PYTHON TRICKS: THE BOOK

A Buffet of Awesome Python Feature

--Dan Bader

# 第1章

## 介绍

### 1.1 Python Trick是什么?

**Python Trick: 用一段**简短的Python代码片来作为演示工具。Python技巧要么用简单的 插图展示 Python的一个方面，要么作为一个激励性的示例，让您对Python能够有更深 入的挖掘并获得直观 的理解。

Python Trick 始于我在Twitter上分享了一周一系列的代码片截图。令我惊讶的是，这些截图得到了热烈的反响，并被分享和转发了很多天。

越来越多的开发者开始询问我如何才能“获得整个系列”。事实上，我那时只写了那么几个的技巧，而且分别是Python方面的不同话题，并没有一个总体规划。那些仅仅是我发在Twitter上的一些有趣实践。

但从这些开发者的询问中我得到一个启示：我的这些简短而有趣的代码示例是值得作为一套教学工具来摸索的。所以，我开始写更多的Python Tricks，并在邮件列表中分享它们。几天之内，上百的Python开发者加入到邮件列表，这个反映也令我也很震惊。在接下来的一段时间内，越来越多的Python开发者联系到我。他们感谢在这些技巧中，有部分编程语言的知识让他们困惑已久，看了之后豁然开朗。听到这个反馈感觉真棒。我认为这些Python Tricks只是代码截图，但是很多开发者从中获得了更多的价值。

那时候，我决定把我的Python Tricks实践内容翻倍，并将其扩展为大约30封电子邮件。其中每一个仍然只是一个标题和一个代码截图，我很快意识到了这种格式的限制。大约在这个时候，一位盲人Python开发者发邮件给我，失望地发现这些Python Tricks是通过他无法用屏幕阅读器阅读的图像分享的。

显然，我需要投入更多的时间在这个项目上，使它更具吸引力，更容易被更多的观众接受。所以，我开始用纯文本和适当的基于HTML的语法高亮来重新构建整个系列的Python Tricks邮件内容。 新版的Python Tricks在这一段时间里很好地被展现了出来。根据我得到的反馈，开发者似乎很高兴他们终于可以复制和粘贴示例代码来使用了。

随着越来越多的开发者注册了邮件列表，我从这些收到的回复和问题中开始发现的一个问题。 一些简单的技巧本身就是一个很好的示例， 但对于有些相对复杂的技巧，没有人来指导读者或给给他们更多的资源来帮助他们形成一个更深入的理解。

不得不说，这是另一个有很大改进空间的地方。我在[dbader.org](https://dbader.org/)上的使命是：“帮助Python开发者变得更加出色”——这显然是一个接近这个目标的机会。

我决定从电子邮件课程中选择最好，最有价值的Python技巧，然后开始基于他们来编写一本新的Python书籍：

* 一本以简短而易于理解的例子来教授这门语言最酷的方面的书。
* 一本就像汇聚了各种完美python特性的自助餐一样的书，并保持高水平性能
* 一本指导你并帮助你加深对Python理解的书。

这本书对于我来说真的是对于Python狂热爱好的表现，也是一次巨大的尝试。我希望您会喜欢阅读，并在这个过程中学习到关于Python的一些知识!

### 1.2 本书将为您带来什么？

我的目的是通过这本书可以让您变为一个更优秀（更高效，更全面，更实用）的Python开发者。你可能想知道，读这本书如何能帮助我做到这些呢？

Python Tricks不是一本按部就班的Python教程。它也不是一本入门阶段的课程。如果您正处于学习Python的初级阶段，本书不会带您成为为专业的Python开发者。但阅读它对你仍旧是有好处的，同时您需要确保您也会通过其他一些资源来夯实您在Python方面的基础技能。

如果您已经掌握了一定的 Python知识，并希望提升自己的技能到更高水平， 那么您将从本书中获得极大的收获。 如果你已经编写了一段时间的Python，并准备做进一步深入的了解，使你的代码变得更加Pythonic，那么，它将会对你非常有用。

如果您已经有了其他编程语言的使用经验，并且您希望快速的掌握Python，那么阅读PythonTricks对您也同样有帮助。 在本书中您将看到大量的实用技巧和设计模式，使您成为一个更高效，更熟练的Python程序员。

### 1.3 如何来阅读本书？

阅读Python Tricks的最佳方式：因为本书可以看作是一系列Python优秀特性展示的集合，书中的每一个 Python Trick 都是独立的，所以您完全可以直接跳到您感兴趣的那部分去阅读。事实上，我也鼓励您用这样的方式去阅读。

