

关于 python 的许多书和在线教程都会教你如何在 Python shell 中执行代码。这种方法执行代码，你需要打开命令行窗口 (Windows 系统) 或终端窗口 (macOS) 然后输入 “python”，出现 Python 命令提示符 (看起来像 >>>)。然后一次一次的输入命令，python 会执行它们。下面是两个典型的例子。

```
>>> 4 + 5
9
>>> print("I'm excited to learn Python.")
I'm excited to learn Python.
```

虽然这种方法执行代码快而有趣，但是随着代码的行数增加，它的放大效果就不好了。当你需要许多行代码才能完成任务时，将本部代码写入 python 脚本然后再运行它要相对易容一些。下面告诉你如何创建 Python 脚本。

如何创建 Python 脚本

要创建 Python 脚本：

1. 打开 Spyder IDE 或者文本编辑器 (如 Windows 系统里的 Notepad, Notepad++, 或 Sublime Text; macOS 系统里的 TextMate, TextWrangler, 或 Sublime Text on macOS)。

2. 在文本文件中写入如下两行代码：

```
#!/usr/bin/env python3
print("Output #1: I'm excited to learn Python.")
```

第一行是特殊的行称为 shebang，它总是在你的 Python 脚本的第一行。注意第一个字符是 Pound 或 hash 符号 (#)。符号 # 在一行注释的前面，所以这一行代码不会被计算机读取或执行。但是 Unix 计算机会用这一行查找 python 的版本以执行文件中的代码。因为 Windows 设备忽略这一行而基于 Unix 的系统例如 macOS 使用它，把这一行写入文件可以使脚本在不同类型的计算机移植。第二行是简单的打印语句。这一行将打印双引号内的文本到命令行 (Windows) 或终端 (macOS) 窗口。

3. 打开另存为 (Save As) 对话框。

4. 在位置框中，导航到你的桌面，使文件保存在你的桌面。

5. 在格式框中选择所有文件 (All Files) 使对话框不选择文件类型。

6. 在另存为框或文件名框中，输入 “first_script.py”。过去，你可能保存文本文件为 .txt 文件。但是，这次，你要保存为 .py 文件来创建 Python 脚本。

7. 点击保存。

现在你就创建了一个 Python 脚本。图 1-1, 1-2, 和 1-3 显示它在 Anaconda Spyder, Notepad++ (Windows), 和 TextWrangler (macOS) 中的样子。

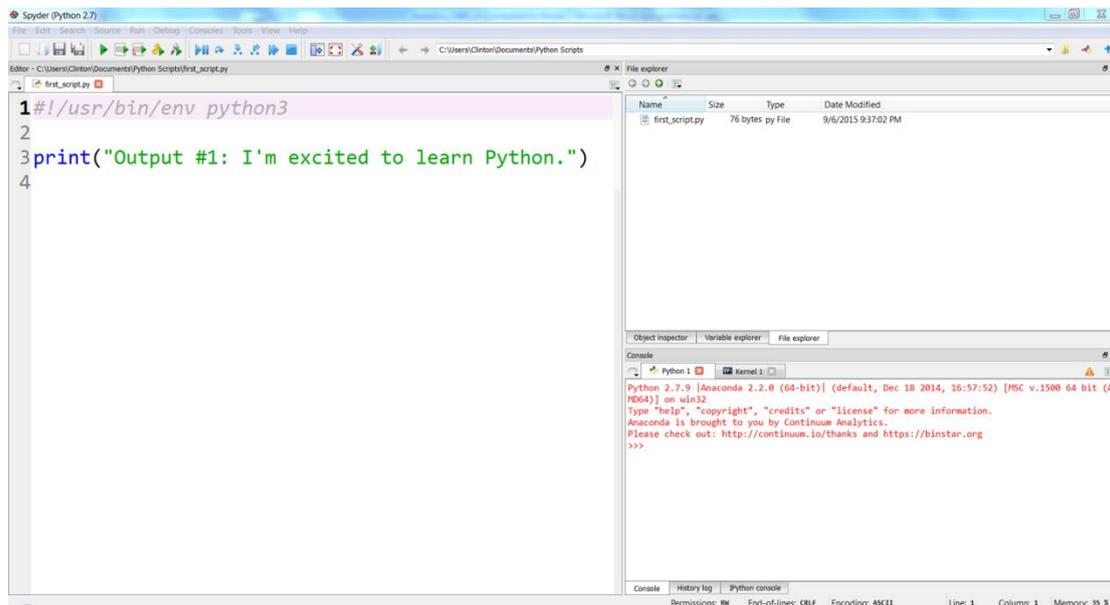


图 1-1. Python 脚本 `first_script.py` 在 Anaconda Spyder 中

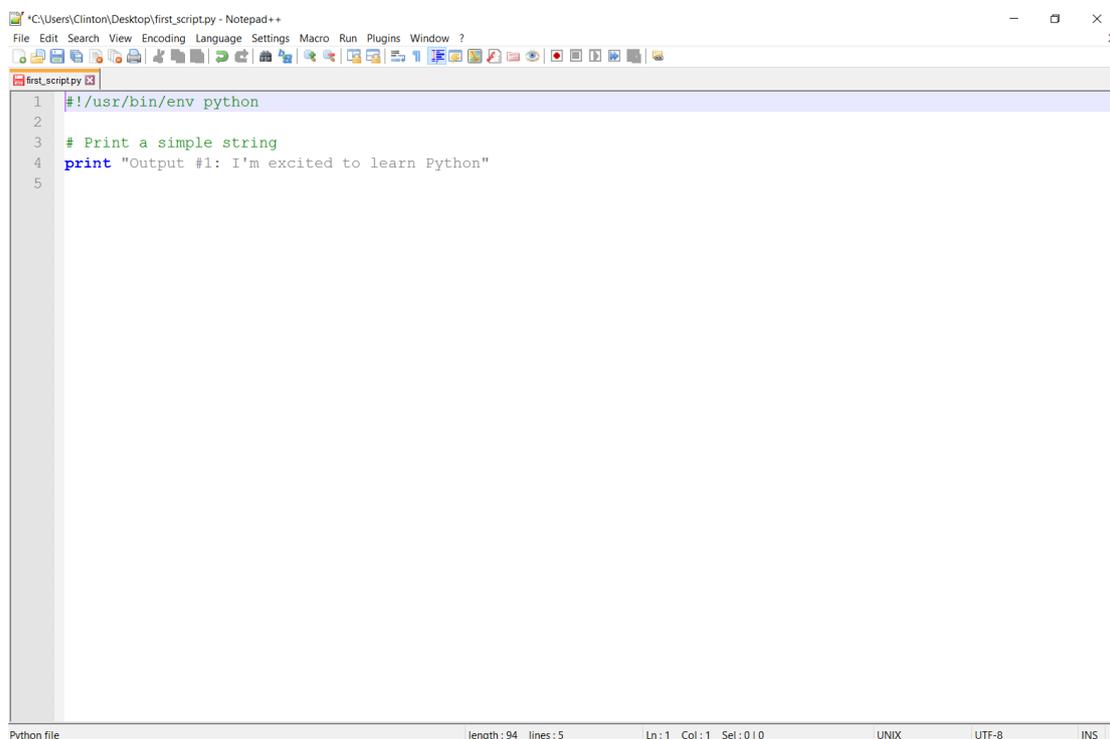


图 1-2 Python 脚本 `first_script.py` 在 Notepad++ 中

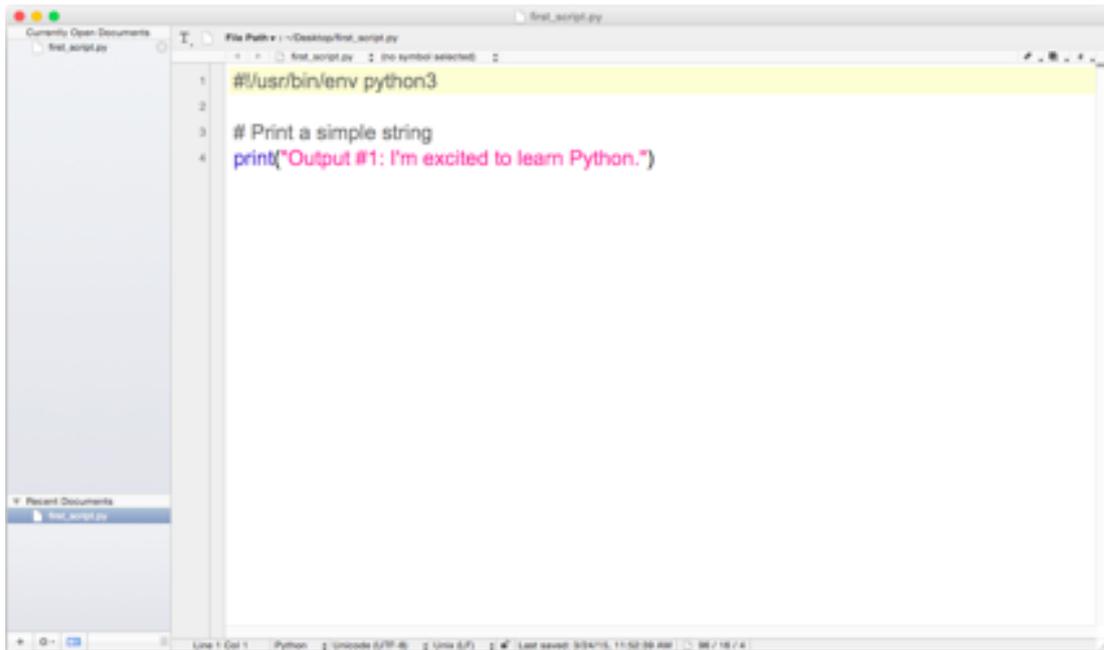


图 1-3 Python 脚本 first_script.py 在 TextWrangler

下一节将解释如何在命令行或终端中执行 Python 脚本。你会发现运行脚本与创建脚本一样容易。

如何运行 Python 脚本

如果你用 Anaconda Spyder IDE 创建脚本,你可以通过点击 IDE 屏幕左上角的绿色三角形(运行按钮)来运行它。当你点击运行按钮时,你会看到输出显示于 IDE 右下方的 Python console 面板。截屏显示了绿色按钮和红框内的输出(见图 Figure 1-4)。本例输出是“Output #1: I’ m excited to learn Python.”

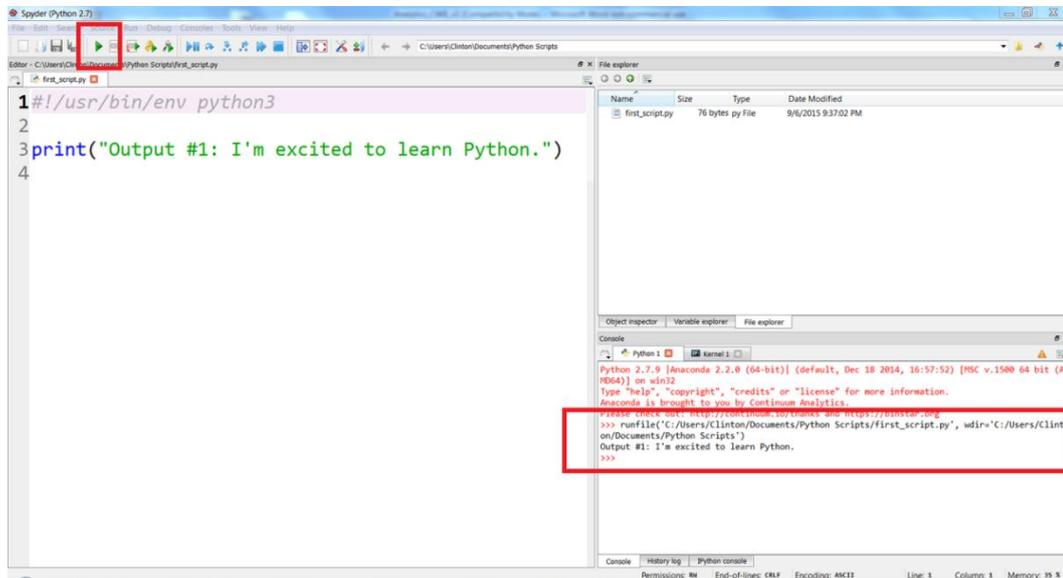


图 1-4 在 Anaconda Spyder IDE 中运行脚本

或者，你可以在命令行或终端运行脚本，方法如下：

Windows 命令行

1. 打开命令行窗口。

当窗口打开时命令行将会在特定的文件夹也称为目录中（如，C:\Users\Clinton 或 C:\Users\Clinton\Documents）。

2. 导航到桌面（我们的 Python 脚本保存的地方）。

你输入以下一行并按回车键就可以实现：

```
cd "C:\Users\[Your Name]\Desktop"
```

用你的计算机用户名替代 [Your Name]，它通常是你的名字。例如，在我的电脑上，我输入：

```
cd "C:\Users\Clinton\Desktop"
```

这时命令行看起来是 C:\Users\[Your Name]\Desktop，这是我们需要到达的地方，因为这是我们保存 Python 脚本的地方。最后一步是运行脚本。

3. 运行 Python 脚本。

输入下面一行并按回车键即可：

```
python first_script.py
```

你会看到输出打印到命令行窗口，如图 1-5：

```
Output #1: I'm excited to learn Python.
```

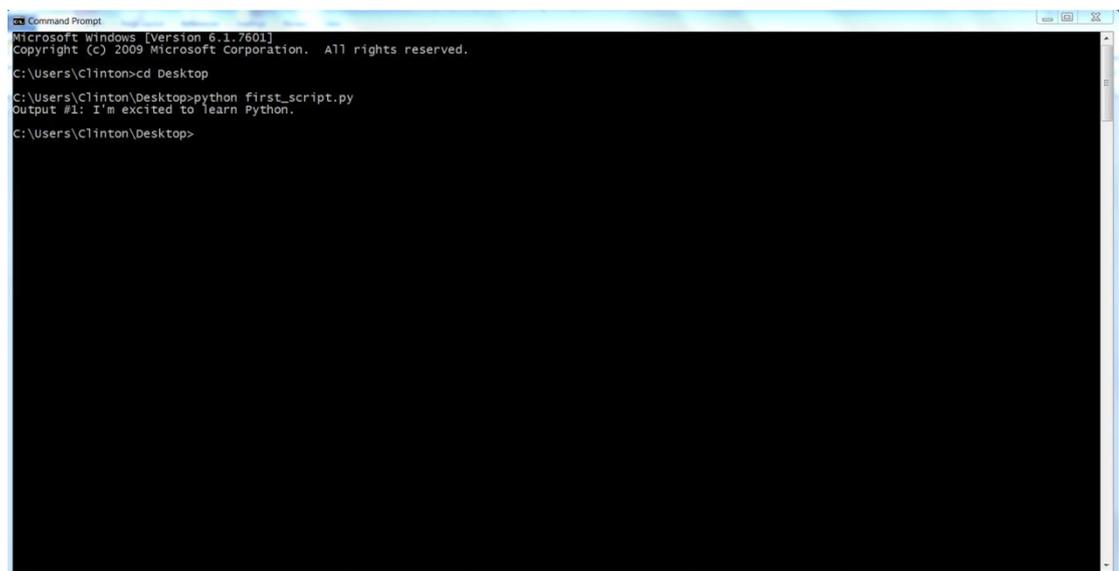


图 1-5 在命令行中运行脚本

终端 (Mac)

1. 打开终端窗口。

当窗口打开时，命令行将在特定的文件夹也称为目录里（如，/Users/clinton 或 /Users/clinton/Documents）。

2. 导航到桌面（我们的 Python 脚本保存的地方）。

你输入以下一行并按回车键就可以实现：

```
cd /Users/[Your Name]/Desktop
```

用你的计算机用户名替代 [Your Name]，它通常是你的名字。例如，在我的电

脑中，我输入：

```
cd /Users/clinton/Desktop
```

这时命令行看起来是 /Users/[Your Name]/Desktop，这是我们需要到达的地方，因为这是我们保存 Python 脚本的地方。下一步是使脚本可执行并运行脚本。

3. 使 Python 脚本可执行。

你输入以下一行并按回车键就可以实现：

```
chmod +x first_script.py
```

Chmod 命令是 Unix 命令是 change access mode 的缩写。+x 指明你添加执行访问模式，而不是读或写访问模式，到你的访问设置使 Python 可以执行脚本中的代码。针对你创建的每个 Python 脚本你都要运行一次 chmod 命令使脚本可执行。只要你运行了一次 chmod 命令你就可以多次运行脚本而不需要重复 chmod 命令。

