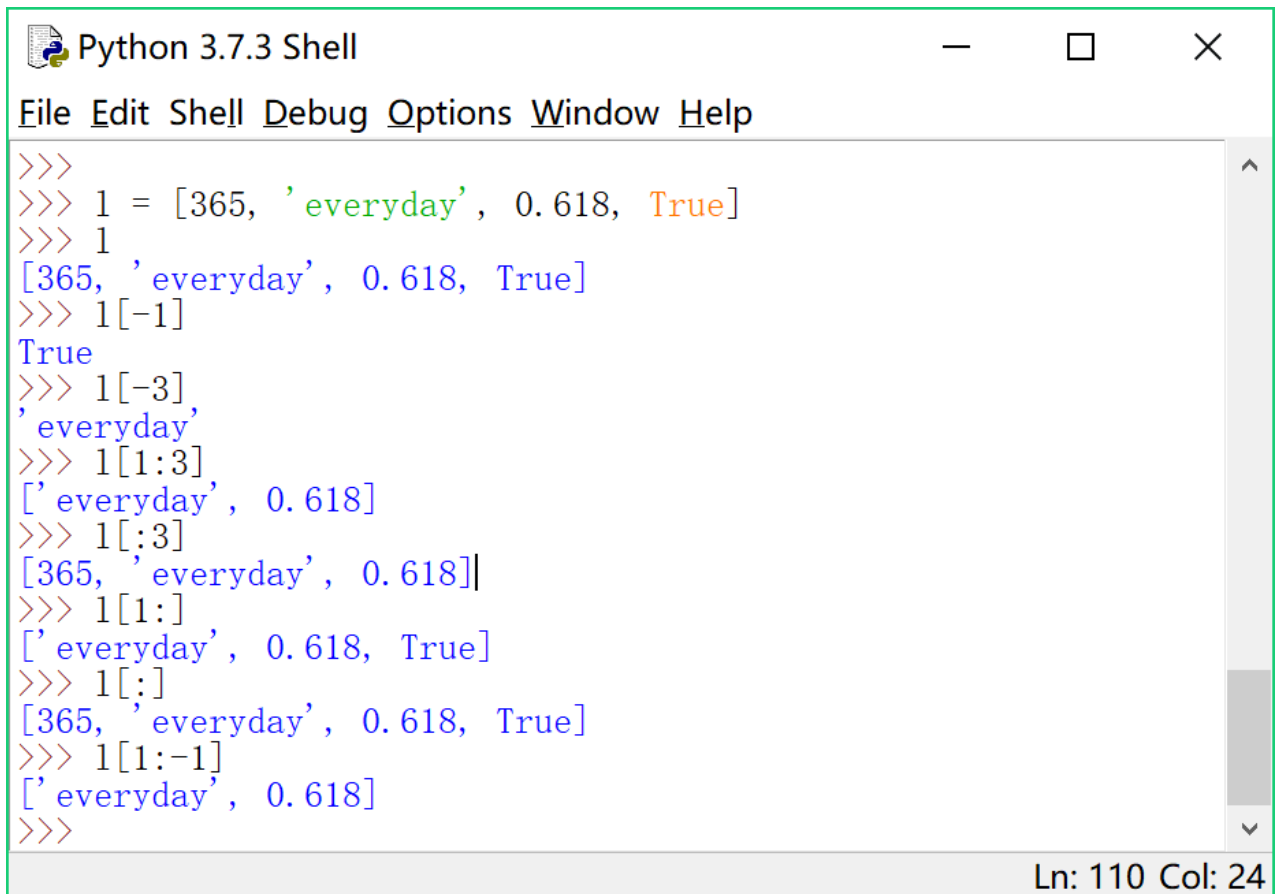
```
l[:]
```

注意：不管是返回一部分，还是整个列表，都是一个新的对象，与不影响原来的列表。

同索引一样，切片中的数字也可以使用负数。比如：

```
l[1:-1]
```

得到['everyday', 0.618]

A screenshot of a Python 3.7.3 Shell window. The window has a title bar with the Python logo and text 'Python 3.7.3 Shell', and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main area is a text editor showing a series of Python commands and their outputs. The commands demonstrate list indexing and slicing on a list 'l' containing [365, 'everyday', 0.618, True]. The status bar at the bottom right shows 'Ln: 110 Col: 24'.

```
>>>
>>> l = [365, 'everyday', 0.618, True]
>>> l
[365, 'everyday', 0.618, True]
>>> l[-1]
True
>>> l[-3]
'everyday'
>>> l[1:3]
['everyday', 0.618]
>>> l[:3]
[365, 'everyday', 0.618]
>>> l[1:]
['everyday', 0.618, True]
>>> l[:]
[365, 'everyday', 0.618, True]
>>> l[1:-1]
['everyday', 0.618]
>>>
```

#—— 点球小游戏 ——#

昨天有了一次罚球的过程，今天我就让它循环5次，并且记录下得分。先不判断胜负。

用score_you表示你的得分，score_com表示电脑得分。开始都为0，每进一球就加1。

```
from random import choice

score_you = 0
score_com = 0
direction = ['left', 'center', 'right']

for i in range(5):
    print ('==== Round %d - You Kick! ==== ' % (i+1))
    print ('Choose one side to shoot:')
    print ('left, center, right')
    you = input()
    print ('You kicked ' + you)
    com = choice(direction)
    print ('Computer saved ' + com)
    if you != com:
        print ('Goal!')
        score_you += 1
    else:
        print ('Oops...')
    print ('Score: %d(you) - %d(com)\n' % (score_you, score_com))
```



```
print ('==== Round %d - You Save! ==== ' % (i+1))

print ('Choose one side to save:')

print ('left, center, right')

you = input()

print ('You saved ' + you)

com = choice(direction)

print ('Computer kicked ' + com)

if you == com:

    print ('Saved!')

else:

    print ('Oops...')

    score_com += 1

print ('Score: %d(you) - %d(com)\n' % (score_you, score_com))
```

这段代码里有两段相似度很高，想想是不是可以有办法可以用个函数把它们分离出来。

```
soccer1.py - C:/Python37/soccer1.py (3.7.3)
File Edit Format Run Options Window Help

from random import choice

score_you = 0
score_com = 0
direction = ['left', 'center', 'right']

for i in range(5):
    print('==== Round %d - You Kick! ==== ' % (i + 1))
    print('Choose one side to shoot:')
    print('left, center, right')
    you = input() # 输入射门方向
    print('You kicked ' + you)
    com = choice(direction) # 电脑随机扑救方向
    print('Computer saved ' + com)
    if you != com: # 方向不同, 球进
        print('Goal!')
        score_you += 1 # 玩家得分
    else:
        print('Oops...')
    print('Score: %d(you) - %d(com)\n' % (score_you, score_com))

    print('==== Round %d - You Save! ==== ' % (i + 1))
    print('Choose one side to save:')
    print('left, center, right')
    you = input() # 输入扑救方向
    print('You saved ' + you)
    com = choice(direction) # 电脑随机射门方向
    print('Computer kicked ' + com)
    if you == com: # 方向相同, 球被扑出
        print('Saved!')
    else:
        print('Oops...')
        score_com += 1 # 电脑得分
    print('Score: %d(you) - %d(com)\n' % (score_you, score_com))
```

Ln: 25 Col: 26

- [← 26.操作list](#)
- [28.字符串的分割 →](#)

【Python 第28课】字符串的分割

字符串和list之间有很多不得不说的事情。比如有同学想要用python去自动抓取某个网页上的下载链接，那就需要对网页的代码进行处理。处理的过程中，免不了要在字符串和list之间进行很多操作。

我们先从最基本的开始。假设你现在拿到了一个英语句子，需要把这个句子中的每一个单词拿出来单独处理。

```
sentence = 'I am an English sentence'
```

这时就需要对字符串进行分割。

```
sentence.split()
```

split() 会把字符串按照其中的空格进行分割，分割后的每一段都是一个新的字符串，最终返回这些字符串组成一个list。于是得到

```
['I', 'am', 'an', 'English', 'sentence']
```

原来字符串中的空格不再存在。

除了空格外，**split()**同时也会按照换行符 **\n**，制表符 **\t** 进行分割。所以应该说，**split**默认是按照 **空白字符** 进行分割。

之所以说默认，是因为**split**还可以指定分割的符号。比如你有一个很长的字符串

```
section = 'Hi. I am the one. Bye.'
```

通过指定分割符号为**'.'**，可以把每句话分开

```
section.split('.')
```

得到

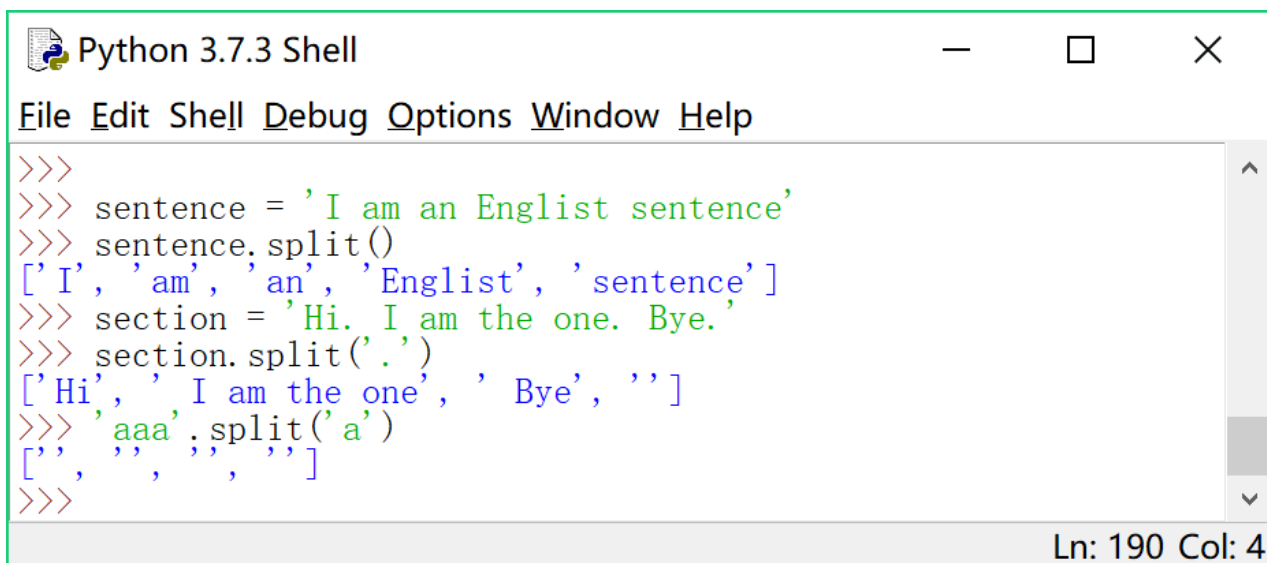
```
['Hi', ' I am the one', ' Bye', '']
```

这时候，**'.'**作为分割符被去掉了，而空格仍然保留在它的位置上。

注意最后那个空字符串。每个**'.'**都会被作为分割符，即使它的后面没有其他字符，也会有一个空串被分割出来。例如

```
'aaa'.split('a')
```

将会得到**['', '', '', '']**，由四个空串组成的list。



```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
>>>
>>> sentence = 'I am an English sentence'
>>> sentence.split()
['I', 'am', 'an', 'English', 'sentence']
>>> section = 'Hi. I am the one. Bye.'
>>> section.split('.')
['Hi', ' I am the one', ' Bye', '']
>>> 'aaa'.split('a')
['', '', '', '']
>>>
```

Ln: 190 Col: 4

既然有把字符串分割成list，那也相应就有把list连接成字符串，这个明天说。

#—— 点球小游戏 ——#

在昨天代码的基础上，我们加上胜负判断，如果5轮结束之后是平分，就继续踢。

所以我们把一轮的过程单独拿出来作为一个函数kick，在5次循环之后再加上一个while循环。

另外，这里把之前的score_you和score_com合并成了一个score数组。这里的原因是，要让kick函数里用到外部定义的变量，需要使用全局变量的概念。暂时想避免说这个，而用list不存在这个问题。（但希望你不要觉得在函数里使用外面定义的变量很自然，后面我们会具体分析）

```
from random import choice

score = [0, 0]

direction = ['left', 'center', 'right']

def kick():

    print('==== You Kick! ====')

    print('Choose one side to shoot:')

    print('left, center, right')

    you = input()

    print('You kicked ' + you)

    com = choice(direction)

    print('Computer saved ' + com)

    if you != com:

        print('Goal!')

        score[0] += 1

    else:

        print('Oops...')

    print('Score: %d(you) - %d(com)\n' % (score[0], score[1]))

    print('==== You Save! ====')

    print('Choose one side to save:')

    print('left, center, right')

    you = input()

    print('You saved ' + you)

    com = choice(direction)

    print('Computer kicked ' + com)

    if you == com:

        print('Saved!')

    else:

        print('Oops...')

        score[1] += 1

    print('Score: %d(you) - %d(com)\n' % (score[0], score[1]))

for i in range(5):

    print('==== Round %d ==== ' % (i + 1))
```

```
kick()
```

```
while (score[0] == score[1]):  
    i += 1  
    print('==== Round %d ==== ' % (i + 1))  
    kick()
```

```
if score[0] > score[1]:  
    print('You Win!')  
else:  
    print('You Lose.')
```

```
soccer2.py - C:/Python37/soccer2.py (3.7.3)
File Edit Format Run Options Window Help

from random import choice

score = [0, 0]
direction = ['left', 'center', 'right']

def kick():
    print('==== You Kick! ====')
    print('Choose one side to shoot:')
    print('left, center, right')
    you = input()
    print('You kicked ' + you)
    com = choice(direction)
    print('Computer saved ' + com)
    if you != com:
        print('Goal!')
        score[0] += 1
    else:
        print('Oops...')
    print('Score: %d(you) - %d(com)\n' % (score[0], score[1]))

    print('==== You Save! ====')
    print('Choose one side to save:')
    print('left, center, right')
    you = input()
    print('You saved ' + you)
    com = choice(direction)
    print('Computer kicked ' + com)
    if you == com:
        print('Saved!')
    else:
        print('Oops...')
        score[1] += 1
    print('Score: %d(you) - %d(com)\n' % (score[0], score[1]))

for i in range(5):
    print('==== Round %d ==== ' % (i + 1))
    kick()

while (score[0] == score[1]):
    i += 1
    print('==== Round %d ==== ' % (i + 1))
    kick()

if score[0] > score[1]:
    print('You Win!')
else:
    print('You Lose.')
|
```

Ln: 48 Col: 0

- [← 27.list切片](#)
- [29.连接list →](#)

【Python 第29课】连接list

今天要说的方法是join。它和昨天说的split正好相反：split是把一个字符串分割成很多字符串组成的list，而join则是把一个list中的所有字符串连接成一个字符串。

join的格式有些奇怪，它不是list的方法，而是字符串的方法。首先你需要有一个字符串作为list中所有元素的连接符，然后再调用这个连接符的join方法，join的参数是被连接的list：

```
s = ';'
li = ['apple', 'pear', 'orange']
fruit = s.join(li)
print (fruit)
```

得到结果 'apple;pear;orange'。

从结果可以看到，分号把list中的几个字符串都连接了起来。

你也可以直接在python shell中输入：

```
';'.join(['apple', 'pear', 'orange'])
```

得到同样的结果。

用来连接的字符串可以是多个字符，也可以是一个空串：

```
''.join(['hello', 'world'])
```

得到'helloworld'，字符串被无缝连接在一起。

点球小游戏

昨天的代码已经能实现一个完整的点球比赛过程，但有同学提出：这不符合真实比赛规则，说好的提前结束比赛呢？！

关于这个，我想了下，可以有好几种解决方法，但似乎都有些绕。所以放到明天单独来讲，把这个小游戏收尾。

- [← 28.字符串的分割](#)
- [30.字符串的索引和切片 →](#)

【Python 第30课】字符串的索引和切片

之前说了，字符串和list有很多不得不说的事情。今天就来说说字符串的一些与list相似的操作。

1. 遍历

通过for...in可以遍历字符串中的每一个字符。

```
word = 'helloworld'

for c in word:

    print(c)
```

2. 索引访问

通过[]加索引的方式，访问字符串中的某个字符。

```
print (word[0])

print (word[-2])
```

与list不同的是，字符串 **不能** 通过索引访问去更改其中的字符。

```
word[1] = 'a'
```

这样的赋值是 **错误** 的！

3. 切片

通过两个参数，截取一段子串，具体规则和list相同。

```
print (word[5:7])

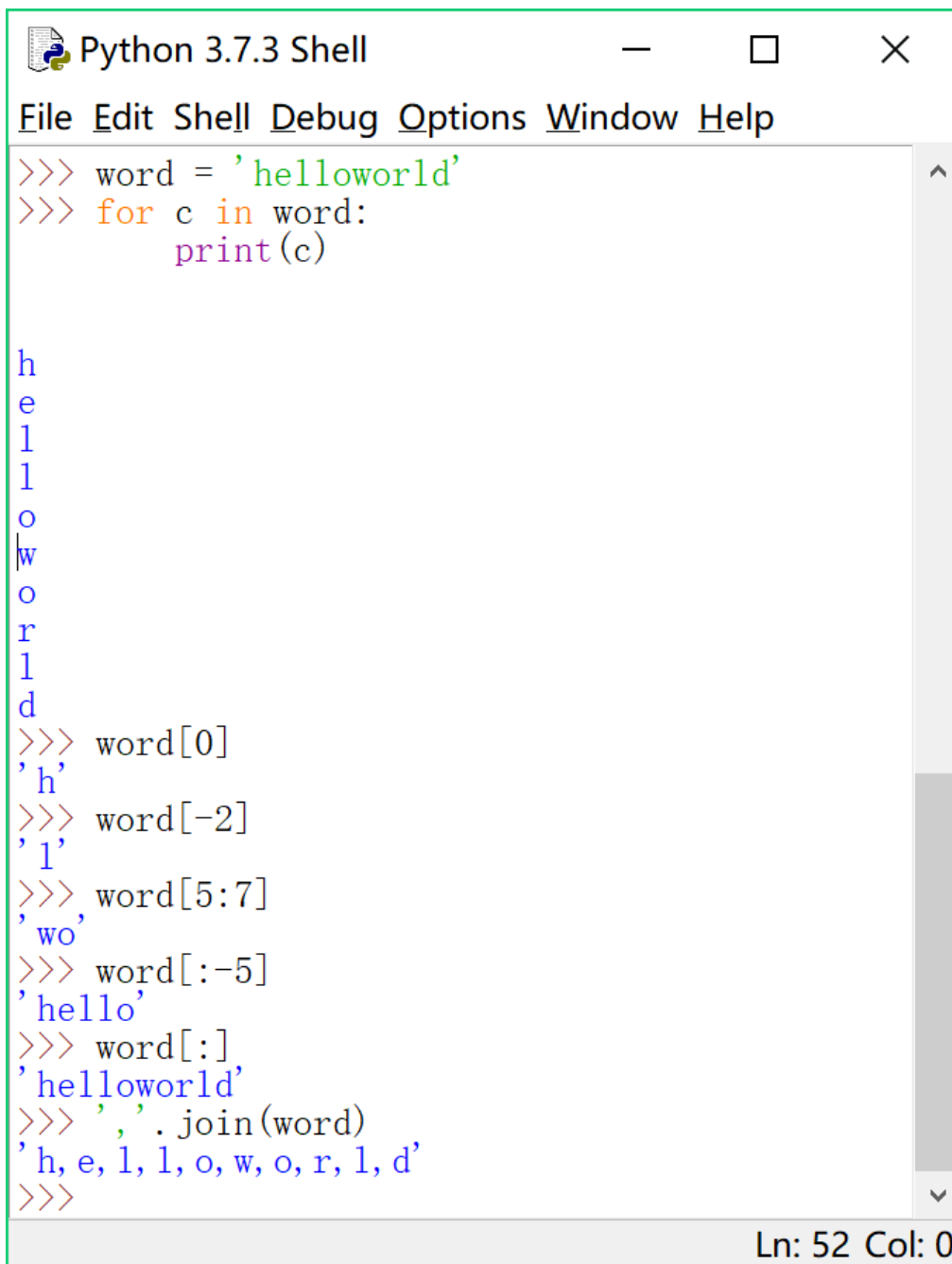
print (word[:-5])

print (word[:])
```

4. 连接字符

join方法也可以对字符串使用，作用就是用连接符把字符串中的每个字符重新连接成一个新字符串。

```
newword = ','.join(word)
```


A screenshot of a Python 3.7.3 Shell window. The window has a title bar with the text 'Python 3.7.3 Shell' and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main area is a text editor showing Python code. The code defines a string 'helloworld' and iterates over its characters, printing each one. This is followed by several indexing and slicing operations on the string. The output of these operations is shown in blue text. At the bottom right of the window, the status bar shows 'Ln: 52 Col: 0'.

```
>>> word = 'helloworld'
>>> for c in word:
    print(c)

h
e
l
l
o
w
o
r
l
d
>>> word[0]
'h'
>>> word[-2]
'l'
>>> word[5:7]
'wo'
>>> word[:-5]
'hello'
>>> word[:]
'helloworld'
>>> ','.join(word)
'h,e,l,l,o,w,o,r,l,d'
>>>
```

