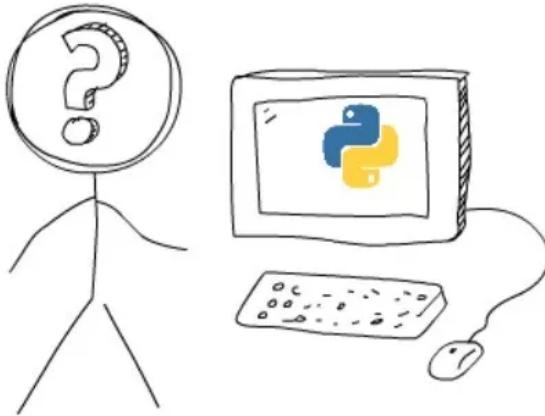


超8千Star，火遍Github的Python反直觉案例集！

大数据文摘 今天



大数据文摘授权转载

作者：Satwik Kansal

译者：暮晨

Python，是一个设计优美的解释型高级语言，它提供了很多能让程序员感到舒适的功能特性。

但有的时候，Python的一些输出结果对于初学者来说似乎并不是那么一目了然。

这个有趣的项目意在收集Python中那些难以理解和反人类直觉的例子以及鲜为人知的功能特性，并尝试讨论这些现象背后真正的原理！

虽然下面的有些例子并不一定会让你觉得WTFs，但它们依然有可能会告诉你一些你所不知道的Python有趣特性。我觉得这是一种学习编程语言内部原理的好办法，而且我相信你也会从中获得乐趣！

如果你是一位经验比较丰富的Python程序员，你可以尝试挑战看是否能一次就找到例子的正确答案。你可能对其中的一些例子已经比较熟悉了，那这也许能唤起你当年踩这些坑时的甜

蜜回忆

Table of Contents/目录

- Table of Contents/目录
- Structure of the Examples/示例结构
- Usage/用法
- Examples/示例
 - Section: Strain your brain!/大脑运动!
 - > Strings can be tricky sometimes!/微妙的字符串 *
 - > Time for some hash brownies!/是时候来点蛋糕了!
 - > Return return everywhere!/到处返回!
 - > Deep down, we're all the same./本质上我们都一样. *
 - > For what?/为什么?
 - > Evaluation time discrepancy/评估时间差异
 - > is is not what it is!/出人意料的 is !
 - > A tic-tac-toe where X wins in the first attempt!/一蹴即至!
 - > The sticky output function/麻烦的输出
 - > is not ... is not is (not ...) / is not ... 不是 is (not ...)
 - > The surprising comma/意外的逗号
 - > Backslashes at the end of string/字符串末尾的反斜杠
 - > not knot!/别纠结!
 - > Half triple-quoted strings/三个引号
 - > Midnight time doesn't exist?/不存在的午夜?
 - > What's wrong with booleans?/布尔你咋了?
 - > Class attributes and instance attributes/类属性和实例属性
 - > yielding None/生成 None
 - > Mutating the immutable!/强人所难
 - > The disappearing variable from outer scope/消失的外部变量
 - > When True is actually False/真亦假
 - > From filled to None in one instruction.../从有到无...
 - > Subclass relationships/子类关系 *
 - > The mysterious key type conversion/神秘的键型转换 *
 - > Let's see if you can guess this?/看看你能否猜到这一点?
 - Section: Appearances are deceptive!/外表是靠不住的!
 - > Skipping lines?/跳过一行?
 - > Teleportation/空间移动 *
 - > Well, something is fishy.../嗯, 有些可疑...
 - Section: Watch out for the landmines!/小心地雷!
 - > Modifying a dictionary while iterating over it/迭代字典时的修改
 - > Stubborn del operator/坚强的 del *
 - > Deleting a list item while iterating/迭代列表时删除元素
 - > Loop variables leaking out!/循环变量泄漏!
 - > Beware of default mutable arguments!/当心默认的可变参数!
 - > Catching the Exceptions/捕获异常
 - > Same operands, different story!/同人不同命!
 - > The out of scope variable/外部作用域变量
 - > Be careful with chained operations/小心链式操作
 - > Name resolution ignoring class scope/忽略类作用域的名称解析
 - > Needle in a Haystack/大海捞针
 - Section: The Hidden treasures!/隐藏的宝藏!

- > Okay Python, Can you make me fly?/Python, 可以带我飞? *
- > goto, but why?/ goto, 但为什么? *
- > Brace yourself!/做好思想准备 *
- > Let's meet Friendly Language Uncle For Life/让生活更友好 *
- > Even Python understands that love is complicated/连Python也知道爱是复杂的 *
- > Yes, it exists! /是的, 它存在!
- > Ininity/无限 *
- > Mangling time! /修饰时间! *
- Section: Miscellaneous/杂项
 - > += is faster/更快的 +=
 - > Let's make a giant string! /来做个巨大的字符串吧!
 - > Explicit typecast of strings/字符串的显式类型转换
 - > Minor Ones/小知识点
- Contributing/贡献
- Acknowledgements/致谢
- License/许可
 - Help/帮助
 - Want to surprise your geeky pythonist friends? /想给你的极客朋友一个惊喜?
 - Need a pdf version? /需要来一份pdf版的?
 - Follow Commit/追踪Commit

示例结构

所有示例的结构都如下所示:

> 一个精选的标题 *

标题末尾的星号表示该示例在第一版中不存在，是最近添加的。

准备代码.

释放魔法...

Output (Python version):

>>> 触发语句

出乎意料的输出结果

(可选): 对意外输出结果的简短描述。

说明:

简要说明发生了什么以及为什么会发生。

如有必要，举例说明

Output:

```
>>>触发声句#一些让魔法变得容易理解的例子  
#一些正常的输入
```

注意：所有的示例都在Python3.5.2版本的交互解释器上测试过，如果不特别说明应该适用于所有Python版本。

用法

我个人建议，最好依次阅读下面的示例，并对每个示例：

仔细阅读设置例子最开始的代码。如果您是一位经验丰富的 Python 程序员，那么大多数时候您都能成功预期到后面的结果。

阅读输出结果

- 确认结果是否如你所料.
- 确认你是否知道这背后的原理

PS: 你也可以在命令行阅读 WTFpython. 我们有 pypi 包 和 npm 包(支持代码高亮).(译：这两个都是英文版的)

安装 npm 包 wtfpython

```
$ npm install -g wtfpython
```

或者，安装 pypi 包 wtfpython

```
$ pip install wtfpython -U
```

现在，在命令行中运行 wtfpython，你就可以开始浏览了。

示例

大脑运动！

微妙的字符串*

1.

```
>>> a = "some_string"
>>> id(a)
140420665652016
>>> id("some" + "_" + "string") # 注意两个的id值是相同的。
140420665652016
```

2.

```
>>> a = "wtf"
>>> b = "wtf"
>>> a is b
True

>>> a = "wtf!"
>>> b = "wtf!"
>>> a is b
False

>>> a, b = "wtf!", "wtf!"
>>> a is b
True
```

3.

```
>>> 'a' * 20 is 'aaaaaaaaaaaaaaaaaaaaaa'  
True  
>>> 'a' * 21 is 'aaaaaaaaaaaaaaaaaaaaaa'  
False
```

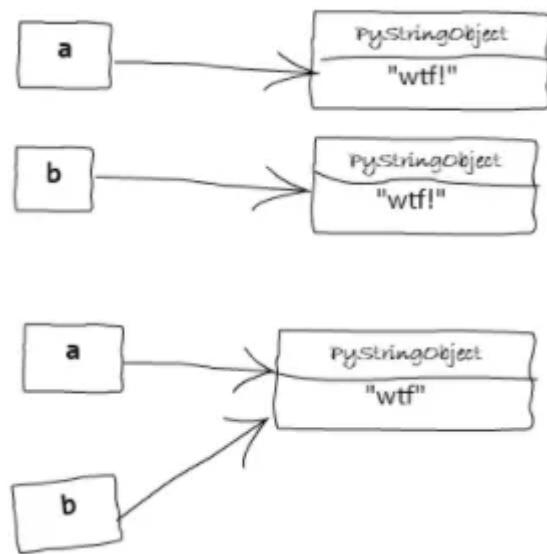
说明：

这些行为是由于 CPython 在编译优化时，某些情况下会尝试使用已经存在的不可变对象而不是每次都创建一个新对象。(这种行为被称作字符串的驻留[string interning])

发生驻留之后，许多变量可能指向内存中的相同字符串对象。(从而节省内存)

在上面的代码中，字符串是隐式驻留的。何时发生隐式驻留则取决于具体的实现。这里有一些方法可以用来猜测字符串是否会被驻留：

- 所有长度为 0 和长度为 1 的字符串都被驻留。
- 字符串在编译时被实现 ('wtf' 将被驻留，但是 ''.join(['w', 't', 'f']) 将不会被驻留)
- 字符串中只包含字母，数字或下划线时将会驻留。所以 'wtf!' 由于包含！而未被驻留。
可以在[这里](#)找CPython对此规则的实现。



当在同一行将 `a` 和 `b` 的值设置为 `"wtf!"` 的时候，Python 解释器会创建一个新对象，然后同时引用第二个变量。如果你在不同的行上进行赋值操作，它就不会“知道”已经有一个 `wtf!` 对象（因为 `"wtf!"` 不是按照上面提到的方式被隐式驻留的）。它是一种编译器优化，特别适用于交互式环境。

常量折叠(constant folding) 是 Python 中的一种窥孔优化(peephole optimization) 技术。这意味着在编译时表达式 `'a'*20` 会被替换为 `'aaaaaaaaaaaaaaaaaaaa'` 以减少运行时的时钟周期。只有长度小于 20 的字符串才会发生常量折叠。(为啥？想象一下由于表达式`'a'*10**10`而生成的 .pyc 文件的大小). 相关的源码实现在这里。

https://github.com/leisurelicht/wtfpython-cn/blob/master/images/string-intern/string_intern.png

是时候来点蛋糕了！

1.

```
some_dict = {}
some_dict[5.5] = "Ruby"
some_dict[5.0] = "JavaScript"
some_dict[5] = "Python"
```

Output:

```
>>> some_dict[5.5]
"Ruby"
>>> some_dict[5.0]
"Python"
>>> some_dict[5]
"Python"
```