当然，您也可以按照书中的顺序阅读所有的PythonTricks。 这样您就不会错过任何一个，当您阅读到最后一页的时候，您就会看到其中所有的示例。

这些技巧中，有一部分会很容易理解，只要阅读完对应的章节，就可以毫无困难地把它们融入日常工作中。 但也有一部分技巧可能需要更多的时间来理解。

如果您在自己的程序中使用某个技巧遇到困难时，可以通过在Python交互式解释器中运行这段示例代码来获得帮助。

如果这样做依旧不能帮您解决疑惑，请随时联系我，以便让我可以为您解答问题，同时可以改进本书中的解释。 从长远来看，这不仅有利于你，也有利于阅读本书的其他读者。

# 第2章

## 样章

### 2.1 对象比较**：“**is” vs “==”

小时候，我的邻居有两只双胞胎猫。 他们看起来似乎完全一样 --- 一样木炭色的毛和一样犀利的绿眼睛。如果抛开一些个性上的差异，只是看它们样子，你根本不能把它们区分开来。 当然，他们本质上是两只不同的猫，两个不同的生物，尽管它们看起来完全一样。

这使我意识到了相等和相同之间存在涵义上的差异。 而这种差异对于理解Python的 “is” 和 “==” 比较运算符是至关重要的。

“==” 运算符是用来比较两个对象是否相等：如果把这两只猫看作是Python对象，并且我们通过 “==” 运算符来将它们进行比较，那么我们得到的答案是： “两个猫相等”。

然而，“is” 运算符是用来比较两个对象的本质：如果我们将我们的猫通过 “is” 运算符来进行比较，我们会得到的答案是 “这是两只不同的猫”。

但是，在我用这两个猫作类比还没有完全搞乱思维之前，让我们来看看一些真正的Python代码。

首先，我们将创建一个新的列表对象并将其命名为a，然后定义另一个指向同一列表对象的变量 b ：

>>> a = [1, 2, 3]

>>> b = a

我们来看看这两个变量。 我们可以看到他们指向相同的列表：

>>> a

[1, 2, 3]

>>> b

[1, 2, 3]

因为两个列表对象看起来是一样的，所以当我们使用 “==” 运算符比较它们的相等性时，我们会得到预期的结果：

>>> a == b

True

但是，这并不能告诉我们a和b是否实际指向同一个对象。 当然，我们知道他们是，因为是我们刚刚分配的，但是假设我们不知道，又该如何知道它们是同一个对象呢？

答案是将两个变量用 “is” 运算符进行比较。 这样可以确认两个变量实际上是否指向同一个列表对象：

>>> a is b

True

让我们看看当我们创建一个列表对象的相同副本时会发生什么。 我们可以通过调用 list() 方法作用在现有列表上来创建一个副本，我们将它命名为c：

>>> c = list(a)

你会再一次看到我们刚刚创建的新列表对象 c 与a和b指向的列表对象相同：

>>> c

[1, 2, 3]

现在就是它变得有趣的地方。 我们使用 “==” 运算符来比较我们的列表副本 c 和初始列表 a 。 你期望看到什么答案呢？

>>> a == c

True

好吧，我希望这和你预想的一样。 这个结果告诉我们的是 c 和 a 有相同的内容。 Python认为它们是相等的。 但他们是否指向同一个对象呢？ 让我们来看看 “is” 运算符：

>>> a is c

False

咦！ 这时我们得到了不一样的答案。 Python告诉我们 c 和 a 指向的是两个不同的对象，即使它们的内容可能是相同的。

那么，我们来回顾一下，试着将 “is”和 “==” 之间的区别分成两个简短的定义：

* 如果两个变量指向相同（同一个）的对象，则 “is” 表达式的计算结果为True。
* 如果变量引用的对象是相等的（具有相同的内容），则 “==” 表达式计算结果为True。

当你需要决定在Python中是使用 “is” 还是 “==”的时候，记得想想那一对双胞胎猫（当然狗也可以）。 如果你这样做，你就会有答案了。

### 2.2 傲娇的逗号

在Python中如果你想要从一个列表，字典或者集合中添加或删除一个元素，这里先给一个提示：记得使用逗号结束每一行。

不知道我在说什么？ 让我举个简单的例子。 假设你的代码里有这个名字列表：

>>> names = ['Alice', 'Bob', 'Dilbert']