#===== 课外的话 =====#

前两天承蒙 MacTalk池建强 老师的推荐，让我们的学习队伍进一步壮大了。很多同学用的是Mac系统，而我是Linux党，很多Mac上的问题我没法帮忙解决。建议关注池老师的微信号，他经常会介绍一些Mac的使用技巧，让你更好地发挥Mac的强大功能。不用Mac的同学也建议去看看，他的文章有关技术和人文，相信你会得到不少启发。

微信号：Sagacity-Mac，直接搜索 mactalk 也可以看到。

- [← 29.连接list](#)
- [31.读文件 →](#)

【Python 第31课】读文件

之前，我们写的程序绝大多数都依赖于从命令行输入。假如某个程序需要输入很多数据，比如一次考试的全班学生成绩，再这么输就略显痛苦了。一个常见的办法就是把学生的成绩都保存在一个文件中，然后让程序自己从这个文件里取数据。

要读取文件，先得有文件。我们新建个文件，就叫它data.txt。在里面随便写上一些话，保存。把这个文件放在接下来你打算保存代码的文件夹下，这么做是为了方便我们的程序找到它。准备工作就绪，可以来写我们的代码了。

打开一个文件的命令很简单：

```
open('文件名')
```

这里的文件名可以用文件的完整路径，也可以是相对路径。因为我们把要读取的文件和代码放在了同一个文件夹下，所以只需要写它的文件名就够了。

```
f = open('data.txt')
```

但这一步只是打开了一个文件，并没有得到其中的内容。变量f保存了这个文件，还需要去读取它的内容。你可以通过 **read()** 函数把文件内所有内容读进一个字符串中。

```
data = f.read()
```

做完对文件的操作之后，记得用 **close()** 关闭文件，释放资源。虽然现在这样一个很短的程序，不做这一步也不会影响运行结果。但养成好习惯，可以避免以后发生莫名的错误。

注意：**close()** 一定要有后面的括号，不然就没有调用这个函数。

完整程序示例：

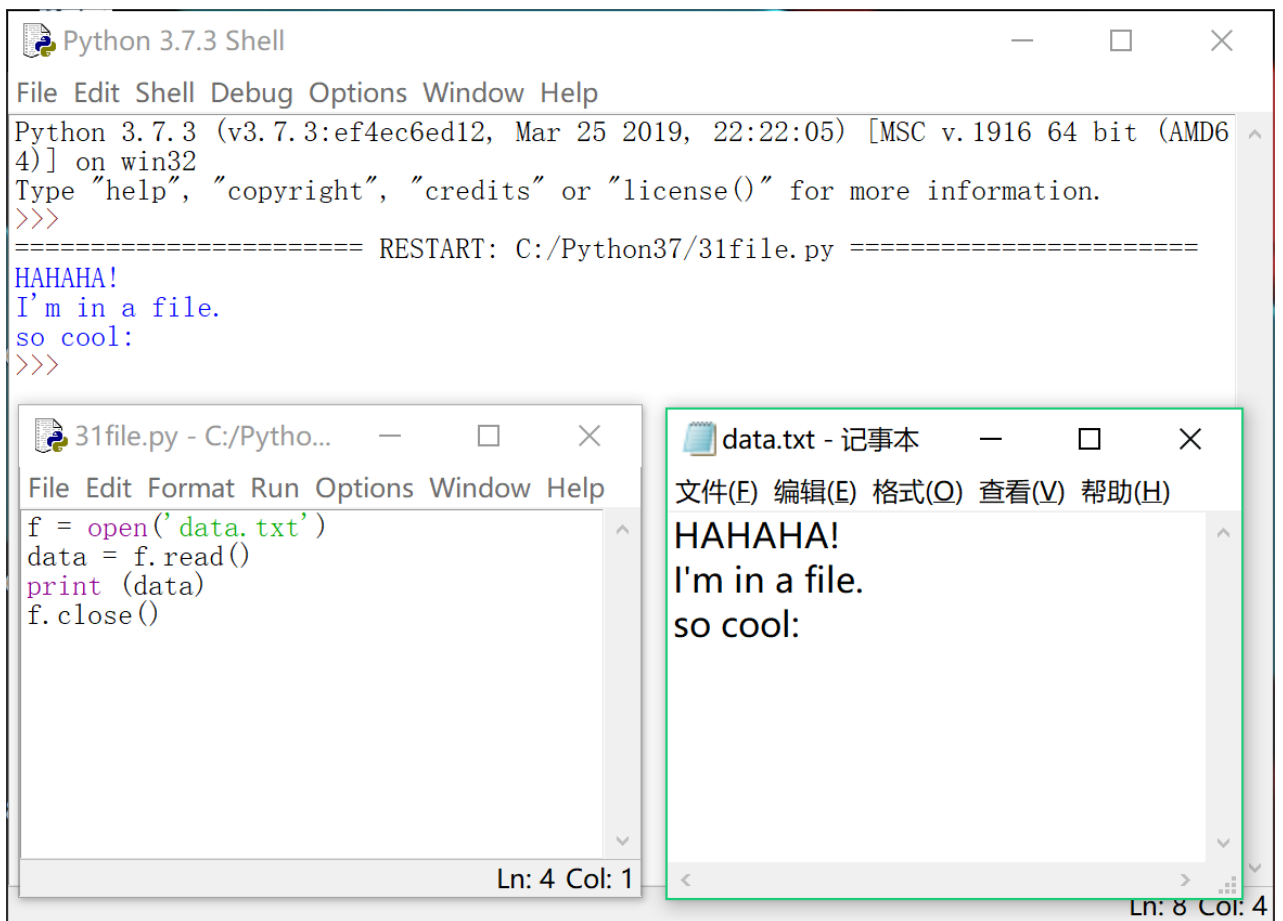
```
f = open('data.txt')
```

```
data = f.read()
```

```
print (data)
```

```
f.close()
```

是不是很简单？



读取文件内容的方法还有

`readline()` #读取一行内容

`readlines()` #把内容按行读取至一个list中

留给你们的任务：用这两个方法分别去替换例子程序的第二行，看看它们的区别。

- [← 30.字符串的索引和切片](#)
- [32.写文件 →](#)

【Python 第32课】写文件

来而不往，非礼也。有读，就要有写。

和把大象关进冰箱一样，写文件也需要三步：

1. 打开文件；
2. 把内容写入文件；
3. 关闭文件。

打开文件我们昨天已经讲过。但python默认是以 **只读模式** 打开文件。如果想要写入内容，在打开文件的时候需要指定打开模式为写入：

```
f = open('output.txt', 'w')
```

'w'就是writing，以这种模式打开文件，原来文件中的内容会被你新写入的内容覆盖掉，如果文件不存在，会自动创建文件。

之前不加参数时，open的模式默认为'r'，reading，只读模式，文件必须存在，否则引发异常。

另外还有一种常用模式是'a'，appending。它也是一种写入模式，但你写入的内容不会覆盖之前的内容，而是添加到原有文件内容后面。

写入内容的方法同样简单：

```
f.write('a string you want to write')
```

write的参数可以是一个字符串，或者一个字符串变量。

示例程序：

```
data = 'I will be in a file.\nSo cool!'
out = open('output.txt', 'w')
out.write(data)
out.close()
```

在你的程序保存目录下，打开output.txt就会看到结果。

注意：有些编辑器的默认执行路径并不是文件所在路径，所以可能造成运行完没发现 output.txt 的情况。这时可以去上层目录找找看。

#————— 课后作业 —————#

留两道课后作业：

1. 从一个文件中读出内容，保存至另一个文件。
2. 从控制台输入一些内容，保存至一个文件。

- [← 31.读文件](#)
- [33.处理文件中的数据 →](#)

【Python 第33课】处理文件中的数据

我们已经知道了如何读取和写入文件。有了这两个操作文件的方法，再加上对文件内容的处理，就能写一些小程序，解决不少日常的数据处理工作。

比如我现在拿到一份文档，里面有某个班级里所有学生的平时作业成绩。因为每个人交作业的次数不一样，所以成绩的数目也不同，没交作业的时候就没有分。我现在需要统计每个学生的平时作业总得分。

记得我小的时候，经常有同学被老师喊去做统计分数这种“苦力”。现在电脑普及了，再这么干就太弱了。用python，几行代码就可以搞定。

看一下我们的文档里的数据：

文件 *scores.txt*

刘备 23 35 44 47 51

关羽 60 77 68

张飞 97 99 89 91

诸葛亮 100

注意：为了减少麻烦，请保证文件中没有多余的空行。另外，因为windows系统默认会隐藏文件类型，所以请确保你创建的文件名正确，不要多加了后缀，误存为 scores.txt.txt

1.先把文件读进来，在这里，由于有中文字符，因此需要对字符进行编码，以免出现报错。

在 windows 中，如果用记事本打开，并且将这些文字一个个手动输入，默认中文编码为 gbk，因此需要：

```
f = open('scores.txt', encoding='gbk')
```

题外话：我并不建议在学习编程时使用windows默认的记事本，它会在文字编码上给你带来很多不必要的麻烦。推荐使用可以方便控制编码的notepad++或sublime text。

而如果是 Mac 或 Linux 系统，或者直接复制了我的文本（浏览器的中文编码多使用 utf-8），则使用了 utf-8 编码，因此需要：

```
f = open('scores.txt', encoding='utf-8')
```

这只是很粗略的总结，编码的问题一直都是编程的一大难题，如果出现报错，就换一种编码再试试。我们之后的教程都是在 windows 系统中的，所以使用 gbk 编码（但不排除你的系统可能是 utf8）。

2.取得文件中的数据。因为每一行都是一条学生成绩的记录，所以用 **readlines**，把每一行分开，便于之后的数据处理：

```
lines = f.readlines()
```

```
f.close()
```

提示：在程序中，经常使用print来查看数据的中间状态，可以便于你理解程序的运行。比如这里你可以print (lines)，看一下内容被存成了什么格式。

3.对每一条数据进行处理。按照空格，把姓名、每次的成绩分割开：

```
for line in lines:
    data = line.split()
```

这里用到之前 [28.字符串的分割](#) 课程中介绍过的 **split** 方法。

接下来的4、5两个步骤都是针对一条数据的处理，所以都是在 **for循环** 的内部。

4.整个程序最核心的部分到了。如何把一个学生的几次成绩合并，并保存起来呢？我的做法是：对于每一条数据，都新建一个字符串，把学生的名字和算好的总成绩保存进去。最后再把这些字符串一起保存到文件中：

```
sum = 0
```

```
score_list = data[1:] # 学生各门课的成绩列表
```

```

for score in score_list:

    sum += int(score)

result = '%s\t: %d\n' % (data[0], sum) # 名字和总分

```

这里几个要注意的点：

1. 对于每一行分割的数据，**data[0]** 是姓名，**data[1:]** 是所有成绩组成的列表。
2. **每次循环中，sum都要先清零。**
3. score是一个字符串，为了做计算，需要转成整数值int。
4. result中，我加了一个制表符\t和换行符\n，让输出的结果更好看些。

5.得到一个学生的总成绩后，把它添加到一个list中。

```

results.append(result)

```

results需要在循环之前 **初始化** results = []

6.最后，全部成绩处理完毕后，把results中的内容保存至文件。因为results是一个字符串组成的list，这里我们直接用writelines方法：

```

output = open('result.txt', 'w', encoding='gbk')

output.writelines(results)

output.close()

```

以下是完整程序，把其中print前面的注释符号去掉，可以查看关键步骤的数据状态。

```

f = open('scores.txt', encoding='gbk')

lines = f.readlines()

# print(lines)

f.close()

results = []

for line in lines:

    # print (line)

    data = line.split()

    # print (data)

    sum = 0

    score_list = data[1:]

    for score in score_list:

        sum += int(score)

    result = '%s \t: %d\n' % (data[0], sum)

    # print (result)

    results.append(result)

# print (results)

output = open('result.txt', 'w', encoding='gbk')

output.writelines(results)

```

output.close()

```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
>>>
===== RESTART: C:/Python37/33calscore.py =====
['刘备 23 35 44 47 51\n', '关羽 60 77 68\n', '张飞 97 99 89 91\n', '诸葛亮 100']
刘备 23 35 44 47 51

['刘备', '23', '35', '44', '47', '51']
刘备      : 200

关羽 60 77 68

['关羽', '60', '77', '68']
关羽      : 205

张飞 97 99 89 91

['张飞', '97', '99', '89', '91']
张飞      : 376

|
诸葛亮 100
['诸葛亮', '100']
诸葛亮    : 100

['刘备 \t: 200\n', '关羽 \t: 205\n', '张飞 \t: 376\n', '诸葛亮 \t: 100\n']
>>>
Ln: 40 Col: 0
```

大功告成，打开文件检验一下结果吧。

```
33calscore.py - C:\Python37\crossinsample\33calscore.py
File Edit Format Run Options Window Help
f = open('scores.txt', encoding='gbk')
lines = f.readlines()
# print(lines)
f.close()

results = []

for line in lines:
    # print (line)
    data = line.split()
    # print (data)

    sum = 0
    for score in data[1:]:
        point = int(score)
        sum += int(score)
    result = '%s \t: %d\n' % (data[0], sum)
    # print (result)

    results.append(result)

# print (results)
output = open('result.txt', 'w', encoding='gbk')
output.writelines(results)
output.close()
Ln: 16 Col: 25
```

scores.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

刘备 23 35 44 47 51
关羽 60 77 68
张飞 97 99 89 91
诸葛亮 100

result.txt - 记事本

文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)

刘备 : 200
关羽 : 205
张飞 : 376
诸葛亮 : 100

建议你把 print 前面的注释依次去掉，一步一步地去查看结果，理解每一步的作用。

- [← 32.写文件](#)
- [34.break →](#)

【Python 第34课】break

我们已经熟悉了循环的使用，包括while和for...in。while循环在条件不满足时结束，for循环遍历完序列后结束。如果在循环条件仍然满足或序列没有遍历完的时候，想要强行跳出循环，就需要用到break语句。

```
while True:

    a = input()

    if a == 'EOF':

        break
```

上面的程序不停接受用户输入。当用户输入一行“EOF”时，程序结束。

```
for i in range(10):

    a = input()

    if a == 'EOF':

        break
```

上面的程序接受用户10次输入，当用户输入一行“EOF”时，程序提前结束。

回到我们最早的那个猜数字小游戏。用break可以加上一个功能，当用户输入负数时，游戏就结束。如此一来，假如有玩家猜了几次之后仍然猜不中，一怒之下想要直接退出游戏，就猜一个负数。

添加的代码是：

```
if answer < 0:

    print ('Exit game...')

    break
```

与break类似的还有一个continue语句，明天说。

- [← 33.处理文件中的数据](#)
- [35.continue →](#)

【Python 第35课】continue

break是彻底地跳出循环，而continue只是略过本次循环的余下内容，直接进入下一次循环。

在我们前面写的那个统计分数的程序里，如果有成绩不足60分，就不记入总成绩。当然，你可以用if判断来实现这个效果。但我们今天要说另一种方法：continue。

```
for score in data[1:]:
    point = int(score)
    if point < 60:
        continue
    sum += point
```



注意：无论是continue还是break，其改变的仅仅是当前所处的最内层循环的运行，如果外层还有循环，并不会因此略过或跳出。

在脑中模拟运行下面这段程序，想想会输出什么结果。再敲到代码里验证一下：

```
i = 0
while i < 5:
    i += 1
    for j in range(3):
        print(j)
        if j == 2:
```

```
        break
    for k in range(3):
        if k == 2:
            continue
        print (k)
    if i > 3:
        break
    print (i)
```

- [← 34.break](#)
- [36.异常处理 →](#)

【Python 第36课】异常处理

在程序运行时，如果我们的代码引发了错误，python就会中断程序，并且输出错误提示。

比如我们写了一句：

```
print (int('0.5'))
```

运行后程序得到错误提示：

Traceback (most recent call last):

File "C:/Python37/test.py", line 1, in <module>

```
print (int('0.5'))
```

ValueError: invalid literal for int() with base 10: '0.5'

意思是，在test.py这个文件，第1行，`print (int('0.5'))`这里，你拿了一个不是10进制能够表示的字符，我没法把它转成int值。

上面的错误可以避免，但在实际的应用中，有很多错误是开发者无法控制的，例如用户输入了一个不合规定的值，或者需要打开的文件不存在。这些情况被称作“异常”，一个好的程序需要能处理可能发生的异常，避免程序因此而中断。

例如我们去打开一个文件：

```
f = open('non-exist.txt')
```

```
print ('File opened!')
```

```
f.close()
```

假如这个文件因为某种原因并没有出现在应该出现的文件夹里，程序就会报错：

IOError: [Errno 2] No such file or directory: 'non-exist.txt'

程序在出错处中断，后面的print不会被执行。

在python中，可以使用try...except语句来处理异常。做法是，把可能引发异常的语句放在try-块中，把处理异常的语句放在except-块中。

把刚才那段代码放入try...except中：

```
try:
```

```
    f = open('non-exist.txt')
```

```
    print ('File opened!')
```

```
    f.close()
```

```
except:
```

```
    print ('File not exists.')
```

```
print ('Done')
```

当程序在try内部打开文件引发异常时，会跳过try中剩下的代码，直接跳转到except中的语句处理异常。于是输出了“File not exists.”。如果文件被顺利打开，则会输出“File opened!”，而不会去执行except中的语句。

但无论如何，整个程序不会中断，最后的“Done”都会被输出。

在try...except语句中，try中引发的异常就像是扔出了一只飞盘，而except就是一只灵敏的狗，总能准确地接住飞盘。

- [← 35.continue](#)
- [37.字典 →](#)

【Python 第37课】字典

今天介绍一个python中的基本类型--字典（dictionary）。

字典这种数据结构有点像是我们平常用的通讯录，有一个名字和这个名字对应的信息。在字典中，名字叫做“键”，对应的内容信息叫做“值”。字典就是一个键/值对的集合。

它的基本格式是（key是键，value是值）：

```
d = {key1 : value1, key2 : value2}
```

键/值对用冒号分割，每个对之间用逗号分割，整个字典包括在花括号中。

关于字典的键要注意的是：

- 1.键必须是唯一的；
- 2.键只能是简单对象，比如字符串、整数、浮点数、bool值。

list就不能作为键，但是可以作为值。

举个简单的字典例子：

```
score = {  
    '萧峰': 95,  
    '段誉': 97,  
    '虚竹': 89  
}
```

python字典中的键/值对没有顺序，我们无法用索引访问字典中的某一项，而是要用键来访问。

```
print (score['段誉'])
```

注意，如果你的键是字符串，通过键访问的时候就需要加引号，如果是数字作为键则不用。

如果你提供的键在字典中不存在，则会报错。另一种访问字典中元素的方法是：

```
score.get('慕容复')
```

这种方法的好处是，即使提供的键不存在，也不会报错，只会返回 None

字典也可以通过for...in遍历：

```
for name in score:  
    print (score[name])
```

注意，遍历的变量中存储的是字典的键。

如果要改变某一项的值，就直接给这一项赋值：

```
score['虚竹'] = 91
```

增加一项字典项的方法是，给一个新键赋值：

```
score['慕容复'] = 88
```

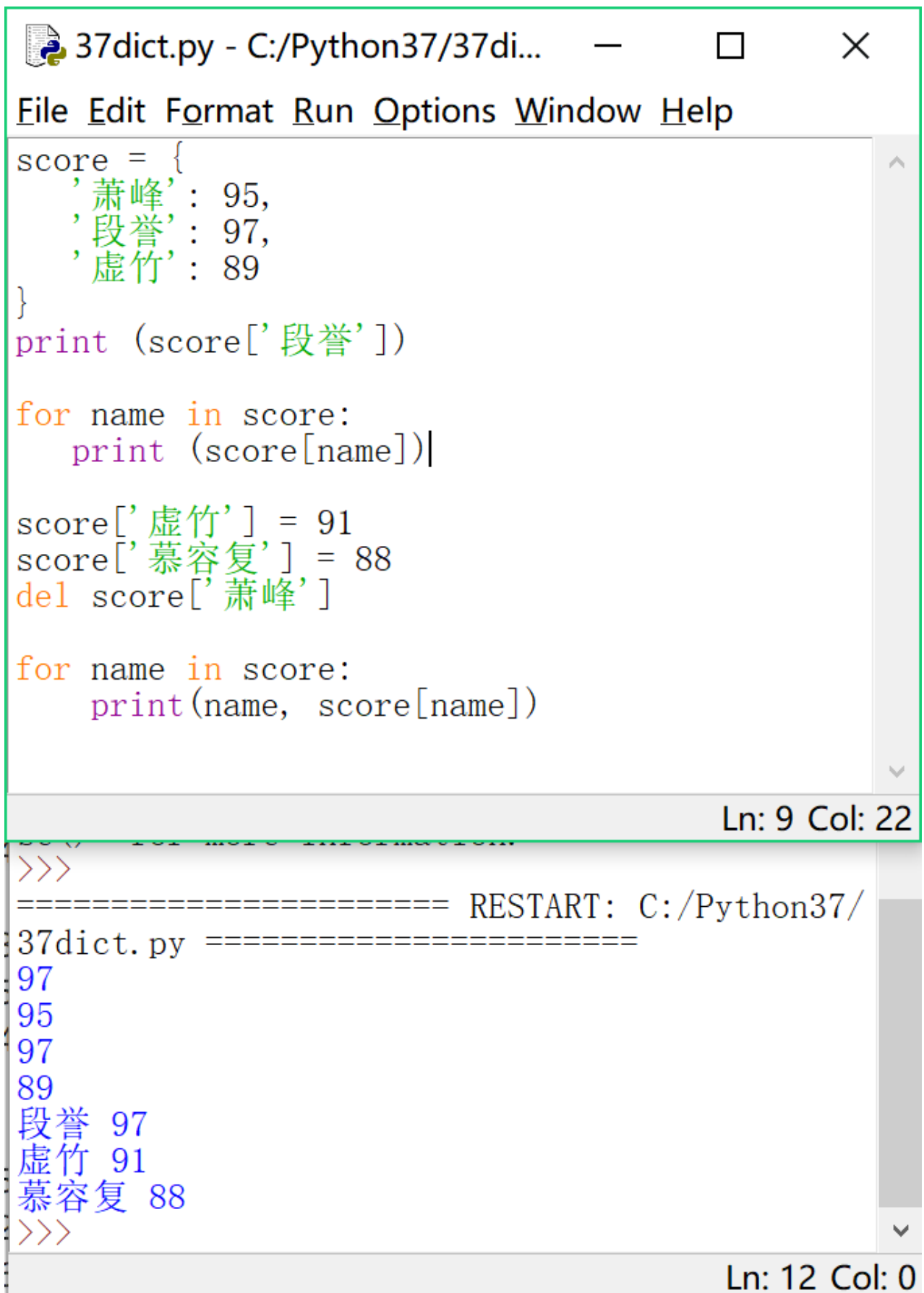
删除一项字典项的方法是del:

```
del score['萧峰']
```