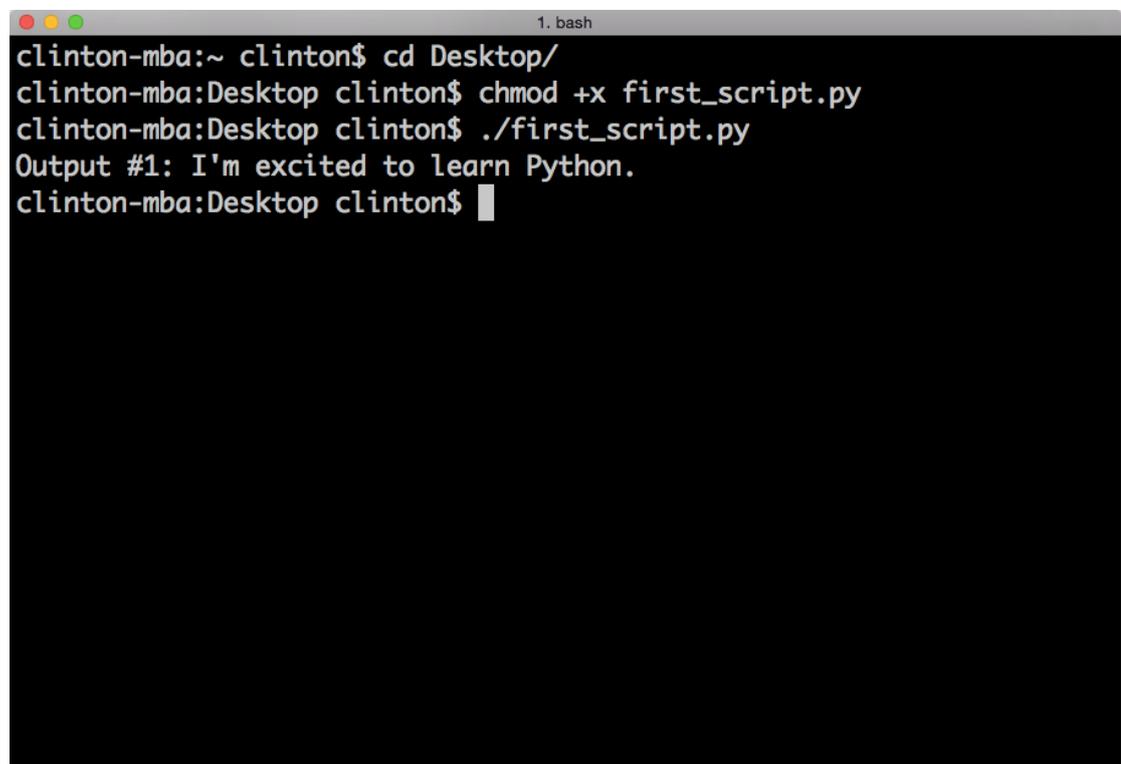
4. 运行 Python 脚本。

你输入以下一行并按回车键就可以实现：

```
./first_script.py
```

你会看到输出打印到终端窗口，如图 1-6：

Output #1: I'm excited to learn Python.

A terminal window titled "1. bash" with a black background and white text. The text shows the following sequence of commands and output:

```
clinton-mba:~ clinton$ cd Desktop/  
clinton-mba:Desktop clinton$ chmod +x first_script.py  
clinton-mba:Desktop clinton$ ./first_script.py  
Output #1: I'm excited to learn Python.  
clinton-mba:Desktop clinton$
```

图 1-6 在终端中运行脚本

与命令行交互的有用提示

这里是一些与命令行交互的有用提示：

向上箭头查找以前的命

命令行和终端的一个优点是你可以用向上箭头来追踪你以前的命令。你现在尝试在命令行或终端里按向上键，你可以追踪到你以前的命令，Windows 中是 `python first_script.py`，Mac 中是 `./first_script.py`。

用这个特点，你可以减少输入次数以运行脚本，这非常方便，特别是脚本名很长或你要提供额外的参数给命令行时（比如输入文件或输出文件的名称）。

用 Ctrl+c 停止运行脚本

现在你已经运行了脚本，现在是时候告诉你如何中断或停止 Python 脚本。有些情况你不得不停止脚本。例如，可能代码里有无限循环，你的脚本永远不会停止运行。有时候你写的脚本需要很长时间才能执行完毕而你想提前中止脚本运行。

你运行脚本后，任何时候想中断或停止，按下 Ctrl+c 即可。它会停止你用命令行启动的进程。你不用担心过多的技术细节。进程是计算机查找命令队列的方法。你写的脚本或程序，计算机将它解释为进程，或者更复杂的，一序列的进程按序执行或同时执行。

查找错误信息

当我们讨论处理困难的脚本时，我们先谈谈当你输入 `./python first_script.py` 时该如何，或想运行任何 Python 脚本，但是运行不正常，你的命令行或终端给你一个错误信息。

首先要放松并看一下错误信息。有时候错误信息会提示你哪行代码出错你可以集中精力解决那行代码的错误（文本编辑器或 IDE 会告诉你行号；如果没有的话你可以找一找菜单或上网查一下如何显示行号）。要认识到错误信息是编程的一部分，学习写代码包括学习如何有效的调试错误。

另外，由于错误信息很常见，通常调试错误会相对容易。你并不一定是第一个遇到错误的人，因此可以上网找解决方案——最好的选择是拷贝你的错误信息到搜索引擎中，并在结果中查找别人是如何解决的。了解 python 的内置意外也是很有用的，这样你可以识别标准的错误信息并知道如何修改错误。你可以在 Python Standard Library 里了解 python 的内置意外，但了上网查找这些错误信息看别人是如何解决的还是很有用的。

给 first_script.py 添加更多的代码

现在你习惯了写 Python 代码并运行你的 Python 脚本了，尝试通过增加更多的代码来编辑 `first_script.py` 并运行脚本。作为扩展实践，增加本章的代码到脚本的底部，重新保存脚本，然后运行脚本。例如，增加下面的二个代码块，然后保存和运行脚本（记住，你可以用向上箭头来追踪你以前的命令而不用重复输入）：

```
# Add two numbers together
x = 4
y = 5
```

```
z = x + y
print("Output #2: Four plus five equals {0:d}.".format(z))
# Add two lists together
a = [1, 2, 3, 4]
b = ["first", "second", "third", "fourth"]
c = a + b
print("Output #3: {0}, {1}, {2}.".format(a, b, c))
```

以#符号开头的行是注释，它可以注释代码并描述它的用途。

这两个例子告诉我们如何给变量赋值，将两个变量相加，然后格式化输出语句。我们来看一下打印语句中的符号，"`{0:d}`".`format(z)`。花括号({})是占位符，它的值将传递给打印语句，本例中是来自变量`z`。`0`指向变量`z`的第一位。本例中，`z`只有一个值，因此`0`就指向那个值；但是，如果`z`是列表或元组并有很多个值，`0`指向`z`的第一值。

冒号(:)分开将要被填入的值。符号`d`指明值应格式化为不带小数的数字。下一节学习如何指明小数位以显示浮点数。

第二个例子展示如何创建列表，合并列表，然后分别打印出变量值，用逗号分开。打印语句中"`{0}, {1}, {2}`".`format(a, b, c)`，表示如何包括多个值到打印语句中。值`a`传递给`{0}`，值`b`传递给`{1}`，值`c`传递给`{2}`。因为三个值都是列表，而不是数字，我们不指明值的数字格式。我们会在后面进行更多的讨论。

为什么在打印时要用.format?

`.format`并不是每个打印语句都必须用的，但是它功能强大，可以节省很多的键盘输入。

上面你刚创建的例子中，注意`print("Output #3: {0}, {1}, {2}.".format(a, b, c))`给出了三个变量的内容并用逗号分开。如果你要得到一样的结果而不用`.format`，你要这样写：`print("Output #3: ", a, ", ", b, ", ", c)`，你要用很多的代码书写。我们会在后面讨论`.format`的使用，现在先习惯它然后你可以在想用的时候用它。

图 1-7 和图 1-8 显示增加新代码之后在 Anaconda Spyder 和 Notepad++ 中的样子。

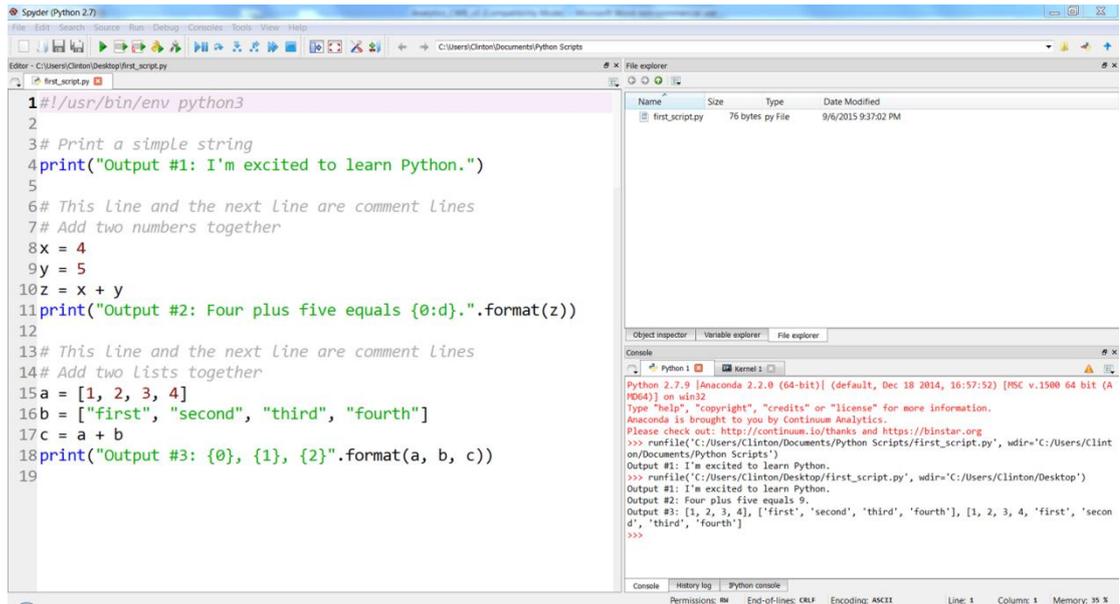


图 1-7. 在 Anaconda Spyder 里增加代码到 `first_script.py`

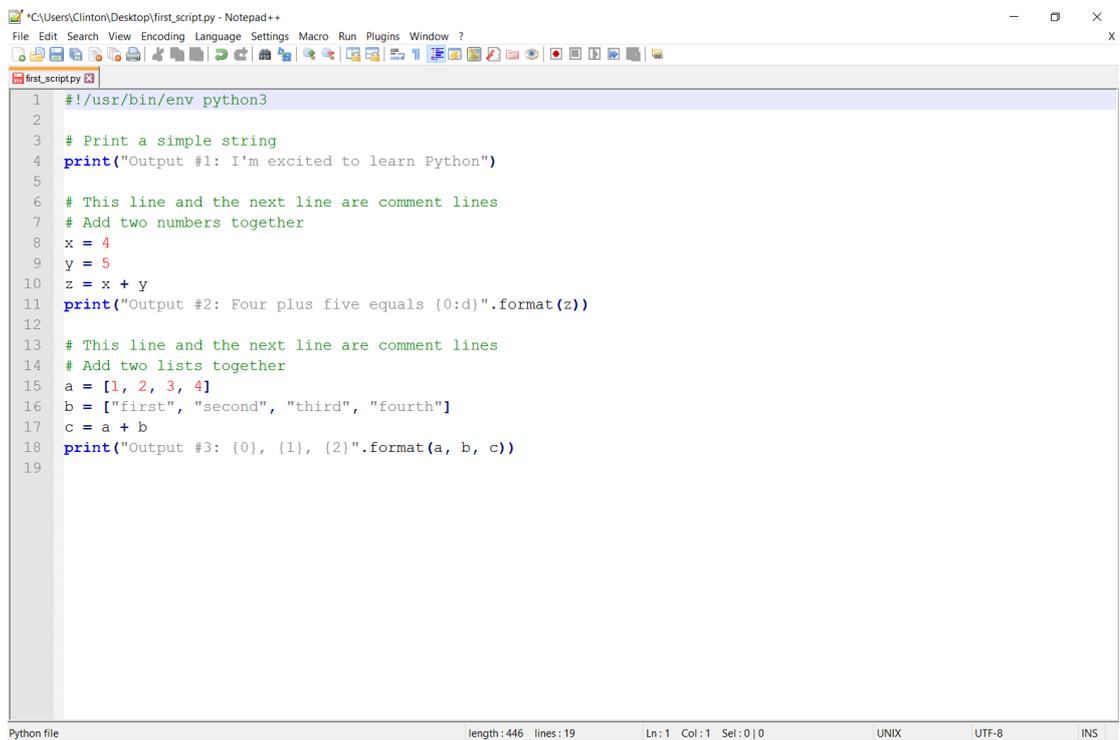


图 1-8. 在 Notepad++ (Windows) 里增加代码到 `first_script.py`

如果你已增加上面的代码到 `first_script.py`，然后保存，运行脚本，你会看到下在的输出（见图 Figure 1-9）：

Output #1: I'm excited to learn Python.

Output #2: Four plus five equals 9.

Output #3: [1, 2, 3, 4], ['first', 'second', 'third', 'fourth'],

[1, 2, 3, 4, 'first', 'second', 'third', 'fourth']

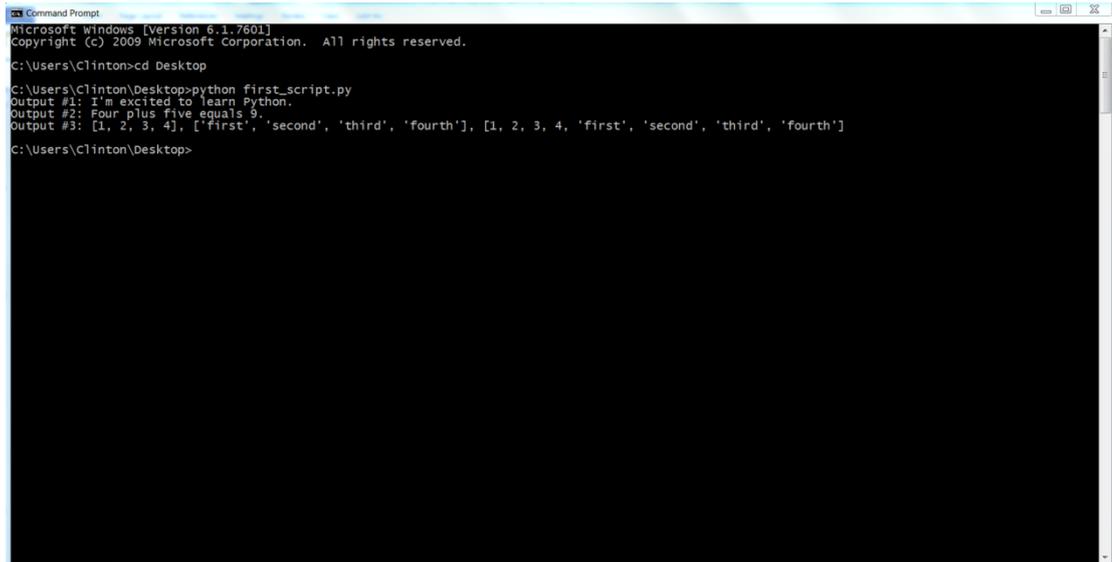


图 1-9. 在 Windows 命令行中运行添加代码的 first_script.py

Python 的基础构件

现在已经可以创建和运行 python 脚本了。后面的章节我们将更详细的了解如何使用 python 脚本。在继续之前，我们需要熟悉 python 的构件。你将理解并更习惯如何组合它们来进行数据分析任务的。首先，我们将处理一些最常见的数据类型，然后用 if 语句让程序进行决策。接下来我们让 python 读写文件，文本和简单的表格文件（CSV）。

数字 Numbers

Python 有多个内置的数字类型。这非常的伟大，因为很多的商务应用需要分析和处理数字。Python 的四种数字类型为整数，浮点数，长型，复数。我们先学习整数和浮点数，因为它们在实际应用中最为常见。你可以在 first_script.py 中添加下面的例子来处理整数和浮点数。看看屏幕的输出结果。

整数 Integers

我们直接进入一些关于整数的例子：

```
x = 9
print("Output #4: {}".format(x))
print("Output #5: {}".format(3**4))
print("Output #6: {}".format(int(8.3)/int(2.7)))
```

Output #4 展示如何给整数赋值，数字 9 赋值给变量 x 并打印变量 x。
Output #5 说明如何对数字 3 进行 4 次方进算。（它等于 81）然后打印结果。
这些数字都由内置的 int 函数作为整数处理。因此 8 除以 2 等于 4.0。

