



第一章

安装 Python

本章内容:

- 下载和安装 Python 解释器
- 使用 IDLE 的交互界面运行指令
- 如何使用本书
- 本书网站: <http://inventwithpython.com>

你好！这本书将通过向你演示如何开发游戏的方式来教你编程。只要你理解了本书中的游戏是如何运行的，你就学会了如何开发自己的游戏。你需要的东西只有一台电脑，某个叫做 Python 解释器的软件以及这本书。我们需要的软件可以从网上免费下载。

我小的时候发现了一本像这本书一样教我编写自己的第一个程序和游戏的书。那本书很有趣而且简单。现在长大了，我依然觉得编程很有意思，何况还会有人付我工钱。不过，即使你不想长大后成为一名程序员，学会编程依然是有用和有趣的。

电脑是非常有用的机器。令人高兴的是学会编程很容易。如果你能读懂这本书，你就可以学会编程。正如一本故事书只是一堆给人看的句子一样，电脑程序也不过是一堆由电脑执行的指令。

这些指令就像你寻找你朋友的家时你朋友给你的一步一步的指示。（在路灯处左转，直走两个街区，一直走到你看到右边的第一幢蓝色房子为止。）电脑也是这样按照你给定的顺序，一句接着一句的执行你的指令。电子游戏本质上就是电脑程序。（并且是很有趣的程序！）

在这本书里，你需要知道的内容会是**这样的**。比如，我在上一段中定义的**程序**。

为了告诉电脑你想让它干什么，你要用电脑能理解的语言来写程序。本书教授的编程语言叫做 Python。编程语言有很多，包括 BASIC, Java, Pascal, Haskell 以及 C++（读作“C 加加”）。

在我小的时候，大多数人把 BASIC 作为学习编程的第一门语言。不过，从那时开始，人们发明了很多新的编程语言，包括 Python。Python 甚至比 BASIC 更容易学，而且它是一门被很多专业程序员所使用的重要的语言。很多大人在工作中使用 Python（写着玩的时候也用 Python）。

这本书中的游戏和你在 Xbox, Playstation 或 Wii 上玩过的游戏比起来，可能会很简单。它们没有精美的画面和好听的音乐，但是，这是因为它们是用来教你基础知识的。我是故意把它们做得这么简单的，这样我们才能专注于学习编程。游戏不一定要复杂才有趣。吊死鬼（Hangman），井字棋（Tic Tac Toe）和密码编写这些游戏虽然简单但也很有趣。

我们也会学习如何在 Python 交互界面中让电脑解决数学问题。（如果你的数学不是很好也不用担心。只要你知道加法和乘法就足够了。编程更注重解决问题和规划，而不仅仅是解决数学问题。）

下载和安装 Python

在我们开始编程之前你要安装 Python 解释器。（你可能需要向大人求助。）**解释器**是一款能理解你用 Python 语言写的指令的程序。没有解释器，你的电脑就无法理解那些指令，你的程序也就无法运行。（从现在起我们将把“Python 解释器”简称为“Python”。）

由于我们要用 Python 语言来编写我们的游戏，所以我们要先到 Python 语言的官方网站（<http://www.python.org>）下载 Python。

我将会教你如何在微软的 Windows 系统下安装 Python，这不是因为 Windows 是我最喜欢的操作系统，而是因为你的电脑装的系统很可能是 Windows。也许你需要别人帮你下载和安装 Python。

你打开 python.org 后应该会在左边看到一系列链接（About, News, Documentation, 下载, 等等）点击**下载**链接，你会转到下载页面。在此页面找到 **Python 3.1 Windows Installer** (Windows binary -- does not include source) 然后点击此链接下载 Windows 版的 Python。

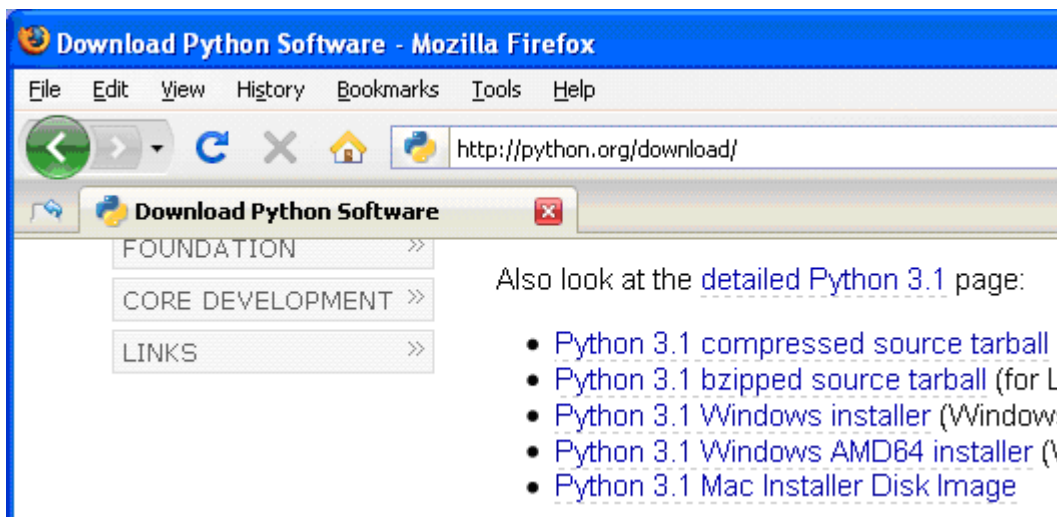


图 1-1: 点击 Windows installer 链接从 <http://www.python.org> 下载 Windows 版的 Python

双击你刚才下载的文件 *python-3.1.msi* 以启动 Python 安装程序。（如果没有开始安装，右键该文件并选择安装。）安装程序启动后，只要点 **Next** 按钮并接受选项（无需做改变）。安装完成后点击 **Finish**。

注意! 确定你安装的是 Python 3，而不是 Python 2。本书中的程序使用 Python 3，如果你在 Python 2 下运行将会出错。

在 Mac OS 下的安装过程差不多也是这样。只是你要从 Python 网站下载.dmg 文件而不是.msi 文件。这个文件在下载页面的下载链接差不多是这样“Mac Installer disk image (3.1.1)”。

如果你的操作系统是 Ubuntu，你可以通过在终端(点击 应用程序 > 附件 > 终端)输入 `sudo apt-get install python3` 然后按回车来安装 Python。安装时你需要输入 root 密码，所以你可能需要让电脑的主人来输入密码。

你去下载的时候可能会有比 3.1 更新的 Python 版本可用，如果是这样，下载最新版本就行了。本书中的游戏程序也可以在新版本中正确运行。如果你有任何问题都可以去谷歌 (Google) 搜索“在<你的操作系统>下安装 Python”。Python 是很流行的语言，所以找到帮助文档应该不难。

本书的网站上也提供了一个安装 Python 的视频教程。（地址：<http://inventwithpython.com/videos/>）。

启动 Python

如果你的操作系统是 Windows XP，你可以通过点击**开始，程序，Python 3.1, IDLE (Python GUI)**来启动 Python。启动 Python 后的界面应该和图 1-2 相似。（不过不同的操作系统会有些许区别。）

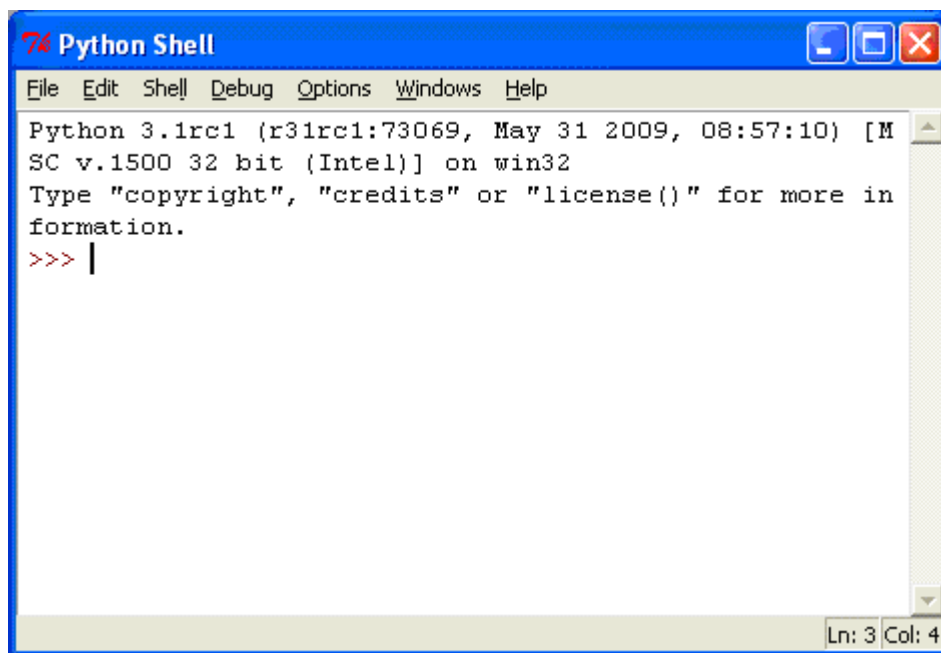


图 1-2: Windows 下的 IDLE 交互界面

IDLE 是 **Interactive DeveLopment Environment**（交互开发环境）的简称。开发环境的作用是使编写 Python 程序更容易。我们将用 IDLE 来输入和运行我们的程序。

你一开始运行 IDLE 时出现的窗口叫做交互界面（interactive shell）。它是一个让你向电脑输入指令的程序。Python 交互界面允许你输入 Python 指令并将这些指令发送给 Python 解释器去执行。由于 Python 交互界面是交互式的，所以当我们在输入 Python 指令时电脑会读取我们的指令并作出一些回应。（理想情况下会按照我们期望的那样回应我们，这取决于我们是不是输入了正确的指令。）

如何使用本书

在你开始使用本书前，你应该了解关于本书的一些东西。《使用 Python 编写自己的游戏》和其他的编程书籍是不同的，因为本书专注于不同游戏的完整源代码。本书不像其他书那样教你编程的概念，然后让你自己去琢磨如何用你所学的知识去编写有趣的游戏。这本书是向你展示有趣的游戏，然后给你解释它们是如何创造出来的。

特色程序

大多数章节是以特色程序的运行结果作为开头的。这些运行结果向你展示了这些程序的输出结果是怎样的，用户输入的内容则用**粗体**显示。这样可以让你对输入代码并运行后的游戏的完整情况有一个了解。

有些章节也会向你展示游戏的完整源代码，不过你要记住：目前你还不用输入每一行代码。相反，你可以先阅读该章节的内容，等你理解了每一行代码是干什么的之后再输入代码。

你也可以从本书的网站下载源代码。访问 <http://inventwithpython.com/source> 然后按照指示下载源代码。

行号与空格

你自己输入代码的时候不要把每行代码开头的行号输入进去。如果你在书上看到这个：

```
9. number = random.randint(1, 20)
```

你不用输入左边的“9”和它后面的空格。只需输入：

```
number = random.randint(1, 20)
```

那些数字只是为了在解释代码的时候可以方便的指代某行代码，它们不是程序的组成部分。

除了行号以外，你还要确定你输入的代码和你在书上看到的是一模一样的。你要注意有些代码不是从页面的最左边开始的，而是缩进了 4 个或 8 个空格。确定你在每行的开头输入了正确的空格数。（由于在 IDLE 中每个字符的宽度是一样的，所以您可以通过数该行的上一行或下一行的字符来确定空格数。）

比如，你可以看到第二行代码缩进了 4 个空格，因为它的空格部分的上一行有四个字符（"while"）。第三行在此基础上又缩进了 4 个空格（第三行的空格部分的上一行有四个字符"if n"）：

```
while guesses < 10:
    if number == 42:
        print('Hello')
```

