设计和历史常见问题

目录

为什么 Python 使用缩进来分组语句?

Guido van Rossum 认为使用缩进进行分组非常优雅，并且大大提高了普通 Python 程序的清晰度。大多数人在一段时间后就会喜欢上这个特性。

由于没有开始/结束括号，因此解析器感知的分组与人类读者之间不会存在分歧。 偶尔 C 程序员会遇到像这样的代码片段:

```
if (x <= y)
        x++;
        y--;
z++;
```

如果条件为真，则只执行 `x++` 语句，但缩进会使你认为情况并非如此。 即使是经验丰富的 C 程序员有时也会长久地盯着它发呆，不明白为什么在 `x > y` 时 `y` 也会减少。

因为没有开始/结束花括号，所以 Python 更不容易发生代码风格冲突。 在 C 中有许多不同的放置花括号的方式。 在习惯了阅读和编写某种特定风格的代码之后，当阅读（或被要求编写）另一种风格的代码时通常都会令人感觉有点不舒服）。

许多编码风格将开始/结束括号单独放在一行上。这使得程序相当长，浪费了宝贵的屏幕空间，使得更难以对程序进行全面的了解。 理想情况下，函数应该适合一个屏幕（例如，20--30行）。 20 行 Python 可以完成比 20 行 C 更多的工作。 这不仅仅是由于没有开始/结束括号 -- 无需声明以及高层级的数据类型也是其中的原因 -- 但基于缩进的语法肯定有帮助。

为什么简单的算术运算得到奇怪的结果?

请看下一个问题。

为什么浮点计算不准确?

用户经常对这样的结果感到惊讶:

```
>>> 1.2 - 1.0
0.19999999999999996
```

并且认为这是 Python 中的一个 bug。其实不是这样。这与 Python 关系不大，而与底层平台如何处理浮点数字关系更大。

CPython 中的 `float` 类型使用C语言的 `double` 类型进行存储。 `float` 对象的值是以固定的精度（通常为 53 位）存储的二进制浮点数，由于 Python 使用 C 操作，而后者依赖于处理器中的硬件实现来执行浮点运算。 这意味着就浮点运算而言，Python 的行为类似于许多流行的语言，包括 C 和 Java。

许多可以轻松地用十进制表示的数字不能用二进制浮点表示。 例如，在输入以下语句后:

```
>>> x = 1.2
```

为 `x` 存储的值是与十进制的值 `1.2` (非常接近) 的近似值，但不完全等于它。 在典型的机器上，实际存储的值是:

```
1.0011001100110011001100110011001100110011001100110011 (二进制)
```

它对应于十进制数值:

```
1.1999999999999999955591079014993738383054733276367187 5 (十进制)
```

典型的 53 位精度为 Python 浮点数提供了 15-16 位小数的精度。

要获得更完整的解释，请参阅 Python 教程中的 浮点算术 一章。

为什么Python字符串是不可变的?

有几个优点。

一个是性能：知道字符串是不可变的，意味着我们可以在创建时为它分配空间，并且存储需求是固定不变的。这也是元组和列表之间区别的原因之一。

另一个优点是，Python 中的字符串被视为与数字一样"基本"。 任何动作都不会将值 8 更改为其他值，在 Python 中，任何动作都不会将字符串 "8" 更改为其他值。

为什么必须在方法定义和调用中显式使用"self"?

这个想法借鉴了 Modula-3 语言。 出于多种原因它被证明是非常有用的。

首先，更明显的显示出，使用的是方法或实例属性而不是局部变量。 阅读 `self.x` 或 `self.meth()` 可以清楚地表明，即使您不知道类的定义，也会使用实例变量或方法。在 C++ 中，可以通过缺少局部变量声明来判断（假设全局变量很少见或容易识别） —— 但是在 Python 中没有局部变量声明，所以必须查找类定义才能确定。 一些 C++ 和 Java 编码标准要求实例属性具有 `m_` 前缀，因此这种显式性在这些语言中仍然有用。

其次，这意味着当要显式引用或从特定类调用该方法时无须特殊语法。 在 C++ 中，如果你想要使用在派生类中被重写的基类方法，你必须使用 `::` 运算符 -- 在 Python 中你可以写成 `baseclass.methodname(self, <argument list>)`。 这特别适用于 `__init__()` 方法，并且也适用于派生类方法想要扩展同名的基类方法因而必须以某种方式调用基类方法的情况。

最后，它解决了实例变量赋值的语法问题：由于 Python 中的局部变量（根据定义！）是指在函数体内被赋值的变量（并且它没有被明确声明为全局变量），因此必须存在某种方式告诉解释器，某次赋值是为了分配一个实例变量而不是一个局部变量，它最好是通过语法实现的（出于效率原因）。 C++ 通过声明来做到这一点，但是 Python 没有声明，仅仅为了这个目的而引入它们会很可惜。 使用显式的 `self.var` 很好地解决了这个问题。 类似地，对于使用实例变量，必须编写 `self.var` 意味着对方法内部的非限定名称的引用不必搜索实例的目录。 换句话说，局部变量和实例变量存在于两个不同的命名空间中，您需要告诉 Python 使用哪个命名空间。

为什么不能在表达式中赋值?