"Python" 消除了 "JavaScript" 的存在？

说明：

Python 字典通过检查键值是否相等和比较哈希值来确定两个键是否相同。
具有相同值的不可变对象在Python中始终具有相同的哈希值。

```
>>> 5 == 5.0
True
>>> hash(5) == hash(5.0)
True
```

注意：具有不同值的对象也可能具有相同的哈希值（哈希冲突）。

当执行 `some_dict[5] = "Python"` 语句时，因为Python将 5 和 5.0 识别为 `some_dict` 的同一个键，所以已有值 "JavaScript" 就被 "Python" 覆盖了。

这个 StackOverflow 的回答 漂亮的解释了这背后的基本原理。

到处返回！

```
def some_func():
    try:
        return 'from_try'
    finally:
        return 'from_finally'
```

Output:

```
>>> some_func()  
  
'from_finally'
```

说明:

当在 "try...finally" 语句的 try 中执行 return, break 或 continue 后, finally 子句依然会执行。

函数的返回值由最后执行的 return 语句决定. 由于 finally 子句一定会执行, 所以 finally 子句中的 return 将始终是最后执行的语句。

本质上,我们都一样. *

```
class WTF:  
    pass
```

Output:

```
>>> WTF() == WTF() # 两个不同的对象应该不相等  
  
False  
  
>>> WTF() is WTF() # 也不相同  
  
False  
  
>>> hash(WTF()) == hash(WTF()) # 哈希值也应该不同  
  
True  
  
>>> id(WTF()) == id(WTF())
```

True**说明：**

当调用 `id` 函数时, Python 创建了一个 `WTF` 类的对象并传给 `id` 函数. 然后 `id` 函数获取其 `id` 值 (也就是内存地址), 然后丢弃该对象. 该对象就被销毁了.

当我们连续两次进行这个操作时, Python 会将相同的内存地址分配给第二个对象. 因为 (在 CPython 中) `id` 函数使用对象的内存地址作为对象的 `id` 值, 所以两个对象的 `id` 值是相同的.

综上, 对象的 `id` 值仅仅在对象的生命周期内唯一. 在对象被销毁之后, 或被创建之前, 其他对象可以具有相同的 `id` 值.

那为什么 `is` 操作的结果为 `False` 呢? 让我们看看这段代码.

```
class WTF(object):
    def __init__(self): print("I")
    def __del__(self): print("D")
```

Output:

```
>>> WTF() is WTF()
I
I
D
D
False
>>> id(WTF()) == id(WTF())
I
D
I
D
True
```

正如你所看到的，对象销毁的顺序是造成所有不同之处的原因。

为什么？

```
some_string = "wtf"
some_dict = {}
for i, some_dict[i] in enumerate(some_string):
    pass
```

Output:

```
>>> some_dict # 创建了索引字典。
{0: 'w', 1: 't', 2: 'f'}
```

说明：

Python 语法 中对 for 的定义是：

```
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
```

其中 exprlist 指分配目标。这意味着对可迭代对象中的每一项都会执行类似 {exprlist} = {next_value} 的操作。

一个有趣的例子说明了这一点：

```
for i in range(4):
    print(i)
    i = 10
```

Output:

```
0
1
```

2

3

你可曾觉得这个循环只会运行一次？

说明：

由于循环在Python中工作方式，赋值语句 `i = 10` 并不会影响迭代循环，在每次迭代开始之前，迭代器(这里指 `range(4)`) 生成的下一个元素就被解包并赋值给目标列表的变量(这里指 `i`)了。

在每一次的迭代中，`enumerate(some_string)` 函数就生成一个新值 `i` (计数器增加) 并从 `some_string` 中获取一个字符。然后将字典 `some_dict` 键 `i` (刚刚分配的) 的值设为该字符。本例中循环的展开可以简化为：

```
>>> i, some_dict[i] = (0, 'w')
>>> i, some_dict[i] = (1, 't')
>>> i, some_dict[i] = (2, 'f')
>>> some_dict
```

评估时间差异

1.

```
array = [1, 8, 15]
g = (x for x in array if array.count(x) > 0)
array = [2, 8, 22]
```

Output:

```
>>> print(list(g))
[8]
```

2.

```
array_1 = [1, 2, 3, 4]
g1 = (x for x in array_1)
array_1 = [1, 2, 3, 4, 5]

array_2 = [1, 2, 3, 4]
g2 = (x for x in array_2)
array_2[:] = [1, 2, 3, 4, 5]
```

Output:

```
>>> print(list(g1))
[1, 2, 3, 4]

>>> print(list(g2))
[1, 2, 3, 4, 5]
```

说明：

在生成器表达式中，`in` 子句在声明时执行，而条件子句则是在运行时执行。

所以在运行前，`array` 已经被重新赋值为 `[2, 8, 22]`，因此对于之前的 `1, 8` 和 `15`，只有 `count(8)` 的结果是大于 `0` 的，所以生成器只会生成 `8`。

第二部分中 `g1` 和 `g2` 的输出差异则是由于变量 `array_1` 和 `array_2` 被重新赋值的方式导致的。

在第一种情况下，`array_1` 被绑定到新对象 `[1,2,3,4,5]`，因为 `in` 子句是在声明时被执行的，所以它仍然引用旧对象 `[1,2,3,4]`(并没有被销毁)。

在第二种情况下，对 `array_2` 的切片赋值将相同的旧对象 `[1,2,3,4]` 原地更新为 `[1,2,3,4,5]`。因此 `g2` 和 `array_2` 仍然引用同一个对象(这个对象现在已经更新为 `[1,2,3,4,5]`)。



由于字数和排版受限，接下来的内容我们将用图片的形式呈现，感兴趣的同学请自行前往 GitHub 链接查看原码。

原文链接：

<https://github.com/satwikkansal/wtfpython>

中文版：

<https://github.com/leisurelicht/wtfpython-cn#section-strain-your-brain>

2018/11/30

GitHub - leisurelicht/wtfpython-cn: wtfpython的中文翻译/施工结束/ 能力有限，欢迎帮我改进翻译

```
D
False
>>> id(WTF()) == id(WTF())
I
D
I
D
True
```

正如你所看到的，对象销毁的顺序是造成所有不同之处的原因。

> For what?/为什么?

```
some_string = "wtf"
some_dict = {}
for i, some_dict[i] in enumerate(some_string):
    pass
```

Output:

```
>>> some_dict # 创建了索引字典。
{0: 'w', 1: 't', 2: 'f'}
```

💡 说明:

- Python 语法 中对 for 的定义是:

```
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
```

其中 exprlist 指分配目标。这意味着对可迭代对象中的每一项都会执行类似 `{exprlist} = {next_value}` 的操作。

一个有趣的例子说明了这一点:

```
for i in range(4):
    print(i)
    i = 10
```

Output:

```
0
1
2
3
```

你可能觉得这个循环只会运行一次？

💡 说明:

- 由于循环在Python中工作方式，赋值语句 `i = 10` 并不会影响迭代循环，在每次迭代开始之前，迭代器(这里指 `range(4)`)生成的下一个元素就被解包并赋值给目标列表的变量(这里指 `i`)了。
- 在每一次的迭代中，`enumerate(some_string)` 函数就生成一个新值 `i` (计数器增加) 并从 `some_string` 中获取一个字符。然后将字典 `some_dict` 键 `i` (刚刚分配的) 的值设为该字符。本例中循环的展开可以简化为:

```
>>> i, some_dict[i] = (0, 'w')
>>> i, some_dict[i] = (1, 't')
>>> i, some_dict[i] = (2, 'f')
>>> some_dict
```

> Evaluation time discrepancy/评估时间差异

2018/11/30

Github - leisurelicht/wtffpython-cn: wtffpython的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

1.

```
array = [1, 8, 15]
g = (x for x in array if array.count(x) > 0)
array = [2, 8, 22]
```

Output:

```
>>> print(list(g))
[8]
```

2.

```
array_1 = [1,2,3,4]
g1 = (x for x in array_1)
array_1 = [1,2,3,4,5]

array_2 = [1,2,3,4]
g2 = (x for x in array_2)
array_2[:] = [1,2,3,4,5]
```

Output:

```
>>> print(list(g1))
[1,2,3,4]
>>> print(list(g2))
[1,2,3,4,5]
```

💡 说明

- 在生成器表达式中, `in` 子句在声明时执行, 而条件子句则是在运行时执行.
- 所以在运行前, `array` 已经被重新赋值为 `[2, 8, 22]`, 因此对于之前的 `1, 8` 和 `15`, 只有 `count(8)` 的结果是大于 `0` 的, 所以生成器只会生成 `8`.
- 第二部分中 `g1` 和 `g2` 的输出差异则是由于变量 `array_1` 和 `array_2` 被重新赋值的方式导致的.
- 在第一种情况下, `array_1` 被绑定到新对象 `[1,2,3,4,5]`, 因为 `in` 子句是在声明时被执行的, 所以它仍然引用旧对象 `[1,2,3,4]` (并没有被销毁).
- 在第二种情况下, 对 `array_2` 的切片赋值将相同的旧对象 `[1,2,3,4]` 原地更新为 `[1,2,3,4,5]`. 因此 `g2` 和 `array_2` 仍然引用同一个对象(这个对象现在已经更新为 `[1,2,3,4,5]`).

> `is` is not what it is!/出人意料的 `is` !

下面是一个在互联网上非常有名的例子.

```
>>> a = 256
>>> b = 256
>>> a is b
True

>>> a = 257
>>> b = 257
>>> a is b
False

>>> a = 257; b = 257
>>> a is b
True
```

💡 说明:

`is` 和 `==` 的区别

- `is` 运算符检查两个运算对象是否引用自同一对象 (即, 它检查两个预算对象是否相同).

2018/11/30

Github - leisurelicht/wtfpython-cn: wtfpython的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

- == 运算符比较两个运算对象的值是否相等.
- 因此 is 代表引用相同, == 代表值相等. 下面的例子可以很好的说明这点.

```
>>> [] == []
True
>>> [] is [] # 这两个空列表位于不同的内存地址,
False
```

256 是一个已经存在的对象, 而 257 不是

当你启动Python 的时候, -5 到 256 的数值就已经被分配好了. 这些数字因为经常使用所以适合被提前准备好.