本书的文字换行

有些代码行太长了，不能在全部写在同一行，因此多出来的代码会写在下一行。当你输入这些代码的时候要全部放到同一行，不要按回车换行。

你可以把左边的行号作为新的一行开始的标志。例如，即使第一行代码换了行，下面的代码也只有两行：

```
1. print('This is the first line! xxxxxxxxxxxxxxxxxxxx
        xxxxxxxxxxxxxxxxxxxx')
2. print('This is the second line! ')
```

在线步进调试程序

如果想步进调试书中的程序，你可以访问 <http://inventwithpython.com/traces>。步进调试的意思是让电脑一次执行一行代码。步进调试页面对每一行代码都进行了的注释和提示，这些内容是用来向你解释程序在干什么的，这样可以帮助你更好的理解程序运行的方式。

在线检查代码

本书中的一些游戏的代码有些长。虽然自己输入这些游戏的源代码对学习 Python 非常有帮助，但你在输入的过程中可能会不小心输入错误而导致你的程序崩溃。而且错误可能不容易查找。

这时你可以把你的源代码复制粘贴到本书的网站的在线 diff 工具中。diff 工具会告诉你书中的代码和你输入的代码的区别。这是一个查找你程序中的输入错误的简单方法。

复制粘贴文本是一项很有用的电脑技能，尤其是在编程的时候。本书的网站上（<http://inventwithpython.com/videos/>）有关于复制粘贴的视频教程。

在线 diff 工具的网址是：<http://inventwithpython.com/diff>。在本书的网站上有关于 diff 工具的使用的视频教程（<http://inventwithpython.com/videos/>）。

总结

本章通过向你展示可以免费下载 Python 的网站 python.org 来帮助你起步。安装完成并运行 Python IDLE 之后，我们就已经为在下一章开始学习编程做好了准备。

本书的网站（<http://inventwithpython.com>）有更多关于每章内容的信息，包括帮助你理解每一行代码的作用的步进调试页面。



第二章 交互界面

本章内容：

- 整数和浮点数
- 表达式
- 值
- 运算符
- 运算表达式
- 在变量中保存值
- 重写变量

在我们开始编写计算机游戏之前要先学一些基本的编程概念。这包括值，运算符，表达式和变量。我们不会在这章开始编程，但是了解这些名词会让学习编程更简单。这是因为大多数编程都是把一些简单的概念组合在一起来写出高级程序的。

让我们从学习如何使用 Python 的交互界面开始吧。

一些简单的数学

在 Windows 下启动 IDLE 的方法是点击开始，程序，然后点击 **Python 3.1**，然后是 **IDLE (Python GUI)**。启动 IDLE 之后，让我们用 Python 做一些简单的数学运算。我们可以把 Python 当做计算器来用。输入 $2+2$ 然后按回车。（有些键盘叫做 Return 键。）如图 2-1 所示，计算机应该输出 4 作为回应，即 $2+2$ 的和。

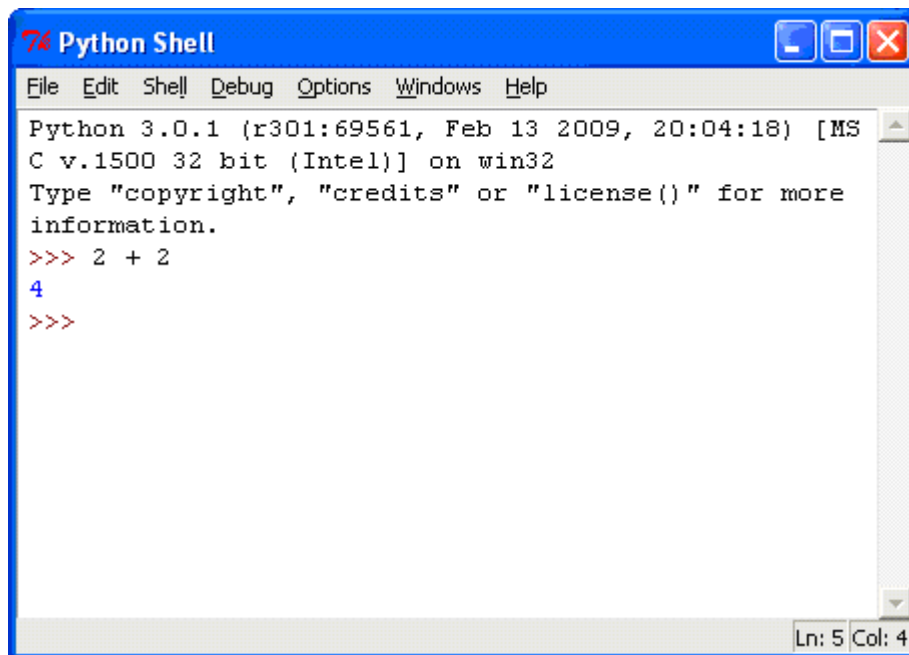


图 2-1: 在交互界面输入 2+2

正如你所看到的，我们可以把 Python 当做计算器来用。这本身不是程序，因为现在我们只是在学习基本概念。符号+告诉计算机把数字 2 和 2 相加。如果要做减法和乘法用 (-) 和 (*) 即可，如：

表 2-1: Python 中的数学运算

2+2	加法
2-2	减法
2*2	乘法
2/2	除法

这些符号 (+, -, *, 和 /) 被称为**运算符**，因为它们告诉计算机对它们周围的数字执行特定的操作。

整数和浮点数

在编程中（在数学中也是），像 4.0 和 99 这样的数被称为**整数**。带有分数或者小数点的数（像 3.5, 42.1 和 5.0）就不是整数。在 Python 中，数字 5 是整数，但是如果把它写作 5.0，那么它就不是整数了。有小数点的数被称为**浮点数**。在数学中，5.0 和 5 一样，都是整数，但是计算机程序和计算机把所有带小数点的数都不当做是整数。

表达式

把这些数学问题输入到交互界面，输完每个问题后按回车。

```
2+2+2+2+2
8*6
10-5+6
2 +      2
```

图 2-2 是你在交互界面输入完上面的内容后 IDLE 的样子

```
interface and no data is ser
*****
IDLE 1.2.1
>>> 2+2
4
>>> 2+2+2+2+2
10
>>> 8*6
48
>>> 10-5+6
11
>>> 2 +      2
4
>>> |
```

图 2-2: 输入完成后 IDLE 的样子



图 2-3: 一个表达式由值和运算符组成

这些数学问题被称为表达式。计算机可以在几秒钟的时间里解决几百万个这样的问题。表达式由**值**（数字）和**运算符**（数学符号）组成。让我们来看看值和表达式到底是什么。

如你所见，前面的例子中的最后一个表达式中数值和运算符之间有很多空格，这是允许的。（不过要记住一定在要从一行的最左边开始写代码，前面不能有空格。）

数字是值的一种。整数是数字一种。但是，即使整数是数字，也不代表所有数字都是整数。（例如，分数和像 2.5 这样的小数都是数字但却不是整数。）

这就像猫是一种宠物，但并不是所有宠物都是猫。人家可以有一只宠物狗或者一只宠物螃蟹。**表达式**是由值（像 8 和 6 这样的整数）和链接这些值的运算符（如乘法符号*）组成的。一个单独的值也是一个表达式。

下一章，我们将会学习在表达式中使用文本。**Python** 不是只能处理数字，它远远不止是一个好用的计算器。

运算表达式

计算机处理 $10+5$ 这个表达式的并得到值 15，这时我们就说这个表达式**被运算了**。运算一个表达式可以让表达式简化为一个值，就像解决数学问题可以将问题简化一个数字，即答案。

表达式 $10+5$ 和 $10+3+2$ 有相同的值，因为它们的运算结果都是 15。即使是单个值也是表达式：表达式 15 运算后的值为 15。

不过，如果你只是在交互界面输入 $5+$ ，你会得到一个错误信息。

```
>>> 5 +  
SyntaxError: invalid syntax
```

这里之所以会出错是因为 $5+$ 不是一个表达式。表达式有值和连接值的运算符，但是在 **Python** 中 $+$ 这个运算符必须用来连接两个东西。我们只给了它一个。这就是出错的原因。句法错误的意思是计算机无法理解你输入的命令，因为你输入错误。只要你输入的命令 **Python** 不能理解它就一定会显示错误信息。

这看起来可能不重要，但是编程不仅仅是告诉计算机要干什么，同时也是要知道如何准确的告诉计算机去做。

表达式中的表达式

表达式也可以包含其他表达式。例如，在 $2+5+8$ 这个表达式中， $2+5$ 本身也是一个表达式。**Python** 会运算 $2+5$ 得到 7，所以原来的表达式就变成了 $7+8$ 。然后 **Python** 在运算这个表达式得到 15。

你可以把表达式想象成一叠煎饼。如果你把两叠煎饼放在一起，你仍然只有一叠煎饼。把很多小叠的煎饼放在一起就可以变成一大叠煎饼。表达式也可以用同样的方式组合在一起形成更长的表达式。但不管一个表达式有多长它最终的运算结果同样只是一个单独的答案，比如 $2+5+8$ 的运算结果是 15。

在变量中保存值

我们编程的时候通常会希望保存我们的表达式的运算结果，这样我们就可以在以后使用它。我们把值存在**变量**中。

你可以把变量想象成可以装值的箱子。我们通过**赋值运算符**“=”来给变量赋值。例如，要把值 15 保存到名为“spam”的变量，只需输入 `spam = 15`。

```
>>> spam = 15
>>>
```

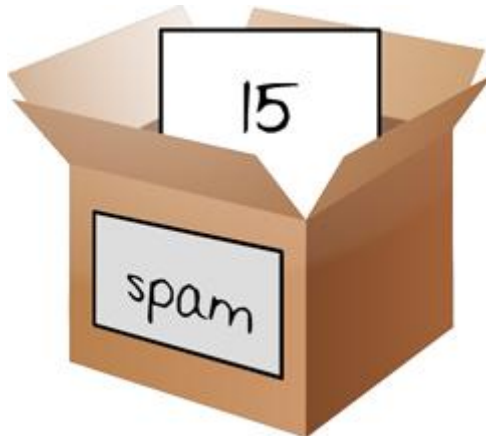


图 2-4: 变量就像可以装值的箱子

你可以把变量想象成一个装有 15 这个值的箱子（如图 2-4 所示）。变量名“spam”就像箱子上的标签（这样我们就可以将它和其他变量区分开来），而保存其中的值就像箱子里的便签条。

你按完回车后除了空行不会看到任何反应。除非你看到了错误信息，否则你就可以认为那条指令已经正确执行了。另一个 `>>>` 提示符会出现，这是提示你输入下一条指令的。

这条语句（被称为**赋值语句**）创建了变量 `spam` 并把值 15 保存到了变量中。与表达式不同的是语句是没有运算值的指令，这也是为什么交互界面中没有在下一行显示任何值的原因。

区分哪些指令是表达式，哪些指令是语句可能会让人感到困惑。你只要记住如果该指令可以运算并得到一个值那它就是表达式。如果没有值，那么就是语句。

一个赋值语句是由变量，等号（=）和表达式组成的。表达式运算得到的值保存在变量中。像 15 这样的值本身也是一个表达式。只有一个值（它本身）组成的表达式是很容易运算的。这些表达式的运算结果就是它们本身。例如，表达式 15 的运算结果是 15！

记住，变量中保存的是值，不是表达式。例如，如果我们有一个这样语句：`spam = 10 + 5`，那么表达式 `10 + 5` 会先被运算得到 15 然后再把值 15 保存到变量 `spam` 中。