浮点数 Floating-point numbers

类似整数，浮点数——具有十进制小数点的数对于许多任务来说是非常重要的。下面是一些关于浮点数据例子：

```
print("Output #7: {:.3f}".format(8.3/2.7))
y = 2.5*4.8
print("Output #8: {:.1f}".format(y))
r = 8/float(3)
print("Output #9: {:.2f}".format(r))
print("Output #10: {:.4f}".format(8.0/3))
```

Output #7 非常类似 Output #6，只是我们将数字作为浮点数进行除法运算，即 8.3 除以 2.7 约等于 3.074。本例中打印语句的符号“{:.3f}”.format (floating_point_number/floating_point_number) 表示如何在打印语句中指明小数点的位置。本例中 .3f 表示打印输出三位小数。

Output #8 表示 2.5 乘 4.8，将结果赋值给变量 y，打印输出一位小数。这两个浮点数相乘的结果为 12，所以打印值为 12.0。

Outputs #9 和 #10 表示两种方法将 8 除以 3。结果都约为 2.667，是浮点数。

类型函数 The type Function

Python 提供了一个称为 type 的函数，你可以用它来查找 python 所处理的对象的更多的信息。你调用它来用于数值变量，它会告诉你它们是整数还是浮点数，它还会告诉你它们是否被当作字符串处理。符号非常简单 type(variable) 返回变量的类型。另外，因为 python 是面向对象的语言，任务具有名字的对象你都可以使用 type 函数不但是变量而且还可以是函数，语句，等等。如果你的代码行为异常，调用 type 函数可以帮助你诊断。

关于 python 数字的重要细节是有多个标准库模块和内置函数和模块可以用来进行数学运算。你已经看到了两种内置的函数 int 和 float 来进行数字操作。另一个有用的标准模块是 math 模块。当你安装完 python 后，python 的标准模块在你的计算机里，但是你运行新脚本的时候，python 只加载非常基础的运算（这也是为什么 python 启动快的部分原因）。为了使用 math 模块，你需要在脚本头部 shebang 下面添加 from math import [function name]。例如添加下面的一行在 first_script.py 头部 shebang 下面：

```
#!/usr/bin/env python3
from math import exp, log, sqrt
```

一旦你添加了这一行代码，你就可以使用三个有用的数学函数了。函数 exp 对括号内的数值进行自然指数次方运算，log 进行自然对数运算，sqrt 进行二次根运算。下面是一些使用数学模块的例子：

```
print("Output #11: {:.4f}".format(exp(3)))
print("Output #12: {:.2f}".format(log(4)))
print("Output #13: {:.1f}".format(sqrt(81)))
```

这三个数学表达式的结果为浮点数分别约为 20.0855，1.39，和 9.0。这只是 math 模块可用的函数的开始。python 内有许多有用的数学函数和模块，用于商务，科学，统计及其它应用，我们将会在后面讨论。你可以在 [the Python Standard Library](#) 获取更多的标准模块和内置函数的信息。

字符串 Strings

字符串是 python 的另一种数据类型。它通常是人类可读的文本，这是通常的看法，但是更一般的是它是字符的一序列只有全部读完才有意义。字符串在很多商务应用中出现，包括供应商和消费者的名字，地址，备注和反馈信息，事件日志，文件等。有些看起来像整数，但它们实际上是字符串。举个例子，邮政编码，01111 与整数 1111 并不相同。邮政编码不能进行加减乘除运算。本节我们简介一些模块，函数，操作符来处理字符串。字符串可以用单引号，双引号，三引号，三个双引号来标记。下面是一些字符串的例子：

```
print("Output #14: {0:s}".format('I\'m enjoying learning Python.'))
print("Output #15: {0:s}".format("This is a long string. Without the
backslash\
it would run off of the page on the right in the text editor and be very\
difficult to read and edit. By using the backslash you can split the long\
string into smaller strings on separate lines so that the whole string
is easy\
to view in the text editor."))
print("Output #16: {0:s}".format('''You can use triple single quotes
for multi-line comment strings.'''))
print("Output #17: {0:s}".format("""You can also use triple double quotes
for multi-line comment strings."""))
```

Output #14 与本章开始的例子相似。它展示用单引号标记字符串。打印语句的结果是：“I’m enjoying learning Python.”。记住，如果我们用双引号标记字符串，没有必要在单引号前加反斜杠来处理“I’m”中的单引号。

Output #15 展示你可以用反斜杠来分割长的行使它易于读取和编辑。虽然脚本中长行被分为多行，但是它仍然是字符串并打印为一个字符串。重要的一点是反斜杠必须是行的最后一个字符。这意味着你偶然按下空格使反斜杠后有个不可见的空格，你的脚本将抛出符号异常而不会是你想要的结果。出于这个原因最好用三个单引号或三个双引号来创建多行字符串。

Outputs #16 和 #17 展示如何使用三个单引号和三个双引号来创建我行字符串。这些例子的输出是：

```
Output #16: You can use triple single quotes
for multi-line comment strings.
Output #17: You can also use triple double quotes
for multi-line comment strings.
```

当你使用三个单引号或双引号时你不需要在顶行的结尾加上反斜杠。另外注意 Output #15 与 Outputs#16 及 #17 的不同。

Output #15 分开多行用行尾的反斜杠，使每一行更短而易读，但是它是一行一行打印出文本的。相反地，Outputs #16 和 #17 用三个单引号和双引号来创建多行字符串，它们打印单独的行。

就像数字一样，有许多标准模块，内置函数，操作符，你可以用来处理字符串。一些有用的操作符和字符串包括+，*，和 len。下面是使用这些操作符的例子：

```
string1 = "This is a "
```

```
string2 = "short string."  
sentence = string1 + string2  
print("Output #18: {0:s}".format(sentence))  
print("Output #19: {0:s} {1:s} {2:s}".format("She is", "very "*4,  
"beautiful."))  
m = len(sentence)  
print("Output #20: {0:d}".format(m))
```

Output #18 展示用+号来连接二个字符串。结果是 This is a short string——运算符+将字符串加在一起，所以你如果想在结果字符串中有空格，你必须添加带双引号空格。(例如,Output #18 的 “a” 后面加)或字符串之间加 (例如, Output #19 的 “very” 之后)。

Output #19 展示如何使用 * 操作符来指定处理次数。本例中, 结果字符串中包括字符 “very ” 的 4 次拷贝。(如, “very” 后跟着空格)。

Output #20 展示如何使用内置函数 len 来确定字符串中的字符数。len 函数也计算空格和标点符号。因此 Output #20 中字符串 This is a short string. 长为 23 字符。

处理字符串的一个有用的标准模块是 string 模块。有了 string 模块, 你可以使用许多有用的函数来处理字符串。下一节讨论使用这些函数的例子。

split

下面的例子展示如何使用 split 函数来将字符串分为子串列表。list 是 python 内置的另一个数据类型, 我们在后面讨论。Split 函数的括号内有二个参数。第一个参数表示分割的字符串的字符数。第二个参数表示进行多少次分割。(如, 2 个 splits 生成 3 个子串):

```
string1 = "My deliverable is due in May"  
string1_list1 = string1.split()  
string1_list2 = string1.split(" ", 2)  
print("Output #21: {0}".format(string1_list1))  
print("Output #22: FIRST PIECE:{0} SECOND PIECE:{1} THIRD PIECE:{2}"\  
.format(string1_list2[0], string1_list2[1], string1_list2[2]))  
string2 = "Your, deliverable, is, due, in, June"  
string2_list = string2.split(',')  
print("Output #23: {0}".format(string2_list))  
print("Output #24: {0} {1} {2}".format(string2_list[1],  
string2_list[5],\  
string2_list[-1]))
```

在 Output #21 中, 括号内没有参数, 所以分割按空格进行 (这是默认的)。因为字符串中有 5 个分割, 字符串分为六个子串列表。生成的列表为 ['My', 'deliverable', 'is', 'due', 'in', 'May']。

在 Output #22 中, 我们指明了二个参数。第一个参数是 “ ”, 表示我们想按空格符分割。第二个参数是 2, 表示我们只想分割前二个空格。因为我们指明二次分割, 所以产生了三个元素的列表。第二个参数很方便传递参数。例如, 你可以解析日志文件, 它包含时间戳, 错误代码, 错误信息, 由空格分开。这种情况

下，你想分割前两个空格来解析时间戳、错误代码，但是要保持错误信息完整。

在 Outputs #23 和 #24 中，括号里的额外参数是逗号，这种情况下，只要有逗号就被分割。结果是列表 ['Your', 'deliverable', 'is', 'due', 'in', 'June']。

join

下面的例子我们展示如何使用 join 函数来组合列表中的子串到一个字串。join 函数前有一个参数，它指明子串之间使用的符号。

```
print("Output #25: {0}".format(', '.join(string2_list)))
```

本例中，参数为逗号，因此 join 函数组合两个子串到一个字串中间带逗号，因为有 6 个子串，所以中间有 5 个逗号。产生的字串为 Your, deliverable, is, due, in, June。

strip

下面的例子展示如何使用 strip, lstrip, 和 rstrip 函数来去除不想要的字符。这些函数都有一个参数指明想要去除的符号。第一个例子集展示如何用 strip, lstrip, 和 rstrip 函数去除空格, tabs 和 newline 符号, 从左边, 右边, 两边进行去除。

```
string3 = " Remove unwanted characters from this string. \t\t \n"
```

```
print("Output #26: string3: {0:s}".format(string3))
```

```
string3_lstrip = string3.lstrip()
```

```
print("Output #27: lstrip: {0:s}".format(string3_lstrip))
```

```
string3_rstrip = string3.rstrip()
```

```
print("Output #28: rstrip: {0:s}".format(string3_rstrip))
```

```
string3_strip = string3.strip()
```

```
print("Output #29: strip: {0:s}".format(string3_strip))
```

string3 的左边有多个空格，右边有 tabs (\t)，空格，newline (\n) 符号。如果你没见过 \t 和 \n 符号，它们是计算机表示 tabs 和 newlines。在 Output #26 中，你看到句首有空格，你看到句末有空白行。你看不到句末的 tabs 和空格，但是它们是存在的。

Outputs #27, #28, 和 #29 展示如何分别从左边, 右边, 两边去除空格、tabs、和 newline 符号。{0:s} 中的 s 表示传递给打印语句的值应格式化为字符串。

第二个例子集展示如何从字串尾部去除字符，在 strip 函数中添加参数。

```
string4 = "$$Here's another string that has unwanted characters. ___-+ "
```

```
print("Output #30: {0:s}".format(string4))
```

```
string4 = "$$The unwanted characters have been removed. ___-+ "
```

```
string4_strip = string4.strip('$_-+')
```

```
print("Output #31: {0:s}".format(string4_strip))
```

本例中，美元符号(\$)，下划线(_)，横杠(-)，加号(+) 需要从字串尾部去除，通过指明这些字符作为参数，我们告诉程序从字串的尾部去除它们。

replace

下面的例子告诉你如何用 replace 函数替代字串的字符或字符集。函数有两个参数，第一个参数是字串中需要查找的字符或字符集，第二个参数是需要替换

的字符或字符集。

```
string5 = "Let's replace the spaces in this sentence with other characters."
```

```
string5_replace = string5.replace(" ", "!@!")  
print("Output #32 (with !@!): {0:s}".format(string5_replace))  
string5_replace = string5.replace(" ", ",")  
print("Output #33 (with commas): {0:s}".format(string5_replace))
```

Output #32 展示用字符替换字符串中的空格，使用 `replace` 函数。结果字符串为 `Let's!@!replace!@!the!@!spaces !@!in!@!this!@!sentence!@!with!@!other!@!characters..`

Output #33 展示用逗号替换字符串中的空格。结果字符串为 `Let's,replace,the,spaces,in,this,sentence,with,other,characters..`

lower, upper, capitalize

最后三个例子展示如何使用 `lower`、`upper` 和 `capitalize` 函数。`lower` 和 `upper` 函数转换字符串中的字符为小写、大写。`capitalize` 函数使第一个字符为大写而其它字符为小写。

```
string6 = "Here's WHAT Happens WHEN You Use lower."  
print("Output #34: {0:s}".format(string6.lower()))  
string7 = "Here's what Happens when You Use UPPER."  
print("Output #35: {0:s}".format(string7.upper()))  
string5 = "here's WHAT Happens WHEN you use Capitalize."  
print("Output #36: {0:s}".format(string5.capitalize()))  
string5_list = string5.split()  
print("Output #37 (on each word):")  
for word in string5_list:  
print("{0:s}".format(word.capitalize()))
```

Outputs #34 和 #35 直接使用 `lower` 和 `upper` 函数。将函数应用到字符串，`string6` 中的所有字符变为小写，而 `string7` 中的所有字符为大写。

Outputs #36 和 #37 展示 `capitalize` 函数。Output #36 展示 `capitalize` 函数将字符串的第一个字符变为大写将其它字符变为小写。

Output #37 在 `for` 循环中使用 `capitalize`。我们将在后面讨论 `for` 循环结构。我们只领略一下。

短语 `for word in string5_list:` 基本的说，“对于列表 `string5_list` 中的每个元素，进行一些操作。”下一个短语，`print word.capitalize()`，就是列表中每一个元素要进行的操作。放在一起，二行代码基本地说，“对于列表 `string5_list` 中的每一个元素，使用 `capitalize` 函数并打印元素。”结果是列表中的每个单词的第一个字符为大写其它字符为小写。Python 中还有很多的模块和函数用于处理字符串。见 [the Python Standard Library](#)。

正则表达式和模式匹配 Regular Expressions and Pattern Matching

许多的商务分析依靠模式匹配，也称为正则表达式。例如，你可能需要分析某个州的订单（例如，Maryland 州）。这种情况，你要查找的模式为单词 `Maryland`。

相似地, 你可能要分析某个供应商的产品质量 (例如, 供应商为 StaplesRUs)。这里, 你要查找的模式为 StaplesRUs。

Python 包括 re 模块, 提供很伟大的函数来查找文本中的指定的模式 (如, 正则表达式)。要在脚本中使用 re 模块提供的函数, 在脚本的顶部添加 import re, 在前面的 import 语句下。现在 first_script.py 看起来像:

```
#!/usr/bin/env python3
from math import exp, log, sqrt
import re
```

通过导入 re 模块, 你可以访问许多 metacharacters 和函数来创建和查找任意复杂的模式。Metacharacters 是正则表达式中的符号, 具有特定意义。通过独特的方式, metacharacters 使正则表达式可以匹配不同的字串。一些常用的 metacharacters 为 |, (), [], ., *, +, ?, ^, \$, 及 (?P<name>)。如果你在正则表达式中看到这些符事情, 软件不是查找这些符号而是它们描述的东西。关于 metacharacters 的更多细节, 请 [Python Standard Library 的“Regular Expression Operations”](#) 一节。

Re 包含多个有用的函数来创建和查找特定的模式 (本节的函数有 re.compile, re.search, re.sub, 和 re.ignorecase 或 re.I)。我们来看一个例子:

```
# Count the number of times a pattern appears in a string
string = "The quick brown fox jumps over the lazy dog."
string_list = string.split()
pattern = re.compile(r"The", re.I)
count = 0
for word in string_list:
    if pattern.search(word):
        count += 1
print("Output #38: {0:d}".format(count))
```

第一行给字串变量赋值 The quick brown fox jumps over the lazy dog。下一行将字串分割为单词列表。下一行使用 re.compile 和 re.I 函数, 及原始字串 r 标记, 来创建一个正则表达式称为 regexp。re.compile 函数编译基于文本的模式到正则表达式。并不是总需要编译正则表达式, 但是这是个很好的实践, 因为这可以增加程序的速度。re.I 函数确保模式对大小写敏感并匹配字串中的 “The” 和 “the”。原始字串标记 r, 保证 Python 不处理字串中的特定序列, 如 \, \t, 或 \n。

这意味着不会有意外来交互特定的序列和正则表达式指定序列。本例中没有特定的字串序列。因些 r 不是必须的。但是在正则表达式中使用原始字串标记是很好的实践。下一行创建名为 count 的变量来存贮字串的匹配次数, 初始值为 0。

下一行是个 for 循环, 它遍历列表变量 string_list 中的每一个元素。抓取的第一个元素是单词 “The”, 第二个元素是单词 “quick”, 如此等等, 对于列表中的每个单词。下一行使用 re.search 函数比较列表中的每个单词和正则表达式。函数返回 true, 如果单词与表达式匹配, 否则返回 false。因此, 陈述为 “如果单词匹配正则表达式, count 的值加 1。”

最后, 打印语句打印正则表达式找到的模式 “The” 的次数, 大写敏感, 本例中, 为 2 次。

这看来很吓人！正则表达式的查找功能非常强大，但是它们让人很难读懂（它们曾被称为“write-only language”），所以如果你不理解也不要过于担心，即便是专家也有困难。