自 Python 3.8 开始，你能做到的!

赋值表达式使用海象运算符 `:=` 在表达式中为变量赋值:

```
while chunk := fp.read(200):
   print(chunk)
```

请参阅 **PEP 572** 了解详情。

为什么Python对某些功能（例如list.index()）使用方法来实现，而其他功能（例如len(List)）使用函数实现?

正如Guido所说：

> (a) 对于某些操作，前缀表示法比后缀更容易阅读 -- 前缀（和中缀！）运算在数学中有着悠久的传统，就像在视觉上帮助数学家思考问题的记法。比较一下我们将 x*(a+b) 这样的公式改写为 x*a+x*b 的容易程度，以及使用原始OO符号做相同事情的笨拙程度。
>
> (b) 当读到写有len(X)的代码时，就知道它要求的是某件东西的长度。这告诉我们两件事：结果是一个整数，参数是某种容器。相反，当阅读x.len()时，必须已经知道x是某种实现接口的容器，或者是从具有标准len()的类继承的容器。当没有实现映射的类有get()或key()方法，或者不是文件的类有write()方法时，我们偶尔会感到困惑。
>
> —https://mail.python.org/pipermail/python-3000/2006-November/004643.html

为什么 join() 是一个字符串方法而不是列表或元组方法?

从 Python 1.6 开始，字符串变得更像其他标准类型，当添加方法时，这些方法提供的功能与始终使用 String 模块的函数时提供的功能相同。这些新方法中的大多数已被广泛接受，但似乎让一些程序员感到不舒服的一种方法是：

```
", ".join(['1', '2', '4', '8', '16'])
```

结果如下:

```
"1, 2, 4, 8, 16"
```

反对这种用法有两个常见的论点。

第一条是这样的：“使用字符串文本(String Constant)的方法看起来真的很难看”，答案是也许吧，但是字符串文本只是一个固定值。如果在绑定到字符串的名称上允许使用这些方法，则没有逻辑上的理由使其在文字上不可用。

第二个异议通常是这样的：“我实际上是在告诉序列使用字符串常量将其成员连接在一起”。遗憾的是并非如此。出于某种原因，把 `split()` 作为一个字符串方法似乎要容易得多，因为在这种情况下，很容易看到:

```
"1, 2, 4, 8, 16".split(", ")
```

是对字符串文本的指令，用于返回由给定分隔符分隔的子字符串（或在默认情况下，返回任意空格）。

join() 是字符串方法，因为在使用该方法时，您告诉分隔符字符串去迭代一个字符串序列，并在相邻元素之间插入自身。此方法的参数可以是任何遵循序列规则的对象，包括您自己定义的任何新的类。对于字节和字节数组对象也有类似的方法。

异常有多快?

如果没有引发异常则 try/except 代码块是非常高效的。 实际上捕获异常是很消耗性能的。 在 2.0 之前的 Python 版本中通常使用这例程:

```
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

只有当你期望dict在任何时候都有key时，这才有意义。如果不是这样的话，你就是应该这样编码：

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

对于这种特定的情况，您还可以使用 `value = dict.setdefault(key, getvalue(key))`，但前提是调用 `getvalue()` 足够便宜，因为在所有情况下都会对其进行评估。

为什么Python中没有switch或case语句?

总的来说，结构化分支语句会在一个表达式具有特定值或值的集合时执行某个代码块。 从 Python 3.10 开始可以简单地通过 `match ... case` 语句来匹配字面值，或特定命名空间中的常量。 一种较旧的替代方案是通过一系列的 `if... elif... elif... else`。

对于需要从大量可能性中进行选择的情况，可以创建一个字典，将case 值映射到要调用的函数。例如：

```
functions = {'a': function_1,
             'b': function_2,
             'c': self.method_1}

func = functions[value]
func()
```

对于对象调用方法，可以通过使用 getattr() 内置检索具有特定名称的方法来进一步简化：