注意，这个键必须已存在于字典中。

如果你想新建一个空的字典，只需要：

```
d = {}
```

A screenshot of a Python 3.7 IDE window titled "37dict.py - C:/Python37/37di...". The window has a menu bar with "File", "Edit", "Format", "Run", "Options", "Window", and "Help". The main editor area contains Python code that defines a dictionary 'score' with names and scores, prints the value for '段誉', iterates over the dictionary, updates '虚竹' to 91, adds '慕容复' with 88, deletes '萧峰', and then prints all remaining items. The status bar at the bottom right shows "Ln: 9 Col: 22". Below the editor is a console window showing the execution output: the dictionary contents, the value for '段誉' (97), the updated '虚竹' (91), and the new entry '慕容复' (88). The console also shows a "RESTART" message. The console status bar at the bottom right shows "Ln: 12 Col: 0".

```
score = {  
    '萧峰': 95,  
    '段誉': 97,  
    '虚竹': 89  
}  
print (score['段誉'])  
  
for name in score:  
    print (score[name])  
  
score['虚竹'] = 91  
score['慕容复'] = 88  
del score['萧峰']  
  
for name in score:  
    print(name, score[name])
```

Ln: 9 Col: 22

```
>>>  
===== RESTART: C:/Python37/  
37dict.py =====  
97  
95  
97  
89  
段誉 97  
虚竹 91  
慕容复 88  
>>>
```

Ln: 12 Col: 0

- [← 36.异常处理](#)
- [38.模块 →](#)

【Python 第38课】模块

如果说我比别人看得更远些,那是因为我站在了巨人的肩上。

-- 牛顿

python自带了功能丰富的标准库，另外还有数量庞大的各种第三方库。使用这些“巨人的”代码，可以让开发事半功倍，就像用积木一样拼出你要的程序。

使用这些功能的基本方法就是使用模块。通过函数，可以在程序里重用代码；通过模块，则可以重用别的程序中的代码。

模块可以理解为是一个包含了函数和变量的py文件。在你的程序中引入了某个模块，就可以使用其中的函数和变量。

来看一个我们之前使用过的模块：

```
import random
```

import语句告诉python，我们要用random模块中的内容。然后便可以使用random中的方法，比如：

```
random.randint(1, 10)
```

```
random.choice([1, 3, 5])
```

注意，函数前面需要加上“random”，这样python才知道你是要调用random中的方法。

想知道random有哪些函数和变量，可以用dir()方法：

```
dir(random)
```

如果你只是用到random中的某一个函数或变量，也可以通过from...import...指明：

```
from math import pi
```

```
print (pi)
```

为了便于理解和避免冲突，你还可以给引入的方法换个名字：

```
from math import pi as math_pi
```

```
print (math_pi)
```

```
命令提示符 - python
Microsoft Windows [版本 10.0.17134.765]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\>python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import random
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom', 'TWOPI', '_BuiltinMethodType', '_MethodType', '_Sequence', '_Set', '_all_', '_builtins', '_cached_', '_doc_', '_file_', '_loader_', '_name_', '_package_', '_spec_', '_acos', '_bisect', '_ceil', '_cos', '_e', '_exp', '_inst', '_itertools', '_log', '_os', '_pi', '_random', '_sha512', '_sin', '_sqrt', '_test', '_test_generator', '_urandom', '_warn', '_betavariate', '_choice', '_choices', '_expovariate', '_gammavariate', '_gauss', '_getrandbits', '_getstate', '_lognormvariate', '_normalvariate', '_paretovariate', '_randint', '_random', '_randrange', '_sample', '_seed', '_setstate', '_shuffle', '_triangular', '_uniform', '_vonmisesvariate', '_weibullvariate']
>>> random.randint(1, 10)
4
>>> random.choice([1, 3, 5])
5
>>> from math import pi as math_pi
>>> math_pi
3.141592653589793
>>> _
```

想要了解python有哪些常用库，可自行搜索。我在群共享里上传了一份中文版的python标准库的非官方文档，供参考。

- [← 37.字典](#)
- [39.用文件保存游戏 \(1\) →](#)

【Python 第39课】用文件保存游戏（1）

到目前为止，python最入门的语法我们都已经有所涉及，相信大家一路学过来，多少也能写出一些小程序。在接下来的课程中，我会基于实例来更深入地介绍python。

现在，我要在最早我们开发的那个猜数字游戏的基础上，增加保存成绩的功能。用到的方法就是前几课讲过的文件读写。今天是第一部分。

在动手写代码前，先想清楚我们要解决什么问题，打算怎么去解决。你可以选择根据每次游戏算出一个得分，记录累计的得分。也可以让每次猜错都扣xx分，猜对之后再加xx分，记录当前分数。而我现在打算记录下我玩了多少次，最快猜出来的轮数，以及平均每次猜对用的轮数。

于是，我要在文件中记录3个数字，如：

3 5 31

它们分别是：总游戏次数，最快猜出的轮数，和猜过的总轮数（这里我选择记录总轮数，然后每次再算出平均轮数）

接下来可以往代码里加功能了，首先是读取成绩。新建好一个game.txt，里面写上：

0 0 0

作为程序的初始数据。

用之前的方法，读入文件：

```
f = open('game.txt')
score = f.read().split()
```

这里，我用了open方法，它和file()的效果一样。另外，我还用了绝对路径。当你写这个程序时，记得用你自己电脑上的路径。

为便于理解，把数据读进来后，分别存在3个变量中。

```
game_times = int(score[0])
min_times = int(score[1])
total_times = int(score[2])
```

平均轮数根据总轮数和游戏次数相除得到：

```
avg_times = total_times / game_times
```

注意两点：

1.python 3的除法运算中，一个"/"的除法，结果为浮点数。如果需要得到结果的整数部分（不是对结果四舍五入），要用"//"，即两个斜杠。而"%"代表对结果取余数。

2.因为0是不能作为除数的，所以这里还需要加上判断：

```
if game_times > 0:
    avg_times = total_times / game_times
else:
    avg_times = 0
```

然后，在让玩家开始猜数字前，输出他之前的成绩信息：

```
print ('你已经玩了%d次，最少%d轮猜出答案，平均%.2f轮猜出答案' % (game_times, min_times, avg_times))
```

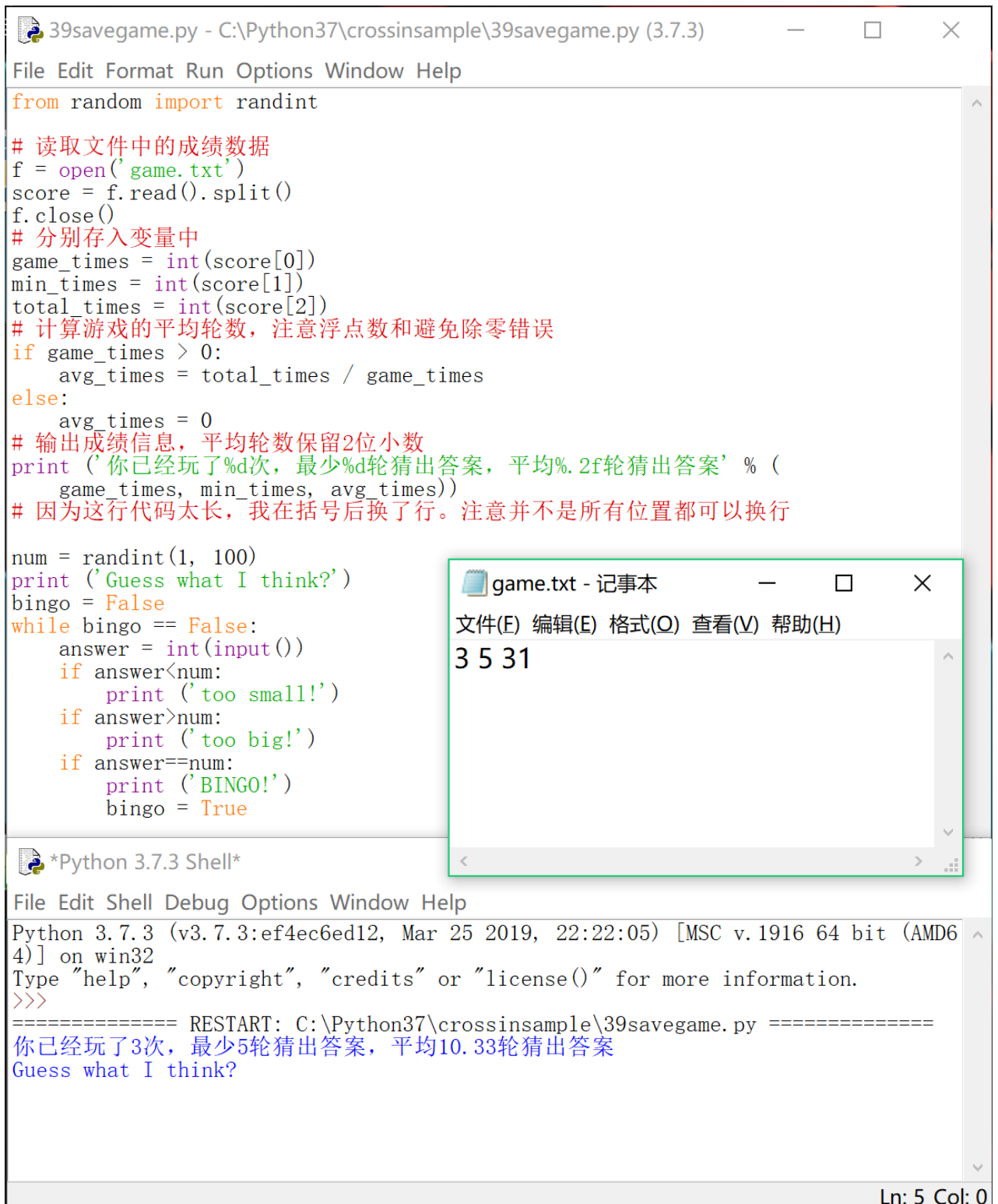
%.2f这样的写法我们以前也用过，作用是保留两位小数。

好了，运行程序看一下效果：

你已经玩了0次，最少0轮猜出答案，平均0轮猜出答案

由于还没有做保存功能，我们手动去文件里改一下成绩看运行效果。（其实有些小游戏就可以用类似的方法作弊）

下一课，我们要把真实的游戏数据保存到文件中。



```
39savegame.py - C:\Python37\crossinsample\39savegame.py (3.7.3)
File Edit Format Run Options Window Help

from random import randint

# 读取文件中的成绩数据
f = open('game.txt')
score = f.read().split()
f.close()
# 分别存入变量中
game_times = int(score[0])
min_times = int(score[1])
total_times = int(score[2])
# 计算游戏的平均轮数，注意浮点数和避免除零错误
if game_times > 0:
    avg_times = total_times / game_times
else:
    avg_times = 0
# 输出成绩信息，平均轮数保留2位小数
print('你已经玩了%d次，最少%d轮猜出答案，平均%.2f轮猜出答案' % (
    game_times, min_times, avg_times))
# 因为这行代码太长，我在括号后换了行。注意并不是所有位置都可以换行

num = randint(1, 100)
print('Guess what I think?')
bingo = False
while bingo == False:
    answer = int(input())
    if answer < num:
        print('too small!')
    if answer > num:
        print('too big!')
    if answer == num:
        print('BINGO!')
        bingo = True

*Python 3.7.3 Shell*
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python37\crossinsample\39savegame.py =====
你已经玩了3次，最少5轮猜出答案，平均10.33轮猜出答案
Guess what I think?
```

- [← 38.模块](#)
- [40.用文件保存游戏 \(2\) →](#)

【Python 第40课】用文件保存游戏（2）

话接上回。我们已经能从文件中读取游戏成绩数据了，接下来就要考虑，怎么把我们每次游戏的结果保存进去。

首先，我们需要有一个变量来记录每次游戏所用的轮数：

```
times = 0
```

然后在游戏每进行一轮的时候，累加这个变量：

```
times += 1
```

当游戏结束后，我们要把这个变量的值，也就是本次游戏的数据，添加到我们的记录中。

如果是第一次玩，或者本次的轮数比最小轮数还少，就记录本次成绩为最小轮数：

```
if game_times == 0 or times < min_times:  
    min_times = times
```

把本次轮数加到游戏总轮数里：

```
total_times += times
```

把游戏次数加1：

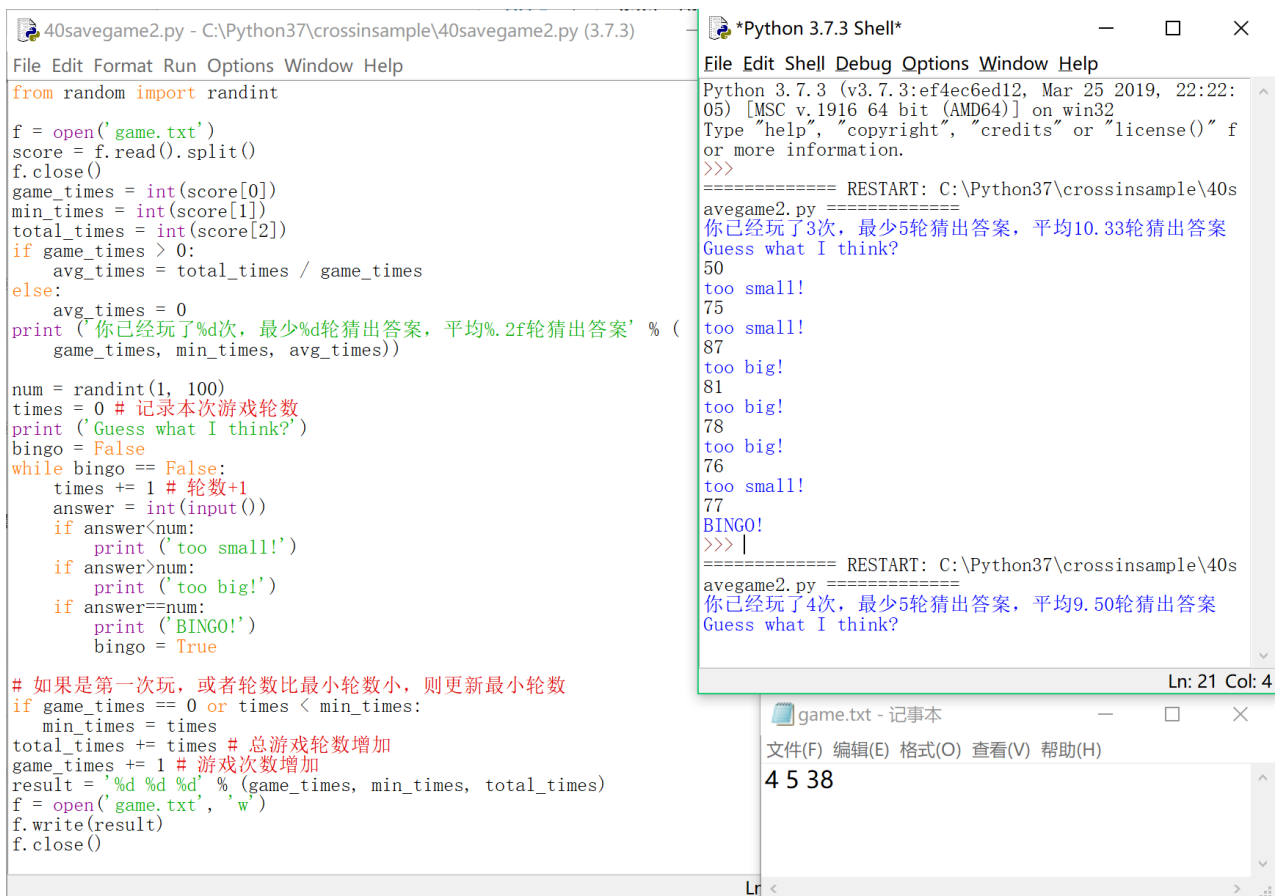
```
game_times += 1
```

现在有了我们需要的数据，把它们拼成我们需要存储的格式：

```
result = '%d %d %d' % (game_times, min_times, total_times)
```

写入到文件中：

```
f = open('game.txt', 'w')  
f.write(result)  
f.close()
```



```
40savegame2.py - C:\Python37\crossinsample\40savegame2.py (3.7.3)
File Edit Format Run Options Window Help
from random import randint
f = open('game.txt')
score = f.read().split()
f.close()
game_times = int(score[0])
min_times = int(score[1])
total_times = int(score[2])
if game_times > 0:
    avg_times = total_times / game_times
else:
    avg_times = 0
print('你已经玩了%d次, 最少%d轮猜出答案, 平均%.2f轮猜出答案' % (
    game_times, min_times, avg_times))

num = randint(1, 100)
times = 0 # 记录本次游戏轮数
print('Guess what I think?')
bingo = False
while bingo == False:
    times += 1 # 轮数+1
    answer = int(input())
    if answer < num:
        print('too small!')
    if answer > num:
        print('too big!')
    if answer == num:
        print('BINGO!')
        bingo = True

# 如果是第一次玩, 或者轮数比最小轮数小, 则更新最小轮数
if game_times == 0 or times < min_times:
    min_times = times
total_times += times # 总游戏轮数增加
game_times += 1 # 游戏次数增加
result = '%d %d %d' % (game_times, min_times, total_times)
f = open('game.txt', 'w')
f.write(result)
f.close()

Python 3.7.3 Shell*
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" f
or more information.
>>>
===== RESTART: C:\Python37\crossinsample\40s
avegame2.py =====
你已经玩了3次, 最少5轮猜出答案, 平均10.33轮猜出答案
Guess what I think?
50
too small!
75
too small!
87
too big!
81
too big!
78
too big!
76
too small!
77
BINGO!
>>> |
===== RESTART: C:\Python37\crossinsample\40s
avegame2.py =====
你已经玩了4次, 最少5轮猜出答案, 平均9.50轮猜出答案
Guess what I think?

game.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
4 5 38
```

按照类似的方法，你也可以记录一些其他的数据，比如设定一种记分规则作为游戏得分。虽然在这个小游戏里，记录成绩并没有太大的乐趣，但通过文件来记录数据的方法，以后会在很多程序中派上用场。

- [← 39.用文件保存游戏（1）](#)
- [41.用文件保存游戏（3） →](#)

【Python 第41课】用文件保存游戏（3）

你的小游戏现在已经可以保存成绩了，但只有一组成绩，不管谁来玩，都会算在里面。所以今天我还要加上一个更多的功能：存储多组成绩。玩家需要做的就是，在游戏开始前，输入自己的名字。而我会根据这个名字记录他的成绩。这个功能所用到的内容我们几乎都说过，现在要把它们结合起来。

首先要输入名字，这是我们用来区分玩家成绩的依据：

```
name = input('请输入你的名字：')
```

接下来，我们读取文件。与之前不同，我们用`readlines`把每组成绩分开来：

```
lines = f.readlines()
```

再用一个字典来记录所有的成绩：

```
scores = {}  
  
for l in lines:  
    s = l.split()  
    scores[s[0]] = s[1:]
```

这个字典中，每一项的`key`是玩家的名字，`value`是一个由剩下的数据组成的数组。这里每一个`value`就相当于我们之前的成绩数据。

我们要找到当前玩家的数据：

```
score = scores.get(name)
```

字典类的`get`方法是按照给定`key`寻找对应项，如果不存在这样的`key`，就返回空值`None`。

所以如果没有找到该玩家的数据，说明他是一个新玩家，我们给他初始化一组成绩：

```
if score is None:  
    score = [0, 0, 0]
```

这是我们拿到的`score`，已经和上一课中的`score`一样了，因此剩下的很多代码都不用改动。

当游戏结束，记录成绩的时候，和之前的方法不一样。我们不能直接把这次成绩存到文件里，那样就会覆盖掉别人的成绩。必须先把成绩更新到`scores`字典中，再统一写回文件中。

把成绩更新到`scores`中，如果没有这一项，会自动生成新条目：

```
scores[name] = [str(game_times), str(min_times), str(total_times)]
```