当你习惯了正则表达式，它会像游戏一样得到你想要的结果。

我们来看另一个例子：

```
# Print the pattern each time it is found in the string
string = "The quick brown fox jumps over the lazy dog."
string_list = string.split()
pattern = re.compile(r"(?P<match_word>The)", re.I)
print("Output #39:")
for word in string_list:
    if pattern.search(word):
        print("{:s}".format(pattern.search(word).group('match_word')))
```

第二个例子与第一个例子的不同之处在于我们想打印每个匹配字符串而不是匹配次数。要捕获匹配的字符串以使你可以打印到屏幕或文件，我们使用 `(?P<name>)` metacharacter 和 `group` 函数。本例中的大部分代码与第一个例子相同，所以我们侧重于新的部分。

第一个新的代码块为 `(?P<name>)`，是 metacharacter，出现在 `re.compile` 函数内。这个 metacharacter 使得匹配字符串在后面的程序中可用，通过符号组名 `<name>`。

本例中我称组为 `<match_word>`。

最后一个新代码块出现在 `if` 语句中。这个代码块，说得基础点是，“如果结果为 `True`（例如，如果单词匹配模式），则查找 `search` 函数返回的结果并捕获组中的值称为 `match_word`，打印值到屏幕。”

这是最后一个例子：

```
# Substitute the letter "a" for the word "the" in the string
string = "The quick brown fox jumps over the lazy dog."
string_to_find = r"The"
pattern = re.compile(string_to_find, re.I)
print("Output #40: {:s}".format(pattern.sub("a", string)))
```

最后一个例子展示如何使用 `re.sub` 函数将一个模式替换为另一个模式。再一次，大部分代码与前两个例子相似。所有我只关注新的部分。第一个新的代码块将正则表达式赋值给变量 `pattern`，使变量可以传递到 `re.compile` 函数。在 `re.compile` 函数之前将正则表达式赋值给变量并不是必须的，但是，如果你的正则表达式很长，很复杂，将它赋值给变量然后传递给 `re.compile` 函数 使你的代码更加可读。

最后新的代码块出现在最后一行。代码块使用 `re.sub` 函数查找模式，`The`，大写敏感，在名为 `string` 的变量中。并用字母 `a` 替代每个出现的模式。替代的结果为 `a quick brown fox jumps over a lazy dog`。

关于正则表达式函数的更多信息请见 [the Python Standard Library](#) 或 Michael Fitzgerald 的书 [Introducing Regular Expressions](#) (O' Reilly)。

在许多商务应用中，日期是重要的考虑。你可能想知道事件什么时候发生，到事件发生所要的时间，或者两件事中的时间。因为日期是许多应用的中心-因为它们时如此奇怪的数据，以 60，24 为倍数工作，“约二十”，“几似准确的 365 和季度”，Python 有特殊的方法来处理日期。Python 包括 datetime 模块，提供伟大的函数来处理日期和时间。要在你的脚本中使用 datetime 模块提供的函数，在脚本的顶部，前需的 import 语句下增加增加 from datetime import date, time, datetime, timedelta。现在 first_script.py 顶部看来像：

```
#!/usr/bin/env python3
from math import exp, log, sqrt
import re
from datetime import date, time, datetime, timedelta
```

当你导入了 datetime 模块，你就有很多的日期和时间对象和函数可用。一些有用的对象和函数包括 today, year, month, day, timedelta, strftime 和 strptime。这些函数使得捕获个别的日期元素（例如，年，月，或日），要给日期增加或减少特定的时间，创建特定格式的日期字符串，从日期字符串创建 datetime 对象，成为可能。

下面是一些使用 datetime 对象和函数的例子。第一个例子集说明 date 对象和 datetime 对象的不同之处。

```
# Print today's date, as well as the year, month, and day elements
today = date.today()
print("Output #41: today: {0!s}".format(today))
print("Output #42: {0!s}".format(today.year))
print("Output #43: {0!s}".format(today.month))
print("Output #44: {0!s}".format(today.day))
current_datetime = datetime.today()
print("Output #45: {0!s}".format(current_datetime))
```

通过使用 date.today(), 你创建 date 对象，包括年，月，日和星期元素，但不包括时间元素，如小时，分，秒。相反，datetime.today() 包括时间元素。{0!s} 中的 !s 指示传递给打印语句的值应格式化为字符串，即便它是数字。最后，你使用年，月，日，星期，来捕获这些日期元素。

下一个例子说明如何用 timedelta 函数来给日期对象增加或减少特定的时间。

```
# Calculate a new date using a timedelta
one_day = timedelta(days=-1)
yesterday = today + one_day
print("Output #46: yesterday: {0!s}".format(yesterday))
eight_hours = timedelta(hours=-8)
print("Output #47: {0!s} {1!s}".format(eight_hours.days,
eight_hours.seconds))
```

本例中，我们用 timedelta 函数将今天的日期减少 1 天。另外，我们可以使用 days=10, hours=-8, 或 weeks=2 于括号内以创建一个变量，它是未来的 10 天，8 小时前，两同前。

需要记住的一点是，使用 `timedelta` 时，它贮存时间于括号内不同于星期，秒，微秒，然后归一化值，使它们独特。这意味着，分，时，星期都转换为 60 秒，3600 秒，7 天，然后归一化，特别的创建星期，秒，微秒“列”（回想小学数学和 1 的列，10 的列，等）。例如，`hours=-8` 的输出是 `(-1 days, 57, 600 seconds)` 而不是简单的 `(-28, 800 秒)`。这被计算为 `86, 400 秒`（每小时进 3,600 秒 * 每天 24 小时 day） - `28, 800 秒`（每小时 3,600 秒 * 8 小时） = `57, 600 秒`。如你所见，一开始归一化负值输出让人惊讶，特别是修约的时候。

第三个例子展示如何从一个日期对象减另一个日期对象。减的结果是 `datetime` 对象，以天、时、分、秒表示的差值。例如，本例的结果是 `“1 day, 0:00:00”`：

```
# Calculate the number of days between two dates
date_diff = today - yesterday
print("Output #48: {}".format(date_diff))
print("Output #49: {}".format(str(date_diff).split()[0]))
```

有时，你只想要这个结果的数值元素。例如，本例中你只要数值 1。从结果中捕获这个数字的一种方法是用你学过的处理字符串的函数。Str 函数将结果转为字符串，split 函数分割字符串，形成子字符串的列表。[0]表示“捕获列不中的第一个元素，”本例中是数字 1。下一节中我们将再次见到符号[0]，它覆盖列表，因为它说明列表索引并展示如何追踪列表元素。

第四个例子集展示如可使用 `strftime` 函数从日期对象创建特定格式字符串：

```
# Create a string with a specific format from a date object
print("Output #50: {}".format(today.strftime('%m/%d/%Y')))
print("Output #51: {}".format(today.strftime('%b %d, %Y')))
print("Output #52: {}".format(today.strftime('%Y-%m-%d')))
print("Output #53: {}".format(today.strftime('%B %d, %Y')))
```

到写本章时，四种打印今天的格式为：

```
01/28/2016
Jan 28, 2016
2016-01-28
January 28, 2016
```

这四个例子展示如何使用格式符号，包括%Y,%B,%b,%m, and %d,以创建不同的日期字符串格式。你可以在 [Python Standard Library 的“datetime—Basic date and time types”](#) 一节看到 `datetime` 使用的其它格式符号

```
# Create a datetime object with a specific format
# from a string representing a date
date1 = today.strftime('%m/%d/%Y')
date2 = today.strftime('%b %d, %Y')
date3 = today.strftime('%Y-%m-%d')
date4 = today.strftime('%B %d, %Y')
```

```
# Two datetime objects and two date objects
# based on the four strings that have different date formats
print("Output #54: {!s}".format(datetime.strptime(date1, '%m/%d/%Y')))
print("Output #55: {!s}".format(datetime.strptime(date2, '%b %d, %Y')))
# Show the date portion only
print("Output #56: {!s}".format(datetime.date(datetime.strptime\
(date3, '%Y-%m-%d'))))
print("Output #57: {!s}".format(datetime.date(datetime.strptime\
(date4, '%B %d, %Y'))))
```

第五个例子集展示如何使用 `strptime` 函数从特定格式的日期字符串创建 `datetime` 对象。本例中, `date1`, `date2`, `date3`, 和 `date4` 为字符串变以不同的格式表示今天的日期。前两个打印语句为转换前两个字符串变量 `date1` 和 `date2` 到 `datetime` 对象的结果。要正确的工作, `strptime` 函数使用的格式需要匹配被传递的字符串变量的格式。这些打印语句的结果是 `datetime` 对象, 2014-01-28 00:00:00。有时, 你只对 `datetime` 对象的日期感兴趣。这种情况下, 你可用嵌套函数, `date` 和 `strptime`, 如最后两个打印语句所示, 来转换 `date-string` 变量到 `datetime` 对象然后返回 `datetime` 对象的日期部分。打印语句的结果是 2014-01-28。当然, 你不想立即打印这个值。你可以赋给日期新的变量然后用变量进行计算产生商务日期间和时间。

列表 Lists

列表在商务分析中非常流行。你可能有客户、产品的、资产的、销售图形等等的列表。但是, 列表—python 中对象的有序集合比那更灵活! 所有这些类型的列表都包含一个相似的对象(如, 包含客户名称的字符串或代表销售图的浮点数), 但是 Python 中的列表并不这么简单。

它们可以包括数字, 字符串, 其它列表, 元组和字典(本章后面描述)的任意组合。因为它们在商务分析中的流行, 灵活, 重要, 很有必要知道如何在 Python 操作列表。如你所期望的, Python 提供了许多函数和操作符来处理列表。下一节说明如何用它们中最常用的最有用的函数和操作符。

创建列表 Create a list

```
# Use square brackets to create a list
# len() counts the number of elements in a list
# max() and min() find the maximum and minimum values
# count() counts the number of times a value appears in a list
a_list = [1, 2, 3]
print("Output #58: {}".format(a_list))
print("Output #59: a_list has {} elements.".format(len(a_list)))
print("Output #60: the maximum value in a_list is
{}.".format(max(a_list)))
print("Output #61: the minimum value in a_list is
{}.".format(min(a_list)))
another_list = ['printer', 5, ['star', 'circle', 9]]
```

```
print("Output #62: {}".format(another_list))
print("Output #63: another_list also has {} elements.".format\
(len(another_list)))
print("Output #64: 5 is in another_list {}
time.".format(another_list.count(5)))
```

这个例子展示如何创建两个简单的列表，a_list 和 another_list。你可以在方括弧内放置元素来创建列表。a_list 包括数字 1, 2, 和 3。another_list 包括字符串 printer, 数字 5 和具有两个字符串和一个数字的列表。这个例子也展示了如何使用四个列表函数 len, min, max, 和 count。

Len 函数显示列表元素的个数。min 和 max 显示列表中的最大值和最小值。count 显示某个值在列表中出现的次数。

索引值 Index values

```
# Use index values to access specific elements in a list
# [0] is the first element; [-1] is the last element
print("Output #65: {}".format(a_list[0]))
print("Output #66: {}".format(a_list[1]))
print("Output #67: {}".format(a_list[2]))
print("Output #68: {}".format(a_list[-1]))
print("Output #69: {}".format(a_list[-2]))
print("Output #70: {}".format(a_list[-3]))
print("Output #71: {}".format(another_list[2]))
print("Output #72: {}".format(another_list[-1]))
```

本例展示你可以如何使用列表的索引值，或者 indices，来访问特定的列表元素。列表索引值自 0 开始，所以你可以通过列表名字后面放方括弧，方括弧内放 0 来访问列表的第一个元素。本例的第一个打印语句，print a_list[0]，打印 a_list 的第一个元素，即数字 1。a_list[1] 访问第二个元素，a_list[2] 访问第三个元素，如此等直到最后一个元素。本例也展示如何使用负索引来访问列表尾部的元素。列表最后一个元素的索引值从 -1 开始，所以你可以能过在列表名后的方括弧内加 -1 来访问最后一个元素。第四个打印语句，print a_list[-1] 打印列表的最后一个元素，即数值 3。a_list[-2] 访问倒数第二个元素，a_list[-3] 访问倒数第三个元素，如此等直到列表的第一个元素。

列表切片 List slices

```
# Use list slices to access a subset of list elements
# Do not include a starting index to start from the beginning
# Do not include an ending index to go all of the way to the end
print("Output #73: {}".format(a_list[0:2]))
print("Output #74: {}".format(another_list[:2]))
print("Output #75: {}".format(a_list[1:3]))
print("Output #76: {}".format(another_list[1:]
```

本例展示如何用列表切片来访问列表元素的子集。你可以通过列表名称后的方括弧内放置两个由冒号分开的索引值来创建切片。列表切片访问列表元素自第一个索引值到第二个索引值前的一个元素。例如，第一个打印语句，print

`a_list[0:2]`，说得基础些是“打印 `a_list` 中的元素，它们的索引值为 0 和 1。”这个打印语句打印 `[1, 2]`，因为它们是前二个元素。本例也展示你不需要第一个索引值如果切片从列表第一个值开始，你也不需要包括第二个索引值如果切片到列表的最后一个元素。例如，最后的打印语句，`print another_list[1:]`，说得基础些是，“打印 `another_list` 中自第二个元素开始的所有的元素。”这个打印语句打印出 `[5, ['star', 'circle', 9]]`，因为它们是列表的最后的两个元素。

拷贝列表 Copy a list

```
# Use [:] to make a copy of a list
a_new_list = a_list[:]
print("Output #77: {}".format(a_new_list))
```

这个例子展示如何拷贝列表。当你需要用某种方法操作列表，如增加或删除列表中的元素或排序列表，而你又不想改变原有的列表时，这种能力是很重要的。要拷贝一个列表，列表名称的方括号内放置冒号，然后赋值给另一个列表。本例中，`a_new_list` 是 `a_list` 的拷贝。所以你可以从 `a_new_list` 增加或删除元素或排序 `a_new_list` 而不需要修改 `a_list`。

拼接列表 Concatenate lists

```
# Use + to add two or more lists together
a_longer_list = a_list + another_list
print("Output #78: {}".format(a_longer_list))
```

这个例子展示如何拼接两个或多个列表。当你需要访问相似的列表，但是你想在分析前组合所有的列表时，这种能力是非常重要的。例如，因为你的数据排序，你需要从数据源产生销售图列表，从另一个数据源产生另一个销售图列表。要拼接这两个列表进行分析，将两个列表的名称用操作符+相加，然后将和赋值给另一个变量。本例中，`a_longer_list` 含有 `a_list` 和 `another_list` 的元素拼接成一个长的列表。

使用 in 和 not in

```
# Use in and not in to check whether specific elements are or are not in
a list
a = 2 in a_list
print("Output #79: {}".format(a))
if 2 in a_list:
print("Output #80: 2 is in {}".format(a_list))
b = 6 not in a_list
print("Output #81: {}".format(b))
if 6 not in a_list:
print("Output #82: 6 is not in {}".format(a_list))
```

这个例子展示如何使用 `in` 和 `not in` 来检查某个元素是不是在列表内。这些表达式的结果是 `true` 或 `false`。这种能力对于商务应用来说很重要，因为它们增加有意义的业务逻辑。例如，它们常用于 `if` 语句中，如“如果 `SupplierList` 中有 `SupplierY` 则进行一些操作，否则进行另一些操作。”后面的章节中我们将

看到更多的例子和其它的控制流表达式

append, remove, pop

```
# Use append() to add additional elements to the end of the list
# Use remove() to remove specific elements from the list
# Use pop() to remove elements from the end of the list
a_list.append(4)
a_list.append(5)
a_list.append(6)
print("Output #83: {}".format(a_list))
    a_list.remove(5)
print("Output #84: {}".format(a_list))
a_list.pop()
a_list.pop()
print("Output #85: {}".format(a_list))
```

这个例子展示如何从列表中增加或删除列表。Append 方法增加一个元素到列表尾部。你可以按照特定的业务规则用这种方法创建列表。例如，要创建 CustomerX 的采购列表，你可以创建空的列表称为 CustomerX，扫描所有采购的主列表，当程序发现主列表中的 CustomerX 时，增加采购值到 CustomerX 列表中。Remove 方法将指定的值从列表中删除。你可以用这种方法按业务规则来删除列表中的错误或输入错误。本例中，remove 方法将数字 5 从 a_list 中删。

Pop 从列表的尾部删除列表。类似 remove 方法，你可以根据业务逻辑用 pop 方法自列表尾部删除错误或输入错误。本例中，两次调用 pop 方法自列表尾部删除数字 6 和 4。

reverse

```
# Use reverse() to reverse a list in-place, meaning it changes the list
# To reverse a list without changing the original list, make a copy first
a_list.reverse()
print("Output #86: {}".format(a_list))
a_list.reverse()
print("Output #87: {}".format(a_list))
```