引用自 <https://docs.python.org/3/c-api/long.html>

当前的实现为-5到256之间的所有整数保留一个整数对象数组, 当你创建了一个该范围内的整数时, 你只需要返回现有对象的引用. 所以改变1的值是有可能的. 我怀疑这种行为在Python中是未定义行为. :-)

```
>>> id(256)
10922528
>>> a = 256
>>> b = 256
>>> id(a)
10922528
>>> id(b)
10922528
>>> id(257)
140084850247312
>>> x = 257
>>> y = 257
>>> id(x)
140084850247440
>>> id(y)
140084850247344
```

这里解释器并没有智能到能在执行 y = 257 时意识到我们已经创建了一个整数 257 , 所以它在内存中又新建了另一个对象.

当 a 和 b 在同一行中使用相同的值初始化时, 会指向同一个对象.

```
>>> a, b = 257, 257
>>> id(a)
140640774013296
>>> id(b)
140640774013296
>>> a = 257
>>> b = 257
>>> id(a)
140640774013392
>>> id(b)
140640774013488
```

- 当 a 和 b 在同一行中被设置为 257 时, Python 解释器会创建一个新对象, 然后同时引用第二个变量. 如果你在不同的行上进行, 它就不会 "知道" 已经存在一个 257 对象了.
- 这是一种特别为交互式环境做的编译器优化. 当你在实时解释器中输入两行的时候, 他们会单独编译, 因此也会单独进行优化. 如果你在 .py 文件中尝试这个例子, 则不会看到相同的行为, 因为文件是一次性编译的.

> A tic-tac-toe where X wins in the first attempt!/一蹴即至!

```
# 我们先初始化一个变量row
row = [""]*3 #row i[', ', ', '']
# 并创建一个变量board
board = [row]*3
```

Output:

```
>>> board
[[' ', ' ', ''], [' ', ' ', ''], [' ', ' ', '']]
```

2018/11/30

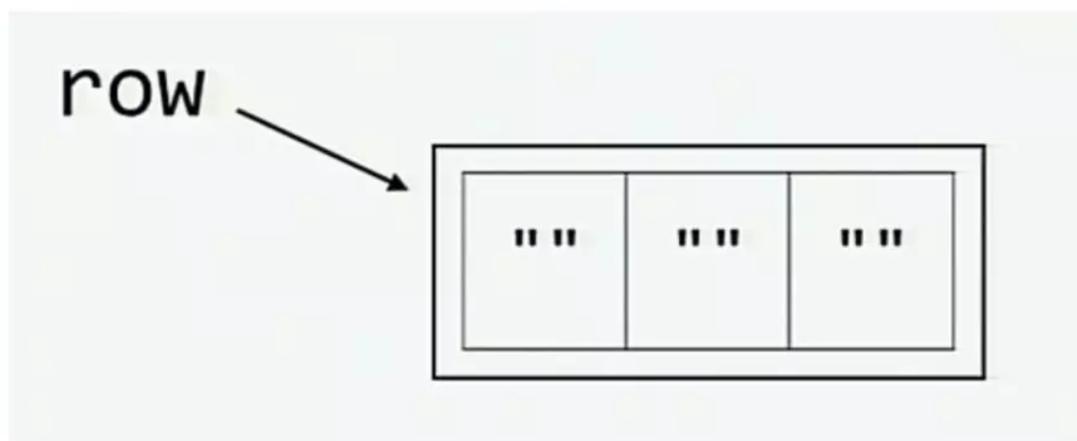
Github - leisurelicht/wtffpython-cn: wtffpython的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

```
>>> board[0]
['', '', '']
>>> board[0][0]
 ''
>>> board[0][0] = "X"
>>> board
[['X', '', ''], ['X', '', ''], ['X', '', '']]
```

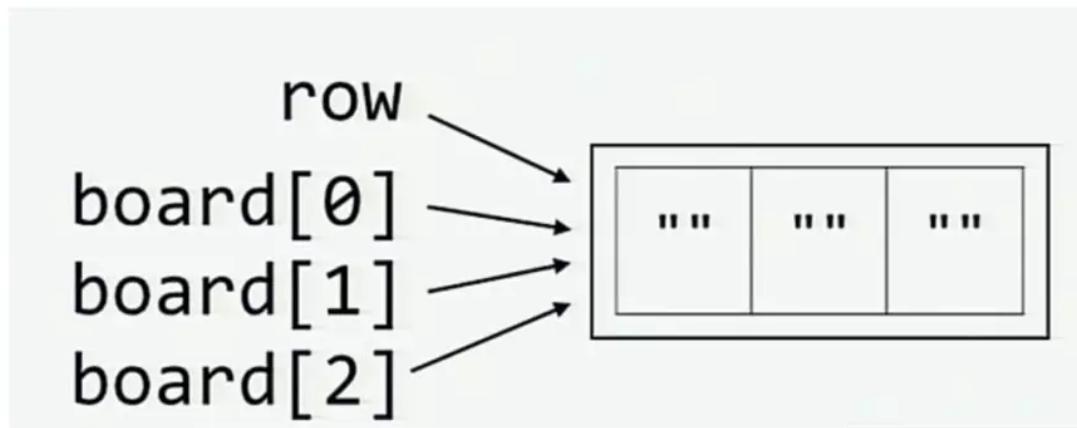
我们有没有赋值过3个 "X" 呢?

说明:

当我们初始化 `row` 变量时, 下面这张图展示了内存中的情况。



而当通过对 `row` 做乘法来初始化 `board` 时, 内存中的情况则如下图所示(每个元素 `board[0]`, `board[1]` 和 `board[2]` 都和 `row` 一样引用了同一列表。)



我们可以通过不使用变量 `row` 生成 `board` 来避免这种情况 ([这个issue](#)提出了这个需求。)

```
>>> board = [['']*3 for _ in range(3)]
>>> board[0][0] = "X"
>>> board
[['X', '', ''], [' ', ' ', ''], [' ', ' ', '']]
```

> The sticky output function/麻烦的输出

```
funcs = []
results = []
for x in range(7):
    def some_func():
        return x
    funcs.append(some_func)
```

<https://github.com/leisurelicht/wtffpython-cn>

10/37

2018/11/30 GitHub - leisurelicht/wtffpyton-cn: wtffpyton的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

```
results.append(some_func())
funcs_results = [func() for func in funcs]
```

Output:

```
>>> results
[0, 1, 2, 3, 4, 5, 6]
>>> funcs_results
[6, 6, 6, 6, 6, 6, 6]
```

即使每次在迭代中将 `some_func` 加入 `funcs` 前的 `x` 值都不相同, 所有的函数还是都返回6.

// 再换个例子

```
>>> powers_of_x = [lambda x: x**i for i in range(10)]
>>> [f(2) for f in powers_of_x]
[512, 512, 512, 512, 512, 512, 512, 512, 512]
```

💡 说明:

- 当在循环内部定义一个函数时, 如果该函数在其主体中使用了循环变量, 则闭包函数将与循环变量绑定, 而不是它的值. 因此, 所有的函数都是使用最后分配给变量的值来进行计算的.
- 可以通过将循环变量作为命名变量传递给函数来获得预期的结果. 为什么这样可行? 因为这会在函数内再次定义一个局部变量.

```
funcs = []
for x in range(7):
    def some_func(x=x):
        return x
    funcs.append(some_func)
```

Output:

```
>>> funcs_results = [func() for func in funcs]
>>> funcs_results
[0, 1, 2, 3, 4, 5, 6]
```

> is not ... is not is (not ...) / is not ... 不是 is (not ...)

```
>>> 'something' is not None
True
>>> 'something' is (not None)
False
```

💡 说明:

- `is not` 是个单独的二进制运算符, 和分别使用 `is` 和 `not` 不同.
- 如果操作符两侧的变量指向同一个对象, 则 `is not` 的结果为 `False`, 否则结果为 `True`.

> The surprising comma/意外的逗号

Output:

```
>>> def f(x, y):
...     print(x, y)
...
>>> def g(x=4, y=5,):
...     print(x, y)
...
```

2018/11/30 GitHub - leisurelicht/wtfpython-cn: wtfpython的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

```
>>> def h(x, **kwargs,):
    File "<stdin>", line 1
        def h(x, **kwargs,):
            ^
SyntaxError: invalid syntax
>>> def h(*args,):
    File "<stdin>", line 1
        def h(*args,):
            ^
SyntaxError: invalid syntax
```

💡 说明:

- 在Python函数的形式参数列表中, 尾随逗号并不一定是合法的.
- 在Python中, 参数列表部分用前置逗号定义, 部分用尾随逗号定义. 这种冲突导致逗号被夹在中间, 没有规则定义它.(译这一句看得我也很懵逼, 只能强翻了. 详细解释看下面的讨论帖会一目了然.)
- 注意: 尾随逗号的问题已经在Python 3.6中被修复了. 而这篇帖子中则简要讨论了Python中尾随逗号的不同用法.

> Backslashes at the end of string/字符串末尾的反斜杠

Output:

```
>>> print("\\ C:\\")
\ C:\\
>>> print(r"\ C:")
\ C:
>>> print(r"\ C:\\")

File "<stdin>", line 1
    print(r"\ C:\\")
           ^
SyntaxError: EOL while scanning string literal
```

💡 说明:

- 在以 `r` 开头的原始字符串中, 反斜杠并没有特殊含义.

```
>>> print(repr(r"wt\f"))
'wt\\\"f'
```

- 解释器所做的只是简单的改变了反斜杠的行为, 因此会直接放行反斜杠及后一个的字符. 这就是反斜杠在原始字符串末尾不起作用的原因.

> not knot!/别纠结!

```
x = True
y = False
```

Output:

```
>>> not x == y
True
>>> x == not y
File "<input>", line 1
    x == not y
           ^
SyntaxError: invalid syntax
```

💡 说明:

- 运算符的优先级会影响表达式的求值顺序, 而在 Python 中 `==` 运算符的优先级要高于 `not` 运算符.
- 所以 `not x == y` 相当于 `not (x == y)`, 同时等价于 `not (True == False)`, 最后的运算结果就是 `True`.