对于每一项成绩，我们要将其格式化：

```
result = ''  
  
for n in scores:  
    line = n + ' ' + ' '.join(scores[n]) + '\n'  
    result += line
```

把`scores`中的每一项按照“名字 游戏次数 最低轮数 总轮数\n”的格式拼成字符串，再全部放到`result`里，就得到了我们要保存的结果。

最后就和之前一样，把`result`保存到文件中。

```
41savegame3.py - C:\Python37\crossinsample\41savegame3.py (3.7.3)
File Edit Format Run Options Window Help

from random import randint

name = input('请输入你的名字：') # 输入玩家名字

f = open('game2.txt', encoding='gbk')
lines = f.readlines()
f.close()
scores = {} # 初始化一个空字典
for l in lines:
    s = l.split() # 把每一行的数据拆分成list
    scores[s[0]] = s[1:] # 第一项作为key, 剩下的作为value
score = scores.get(name) # 找到当前玩家的数据
if score is None: # 如果没有找到
    score = [0, 0, 0]

game_times = int(score[0])
min_times = int(score[1])
total_times = int(score[2])
if game_times > 0:
    avg_times = total_times / game_times
else:
    avg_times = 0
# 加上显示玩家的名字
print('%.2f, 你已经玩了%d次, 最少%d轮猜出答案, 平均%.2f轮猜出答案' % (
    name, game_times, min_times, avg_times))

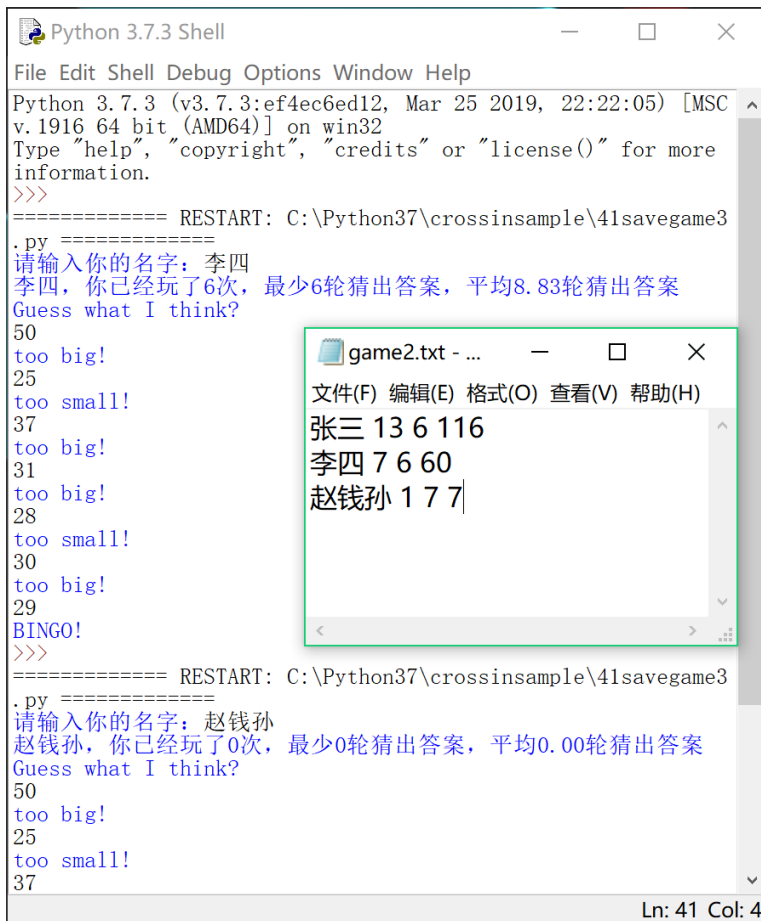
num = randint(1, 100)
times = 0 # 记录本次游戏轮数
print('Guess what I think?')

num = randint(1, 100)
times = 0 # 记录本次游戏轮数
print('Guess what I think?')
bingo = False
while bingo == False:
    times += 1 # 轮数+1
    answer = int(input())
    if answer < num:
        print('too small!')
    if answer > num:
        print('too big!')
    if answer == num:
        print('BINGO!')
        bingo = True

if game_times == 0 or times < min_times:
    min_times = times
total_times += times
game_times += 1

# 把成绩更新到对应的玩家数据中
# 加str转成字符串, 为后面的格式化做准备
scores[name] = [str(game_times), str(min_times), str(total_times)]
result = '' # 初始化一个空字符串, 用来储存数据
for n in scores:
    # 把数据按照"name game_times min_times total_times"格式化
    # 结尾要加上\n换行
    line = n + ', ' + ', '.join(scores[n]) + '\n'
    result += line # 添加到result中

f = open('game2.txt', 'w', encoding='gbk')
f.write(result)
f.close()
```



```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC
v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>>
===== RESTART: C:\Python37\crossinsample\41savegame3
.py =====
请输入你的名字: 李四
李四, 你已经玩了6次, 最少6轮猜出答案, 平均8.83轮猜出答案
Guess what I think?
50
too big!
25
too small!
37
too big!
31
too big!
28
too small!
30
too big!
29
BINGO!
>>>
===== RESTART: C:\Python37\crossinsample\41savegame3
.py =====
请输入你的名字: 赵钱孙
赵钱孙, 你已经玩了0次, 最少0轮猜出答案, 平均0.00轮猜出答案
Guess what I think?
50
too big!
25
too small!
37
```

game2.txt - ...

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

张三 13 6 116
李四 7 6 60
赵钱孙 1 7 7

Ln: 41 Col: 4

如果你充分理解了这个程序，恭喜你，你对文件处理已经有了一个基本的了解。在日常工作学习中，如果需要处理一些大量重复机械的文件操作，比如整理格式、更改文件中的部分文字、统计数据等等，都可以试着用python来解决。

- [← 40.用文件保存游戏 \(2\)](#)
- [42.函数的默认参数 →](#)

【Python 第42课】函数的默认参数

今天分享一点小技巧。之前我们用过函数，比如：

```
def hello(name):  
    print ('hello ' + name)
```

然后我们去调用这个函数：

```
hello('world')
```

程序就会输出

```
hello world
```

如果很多时候，我们都是用world来调用这个函数，少数情况才会去改参数。那么，我们就可以给这个函数一个默认参数：

```
def hello(name = 'world'):  
    print ('hello ' + name)
```

当你没有提供参数值时，这个参数就会使用默认值；如果你提供了，就用你给的。

这样，在默认情况下，你只要调用

```
hello()
```

就可以输出

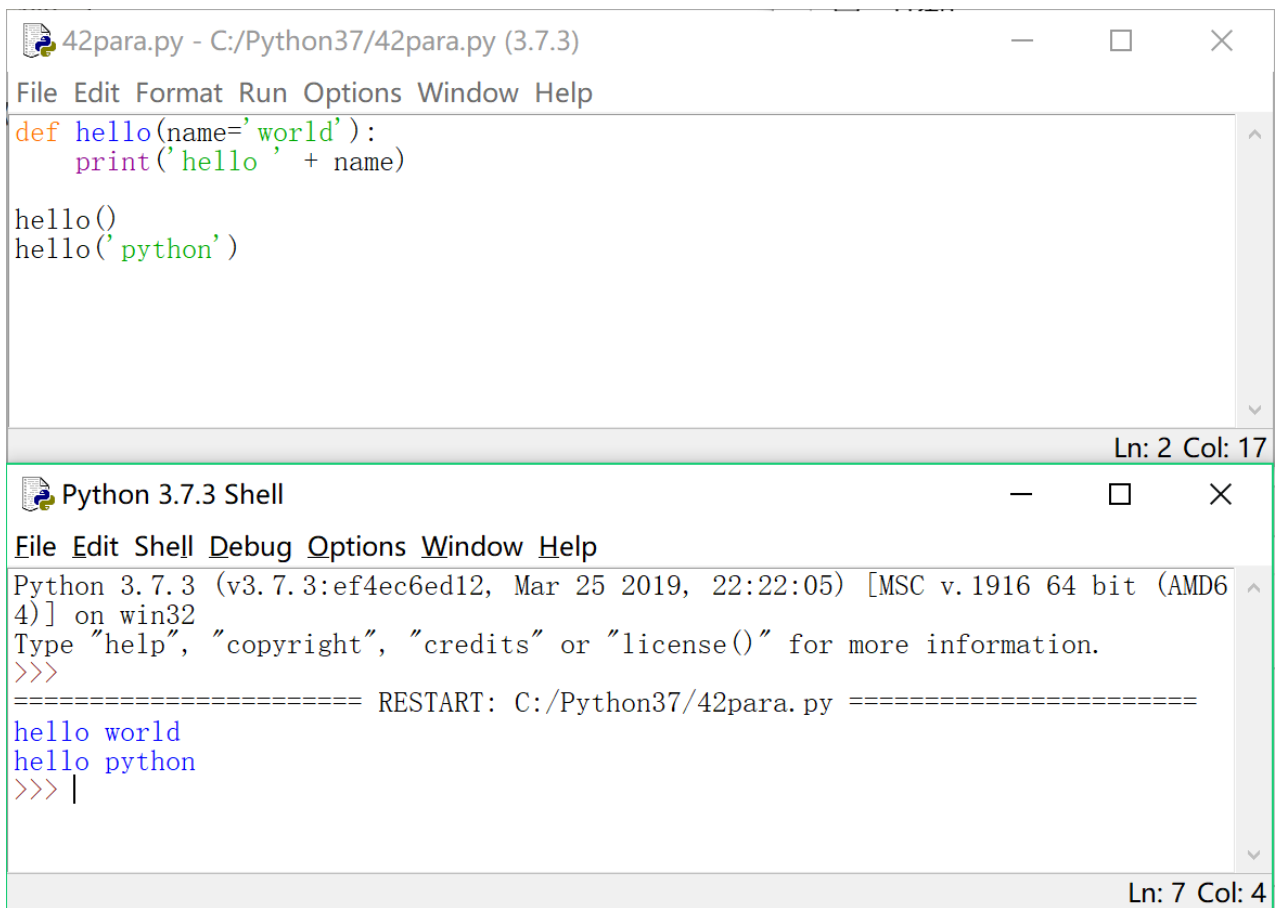
```
hello world
```

同样你也可以指定参数：

```
hello('python')
```

输出

```
hello python
```

The image shows two windows from a Python IDE. The top window, titled '42para.py - C:/Python37/42para.py (3.7.3)', contains the following Python code:

```
def hello(name='world'):
    print('hello ' + name)

hello()
hello('python')
```

The bottom window, titled 'Python 3.7.3 Shell', shows the execution of the script. It displays the Python version and architecture, followed by the output of the script:

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Python37/42para.py =====
hello world
hello python
>>> |
```

注意，当函数有多个参数时，如果你想给部分参数提供默认参数，那么这些参数必须在参数的末尾。比如：

```
def func(a, b=5):
```

是正确的

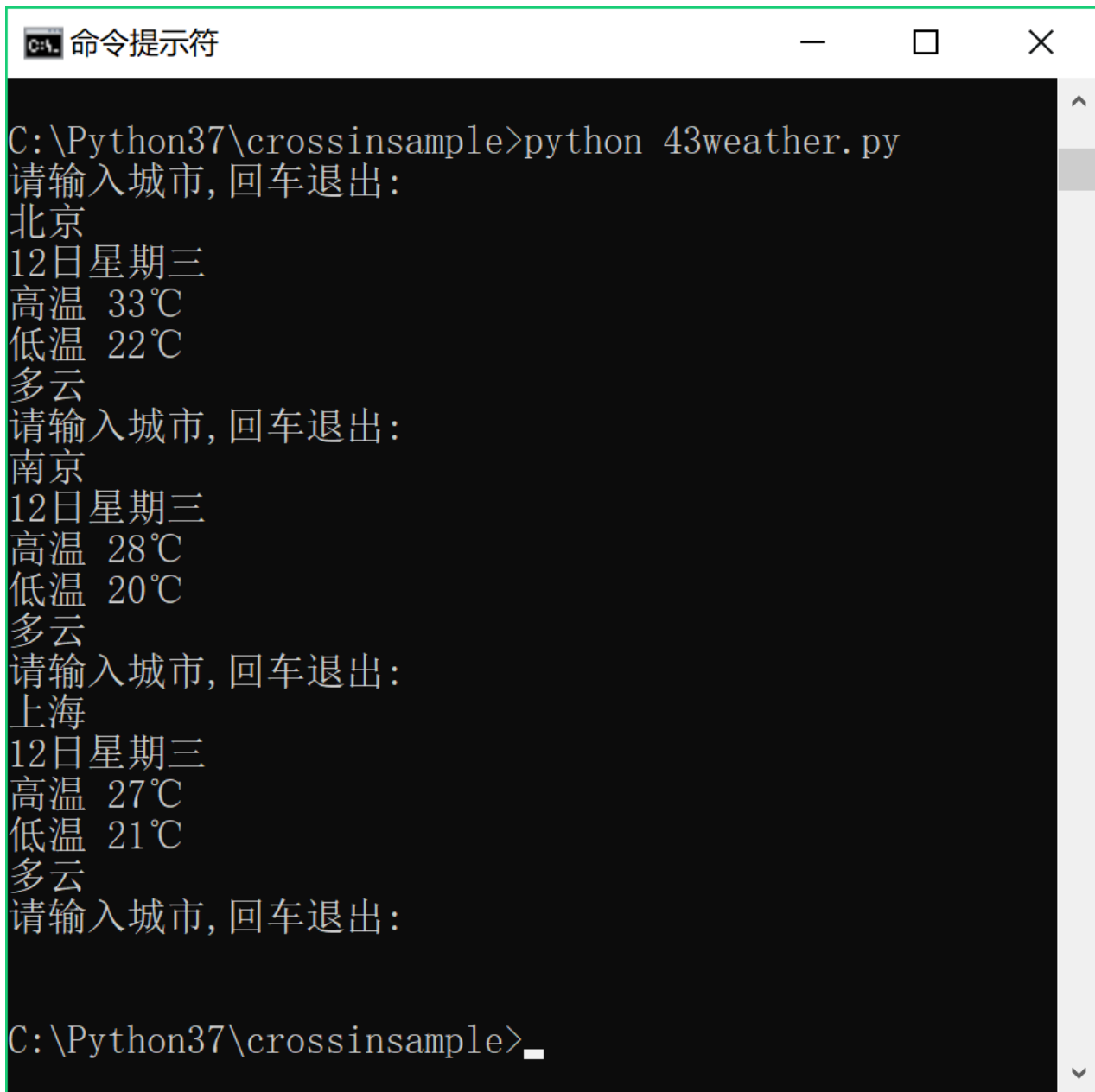
```
def func(a=5, b):
```

就会出错

- [← 41.用文件保存游戏 \(3\)](#)
- [43.查天气 \(1\) →](#)

【Python 第43课】查天气（1）

给大家看一个程序：



```
C:\Python37\crossinsample>python 43weather.py
请输入城市, 回车退出:
北京
12日星期三
高温 33°C
低温 22°C
多云
请输入城市, 回车退出:
南京
12日星期三
高温 28°C
低温 20°C
多云
请输入城市, 回车退出:
上海
12日星期三
高温 27°C
低温 21°C
多云
请输入城市, 回车退出:

C:\Python37\crossinsample>
```

你输入一个城市的名称，就会告诉你这个城市现在的天气情况。接下来的几节课，我就说一下怎么实现这样一个小程序。

之所以能知道一个城市的天气，是因为用了这样一个天气查询接口：

http://wthrcdn.etouch.cn/weather_mini?city=北京

此接口可以返回 city 对应城市昨日天气及5天内的天气预报，包括气温、指数、空气质量、风力等。

试着在浏览器里访问一下该网址，你会自动下载一个 weather_mini 的文件，这个文件里面就是咱们想要的数。文件用记事本就可以打开，打开后，里面看上去有点像python中字典类的文字是一种称作json格式的数据。（因为接口有时候会对结果进行 gzip 压缩，有时打开看到的内容会是乱码）

而我们的程序要做的事情，就是按照用户输入的城市名称，去这个接口请求对应的天气信息，再把结果展示给用户。

于是，在这个程序中，我们要用到一个新模块：

requests

用来发送网络请求，获取数据

听上去似乎还挺不算太复杂？这是因为 requests 库帮我们做了很多原本需要我们完成的事情，包括gzip压缩、字符编码、json的自动处理。如果用Python自带的urllib.request库，工作量就会增加很多。

今天先卖个关子，不说具体的写法。想挑战的同学可以试试在我说之前就把这个程序搞定。

- [← 42.函数的默认参数](#)
- [44.查天气 \(2\) →](#)

【Python 第44课】查天气（2）

今天有些事情耽搁了，课程来得晚了些。

先来看python中的requests库，这是一个用来获取网络资源的模块。我们平常上网，在浏览器地址栏中输入一个网址，浏览器根据这个网址拿到一些内容，然后展现在页面上，这大约就是浏览网页的过程。类似的，requests会跟你提供的网址，请求对应的内容。

requests是一个第三方库，它需要下载后安装到python的安装目录下，所以我们首先要做的是下载安装，这些都可以使用pip来进行：

```
pip install requests
```

安装完毕后，我们在代码中：

```
import requests
```

如果没有报错，就说明安装成功了。

使用requests库打开一个链接的方法很简单：

```
import requests

req = requests.get('http://www.baidu.com')

print(req)

req.encoding = 'utf8'

content = req.text

print(content)
```

我们引入requests的模块，用其中的get方法打开百度，并将结果保存在变量req中，输出req可以看到，返回了一个Response和一个数字，其中数字200就代表请求成功。

要查看访问到的数据，只要访问返回结果的text属性即可。运行后，你会看到控制台中输出了一堆看不懂的代码文字。这段代码中有html，有css，还有javascript。我们在浏览器中看到的网页大部分就是由这些代码所组成。如果你把content保存到一个以“.html”结尾的文件中（保存文件的方法前面已经说过很多），再打开这个html文件，就会看到“百度的首页”，只是这个首页在你的电脑上，所以你无法进行搜索。

回到我们的查天气程序，我们首先要获取用户输入，拼接成要请求的url地址，并且用requests进行请求。

```
while True:

    city = input('请输入城市,回车退出:\n')

    if not city:

        break

    req = requests.get('http://wthrcdn.etouch.cn/weather_mini?city=%s' % city)

    print(req.text)
```

程序使用while循环以便重复查询，如果没有输入任何内容，程序会自动退出，用了if判断是否存在city。

运行一下看看能不能得到结果。如果提示编码的错误，试试在文件最开始加上：

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

43weather.py - C:\Python37\crossinsample\43weather.py (3.7.3)

File Edit Format Run Options Window Help

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

import requests

while True:
    city = input('请输入城市, 回车退出:\n')
    if not city:
        break

    req = requests.get('http://wthrcdn.etouch.cn/weather_mini?city=%s' % city)
    print(req.text)
```

Ln: 11 Col: 18

Python 3.7.3 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python37\crossinsample\43weather.py =====
请输入城市, 回车退出:
北京
{"data":{"yesterday":{"date":"30日星期二","high":"高温 34℃","fx":"南风","low":"低温 24℃","f1":"<![CDATA[<3级]]>","type":"晴"},"city":"北京","forecast":[{"date":"31日星期三","high":"高温 35℃","fengli":"<![CDATA[<3级]]>","low":"低温 25℃","fengxiang":"东北风","type":"晴"}, {"date":"1日星期四","high":"高温 32℃","fengli":"<![CDATA[<3级]]>","low":"低温 24℃","fengxiang":"东风","type":"中雨"}, {"date":"2日星期五","high":"高温 31℃","fengli":"<![CDATA[<3级]]>","low":"低温 24℃","fengxiang":"东风","type":"雷阵雨"}, {"date":"3日星期六","high":"高温 29℃","fengli":"<![CDATA[<3级]]>","low":"低温 23℃","fengxiang":"东南风","type":"雷阵雨"}, {"date":"4日星期天","high":"高温 31℃","fengli":"<![CDATA[<3级]]>","low":"低温 24℃"}}
```

Ln: 2 Col: 19

可以看到，已经拿到了json格式的天气信息。下一课我们再来处理它。

- [← 43.查天气 \(1\)](#)
- [45.查天气 \(3\) →](#)

【Python 第45课】查天气（3）

看一下我们通过 req.text 已经拿到的json格式的天气预报数据：

```
{
  "data":{
    "yesterday":{"date":"30日星期二",...,
    "city":"北京",
    "forecast":[{"date":"31日星期三",...,
    "ganmao":"各项气象条件适宜...",
    "wendu":"30"
  },
  "status":1000,
  "desc":"OK"
}
```

直接在命令行中看到的应该还是没有换行和空格的一长串字符，这里我把格式整理了一下，并省略了部分信息。可以看出，它像是一个字典的结构，但是有很多层。最外层有三个key--"data", "status"和"desc"，data的value是另一个字典，里面包含了"yesterday"和"forecast"等好几项天气信息，现在我们最关心的就是"forecast"值列中的第一项。

虽然看上去像字典，但它对于程序来说，仍然是一个字符串，只不过是一个满足json格式的字符串。我们可以直接用requests模块的json()方法，将请求得到的json格式的字符串直接转成一个真正的字典。

```
dic_city = req.json()
print(dic_city)
```

可以看到，与直接输出req.text不同，requests的json()方法不仅帮我们把json格式的数据转化成了字典，还一并帮我们把编码的问题解决了。