```
class MyVisitor:
    def visit_a(self):
        ...

    def dispatch(self, value):
        method_name = 'visit_' + str(value)
        method = getattr(self, method_name)
        method()
```

建议对方法名使用前缀，例如本例中的 `visit_` 。如果没有这样的前缀，如果值来自不受信任的源，攻击者将能够调用对象上的任何方法。

模仿带有穿透方式的分支，就像 C 的 switch-case-default 那样是有可能的，但更为困难，也无甚必要。

难道不能在解释器中模拟线程，而非得依赖特定于操作系统的线程实现吗?

答案1： 不幸的是，解释器为每个Python堆栈帧推送至少一个C堆栈帧。此外，扩展可以随时回调Python。因此，一个完整的线程实现需要对C的线程支持。

答案2： 幸运的是， Stackless Python 有一个完全重新设计的解释器循环，可以避免C堆栈。

为什么lambda表达式不能包含语句?

Python 的 lambda 表达式不能包含语句，因为Python的语法框架不能处理嵌套在表达式内部的语句。然而，在 Python 中，这并不是一个严重的问题。 与其他语言中添加功能的 lambda 形式不同，Python 的 lambda 只是一种速记符号，如果您懒得定义函数的话。

函数已经是 Python 中的第一等对象，并且可以在局部作用域中声明。 因此使用 lambda 而非局部定义函数的唯一优点是你不需要为函数指定名称 -- 但那只是一个被赋值为函数对象（它的类型与 lambda 表达式所产生的对象完全相同）的局部变量!

可以将Python编译为机器代码，C或其他语言吗?

Cython 会将带有可选标注的修改版 Python 编译为 C 扩展。 Nuitka 是一个 Python 转 C++ 代码的新兴编译器，其目标是支持完整的 Python 语言。

Python如何管理内存?

Python 内存管理的细节取决于实现。 Python 的标准实现 CPython 使用引用计数来检测不可访问的对象，并使用另一种机制来收集引用循环，定期执行循环检测算法来查找不可访问的循环并删除所涉及的对象。 `gc` 模块提供了执行垃圾回收、获取调试统计信息和优化收集器参数的函数。

不过，其他实现 (如 Jython 或 PyPy)，可能会依赖不同的机制，如完全的垃圾回收器。 如果你的 Python 代码依赖于引用计数实现的行为则这种差异可能会导致某些微妙的移植问题。

在一些Python实现中，以下代码（在CPython中工作的很好）可能会耗尽文件描述符:

```
for file in very_long_list_of_files:
    f = open(file)
    c = f.read(1)
```

实际上，使用 CPython 的引用计数或器方案，每次对 `f` 的新赋值都会关闭之前的文件。 然而，对于传统的 GC，这些文件对象将只能以不同的并且可能很长的间隔被收集（和关闭）。

如果要编写可用于任何python实现的代码，则应显式关闭该文件或使用 `with` 语句；无论内存管理方案如何，这都有效：