2018/11/30

Github - leisurelicht/wtffpythont-cn: wtffpythont的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

- 之所以 `x == not y` 会抛一个 `SyntaxError` 异常, 是因为它会被认为等价于 `(x == not) y`, 而不是你一开始期望的 `x == (not y)` .
- 解释器期望 `not` 标记是 `not in` 操作符的一部分(因为 `==` 和 `not in` 操作符具有相同的优先级), 但是它在 `not` 标记后面找不到 `in` 标记, 所以会抛出 `SyntaxError` 异常.

> Half triple-quoted strings/三个引号

Output:

```
>>> print('wtffpythont''')
wtffpythont
>>> print("wtffpythont""")
wtffpythont
>>> # 下面的语句会抛出 `SyntaxError` 异常
>>> # print(''wtffpythont')
>>> # print("""wtffpythont")
```



- Python 提供隐式的字符串链接, 例如,

```
>>> print("wtf" "python")
wtffpythont
>>> print("wtf" "") # or "wtf"""
wtf
```

- `'''` 和 `"""` 在 Python 中也是字符串定界符, Python 解释器在先遇到三个引号的时候会尝试再寻找三个终止引号作为定界符, 如果不存在则会导致 `SyntaxError` 异常.

> Midnight time doesn't exist?/不存在的午夜?

```
from datetime import datetime

midnight = datetime(2018, 1, 1, 0, 0)
midnight_time = midnight.time()

noon = datetime(2018, 1, 1, 12, 0)
noon_time = noon.time()

if midnight_time:
    print("Time at midnight is", midnight_time)

if noon_time:
    print("Time at noon is", noon_time)
```

Output:

```
('Time at noon is', datetime.time(12, 0))
```

midnight_time 并没有被输出.



在 Python 3.5 之前, 如果 `datetime.time` 对象存储的 UTC 的午夜时间(译: 就是 00:00), 那么它的布尔值会被认为是 `False`. 当使用 `if obj:` 语句来检查 `obj` 是否为 `null` 或者某些“空”值的时候, 很容易出错.

> What's wrong with booleans?/布尔你咋了?

1.

```
# 一个简单的例子, 统计下面可迭代对象中的布尔型值的个数和整型值的个数
mixed_list = [False, 1.0, "some_string", 3, True, [], False]
```

2018/11/30

GitHub - leisurelicht/wtfpython-cn: wtfpython的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

```

integers_found_so_far = 0
booleans_found_so_far = 0

for item in mixed_list:
    if isinstance(item, int):
        integers_found_so_far += 1
    elif isinstance(item, bool):
        booleans_found_so_far += 1

```

Output:

```

>>> booleans_found_so_far
0
>>> integers_found_so_far
4

```

2.

```

another_dict = {}
another_dict[True] = "JavaScript"
another_dict[1] = "Ruby"
another_dict[1.0] = "Python"

```

Output:

```

>>> another_dict[True]
"Python"

```

3.

```

>>> some_bool = True
>>> "wtf"*some_bool
'wtf'
>>> some_bool = False
>>> "wtf"*some_bool
''

```

说明:

- 布尔值是 int 的子类

```

>>> isinstance(True, int)
True
>>> isinstance(False, int)
True

```

- 所以 True 的整数值是 1, 而 False 的整数值是 0.

```

>>> True == 1 == 1.0 and False == 0 == 0.0
True

```

- 关于其背后的原理, 请看这个 StackOverflow 的[回答](#).

> Class attributes and instance attributes/类属性和实例属性

1.

```

class A:
    x = 1

class B(A):
    pass

```

2018/11/30

GitHub - leisurelicht/wtffpyton-cn: wtffpyton的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

```
class C(A):
    pass
```

Output:

```
>>> A.x, B.x, C.x
(1, 1, 1)
>>> B.x = 2
>>> A.x, B.x, C.x
(1, 2, 1)
>>> A.x = 3
>>> A.x, B.x, C.x
(3, 2, 3)
>>> a = A()
>>> a.x, A.x
(3, 3)
>>> a.x += 1
>>> a.x, A.x
(4, 3)
```

2.

```
class SomeClass:
    some_var = 15
    some_list = [5]
    another_list = [5]
    def __init__(self, x):
        self.some_var = x + 1
        self.some_list = self.some_list + [x]
        self.another_list += [x]
```

Output:

```
>>> some_obj = SomeClass(420)
>>> some_obj.some_list
[5, 420]
>>> some_obj.another_list
[5, 420]
>>> another_obj = SomeClass(111)
>>> another_obj.some_list
[5, 111]
>>> another_obj.another_list
[5, 420, 111]
>>> another_obj.another_list is SomeClass.another_list
True
>>> another_obj.another_list is some_obj.another_list
True
```

说明:

- 类变量和实例变量在内部是通过类对象的字典来处理(译:就是 `__dict__` 属性). 如果在当前类的字典中找不到的话就去它的父类中寻找.
- `+=` 运算符会在原地修改可变对象, 而不是创建新对象. 因此, 修改一个实例的属性会影响其他实例和类属性.

> yielding None/生成 None

```
some_iterable = ('a', 'b')

def some_func(val):
    return "something"
```

Output:

```
>>> [x for x in some_iterable]
['a', 'b']
>>> [(yield x) for x in some_iterable]
```

2018/11/30

GitHub - leisurelicht/wtfpython-cn: wtfpython的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

```
<generator object <listcomp> at 0x7f70b0a4ad58>
>>> list([(yield x) for x in some_iterable])
['a', 'b']
>>> list((yield x) for x in some_iterable)
['a', None, 'b', None]
>>> list(some_func((yield x)) for x in some_iterable)
['a', 'something', 'b', 'something']
```

⌚ 说明:

- 来源和解释可以在这里找到: <https://stackoverflow.com/questions/32139885/yield-in-list-comprehensions-and-generator-expressions>
- 相关错误报告: <http://bugs.python.org/issue10544>

> Mutating the immutable!/强人所难

```
some_tuple = ("A", "tuple", "with", "values")
another_tuple = ([1, 2], [3, 4], [5, 6])
```

Output:

```
>>> some_tuple[2] = "change this"
TypeError: 'tuple' object does not support item assignment
>>> another_tuple[2].append(1000) # 这里不出现错误
>>> another_tuple
([1, 2], [3, 4], [5, 6, 1000])
>>> another_tuple[2] += [99, 999]
TypeError: 'tuple' object does not support item assignment
>>> another_tuple
([1, 2], [3, 4], [5, 6, 1000, 99, 999])
```

我还以为元组是不可变的呢...

⌚ 说明:

- 引用 <https://docs.python.org/2/reference/datamodel.html>

不可变序列 不可变序列的对象一旦创建就不能再改变. (如果对象包含对其他对象的引用, 则这些其他对象可能是可变的并且可能会被修改; 但是, 由不可变对象直接引用的对象集合不能更改.)

- += 操作符在原地修改了列表. 元素赋值操作并不工作, 但是当异常抛出时, 元素已经在原地被修改了.

(译: 对于不可变对象, 这里指tuple, += 并不是原子操作, 而是 extend 和 = 两个动作, 这里 = 操作虽然会抛出异常, 但 extend 操作已经修改成功了. 详细解释可以看[这里](#))

> The disappearing variable from outer scope/消失的外部变量

```
e = 7
try:
    raise Exception()
except Exception as e:
    pass
```

Output (Python 2.x):

```
>>> print(e)
# prints nothing
```

Output (Python 3.x):

```
>>> print(e)
NameError: name 'e' is not defined
```

2018/11/30

GitHub - leisurelicht/wtffpyton-cn: wtffpyton的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

说明:

- 出处: https://docs.python.org/3/reference/compound_stmts.html#except

当使用 `as` 为目标分配异常的时候, 将在`except`子句的末尾清除该异常.

这就好像

```
except E as N:  
    foo
```

会被翻译成

```
except E as N:  
    try:  
        foo  
    finally:  
        del N
```

这意味着异常必须在被赋值给其他变量才能在 `except` 子句之后引用它. 而异常之所以会被清除, 则是由于上面附加的回溯信息(trackback)会和栈帧(stack frame)形成循环引用, 使得该栈帧中的所有本地变量在下一次垃圾回收发生之前都处于活动状态. (译: 也就是说不会被回收)

- 子句在 Python 中并没有独立的作用域. 示例中的所有内容都处于同一作用域内, 所以变量 `e` 会由于执行了 `except` 子句而被删除. 而对于有独立的内部作用域的函数来说情况就不一样了. 下面的例子说明了这一点:

```
def f(x):  
    del(x)  
    print(x)  
  
x = 5  
y = [5, 4, 3]
```

Output:

```
>>>f(x)  
UnboundLocalError: local variable 'x' referenced before assignment  
>>>f(y)  
UnboundLocalError: local variable 'x' referenced before assignment  
>>> x  
5  
>>> y  
[5, 4, 3]
```

- 在 Python 2.x 中, `Exception()` 实例被赋值给了变量 `e`, 所以当你尝试打印结果的时候, 它的输出为空. (译: 正常的Exception实例打印出来就是空)

Output (Python 2.x):

```
>>> e  
Exception()  
>>> print e  
# 没有打印任何内容!
```

> When True is actually False/真亦假

```
True = False  
if True == False:  
    print("I've lost faith in truth!")
```

Output:

```
I've lost faith in truth!
```

2018/11/30

GitHub - leisurelicht/wtffpythont-cn: wtffpythont-cn的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

⌚ 说明:

- 最初, Python 并没有 `bool` 型(人们用0表示假值, 用非零值比如1作为真值). 后来他们添加了 `True`, `False`, 和 `bool` 型, 但是, 为了向后兼容, 他们没法把 `True` 和 `False` 设置为常量, 只是设置成了内置变量.
- Python 3 由于不再需要向后兼容, 终于可以修复这个问题了, 所以这个例子无法在 Python 3.x 中执行!