```
{'data': {'yesterday': {'date': '30日星期二', 'high': '高温 34°C', 'fx': '南风', 'low': '低温 24°C', 'fl': '<![CDATA[<3级]]>', 'type': ...
```

你可能会觉得json格式的字符串和字典是一样的，但如果你用type方法看一下它们的类型：

```
print (type(req.text))
print (type(req.json()))
```

就知道区别在哪里了。

之后的事情就比较容易了。首先使用字典的get方法，若城市名错误返回的字典中没有'data'这个键，程序不会报错，而是返回 None

```
city_data = dic_city.get('data')
```

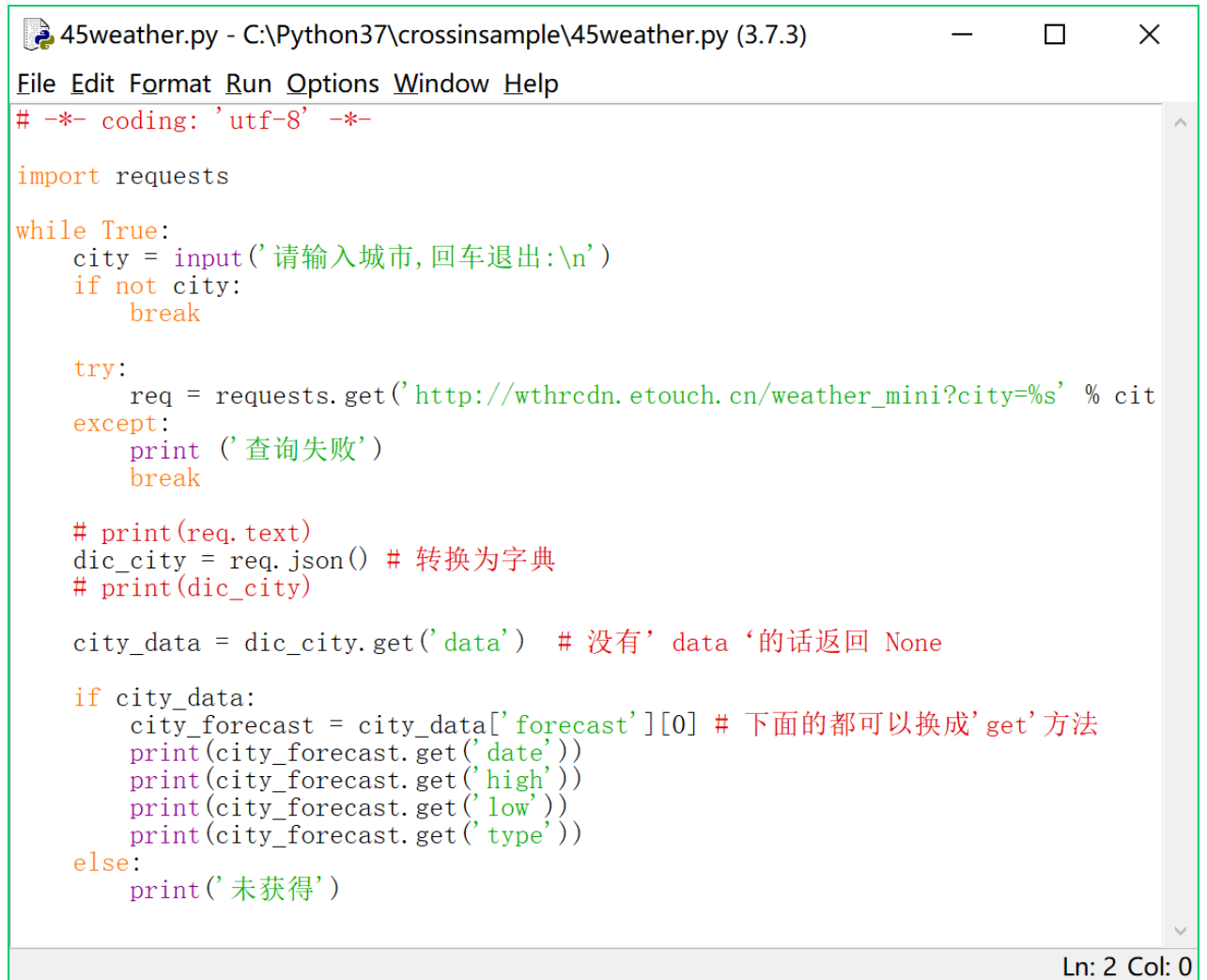
接着，分别输出查询城市当天的日期、最高温、最低温和天气类型，或者进行没有找到城市时的处理：

```
if city_data:
    city_forecast = city_data['forecast'][0] # 下面的都可以换成'get'方法
    print(city_forecast.get('date'))
    print(city_forecast.get('high'))
    print(city_forecast.get('low'))
    print(city_forecast.get('type'))
```

```
else:
    print('未获得')
```

另外，为了防止在请求过程中出错，我加上了一个异常处理。

```
try:
    ###
    ###
except:
    print ('查询失败')
```



```
45weather.py - C:\Python37\crossinsample\45weather.py (3.7.3)
File Edit Format Run Options Window Help
# -*- coding: 'utf-8' -*-

import requests

while True:
    city = input('请输入城市,回车退出:\n')
    if not city:
        break

    try:
        req = requests.get('http://wthrcdn.etouch.cn/weather_mini?city=%s' % city)
    except:
        print ('查询失败')
        break

    # print(req.text)
    dic_city = req.json() # 转换为字典
    # print(dic_city)

    city_data = dic_city.get('data') # 没有' data '的话返回 None

    if city_data:
        city_forecast = city_data['forecast'][0] # 下面的都可以换成'get'方法
        print(city_forecast.get('date'))
        print(city_forecast.get('high'))
        print(city_forecast.get('low'))
        print(city_forecast.get('type'))
    else:
        print('未获得')
```

Ln: 2 Col: 0

```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Python37\crossinsample\45weather.py =====
请输入城市,回车退出:
北京
31日星期三
高温 35°C
低温 25°C
晴
请输入城市,回车退出:
南京
31日星期三
高温 38°C
低温 28°C
多云
请输入城市,回车退出:
Cossin
未获得
请输入城市,回车退出:
>>>
```

Ln: 22 Col: 4

- [← 44.查天气 \(2\)](#)
- [46.面向对象 \(1\) →](#)

【Python 第46课】面向对象（1）

我们之前已经写了不少小程序，都是按照功能需求的顺序来设计程序。这种被称为‘面向过程’的编程。

还有一种程序设计的方法，把数据和对数据的操作用一种叫做“对象”的东西包裹起来。这种被成为“面向对象”的编程。这种方法更适合较大型的程序开发。

面向对象编程最主要的两个概念就是：类（class）和对象（object）

类是一种抽象的类型，而对象是这种类型的实例。

举个现实的例子：

“笔”作为一个抽象的概念，可以被看成是一个类。而一支实实在在的笔，则是“笔”这种类型的对象。

一个类可以有属于它的函数，这种函数被称为类的“方法”。

一个类/对象可以有属于它的变量，这种变量被称作“域”。

域根据所属不同，又分别被称作“类变量”和“实例变量”。

继续笔的例子。一个笔有书写的功能，所以“书写”就是笔这个类的一种方法。

每支笔有自己的颜色，“颜色”就是某支笔的域，也是这支笔的实例变量。

而关于“类变量”，我们假设有一种限量版钢笔，我们为这种笔创建一种类。而这种笔的“产量”就可以看做这种笔的类变量。因为这个域不属于某一支笔，而是这种类型的笔的共有属性。

域和方法被合称为类的属性。

python是一种高度面向对象的语言，它其中的所有东西其实都是对象。所以我们之前也一直在使用着对象。看如下的例子：

```
s = 'how are you'
```

```
#s被赋值后就是一个字符串类型的对象
```

```
l = s.split()
```

```
#split是字符串的方法，这个方法返回一个list类型的对象
```

```
#l是一个list类型的对象
```

通过dir()方法可以查看一个类/变量的所有属性：

```
dir(s)
```

```
dir(list)
```

下节课，我们来自已创建一个类。

- [← 45.查天气 \(3\)](#)
- [47.面向对象 \(2\) →](#)

【Python 第47课】面向对象（2）

昨天介绍了面向对象的概念，今天我们来创建一个类。

```
class MyClass:
```

```
    pass
```

```
mc = MyClass()
```

```
print (mc)
```

关键字class加上类名用来创建一个类。之后缩进的代码块是这个类的内部。在这里，我们用pass语句，表示一个空的代码块。

类名加圆括号()的形式可以创建一个类的实例，也就是被称作对象的东西。我们把这个对象赋值给变量mc。于是，mc现在就是一个MyClass类的对象。

看一下输出结果：

```
<__main__.MyClass instance at 0x7fd1c8d01200>
```

这个意思就是说，mc是__main__模块中MyClass来的一个实例（instance），后面的一串十六进制的数字是这个对象的内存地址。

我们给这个类加上一些域：

```
class MyClass:
```

```
    name = 'Sam'
```

```
    def sayHi(self):
```

```
        print ('Hello %s' % self.name)
```

```
mc = MyClass()
```

```
print (mc.name)
```

```
mc.name = 'Lily'
```

```
mc.sayHi()
```

我们给MyClass类增加了一个类变量name，并把它值设为'Sam'。然后又增加了一个类方法sayHi。

调用类变量的方法是“对象.变量名”。你可以得到它的值，也可以改变它的值。

注意到，类方法和我们之前定义的函数区别在于，第一个参数必须为self。而在调用类方法的时候，通过“对象.方法名()”格式进行调用，而不需要额外提供self这个参数的值。self在类方法中的值，就是你调用的这个对象本身。

输出结果：

```
Sam
```

```
Hello Lily
```

之后，在你需要用到MyClass这种类型对象的地方，就可以创建并使用它。

- [← 46.面向对象（1）](#)
- [48.面向对象（3） →](#)

【Python 第48课】面向对象（3）

面向对象是比较复杂的概念，初学很难理解。我曾经对人夸张地说，面向对象是颠覆你编程三观的东西，得花上不少时间才能搞清楚。我自己当年初学Java的时候，也是折腾了很久才理清点头绪。所以我在前面的课程中没有去提及类和对象这些概念，不想在一开始给大家造成混淆。

在刚开始编程的时候，从上到下一行行执行的简单程序容易被理解，即使加上if、while、for之类的语句以及函数调用，也还是不算困难。有了面向对象之后，程序的执行路径就变得复杂，很容易让人混乱。不过当你熟悉之后会发现，面向对象是比面向过程更合理的程序设计方式。

今天我用一个例子来展示两种程序设计方式的不同。

假设我们有一辆汽车，我们知道它的速度(60km/h)，以及A、B两地的距离(100km)。要算出开着这辆车从A地到B地花费的时间。（很像小学数学题是吧？）

面向过程的方法：

```
speed = 60.0

distance = 100.0

time = distance / speed

print (time)
```

面向对象的方法：

```
class Car:

    speed = 0

    def drive(self, distance):

        time = distance / self.speed

        print (time)

car = Car()

car.speed = 60.0

car.drive(100.0)
```

看上去似乎面向对象没有比面向过程更简单，反而写了更多行代码。

但是，如果我们让题目再稍稍复杂一点。假设我们又有了一辆更好的跑车，它的速度是150km/h，然后我们除了想从A到B，还要从B到C（距离200km）。要求分别知道这两种车在这两段路上需要多少时间。

面向过程的方法：

```
speed1 = 60.0

distance1 = 100.0

time1 = distance1 / speed1

print (time1)

distance2 = 200.0

time2 = distance2 / speed1

print (time2)

speed2 = 150.0
```

```
time3 = distance1 / speed2
```

```
print (time3)
```

```
time4 = distance2 / speed2
```

```
print (time4)
```

面向对象的方法：

```
class Car:
```

```
    speed = 0
```

```
    def drive(self, distance):
```

```
        time = distance / self.speed
```

```
        print (time)
```

```
car1 = Car()
```

```
car1.speed = 60.0
```

```
car1.drive(100.0)
```

```
car1.drive(200.0)
```

```
car2 = Car()
```

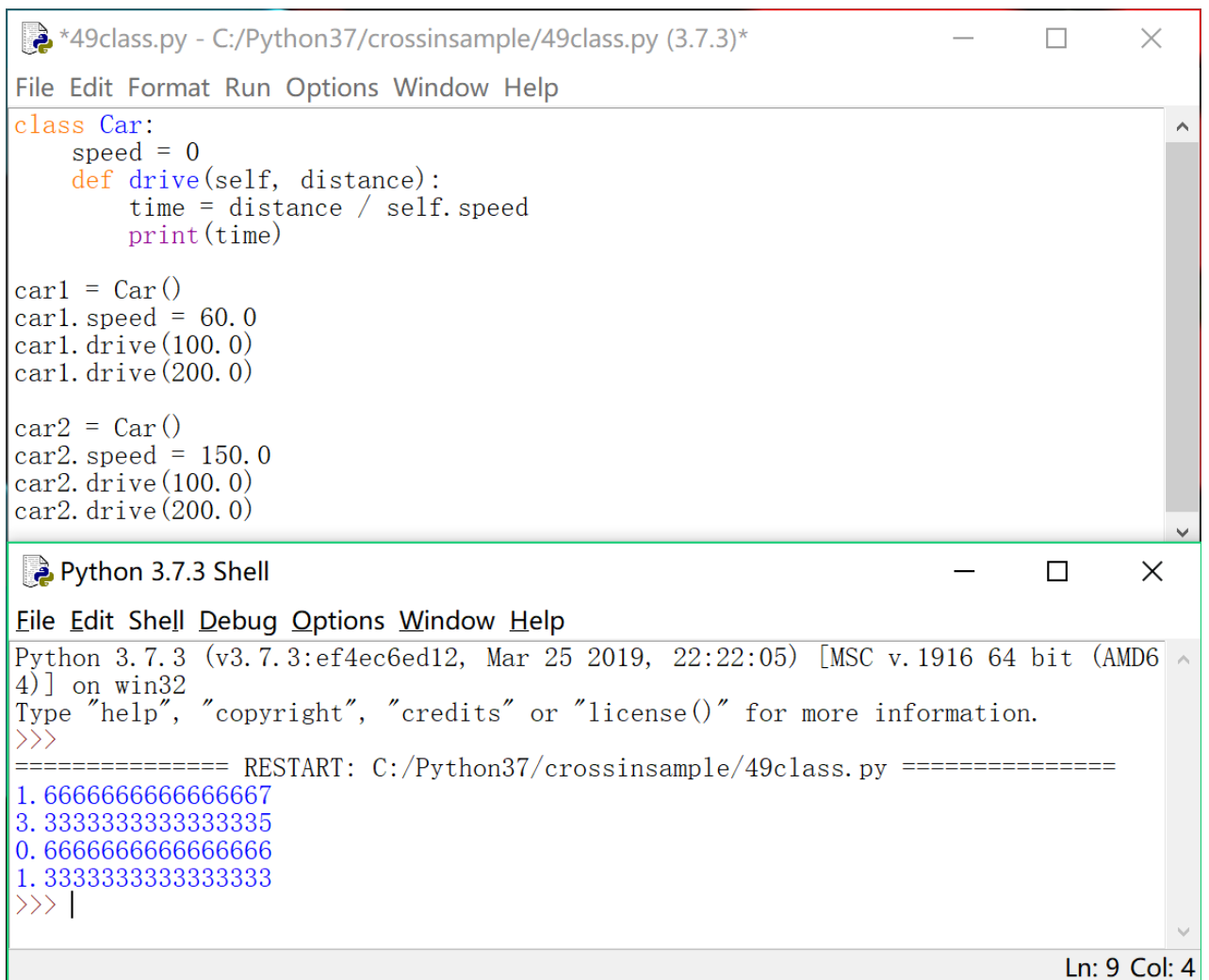
```
car2.speed = 150.0
```

```
car2.drive(100.0)
```

```
car2.drive(200.0)
```

对比两种方法，面向过程把数据和处理数据的计算全部放在一起，当功能复杂之后，就会显得很混乱，且容易产生很多重复的代码。而面向对象，把一类数据和处理这类数据的方法封装在一个类中，让程序的结构更清晰，不同的功能之间相互独立。这样更有利于进行模块化的开发方式。

面向对象的水还很深，我们这里只是粗略一瞥。它不再像之前print、while这些概念那么一目了然。但也没必要对此畏惧，等用多了自然就熟悉了。找一些实例亲手练练，会掌握得更快。遇到问题时，欢迎来论坛和群里讨论。



The image shows a screenshot of a Python IDE with two windows. The top window, titled '*49class.py - C:/Python37/crossinsample/49class.py (3.7.3)*', contains the following Python code:

```
class Car:
    speed = 0
    def drive(self, distance):
        time = distance / self.speed
        print(time)

car1 = Car()
car1.speed = 60.0
car1.drive(100.0)
car1.drive(200.0)

car2 = Car()
car2.speed = 150.0
car2.drive(100.0)
car2.drive(200.0)
```

The bottom window, titled 'Python 3.7.3 Shell', shows the output of running the code:

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Python37/crossinsample/49class.py =====
1. 6666666666666667
3. 3333333333333335
0. 6666666666666666
1. 3333333333333333
>>> |
```

The status bar at the bottom right of the shell window indicates 'Ln: 9 Col: 4'.

- [← 47.面向对象 \(2\)](#)
- [49.面向对象 \(4\) →](#)

【Python 第49课】面向对象（4）

上一课举了一个面向对象和面向过程相比较的例子之后，有些同学表示，仍然没太看出面向对象的优点。没关系，那是因为我们现在接触的程序还不够复杂，等以后你写的程序越来越大，就能体会到这其中的差别了。

今天我们就来举一个稍稍再复杂一点的例子。

仍然是从A地到B地，这次除了有汽车，我们还有一辆自行车！

自行车和汽车有着相同的属性：速度（speed）。还有一个相同的方法（drive），来输出行驶/骑行一段距离所花的时间。但这次我们要给汽车增加一个属性：每公里油耗（fuel）。而在汽车行驶一段距离的方法中，除了要输出所花的时间外，还要输出所需的油量。

面向过程的方法，你可能需要写两个函数，然后把数据作为参数传递进去，在调用的时候要搞清应该使用哪个函数和哪些数据。有了面向对象，你可以把相关的数据和方法封装在一起，并且可以把不同类中的相同功能整合起来。这就需要用到面向对象中的另一个重要概念：继承。

我们要使用的方法是，创建一个叫做Vehicle的类，表示某种车，它包含了汽车和自行车所共有的东西：速度，行驶的方法。然后让Car类和Bike类都继承这个Vehicle类，即作为它的子类。在每个子类中，可以分别添加各自独有的属性。

Vehicle类被称为基本类或超类，Car类和Bike类被成为导出类或子类。

```
class Vehicle:

    def __init__(self, speed):

        self.speed = speed

    def drive(self, distance):

        print ('need %f hour(s)' % (distance / self.speed))

class Bike(Vehicle):

    pass

class Car(Vehicle):

    def __init__(self, speed, fuel):

        Vehicle.__init__(self, speed)

        self.fuel = fuel

    def drive(self, distance):

        Vehicle.drive(self, distance)

        print ('need %f fuels' % (distance * self.fuel))

b = Bike(15.0)

c = Car(80.0, 0.012)

b.drive(100.0)

c.drive(100.0)
```