你第一次用赋值语句给一个变量赋值的时候 Python 会自动创建那个变量。在此之后，再次给那个变量赋值的话只会把先前保存在变量中的值替换掉。

现在让我们来看看我们是否正确的创建了我们的变量。如果我们输入 `spam`，我们应该会看到保存在变量中的值。

```
>>> spam = 15
>>> spam
15
>>>
```

现在，`spam` 运算后的值是变量中保存的值 15。

给你看一个有趣又有点绕的例子。如果我们输入 `spam + 5`，我们会得到 20，如下所示：

```
>>> spam = 15
>>> spam + 5
20
>>>
```

这看起来可能有点怪，但如果我们记得我们赋了 15 这个值给 `spam` 的话就不会觉得怪了。因为我们把 15 赋给了 `spam`，所以写 `spam+5` 就和写 `15+5` 这个表达式是一样的。

如果你试图在创建一个变量之前使用它，Python 会给你一个错误信息，因为根本不存在这个变量。要是你把变量的名字打错了也会出错。

我们通过写另一个赋值语句的方式来改变我们保存在变量中的值。例如，你可以试试下面的代码：

```
>>> spam = 15
>>> spam + 5
20
>>> spam = 3
>>> spam + 5
8
>>>
```

我们第一次输入 `spam+5` 时这个表达式的运算结果是 20，因为我们保存在变量 `spam` 中的值是 15。但是，当我们输入 `spam = 3` 后，15 这个值就被替换掉了，或者说是被 3 重写了。现在我们输入 `spam+5`，这个表达式的运算结果是 8 因为 `spam` 现在的值是 3。

要想知道变量现在的值只需在交互界面中需输入变量名按回车即可。

现在我们来查看一些有趣的代码。由于变量只是值的名字，所以我们可以用变量写出像下面这样的表达式：

```
>>> spam = 15
>>> spam + spam
30
>>> spam - spam
0
>>>
```

当变量 `spam` 中保存的整型值是 15 时，输入 `spam + spam` 和输入 `15+15` 是一样的，得到的结果都是 30。而 `spam - spam` 和 `15-15` 也是一样的，得到的结果都是 0。上面的这些表达式都使用了 `spam` 这个变量两次。你可以在表达式中使用变量任意次。记住，每次你在 Python 中使用变量的时候，它都会对变量名进行运算并得到保存在变量中的值。

我们甚至可以使用保存在变量 `spam` 中的值给变量 `spam` 赋值：

```
>>> spam = 15
>>> spam = spam + 5
20
>>>
```

赋值语句 `spam = spam + 5` 的意思是：变量 `spam` 的新值是 `spam` 当前的值加上 5。记住，等号 (=) 右边的表达式运算得到的值将会被赋给等号左边的变量。我们也可以通过重复多次给 `spam+5` 的方式来不断地增加 `spam` 的值。

```
>>> spam = 15
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam
30
>>>
```

使用多个变量

我们编程的时候肯定不想被限制只能使用一个变量。通常我都会需要多个变量。

例如，让我们分别给两个名为 `eggs` 和 `fizz` 的变量赋值，像这样：

```
>>> fizz = 10
>>> eggs = 15
```

现在变量 `fizz` 的值是 10，`eggs` 的值是 15。

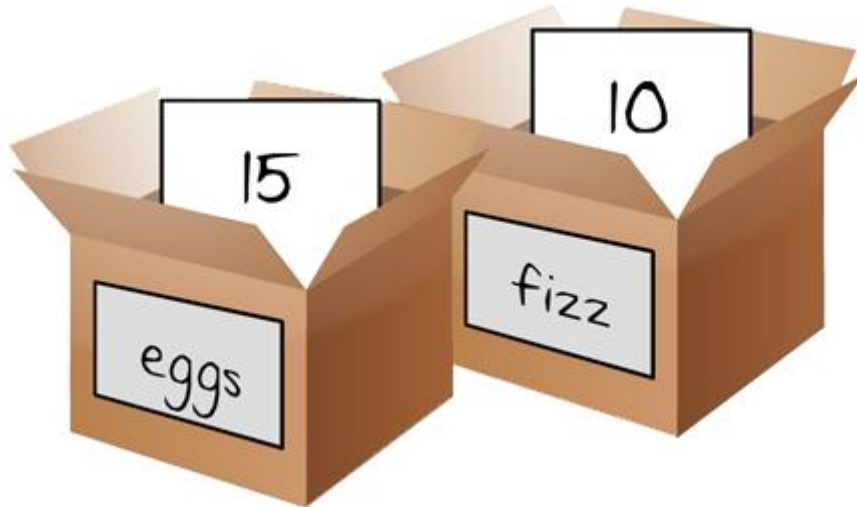


图 2-5: 变量“fizz”和“eggs” 的值

这次我们不改变变量 `spam` 的值，我们重新赋一个新的值给它。输入 `spam = fizz + eggs` 按回车，然后输入 `spam` 按回车来查看 `spam` 的新值。你能猜到是什么吗？

```
>>> fizz = 10
>>> eggs = 15
>>> spam = fizz + eggs
>>> spam
25
>>>
```

变量 `spam` 现在的值是 25 因为我们把 `fizz` 和 `eggs` 相加的时候，其实是把 `fizz` 和 `eggs` 中的值相加了。

重写变量

改变变量中保存的值很容易。只要对该变量执行另一条赋值语句即可。让我们看看如果我们如下代码会发生什么：

```
>>> spam = 42
>>> print(spam)
42
>>> spam = 'Hello'
>>> print(spam)
Hello
```

最初，我们把 42 保存到变量 spam 中了。这就是为什么输入 print(spam) 时会输出 42。但是当我们执行 spam = 'Hello' 这条语句时 42 就被字符串“Hello”给替换掉了。

用新的值替换变量中当前的值被称为**重写变量**。值得注意的是旧的值永远不存在了。如果你想把旧的值保存起来以备以后使用，那么你就要在重写前把它保存在另一个变量中。

```
>>> spam = 42
>>> print(spam)
42
>>> oldSpam = spam
>>> spam = 'Hello'
>>> print(spam)
Hello
>>> print(oldSpam)
42
```

在上面的例子中，在重写变量 spam 的值之前我们把它当前的值复制到了一个叫做 oldSpam 的变量中。在这个时间点上 spam 和 oldSpam 保存的值都是 42。在下一行代码中字符串'Hello'被保存到了 spam 而 oldSpam 则保持不变。

总结

在这一章你学了写 Python 指令的基础。Python 需要你严谨的告诉它要干什么，因为计算机没有常识，并且只能理解一些简单的指令。你知道了 Python 可以运算表达式（即把表达式简化为一个值），以及表达式是由值（如 2 或 5）和运算符（+或-）组成的。你也学到了可以把值保存到变量中以备以后在程序中使用。

在下一章，我们会学习更多基本概念，然后你就可以写你的第一个程序了！



第三章 字符串

本章内容：

- 流程
- 字符串
- 字符串的连接
- 数据类型（如字符串和整数型）
- 用 IDLE 写源代码
- 在 IDLE 中保存和运行程序
- `print()` 函数.
- `input()` 函数
- 注释
- 大写变量名
- 大小写敏感

关于整数型和数学我们已经知道得够多了。Python 不仅仅是计算器。所以现在就让我们来看看 Python 是如何处理文本的。在这一章，我们将学习如何把文本保存到变量中，连接文本以及在屏幕上显示它们。我们的大多数游戏都要向游戏玩家显示文本，同时也需要我们的玩家通过键盘输入文本。我们也会在这一章编写第一个程序，这个程序会在屏幕上显示“Hello World!”并询问玩家的名字。

字符串

在 Python 中，我们把小块文本称为**字符串**。我们可以把字符串保存在变量中就像我们可以把数字保存在变量中一样。我们输入字符串的时候把它们放在两个单引号（'）之间，像这样：

```
>>> spam = 'hello'  
>>>
```

单引号的作用是告诉电脑字符串从哪开始到哪结束（它不是字符串的一部分）。

现在，如果你输入 `spam`，你应该会看到变量 `spam` 的内容（字符串 `'Hello'`）。这是因为 Python 会输出变量中保存的值（在这里是字符串 `'Hello'`）。

```
>>> spam = 'hello'
>>> spam
'hello'
>>>
```

字符串几乎可以包含键盘上的任何字符（如果没有使用转义字符就无法在字符串中包含单引号。我们会在以后讨论转义字符。）下面都是字符串的例子：

```
'hello'
'Hi there!'
'KITTENS'
'7 apples, 14 oranges, 3 lemons'
'Anything not pertaining to elephants is irrelephant.'
'A long time ago in a galaxy far, far away...'
'O*&#wY%*&OCfsdYO*&gfC%YO*&%3yc8r2'
```

正如在上一章中我们可以把数字值组合在一起一样，我们也可以使用操作符把字符串组合在一起形成表达式。

字符串的连接

你可以使用 `+` 这个操作符把一个字符串连接到另一个字符串的后面，这叫做字符串的连接。你可以在交互界面输入 `'Hello' + 'World!'` 试试看：

```
>>> 'Hello' + 'World!'
'HelloWorld!'
>>>
```

为了让字符串保持合理的距离，我们可以在 `'Hello'` 这个字符串后面的那个单引号前加一个空格，如下所示：

```
>>> 'Hello ' + 'World!'
'Hello World!'
>>>
```

操作符 `+` 对字符串和整数型的作用是不同的，因为它们是不同的数据类型。所有的值都有一种数据类型。值 `'Hello'` 的数据类型是字符串。值 `5` 的数据类型则是整数型。数据的数据类型告诉我们（和电脑）该值是什么样的数据。

在 IDLE 的编辑器中写程序

到目前为止，我们都是交互界面中每次输入一条指令。但是在我们实际编程中需要一次输入多条指令并让它们一次性运行完。让我们开始写我们的第一个程序吧！

给我们提供交互界面的程序叫做 IDLE（Interactive DeveLopement Environment, 交互式开发环境）。IDLE 还有另一部分叫做文件编辑器(file editor)。

点击在 Python Shell 最上面的 **File** 菜单，然后选择 **New Window**。这时会出现一个让我们写代码的空白窗口。这就是**文件编辑器**。

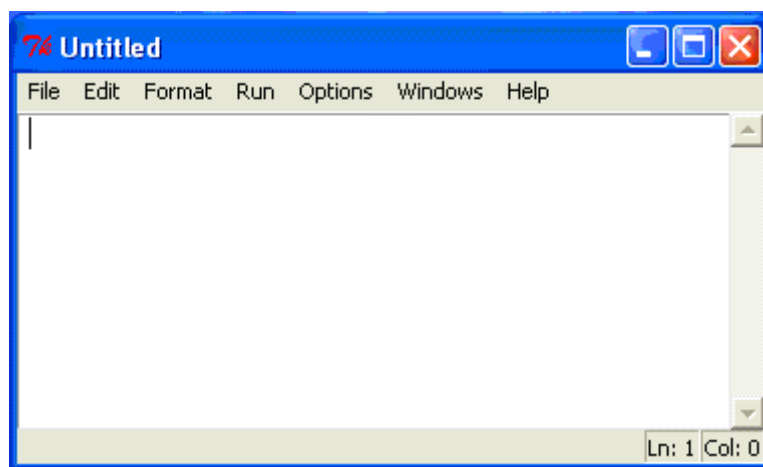


图 3-1: 文件编辑器

Hello World!