> From filled to None in one instruction.../从有到无...

```
some_list = [1, 2, 3]
some_dict = {
    "key_1": 1,
    "key_2": 2,
    "key_3": 3
}

some_list = some_list.append(4)
some_dict = some_dict.update({"key_4": 4})
```

Output:

```
>>> print(some_list)
None
>>> print(some_dict)
None
```

⌚ 说明:

大多数修改序列/映射对象的方法, 比如 `list.append`, `dict.update`, `list.sort` 等等. 都是原地修改对象并返回 `None`. 这样做的原因是, 如果操作可以原地完成, 就可以避免创建对象的副本, 来提高性能. (参考[这里](#))

> Subclass relationships/子类关系 *

Output:

```
>>> from collections import Hashable
>>> issubclass(list, object)
True
>>> issubclass(object, Hashable)
True
>>> issubclass(list, Hashable)
False
```

子类关系应该是可传递的, 对吧? (即, 如果 A 是 B 的子类, B 是 C 的子类, 那么 A 应该是 C 的子类)

⌚ 说明:

- Python 中的子类关系并不必须是传递的. 任何人都可以在元类中随意定义 `__subclassescheck__`.
- 当 `issubclass(cls, Hashable)` 被调用时, 它只是在 `cls` 中寻找 `__hash__` 方法或继承自 `__hash__` 的方法.
- 由于 `object` 是可散列的(`hashable`), 但是 `list` 是不可散列的, 所以它打破了这种传递关系.
- 在[这里](#)可以找到更详细的解释.

> The mysterious key type conversion/神秘的键型转换 *

```
class SomeClass(str):
    pass

some_dict = {'s': 42}
```

Output:

<https://github.com/leisurelicht/wtffpythont-cn>

18/37

2018/11/30

GitHub - leisurelicht/wtffpythont-cn: wtffpythont的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

```
>>> type(list(some_dict.keys())[0])
str
>>> s = SomeClass('s')
>>> some_dict[s] = 40
>>> some_dict # 预期: 两个不同的键值对
{'s': 40}
>>> type(list(some_dict.keys())[0])
str
```

💡 说明:

- 由于 `SomeClass` 会从 `str` 自动继承 `__hash__` 方法, 所以 `s` 对象和 "s" 字符串的哈希值是相同的.
- 而 `SomeClass("s") == "s"` 为 `True` 是因为 `SomeClass` 也继承了 `str` 类 `__eq__` 方法.
- 由于两者的哈希值相同且相等, 所以它们在字典中表示相同的键.
- 如果想要实现期望的功能, 我们可以重定义 `SomeClass` 的 `__eq__` 方法.

```
class SomeClass(str):
    def __eq__(self, other):
        return (
            type(self) is SomeClass
            and type(other) is SomeClass
            and super().__eq__(other)
        )

# 当我们自定义 __eq__ 方法时, Python 不会再自动继承 __hash__ 方法
# 所以我们也需要定义它
__hash__ = str.__hash__

some_dict = {'s':42}
```

Output:

```
>>> s = SomeClass('s')
>>> some_dict[s] = 40
>>> some_dict
{'s': 40, 's': 42}
>>> keys = list(some_dict.keys())
>>> type(keys[0]), type(keys[1])
(__main__.SomeClass, str)
```

> Let's see if you can guess this?/看看你能否猜到这一点?

```
a, b = a[b] = {}, 5
```

Output:

```
>>> a
{5: ({...}, 5)}
```

💡 说明:

- 根据 [Python 语言参考](#), 赋值语句的形式如下

```
(target_list "=")+ (expression_list | yield_expression)
```

赋值语句计算表达式列表(expression list)(牢记 这可以是单个表达式或以逗号分隔的列表, 后者返回元组)并将单个结果对象从左到右分配给目标列表中的每一项.

- (`target_list` "=")+ 中的 + 意味着可以有一个或多个目标列表. 在这个例子中, 目标列表是 `a, b` 和 `a[b]` (注意表达式列表只能有一个, 在我们的例子中是 `{}, 5`).

2018/11/30

Github - leisurelicht/wtffpythont-cn: wtffpythont的中文翻译/施工结束/能力有限, 欢迎帮我改进翻译

- 表达式列表计算结束后, 将其值自动解包后从左到右分配给目标列表(target list). 因此, 在我们的例子中, 首先将 {}, 5 元祖并赋值给 a, b, 然后我们就可以得到 a = {} 且 b = 5.
- a 被赋值的 {} 是可变对象.
- 第二个目标列表是 a[b] (你可能觉得这里会报错, 因为在之前的语句中 a 和 b 都还没有被定义. 但是别忘了, 我们刚刚将 a 赋值 {} 且将 b 赋值为 5).
- 现在, 我们将通过将字典中键 s 的值设置为元祖 ((,), 5) 来创建循环引用 (输出中的 {...} 指与 a 引用了相同的对象). 下面是一个更简单的循环引用的例子

```
>>> some_list = some_list[0] = [0]
>>> some_list
[[...]]
>>> some_list[0]
[[...]]
>>> some_list is some_list[0]
True
>>> some_list[0][0][0][0][0][0] == some_list
True
```

我们的例子就是这种情况 (a[b][0] 与 a 是相同的对象)

- 总结一下, 你也可以把例子拆成

```
a, b = {}, 5
a[b] = a, b
```

并且可以通过 a[b][0] 与 a 是相同的对象来证明是循环引用

```
>>> a[b][0] is a
True
```

Section: Appearances are deceptive!/外表是靠不住的!

> Skipping lines?/跳过一行?

Output:

```
>>> value = 11
>>> value = 32
>>> value
11
```

什么鬼?

注意: 如果你想要重现的话最简单的方法是直接复制上面的代码片段到你的文件或命令行里.

💡 说明:

一些非西方字符虽然看起来和英语字母相同, 但会被解释器识别为不同的字母.

```
>>> ord('e') # 西里尔语的 'е' (Ye)
1877
>>> ord('e') # 拉丁语的 'e', 用于英文并使用标准键盘输入
101
>>> 'e' == 'е'
False

>>> value = 42 # 拉丁语 e
>>> value = 23 # 西里尔语 'е', Python 2.x 的解释器在这会抛出 'SyntaxError' 异常
>>> value
42
```

2018/11/30

Github - leisurelicht/wtffpythont-cn: wtffpythont的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

内置的 `ord()` 函数可以返回一个字符的 Unicode 代码点. 这里西里尔语 'е' 和拉丁语 'e' 的代码点不同证实了上述例子.

> Teleportation/空间移动 *

```
import numpy as np

def energy_send(x):
    # 初始化一个 numpy 数组
    np.array([float(x)])

def energy_receive():
    # 返回一个空的 numpy 数组
    return np.empty(), dtype=np.float).tolist()
```

Output:

```
>>> energy_send(123.456)
>>> energy_receive()
123.456
```

谁来给我发个诺贝尔奖?

💡 说明:

- 注意在 `energy_send` 函数中创建的 numpy 数组并没有返回, 因此内存空间被释放并可以被重新分配.
- `numpy.empty()` 直接返回下一段空闲内存, 而不重新初始化. 而这个内存点恰好就是刚刚释放的那个(通常情况下, 并不绝对).

> Well, something is fishy.../嗯, 有些可疑...

```
def square(x):
    """
    一个通过加法计算平方的简单函数.
    """
    sum_so_far = 0
    for counter in range(x):
        sum_so_far = sum_so_far + x
    return sum_so_far
```

Output (Python 2.x):

```
>>> square(10)
10
```

难道不应该是100吗?

注意: 如果你无法重现, 可以尝试运行这个文件[mixed_tabs_and_spaces.py](#).

💡 说明:

- 不要混用制表符(tab)和空格(space)! 在上面的例子中, `return` 的前面是"1个制表符", 而其他部分的代码前面是"4个空格".
- Python是这么处理制表符的:

首先, 制表符会从左到右依次被替换成8个空格, 直到被替换后的字符总数是八的倍数 <...>

- 因此, `square` 函数最后一行的制表符会被替换成8个空格, 导致`return`语句进入循环语句里面.
- Python 3 很友好, 在这种情况下会自动抛出错误.

Output (Python 3.x):

```
TabError: inconsistent use of tabs and spaces in indentation
```

Section: Watch out for the landmines! / 小心地雷!

> Modifying a dictionary while iterating over it / 迭代字典时的修改

```
x = {0: None}

for i in x:
    del x[i]
    x[i+1] = None
    print(i)
```

Output (Python 2.7- Python 3.5):

```
0
1
2
3
4
5
6
7
```

是的, 它运行了8次然后才停下来.



- Python不支持对字典进行迭代的同时修改它.
- 它之所以运行8次, 是因为字典会自动扩容以容纳更多键值(我们有8次删除记录, 因此需要扩容). 这实际上是一个实现细节.(译: 应该是因为字典的初始最小值是8, 扩容会导致散列表地址发生变化而中断循环.)
- 在不同的Python实现中删除键的处理方式以及调整大小的时间可能会有所不同.(译: 就是什么时候扩容在不同版本中可能是不同的, 在3.6及3.7的版本中到5就会自动扩容了. 以后也有可能再次发生变化. 顺带一提, 后面两次扩容会扩展为32和256. 8->32->256)
- 更多的信息, 你可以参考这个StackOverflow的回答, 它详细的解释一个类似的例子.

> Stubborn del operator / 坚强的 del *

```
class SomeClass:
    def __del__(self):
        print("Deleted!")
```

Output: 1.

```
>>> x = SomeClass()
>>> y = x
>>> del x # 这里应该会输出 "Deleted!"
>>> del y
Deleted!
```

噢, 终于删除了. 你可能已经猜到了在我们第一次尝试删除 x 时是什么让 `__del__` 免于被调用的. 那让我们给这个例子增加点难度.