解释一下代码：

`__init__` 函数会在类被创建的时候自动调用，用来初始化类。它的参数，要在创建类的时候提供。于是我们通过提

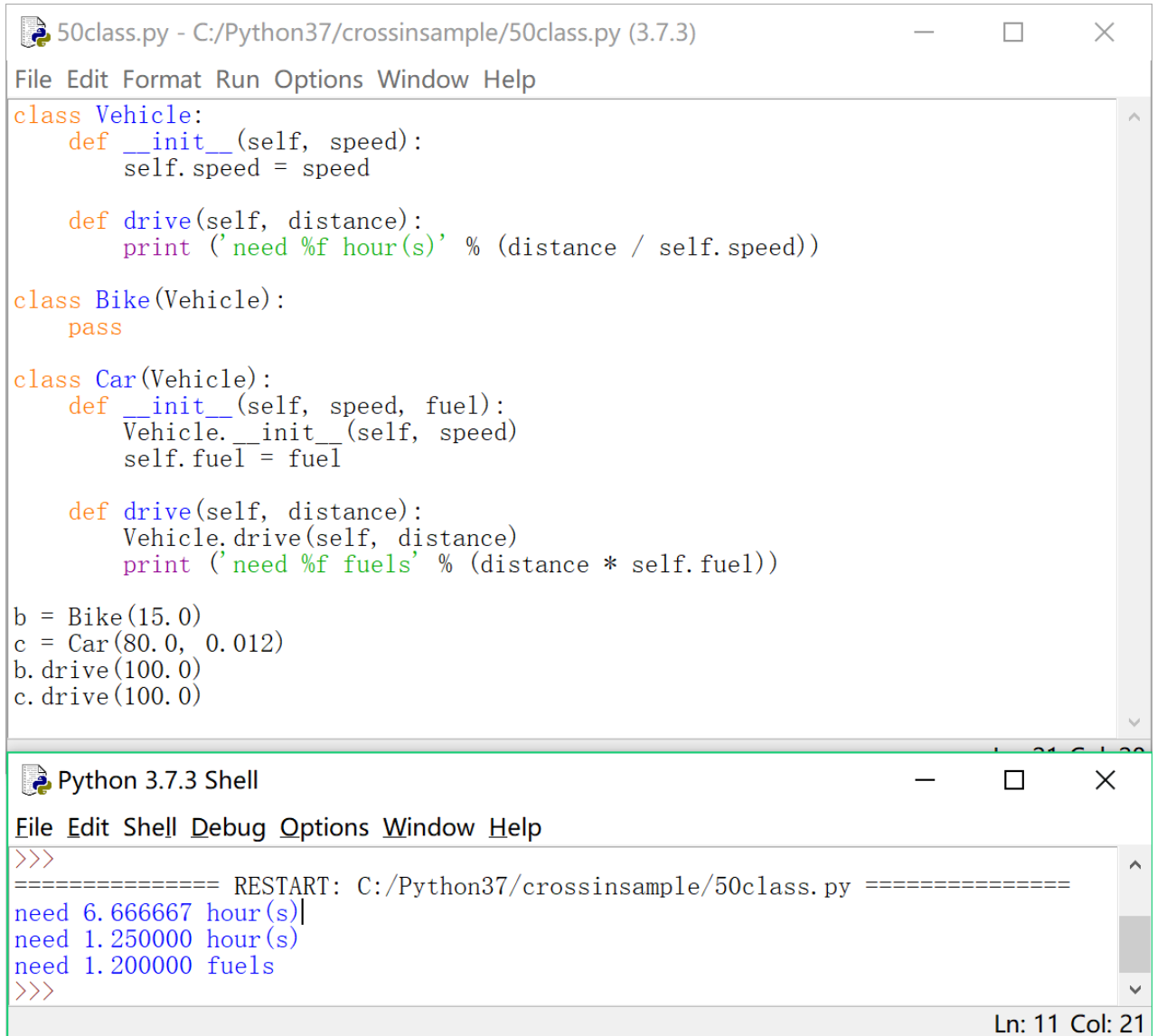
提供一个数值来初始化speed的值。

注意：__init__是python的内置方法，类似的函数名前后是两个下英文划线，如果写错了，则不会起到原本应有的作用。

class定义后面的括号里表示这个类继承于哪个类。Bike(Vehicle)就是说Bike是继承自Vehicle中的子类。Vehicle中的属性和方法，Bike都会有。因为Bike不需要有额外的功能，所以用pass在类中保留空块，什么都不用写。

Car类中，我们又重新定义了__init__和drive函数，这样会覆盖掉它继承自Vehicle的同名函数。但我们依然可以通过“Vehicle.函数名”来调用它的超类方法。以此来获得它作为Vehicle所具有的功能。注意，因为是通过类名调用方法，而不是像之前一样通过对象来调用，所以这里必须提供self的参数值。在调用超类的方法之后，我们又给Car增加了一个fuel属性，并且在drive中多输出一行信息。

最后，我们分别创建一个速度为15的自行车对象，和一个速度为80、耗油量为0.012的汽车，然后让它们去行驶100的距离。



The screenshot shows a Python IDE window titled "50class.py - C:/Python37/crossinsample/50class.py (3.7.3)". The code defines three classes: Vehicle, Bike, and Car. Vehicle has an __init__ method that sets speed and a drive method that prints the time needed. Bike inherits from Vehicle and uses pass. Car inherits from Vehicle, overrides __init__ to also set fuel, and overrides drive to also print fuel consumption. Below the class definitions, three objects are created: b (Bike with speed 15.0), c (Car with speed 80.0 and fuel 0.012), and their drive methods are called for a distance of 100.0. Below the code editor is a "Python 3.7.3 Shell" window showing the execution output: a restart message, followed by the printed results for each object's drive method call. The status bar at the bottom right indicates "Ln: 11 Col: 21".

```
class Vehicle:
    def __init__(self, speed):
        self.speed = speed

    def drive(self, distance):
        print('need %f hour(s)' % (distance / self.speed))

class Bike(Vehicle):
    pass

class Car(Vehicle):
    def __init__(self, speed, fuel):
        Vehicle.__init__(self, speed)
        self.fuel = fuel

    def drive(self, distance):
        Vehicle.drive(self, distance)
        print('need %f fuels' % (distance * self.fuel))

b = Bike(15.0)
c = Car(80.0, 0.012)
b.drive(100.0)
c.drive(100.0)
```

```
>>>
===== RESTART: C:/Python37/crossinsample/50class.py =====
need 6.666667 hour(s)|
need 1.250000 hour(s)
need 1.200000 fuels
>>>
```

Ln: 11 Col: 21

- [← 48.面向对象 \(3\)](#)
- [50.and-or技巧 →](#)

【Python 第50课】and-or技巧

今天介绍一个python中的小技巧：and-or

看下面这段代码：

```
a = "heaven"
b = "hell"
c = True and a or b
print (c)

d = False and a or b
print (d)
```

输出：

heaven

hell

结果很奇怪是不是？

表达式从左往右运算，1和"heaven"做and的结果是"heaven"，再与"hell"做or的结果是"heaven"；0和"heaven"做and的结果是0，再与"hell"做or的结果是"hell"。

抛开绕人的and和or的逻辑，你只需记住，在一个bool and a or b语句中，当bool条件为真时，结果是a；当bool条件为假时，结果是b。

有学过c/c++的同学应该会发现，这和bool?a:b表达式很像。

有了它，原本需要一个if-else语句表述的逻辑：

```
if a > 0:
    print ("big")
else:
    print ("small")
```

就可以直接写成：

```
print ((a > 0) and "big" or "small")
```

然而不幸的是，如果直接这么用，有一天你会踩到坑的。和c语言中的?:表达式不同，这里的and or语句是利用了python中的逻辑运算实现的。当a本身是个假值（如0, ""）时，结果就不会像你期望的那样。

比如：

```
a = ""
b = "hell"
c = True and a or b
print (c)
```

得到的结果不是""而是"hell"。因为""和"hell"做or的结果是"hell"。

所以，and-or真正的技巧在于，确保a的值不会为假。最常用的方式是使a成为[a]、b成为[b]，然后使用返回列表的第一个元素：

```
a = ""
b = "hell"
c = (True and [a] or [b])[0]
```



```
print (c)
```

由于[a]是一个非空列表，所以它决不会为假。即使a是0或者"或者其它假值，列表[a]也为真，因为它有一个元素。

在两个常量值进行选择时，and-or会让你的代码更简单。但如果你觉得这个技巧带来的副作用已经让你头大了，没关系，用if-else可以做相同的事情。不过在python的某些情况下，你可能没法使用if语句，比如lambda函数中，这时候你可能就需要and-or的帮助了。

什么是lambda函数？呵呵，这是python的高阶玩法，暂且按住不表，以后有机会再说。

- [← 49.面向对象 \(4\)](#)
- [51.元组 →](#)

【Python 第51课】元组

上一次pygame的课中有这样一行代码：

```
x, y = pygame.mouse.get_pos()
```

这个函数返回的其实是一个“元组”，今天我们来讲讲这个东西。

元组（tuple）也是一种序列，和我们用了很多次的list类似，只是元组中的元素在创建之后就不能被修改。

如：

```
postion = (1, 2)

geeks = ('Sheldon', 'Leonard', 'Rajesh', 'Howard')
```

都是元组的实例。它有和list同样的索引、切片、遍历等操作（参见25~27课）：

```
print (postion[0])

for g in geeks:

    print (g)

print (geeks[1:3])
```

其实我们之前一直在用元组，就是在print语句中：

```
print ('%s is %d years old' % ('Mike', 23))
```

('Mike', 23)就是一个元组。这是元组最常见的用处。

再来看一下元组作为函数返回值的例子：

```
def get_pos(n):

    return (n/2, n*2)
```

得到这个函数的返回值有两种形式，一种是根据返回值元组中元素的个数提供变量：

```
x, y = get_pos(50)

print (x)

print (y)
```

这就是我们在开头那句代码中使用的方式。

还有一种方法是用一个变量记录返回的元组：

```
pos = get_pos(50)

print (pos[0])

print (pos[1])
```

- [← 50.and-or技巧](#)
- [52.数学运算 →](#)

【Python 第52课】数学运算

今天从打飞机游戏里中断一下，说些python的基础。

在用计算机编程解决问题的过程中，数学运算是很常用的。python自带了一些基本的数学运算方法，这节课给大家介绍一二。

python的数学运算模块叫做math，再用之前，你需要

```
import math
```

math包里有两个常量：

```
math.pi # 圆周率π: 3.141592...
```

```
math.e # 自然常数: 2.718281...
```

数值运算：

```
math.ceil(x)
```

```
# 对x向上取整，比如x=1.2，返回2.0 (py3返回2)
```

```
math.floor(x)
```

```
# 对x向下取整，比如x=1.2，返回1.0 (py3返回1)
```

```
math.pow(x, y)
```

```
# 指数运算，得到x的y次方
```

```
math.log(x)
```

```
# 对数，默认基底为e。可以使用第二个参数，来改变对数的基底。比如math.log(100, 10)
```

```
math.sqrt(x)
```

```
# 平方根
```

```
math.fabs(x)
```

```
# 绝对值
```

三角函数：

```
math.sin(x)
```

```
math.cos(x)
```

```
math.tan(x)
```

```
math.asin(x)
```

```
math.acos(x)
```

```
math.atan(x)
```

注意：这里的x是以弧度为单位，所以计算角度的话，需要先换算

角度和弧度互换：

```
math.degrees(x)
```

弧度转角度

```
math.radians(x)
```

角度转弧度

以上是你平常可能会用到的函数。除此之外，还有一些，这里就不罗列，可以去

<https://docs.python.org/3.7/library/math.html>

查看官方的完整文档。

有了这些函数，可以更方便的实现程序中的计算。比如中学时代算了无数次的

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

现在你就可以写一个函数，输入一元二次方程的a、b、c系数，直接给你数值解。好，这题就留作课后作业吧。

晚上有事，所以今天的课比平常来得早一些。以后我还会不定期地介绍python中的模块，例如random（随机数）、re（正则表达式）、time（时间）、urllib.request（网络请求）等等。

- [← 51.元组](#)
- [53.真值表 →](#)

【Python 第53课】真值表

逻辑判断是编程中极为常用的知识。之前的课我们已经说过，见第6课和第11课。但鉴于逻辑运算的重要性，今天我再把常用的运算结果总结一下，供大家参考。

这种被称为“真值表”的东西，罗列了基本逻辑运算的结果。你不一定要全背下来，但应该对运算的规律有所了解。

为了便于看清，我用 \Leftrightarrow 来表示等价关系。

\Leftarrow 左边表示逻辑表达式， \Rightarrow 右边表示它的结果。

NOT

`not False` \Leftrightarrow True

`not True` \Leftrightarrow False

（not的结果与原值相反）

OR

`True or False` \Leftrightarrow True

`True or True` \Leftrightarrow True

`False or True` \Leftrightarrow True

`False or False` \Leftrightarrow False

（只要有一个值为True，or的结果就是True）

AND

`True and False` \Leftrightarrow False

`True and True` \Leftrightarrow True

`False and True` \Leftrightarrow False

`False and False` \Leftrightarrow False

（只要有一个值为False，and的结果就是False）

NOT OR

`not (True or False)` \Leftrightarrow False

`not (True or True)` \Leftrightarrow False

`not (False or True)` \Leftrightarrow False

`not (False or False)` \Leftrightarrow True

NOT AND

`not (True and False)` \Leftrightarrow True

`not (True and True)` \Leftrightarrow False

`not (False and True)` \Leftrightarrow True

`not (False and False)` \Leftrightarrow True

!=

`1 != 0` \Leftrightarrow True

$1 \neq 1 \Leftrightarrow \text{False}$

$0 \neq 1 \Leftrightarrow \text{True}$

$0 \neq 0 \Leftrightarrow \text{False}$

$=$

$1 = 0 \Leftrightarrow \text{False}$

$1 = 1 \Leftrightarrow \text{True}$

$0 = 1 \Leftrightarrow \text{False}$

$0 = 0 \Leftrightarrow \text{True}$

以上就是基本的逻辑运算，你会在编程中反复用到它们。就算刚开始搞不清也没关系，多写几段代码就会熟悉了。

另外再说一下，我们论坛上开了一个叫做“编程擂台”新版块，目的是给已经掌握一定编程基础的同学提供一些练手的选择，巩固所学知识，可以学以致用。欢迎大家来这里把想要尝试的项目写出来，或者去挑战别人放出的项目。现在除了我自己抛砖引玉放的两个小程序外，还有另外两位同学发布了自己已经完成或正在尝试的程序--一个抓取“糗事百科”更新的小程序和一个“单词听写软件”。期待看到更多的程序。无论大小，无论难易，敢于尝试就是迈出了第一步。否则，光靠看教程是没法真正学会编程的。

- [← 52.数学运算](#)
- [54.正则表达式\(1\) →](#)

【Python 第54课】正则表达式（1）

今天来挖个新坑，讲讲正则表达式。

什么是正则表达式？在回答这个问题之前，先来看看为什么要有正则表达式。

在编程处理文本的过程中，经常会需要按照某种规则去查找一些特定的字符串。比如知道一个网页上的图片都是叫做'image/8554278135.jpg'之类的名字，只是那串数字不一样；又或者在一堆人员电子档案中，你要把他们的电话号码全部找出来，整理成通讯录。诸如此类工作，如果手工去做，当量大的时候那简直就是悲剧。但你知道这些字符信息有一定的规律，可不可以利用这些规律，让程序自动来做这些无聊的事情？答案是肯定的。这时候，你就需要一种描述这些规律的方法，正则表达式就是干这事的。

正则表达式就是记录文本规则的代码。

所以正则表达式并不是python中特有的功能，它是一种通用的方法。python中的正则表达式库，所做的事情是利用正则表达式来搜索文本。要使用它，你必须会自己用正则表达式来描述文本规则。之前多次有同学表示查找文本的事情经常会遇上，希望能介绍一下正则表达式。既然如此，我们就从正则表达式的基本规则开始说起。

1.

首先说一种最简单的正则表达式，它没有特殊的符号，只有基本的字母或数字。它满足的匹配规则就是完全匹配。例如：有个正则表达式是“hi”，那么它就可以匹配出文本中所有含有hi的字符。

来看如下的一段文字：

Hi, I am Shirley Hilton. I am his wife.

如果我们用“hi”这个正则表达式去匹配这段文字，将会得到两个结果。因为是完全匹配，所以每个结果都是“hi”。这两个“hi”分别来自“Shirley”和“his”。默认情况下正则表达式是严格区分大小写的，所以“Hi”和“Hilton”中的“H”被忽略了。

为了验证正则表达式匹配的结果，你可以用以下这段代码做实验：

```
import re

text = "Hi, I am Shirley Hilton. I am his wife."

m = re.findall(r"hi", text)

if m:

    print (m)

else:

    print ('not match')
```

暂时先不解释这其中代码的具体含义，你只要去更改text和findall中的字符串，就可以用它来检测正则表达式的实际效果。

2.

如果我们只想找到“hi”这个单词，而不把包含它的单词也算在内，那就可以使用“\bhi\b”这个正则表达式。在以前的字符串处理中，我们已经见过类似“\n”这种特殊字符。在正则表达式中，这种字符更多，以后足以让你眼花缭乱。

“\b”在正则表达式中表示单词的开头或结尾，空格、标点、换行都算是单词的分割。而“\b”自身又不会匹配任何字符，它代表的只是一个位置。所以单词前后的空格标点之类不会出现在结果里。

在前面那个例子里，“\bhi\b”匹配不到任何结果。但“\bhi”的话就可以匹配到1个“hi”，出自“his”。用这种方法，你可以找出一段话中所有单词“Hi”，想一下要怎么写。

3.

最后再说一下[]这个符号。在正则表达式中，[]表示满足括号中任一字符。比如“[hi]”，它就不是匹配“hi”了，而是匹配“h”或者“i”。

在前面例子中，如果把正则表达式改为“[Hh]i”，就可以既匹配“Hi”，又匹配“hi”了。

- [← 53.真值表](#)
- [55.正则表达式 \(2\) →](#)

【Python 第55课】正则表达式（2）

有同学问起昨天那段测试代码里的问题，我来简单说一下。

1.

```
r"hi"
```

这里字符串前面加了r，是raw的意思，它表示对字符串不进行转义。为什么要加这个？你可以试试print "bhi"和r"bhi"的区别。

```
>>> print ("\bhi")
```

```
hi
```

```
>>> print (r"\bhi")
```

```
\bhi
```

可以看到，不加r的话，\b就没有了。因为python的字符串碰到“\”就会转义它后面的字符。如果你想在字符串里打“\”，则必须要打“\\”。

```
>>> print ("\\bhi")
```

```
\bhi
```

这样的话，我们的正则表达式里就会多出很多“\\”，让本来就已经复杂的字符串混乱得像五仁月饼一般。但加上了“r”，就表示不要去转义字符串中的任何字符，保持它的原样。

2.

```
re.findall(r"hi", text)
```

re是python里的正则表达式模块。findall是其中一个方法，用来按照提供的正则表达式，去匹配文本中的所有符合条件的字符串。返回结果是一个包含所有匹配的list。

3.

今天主要说两个符号“.”和“*”，顺带说下“S”和“?”。

“.”在正则表达式中表示除换行符以外的任意字符。在上节课提供的那段例子文本中：

```
Hi, I am Shirley Hilton. I am his wife.
```

如果我们用“i.”去匹配，就会得到

```
['i', 'ir', 'il', 'is', 'if']
```

你若是暴力一点，也可以直接用“.”去匹配，看看会得到什么。

与“.”类似的一个符号是“S”，它表示的是不是空白符的任意字符。注意是大写字符S。

4.

在很多搜索中，会用“?”表示任意一个字符，“*”表示任意数量连续字符，这种被称为通配符。但在正则表达式中，任意字符是用“.”表示，而“*”则不是表示字符，而是表示数量：它表示前面的字符可以重复任意多次（包括0次），只要满足这样的条件，都会被表达式匹配上。

结合前面的“.*”，用“I.*e”去匹配，想一下会得到什么结果？

```
['I am Shirley Hilton. I am his wife']
```

是不是跟你想的有些不一样？也许你会以为是

```
['I am Shirle', 'I am his wife']
```

这是因为“*”在匹配时，会匹配尽可能长的结果。如果你想让他匹配到最短的就停止，需要用“.*?”。如“I.*?e”，就会得到第二种结果。这种匹配方式被称为懒惰匹配，而原本尽可能长的方式被称为贪婪匹配。

最后留一道习题：

从下面一段文本中，匹配出所有s开头，e结尾的单词。

site sea sue sweet see case sse ssee loses

- [← 54.正则表达式 \(1\)](#)
- [56.正则表达式 \(3\) →](#)

【Python 第56课】正则表达式（3）

先来公布上一课习题的答案：

```
\bs\S*e\b
```

有的同学给出的答案是"\bs.*?e\b"。测试一下就会发现，有奇怪的'sea sue'和'sweet see'混进来了。既然是单词，我们就不要空格，所以需要"\S"而不是"."

昨天有位同学在论坛上说，用正则表达式匹配出了文件中的手机号。这样现学现用很不错。匹配的规则是"1.*?\n"，在这个文件的条件下，是可行的。但这规则不够严格，且依赖于手机号结尾有换行符。今天我来讲讲其他的方法。

匹配手机号，其实就是找出一串连续的数字。更进一步，是11位，以1开头的数字。

还记得正则第1讲里提到的[]符号吗？它表示其中任意一个字符。所以要匹配数字，我们可以用

```
[0123456789]
```

由于它们是连续的字符，有一种简化的写法：[0-9]。类似的还有[a-zA-Z]的用法。

还有另一种表示数字的方法：

```
\d
```

要表示任意长度的数字，就可以用

```
[0-9]*
```

或者

```
\d*
```

但要注意的是，*表示的任意长度包括0，也就是没有数字的空字符也会被匹配出来。一个与*类似的符号+，表示的则是1个或更长。

所以要匹配出所有的数字串，应当用

```
[0-9]+
```

或者

```
\d+
```

如果要限定长度，就用{}代替+，大括号里写上你想要的长度。比如11位的数字：

```
\d{11}
```

想要再把第一位限定为1，就在前面加上1，后面去掉一位：

```
1\d{10}
```

OK. 总结一下今天提到的符号：

```
[0-9]
```

```
\d
```

```
+
```

```
{}
```

现在你可以去一个混杂着各种数据的文件里，抓出里面的手机号，或是其他你感兴趣的数字了。

- [← 55.正则表达式（2）](#)
- [57.正则表达式（4） →](#)

【Python 第57课】正则表达式（4）

1.

我们已经了解了正则表达式中的一些特殊符号，如\b、\d、.、\S等等。这些具有特殊意义的专用字符被称作“元字符”。常用的元字符还有：

\w - 匹配字母或数字或下划线或汉字（我试验下了，发现python 3.x版本可以匹配汉字，但2.x版本不可以）

\s - 匹配任意的空白符

^ - 匹配字符串的开始

\$ - 匹配字符串的结束

2.

\S其实就是\s的反义，任意不是空白符的字符。同理，还有：

\W - 匹配任意不是字母，数字，下划线，汉字的字符

\D - 匹配任意非数字的字符

\B - 匹配不是单词开头或结束的位置

[a]的反义是[^a]，表示除a以外的任意字符。[^abcd]就是除abcd以外的任意字符。

3.