图 3-2: 文件编辑器窗口的右下角显示了我们的光标的位置。

现在光标在第 12 行。

程序员们学习一门新的语言的时候有一个传统，那就是把在屏幕上显示“Hello world!”的程序作为第一个程序。现在我们要写自己的 Hello World 程序了。

输入代码的时候不要把左边的行号也输入进去。行号的作用是方便我们在解

释代码的时候快速指向该行代码。你可以看一下文件编辑器的右下角，那里会显示你的光标当前所在的行号。

将下面的文本输入到文件编辑器。我们把这些文本叫做程序的源代码因为它包含了决定程序如何在 Python 中执行的指令。（记住，不要输入行号！）

注意！ 以下程序要在 Python 3 解释器下运行，不是 Python 2.6（或者其它 2.X 版本）。你要确认你安装的 Python 的版本是正确的。（如果你已经安装了 Python 2，你可以同时安装 Python 3。）如需下载和安装 Python 3，可访问 <http://python.org/download/releases/3.1.1/>。

hello.py

此代码可从 <http://inventwithpython.com/hello.py> 下载。

如果你输入代码时出现了错误，可以使用在线 diff 工具

（<http://inventwithpython.com/diff>）将你的代码和书中的作对比，你也可以给本书的作者发邮件（al@inventwithpython.com）。

```
1. # This program says hello and asks for my name.  
2. print('Hello world!')  
3. print('What is your name?')  
4. myName = input()  
5. print('It is good to meet you, ' + myName)
```

IDLE 会用不同的颜色标示不同的指令。你输入完所有代码后，你的窗口应该是这样的：

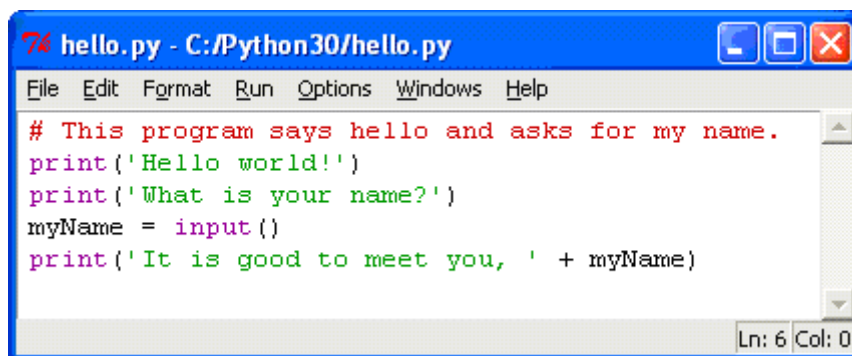


图 3-3：输入完代码后文件编辑器的样子

保存你的程序

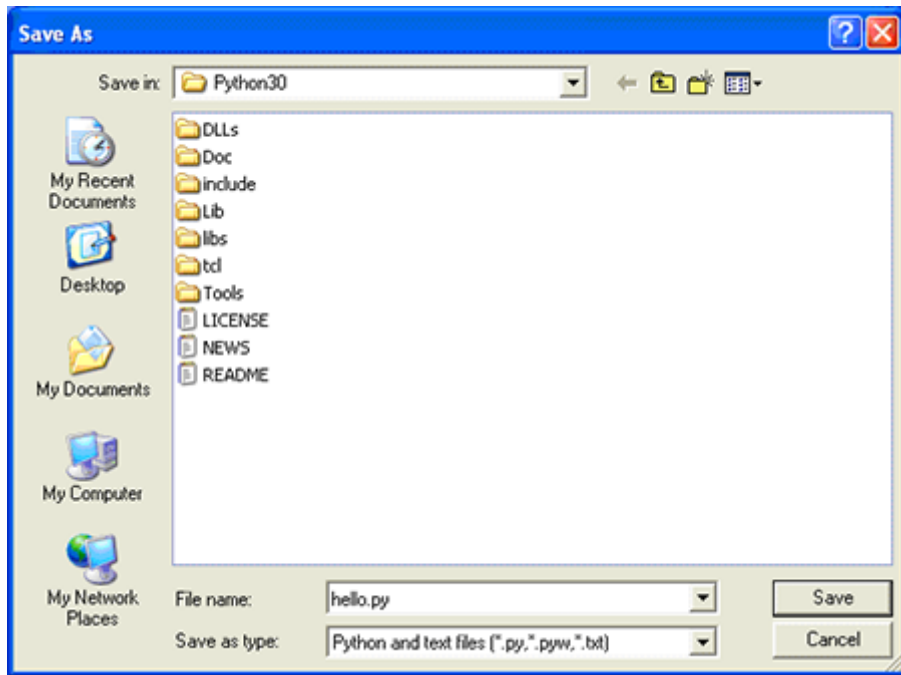


图 3-4:保存程序

当你完成了代码的输入后要把它保存起来，这样你就不用每次启动 IDLE 都要重新输入。保存的方法是在文件编辑器的最顶上的菜单栏选择 File，然后点击 **Save As**。这时会弹出一个另存为的窗口。在文件名的输入框中输入 `hello.py` 然后点击**保存**。（见图 3-4）

你应该每写一些代码就保存一次。这样的话，即使电脑崩溃导致你意外退出 IDLE 也只有你在最后一次保存后输入的代码会丢失。按 `Ctrl-S` 可以快速保存文件，根本不用鼠标。

本书的网站上有关于如何使用文件编辑器的视频教程。网址：<http://inventwithpython.com/videos/>

如果你的得到这样的错误信息：

```
Hello world!  
What is your name?  
Albert
```

```
Traceback (most recent call last):  
  File "C:/Python26/test1.py", line 4, in <module>  
    myName = input()  
  File "<string>", line 1, in <module>  
NameError: name 'Albert' is not defined
```

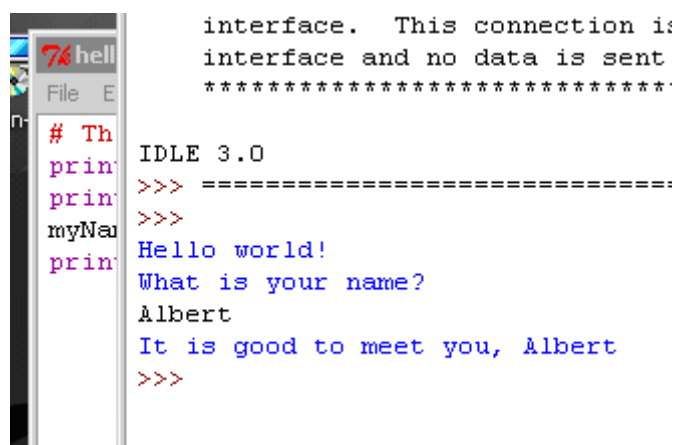
.....这说明你在 Python 2 下运行此程序而不是 Python 3。你可以选择安装 Python 3 或者把本书中的代码转换成在 Python 2 下可运行的代码。附录 A 列出了你使用本书时需要知道的 Python 2 和 Python 3 的区别。

打开你保存的程序

如果想打开已保存的程序,选择 **File > Open**。选择你要打开的文件(hello.py) 点击打开按钮。你保存的程序 hello.py 现在应该在文件编辑器中打开了。

现在我们要运行我们的程序了。从菜单中选择 **File**, 然后选择 **Run > Run Module** 或者直接按键盘上的 **F5**。这时你应该会看到你的程序在 IDLE 中运行了。记住,你要在文件编辑器中按 **F5**, 而不是在交互界面中。

当你的程序让你输入你的名字时,输入你的名字如图 3-5 所示:



```
interface. This connection is:
interface and no data is sent
*****

IDLE 3.0
>>> =====
>>>
Hello world!
What is your name?
Albert
It is good to meet you, Albert
>>>
```

图 3-5: 交互界面运行 Hello World 程序时的样子

现在,如果你按回车键你的程序应该会问候你(用户)。恭喜!你已经完成了你的第一个程序。你现在是一个初级程序员了。(如果你喜欢,你可以再运行一次这个程序,再按一次 **F5** 即可。)

“Hello World”程序是如何工作的

这个程序是怎么工作的呢?这么说吧,我们输入到电脑的每一条语句都会被 Python 解释为电脑能理解的语句。电脑程序就像一个食谱。第一步先做,然后是第二步,然后.....直到最后一步。每一个指示都有一定的顺序,从程序的最上面开始执行,一直到指示的最下面。程序执行完第一条指示后就执行第二条,然后第三条,等等。

我们把程序执行的顺序称为**流程**。

现在来看看程序的每一行代码都是干什么的，从第一行开始。

注释

```
1. # This program says hello and asks for my name.
```

这一行是**注释**。所有以符号#（读作**井字符**）开头的文本都是注释。注释不是写给电脑看的，而是写给像你这样的程序员看的。电脑会忽略它们。这是用来提示你程序的功能的，或是告诉那些看你的代码的人你的代码是干什么的。

程序通常在代码的最上面写一个说明程序名字的注释。**IDLE** 程序会用红色来显示注释以作区别。

函数

函数是你程序中的小程序。它包含从头执行到尾的代码。**Python** 提供了一些内建函数供我们使用。函数的好处是我们只需要知道它的功能，而不用理会它是如何实现。（你需要知道 **print()** 函数是用来在屏幕显示文本的，但不需要知道这是怎么实现的。）

调用函数是指用来一行代码来让我们的程序执行函数内部的代码。比如，你的程序可以在你想在屏幕上显示字符串的时候调用 **print()** 函数。输入到 **print()** 函数括号中的内容将会被显示在屏幕上。如果我们想在屏幕上显示 **Hello World!**，我们只需要输入函数名 **print** 紧接着输入左括号，然后输入字符串 **'Hello World!'**，最后输入右括号。

print () 函数

```
2. print('Hello world!')
3. print('What is your name?')
```

这行代码调用了 **print** 函数通常写作 **print()**（要输出字符串写在括号内）

我们在函数名字的后面加上括号是为了清楚的说明我们是指一个名为 **print()** 的函数，而不是名为 **print** 的变量。函数后面的括号让我们知道我们是在讨论函数，这有点像数字 **'42'** 两边的单引号，它说明我们是在讨论字符串 **'42'** 而不是整数 **42**。

第 3 行再一次调用了 **print()** 函数。这次程序输出了 **“What is your name?”**。

input() 函数

```
4. myName = input()
```

这行代码包括一个带变量（myName）的赋值语句和一个函数调用语句（input()）。当input()函数被调用时，程序会等待用户输入文本。用户输入的文本字符串（你的名字）就是函数的输出值。

与表达式一样，函数调用语句也会运算然后得到一个值。函数运算得到的值叫做**返回值**。（事实上，我们也可以用“返回”来表示“运算”。）在这个例子中input()函数的返回值是用户输入的字符串——他们的名字。如果用户输入Albert，那么input()函数运算得到的值是字符串'Albert'。

input()函数不需要在它后面的括号中输入内容（不像print()函数），这就是为什么它括号里没内容的原因。

```
5. print('It is good to meet you, ' + myName)
```

在最后一行代码中我们又遇到了print()函数。这次我们用加法操作符(+)来连接字符串'It is good to meet you, '和由用户输入并保存在变量myName中字符串。这就是我们让程序用名字来问好我们的过程。

结束程序

当我们的程序执行到最后一行代码时就停止了。此时程序已经**终止**或者说**退出了**，所有的变量都失效了，包括我们保存在myName中的字符串。如果你尝试用不同的名字再一次运行程序，如Carolyn，它会认为那是你的名字。

```
Hello world!  
What is your name?  
Carolyn  
It is good to meet you, Carolyn
```

记住，电脑只按照你编的程序去执行。在我们的第一个程序中，我们编写了一个询问用户的名字，让用户输入字符串，然后说你好并显示用户输入的字符串的程序。

但是电脑是有点笨的。这个程序不管你输入的是你的名字，还是其他人的名字，或者其他东西。你可以输入任何你想输入的内容，电脑会用相同的方式对待它们。

```
Hello world!  
What is your name?  
poop  
It is good to meet you, poop
```