本例展示如何使用 reverse 函数来倒序列表。“Inplace”表示与原列表相反的变换。例如，第一次调用 reverse 函数将 a_list 变为 [3, 2, 1]。第二次调用 reverse 函数，将 a_list 变为原有的顺序。要使用列表的倒序拷贝而不想修改原有的列表，首先拷贝列表，然后对拷贝进行例序。

sort

```
# Use sort() to sort a list in-place, meaning it changes the list
# To sort a list without changing the original list, make a copy first
unordered_list = [3, 5, 1, 7, 2, 8, 4, 9, 0, 6]
print("Output #88: {}".format(unordered_list))
list_copy = unordered_list[:]
    list_copy.sort()
```

```
print("Output #89: {}".format(list_copy))  
print("Output #90: {}".format(unordered_list))
```

这个例子展示如何用 `sort` 函数来在位排序列表。与 `reverse` 方法相似，这个在位排序改变原有的列表到新的排序后的列表。要使用排序的列表拷贝而不修改原有的列表，首先拷贝列表，然后排序拷贝。

sorted

```
# Use sorted() to sort a collection of lists by a position in the lists  
my_lists = [[1, 2, 3, 4], [4, 3, 2, 1], [2, 4, 1, 3]]  
my_lists_sorted_by_index_3 = sorted(my_lists, key=lambda index_value:\  
index_value[3])  
print("Output #91: {}".format(my_lists_sorted_by_index_3))
```

这个例子展示用 `sorted` 函数组合关键函数来排序列表，通过指定每个列表的索引位置。

关键函数指明用于排序列表的关键字。本例中，`key` 是一个 `lambda` 函数，即是说，“通过索引位置 3 的值来排序列表（如，列表中的第四人值）”（对于 `lambda` 函数我们在后面讨论）。

通过使用 `sorted` 函数，用每个列表的第四个值作为排序的 `key`，第二个列表 `[4, 3, 2, 1]` 成为第一个列表，第三个列表 `[2, 4, 1, 3]` 成为第二个列表，第一个列表 `[1, 2, 3, 4]` 成为第三个列表。另外注意，尽管 `sort` 在位排序列表，改变原用的列表，`sorted` 返回新的列表并不改变原有的列表。

下面的排序的例子使用标准的 `operator` 模块，它提供了排序列表，元组，字典的函数，通过多个 `keys`。要使用在你的脚本中使用 `operator` 模块的 `itemgetter` 函数，在你的脚本本顶部增加 `from operator import itemgetter`：
#!/usr/bin/env python3

```
from math import exp, log, sqrt  
import re  
from datetime import date, time, datetime, timedelta  
from operator import itemgetter
```

通过导入 `operator` 模块的 `itemgetter` 函数，你可以排序列表集合，通过每个列表的多个位置：

```
# Use itemgetter() to sort a collection of lists by two index positions  
my_lists = [[123, 2, 2, 444], [22, 6, 6, 444], [354, 4, 4, 678], [236, 5, 5, 678],  
\[578, 1, 1, 290], [461, 1, 1, 290]]  
my_lists_sorted_by_index_3_and_0 = sorted(my_lists,  
key=itemgetter(3, 0))  
print("Output #92: {}".format(my_lists_sorted_by_index_3_and_0))
```

本例展示如何使用 `sorted()` 函数和 `itemgetter` 函数排序列表集合，通过每个列表的多个索引值。`key` 函数指定用于排序列表的 `key`。本例中，`key` 是 `itemgetter` 函数并且它有两个索引值，`three` 和 `zero`。这句话，说，“首先通过索引位置三排序列表，然后，维护排序的列表，然后通过索引位置 0 排序列表。”通过多们位置来排序列表和其它数据容器的能力是很有用的，因为你能常需要通

过多个值来排序数据。例如，如果你有每日销售业务数据，你想先通过日期然后通过业务量来排序数据。或者，如果你有供应商数据，你想先通过供应商的名字然后通过每个供应商的接收日期来排序数据。

Sorted 和 itemgetter 函数提供这种能力。更多处理列表的函数的信息，请见 [Python Standard Library](#)。

元组 Tuples

元组与列表相似，只是它们不能被修改。因为元组不能被修改，所以没有修改元组的函数。有两种相似的数据结构看起来很奇怪，但元组有重要的作用，可能修改的列表不能拥有。例如，作为字典的索引。元组比列表少用，所以我们简单的过一下这个主题。

创建元组 Create a tuple

```
# Use parentheses to create a tuple
my_tuple = ('x', 'y', 'z')
print("Output #93: {}".format(my_tuple))
print("Output #94: my_tuple has {} elements".format(len(my_tuple)))
print("Output #95: {}".format(my_tuple[1]))
longer_tuple = my_tuple + my_tuple
print("Output #96: {}".format(longer_tuple))
```

这个例子展示如何创建元组。你可以通过在圆括号中放置元素来创建元组。本例也展示你可以使用许多与列表相同的函数和操作符来处理元组。例如，len 显示元组元素个数，元组索引和切片来访问元组中特定元素，+操作符拼接元组。

Unpack tuples

```
# Unpack tuples with the lefthand side of an assignment operator
one, two, three = my_tuple
print("Output #97: {0} {1} {2}".format(one, two, three))
var1 = 'red'
var2 = 'robin'
print("Output #98: {} {}".format(var1, var2))
# Swap values between variables
var1, var2 = var2, var1
print("Output #99: {} {}".format(var1, var2))
```

这个例子展示元素的一个有趣的方面，unpacking。它可以解包元组的元素到单独立变量，通过放置变量到赋值符的左边。本例中，字符串 x, y, 和被解包到 z 变量 one, two, 和 three。这个函数对于变换不同变量的值有用。本例的最后，var2 的值赋给 var1, var1 值赋给 var2。python 同时计算两部分。这样，red robin 成为 robin red。

将元组转为列表或将列表转为元组

```
# Convert tuples to lists and lists to tuples
my_list = [1, 2, 3]
my_tuple = ('x', 'y', 'z')
```

```
print("Output #100: {}".format(tuple(my_list)))  
print("Output #101: {}".format(list(my_tuple)))
```

最后，可以将元组转为列表，也可以将列表转为元组。这种功能类似于 `str` 函数，可以将元素转为字符串。要将列表转为元组，将列表的名字放在 `tuple()` 函数内。相似的，要转换元组为列表，将元组的名字放在 `list()` 函数内。关于元组的更多信息，请见 [Python Standard Library](#)。

字典 Dictionaries

Python 里的字典本质上是列表，包含成对的信息，具有独特的标识。像列表，字典在许多商务分析中非常流行。商务分析可能包括客户字典（客户编码为键），产品字典（产品序列号编码为键）。资产字典，销售图字典，等。python 里，这些数据结构称为字典，便它们也称为 `associative arrays`, `key-value stores`, 及其它语言称为 `hashes`。

列表和字典对于许多商务应用都是有用的数据结构，但是列表和字典有一些重要的不同之处，要有效的使用字典你得理解它们。

- 在列表里，你用连续的整数访问各个值，称为索引。在字典里，你用整数，字符串，或其它 python 对象，称为键，访问个别的值。这使字典比列表更有用，当需要唯一的键的情况下，并且不连续的整数，更有意义的映射到值。在列表里，值是有序的，因为索引是连续的整数。在字典里，值是无序的因为索引不只是数值。你可以在字典里定义项目的顺序，但是字典没有内置的顺序。
- 在列表里，给不存在的位置（索引）赋值是非法的。在字典里，可以按需生成新的位置（键）。
- 因为它们是无序的，字典可以提供更快的响应，当你需要索引或增加值时（计算机不需要重新赋值给索引，当你插入项目时）。这是重要的考虑，当你要处理越来越多的数据时。

除了它们的流行，灵活，在许多商务应用中重要，重要的是要知道如何在 python 中处理字典。下面的例子说明如何用一些最为常见和有用的函数和操作符来处理字典。

创建字典 Create a dictionary

```
# Use curly braces to create a dictionary  
# Use a colon between keys and values in each pair  
# len() counts the number of key-value pairs in a dictionary  
empty_dict = { }  
a_dict = {'one':1, 'two':2, 'three':3}  
print("Output #102: {}".format(a_dict))  
print("Output #103: a_dict has {!s} elements".format(len(a_dict)))  
another_dict = {'x':'printer', 'y':5, 'z':['star', 'circle', 9]}  
print("Output #104: {}".format(another_dict))  
print("Output #105: another_dict also has {!s} elements"\  
.format(len(another_dict)))
```

本例展示如何创建字典。要创建空的字典，在等号的左边给定字典的名字，在等号的右边加上花括号。本例中的第二个字典，`a_dict`，说明一种方法来增加

键和值到小字典。a_dict 展示用帽号分开键和值，键值对由逗号分开，放在花括号内。键是字符串，带单括号或双括号，值可以是数字，字符串，列表其它字典，其它 Python 对象。a_dict 中，值为整数，但是在 another_dict 中上，值为字符串，数字，列表。最后，本例展示 len 函数显示键值对的数目。

访问字典中的值 Access a value in a dictionary

```
# Use keys to access specific values in a dictionary
print("Output #106: {}".format(a_dict['two']))
print("Output #107: {}".format(another_dict['z'])
```

要访问特定的值，写出字典的名字，开放的方括号，特定的键名，和闭的方括号。本例中，a_dict['two']的结果是整数2，another_dict['z']的结果是列表['star', 'circle', 9]。

拷贝 copy

```
# Use copy() to make a copy of a dictionary
a_new_dict = a_dict.copy()
print("Output #108: {}".format(a_new_dict))
```

要拷贝字典，添加 copy 函数到字典名的末尾，赋值给新的字典。本例中，a_new_dict 是原字典 a_dict 的拷贝。

keys, values, 和 items

```
# Use keys(), values(), and items() to access
# a dictionary's keys, values, and key-value pairs, respectively
print("Output #109: {}".format(a_dict.keys()))
a_dict_keys = a_dict.keys()
print("Output #110: {}".format(a_dict_keys))
print("Output #111: {}".format(a_dict.values()))
print("Output #112: {}".format(a_dict.items()))
```

要访问字典中的键，在字典名后添加 keys 函数。结果为键的列表。要访问字典的值，在字典名的后面添加 values 函数，结果为字典值的列表。要访问字典的键和值，在字典名后添加 items。结果是键值对名的元组的列表。例如，a_dict.items()的结果是 [('three', 3), ('two', 2), ('one', 1)]。在下一节的控制流中，我们会看到如何用 for 循环解包和访问字典的所有的键和值。

使用 in, not in, 和 get

```
if 'y' in another_dict:
    print("Output #114: y is a key in another_dict: {}".format(another_dict.keys()))
if 'c' not in another_dict:
    print("Output #115: c is not a key in another_dict: {}".format(another_dict.keys()))
print("Output #116: {}".format(a_dict.get('three')))
print("Output #117: {}".format(a_dict.get('four')))
print("Output #118: {}".format(a_dict.get('four', 'Not in dict')))
```

本例展示用两路不同的方法检查特定的键是不是在字典内。第一种方法是用 if 语句和 in 或 not in 以及字典名检查物定的键。使用 in, if 语句检查 y 是不是 another_dict 的键。如果结果为真(例如, 如果 y 是 another_dict 的键), 则打印语句执行。否则不执行打印语句。这些 if in 和 if not in 语句通常用于检查键的存在, 组合一些额外的符号, 来添加新的键到字典。我们后面将看到增加键到字典的例子。

行首缩进是什么? What' s with the Indentation?

你会注意到 if 语句后的行是行首缩进的。Python 用行首缩进来告知指令不是一个逻辑块—当 if 语句为真进, if 语句后的所有行首缩进被执行。Python 解释器继续运行。你将会在后面看到相样的行首缩进用作别的逻辑块。但是现在重要的是要注意到 python 使用行首缩进是有意义的, 你要符合它。如果你用 IDE 或者像 Sublime Text 之类的编辑器, 每次使用 Tab 键时软件会告诉你一致的空格数。如果你用一般的本文编辑器如 Notepad, 你要小心使用相同数量的空格的首行缩进(通常为四)。最后要注意的是文本编辑器偶然会插入 tab 符号而不是正确数量的空格, 让你很困惑, 因为代码看起来是对的, 但是你会得到错误信息(类似于“Unexpected indent in line 37”)。通常, python 使用行首缩进让代码更易读因为你更容易知道程序的决策进程在哪里了。

另一种检查特定的键并追踪键的相应值的方法是用 get 函数。不同于上面的方法检测键, get 函数返回键对应的值, 如果键在字典中, 否则返回 None, 如果键不在字典中。另外, get 函数也允许第二个可选的 参数, 即如果键不存在返回的值。这种方法可能返回一些不是 None 的值如果键不存在于字典。

Sorting

```
# Use sorted() to sort a dictionary
# To sort a dictionary without changing the original dictionary,
# make a copy first
print("Output #119: {}".format(a_dict))
dict_copy = a_dict.copy()
ordered_dict1 = sorted(dict_copy.items(), key=lambda item: item[
print("Output #120 (order by keys): {}".format(ordered_dict1))
ordered_dict2 = sorted(dict_copy.items(), key=lambda item: item[1])
print("Output #121 (order by values): {}".format(ordered_dict2))
ordered_dict3 = sorted(dict_copy.items(), key=lambda x: x[1],
reverse=True)
print("Output #122 (order by values, descending):
{}".format(ordered_dict3))
ordered_dict4 = sorted(dict_copy.items(), key=lambda x: x[1],
reverse=False)
print("Output #123 (order by values, ascending):
{}".format(ordered_dict4))
```

本例展示如何用不同的方法排序字典。如本节刚开始所述, 字典不需要明确的顺序, 但是你可以用上面的代码块来排序字典。可以基于字典的键和值来排序字典。如果值为数字, 则可以升序和降序。本例中, 我用 copy 函数来拷贝 a_dict。

拷贝称为 dict_copy。字典的拷贝可以确保原始字典不变。下一行包括 sorted 函数，items 函数的结果是元组的列表，lambda 作为 key 用于 sorted 函数。这一行有很多步，所以我们业解包一下。这一行的目的是要排序键值元组的列表，它是 items 函数的结果。key 是排序的标准，它等于一些简单的 lambda 函数。（lambda 函数是短函数，它在运行进返回表达式）。在 lambda 函数中，item 是参数，它指向 items 函数返回的键值元组。返回的表达式出现在帽号后。这个表达式是 item[0]，所以元组的第一个元素（如，键）返回并作为 sorted 函数的键。总之，这行代码说得基础点是，“按升序排序字典中的键值对，基于字典中的键”。下一个 sorted 函数用 item[1] 而不是 item[0]，所以这按升序排序键值对，基于字典的值。sorted 函数的后两个版本与前面的版本相似，因为这三个版本用字典值作为排序的 key。因为这个字典的值是数字的，它们可以升序或降序排列。后两个版本展示如何用 sorted 函数的 reverse 参数来指明输出是升序还是降序。reverse=True 对应于降序，所以键值对按它们的值降序排列。

关于字典处理的更多信息见 [Python Standard Library](#)。

控制流 Control flow

控制流对于在程序中包令有意义的业务逻辑来说是非常重要的元素。许多的商务过程和分析依赖于诸如“如果客户花更多特定的数量，则这么做这么做”或者“如果销售分类为 A 则编码为 X，否则销售为分类 B 编码它们为 Y，否则编码为 Z。”之类的逻辑。这些逻辑语句可以在代码中用控制元素表达。Python 提供多个控制流元素，包括 if-elif-else 语句，for 循环，range 函数，while 循环。如它们的名字所示，if-else 语句使能逻辑如“如果这样则这么做，否则做别的。”。else 代码块不是必须的，但可以使你的代码更明了。for 循环使你可以迭代队列中的所有值，队列可以是列表，元组或字符串。你可以用 range 函数加上 len 函数，作用于列表来产生索引值的序列，然后你可以用 for 循环。最后，如果 while 的条件为真则 while 循环执行主体上的代码。

if-else

```
# if-else statement
x = 5
if x > 4 or x != 9:
    print("Output #124: {}".format(x))
else:
    print("Output #124: x is not greater than 4")
```

这个例子展示简单的 if-else 语句。if 条件检查 x 是否大于或 x 不等于（!= 是“不等于”的缩写）。加上 or 操作符，当发现表达式为真时计算停止。本例中，x 等于 5，而 5 大于 4，所以 x != 9 不用计算。第一个条件 x > 4 为真，所以执行 print x，打印的结果为数字 5。如果 if 块中的条件都不为真，则执行 else 块中的 print 语句。

if-elif-else

```
# if-elif-else statement
if x > 6:
    print("Output #125: x is greater than six")
```

```
elif x > 4 and x == 5:  
print("Output #125: {}".format(x*x))  
else:  
print("Output #125: x is not greater than 4")
```