之前我们用过*、+、{}来表示字符的重复。其他重复的方式还有：

? - 重复零次或一次

{n} - 重复n次或更多次

{n,m} - 重复n到m次

正则表达式不只是用来从一大段文字中抓取信息，很多时候也被用来判断输入的文本是否符合规范，或进行分类。来点例子看看：

^\w{4,12}\$

这个表示一段4到12位的字符，包括字母或数字或下划线或汉字，可以用来作为用户注册时检测用户名的规则。（但汉字在python2.x里面可能会有问题）

\d{15,18}

表示15到18位的数字，可以用来检测身份证号码

^1\d*x?

以1开头的一串数字，数字结尾有字母x，也可以没有。有的话就带上x。

另外再说一下之前提到的转义字符\。如果我们确实要匹配.或者*字符本身，而不是要它们所代表的元字符，那就需要用\或*。本身也需要用\。

比如"d+\\.d+"可以匹配出123.456这样的结果。

留一道稍稍有难度的习题：

写一个正则表达式，能匹配出多种格式的电话号码，包括

(021)88776543

010-55667890

02584453362

0571 66345673

- [← 56.正则表达式 \(3\)](#)
- [58.正则表达式 \(5\) →](#)

【Python 第58课】正则表达式（5）

听说有人已经开始国庆假期了，甚至还有人中秋之后就请了年假一休到底，表示羡慕嫉妒恨！今天发完这课，我也要进入休假状态，谁也别拦着我。

来说上次的习题：

(021)88776543

010-55667890

02584453362

0571 66345673

一个可以匹配出所有结果的表达式是

`\(?:0\d{2,3}[D-]?\d{7,8}`

解释一下：

`\(?`

`()`在正则表达式里也有着特殊的含义，所以要匹配字符`"("`，需要用`"\""`。`?`表示这个括号是可有可无的。

`0\d{2,3}`

区号，`0xx`或者`0xxx`

`[D-]?`

在区号之后跟着的可能是`)`、`"`、`-`，也可能什么也没有。

`\d{7,8}`

7或8位的电话号码

可是，这个表达式虽然能匹配出所有正确的数据（一般情况下，这样已经足够），但理论上也会匹配到错误的数。因为`()`应当是成对出现的，表达式中对于左右两个括号并没有做关联处理，例如`(02188776543)`这样的数据也是符合条件的。

我们可以用正则表达式中的`"|"`符号解决这种问题。`"|"`相当于python中`"or"`的作用，它连接的两个表达式，只要满足其中之一，就会被算作匹配成功。

于是我们可以把`()`的情况单独分离出来：

`\(0\d{2,3})\d{7,8}`

其他情况：

`0\d{2,3}[D-]?\d{7,8}`

合并：

`\(0\d{2,3})\d{7,8}|0\d{2,3}[D-]?\d{7,8}`

使用`"|"`时，要特别提醒注意的是不同条件之间的顺序。匹配时，会按照从左往右的顺序，一旦匹配成功就停止验证后面的规则。假设要匹配的电话号码还有可能是任意长度的数字（如一些特殊的服务号码），你应该把

`|d+`

这个条件加在表达式的最后。如果放在最前面，某些数据就可能会被优先匹配为这一条件。你可以写个测试用例体会一下两种结果的不同。

关于正则表达式，我们已经讲了5篇，介绍了正则表达式最最皮毛的一些用法。接下来，这个话题要稍稍告一段落。推荐一篇叫做《正则表达式30分钟入门教程》的文章（直接百度一下就能找到，我也会转到论坛上），想要对正则表达式进一步学习的学生可以参考。这篇教程是个标题党，里面涉及了正则表达式较多的内容，30分钟绝

对看不完。

好了，祝大家过个欢脱的长假，好好休息，多陪家人。

- [← 57.正则表达式 \(4\)](#)
- [59.随机数 →](#)

【Python 第59课】随机数

有些时日没发新课了，今天来说一说python中的random模块。

random模块的作用是产生随机数。之前的小游戏中用到过random中的randint:

```
import random

num = random.randint(1,100)
```

random.randint(a, b)可以生成一个a到b间的随机整数，包括a和b。

a、b都必须是整数，且必须 $b \geq a$ 。当等于的时候，比如：

```
random.randint(3, 3)
```

的结果就永远是3

除了randint，random模块中比较常用的方法还有：

```
random.random()
```

生成一个0到1之间的随机浮点数，包括0但不包括1，也就是[0.0, 1.0)。

```
random.uniform(a, b)
```

生成a、b之间的随机浮点数。不过与randint不同的是，a、b无需是整数，也不用考虑大小。

```
random.uniform(1.5, 3)
```

```
random.uniform(3, 1.5)
```

这两种参数都是可行的

```
random.uniform(1.5, 1.5)
```

永远得到1.5

```
random.choice(seq)
```

从序列中随机选取一个元素。seq需要是一个序列，比如list、元组、字符串。

```
random.choice([1, 2, 3, 5, 8, 13]) #list
```

```
random.choice('hello') #字符串
```

```
random.choice(['hello', 'world']) #字符串组成的list
```

```
random.choice((1, 2, 3)) #元组
```

都是可行的用法。

```
random.randrange(start, stop, step)
```

生成一个从start到stop（不包括stop），间隔为step的一个随机数。start、stop、step都要为整数，且 $start < stop$ 。

比如：

```
random.randrange(1, 9, 2)
```

就是从[1, 3, 5, 7]中随机选取一个。

start和step都可以不提供参数，默认是从0开始，间隔为1。但如果需要指定step，则必须指定start。

```
random.randrange(4) # [0, 1, 2, 3]
```

```
random.randrange(1, 4) # [1, 2, 3]
```


下面这两种方式在效果上等同

```
random.randrange(start, stop, step)
```

```
random.choice(range(start, stop, step))
```

```
random.sample(population, k)
```

从population序列中，随机获取k个元素，生成一个新序列。sample不改变原来序列。

```
random.shuffle(x)
```

把序列x中的元素顺序打乱。shuffle直接改变原有的序列。

以上是random中常见的几个方法。如果你在程序中需要其中某一个方法，也可以这样写：

```
from random import randint
```

```
randint(1, 10)
```

另外，有些编程基础的同学可能知道，在随机数中有个seed的概念，需要一个真实的随机数，比如此刻的时间、鼠标的位置等等，以此为基础产生伪随机数。在python中，默认用系统时间作为seed。你也可以手动调用randomseed(x)来指定seed。

- [← 58.正则表达式 \(5\)](#)
- [60.计时 →](#)

【Python 第60课】计时

Python中有一个time模块，它提供了一些与时间相关的方法。利用time，可以简单地计算出程序运行的时间。对于一些比较复杂、耗时较多的程序，可以通过这种方法了解程序中哪里是效率的瓶颈，从而有针对性地进行优化。

在计算机领域有一个特殊的时间，叫做epoch，它表示的时间是1970-01-01 00:00:00 UTC。

Python中time模块的一个方法

```
time.time()
```

返回的就是从epoch到当前的秒数（不考虑闰秒）。这个值被称为unix时间戳。

于是我们可以用这个方法得到程序开始和结束所用的时间，进而算出运行的时间：

```
import time

starttime = time.time()

print ('start:%f' % starttime)

for i in range(10):

    print (i)

endtime = time.time()

print ('end:%f' % endtime)

print ('total time:%f' % (endtime-starttime))
```

在程序中的不同位置调用time.time()就可以得到运行到那个地方的时间，了解不同部分消耗的时间。

有了这个方法，我们还可以在Pygame课程中的打飞机游戏里，得到每一次游戏主循环刷新的时间，计算出游戏的每秒帧数，显示在屏幕上。

顺便再说下time中的另一个很有用的方法：

```
time.sleep(secs)
```

它可以让程序暂停secs秒。例如：

```
import time

print (1)

time.sleep(3)

print (2)
```

在抓取网页的时候，适当让程序sleep一下，可以减少短时间内的请求，提高请求的成功率。

- [← 59.随机数](#)
- [61.调试程序 →](#)

【Python 第61课】调试程序

写代码，不可避免地会出现bug。很多人在初学编程的时候，当写完程序运行时，发现结果与自己预料中的不同，或者程序意外中止了，就一时没了想法，不知道该从何下手，只能反复重新运行程序，期待忽然有次结果就对了。

今天我就来讲讲代码遇到问题时的一些简单处理方法。

1. 读错误信息

来看如下一个例程：

```
import random

a = 0

for i in range(5):

    b = random.choice(range(5))

    a += i / b

print (a)
```

这个程序中，i从0循环到4，每次循环中，b是0到4中的一个随机数。把i/b的结果累加到a上，最后输出结果。

运行这段程序，有时候会输出结果，有时候却跳出错误信息：

Traceback (most recent call last):

File "C:\Users\Crossin\Desktop\py\test.py", line 5, in <module>

a += i / b

ZeroDivisionError: integer division or modulo by zero

有些同学看见一段英文提示就慌了。其实没那么复杂，python的错误提示做得还是很标准的。

它告诉我们错误发生在test.py文件中的第5行

a += i / b

这一句上。

这个错误是“ZeroDivisionError”，也就是除零错。

“integer division or modulo by zero”，整数被0除或者被0模(取余数)。

因为0不能作为除数，所以当b随机到0的时候，就会引发这个错误。

知道了原因，就可以顺利地解决掉这个bug。

以后在写代码的时候，如果遇到了错误，先别急着去改代码。试着去读一读错误提示，看看里面都说了些啥。

2. 输出调试信息

我们在所有课程的最开始就教了输出函数“print”。它是编程中最简单的调试手段。有的时候，仅从错误提示仍然无法判断出程序错误的原因，或者没有发生错误，但程序的结果就是不对。这种情况下，通过输出程序过程中的一些状态，可以帮助分析程序。

把前面那个程序改造一下，加入一些与程序功能无关的输出语句：

```
import random

a = 0

for i in range(5):

    print ('i: %d' % i)
```

```
b = random.choice(range(5))

print ('b: %d' % b)

a += i / b

print ('a: %d' % a)

print ()

print (a)
```

运行后的输出结果（每次结果都会不一样）：

i: 0

b: 3

a: 0

i: 1

b: 3

a: 0

i: 2

b: 3

a: 0

i: 3

b: 0

Traceback (most recent call last):

File "C:\Users\Crossin\Desktop\py\test.py", line 7, in <module>

a += i / b

ZeroDivisionError: integer division or modulo by zero

当b的值为0时，发生了除零错。这次可以更清晰地看出程序出错时的状态。

在真实开发中，程序的结构可能会非常复杂。通过输出调试信息，可以有效地缩小范围、定位错误发生的位置，确认错误发生时的场景，进而找出错误原因。

- [← 60. 计时](#)
- [62. python 2 到 3 的新手坑 →](#)

【Python 第62课】python 2 到 3 的新手坑

昨天挖了个坑，论坛上已经有不少解答了，还有c语言的版本。今天先不填坑，让题目再飞一会儿，没做的同学可以周末试着写写玩儿。

周三的时候去参加“编程一小时”活动，过程中发现，python版本2和版本3之间一些小改动把很多人都给坑了，花了大量的时间在这件事情上。所以今天来讲一下最大的两个坑：print 和 input。

print

我们在课程最开始的时候就讲过 print，在版本2的使用方法是：

```
print 'this is version 2'
```

也可以是

```
print('this is version 2')
```

但到了3，就只能加上括号，像一个函数一样来使用 print：

```
print('this is version 3')
```

假如你看了基于2的教程（比如我写的），然后又装了python 3，可能会奇怪为什么完全照着写，结果却不一样。

另外2里不换行输出是加上逗号：

```
print '*',
```

到了3里就要改成：

```
print('*', end=' ')
```

input

而 input 就更绕一点。2里面有两个用来从命令行接受输入的函数：input 和 raw_input。

```
value = input()
```

input 接收的是一个值或变量，也就是说，你如果输 123，程序接收到的就是整数 123，你输 True，就是 bool 值 True。如果你输了 abc，程序会认为这是一个叫做 abc 的变量，而假如你没有定义过这个变量，就会报错。

所以，当你想用 input 得到一段文字的话，必须把文字写在引号 "" 或 " 中。

```
text = raw_input()
```

raw_input 接收的则是你输入的字符串，而不管你输的是什么内容。如果你直接拿 raw_input 得到的“数字”去比较大，则会得到奇怪的结果。

在版本3里，为了减少混乱，这两种输入方式被合并了。只是合并的方式又坑了新手：它保留了 input 这个名字和 raw_input 的效果。3里只有input函数，它接收你输入的字符串，不管你输的是什么。

```
text = input()
```

这种情况下，不管你是看着3的教材用2，还是看着2的教材用3，都会踩到这个坑。3里直接拿 input 得到的“数字”比较大，会报错类型不同无法比较。

那么在3里，如何像2一样得到用户输入的一个值呢？方法是 eval()：

```
value = eval(input())
```

或者，如果你只需要一个整数值，也可以：

```
value = int(input())
```

除了一开始遇到的这两个坑外，还有其他一些可能遇到的变动，这里以3与2相比的差异来说：

- 打开文件不再支持 file 方法，只能用 open
- range不再返回列表，而是一个可迭代的range对象

- 除法 / 不再是整除，而是得到浮点数，整除需要用双斜杠 //
- urllib和urllib2合并成了urllib，常用的urllib2.urlopen()变成了urllib.request.urlopen()
- 字符串及编码相关有大变动，简单来说就是原来的str变成了新的bytes，原来的unicode变成了新的str。具体内容比较多，可以公众号回复 编码，有一篇专门讲3的字符编码问题

变动不止这些，这里仅列出初学者比较常见的几个。更多问题可以给我们留言，或者在 bbs.crossincode.com 上发帖提问，看到后会回复。

- [← 61.调试程序](#)
- [63.python shell →](#)

【Python 第63课】python shell

各位好久不见，我终于又更新了:D。今天抽空来讲点非常非常基础的东西，关于在哪里写 python。

如果你已经编写过自己的程序，相信对这些内容已经熟悉。但很多刚刚接触编程的人，对于在 python 里编写并运行代码，还时常有些疑问。

一般来说，有两种运行 python 代码的方法：

1. 使用交互式的带提示符的解释器
2. 使用源文件

第一种方法，所谓“交互式的带提示符的解释器”，也被称做 python shell。当你安装好 python，并正确配置系统变量 PATH 后（linux 和 mac 上通常都预装并配置好了 python），在命令行里输入 python，会看到诸如以下的提示：

```
$ python
Python 3.6.7 (default, Oct 25 2018, 09:16:13)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

这就是 python shell。三个右括号 >>> 是 python 输入的提示符，它表示 python 解释器已经准备好了，等待你的命令。

在这里输入 python 的命令，并输入回车，python 解释器将在下一行立刻给出运行的结果。

以最简单的 print 方法为例：

```
>>> print ('hello world')
hello world
>>>
```

当输入完并回车之后，python 会立刻在后面输出你指定的字符串。

python shell 可以非常方便的运行 python 语句，这一点对于调试、快速组建和测试相当有用。当你在编写代码的过程中，对一些方法不确定的时候，可以通过 python shell 来进行试验。一(cheng)些(xu)人(yuan)甚至把 Python shell 当作计算器来使用：

```
>>> 1 + 1
2
>>> x = 1
>>> y = 2
>>> x + y
3
>>>
```

注意上面的例子中，我们在 python shell 中定义了变量。这些变量值在 python shell 打开时一直有效，关闭后变量值则会丢失，除非你通过其他的方式进行保存。

但 python shell 不足的地方是，当你写多行代码时很不方便。你可以尝试写一个 if 语句：

```
>>> if 2 > 1:
...     print ('right')
... 
```

```
right
```

```
>>>
```

你需要在第二行开头加上空格，并用两个回车结束语句。当你发现前面某行有错误时，想要回去修改就很麻烦。

另外，python shell 里写好的代码也很难保存（至少我目前还不知道有什么可行的方法）。所以一般并不会用它来“真正地”写代码。当你需要写一个相对完整的 python 程序时，你需要写在一个“源文件”中。这就是运行 python 的第二种方法。

仍然以 print 为例：

用一个文本编辑器新建一个文件，在里面输入：

```
print ("hello world")
```

保存这个文件为 hello.py。注意，有些同学可能会习惯命名为 print.py，请不要这么做。不要以任何 python 的内置方法或者你会使用到的模块名来命名你自己的代码文件。

然后在命令行中，进入到这个文件所在的文件夹，输入 python hello.py。你会看到：

```
$python hello.py
```

```
hello world
```

```
$
```

这时候不会进入 python shell，而是直接输出了程序的结果。换句话说，python 执行了我们写在源文件 hello.py 中的代码。

python 自带了一个叫做 IDLE 的编辑器。你可以在程序列表里找到并打开它，也可以通过命令行中输入 idle 打开。

打开后，你会看到一个默认的窗口，里面有我们前面说到的 >>> 提示符。这就是 IDLE 里的 python shell，和我们前面说的命令行中的效果是一样的，可以用来调试简单的命令。在这里，你还可以通过快捷键 alt + p（mac 上是 ctrl + p）来返回之前输入过的命令。

如果要编辑源文件，则需要在菜单栏中选择 File -> New Window。这时打开的新窗口就是源文件窗口。在里面写好你的 python 代码后，点击菜单栏上的 Run -> Run Module（或者按键盘上的 F5），按提示保存后，就会运行这个源文件，输出结果将会在之前的 python shell 窗口中显示。对于初学者来说，这还是比较方便的开发方式。

当然，还有很多 python 的 IDE（集成开发环境）可供选择，在此我不一一介绍。但基本都包括 python shell 和源代码编辑这两块核心功能。选择合适的方式去写 python，会让你事半功倍。

最后说点题外话：最近工作上的事情比较忙，一心难以二用，因此耽搁了很久也没有更新文章。目前这种状况仍然没有改变，所以还是不能保证更新的频率，望诸位理解。好在 python 的入门知识已基本都讲到了，点击文章末尾的“阅读原文”可进入之前的文章目录。接下来，我会挑一些小技巧或者常见问题进行分享。同时，我依然会在微信、论坛和 QQ 群里出现，尽量回答大家的问题，欢迎来各种讨论。

- [← 62.python 2 到 3 的新手坑](#)
- [64.列表解析 →](#)

【Python 第64课】列表解析

大家假期过得可好？今天来讲讲 Python 里一个我非常喜欢的特性--列表解析（List Comprehension）。所谓列表解析（也有翻译成列表综合），就是通过一个已有的列表生成一个新的列表。

直接看例子：

假设有一个由数字组成的 list，现在需要把其中的偶数项取出来，组成一个新的 list。一种比较“正常”的方法是：

```
list_1 = [1, 2, 3, 5, 8, 13, 22]
list_2 = []
for i in list_1:
    if i % 2 == 0:
        list_2.append(i)
print (list_2)
```

输出

```
[2, 8, 22]
```

此方法通过循环来遍历列表，对其中的每一个元素进行判断，若模取2的结果为0则添加至新列表中。

使用列表解析实现同样的效果：

```
list_1 = [1, 2, 3, 5, 8, 13, 22]
list_2 = [i for i in list_1 if i % 2 == 0]
print (list_2)
```

输出

```
[2, 8, 22]
```

[i for i in list_1] 会把 list_1 中的每一个元素都取出来，构成一个新的列表。

如果需要对其中的元素进行筛选，就在后面加上判断条件 if。所以 [i for i in list_1 if i % 2 == 0] 就是把 list_1 中满足 i % 2 == 0 的元素取出来组成新列表。

进一步的，在构建新列表时，还可以对于取出的元素做操作。比如，对于原列表中的偶数项，取出后要除以2，则可以通过 [i/2 for i in list_1 if i % 2 == 0] 来实现。输出为 [1, 4, 11]。

在实际开发中，适当地使用列表解析可以让代码更加简洁、易读，降低出错的可能。

留一道作业：

用一行 Python 代码实现：把1到100的整数里，能被2、3、5整除的数取出，以分号（;）分隔的形式输出。

- [← 63.python shell](#)
- [65.函数的参数传递（1）→](#)

【Python 第65课】函数的参数传递（1）

本篇面向读者：有一点点 Python 基础

关键字：函数，参数，默认值

先说下上次课最后留的那题，我自己的解法：

```
print (';'.join([str(i) for i in range(1,101) if i % 2 == 0 and i % 3 == 0 and i % 5 == 0]))
```

另外，关于上次说的 List Comprehension，我在文中称之为“列表综合”，这是引自《简明 Python 教程》的翻译。也有同学表示叫做“列表解析”或“列表表达式”。都是一个意思，其实在写这课之前，我从来都不去“叫”它，只知道这么用而已。

我们曾经讲过 Python 中函数的参数传递（见第21课）。最基本的方式是：

定义

```
def func(arg1, arg2):
    print (arg1, arg2)
```

调用

```
func(3, 7)
```

我们把函数定义时的参数名（arg1、arg2）称为形参，调用时提供的参数（3、7）称为实参。

这种方式是根据调用时提供参数的位置进行匹配，要求实参与行参的数量相等，默认按位置匹配参数。调用时，少参数或者多参数都会引起错误。这是最常用的一种函数定义方式。

在调用时，也可以根据形参的名称指定实参。如：

```
func(arg2=3, arg1=7)
```

但同样，必须提供所有的参数。看看和func(3, 7)的运行结果有什么不同。

Python 语言还提供了其他一些更灵活的参数传递方式，如：