变量名

虽然电脑不在乎你给你的变量取什么名字，但是你要在乎。给变量取一个可以反映它包含数据的类型可以让你的程序更好理解。我们可以把变量命名为 `abrahamLincoln` 或 `nAmE`。电脑同样正常可以运行该程序。（只要你一直使用 `abrahamLincoln` 或 `nAmE`）

变量名（Python 的其他方面也一样）是大小写敏感的。**大小写敏感**是指相同的变量名如果大小写不同就会被认为是完全不同的变量。因此，在 Python 中 `spam`，`SPAM`，`Spam` 和 `sPAM` 会被认为是四个不同的变量。它们可独立保存不同的值。

用相同的变量名，不同的大小写组合来命名变量是很不好的做法。如果你把一个人的名保存在变量 `name` 中，把姓保存在变量 `NAME` 中，当你几个星期后再看你的代码时你会觉得很困惑。变量 `name` 指名和变量 `NAME` 指姓，还是反过来？

如果你不小心把变量 `name` 和 `NAME` 对换了，你的程序会正常运行（即没有语法错误），但是运行结果是错误的。这种代码缺陷叫做**漏洞(bug)**。写代码的时候意外导致程序有漏洞是很常见的。这就是为什么选择有意义的变量名很重要的原因。

如果变量名包含一个以上的单词，最好从第二个单词开始大写第一个字母。如果你用一个变量来保存你早餐所吃的东西，把该变量命名为 `whatIHadForBreakfastThisMorning` 会比 `whatihadforbreakfastthismorning` 更具可读性。这是用 Python 编程的一个**传统**（做事的可选的惯例）。当然，起名字要越简单越好，比如 `todayBreakfast`。变量中每个单词的首字母大写可以让程序读起来更容易。

总结

既然我们已经学会了如何处理文本，那就可以开始编写可以与用户互动的程序了。这之所以重要是因为文本是电脑和用户交流的主要方式。玩家借助 `input()` 函数通过键盘向程序输入文本。当 `print()` 函数被执行时，电脑将会在屏幕上显示文本。

字符串是我们可以使用另一种数据类型。我们可以使用操作符 `+` 来连接字符串。使用操作符 `+` 来连接两个字符串以形成新的字符串和用操作符 `+` 把两个整数相加得到新的整数（和）是一样的。

在下一章中，我们将会学习更多关于变量的内容。这样我们的程序才可以记住玩家输入的文本和数字。当我们学会如何使用文本，数字和变量之后，就可以开始编写游戏了。



第四章

猜数字游戏

本章内容:

- import 语句
- 模块
- 参数
- while 语句
- 条件
- 块
- 布尔值
- 比较运算符
- =和==的区别
- if 语句
- 关键字 break
- str()函数和 int()函数
- random.randint()函数

猜数字游戏

我们准备编写一个猜数字的游戏。在这个游戏中，计算机会从 1 到 20 中产生一个随机数，然后让你去猜那个数字。你只有 6 次机会，不过计算机会提示你猜的数字是大了还是小了。如果你在 6 次内猜中那个数字你就赢了。

这是一个很好的游戏，因为它在相对短的代码中使用了随机数，循环和用户输入。你编写这个游戏的时候将会学到如何将值转换成不同的数据类型（以及为什么要这么做）。

由于本程序是一个游戏，所以我们会把用户称作玩家，不过“用户”这个词也对。

“猜数字”运行演示

这是这个程序运行时的样子。玩家输入的文本用**粗体**显示。

```
Hello! What is your name?  
Albert  
Well, Albert, I am thinking of a number between 1 and 20.  
Take a guess.  
10  
Your guess is too high.  
Take a guess.  
2  
Your guess is too low.  
Take a guess.  
4  
Good job, Albert! You guessed my number in 3 guesses!
```

准确的把下面的代码输入到计算机，并点击 **File** 菜单然后选择 **Save As**，把它保存为 *guess.py* 然后按 F5 运行。如果你现在不理解代码，不用担心，我稍后会解释。

猜数字游戏的源代码

下面的代码是猜数字游戏的源代码。你输入这些代码的时候要特别注意某些代码行前面的空格。有些代码行前面有 4 或 8 个空格。完成输入之后，把它保存为 *guess.py*。你可以从文件编辑器运行该程序，按 F5 即可。如果你遇到错误信息，检查一下你输入的内容是否和书中的一致。

如果你不想自己输入这些代码，你可以从本书的官方网站下载。网址：
<http://inventwithpython.com/chapter4>。

注意！ 确定你是在 Python 3 而不是 Python 2 下运行你的程序。本书使用的是 Python 3，如果你在 Python 2 下运行会得到错误信息。想知道自己的 Python 的版本可以点击 **Help** 然后 **About IDLE** 查看。

guess.py

这些代码可以从 <http://inventwithpython.com/guess.py> 下载

如果输入完后得到错误信息，可以用在线 diff 工具(<http://inventwithpython.com/diff>)把你输入的代码和书中的代码对比，或者给作者发邮件 al@inventwithpython.com

```
1. # This is a guess the number game.  
2. import random
```

```

3.
4. guessesTaken = 0
5.
6. print('Hello! What is your name?')
7. myName = input()
8.
9. number = random.randint(1, 20)
10.    print('Well, ' + myName + ', I am thinking of a number between
      1 and 20.')
11.
12.    while guessesTaken < 6:
13.        print('Take a guess.') # There are four spaces in front
      of print.
14.        guess = input()
15.        guess = int(guess)
16.
17.        guessesTaken = guessesTaken + 1
18.
19.        if guess < number:
20.            print('Your guess is too low.') # There are eight
      spaces in front of print.
21.
22.            if guess > number:
23.                print('Your guess is too high.')
24.
25.            if guess == number:
26.                break
27.
28.    if guess == number:
29.        guessesTaken = str(guessesTaken)
30.        print('Good job, ' + myName + '! You guessed my number
      in ' + guessesTaken + ' guesses!')
31.
32.    if guess != number:
33.        number = str(number)
34.        print('Nope. The number I was thinking of was ' + number)

```

即使我们往文件编辑器输入代码的时候也可以回到交互界面去输入单个指令，这样可以知道每个指令是干什么的。交互界面是一个做实验的好地方。想回到交互界面只需点一下它的窗口，或者在任务栏点一下它。在 Windows 或 Mac OS X 下任务栏在屏幕的最下面。在 Linux 下任务栏应该是在屏幕的最上面。

如果你输入完后程序不能运行，检查一下你输入的内容是否和书中的一致。你也可以把你输入的代码复制到在线 diff 工具(<http://inventwithpython.com/diff>)。

这个工具会给出你输入的代码与书中代码的区别。在文件编辑器中按 **Ctrl-A** “全选” 你输入的文本，然后按 **Ctrl-C** 复制，最后把它粘贴到在线 **diff** 工具的文本框中点击 “**Compare**” 进行比较。这个网站会给出你的代码与书中代码的区别。

在本书网站（<http://inventwithpython.com>）有书中每个程序的 **diff** 工具。网站上也有关于如何使用 **diff** 工具的视频教程（<http://inventwithpython.com/videos/>）。

Import 语句

让我们一行一行的看一下这些代码，看这个程序是怎么工作的。

```
1. # This is a guess the number game.
```

这是一行注释。在第三章的 **Hello World** 程序中我们已经介绍了注释。记住 Python 会忽略#后面的所有内容。这个只是用来提示我们这个程序是干什么的。

```
2. import random
```

这是一个 **import 语句**。语句不是函数（注意：不管是 **import** 还是 **random** 后面都没有括号）。语句就是执行一些动作，但是没有返回值的指令。你已经接触的语句有：赋值语句，这个语句使用来保存一个值到变量的（但是语句不返回任何值）。

Python 自带了很多内建函数，还有一些分布在不同程序中的函数叫做模块。**模块**也是一种 Python 程序。我们如果想使用模块中的函数要先用 **import** 语句把模块导入到我们的程序中。在这个例子中，我们导入的是 **random** 模块。

import 语句由关键字 **import** 和模块名组成。第 2 行代码就是一个 **import** 语句，这个语句导入了一个叫做 **random** 的模块，这个模块中有一些和随机数相关的函数。（后面我们希望计算机产生随机数让我们猜的时候会用到这些函数。）

```
4. guessesTaken = 0
```

这行创建了一个名为 **guessesTaken** 的变量。我们会把用户猜的次数保存到这个变量。由于玩家还没有猜，所以我们保存了一个整数 0。

```
6. print('Hello! What is your name?')
7. myName = input()
```

第 6 行和第 7 行和我们在第 3 章的 **Hello World** 程序中看到的一样。如果程序员们之前为同一个功能写过代码，他们通常会重用这些代码。

第 6 行调用了 `print()` 函数。函数其实就是一些在我们的程序内部运行的小程序，当我们的程序调用它时，这个小程序就会被运行。`print()` 函数内部的代码会把你输入到括号内的内容显示在屏幕上。

当这两行运行结束时，玩家的名字，即玩家输入的字符，会被保存到变量 `myName` 中。（注意，那个字符串不一定是玩家的名字。玩家输入什么计算机就显示什么。）

random.randint()函数

```
9. number = random.randint(1, 20)
```

在第 9 行我们调用了一个名为 `randint()` 的新函数，并且把函数的返回值保存到名为 `number` 的变量中。函数调用属于表达式，因为它有运算值。我们把这个值称为函数的返回值。

因为 `randint()` 函数是由 `random` 模块提供的，所以我们要在 `randint()` 前加上 `random.`（点不能漏！）来告诉程序它在 `random` 模块中。

`randint()` 函数会返回一个介于我们给出的两个整数（包括这两个数）之间的一个随机整数。在这个例子中我们在函数名后面的括号中输入了整数 1 和 20，并用逗号隔开。`randint()` 函数返回的值保存在名为 `number` 的变量中，这个数就是玩家要猜的数。

等一下，现在我们先回到交互界面，输入 `import random` 来导入 `random` 模块。然后输入 `random.randint(1,20)`，看看函数的返回值是什么。它应该返回一个介于 1 到 20 之间的整数。现在再输入一次一样的代码，它应该会返回一个不同的整数。这是因为每调用一次 `randint()` 函数它就返回一个随机数，就像你每扔一次骰子就会得到一个随机数一样。

```
>>> import random
>>> random.randint(1, 20)
12
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
3
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
7
>>>
```

只要我们在游戏中想使用随机数就可以用 `randint()` 这个函数。而且我们会在很多游戏中使用随机数。（想一下有多少棋盘游戏使用骰子。）

你也可以通过改变函数的参数来改变随机数的范围。比如，输入 `random.randint(1,4)` 你就会得到介于 1 和 4（包括 1 和 4）之间的数。或者试一下输入 `random.randint(1000,2000)` 以得到介于 1000 和 2000 之间的数。下面是一个调用 `random.randint()` 函数的例子，让我们看一下它会返回什么值。你得到的值可能会和下面的不同，毕竟它们是随机数。

```
>>> random.randint(1, 4)
3
>>> random.randint(1, 4)
4
>>> random.randint(1000, 2000)
1294
>>> random.randint(1000, 2000)
1585
>>>
```

我们稍微修改一下游戏的代码就可以让游戏有不同的表现。试着把第 9 行和第 10 行：

```
9. number = random.randint(1, 20)
10. print('Well, ' + name + ', I am thinking of a number between
    1 and 20.')
```

改成这样：

```
9. number = random.randint(1, 100)
10. print('Well, ' + name + ', I am thinking of a number between
    1 and 100.')
```

现在计算机将给出介于 1 和 100 之间的一个整数。改变第 9 行就可以改变随机数的范围，但是要记得改变第 10 行，这样玩家才能知道新的范围。

调用模块内的函数

顺便说一下，一定要输入 `random.randint(1,20)`，而不是 `randint(1,20)`，否则计算机将不知道要在 `random` 模块中寻找 `randint()` 函数，你会得到这样的错误信息：

```
>>> randint(1, 20)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'randint' is not defined
>>>
```