第二个例子展示更复杂一点的 if-elif-else 语句。与前面的例子相似，if 块简单的检查 x 是否大于 6。如果这个条件为真，则计算停止，执行相应的 print 语句。正如发生的那样，5 不大于 6，所以在计算下一个 elif 语句。这个语句检查 x 是否大于 4，而 x 为 5。加上 and 操作符，当表达式为假时停止计算。本例中，x 等于 5，5 大于 4，x 评估为 5，所以执行 print x*x，打印的结果是 25。因为我们使用等号给对象赋值，我们用双等号 (==) 评估是否相等。如果 if 或 elif 块都不为真，则 else 块中的打印语句执行。

for loops

```
y = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct',  
\ 'Nov', 'Dec']  
z = ['Annie', 'Betty', 'Claire', 'Daphne', 'Ellie', 'Franchesca', 'Greta',  
\ 'Holly', 'Isabel', 'Jenny']  
print("Output #126:")  
for month in y:  
    print("{}s".format(month))  
    print("Output #127: (index value: name in list)")  
for i in range(len(z)):  
    print("{0!s}: {1:s}".format(i, z[i]))  
    print("Output #128: (access elements in y with z's index values)")  
for j in range(len(z)):  
    if y[j].startswith('J'):  
        print("{}s".format(y[j]))  
        print("Output #129:")  
for key, value in another_dict.items():  
    print("{0:s}, {1}".format(key, value))
```

这四个 for 循环的例子说明如何使用 for 循环来遍历队列。这对于后面的章节和商务应用来说来说是重要的能力。

第一个 for 循环的例子展示的基本符号是 for variable in sequence, do something。variable 是队列中的每个值的临时占位符，它只有在 for 循环中被识别。本例中，变量名是 month。sequence 是你遍历的队列名。本例中，队列名是 y，它是 months 的列表。因此，本例说，“对于 y 中的每个值，打印它的值”。第二个 for 循环的例子展示如何用 range 函数加上 len 函数产生索引的序列，你可以把它用于 for 循环。要理解这些复合函数的交互，要全面的评估它们。len 函数计算列表 z 中的值的数目，即 10。然后 range 函数产生一序列的整数，自 0 起到小于 len 函数的结果，本例中，为 0 到 9。因此，这个 for 循环基本是说，“对于自 0 到 9 的一序列整数 i，打印整数 i 接着空格接着列表 z 中索引为 i 的值。如你所见，for 循环中使用 range 函数加上 len 函数在本书中有许多的例子，因为这种组合对于许多的商务应用非常有用。

第三个 for 循环的例子展示使用一个队列的索引值来访问另一个队列的相

同索引的值。它也展示如何在 for 循环中包含 if 语句来引入业务逻辑。本例中，我使用 range 和 len 函数产生索引形成列表 z。然后，if 语句检查具有这些索引的值是否在列表 y 中 (y[0]= 'Jan', y[1]='Feb', ..., y[9]= 'Oct')，以字母 J 开始。

最后一个 for 循环展示一种方法来遍历和访问字典中的键值对。for 循环的第一行，items 函数返回字典的键值对元组。for 循环中的 key 和 value 变量返回这些值。主体中的 print 语句包含 srt 函数以确保每个键值为字串并打印每个键值对，由空格和行分开打印。

紧凑的 for 循环：列表，集合，和字典推导式 comprehensions

Python 中，列表，集合，字典推导式是一种紧凑的书写 for 循环的方法。列表推导式出现在方括号内，而集合推导式和字典推导式出现在花括号内。所有推导式可以包含条件逻辑（如 if-else 语句）。

列表推导式 List comprehension.

下面的例子展示如何使用列表推导式选择满足条件的列表的子集。

```
# Select specific rows using a list comprehension
my_data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
rows_to_keep = [row for row in my_data if row[2] > 5]
print("Output #130 (list comprehension): {}".format(rows_to_keep))
```

本例中，这个列表推导式是说，“对于 my_data 的每一行，如果索引位置 2 的值大于 5 则保留行。” 因为 6 和 9 大于 5, 所以 rows_to_keep 中保留的列表为 [4, 5, 6] 和 [7, 8, 9]。

集合推导式 Set comprehension.

下面的例子展示如何使用集合推导式来选择元组列表中的独特元组集合。

```
# Select a set of unique tuples in a list using a set comprehension
my_data = [(1, 2, 3), (4, 5, 6), (7, 8, 9), (7, 8, 9)]
set_of_tuples1 = {x for x in my_data}
print("Output #131 (set comprehension): {}".format(set_of_tuples1))
set_of_tuples2 = set(my_data)
print("Output #132 (set function): {}".format(set_of_tuples2))
```

本例中，集合推导式是说，“对于 my_data 中的每一个元组，如果它是独特的元组则保留。” 你能看出表达式为集合推导式而不是列表推导式因为它包含花括号而不是方括号，它也不是字典推导式因为它没有键值对符号。

本例中的第二个 print 语句展示你可以获得与集合推导式相同的结果，使用 python 的内置 set 函数。对于本例，使用内置的 set 函数很合理因为它比集合推导式更紧凑更易读。

字典推导式 Dictionary comprehension.

下面的例子展示如何使用字典推导式来选择满足特定条件的字典的键值对的子集。：

```
# Select specific key-value pairs using a dictionary comprehension
my_dictionary = {'customer1': 7, 'customer2': 9, 'customer3': 11}
```

```
my_results = {key : value for key, value in my_dictionary.items() if \
value > 10}
```

```
print("Output #133 (dictionary comprehension): {}".format(my_results))
```

本例中，字典推导式是说，“对于 my_dictionary 中的每一个键值对，如果值大于 10 则保留键值对。”因为值 11 大于 10，my_results 中保留的键值对为 {'customer3': 11}。

while loops

```
print("Output #134:")
```

```
x = 0
```

```
while x < 11:
```

```
print("{}!".format(x))
```

```
x += 1
```

本例展示使用 while 循环来打印自 0 到 10 的数字。x=0 初始化变量 x 为 0。然后 while 循环评估 x 是否小于 11。因为 x 小于 11，while 循环的主体打印变量 x 的值，接着空格。然后将变量 x 的值加 1。再次，while 循环评估 x，现在 x 为 1，小于 11。因为这样，所以 while 循环的主体再一次执行。直到 x 的值由 10 增加到 11。现在，while 循环评估 x 是否小于 11 的时候，表达式评估为假，不执行 while 循环的主体。当你知道主体要执行多少次的时候使用 while 是很有用的。通常你不知道主体要执行多少次，所以使用 for 循环更有用。

函数 Functions

有时候你会发现写你自己的函数比使用 python 内置的函数或别人写的函数更有用。

例如，你注意到你不断的重复代码块的时候，你可能会转向考虑使用函数。这种情况，函数可能存在于 python 库中或别的导入模块中。如果函数已存在，则使用现有的函数很合理。然而，有时候你想要的函数可能没有现存的，这时你需要创建自己的函数。

要在 Python 创建函数，行首要以 def 关键字开始，接着是函数的名称，接着是圆括号，接着是冒号。构成函数主体的代码需要行首缩进。最后，如果函数需要返回一个或多个值，则用 return 关键字来返回函数的结果。下面的例子展示如何在 python 中创建和使用函数。

```
# Calculate the mean of a sequence of numeric values
```

```
def getMean(numericValues):
```

```
return sum(numericValues)/len(numericValues) if len(numericValues) > 0
else float('nan')
```

```
my_list = [2, 2, 4, 4, 6, 6, 8, 8]
```

```
print("Output #135 (mean): {}".format(getMean(my_list)))
```

这个例子展示如何创建函数来计算一序列数值的平均值。函数的名称为 getMean。在括号内有一个短语代表传递给函数的一序列数字。这是在函数内定义的变量。在函数内，平均值由数字的和除以数字的数目得到。另外，我用 if-else 语句检查队列是否有值。如果有，则返回平均值。如果没有，则返回 nan（例如，不是一个数字）。

如果我忽略 if-else 语句并且队列中刚好没有值，测程序抛出除零的错误。

最后，return 关键字返回函数的结果。本例中，my_list 有 8 个数字。my_list 传递给 getMean() 函数。8 个数字的和为 40，40 除以 8 等于 5。因此，打印语句打印出整数 5。

你可以猜到，已有求平均值值的函数存在，如 NumPy 中就有。所以你可以得到相同的结果，通过导入 NumPy 并使用它的函数 mean。

```
import numpy as np
print np.mean(my_list)
```

再次，当你需要的函数已存在于 pthon 或别的模块时，使用现存的已测试过的函数是合理的。Google 或 Bing 中查找 “<a description of the functionality you’re looking for> Python function” 是你的朋友。但是你要完成你自己的特定业务时，知道如何创建函数是值得的。

Exceptions

编写稳健的程序的一个重要方面是有效的处理错误和异常。你写的程序通常假定要处理的数据类型，但是如查有些数据不符合你的假定，会使程序抛出错误。Python 有一些内置的意外。一些常见的意外为 IOError, IndexError, KeyError, NameError, SyntaxError, TypeError, UnicodeError, 以及 ValueError。

你可以在网上找到更多的意外，见 [Python Standard Library “Built-in Exceptions”](#) 一节。

使用 try-except 处理错误信息是你的第一道防线。可以让你的程序继续运行即便数据不是很好。

下一节展示两种版本的 try-except 块来有效的捕获和处理意外。这些例子修改自上一节的函数例子，展示如何用 try-except 处理空的列表而不是用 if 语句。

try-except

```
# Calculate the mean of a sequence of numeric values
def getMean(numericValues):
    return sum(numericValues)/len(numericValues)
my_list2 = [ ]
# Short version
try:
    print("Output #138: {}".format(getMean(my_list2)))
except ZeroDivisionError as detail:
    print("Output #138 (Error): {}".format(float('nan')))
    print("Output #138 (Error): {}".format(detail))
```

在这个版本里，函数 getMean() 并不包括 if 语句来检查队列是否包含任务值。如果队列是空的，如 my_list2，则应用函数返回 ZeroDivisionError 结果。要使用 try-except 块，将你要执行的代码放在 try 块，然后用 except 块处理任何潜在的错误并打印错误帮助信息。一些时候，意外会有关联值。你可以访问这些意外值通过在行 except 包含 as 短语然后打印你给意外值的名字。因为 my_list2 不含任何值，except 执行，它打印 nan，然后是 Error: float division by zero。

try-except-else-finally

```
# Long version
try:
result = getMean(my_list2)
except ZeroDivisionError as detail:
print "Output #142 (Error): " + str(float('nan'))
print "Output #142 (Error):", detail
else:
print "Output #142 (The mean is):", result
finally:
print "Output #142 (Finally): The finally block is executed every time"
```

这种长的版本包括 else 和 finally 块，除了 try 和 except 块。如果 try 块成功则执行 else。因此，如果传递给 getMean() 函数的数字队列包含任何数字，这些值的平均值将传递给 try 块中的变量，然后执行 else 块。例如，如果我们用 +my_list1+，它会打印 The mean is: 5.0。因为 my_list2 不包含任何值，except 块执行并打印 nan 和 Error: float division by zero。然后执行 finally 块，它总是这样，并打印 The finally block is executed every time。

读文本文件

你的数据，几乎毫无意外的贮存于文件中。这些文件可以是文本文件，CSV 文件，Excel 文件，或其它类型的文件。了解如何访问这些文件并读取数据，可以为你用 python 处理，操作，分析数据提供工具。当你的程序可以每秒处理许多文件的时候，你会看到编程的收获而不是只是一次完成一个任务。你要告诉 python 你的脚本要处理什么文件。你可以硬编码文件名到你的程序中，但是对于许多不同文件的程序来说会变得困难。一种通用的读取文件的方法是在命令行窗口或终端窗口的命令行中，将文件路径放在你的脚本的名字后面。

要使用这种方法，你要在你的脚本顶部导入内置的 sys 模块。要在你的脚本里使用 sys 模块提供的函数，在你的脚本顶部添加 import sys。

```
#!/usr/bin/env python3
from math import exp, log, sqrt
import re
from datetime import date, time, datetime, timedelta
from operator import itemgetter
import sys
```

导入 sys 模块会将 argv 放置在你的处理里。这个变量捕获命令行参数列表——你输入给命令行的所有东西，包括脚本名称——传递给 python 脚本。像其它列表，argv 有索引，argv[0] 是脚本名，argv[1] 是第一个通过命令行传递给脚本的额外的参数。在本例中是 first_script.py 脚本要读取的文件路径。

创建文本文件

为了读取文件，我们首先要创建一个文本文件。可以这么做：

1. 打开 Spyder IDE 或文本编辑器（如 Windows 系统里的 Notepad，Notepad++，

或 Sublime Text ;macOS 里的 TextMate, TextWrangler, 或 Sublime Text on macOS)。

2. 在文本文件里写入以下 6 行 (见图 Figure 1-10):

```
I'm  
already  
much  
better  
at  
Python.
```

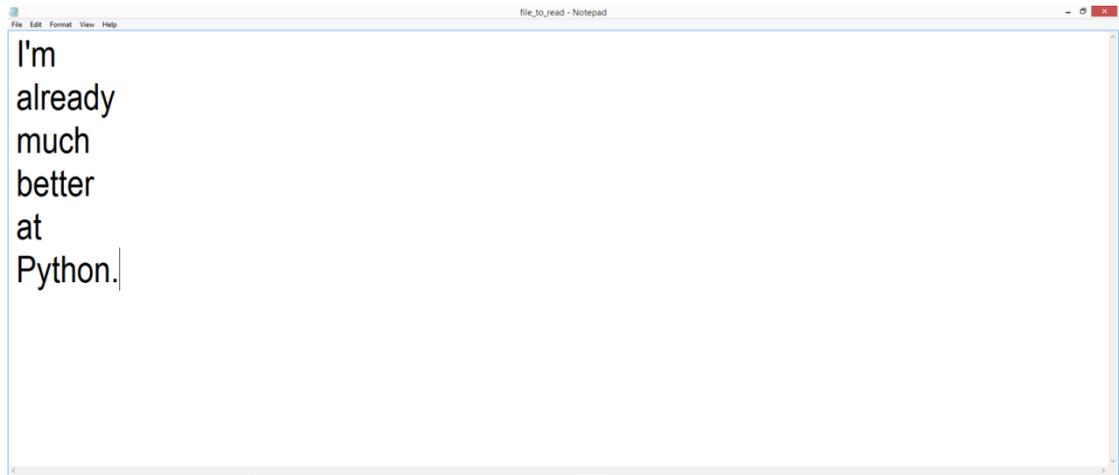


图 1-10. 在 Windows 里的 Notepad++ 中的文本文件 file_to_read.txt

3. 保存文件到桌面, 文件名为 file_to_read.txt.

4. 在 first_script.py 的底部增加下面的代码:

```
# READ A FILE  
# Read a single text file  
input_file = sys.argv[1]  
print "Output #143: "  
filereader = open(input_file, 'r')  
for row in filereader:  
print row.strip()  
filereader.close()
```

本例中的第一行用 `sys.argv` 列表来捕获我们想要读取的文件的文件路径并赋值给变量 `input_file`。第二行创建文件对象, `filereader`, 它包含用 'r' (读) 模式打开的 `input_file` 的多行。下一行的 `for` 循环读取 `filereader` 对象中的行, 每次一行。for 循环的主体打印各行。在打印前, `strip` 函数自读取的行的尾部去除空格, `tab`, `newline` 字符。最后的一行关闭 `file Reader`, 一旦输入文件的所有行被读取并打印到屏幕。

5. 重新存存 first_script.py.