2.

```
>>> x = SomeClass()
>>> y = x
>>> del x
>>> y # 检查一下y是否存在
<__main__.SomeClass instance at 0x7f98a1a67fc8>
>>> del y # 像之前一样, 这里应该会输出 "Deleted!"
```

2018/11/30

GitHub - leisurelicht/wtffpythont-cn: wtffpythont的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

```
>>> globals() # 好吧, 并没有. 让我们看一下所有的全局变量
Deleted!
{'__builtins__': <module '__builtin__' (built-in)>, 'SomeClass': <class __main__.SomeClass at 0x7f98a1a5f668>,
```

好了, 现在它被删除了 😊

Q 说明:

- `del x` 并不会立刻调用 `x.__del__()`.
- 每当遇到 `del x`, Python 会将 `x` 的引用数减1, 当 `x` 的引用数减到0时就会调用 `x.__del__()`.
- 在第二个例子中, `y.__del__()` 之所以未被调用, 是因为前一条语句 (`>>> y`) 对同一对象创建了另一个引用, 从而防止在执行 `del y` 后对象的引用数变为0.
- 调用 `globals` 导致引用被销毁, 因此我们可以看到 "Deleted!" 终于被输出了.
- (译: 这其实是 Python 交互解释器的特性. 它会自动让 `_` 保存上一个表达式输出的值. 详细可以看[这里](#))

> Deleting a list item while iterating/迭代列表时删除元素

```
list_1 = [1, 2, 3, 4]
list_2 = [1, 2, 3, 4]
list_3 = [1, 2, 3, 4]
list_4 = [1, 2, 3, 4]

for idx, item in enumerate(list_1):
    del item

for idx, item in enumerate(list_2):
    list_2.remove(item)

for idx, item in enumerate(list_3[:]):
    list_3.remove(item)

for idx, item in enumerate(list_4):
    list_4.pop(idx)
```

Output:

```
>>> list_1
[1, 2, 3, 4]
>>> list_2
[2, 4]
>>> list_3
[]
>>> list_4
[2, 4]
```

你能猜到为什么输出是 [2, 4] 吗?

Q 说明:

- 在迭代时修改对象是一个很愚蠢的主意. 正确的做法是迭代对象的副本, `list_3[:]` 就是这么做的.

```
>>> some_list = [1, 2, 3, 4]
>>> id(some_list)
139798789457608
>>> id(some_list[:]) # 注意python为切片列表创建了新对象.
139798779601192
```

`del`, `remove` 和 `pop` 的不同:

- `del var_name` 只是从本地或全局命名空间中删除了 `var_name` (这就是为什么 `list_1` 没有受到影响).
- `remove` 会删除第一个匹配到的指定值, 而不是特定的索引. 如果找不到值则抛出 `ValueError` 异常.
- `pop` 则会删除指定索引处的元素并返回它, 如果指定了无效的索引则抛出 `IndexError` 异常.

为什么输出是 [2, 4] ?

2018/11/30

Github - leisurelicht/wtfpython-cn: wtfpython的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

- 列表迭代是按索引进行的, 所以当我们从 `list_2` 或 `list_4` 中删除 1 时, 列表的内容就变成了 `[2, 3, 4]`. 剩余元素会依次位移, 也就是说, 2 的索引会变为 0, 3 会变为 1. 由于下一次迭代将获取索引为 1 的元素 (即 3), 因此 2 将被彻底的跳过. 类似的情况会交替发生在列表中的每个元素上.
- 参考这个StackOverflow的回答来解释这个例子
- 关于Python中字典的类似例子, 可以参考这个Stackoverflow的回答.

> Loop variables leaking out!/循环变量泄漏!

1.

```
for x in range(7):
    if x == 6:
        print(x, ': for x inside loop')
print(x, ': x in global')
```

Output:

```
6 : for x inside loop
6 : x in global
```

但是 x 从未在循环外被定义...

2.

```
# 这次我们先初始化x
x = -1
for x in range(7):
    if x == 6:
        print(x, ': for x inside loop')
print(x, ': x in global')
```

Output:

```
6 : for x inside loop
6 : x in global
```

3.

```
x = 1
print([x for x in range(5)])
print(x, ': x in global')
```

Output (on Python 2.x):

```
[0, 1, 2, 3, 4]
(4, ': x in global')
```

Output (on Python 3.x):

```
[0, 1, 2, 3, 4]
1 : x in global
```

说明:

- 在 Python 中, for 循环使用所在作用域并在结束后保留定义的循环变量. 如果我们曾在全局命名空间中定义过循环变量. 在这种情况下, 它会重新绑定现有变量.
- Python 2.x 和 Python 3.x 解释器在列表推导式示例中的输出差异, 在文档 [What's New In Python 3.0](#) 中可以找到相关的解释:

2018/11/30

GitHub - leisurelicht/wtfpython-cn: wtfpython的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

"列表推导不再支持句法形式 [... for var in item1, item2, ...]. 取而代之的是 [... for var in (item1, item2, ...)]. 另外, 注意列表推导具有不同的语义: 它们更接近于 list() 构造函数中生成器表达式的语法糖(译: 这一句我不是很明白), 特别是循环控制变量不再泄漏到周围的作用域中."

> Beware of default mutable arguments!/当心默认的可变参数!

```
def some_func(default_arg=[]):
    default_arg.append("some_string")
    return default_arg
```

Output:

```
>>> some_func()
['some_string']
>>> some_func()
['some_string', 'some_string']
>>> some_func([])
['some_string']
>>> some_func()
['some_string', 'some_string', 'some_string']
```

说明:

- Python中函数的默认可变参数并不是每次调用该函数时都会被初始化. 相反, 它们会使用最近分配的值作为默认值. 当我们明确的将 [] 作为参数传递给 some_func 的时候, 就不会使用 default_arg 的默认值, 所以函数会返回我们所期望的结果.

```
def some_func(default_arg=[]):
    default_arg.append("some_string")
    return default_arg
```

Output:

```
>>> some_func.__defaults__ # 这里会显示函数的默认参数的值
([],)
>>> some_func()
>>> some_func.__defaults__
(['some_string'],)
>>> some_func()
>>> some_func.__defaults__
(['some_string', 'some_string'],)
>>> some_func([])
>>> some_func.__defaults__
(['some_string', 'some_string'],)
```

- 避免可变参数导致的错误的常见做法是将 None 指定为参数的默认值, 然后检查是否有值传给对应的参数. 例:

```
def some_func(default_arg=None):
    if not default_arg:
        default_arg = []
    default_arg.append("some_string")
    return default_arg
```

> Catching the Exceptions/捕获异常

```
some_list = [1, 2, 3]
try:
    # 这里会抛出异常 ``IndexError``
    print(some_list[4])
except IndexError, ValueError:
    print("Caught!")

try:
    # 这里会抛出异常 ``ValueError``
    print("Not caught")
```

<https://github.com/leisurelicht/wtfpython-cn>

25/37

2018/11/30

GitHub - leisurelight/wtfpython-cn: wtfpython的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

```
some_list.remove(4)
except IndexError, ValueError:
    print("Caught again!")
```

Output (Python 2.x):

```
Caught!
ValueError: list.remove(x): x not in list
```

Output (Python 3.x):

```
File "<input>", line 3
  except IndexError, ValueError:
          ^
SyntaxError: invalid syntax
```

⌚ 说明:

- 如果你想要同时捕获多个不同类型的异常时, 你需要将它们用括号包成一个元组作为第一个参数传递. 第二个参数是可选名称, 如果你提供, 它将与被捕获的异常实例绑定. 例,

```
some_list = [1, 2, 3]
try:
    # 这里会抛出异常 ``ValueError``
    some_list.remove(4)
except (IndexError, ValueError), e:
    print("Caught again!")
    print(e)
```

Output (Python 2.x):

```
Caught again!
list.remove(x): x not in list
```

Output (Python 3.x):

```
File "<input>", line 4
  except (IndexError, ValueError), e:
          ^
IndentationError: unindent does not match any outer indentation level
```

- 在 Python 3 中, 用逗号区分异常与可选名称是无效的; 正确的做法是使用 as 关键字. 例,

```
some_list = [1, 2, 3]
try:
    some_list.remove(4)

except (IndexError, ValueError) as e:
    print("Caught again!")
    print(e)
```

Output:

```
Caught again!
list.remove(x): x not in list
```

> Same operands, different story!/同人不同命!

1.

2018/11/30

Github - leisurelicht/wtffpyton-cn: wtffpyton的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

```
a = [1, 2, 3, 4]
b = a
a += [5, 6, 7, 8]
```

Output:

```
>>> a
[1, 2, 3, 4, 5, 6, 7, 8]
>>> b
[1, 2, 3, 4]
```

2.

```
a = [1, 2, 3, 4]
b = a
a += [5, 6, 7, 8]
```

Output:

```
>>> a
[1, 2, 3, 4, 5, 6, 7, 8]
>>> b
[1, 2, 3, 4, 5, 6, 7, 8]
```

💡 说明:

- `a += b` 并不总是与 `a = a + b` 表现相同. 类实现 `op=` 运算符的方式也许 是不同的. 列表就是这样做的.
- 表达式 `a = a + [5,6,7,8]` 会生成一个新列表, 并让 `a` 引用这个新列表, 同时保持 `b` 不变.
- 表达式 `a += [5,6,7,8]` 实际上是使用的是 "extend" 函数, 所以 `a` 和 `b` 仍然指向已被修改的同一列表.

> The out of scope variable/外部作用域变量

```
a = 1
def some_func():
    return a

def another_func():
    a += 1
    return a
```

Output:

```
>>> some_func()
1
>>> another_func()
UnboundLocalError: local variable 'a' referenced before assignment
```

💡 说明:

- 当你在作用域中对变量进行赋值时, 变量会变成该作用域内的局部变量. 因此 `a` 会变成 `another_func` 函数作用域中的局部变量, 但它在函数作用域中并没有被初始化, 所以会引发错误.
- 可以阅读[这个简短却很棒的指南](#), 了解更多关于 Python 中命名空间和作用域的工作原理.
- 想要在 `another_func` 中修改外部作用域变量 `a` 的话, 可以使用 `global` 关键字.

```
def another_func():
    global a
    a += 1
    return a
```

2018/11/30

GitHub - leisurelicht/wtffpyton-cn: wtffpyton的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

Output:

```
>>> another_func()
2
```

> Be careful with chained operations/小心链式操作

```
>>> (False == False) in [False] # 可以理解
False
>>> False == (False in [False]) # 可以理解
False
>>> False == False in [False] # 为毛?
True

>>> True is False == False
False
>>> False is False is False
True

>>> 1 > 0 < 1
True
>>> (1 > 0) < 1
False
>>> 1 > (0 < 1)
False
```



根据 <https://docs.python.org/2/reference/expressions.html#not-in>

形式上, 如果 a, b, c, \dots, y, z 是表达式, 而 op_1, op_2, \dots, op_N 是比较运算符, 那么除了每个表达式最多只出现一次以外 $a op_1 b op_2 c \dots y op_N z$ 就等于 $a op_1 b$ and $b op_2 c$ and ... $y op_N z$.

虽然上面的例子似乎很愚蠢, 但是像 $a == b == c$ 或 $0 <= x <= 100$ 就很棒了.