记住，在调用 `random.randint()` 函数前要先导入 `random` 模块。这就是为什么 `import` 语句通常程序的最来头的原因。

将参数传递给函数

在函数 `random.randint(1,20)` 括号内的数值叫做参数。**参数**是函数被调用时，传递给函数的值。参数决定函数的行为。就像玩家的输入改变我们的程序的行为一样，参数是改变函数行为的输入。

有些函数要求你在调用它们时要给它们传参数。看看这些例子：

```
input()
print('Hello')
random.randint(1, 20)
```

`input()` 函数没有参数，但是 `print()` 函数有一个参数，`randint()` 则有两个。如上面的代码所示，当有多于一个参数时我们用逗号来分隔。程序员们会说参数由逗号**界定**（即分隔）。这是计算机识别一个值的结束和另一个值的开始的根据。

如果你在调用函数时传的参数比要求的要多或少，`Python` 会给出如下错误信息：

```
>>> random.randint(1)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    random.randint(1)
TypeError: randint() takes exactly 3 positional arguments (2
given)
>>> random.randint(1, 2, 3)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    random.randint(1, 2, 3)
TypeError: randint() takes exactly 3 positional arguments (4
given)
>>>
```

在这个例子中，我们先调用了 `randint()` 函数并只传了一个参数（太少），然后再调用了 `randint()` 函数并传了三个参数（太多）。

注意，错误信息提示我们说我们传递了 2 个和 4 个参数，而我们只传 1 个和 3 个。这是因为 Python 总会传一个额外且不可见的参数。关于这个参数的知识超出了本书的范围，你也不用在意。

欢迎界面

第 10 行和第 12 行显示一些欢迎玩家的信息并告诉他们游戏的相关信息，然后就开始让玩家开始猜数字了。第 10 行代码很简单，但是第 12 行介绍了一个很有用的一个概念叫做循环。

```
10.     print('Well, ' + myName + ', I am thinking of a number between  
        1 and 20.')
```

在第 10 行代码中，`print()`函数欢迎了一下玩家，然后告诉玩家计算机在想一个随机数。

但是，等一下，我不是说 `print()`函数只能接受一个字符串吗？那看起来可能像是有很多个字符串。但是如果你仔细看，你就会发现字符串之间用加号连接在一起了，这样三个字符串就变成了一个字符串。这个字符串就是 `print()`函数在屏幕上显示出来的那个。那些逗号看起来像是用来分隔字符串，但是如果你仔细观察，你会发逗号是在引号内的，它们是字符串的一部分。

循环

第 12 行代码有一个 `while` 语句，这标志着 `while` 循环的开始。**循环**是指某一部分代码的重复执行。但是在我们学习 `while` 循环之前，我们需要先了解一些概念。这些概念包括块，布尔值，比较运算符，条件以及 `while` 语句。

块

块是指拥有相同缩进的组合在一起的一行或多行代码。你可以通过代码行的**缩进**（指代码行前面的空格数）来判断一个块的开始与结束。

一个块的开始通常有四个空格的缩进。其他所有拥有相同空格数缩进的都属于这个块。块中块通常以另外四个空格的缩进为开始（总共有八个空格）。如果有与块的开头相同缩进语句出现，说明块结束了。

下图用数字标出了块。空格则用黑点填充，这样方便计数。

```

12. while guessesTaken < 6:
13. ....print('Take a guess.')
14. ....guess = input()
15. ....guess = int(guess)
16.
17. ....guessesTaken = guessesTaken + 1
18.
19. ....if guess < number:
20. ....    ....print('Your guess is too low.')
21.
22. ....if guess > number:
23. ....    ....print('Your guess is too high.')

```

图 4-1: 块及其缩进。黑点代表空格

例如，我们可以看看图 4-1 中的代码。空格已用黑点替换，以便计数。第 12 行的缩进为零个空格所以它不属于任何块。第 13 行有四个空格的缩进。由于这个缩进大于上一行的缩进，所以我们可以知道，这是一个新的块的开始。第 14 行，15，17 和 19 行也有四个空格的缩进。这些行的缩进与上一行的数量相同，所以我们知道他们是属于同一个块的。（当我们找缩进时忽略空白行）

第 20 行有 8 个空格的缩进。8 个空格大于四个空格，所以我们知道一个新的块已经开始。这是一个块内的另一个块。

第 22 行只有四个空格，少于前一行的空格数。由缩进有所减少可知该块已经结束。第 22 行与其他同样有四个空格的缩进的代码行属于同一个块。

第 23 行的缩进增加到了 8 个空格，所以又开始了一个新的块。

总的来说，第 12 行是不属于任何块的。第 13 至 23 行在一个块中（用方框 1 圈出）。第 20 行属于块中的另一个块（用方框 2 标出）。第 23 行也是属于块中的另一个块（用方框 3 标出）。

在 IDLE 中，每个字母的是宽度相同的。你可以通过数上一行代码的字符数来确定本行代码的缩进的空格数。

在这个图中，方框 1 中的代码都属于同一个块，块 2 和块 3 则在块 1 内。块 1 的缩进为 4 个空格，而块 2 和块 3 的缩进是 8 个空格。块可以只由一行代码组成。你可以看到块 2 和块 3 都只有一行代码。

布尔数据类型

布尔数据类型只有两个值：**True** 或 **False**（真或假）。这些值是对大小写敏感的，并且它们不是字符串，也就是说，不能在它们两边输入引号。我们会使用布尔值来构成条件语句（下面会说到）。

比较运算符

在第 12 行代码中我们用到了 **while** 语句。

```
12. while guessesTaken < 6:
```

紧跟在关键字 **while** 后面的表达式 (`guessesTaken < 6`) 包含两个值（变量 `guessesTaken` 中的值和整数值 6）。这两个值由一个运算符 (`<`，这是“小于”的意思) 连接。“`<`”这个符号被称为**比较运算符**。

比较运算符的作用是比较两个值并得到一个布尔值：**True** 或 **False**。表 4-1 列出了所有的比较运算符。

表 4-1: 比较运算符

运算符	运算符的名称
<code><</code>	小于
<code>></code>	大于
<code><=</code>	小于等于
<code>>=</code>	大于等于
<code>==</code>	等于
<code>!=</code>	不等于

条件

条件是指通过比较运算符（如`<`或`>`）连接两个值然后运算得到一个布尔值的表达式。条件不过是能够运算到 **True**（真）或 **False**（假）的表达式的一个名称。在表 4-1 中列出了其他比较运算符。

条件总是会运算得到一个布尔值：**True**（真）或 **False**（假）。例如，我们代码中的条件，`guessesTaken < 6` 的意思是“`guessesTaken` 中保存的值是小于 6 吗？”如果是，则表达式运算的结果是 **True**，如果不是结果是 **False**。

在猜数字游戏这个程序中，我在第 4 行代码把值 0 保存到了 `guessesTaken`。因为 0 小于 6，所以这个表达式得到的结果是布尔值 **True**。记住，条件不过是表达式的另一个名字，只是这些表达式用到了像`<`或`!=`这样的比较运算符。

用布尔值，比较运算符和条件做实验

把下面的表达式输入到交互界面，看看它们的布尔值：

```
>>> 0 < 6
True
>>> 6 < 0
False
>>> 50 < 10
False
>>> 10 < 11
True
>>> 10 < 10
False
```

条件 $0 < 6$ 返回的布尔值是 `True`，因为 0 小于 6。但由于 6 不小于 0，所以条件 $6 < 0$ 返回 `False`。50 不小于 10，所以 $50 < 10$ 返回 `False`。10 小于 11，所以 $10 < 11$ 结果是 `True`。

但是为什么 $10 < 10$ 的结果是 `False`？因为数字 10 不小于数字 10。它们是相等的。如果一个叫做爱丽丝的女孩和一个叫做鲍勃的男孩一样高，你不会说爱丽丝比鲍勃高或者鲍勃比爱丽丝高。这个两个语句都是假的。

现在再试一些条件，看看这些比较运算符的工作方式：

```
>>> 10 == 10
True
>>> 10 == 11
False
>>> 11 == 10
False
>>> 10 != 10
False
>>> 10 != 11
True
>>> 'Hello' == 'Hello'
True
>>> 'Hello' == 'Good bye'
False
>>> 'Hello' == 'HELLO'
False
>>> 'Good bye' != 'Hello'
True
```

请注意赋值运算符(=)和“相等”比较运算符(==)的区别。等号(=)用来给变量赋值，而“相等”(==)则是用来比较表达式中的两个值是否相等。这两个很容易在使用中混淆，所以输入的时候要特别注意。

两个不同数据类型的值**永远**不会相等。比如，在交互界面输入如下代码：

```
>>> 42 == 'Hello'
False
>>> 42 != '42'
False
```

使用 while 循环语句

while 语句标志着一个循环的开始。有时候我们想让程序重复做同样的事情。当程序执行到 while 语句时，它会判断 while 关键字后面的条件。如果条件为 True，那么执行 while 内部的循环体。（在我们的程序中 while 循环体开始于第 13 行。）如果条件为 False，则会跳过循环体。（在我们的程序中循环体后的第一行代码是第 28 行。）

一个 while 语句的条件后面永远都有分号(;)。

```
12. while guessesTaken < 6:
```

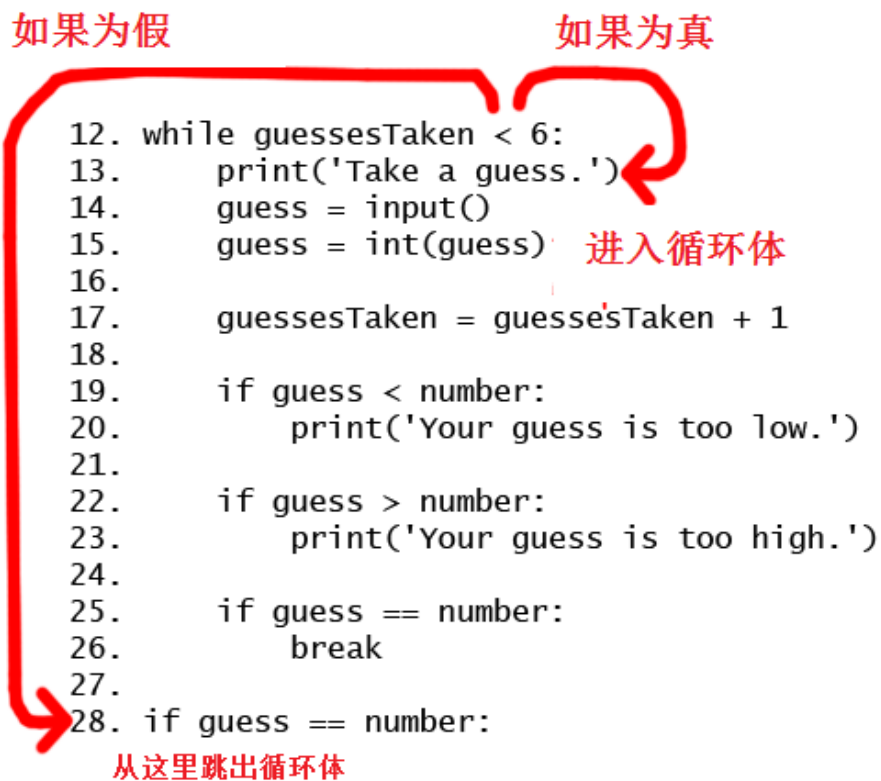


图 4-2: while 循环的条件

图 4-2 给我们展示了程序执行的流程。如果条件为 `True`（第一次为真，因为 `guessesTaken` 的值是 0），程序会在第 13 行进入循环体，然后一直执行。一旦程序到达循环体的最后，它不是继续执行下一行，而是跳回 `while` 语句那一行（第 12 行）。然后它再判断条件的真假，如果仍然为 `True`，程序就会再一次进入循环体。