6. 要读取文本文件, 输入下面的行, 如图 1-11, 然后按回车键:

```
python first_script.py file_to_read.txt
```

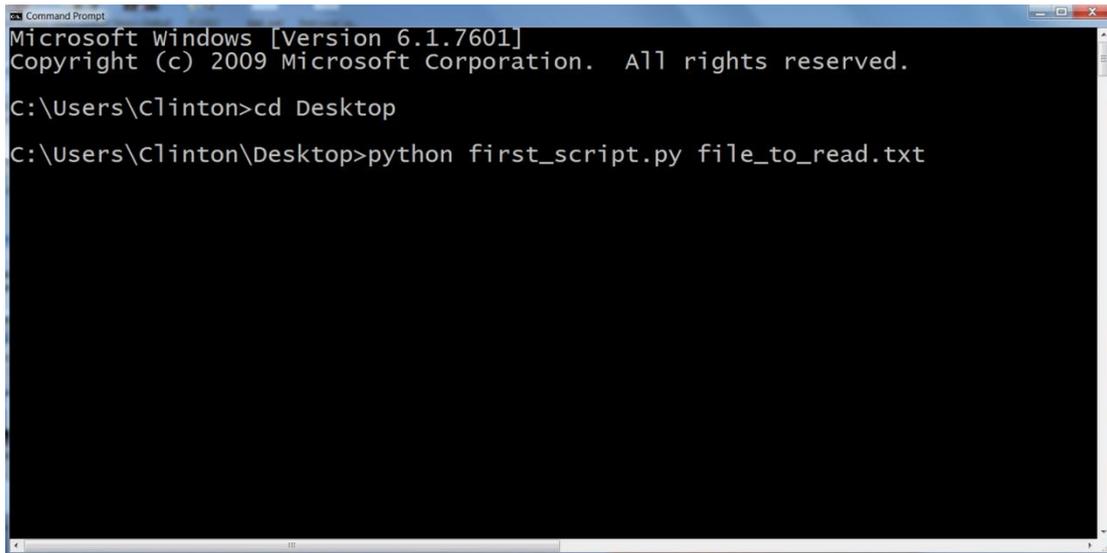


图 1-11. 命令行窗口中 Python 脚本和它要处理的文本文件

到此，你已经用 python 读取文本文件了。你可以看到屏幕下方的如下打印输出(见图 1-12)：

```
I'm
already
much
better
at
Python.
```

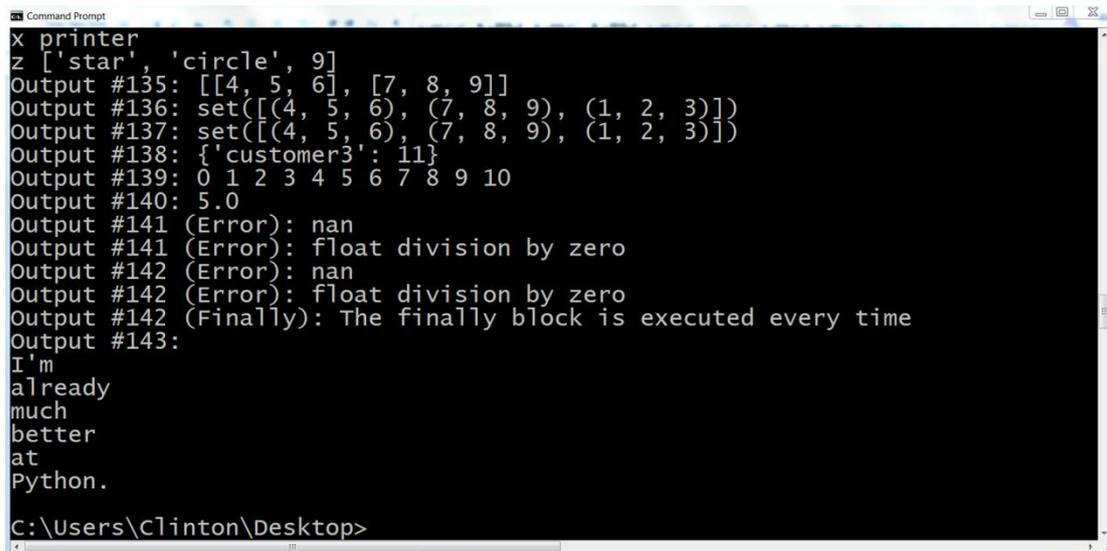


图 1-12. 在命令行窗口中 first_script.py 脚本处理文本文件的输出

脚本和输入文件在同一位置

在命令行中可以简单的输入 python first_script.py file_to_read.txt 因为 first_script.py 和 file_to_read.txt 在相同的位置—即桌面。如果文本

文件与脚本不在同一位置,你要提供文件的完整路径,脚本才知道在哪里找到它。例如,如果文件在 Documents 目录,而不是在桌面,你可以在命令行中用如下路径读取文本文件:

```
python first_script.py "C:\Users\[Your  
Name]\Documents\file_to_read.txt"
```

现代的文件读取符号 Modern File-Reading Syntax

我们用于创建 filereader 对象的代码是创建文件对象的旧的方法。这种方法工作得不错,但是它让文件对象一直打开直到用 close 函数关闭它或脚本终止。虽然这种行为不会有伤害,但是并不简洁并且可能在复杂的脚本中出现错误。从 Python 2.5 起,你可以用 with 语句创建文件对象。这种符号自动关闭文件对象,当 with 语句退出时。

```
input_file = sys.argv[1]  
print("Output #144:")  
with open(input_file, 'r', newline='') as filereader:  
for row in filereader:  
print("{}".format(row.strip()))
```

如你所见,with 语句的版本与前面的版本非常相似,但是它不用调用 close 函数。这个例子说明如何用 sys.argv 来访问和打印一个文本文件的内容。这是一个简单的例子,但是我们可以以它为基础访问其它类型的文件,访问多个文件,写入到输出文件。

下一节用覆盖 glob 模块,它让你读取和处理多个输入文件,只要少量的代码。因为 glob 模块的强大来自于指向目录(如,目录而不是文件),我们去掉或注释掉前面的文件读取代码,使我们可以用 argv[1] 指向一个目录而不是一个文件。注释掉的意思是用 hash 符号放在代码的前面使计算机忽略这行代码,所以你这么做时,first_script.py 看起来是这样子的:

```
## Read a text file (older method) ##  
#input_file = sys.argv[1]  
#print("Output #143:")  
#filereader = open(input_file, 'r', newline='')  
#for row in filereader:  
# print("{}".format(row.strip()))  
#filereader.close()  
## Read a text file (newer method) ##  
#input_file = sys.argv[1]  
#print("Output #144:")  
#with open(input_file, 'r', newline='') as filereader:  
# for row in filereader:  
# print("{}".format(row.strip()))
```

通过这些改变,我们可以增加下一节讨论的 glob 代码来处理多个文件了。

用 glob 读取多个文件

在许多的商务应用中,相同的或相似的行为会在多个文件中发生。例如,你可能需要从多个文件中选择数据子集。自多个文件中计算如总数和均值之类有统

计量，或自多个文件中计算数据子集的统计量。随着文件数量的增加，手工处理的难度增加，出错的机率增加。一种读取多个文件的方法是在命令行中 python 脚本的后面添加目录路径。要用这种方法，你需要在你的脚本顶部导入内置的 os 和 glob 模块。在脚本顶部添加 import os 和 import glob:

```
#!/usr/bin/env python3
from math import exp, log, sqrt
import re
from datetime import date, time, datetime, timedelta
from operator import itemgetter
import sys
import glob
import os
```

当你导入 os 模块后，你有多个有用的 pathname 函数可用。例如 os.path.join 智能的将一个或多个路径成分联合在一起。glob 模块会查找与指定模式匹配的所有的 pathnames。通过使用 os 和 glob，你可以找到指定目录中的与特定模式匹配的所有的文件。为了读取多个文本文件，我们需要创建另外的文本文件。

创建另外的文本文件

1. 打开 Spyder IDE 或文本编辑器（如 Windows 系统的 Notepad, Notepad++, 或 Sublime Text; macOS 的 TextMate, TextWrangler, 或 Sublime Text on）。
2. 在文本文件中写入以下 8 行（图 1-13）:

```
This
text
comes
from
a
different
text
File
```

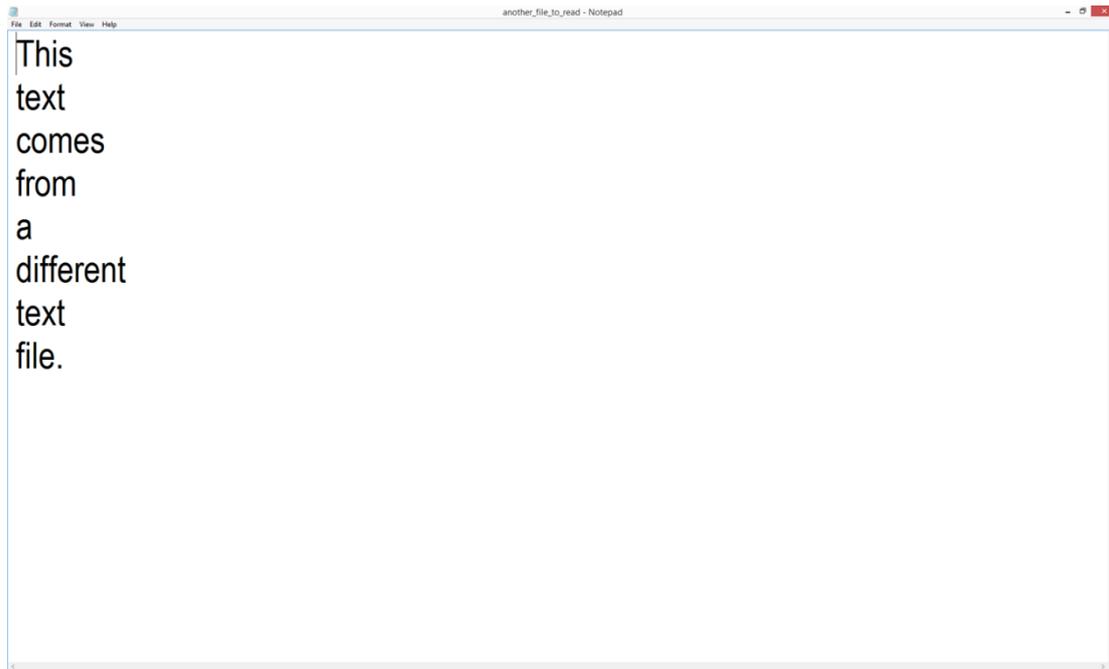


图 1-13. Notepad++中的文本文件, another_file_to_read.txt

3. 在你的桌面中保存 another_file_to_read.txt.

4. 在 first_script.py 脚本底部增加下面的代码:

```
# Read multiple text files
print("Output #145:")
inputPath = sys.argv[1]
for input_file in glob.glob(os.path.join(inputPath, '*.txt')):
with open(input_file, 'r', newline='') as filereader:
for row in filereader:
print("{}".format(row.strip()))
```

本例的第一行与读取一个文本文件的例子相似,例外的是本例我们提供目录路径而不是文件路径。这里我们的目录包含两个文本文件。

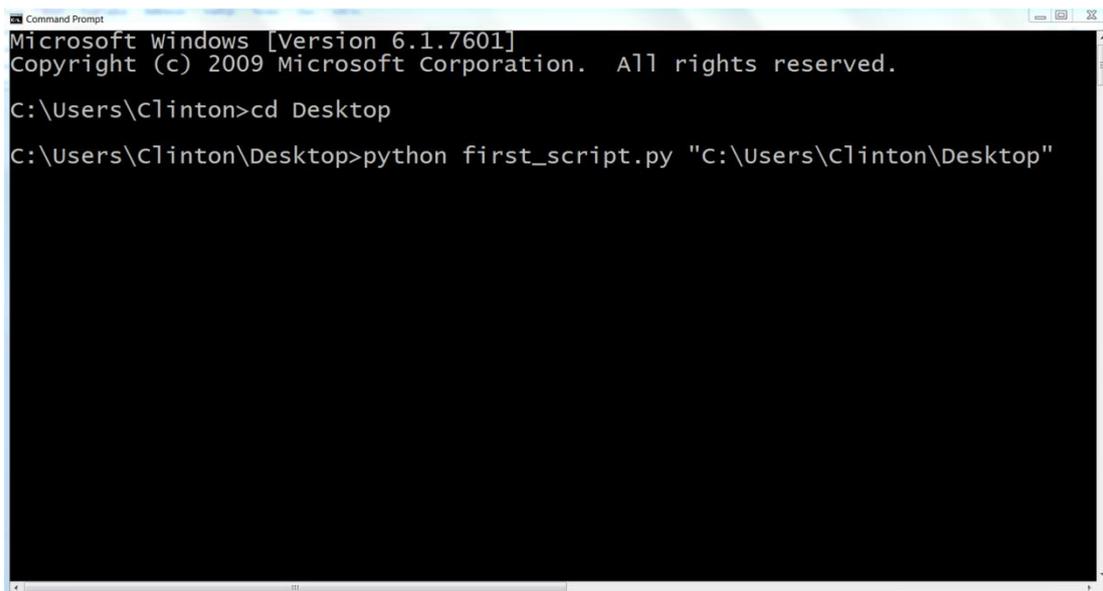
第二行是 for 循环,用 os.path.join 函数和 glob.glob 函数来查找指定目录中特定模式的所有文件。特定的目录路径包含于 inputPath 变量,我们将提供给命令行。os.path.join 函数联合所有的符合特定模式的由 glob.glob 扩展的文件名及目录路径。本例中,我们用*.txt 模式匹配所有以 .txt 结尾的文件名。因为这是一个 for 循环,本行的余下符号看起来相似。input_file 是由 glob.glob 函数创建的每个文件名列表的占位符。

这一行,基本的来说是,“对于匹配文件列表中的每个文件,做如下....”。余下的代码与读取一个文件的代码相似。以读模式打开 input_file 变量,并创建 filereader 对象。对于 filereader 对象的每一行,自行的尾部去除空格, tabs, 和 newline 字符。

5. 重新保存 first_script.py.

6. 本读取文本文件,输入下面的一行,如图 1-14,然后按回车键:

```
python first_script.py "C:\Users\[Your Name]\Desktop"
```

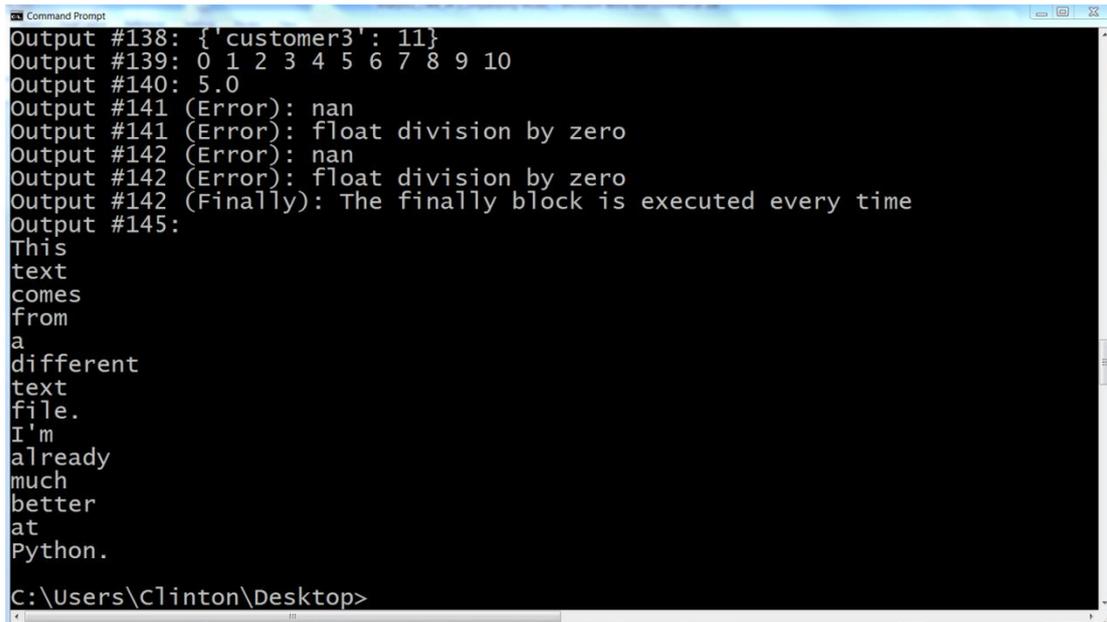


```
Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Clinton>cd Desktop
C:\Users\Clinton\Desktop>python first_script.py "C:\Users\Clinton\Desktop"
```

图 1-14. 命令行窗口中包含 inputfiles 的桌面目录和 Python 脚本
到这里，你已经可以用 python 读取多个文本文件了。你可以看到屏幕下面如下的打印输出(图 1-15):

```
This
text
comes
from
a
different
text
file.
I'm
Already
much
better
at
Python.
```



```
Command Prompt
Output #138: {'customer3': 11}
Output #139: 0 1 2 3 4 5 6 7 8 9 10
Output #140: 5.0
Output #141 (Error): nan
Output #141 (Error): float division by zero
Output #142 (Error): nan
Output #142 (Error): float division by zero
Output #142 (Finally): The finally block is executed every time
Output #145:
This
text
comes
from
a
different
text
file.
I'm
already
much
better
at
Python.
C:\Users\Clinton\Desktop>
```

图 1-15. 在命令行窗口中 first_script.py 处理多个文本文件的输出

学习这一技术的重要方面是它的放大性。本例中只有两个文件，但是它可以处理成百上千个或更多的文件。通过学习使用 glob.glob 函数，你可以处理许多的文件而只用手工处理所需的一小部分时间。