- $\text{False is False is False}$ 相当于 $(\text{False is False}) \text{ and } (\text{False is False})$
- $\text{True is False == False}$ 相当于 $\text{True is False} \text{ and } \text{False == False}$, 由于语句的第一部分 (True is False) 等于 False , 因此整个表达式的结果为 False .
- $1 > 0 < 1$ 相当于 $1 > 0$ and $0 < 1$, 所以最终结果为 True .
- 表达式 $(1 > 0) < 1$ 相当于 $\text{True} < 1$ 且

```
>>> int(True)
1
>>> True + 1 # 与这个例子无关, 只是好玩
2
```

所以 $1 < 1$ 等于 False

> Name resolution ignoring class scope/忽略类作用域的名称解析

1.

```
x = 5
class SomeClass:
    x = 17
    y = (x for i in range(10))
```

Output:

```
>>> list(SomeClass.y)[0]
5
```

2.

<https://github.com/leisurelicht/wtffpyton-cn>

28/37

2018/11/30 GitHub - leisurelicht/wtffpyton-cn: wtffpyton的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

```
x = 5
class SomeClass:
    x = 17
    y = [x for i in range(10)]
```

Output (Python 2.x):

```
>>> SomeClass.y[0]
17
```

Output (Python 3.x):

```
>>> SomeClass.y[0]
5
```

说明:

- 类定义中嵌套的作用域会忽略类内的名称绑定。
- 生成器表达式有它自己的作用域。
- 从 Python 3.X 开始, 列表推导式也有自己的作用域。

> Needle in a Haystack/大海捞针

1.

```
x, y = (0, 1) if True else None, None
```

Output:

```
>>> x, y # 期望的结果是 (0, 1)
((0, 1), None)
```

几乎每个 Python 程序员都遇到过类似的情况。

2.

```
t = ('one', 'two')
for i in t:
    print(i)

t = ('one')
for i in t:
    print(i)

t = ()
print(t)
```

Output:

```
one
two
o
n
e
tuple()
```

说明:

- 对于 1, 正确的语句是 `x, y = (0, 1) if True else (None, None)`。
- 对于 2, 正确的语句是 `t = ('one',)` 或者 `t = 'one',` (缺少逗号) 否则解释器会认为 `t` 是一个字符串, 并逐个字符对其进行迭代。

2018/11/30

Github - leisurelicht/wtfpython-cn: wtfpython的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

- `()` 是一个特殊的标记, 表示空元组.

Section: The Hidden treasures!/隐藏的宝藏!

This section contains few of the lesser-known interesting things about Python that most beginners like me are unaware of (well, not anymore).

> Okay Python, Can you make me fly?/Python, 可否带我飞? *

好,去吧.

```
import antigravity
```

Output: 嘿.. 这是个超级秘密.

💡 说明:

- `antigravity` 模块是 Python 开发人员发布的少数复活节彩蛋之一.
- `import antigravity` 会打开一个 Python 的[经典 XKCD 漫画](#)页面.
- 不止如此. 这个复活节彩蛋里还有一个复活节彩蛋. 如果你看一下[代码](#), 就会发现还有一个函数实现了 [XKCD's geohashing 算法](#).

> goto , but why?/ goto , 但为什么? *

```
from goto import goto, label
for i in range(9):
    for j in range(9):
        for k in range(9):
            print("I'm trapped, please rescue!")
            if k == 2:
                goto .breakout # 从多重循环中跳出
label .breakout
print("Freedom!")
```

Output (Python 2.3):

```
I'm trapped, please rescue!
I'm trapped, please rescue!
Freedom!
```

💡 说明:

- 2004年4月1日, Python 宣布 加入一个可用的 `goto` 作为愚人节礼物.
- 当前版本的 Python 并没有这个模块.
- 就算可以用, 也请不要使用它. [这里是为什么Python中没有 goto 的原因](#).

> Brace yourself!/做好思想准备 *

如果你不喜欢在Python中使用空格来表示作用域, 你可以导入 C 风格的 {},

```
from __future__ import braces
```

Output:

```
File "some_file.py", line 1
    from __future__ import braces
SyntaxError: not a chance
```

2018/11/30

Github - leisurelicht/wtffpythont-cn: wtffpythont-cn的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

想用大括号? 没门! 觉得不爽, 请去用java.

💡 说明:

- 通常 `__future__` 会提供 Python 未来版本的功能. 然而, 这里的“未来”是一个讽刺.
- 这是一个表达社区对此类问题态度的复活节彩蛋.

> Let's meet Friendly Language Uncle For Life/让生活更友好 *

Output (Python 3.x)

```
>>> from __future__ import barry_as_FLUFL
>>> "Ruby" != "Python" # 这里没什么疑问
File "some_file.py", line 1
    "Ruby" != "Python"
          ^
SyntaxError: invalid syntax

>>> "Ruby" <> "Python"
True
```

这就对了.

💡 说明:

- 相关的 [PEP-401](#) 发布于 2009年4月1日 (所以你现在知道这意味着什么了吧).
- 引用 PEP-401
 - | 意识到 Python 3.0 里的 `!=` 运算符是一个会引起手指疼痛的恐怖错误, FLUFL 将 `<>` 运算符恢复为唯一写法.
- Uncle Barry 在 PEP 中还分享了其他东西: 你可以[在这里](#)获得他们.
- (译: 虽然文档中没写, 但应该是只能在交互解释器中使用.)

> Even Python understands that love is complicated/连Python也知道爱是难言的 *

```
import this
```

等等, this 是什么? this 是爱 ❤

Output:

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
优美胜于丑陋 (Python 以编写优美的代码为目标)

Explicit is better than implicit.
明了胜于晦涩 (优美的代码应当是明了的, 命名规范, 风格相似)

Simple is better than complex.
简洁胜于复杂 (优美的代码应当是简洁的, 不要有复杂的内部实现)

Complex is better than complicated.
复杂胜于凌乱 (如果复杂不可避免, 那代码间也不能有难懂的关系, 要保持接口简洁)

Flat is better than nested.
扁平胜于嵌套 (优美的代码应当是扁平的, 不能有太多的嵌套)

Sparse is better than dense.
间隔胜于紧凑 (优美的代码有适当的间隔, 不要奢望一行代码解决问题)

Readability counts.
可读性很重要 (优美的代码一定是可读的)

Special cases aren't special enough to break the rules.
没有特例特殊到需要违背这些规则 (这些规则至高无上)

Although practicality beats purity.
尽管我们更倾向于实用性

Errors should never pass silently.
不要安静的包容所有错误

Unless explicitly silenced.
除非你确定需要这样做 (精准地捕获异常, 不写 except:pass 风格的代码)

In the face of ambiguity, refuse the temptation to guess.
```

2018/11/30

GitHub - leisurelicht/wtffpython-cn: wtffpython的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

拒绝诱惑你去猜测的暧昧事物
 There should be one-- and preferably only one --obvious way to do it.
 而是尽量找一种, 最好是唯一一种明显的解决方案 (如果不确定, 就用穷举法)
 Although that way may not be obvious at first unless you're Dutch.
 虽然这并不容易, 因为你不是 Python 之父 (这里的 Dutch 是指 Guido)
 Now is better than never.
 现在行动好过永远不行动
 Although never is often better than *right* now.
 尽管不行动要好过鲁莽行动
 If the implementation is hard to explain, it's a bad idea.
 如果你无法向人描述你的方案, 那肯定不是一个好方案
 If the implementation is easy to explain, it may be a good idea.
 如果你能轻松向人描述你的方案, 那也许会是一个好方案 (方案测评标准)
 Namespaces are one honking great idea -- let's do more of those!
 命名空间是一种绝妙的理念, 我们应当多加利用 (倡导与号召)

这是 Python 之禅!

```
>>> love = this
>>> this is love
True
>>> love is True
False
>>> love is False
False
>>> love is not True or False
True
>>> love is not True or False; love is love # 爱是难言的
True
```

说明:

- this 模块是关于 Python 之禅的复活节彩蛋 (PEP 20).
- 如果你认为这已经够有趣的了, 可以看看 [this.py](#) 的实现. 有趣的是, Python 之禅的实现代码违反了他自己 (这可能是唯一会发生这种情况的地方).
-

至于 love is not True or False; love is love , 意外却又不言而喻.

> Yes, it exists!/是的, 它存在!

循环的 else . 一个典型的例子:

```
def does_exists_num(l, to_find):
    for num in l:
        if num == to_find:
            print("Exists!")
            break
    else:
        print("Does not exist")
```

Output:

```
>>> some_list = [1, 2, 3, 4, 5]
>>> does_exists_num(some_list, 4)
Exists!
>>> does_exists_num(some_list, -1)
Does not exist
```

异常的 else . 例.

```
try:
    pass
except:
    print("Exception occurred!!!")
else:
    print("Try block executed successfully...")
```

2018/11/30

GitHub - leisurelicht/wtffpythont-cn: wtffpythont的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

Output:

```
Try block executed successfully...
```

💡 说明:

- 循环后的 else 子句只会在循环没有触发 break 语句, 正常结束的情况下才会执行.
- try 之后的 else 子句也被称为 "完成子句", 因为在 try 语句中到达 else 子句意味着try块实际上已成功完成.

> Infinity/无限 *

英文拼写是有意的, 请不要为此提交补丁.(译: 我并不理解这里故意拼错的意义)

Output (Python 3.x):

```
>>> infinity = float('infinity')
>>> hash(infinity)
314159
>>> hash(float('-inf'))
-314159
```

💡 说明:

- infinity 的哈希值是 $10^5 \times \pi$.
- 有意思的是, float('-inf') 的哈希值在 Python 3 中是 "- $10^5 \times \pi$ ", 而在 Python 2 中是 "- $10^5 \times e$ ".

> Mangling time!/修饰时间! *

```
class Yo(object):
    def __init__(self):
        self.__honey = True
        self.bitch = True
```

Output:

```
>>> Yo().__bitch
True
>>> Yo().__honey
AttributeError: 'Yo' object has no attribute '__honey'
>>> Yo().__Yo__honey
True
```

为什么 Yo().__Yo__honey 能运行? 只有印度人理解.(译: 这又是什么梗?)

💡 说明:

- **名字修饰** 用于避免不同命名空间之间名称冲突.
- 在 Python 中, 解释器会通过给类中以 __ (双下划线)开头且结尾最多只有一个下划线的类成员名称加上 _NameOfTheClass 来修饰(mangles)名称.
- 所以 要访问 __honey 对象,我们需要加上 __Yo 以防止与其他类中定义的相同名称的属性发生冲突.

Section: Miscellaneous/杂项

> += is faster/更快的 +=

2018/11/30

Github - leisurelicht/wtfpython-cn: wtfpython的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

```
# 用 "+" 连接三个字符串:
>>> timeit.timeit("s1 = s1 + s2 + s3", setup="s1 = ' ' * 100000; s2 = ' ' * 100000; s3 = ' ' * 100000", number=1)
0.25748300552368164
# 用 "+=" 连接三个字符串:
>>> timeit.timeit("s1 += s2 + s3", setup="s1 = ' ' * 100000; s2 = ' ' * 100000; s3 = ' ' * 100000", number=100)
0.012188911437988281
```

说明:

- 连接两个以上的字符串时 `+=` 比 `+` 更快, 因为在计算过程中第一个字符串(例如, `s1 += s2 + s3` 中的 `s1`)不会被销毁。(译:就是 `+=` 执行的是追加操作, 少了一个销毁新建的动作。)

> Let's make a giant string!/来做个巨大的字符串吧!