这就是循环的工作方式。只要条件为 `True`，程序就会一直执行，直到条件为 `False`。也就是说，只要 `guessesTaken` 是等于或者小于 6 的程序就会一直执行。

你可以这样理解 `while` 语句，“只要条件为 `True`，就一直执行循环体内的所有代码”。

你可以通过改变玩家猜测的次数来改变游戏的难度。只需改变下面这行

```
12. while guessesTaken < 6:
```

改成：

```
12. while guessesTaken < 4:
```

.....这样玩家就只有 4 次猜测机会而不是 6 次。把条件改成 `guessesTaken < 4`，我们确保了循环内的代码只会执行 4 次而不是 6 次。这会让游戏更难。想要把游戏变简单，只需把条件改成 `guessesTaken < 8` 或 `guessesTaken < 10`，这样会让程序多循环几次，玩家也就会有更多的猜测机会。

当然，如果我们删除 17 行（`guessesTaken = guessesTaken + 1`），那 `guessesTaken` 就不会增加那条件就永远为 `True`。这让玩家有无数次猜测机会。

玩家的猜测

第 13 行至 17 行请求玩家猜神秘数字，并然他们输入他们的猜测。我们把这个猜测保存在变量中，然后把字符串转换成一个整数。

```
13.     print('Take a guess.') # There are four spaces in front
      of print.
14.     guess = input()
```

程序现在请求玩家输入他们的猜测。我们输入猜测，这个数字会被保存在名为 `guess` 的变量中。

用 `int()` 函数把字符串转换成整数

```
15.     guess = int(guess)
```

在第 15 行,我们调用了一个叫做 `int()` 的新函数。`int()` 函数有一个参数。`input()` 函数返回的是玩家输入的值的字符串。但在我们的程序中,我们需要一个整数,而不是字符串。如果玩家输入 5 作为他们的猜测,`input()` 函数会返回字符串值 '5' 而不是整数值 5。记住,Python 认为字符串 '5' 和整数 5 是不同的。因此 `int()` 函数是用来处理我们传给它的字符串然后返回这个值的整数形式。

我们用 `int()` 函数在交互界面做做实验吧。输入如下代码:

```
>>> int('42')
42
>>> int(42)
42
>>> int('hello')
```

```
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
int('forty-two')
ValueError: invalid literal for int() with base 10: 'hello'
>>> int('forty-two')
```

```
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
int('forty-two')
ValueError: invalid literal for int() with base 10:
'forty-two'
>>> int(' 42 ')
42
>>> 3 + int('2')
5
```

我们可以看到 `int('42')` 会返回整数值 42, `int(42)` 也同样会返回相同的值。(虽然把整数转换整数这个做法有点无聊)。不过,虽说你可以把字符串传给 `int()` 函数,但你也不能真的传入字符串。比如,传 'hello' 给 `int()` 函数(就是 `int('hello')`)会导致错误。传给 `int()` 函数的字符串必须是由数字组成的。

我们传给 `int()` 函数的整数必须是数字不能是文字,就是说 `int('四十二')` 同样会导致错误。如果我们的字符串左右两边任意一边有空格是不会出错的,这么说来, `int()` 函数还是有点容忍度的。这就是为什么 `int('42')` 能正常运行的原因。

`3+int('2')` 这行展示了一个把整数 3 和 `int('2')` 的返回值相加的一的表达式。这个表达式等价于 `3+2`, 得到的结果是 5。因此,虽然我们不能把整数和字符串相加 (`3+'2'` 会出错), 但我们可以让转化成整数的字符串和整数相加。

记住,我们程序的后面的第 15 行中的变量 `guess` 原来保存的是用户输入的字符串类型的值。我们会用 `int()` 函数返回的整数值复写我们保存在 `guess` 中的值。

因为后面我们要用这个值去和计算机产生的随机数比较。我们只能比较两个整数的值的大小。我们不能判断整数的值是比字符串大还是小，即使字符串的值是像'5'这样的数字。

在猜数字游戏中，如果玩家输入的不是数字，那么 `int()` 函数会出错，而程序则会崩溃。在本书的其他游戏中，我们将会增加一些检查类似这样的错误的代码，并给玩家一个机会重新输入正确的内容。

注意，`int(guess)` 这句并不会改变 `guess` 的值。`int(guess)` 这行代码只是把保存在变量 `guess` 中的值转换成整数值。我们必须把它的返回值重新赋给变量 `guess`，这样才能改变 `guess` 中的值，也就是要这样：`guess = int(guess)`。

递增变量

```
17.         guessesTaken = guessesTaken + 1
```

一旦玩家猜了一次，我们就要增加玩家已经猜的次数。

我们第一次进入循环体时，`guessesTaken` 的值是 0。Python 会使用这个值并给它加 1。0+1 就是 1。然后 Python 会把新的值 1 保存到 `guessesTaken`。

你可以把第 17 行理解为，“`guessesTaken` 应该比现有的值大 1”。

当我们给整数值加 1 时，程序员们会说变量在**递增**（因为它增加了 1）。当我们把变量减小时，变量就在**递减**（因为它减小了 1）。下次循环体再循环时，`guessesTaken` 就会是 1 而且它的值会增加到 2。

if 语句

玩家猜的数是不是太小了？

第 19 和第 20 行检查玩家猜的数字是否比计算机随机产生的神秘数字要小。如果是，我们就在屏幕输出以下信息告诉他们猜的数太小。

```
19.         if guess < number:
20.             print('Your guess is too low.') # There are eight
                spaces in front of print.
```

第 19 行以 `if` 语句的开头关键字 `if` 开始。关键字 `if` 之后是条件。第 20 行以一个新的代码块开始，你可以看得出来（因为第 20 行的缩进比第 19 行增加了）。跟在关键字 `if` 后面的代码块叫做 `if` 代码块。如果你想在条件为 `True` 的时候执行一小段代码就可以用 `if` 语句。第 19 行有一个 `if` 语句，它的条件为 `guess < number`。

如果条件为真，if 代码块的代码会被执行，如果为假，那段代码则会被跳过。

```
if fizzy < 10:
    # 条件块
```

if 关键字 条件

```
while fizzy > 6:
    # 条件块
```

while 关键字 条件

图 4-3: if 和 while 语句

和 while 语句一样，if 语句同样由关键字，条件，分号，然后一个代码块组成。图 4-3 是这两语句的比较。

if 语句和 while 语句的原理几乎相同。但与 while 循环不同的是 if 语句执行完代码块后不会跳回代码的开始处，而是继续下一行代码。也就是说，if 语句不会循环。

如果条件为真，则 if 代码块内的所有代码都会被执行。第 19 行这个代码块的唯一一行代码是调用 print() 函数。

如果玩家输入的整数比计算机随机产生的要小，程序就会输出 Your guess is too low。如果相等或者大于或者相等（不管那种情况，if 关键字后面的条件都是假），因此这个代码块会被跳过。

玩家猜的数是不是太大了？

第 22 行和第 26 行检查玩家输入的整数是太大还是刚好和神秘数字相等。

```
22.         if guess > number:
23.             print('Your guess is too high.')
```

如果玩家的输入比随机整数大，程序就会进入 if 语句后面的代码块。print() 这一行会告诉玩家他的输入太大了。

用 break 语句提前跳出循环

```
25.         if guess == number:
26.             break
```

这个 `if` 语句的条件用来判断玩家猜的数字是否和随机数相等。如果是，进入第 26 行的 `if` 代码块。

这个代码块中的代码是一个 **break** 语句，这是用来让程序立刻跳出循环的。（`break` 语句不会再去检查 `while` 循环的条件，而是立刻跳出循环）。

`break` 语句只有关键字 `break` 本身，没有条件和分号。

如果玩家的输入与随机数不相等，程序不会直接跳出循环，而是会执行到 `while` 循环体的最后。到了循环体的最后之后，程序会回到循环的开始处然后在检查条件（`guessesTaken < 6`）。当 `guessesTaken = guessesTaken + 1` 这行代码执行完之后，`guessesTaken` 的新值是 1。因为 1 小于 6，所以程序又会进入循环体。

如果玩家猜的数字一直都太小或者太大，`guessesTaken` 就会变成 2，然后变为 3，然后 4，然后 5，最后变为 6。如果玩家猜中了，这个语句 `if guess == number` 的条件就会变成真的（`True`），那程序就会执行 `break` 语句。否则，程序会继续循环。但是当 `guessesTaken` 保存的值为 6 时，`while` 语句的条件就为假（`False`），因为 6 不小于 6。又因为 `while` 语句的条件是假的（`False`），程序就不会再进入循环体，而是跳到循环体的后面。

玩家用完了猜测的机会后，剩下的代码就会被执行。（不管玩家是猜对了，还是猜错了。）玩家跳出循环的原因决定了他们赢还是输，并且程序会在屏幕上显示对应的信息。

看玩家是不是赢了

```
28.     if guess == number:
```

与第 25 行不同的是，这行代码没有缩进。这就意味着 `while` 循环已经结束，并且这是循环后的第一行代码。不管是因为 `while` 语句的条件为假（玩家用完了猜测的机会），还是我们执行了 `break` 语句（玩家猜对了），我们都会跳出 `while` 循环。第 28 行代码是用来检查玩家是否猜对了的。如果是，我们就执行 `if` 语句里面的代码。

```
29.         guessesTaken = str(guessesTaken)
30.         print('Good job, ' + myName + '! You guessed my number
           in ' + guessesTaken + ' guesses!')
```

第 29 和 30 行是 `if` 语句内的代码。只有当第 28 行的 `if` 语句的条件为真（`True`）的时候它们才会被执行（也就是玩家才对的时候）。

在第 29 行我们调用了一个名为 `str()` 的新函数，这个函数会把传给它的参数以字符串类型返回。我们用这个函数把 `guessesTaken` 中保存的整数转换成字符串。

第 29 行用来告诉玩家他们赢了，并且告诉他们用了几次机会。注意，这行代码中我们把 `guessesTaken` 的值转换成了字符串类型，因为我们只能把字符串相加（也就是，连接）。如果我们试图将字符串和整数相加，Python 解释器显示错误信息。

看玩家是不是输了

```
32.     if guess != number:
```

在第 32 行我们在 `if` 语句的条件中用到了比较运算符 `!=`，它的意思是“不等于”。如果玩家猜的数字太小或者太大（也就是，不等于）电脑产生的神秘数字，那么这个条件的值就是真（`True`），然后我们就会执行 `if` 语句后面的第 33 行代码。

第 33 和 34 行是 `if` 语句内的循环，这些代码只有在 `if` 语句的条件为真（`True`）时才会被执行。

```
33.         number = str(number)
34.         print('Nope. The number I was thinking of was ' + number)
```

这个代码块用来告诉玩家神秘数字，因为他们猜错了。但是我们先要保存 `number` 的新的版本的价值。

这一行同样也在 `if` 语句内，而且只有在 `if` 语句的条件为真的时候才会被执行。现在，我们到了代码的最后了，程序结束了。

可喜可贺！我们刚刚完成了我们的第一个真正的游戏！

总结：编程到底是什么？

如果有人问你，“编程到底是什么？”你会怎么说呢？编程就是写代码的行为，也就是说，编写一些电脑可以执行的程序。

“但是程序到底是什么？”当你看到别人使用电脑程序的时候（比如，玩我们的猜数字游戏），你所看的不过是显示在屏幕上的一些文字。程序根据它的指令（即程序）和玩家按的键（即**输入**）决定展示什么内容（即**输出**）。程序有确切的指令来指示它们向用户展示什么内容。**程序**就是指令的集合。