写文本文件

到此，许多的函数都包括 print 语句，将输出发送到命令行或终端窗口。你在调试程序和检查输出结果时使用打印很有用。但是，许多情况下，只要你知道输出是正确的，你就想将结果输出到文件中以进一步的分析，报告或贮存。Python 提供了两种简单的方法来写文本文件或逗号分隔的文件。write 方法将各个字符串写入文件，writelines 方法将字符串队列写入文件。下面的例子使用 range 和 len 函数的组合来追踪列表值的索引使逗号放在值和 newline 字符之间，最后的值之后放 newline 符事情。

添加代码到 first_script.py

1. 添加下面的代码到 first_script.py 底部：

```
# WRITE TO A FILE
# Write to a text file
my_letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
max_index = len(my_letters)
output_file = sys.argv[1]
filewriter = open(output_file, 'w')
for index_value in range(len(my_letters)):
if index_value < (max_index-1):
filewriter.write(my_letters[index_value]+'\\t')
else:
```

```
filewriter.write(my_letters[index_value]+'\\n')  
filewriter.close()  
print "Output #146: Output written to file"
```

本例中，变量 `my_letters` 是字符串列表。我们想要打印这些字母到文本文件中，每个用 `tab` 分开。本例中的一个复杂是保证字母之间 `tabs` 以及最后之母之后的 `newline` 符号。想知道我们什么时候到达最后一个字母，我们要追踪列表中的字母的索引。`len` 函数计算列表中值的数目，所以 `max_index` 等于 10。再次，我们在命令行窗口或终端中用 `sys.argv[1]` 提供输出文件的名称和路径。我们创建文件对象 `filewriter`，但并不是打开它来读取文件而是打开文件来写，用 `'w'`（写）模式。我们用 `for` 循环来遍历列表 `my_letters` 中的值，我们用 `range` 函数组合 `len` 函数来追踪列表中值的索引。`if-else` 逻辑所我们区别列表中的最后一个字母和列表中前面所有的其它字母。`if-else` 逻辑是这样工作的：`my_letters` 包含 10 个值，但索引从 0 开始，所以字母的索引值为 0, 1, 2, 3, 4, 5, 6, 7, 8, 9。因此 `my_letters[0]` 是 `a`，`my_letters[9]` 是 `j`。`if` 块评估索引值 `x` 是否小于 9。`max_index - 1` 或 `10 - 1 = 9`。条件为真直到列表中的最后一个字母。因此，`if` 块说，“直到列表中的最后一个字母，写入字母和 `tab` 到输出文件中。”当我们查找列表中最后一个字母时索引值为 9，它不小于 9，所以 `if` 块评估为假并执行 `else` 块。`else` 块中的 `write` 语句说，“写最后一个字母加上 `newline` 符到输出文件中。”

2. 注释掉早期的代码来读取多个文件。

为了看这些文件的动作，我们需要写到一个文件并观察输出。因为我们再一次使用 `argv[1]` 来指明输出文件的路径和文件名，我们删除或注释掉前面的 `glob` 代码使我们可以用 `argv[1]` 来指明输出文件。如果你选择注释掉前面的 `glob` 代码，`first_script.py` 看起来这样子：

```
## Read multiple text files  
#print("Output #145:")  
#inputPath = sys.argv[1]  
#for input_file in glob.glob(os.path.join(inputPath, '*.txt')):  
# with open(input_file, 'r', newline='') as #filereader:  
# for row in filereader:  
# print("{}".format(row.strip()))
```

3. 重新保存 `first_script.py`。

4. 要写入文本文件，输入下面的代码，如图 1-16，然后按回车键：

```
python first_script.py "C:\Users\[Your  
Name]\Desktop\write_to_file.txt"
```

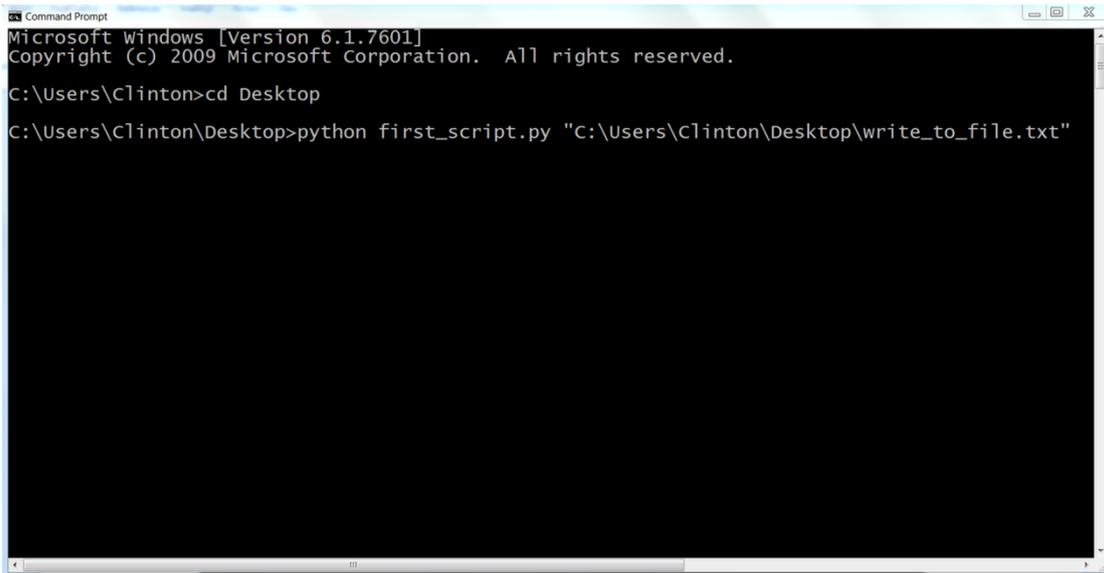


图 1-16. 在命令行窗口中，Python 脚本和它要写的输出文件的路径和名称

5. 打开输出文件 `write_to_file.txt`.

你现在可以用 Python 来写入输出到文件了。完成了这些步骤，你看不到屏幕上的任何输出，但是你最小化所有打开的窗口，看一下桌面，有一个新的文本文件称为 `write_to_file.txt`，它包含来自 `my_letters` 列表的所有字母，由 tabs 分开，最后是一个 `newline` 符号。如图 1-17。

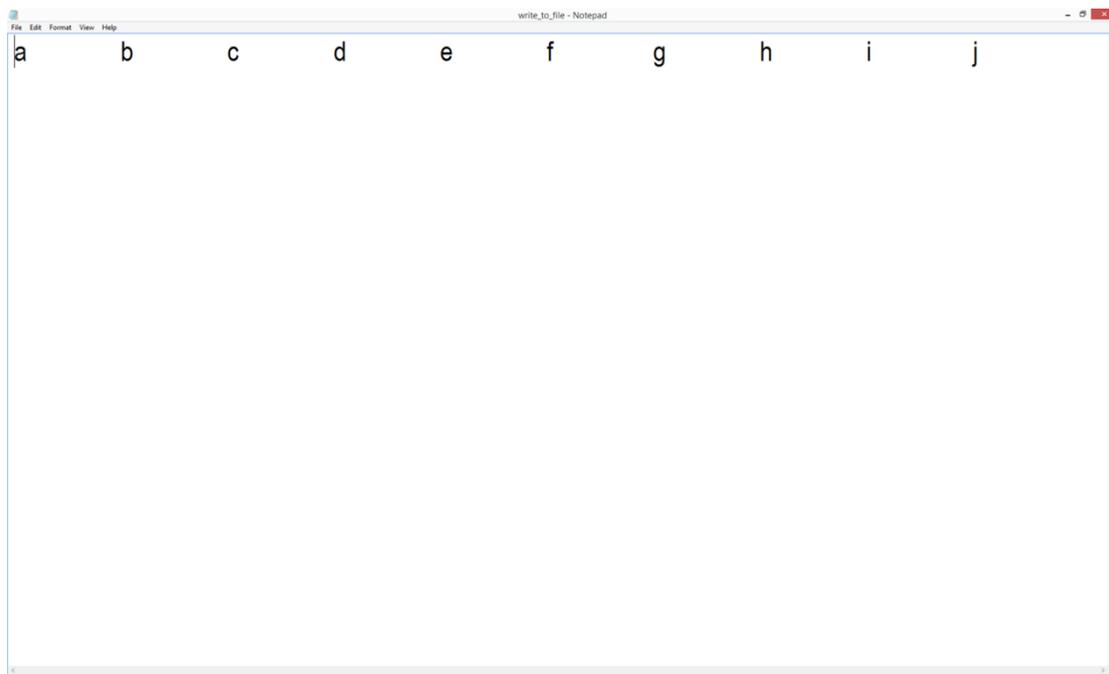


图 1-17. 桌面上 `first_script.py` 创建的输出文件 `write_to_file.txt`

下一个例子与这个例子相似，例外的是它展示用 `str` 函数来转换值到字符，以致它们可以用 `write` 函数写入文件。它也展示了 'a' (append) 模式以在已有的输出文件后面增加输出。

写入 Comma-Separated Values (CSV) File

1. 在 first_script.py 后面增加以下代码:

```
# Write to a CSV file
my_numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
max_index = len(my_numbers)
output_file = sys.argv[1]
filewriter = open(output_file, 'a')
for index_value in range(len(my_numbers)):
    if index_value < (max_index-1):
        filewriter.write(str(my_numbers[index_value])+',')
    else:
        filewriter.write(str(my_numbers[index_value])+'\n')
filewriter.close()
print "Output #147: Output appended to file"
```

本例与前例相似,但是它展示如何添加到已有的输出文件,如何转换列表中非字符串到列表,以致以它们可以用 write 函数写入文件。本例中,列表包含整数。write 函数写字串,所以你要用 str 函数转换非字符串到字串,然后才能用 write 函数写入到文件中。第一次遍历用 for 循环, str 函数写 0 到输出文件,接着写逗号。用这种方法继续写入列表中的数字到输出文件直到列表中的最后一个数字,此时, else 块执行,最后一个数字后接 newline 符而不是逗号被写到输出文件。

注意我们打开的文件对象 filewriter,是用 append 模式 ('a')而不是写模式 ('w')。如果我们在命令行中提供相同的输出文件名,那么代码的输出将被添加到之前写的 write_to_file.txt 文件里。可选地,如果我们用写模式打开 filewriter,则前面的输出将被删除,只有本代码的输出被添加到 write_to_file.txt。你会发现 append 模式打开文件对象强大,当你要处理多个文件,并添加所有的数据到一个输出文件时。

2. 重新保存 first_script.py.

3. 要添中到文本文件,输入以入代码然后按回车键:

```
python first_script.py "C:\Users\[Your
Name]\Desktop\write_to_file.txt"
```

4. 打开输出文件, write_to_file.txt.

你现在已经用 python 写入和添加输出到文本文件了。完成了这些步骤,你看不到任何输出打印到屏幕,但是当你打开 write_to_file.txt 文件时,你会发现有新的第二行包含 my_numbers 中的数字,由逗号分隔,最后是个 newline 符号。如图 1-18 所示。

最后,本例展示了一种有效的方法来写入 CSV 文件。事实上,如果我们不从前面的基于 tab 的文件(由 tab 符号而不是逗号分隔)写输出到输出文件,我们将文件命名为 write_to_file.csv 而不是 write_to_file.txt,则我们创建了 CSV 文件。

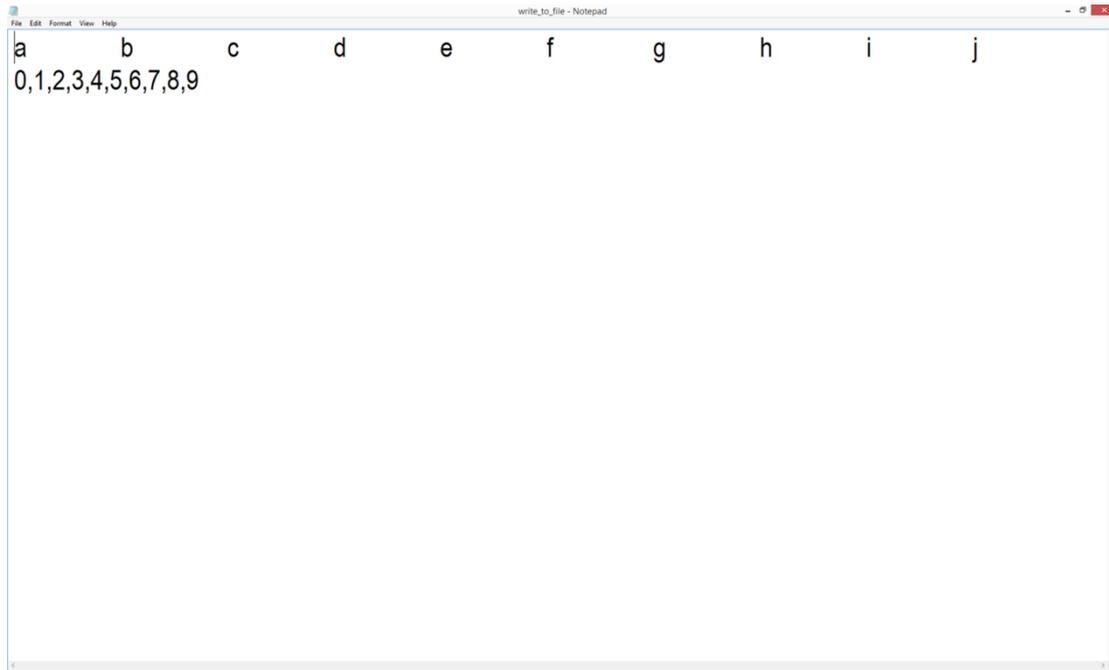


图 1-18. first_script.py 添加的输出文件，write_to_file.txt，在桌面上

打印语句 print Statements

打印语句是调试任何程序的重要帮助。如你所见，本章中的许多例子包含打印语句作为输出。然而，你也可以在你的代码中增加打印语句以临时的观察中间输出。如果你的代码根本不工作或者产生的结果不是你想要的，那么在有意义的地方增加打印语句从程序的顶部观察初始的计算是不是你想要的。如果是，继续向下检查代码看它们是否也如你所愿的工作。从你的脚本顶部开始，你可以确保你识别出现错误结果的第一个位置并在那修改代码，然后测试余下的代码。本节的信息是，“不要怕使用打印语句来调试你的代码并确保这正确的工作。”你可以注释掉或去掉打印语句当你认为你的代码正确的工作时。

本节我们已经讨论了很多东西。我们讨论了如何导入模块，基础的数据类型，函数和方法，模式匹配，打印语句，日期，控制流，函数，意外，读单一文件和多个文件，以及写文本文件和逗号分隔文件。如果你按照本章的例子，你已经写了 500 行 python 代码了。最好的部分是它们是进行更加复杂的文件处理和数据操作工作的基本构件。通过本章的例子，你已经可以准备理解和掌握后面章节的技术了。

补充：

Array [] 包括 vectors 和 matrices，用于数值型数据的操作。在 numpy 中定义。注意 vectors 和 1-d arrays 是不同的：vectors 不可以被转置！对于 arrays，“+” 将相应的元素相加；而且 array 的 .dot 方法执行两个 arrays 的标量加。（对于 Python 3.5 以上，这也可以用 “@” 操作符。）。

```
In [10]: myArray2 = np.array(myList2)
```

```
In [11]: myArray3 = np.array(myList3)
```

```
In [12]: myArray2 + myArray3
```

```
Out[12]: array([5, 7, 9])
```

```
In [13]: myArray2.dot(myArray3)
Out[13]: 32
```

DataFrame 优化的数据结构，用于命名的，统计数据。在 pandas 中定义。

从其它格式文件输入数据

Matlab scipy 包的 `scipy.io.loadmat` 命令支持从 Matlab 文件读取数据。

Clipboard 如果你的数据在剪贴板中，你可以用 `pd.read_clipboard()` 直接导入数据。

Other file formats pandas 也支持 SQL 数据库和其它格式的文件。访问它们最简单的方法是输入 `pd.read_ + TAB`，它会显示所有支持的读取数据到 pandas DataFrames 的方法。

Matlab

下面的命令从 Matlab 文件“data.mat”返回字符串，数字，向量，矩阵变量。Matlab 的变量包括标量，字符串，向量，矩阵，结构，分别称为数字，文本，向量，矩阵和结构。

```
from scipy.io import loadmat
data = loadmat('data.mat')
number = data['number'][0,0]
text = data['text'][0]
vector = data['vector'][0]
matrix = data['matrix']
struct_values = data['structure'][0,0][0][0]
strunct_string = data['structure'][0,0][1][0]
```