```
def add_string_with_plus(iters):
    s = ""
    for i in range(iters):
        s += "xyz"
    assert len(s) == 3*iters

def add_bytes_with_plus(iters):
    s = b""
    for i in range(iters):
        s += b"xyz"
    assert len(s) == 3*iters

def add_string_with_format(iters):
    fs = "{}".format(iters)
    s = fs.format(*(["xyz"]*iters))
    assert len(s) == 3*iters

def add_string_with_join(iters):
    l = []
    for i in range(iters):
        l.append("xyz")
    s = "".join(l)
    assert len(s) == 3*iters

def convert_list_to_string(l, iters):
    s = "".join(l)
    assert len(s) == 3*iters
```

Output:

```
>>> timeit(add_string_with_plus(1000))
1000 loops, best of 3: 972 µs per loop
>>> timeit(add_bytes_with_plus(1000))
1000 loops, best of 3: 815 µs per loop
>>> timeit(add_string_with_format(1000))
1000 loops, best of 3: 508 µs per loop
>>> timeit(add_string_with_join(1000))
1000 loops, best of 3: 878 µs per loop
>>> l = ["xyz"]*1000
>>> timeit(convert_list_to_string(l, 1000))
1000 loops, best of 3: 80 µs per loop
```

让我们将迭代次数增加10倍.

```
>>> timeit(add_string_with_plus(10000)) # 执行时间线性增加
100 loops, best of 3: 9.75 ms per loop
>>> timeit(add_bytes_with_plus(10000)) # 二次增加
1000 loops, best of 3: 974 ms per loop
>>> timeit(add_string_with_format(10000)) # 线性增加
100 loops, best of 3: 5.25 ms per loop
>>> timeit(add_string_with_join(10000)) # 线性增加
100 loops, best of 3: 9.85 ms per loop
>>> l = ["xyz"]*10000
```

2018/11/30

GitHub - leisurelicht/wtffpythont-cn: wtffpythont-cn的中文翻译/施工结束/ 能力有限, 欢迎帮我改进翻译

```
>>> timeit(convert_list_to_string(l, 100000)) # 线性增加
1000 loops, best of 3: 723 µs per loop
```

Q 说明:

- 你可以在这获得更多 `timeit` 的相关信息. 它通常用于衡量代码片段的执行时间.
- 不要用 `+` 去生成过长的字符串. 在 Python 中, `str` 是不可变的, 所以在每次连接中你都要把左右两个字符串复制到新的字符串中. 如果你连接四个长度为 10 的字符串, 你需要拷贝 $(10+10) + ((10+10)+10) + (((10+10)+10)+10) = 90$ 个字符而不是 40 个字符. 随着字符串的数量和大小的增加, 情况会变得越发的糟糕 (就像 `add_bytes_with_plus` 函数的执行时间一样)
- 因此, 更建议使用 `.format` 或 `%` 语法 (但是, 对于短字符串, 它们比 `+` 稍慢一点).
- 又或者, 如果你所需的内容已经以可迭代对象的形式提供了, 使用 `''.join(可迭代对象)` 要快多了.
- `add_string_with_plus` 的执行时间没有像 `add_bytes_with_plus` 一样出现二次增加是因为解释器会如同上一个例子所讨论的一样优化 `+=`. 用 `s = s + "x" + "y" + "z"` 替代 `s += "xyz"` 的话, 执行时间就会二次增加了.

```
def add_string_with_plus(iters):
    s = ""
    for i in range(iters):
        s = s + "x" + "y" + "z"
    assert len(s) == 3*iters

>>> timeit(add_string_with_plus(10000))
100 loops, best of 3: 9.87 ms per loop
>>> timeit(add_string_with_plus(100000)) # 执行时间二次增加
1 loops, best of 3: 1.09 s per loop
```

> Explicit typecast of strings/字符串的显式类型转换

```
a = float('inf')
b = float('nan')
c = float('-iNf') # 这些字符串不区分大小写
d = float('nan')
```

Output:

```
>>> a
inf
>>> b
nan
>>> c
-inf
>>> float('some_other_string')
ValueError: could not convert string to float: some_other_string
>>> a == -c #inf==inf
True
>>> None == None # None==None
True
>>> b == d #但是 nan!=nan
False
>>> 50/a
0.0
>>> a/a
nan
>>> 23 + b
nan
```

Q 说明:

`'inf'` 和 `'nan'` 是特殊的字符串(不区分大小写), 当显示转换成 `float` 型时, 它们分别用于表示数学意义上的 "无穷大" 和 "非数字".

> Minor Ones/小知识点

- `join()` 是一个字符串操作而不是列表操作.(第一次接触会觉得有点违反直觉)

2018/11/30

Github - leisurelicht/wtfpython-cn: wtfpython的中文翻译/施工结束/能力有限, 欢迎帮我改进翻译

Q 说明: 如果 `join()` 是字符串方法 那么它就可以处理任何可迭代的对象(列表, 元组, 迭代器). 如果它是列表方法, 则必须在每种类型中单独实现. 另外, 在 `list` 对象的通用API中实现一个专用于字符串的方法没有太大的意义.

- 看着奇怪但能正确运行的语句:

- `[] = ()` 语句在语义上是正确的 (解包一个空的 tuple 并赋值给 list)
- `'a'[0][0][0][0]` 在语义上也是正确的, 因为在 Python 中字符串同时也是序列(可迭代对象支持使用整数索引访问元素).
- `3 --0-- 5 == 8` 和 `--5 == 5` 在语义上都是正确的, 且结果等于 `True`.(译:这个我也不懂)

- 鉴于 `a` 是一个数组, `++a` 和 `--a` 都是有效的 Python 语句, 但其效果与 C, C++ 或 Java 等不一样.

```
>>> a = 5
>>> a
5
>>> ++a
5
>>> --a
5
```

Q 说明:

- python 里没有 `++` 操作符. 这其实是两个 `+` 操作符.
- `++a` 被解析为 `+(+a)` 最后等于 `a`. `--a` 同理.
- 这个 StackOverflow 回答 讨论了为什么 Python 中缺少增量和减量运算符.

- Python 使用 2 个字节存储函数中的本地变量. 理论上, 这意味着函数中只能定义 65536 个变量. 但是, Python 内置了一个方便的解决方案, 可用于存储超过 2^{16} 个变量名. 下面的代码演示了当定义了超过 65536 个局部变量时堆栈中发生的情况(警告: 这段代码会打印大约 2^{18} 行文本, 请做好准备!):

```
import dis
exec("""
def f():
    """ + """
    """.join(["X"+str(x)+"=" + str(x) for x in range(65539)]))

f()
print(dis.dis(f))
```

- 你的 Python 代码 并不会多线程同时运行(是的, 你没听错!). 虽然你觉得会产生多个线程并让它们同时执行你的代码, 但是, 由于 全局解释锁 的存在, 你所做的只是让你的线程依次在同一个核心上执行. Python 多线程适用于IO密集型的任务, 但如果想要 并行 处理CPU密集型的任务, 你应该会想使用 `multiprocessing` 模块.
- 列表切片超出索引边界而不引发任何错误

```
>>> some_list = [1, 2, 3, 4, 5]
>>> some_list[111:]
[]
```

- `int('۱۲۳۴۵۶۷۸۹')` 在 Python 3 中会返回 `123456789`. 在 Python 中, 十进制字符包括数字字符, 以及可用于形成十进制数字的所有字符, 例如: U+0660, ARABIC-INDIC DIGIT ZERO. 这有一个关于此的 [有趣故事](#).
- `'abc'.count('')` == 4. 这有一个 `count` 方法的相近实现, 能更好的说明问题

```
def count(s, sub):
    result = 0
    for i in range(len(s) + 1 - len(sub)):
        result += (s[i:i + len(sub)]) == sub
    return result
```

这个行为是由于空子串(`''`)与原始字符串中长度为0的切片相匹配导致的.

最后，再次给出原文链接以及中文版：

<https://github.com/satwikkansal/wtfpython>

<https://github.com/leisurelicht/wtfpython-cn#section-strain-your-brain>

【今日机器学习概念】
Have a Great Definition



一种模型选择策略。该策略中，对最每组学习算法和/或者超参数的组合训练一个模型。然后选择其中最优模型。

ChrisAlbon



十万+条数据分析
千余调研问卷
八位专家深度访谈
三个月调研

顶级数据团队 2018全景报告

A ROADMAP TO A TOP DATA-DRIVEN ENTERPRISE

扫码领取报告
精华版





普及数据思维 传播数据文化

We talk data We deliver knowledge

关于转载授权大数据文摘作品，
请在大数据文摘后台留言“机构名称+文章标题+转载”，
申请过授权的不必再次申请，只要按约定转载即可。



长按二维码 · 关注大数据文摘

「大数据文摘团队」等你来
线上公开分享嘉宾 · 实习／全职编辑记者
感兴趣请联系zz@bigdatadigest.cn