当你对这个名字列表进行修改后，通过Git diff查看，你很难知道这个列表究竟被修改了什么内容。 因为大多数源代码管理系统是基于行审查的，并且难以突出显示单行中的多处更改。

一个快速的解决方案就是改变代码风格，通过跨越多行编写代码来分离列表，字典或集合中的每个元素，如下所示：

>>> names = [

... 'Alice',

... 'Bob',

… 'Dilbert'

... ]

这样每行就只有一个元素，当你在源代码管理系统中查看一个diff的时候，很清楚可以知道哪一个元素被添加，删除或修改。 这虽然是一个小小的改变，但我发现它可以帮助我避免一些低级错误。 这也让我的同事在审查我的代码时更容易的知道代码是否有变更。

即便如此，有两种情况在编写代码时仍然会对我造成困扰。 无论何时在列表末尾添加新元素，或者删除最后一个元素，都必须手动更新逗号的位置来满足格式的一致性。

假设您想为此列表添加一个新的名字（Jane）。 如果添加 Jane ，则需要在Dilbert行之后添加一个逗号以避免出现下面这样令人讨厌的错误：

>>> names = [

... 'Alice',

... 'Bob',

... 'Dilbert' *# <-- 缺少逗号!*

... 'Jane'

]

当你检查这个列表的内容时，你会神奇地发现：

>>> names

['Alice', 'Bob', 'DilbertJane']

如你所见，Python将字符串Dilbert和Jane合并到DilbertJane中。 这种所谓的“字符串连接”是Python故意的且可存储的写法。 这是一种极易搬起石头砸自己脚的方式，可能会在程序中引入难以调试寻找到的Bug：

“多个相邻的字符串或字节文字（用空格分隔），即便是可能使用了不同的引用符号，这也是允许的， 其含义等同于将他们连接起来。”[[1]](#footnote-2)

诚然，字符串连接在某些情况下仍然是一个很有用的特性。 例如，您可以使用它来减少在多行中拆分长字符串所需的反斜杠数量：

my\_str = ('This is a super long string constant '

'spread out across multiple lines. '

'And look, no backslash characters needed!')

但另一方面，我们刚刚也看到，相同的特性也可以变成是一个缺陷。 现在，我们该如何解决这个问题呢？

很简单，在Dilbert之后添加缺少的逗号可防止两个字符串合并为一个：

>>> names = [

... 'Alice',

... 'Bob',

... 'Dilbert',

... 'Jane'

]

但是现在我们现在饶了一圈后发现，又回到开始的问题。 为了添加这一个新名字到列表中，我不得不修改两行代码来实现。 这同样使得我们难以在Git diff中看出修改了什么...是有人添加了一个新的名字？ 但是有人更新了 Dilbert’s 的名字没有呢？

幸运的是，Python的语法允许有一些冗余来彻底解决这个逗号的问题。 要想避免这个问题，你首先需要让自己适应一种新的编码风格。让我告诉你怎么做。

在Python中，你可以在列表，字典或集合中的每一个元素后面都跟上一个逗号，包括最后一个元素。这样的话，你只需要记住永远用逗号来结束每一行，从而避免考虑是否需要逗号所带来的问题。

以下是最终示例的样子：

>>> names = [

... 'Alice',

... 'Bob',

... 'Dilbert',

... ]

你有发现在Dilbert之后的逗号吗？ 这样可以轻松添加或删除新元素，而无需修改逗号的位置。 这样可以保证你的每一行都是一致的，你的源代码在差异审查时也会更加清晰，审查你代码的人也会很高兴。 怎么样，有时候魔法就藏在小东西里，不是吗？

### 要点总结：

* 巧妙的格式和逗号放置可以使你的列表，字典或集合更容易维护。
* Python的字符串连接功能可以为你带来一些方便之处，同时也可能会引入难以调试查找到的错误。

---------------------------------

## 这是“PythonTricks：The Book”的一个样章

本书的完整版本将包括更多的PythonTricks，通过有趣和易于理解的示例和解释来教会你Python的深层次内容和编程技巧。

如果您喜欢本示例章节，您可以在dbader.org/pytricks-book上购买本书的完整版本。

---------------------------------

1. [cf. Python Docs: “String literal concatenation”](https://docs.python.org/3.6/reference/lexical_analysis.html" \l "string-literal-concatenation) [↑](#footnote-ref-2)