```
func2(a=1, b=2, c=3)
func3(*args)
func4(**kwargs)
```

今天我们先说说func2这种方式。

这种方式可以理解为，在一般函数定义的基础上，增加了参数的默认值。这样定义的函数可以和原来一样使用，而当你没有提供足够的参数时，会用默认值作为参数的值。

例如：

定义

```
def func(arg1=1, arg2=2, arg3=3):
    print (arg1, arg2, arg3)
```

调用

```
func(2, 3, 4)
func(5, 6)
func(7)
```

输出为

2 3 4

5 6 3

7 2 3

提供的参数会按顺序先匹配前面位置的参数，后面未匹配到的参数使用默认值。

也可以指定其中的部分参数，如：

```
func(arg2=8)
```

```
func(arg3=9, arg1=10)
```

输出为

1 8 3

10 2 9

或者混合起来用：

```
func(11, arg3=12)
```

输出为

11 2 12

但要注意，没有指定参数名的参数必须在所有指定参数名的参数前面，且参数不能重复。以下的调用都是错误的：

```
func(arg1=13, 14)
```

```
func(15, arg1=16)
```

定义参数默认值的函数可以在调用时更加简洁。大量 Python 模块中的方法都运用了这一方式，让使用者在调用时可以提供尽可能少的参数。

接下来的几次课，我会继续介绍其他的参数传递方式。

- [← 64.列表解析](#)
- [66.函数的参数传递（2） →](#)

【Python 第66课】函数的参数传递（2）

接着上一次的内容，来介绍一种更加灵活的参数传递方式：

```
def func(*args)
```

这种方式的厉害之处在于，它可以接受任意数量的参数。来看具体例子：

```
def calcSum(*args):  
    sum = 0  
    for i in args:  
        sum += i  
    print (sum)
```

调用：

```
calcSum(1,2,3)  
calcSum(123,456)  
calcSum()
```

输出：

```
6  
579  
0
```

在变量前加上星号前缀（*），调用时的参数会存储在一个 tuple（元组）对象中，赋值给形参。在函数内部，需要对参数进行处理时，只要对这个 tuple 类型的形参（这里是 args）进行操作就可以了。因此，函数在定义时并不需要指明参数个数，就可以处理任意参数个数的情况。

不过有一点需要注意，tuple 是有序的，所以 args 中元素的顺序受到赋值时的影响。如：

```
def printAll(*args):  
    for i in args:  
        print (i, end=' ')  
    print ()
```

调用：

```
printAll(1,2,3)  
printAll(3,2,1)
```

输出：

```
1 2 3  
3 2 1
```

虽然3个参数在总体上是相同的，但由于调用的顺序不一样，结果也是不同的。

还有一种参数传递方式，既可以按参数名传递参数，不受位置的限制，又可以像 tuple 传递一样不受数量限制。这个我将在下次课中做介绍。

- [← 65.函数的参数传递（1）](#)
- [67.函数的参数传递（3） →](#)

【Python 第67课】函数的参数传递（3）

今天来说说最为灵活的一种参数传递方式：

```
func(**kargs)
```

上次说的 `func(*args)` 方式是把参数作为 tuple 传入函数内部。而 `func(**kargs)` 则是把参数以键值对字典的形式传入。

示例：

```
def printAll(**kargs):
    for k in kargs:
        print (k, ': ', kargs[k])
```

```
printAll(a=1, b=2, c=3)
```

```
printAll(x=4, y=5)
```

输出：

```
a : 1
```

```
c : 3
```

```
b : 2
```

```
y : 5
```

```
x : 4
```

字典是无序的，所以在输出的时候，并不一定按照提供参数的顺序。同样在调用时，参数的顺序无所谓，只要对应合适的形参名就可以了。于是，采用这种参数传递的方法，可以不受参数数量、位置的限制。

当然，这还不够。Python 的函数调用方式非常灵活，前面所说的几种参数调用方式，可以混合在一起使用。看下面这个例子：

```
def func(x, y=5, *a, **b):
    print (x, y, a, b)
```

```
func(1)
```

```
func(1,2)
```

```
func(1,2,3)
```

```
func(1,2,3,4)
```

```
func(x=1)
```

```
func(x=1,y=1)
```

```
func(x=1,y=1,a=1)
```

```
func(x=1,y=1,a=1,b=1)
```

```
func(1,y=1)
```

```
func(1,2,3,4,a=1)
```

```
func(1,2,3,4,k=1,t=2,o=3)
```

输出：

```
1 5 () {}
```

```

1 2 () {}
1 2 (3,) {}
1 2 (3, 4) {}
1 5 () {}
1 1 () {}
1 1 () {'a': 1}
1 1 () {'a': 1, 'b': 1}
1 1 () {}
1 2 (3, 4) {'a': 1}
1 2 (3, 4) {'k': 1, 't': 2, 'o': 3}

```

在混合使用时，首先要注意函数的写法，必须遵守：

- 带有默认值的形参(arg=)须在无默认值的形参(arg)之后；
- 元组参数(*args)须在带有默认值的形参(arg=)之后；
- 字典参数(**kargs)须在元组参数(*args)之后。

可以省略某种类型的参数，但仍需保证此顺序规则。

调用时也需要遵守：

- 指定参数名称的参数要在无指定参数名称的参数之后；
- 不可以重复传递，即按顺序提供某参数之后，又指定名称传递。

而在函数被调用时，参数的传递过程为：

1. 按顺序把无指定参数的实参赋值给形参；
2. 把指定参数名称(arg=v)的实参赋值给对应的形参；
3. 将多余的无指定参数的实参打包成一个 tuple 传递给元组参数(*args)；
4. 将多余的指定参数名的实参打包成一个 dict 传递给字典参数(**kargs)。

是不是乍一看有点绕？没关系，赶紧打开你的编辑器，自行体会一下不同调用方式的用法。然后在未来的编程实践中慢慢熟悉吧。

- [← 66.函数的参数传递 \(2\)](#)
- [68.lambda 表达式 →](#)

【Python 第68课】lambda 表达式

Python 是一门简洁的语言，lambda 表达式则充分体现了 Python 这一特点。

lambda 表达可以被看做是一种匿名函数。它可以让你快速定义一个极度简单的单行函数。譬如这样一个实现三个数相加的函数：

```
def sum(a, b, c):  
    return a + b + c
```

```
print (sum(1, 2, 3))
```

```
print (sum(4, 5, 6))
```

输出：

```
6
```

```
15
```

如果使用 lambda 表达式来实现：

```
sum = lambda a, b, c: a + b + c
```

```
print (sum(1, 2, 3))
```

```
print (sum(4, 5, 6))
```

输出：

```
6
```

```
15
```

两种方法的结果是相同的。

lambda 表达式的语法格式：

lambda 参数列表: 表达式

定义 lambda 表达式时，参数列表周围没有括号，返回值前没有 return 关键字，也没有函数名称。

它的写法比 def 更加简洁。但是，它的主体只能是一个表达式，不可以是代码块，甚至不能是命令（print 不能用在 lambda 表达式中）。所以 lambda 表达式能表达的逻辑很有限。

lambda 表达式创建了一个函数对象，可以把这个对象赋值给一个变量进行调用，就像上面的例子中一样。

来看一个复杂一点的例子，把 lambda 表达式用在 def 函数定义中：

```
def fn(x):  
    return lambda y: x + y
```

```
a = fn(2)
```

```
print (a(3))
```

输出：

```
5
```

这里，fn 函数的返回值是一个 lambda 表达式，也就等于是一个函数对象。当以参数2来调用 fn 时，得到的结果就是：

```
lambda y: 2 + y
```

`a = fn(2)` 就相当于:

```
a = lambda y: 2 + y
```

所以 `a(3)` 的结果就是5。

`lambda` 表达式其实只是一种编码风格，这种写法更加 `pythonic`。这并不意味着你一定要使用它。事实上，任何可以使用 `lambda` 表达式的地方，都可以通过普通的 `def` 函数定义来替代。在一些需要重复使用同一函数的地方，`def` 可以避免重复定义函数。况且 `def` 函数更加通用，某些情况可以带来更好地代码可读性。

而对于像 `filter`、`sort` 这种需要内嵌函数的方法，`lambda` 表达式就会显得比较合适。这个我以后会再单独介绍。

当然对于初学者来说，了解 `lambda` 表达式还有一个重要作用就是，看懂别人写的代码。

- [← 67.函数的参数传递 \(3\)](#)
- [69.变量的作用域 →](#)

【Python 第69课】变量的作用域

在写代码的时候，免不了要使用变量。但程序中的一个变量并不一定是在哪里都可以被使用，根据情况不同，会有不同的“有效范围”。看这样一段代码：

```
def func(x):
    print ('X in the beginning of func(x): ', x)
    x = 2
    print ('X in the end of func(x): ', x)
```

```
x = 50
func(x)
print ('X after calling func(x): ', x)
```

输出：

X in the beginning of func(x): 50

X in the end of func(x): 2

X after calling func(x): 50

变量 x 在函数内部被重新赋值。但在调用了函数之后，x 的值仍然是50。为什么？

这就得说一下变量的“作用域”：

当函数内部定义了一个变量，无论是作为函数的形参，或是另外定义的变量，它都只在这个函数的内部起作用。函数外即使有和它名称相同的变量，也没有什么关联。这个函数体就是这个变量的作用域。像这样在函数内部定义的变量被称为“局部变量”。

要注意的是，作用域是从变量被定义的位置开始。像这样的写法是有问题的：

```
def func():
    print (y)
    y = 2
    print (y)
```

报错：

UnboundLocalError: local variable 'y' referenced before assignment

因为在 y=2 之前，y 并不存在，调用 y 的值就会出错。

回到开始那个例子：

在函数 func 外部，定义的变量 x，赋值为 50，作为参数传给了函数 func。而在函数 func 内部，变量 x 是形参，它的作用域是整个函数体内部。它与外面的那个 x 没有关系。只不过它的初始值是由外面那个 x 传递过来的。

所以，虽然函数体内部的 x 被重新赋值为 2，也不会影响外面那个 x 的值。

不过有时候，我们希望能够在函数内部去改变一些变量的值，并且这些变量在函数外部同样被使用到。怎么办？

一种方法是，用 return 把改变后的变量值作为函数返回值传递出来，赋值给对应的变量。比如开始的那个例子，可以在函数结尾加上

```
return x
```

然后把调用改为

```
x = func(x)
```

还有一种方法，就是使用“全局变量”。

在 Python 的函数定义中，可以给变量名前加上 `global` 关键字，这样其作用域就不再局限在函数块中，而是全局的作用域。

通过 `global` 改写开始的例子：

```
def func():  
    global x  
  
    print ('X in the beginning of func(x): ', x)  
  
    x = 2  
  
    print ('X in the end of func(x): ', x)
```

```
x = 50
```

```
func()
```

```
print ('X after calling func(x): ', x)
```

输出：

X in the beginning of func(x): 50

X in the end of func(x): 2

X after calling func(x): 2

函数 `func` 不再提供参数调用。而是通过 `global x` 告诉程序：这个 `x` 是一个全局变量。于是函数中的 `x` 和外部的 `x` 就成为了同一个变量。这一次，当 `x` 在函数 `func` 内部被重新赋值后，外部的 `x` 也随之改变。

前面讲的局部变量和全局变量是 Python 中函数作用域最基本的情况。实际上，还有一些略复杂的情况，比如：

```
def func():  
    print ('X in the beginning of func(x): ', x)  
  
    # x = 2  
  
    print ('X in the end of func(x): ', x)
```

```
x = 50
```

```
func()
```

```
print ('X after calling func(x): ', x)
```

输出：

X in the beginning of func(x): 50

X in the end of func(x): 50

X after calling func(x): 50

程序可以正常运行。虽然没有指明 `global`，函数内部还是使用到了外部定义的变量。然而一旦加上

```
x = 2
```

这句，程序就会报错。因为这时候，`x` 成为一个局部变量，它的作用域从定义处开始，到函数体末尾结束。

建议在写代码的过程中，显式地通过 `global` 来使用全局变量，避免在函数中直接使用外部变量。

- [← 68.lambda 表达式](#)
- [70.map 函数 →](#)

【Python 第70课】map 函数

来看两个问题：

1. 假设有一个数列，如何把其中每一个元素都翻倍？
2. 假设有两个数列，如何求和？

第一个问题，普通程序员大概会这么写：

```
lst_1 = [1,2,3,4,5,6]
lst_2 = []
for item in lst_1:
    lst_2.append(item * 2)
print (lst_2)
```

Python 程序员大概会这么写：

```
lst_1 = [1,2,3,4,5,6]
lst_2 = [i * 2 for i in lst_1]
print (lst_2)
```

这是我在《【Python 第65课】列表综合》里说到的方法。

今天来说另一种 Python 程序员常用的写法 -- map：

```
lst_1 = [1,2,3,4,5,6]
def double_func(x):
    return x * 2
lst_2 = map(double_func, lst_1)
print(list(lst_2))
```

（注：在py3里必须将lst_2转换成list对象后输出）

map 是 Python 自带的内置函数，它的作用是把一个函数应用在一个（或多个）序列上，把列表中的每一项作为函数输入进行计算，再把计算的结果以列表的形式返回。

map 的第一个参数是一个函数，之后的参数是序列，可以是 list、tuple。

所以刚刚那个问题也可以写成：

```
lst_1 = (1,2,3,4,5,6)
lst_2 = map(lambda x: x * 2, lst_1)
print(list(lst_2))
```

这里原数据改为了元组，函数用 lambda 表达式替代。可参考《【Python 第69课】lambda 表达式》。

map 中的函数可以对多个序列进行操作。最开始提出的第二个问题，除了通常的 for 循环写法，如果用列表综合的方法比较难实现，但用 map 就比较方便：

```
lst_1 = [1,2,3,4,5,6]
lst_2 = [1,3,5,7,9,11]
lst_3 = map(lambda x, y: x + y, lst_1, lst_2)
print(list(lst_3))
```

map 中的函数会对对应的列表中依次取出元素，作为参数使用，同样将结果以列表的形式返回。所以要注意的是，函数的参数个数要与 map 中提供的序列组数相同，即函数有几个参数，就得有几组数据。

对于每组数据中的元素个数，如果有某组数据少于其他组，map 会以 None 来补全这组参数。

- [← 69.变量的作用域](#)
- [71.reduce 函数 →](#)

【Python 第71课】reduce 函数

上次说了 Python 中一个比较有意思的内置函数 map，今天再来介绍另一个类似的函数：reduce

map 可以看作是把一个序列根据某种规则，映射到另一个序列。reduce 做的事情就是把一个序列根据某种规则，归纳为一个输出。在 Python3 里，reduce 已经被移出内置函数，使用 reduce 需要先通过 `from functools import reduce` 引入。

上栗子。以前我们给过一个习题，求1累加到100的和。寻常的做法大概是这样：

```
sum = 0

for i in range(1, 101):

    sum += i

print (sum)
```

如果用 reduce 函数，就可以写成：

```
from functools import reduce

lst = range(1, 101)

def add(x, y):

    return x + y

print (reduce(add, lst))
```

解释一下：

```
reduce(function, iterable[, initializer])
```

第一个参数是作用在序列上的方法，第二个参数是被作用的序列，这与 map 一致。另外有一个可选参数，是初始值。

function 需要是一个接收2个参数，并有返回值的函数。它会从序列 iterable 里从左到右依次取出元素，进行计算。每次计算的结果，会作为下次计算的第一个参数。

提供初始值 initializer 时，它会作为第一次计算的第一个参数。否则，就先计算序列中的前两个值。

如果把刚才的 lst 换成 [1,2,3,4,5]，那 reduce(add, lst) 就相当于 (((1+2)+3)+4)+5)。

同样，可以用 lambda 函数：

```
from functools import reduce

reduce((lambda x, y: x + y), range(1, 101))
```

所以，在对于一个序列进行某种统计操作的时候，比如求和，或者诸如统计序列中元素的出现个数等（可尝试下如何用 reduce 做到），可以选择使用 reduce 来实现。相对可以使代码更简洁。

我觉得，写代码的可读性是很重要的事情，简洁易懂的代码，既容易让别人看懂，也便于自己以后的维护。同时，较少的代码也意味着比较高的开发效率和较少的出错可能。应尽量避免写混乱冗长的代码。当然，也不用为了一味追求代码的精简，总是想方设法把代码写在一行里。那就又走了另一个极端，同样也缺乏可读性。而至于是否使用类似 map、reduce 这样的方法，也是根据 need 和个人习惯，我认为并没有一定的规则限制。

- [← 70.map 函数](#)
- [72.多线程 →](#)

【Python 第72课】多线程

很多人使用 python 编写“爬虫”程序，抓取网上的数据。

举个例子，通过这个接口：

http://wthrcdn.etouch.cn/weather_mini?city=北京

查询天气。

如果我们想用这套代码同时抓取10个城市的天气，如果一个一个的抓，效率就很低了。

然而想一下，我们抓一个城市天气的过程是独立，并不依赖于其他城市的结果。因此没必要排好队一个一个地按顺序来。那么有没有什么办法可以同时抓取好几个城市的天气呢？

答案就是：多线程。

来说一种简单的多线程方法：

python 里有一个 threading 模块，其中提供了一个函数：

```
threading.Thread(target=function, args=(), kwargs={})
```

function 是开发者定义的线程函数，

args 是传递给线程函数的参数，必须是tuple类型，

kwargs 是可选参数，字典类型。

调用 threading.Thread 之后，会创建一个新的线程，参数 target 指定线程将要运行的函数，args 和 kwargs 则指定函数的参数来执行 function 函数。

改写一下前面的代码，将抓取的部分放在一个函数中：

```
def get_weather(city):
    req = requests.get('http://wthrcdn.etouch.cn/weather_mini?city=%s' % city)
    dic_city = req.json()

    city_data = dic_city.get('data') # 没有'data'的话返回 []
    print(city_data.get('city'))

    if city_data:
        city_forecast = city_data['forecast'][0] # 下面的都可以换成'get'方法
        print(city_forecast.get('date'))
        print(city_forecast.get('high'))
        print(city_forecast.get('low'))
        print(city_forecast.get('type'))
    else:
        print('未获得')

    print()
```

之后，程序采用了三个循环，在第一个循环中，针对每一个城市，都创建了一个新线程，并将线程加入到一个列表中，用于之后的启动。

```
threads = []
```

```

cities = ['北京', '南京', '上海', '深圳', '广州', '杭州', '苏州', '天津', '西安', '成都']

files = range(len(cities))

for i in files: # 创建线程

    t = threading.Thread(target=get_weather, args=(cities[i],))

    threads.append(t)

```

在第二个循环中，start 正式开启子线程：

```

for i in files:

    threads[i].start()

```

在第三个循环中，join 用来同步数据，主线程运行到这一步，将会停下来等待子线程运行完毕。没有这句，主线程则会忽略子线程，运行完自己的代码后结束程序。

```

for i in files:

    threads[i].join()

```

```

72threading.py x
1  # -*- coding: 'utf-8' -*-
2
3  import requests, threading
4
5  def get_weather(city):
6      req = requests.get('http://wthrcdn.etouch.cn/weather_mini?city=%s' % city)
7      dic_city = req.json()
8
9      city_data = dic_city.get('data') # 没有'data'的话返回 []
10
11     if city_data:
12         city_forecast = city_data['forecast'][0] # 下面的都可以换成'get'方法
13         print(
14             city_data.get('city'),
15             city_forecast.get('date'),
16             city_forecast.get('high'),
17             city_forecast.get('low'),
18             city_forecast.get('type')
19         )
20     else:
21         print('未获得')
22
23     threads = []
24     cities = ['北京', '南京', '上海', '深圳', '广州', '杭州', '苏州', '天津', '西安', '成都']
25     files = range(len(cities))
26     for i in files: # 创建线程
27         t = threading.Thread(target=get_weather, args=(cities[i],))
28         threads.append(t)
29     for i in files:
30         threads[i].start()
31     for i in files:
32         threads[i].join()
33     print('结束获取')

```

```

南京 12日星期三 高温 28℃ 低温 20℃ 多云
成都 12日星期三 高温 30℃ 低温 22℃ 晴
天津 12日星期三 高温 34℃ 低温 22℃ 多云
上海 12日星期三 高温 27℃ 低温 21℃ 多云
杭州 12日星期三 高温 26℃ 低温 21℃ 多云
北京 12日星期三 高温 33℃ 低温 22℃ 多云
广州 12日星期三 高温 29℃ 低温 24℃ 大雨
西安 12日星期三 高温 33℃ 低温 19℃ 多云
苏州 12日星期三 高温 26℃ 低温 21℃ 多云
深圳 12日星期三 高温 30℃ 低温 24℃ 雷阵雨
结束获取

```

Process finished with exit code 0

从输出结果可以看出：

- 在程序刚开始运行时，已经发送所有请求
- 收到的请求并不是按发送顺序，先收到就先显示
- 同样记录了所有10个城市的天气

所以，对于这种耗时长，但又独立的任务，使用多线程可以大大提高运行效率。但在代码层面，可能额外需要做一些处理，保证结果正确。如上例中，如果需要按输入列表城市顺序进行输出，就要另行排序。

多线程通常会用在网络收发数据、文件读写、用户交互等待之类的操作上，以避免程序阻塞，提升用户体验或提高执行效率。

多线程的实现方法不止这一种。另外多线程也会带来一些单线程程序中不会出现的问题。这里只是简单地开个头。

- [← 71.reduce 函数](#)
- [返回首页](#)