“什么样的指令？”其实，只有几种不同的指令而已。

1.表达式，由值和运算符组成。表达式是可以运算得到一个值的，如 $2+2$ 运算得到4 或者 `'Hello' + ' ' + 'World'` 运算得到 `'Hello World'`。函数调用也属于表达式，因为它们本身会运算得到一个值，而且这些值用过运算符和其他值连接在一起。在关键字 `if` 和 `while` 后面的表达式，我们通常也把他们都称为条件。

2.赋值语句，就是用来把值包保存到变量中的，以便我们以后在程序中使用。

3.`if`,`while` 和 `break` 语句叫做**流程控制语句**，因为可以决定执行哪些语句。程序正常的执行流程是从头到尾一行行的执行。但是这些流程控制语句可以让程序跳过一些语句，循环一些语句，或者跳出循环。函数调用语句同样会改变程序的流程，程序会跳到函数里面执行函数。

4.`print()`是用来在屏幕上展示文本的。而 `input()`则可以通过键盘获取用户的输入。这叫做 **I/O**(读作“哎-欧”)，因为它用来处理程序的输入和输出的。

没了，就那四种。当然，这四种指令还有很多细节。在本书中，你还将会学到新的数据类型，`if`，`while` 和 `break` 意外的流程控制语句，以及几种新的函数。还有不同的 I/O（来自鼠标的输入，和输出声音、图形和图像，而不只是文本。）

使用你的程序的人，其实只在乎后面那种 I/O。用户通过硬盘输入，然后从屏幕上获取信息，听扬声器发出的声音。但是对电脑而言，要决定输出什么则需要程序来指示它，也就是你写的那些指令。

程序步进调试网页

如果你有连接互联网并且有浏览器，你可以访问本书的网站，地址 <http://inventwithpython.com/traces>，这是一个步进调试本书中所有的程序的网页。通过这工具一步一步的执行程序，也许可以帮助你理解猜数字游戏是如何工作的。这个网站模拟了程序的执行过程。不过并不是真的执行。

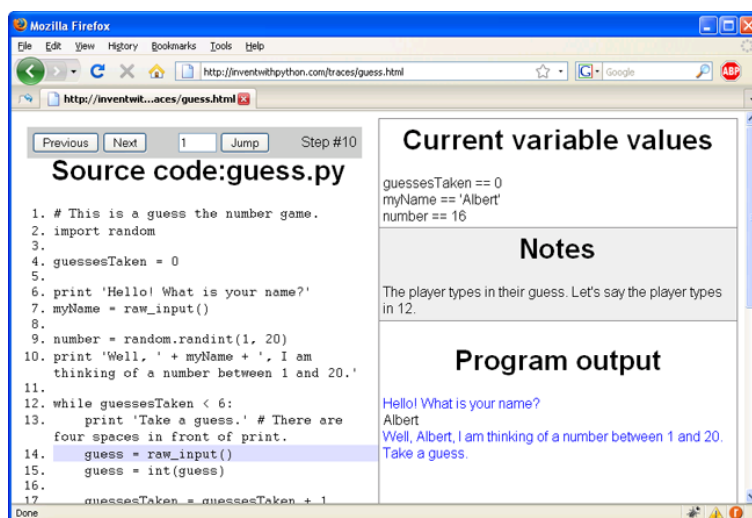


图 4-4: 步进调试页面

页面的左边是程序的源代码，高亮的那行代码是即将被执行的代码。要想执行这行代码并且跳到下一行，点“Next”即可。也可以点击“Previous”回到上一行。或者直接白色框中输入行号点击“Jump”就可跳到相应的代码行。

网页的右边有三个区域。“Current variable values”是用来告诉你被赋值的变量和值的。“Notes”区域则是用来提示你高亮的代码行是干什么的。“Program output”区域显示程序输出的内容，以及输入到程序的内容。（这个网页会在程序要求输入值的时候自动输入一个值。）

好了，有时间就去这些页面点点“Next”和“Previous”，就像我们上面做的那样。

本书的网站上还有一个教你如何使用步进调试工具的视频教程，请访问：<http://inventwithpython.com/videos/>。



第五章

讲笑话游戏

本章内容:

- 使用 `print()` 函数的 `end` 关键字参数
- 转义字符
- 单引号和双引号的使用

`print()` 函数全解析

本书中的大多数游戏都会用到文本的输入和输出。输入是指用户通过键盘向电脑键入内容。输出则是显示在屏幕上的文本。在 Python 中，`print()` 函数用来在屏幕上输出文本。我们已经学过 `print()` 函数的一些基本用法，但是关于 Python 中的字符串和 `print()` 函数的用法，我们还有很多要学的。

游戏运行演示

```
What do you get when you cross a snowman with a vampire?
```

```
Frostbite!
```

```
What do dentists call an astronaut's cavity?
```

```
A black hole!
```

```
Knock knock.
```

```
Who's there?
```

Interrupting cow.

Interrupting cow wh-MOO!

讲笑话游戏源代码

这是我们讲笑话小游戏的源代码。你要把它输入到文本编辑器中，然后保存为 `jokes.py`。如果你不想自己输入这些代码可以到本书的官方网站去下载。网址：<http://inventwithpython.com/chapter5>。

注意！ 确定你是在 Python 3 而不是 Python 2 下运行你的程序。本书使用的是 Python 3，如果你在 Python 2 下运行会得到错误信息。想知道自己的 Python 的版本可以点击 **Help** 然后 **About IDLE** 查看。

jokes.py

这些代码可以从 <http://inventwithpython.com/jokes.py> 下载

如果输入完后得到错误信息，可以用在线 `diff` 工具(<http://inventwithpython.com/diff>)把你输入的代码和书中的代码对比，或者给作者发邮件 al@inventwithpython.com

```
1. print('What do you get when you cross a snowman with a vampire?')
2. input()
3. print('Frostbite!')
4. print()
5. print('What do dentists call a astronaut\'s cavity?')
6. input()
7. print('A black hole!')
8. print()
9. print('Knock knock.')
10.  input()
11.  print("Who's there?")
12.  input()
13.  print('Interrupting cow.')
14.  input()
15.  print('Interrupting cow wh', end='')
16.  print('-MOO!')
```

如果你不理解这个程序的代码，别担心。你只要保存并运行它就可以了。记住，如果你的程序有漏洞（`bug`），你可以用本书的在线 `diff` 工具，地址：<http://inventwithpython.com/chapter5>

它是怎么工作的

我们来仔细看看这些代码。

```
1. print('What do you get when you cross a snowman with a vampire?')
2. input()
3. print('Frostbite!')
4. print()
```

这里我们调用了 `print()` 函数三次。因为我们不想玩家直接看到笑话的妙语部分，所以我们在调用了 `print()` 函数之后调用了 `input()` 函数。这样玩家就会先看到第一行字，按了回车之后才会看到笑话的妙语。

玩家也可以输入一个字符串后再按回车，不过由于我们没有把这个字符串保存在变量中，所以程序不会理会它，而会直接执行下一行代码。

我们没有给最后一个 `print()` 函数传参数。这是为了让程序输出空白行。空白行很有用，它可以让文本之间有合理的距离，而不会堆在一起。

转义字符

```
5. print('What do dentists call a astronaut\'s cavity?')
6. input()
7. print('A black hole!')
8. print()
```

你也许注意到了，在上面的第一个 `print()` 函数中的单引号前面有一个斜杠。这个反斜杠 (`\` 是反斜杠，`/` 是正斜杠) 告诉我们斜杠后面的字符是一个**转义字符**。转义字符可以帮助我们输出那些在源代码中不方便输入的字符。转义字符有很多种，我们在 `print()` 函数中用到的是单引号。

在这里我们必须使用单引号转义字符，否则 `Python` 解释器会误以为这是字符串结束的标志，但事实上这个引号是字符串的一部分。当我们输出这个字符串的时候，那个反斜杠不会输出。

其他转义字符

那如果你真正想输出反斜杠的时候怎么办呢？下面这行代码是不行的：

```
>>> print('He flew away in a green\teal helicopter.')
```

上面这行代码的输出结果如下：

```
He flew away in a green    eal helicopter.
```

这是因为“teal”中的“t”被视为是转义字符，因为它前面有一个反斜杠。转义字符 t 是模拟你键盘上的 Tab 键的。转义字符主要用来表示不能直接输入的字符。

用这行代码试试：

```
>>> print('He flew away in a green\\teal helicopter.')
```

下面是 Python 的转义字符列表：

表 5-1:转义字符

转移字符	实际输出的内容
\\	反斜杠(\)
\'	单引号(')
\"	双引号(")
\n	下一行
\t	Tab 键

单引号和双引号

在 Python 中字符串不一定要位于单引号之间。你也可以把它们放在双引号之间。下面两行代码输出结果相同：

```
>>> print('Hello world')
Hello world
>>> print("Hello world")
Hello world
```

但这两种方式不能混用。如果你试图输入下面这行代码就会出错：

```
>>> print('Hello world")
SyntaxError: EOL while scanning single-quoted string
>>>
```

我喜欢用单引号，因为输入单引号的时候不用按住 shift 键。这样输入起来比较简单，而且用哪种电脑都不会介意。

不过你要记住，正如你想在单引号之间输入单引号时要用转义字符一样，在双引号之间输入双引号也要用转义字符。比如，你可以看看这几行代码：

```
>>> print('I asked to borrow Abe\'s car for a week. He said,
"Sure."')
```

```
I asked to borrow Abe's car for a week. He said, "Sure."
>>> print("He said, \"I can't believe you let him borrow your
car.\")
He said, "I can't believe you let him borrow your car."
```

你有没有注意到？当你在单引号之间输入双引号的时候不用使用转义字符，在双引号之间输入单引号的时候也不用使用转义字符。Python 解释器很聪明，它知道如果一个字符串由一种引号开始，那么另一种引号就不代表它的结束。

end 关键字参数

```
9. print('Knock knock.')
10.     input()
11.     print("Who's there?")
12.     input()
13.     print('Interrupting cow.')
14.     input()
15.     print('Interrupting cow wh', end='')
16.     print('-MOO!')
```

你有没有注意到第 15 行的 `print()` 函数的第二个参数？通常，`print()` 函数在输出一行字符串的同时会换行。（这也是调用一个没有参数的 `print()` 函数会输出一行空白行的原因。）但是 `print()` 函数还有一个可选的参数（这个参数名为 `end`）。我们把 `print()` 函数的这个参数叫做**关键字参数**。这个 `end` 参数很特别，而且如果想使用这个参数我们要用 `end=` 这样的语法。

注意，当你输入关键字和关键字参数的时候只用一个 `=` 号。也就是 `end=`，而不是 `end==`。

给 `end` 一个空白字符串会使 `print()` 函数在它输出的字符串的后面加一个空格而不是换行。这就是 `'-MOO!'` 出现在前一行的后面而不是下一行的原因。也就是在输出字符串 `'Interrupting cow wh'` 后没有换行。

总结

这一章我们研究了 `print()` 函数的不同用法。我们也知道了转义字符的作用是输出那些不方便或者无法输入的字符。转义字符以反斜杠 `\` 开始，后面接需要转义的字母。例如，`\n` 是指换行。如果想在字符串中使用反斜杠，你可以用 `\\` 这个转义字符。

默认情况下 `print()` 函数会自动在输出一行字符串后换行。大多数情况下这是很方便的。但是有时候我们不想换行，这时我们就可以给 `end` 关键字一个空白字

字符串。比如，想输出'spam'并且不换行，那你就可以这样写代码 `print('spam', end='')`。

通过这种控制，我们可以更加灵活和准确的输出字符串，得到我们想要的效果。