```
for file in very_long_list_of_files:
    with open(file) as f:
        c = f.read(1)
```

为什么CPython不使用更传统的垃圾回收方案？

首先，这不是C标准特性，因此不能移植。(是的，我们知道Boehm GC库。它包含了 *大多数* 常见平台（但不是所有平台）的汇编代码，尽管它基本上是透明的，但也不是完全透明的; 要让Python使用它，需要使用补丁。）

当 Python 嵌入到其他应用程序中时传统的 GC 也会成为一个问题。 在独立的 Python 中可以用 GC 库提供的版本来替换标准的 `malloc()` 和 `free()`，而嵌入 Python 的应用程序可能想要 *自行* 替代 `malloc()` 和 `free()`，并不想要 Python 的版本。 现在，CPython 可以适用于任何正确实现了 `malloc()` 和 `free()` 的版本。

CPython退出时为什么不释放所有内存？

当Python退出时，从全局命名空间或Python模块引用的对象并不总是被释放。 如果存在循环引用，则可能发生这种情况 C库分配的某些内存也是不可能释放的（例如像Purify这样的工具会抱怨这些内容）。 但是，Python在退出时清理内存并尝试销毁每个对象。

如果要强制 Python 在释放时删除某些内容，请使用 `atexit` 模块运行一个函数，强制删除这些内容。

为什么有单独的元组和列表数据类型？

列表和元组虽然在许多方式都很相似，但它们的使用方式有本质上的不同。 元组可被当作是类似于 Pascal `records` 或 C `structs`；它们是由可能具有不同类型但可作为一个分组进行操作的相关数据组成的小多项集。 例如，一个笛卡尔坐标可适当地用由两个或三个数字组成的元组来表示。

另一方面，列表更像其他语言中的数组。 它们倾向于保存可变数量的全都具有相同类型并将被逐个操作的对象。 例如，`os.listdir('.')` 返回一个代表当前目录中文件的字符串列表。 当你向该目录添加一两个文件时在此输出上执行操作的函数通常不会中断。

元组是不可变的，这意味着一旦创建了元组，就不能用新值替换它的任何元素。列表是可变的，这意味着您始终可以更改列表的元素。只有不变元素可以用作字典的key，因此只能将元组和非列表用作key。

列表是如何在CPython中实现的？

CPython的列表实际上是可变长度的数组，而不是lisp风格的链表。该实现使用对其他对象的引用的连续数组，并在列表头结构中保留指向该数组和数组长度的指针。

这使得索引列表 `a[i]` 的操作成本与列表的大小或索引的值无关。

当添加或插入项时，将调整引用数组的大小。并采用了一些巧妙的方法来提高重复添加项的性能; 当数组必须增长时，会分配一些额外的空间，以便在接下来的几次中不需要实际调整大小。

字典是如何在CPython中实现的?

CPython的字典实现为可调整大小的哈希表。与B-树相比，这在大多数情况下为查找（目前最常见的操作）提供了更好的性能，并且实现更简单。

字典的工作方式是使用 `hash()` 内置函数计算字典中存储的每个键的哈希码。 哈希码会根据键和基于进程的种子值而大幅改变；例如，`'Python'` 的哈希码可能为 `-539294296` 而 `'python'` 这个只相差一丁点的字符串的哈希码却可能为 `1142331976`。 随后哈希码将被用来计算在一个内部数组中相应值的存储位置。 假设你存储的键都具有不同的哈希值，这意味着字典会耗费恒定的时间 -- 即大 O 表示法的 *O*(1) -- 要检索一个键。

为什么字典key必须是不可变的?

字典的哈希表实现使用从键值计算的哈希值来查找键。如果键是可变对象，则其值可能会发生变化，因此其哈希值也会发生变化。但是，由于无论谁更改键对象都无法判断它是否被用作字典键值，因此无法在字典中修改条目。然后，当你尝试在字典中查找相同的对象时，将无法找到它，因为其哈希值不同。如果你尝试查找旧值，也不会找到它，因为在该哈希表中找到的对象的值会有所不同。

如果你想要一个用列表索引的字典，只需先将列表转换为元组；用函数 `tuple(L)` 创建一个元组，其条目与列表 `L` 相同。 元组是不可变的，因此可以用作字典键。

已经提出的一些不可接受的解决方案：

- 哈希按其地址（对象ID）列出。这不起作用，因为如果你构造一个具有相同值的新列表，它将无法找到；例如:

  ```
  mydict = {[1, 2]: '12'}
  print(mydict[[1, 2]])
  ```

  会引发一个 `KeyError` 异常，因为第二行中使用的 `[1, 2]` 的 id 与第一行中的 id 不同。换句话说，应该使用 `==` 来比较字典键，而不是使用 `is` 。

- 使用列表作为键时进行复制。这没有用的，因为作为可变对象的列表可以包含对自身的引用，然后复制代码将进入无限循环。

- 允许列表作为键，但告诉用户不要修改它们。当你意外忘记或修改列表时，这将产生程序中的一类难以跟踪的错误。它还使一个重要的字典不变量无效： `d.keys()` 中的每个值都可用作字典的键。

- 将列表用作字典键后，应标记为其只读。问题是，它不仅仅是可以改变其值的顶级对象；你可以使用包含列表作为键的元组。将任何内容作为键关联到字典中都需要将从那里可到达的所有对象

标记为只读 —— 并且自引用对象可能会导致无限循环。

如果你有需要，以下技巧可以绕过这个问题，但使用它必须自担风险：你可以将一个可变结构体包装在一个同时具有 `__eq__()` 和 `__hash__()` 方法的类实例中。 然后你必须确保存放在字典（或其他基于哈希值的结构体）中的所有此类包装器对象的哈希值在该字典（或其他结构体）中保持固定。

```
class ListWrapper:
    def __init__(self, the_list):
        self.the_list = the_list

    def __eq__(self, other):
        return self.the_list == other.the_list

    def __hash__(self):
        l = self.the_list
        result = 98767 - len(l)*555
        for i, el in enumerate(l):
            try:
                result = result + (hash(el) % 9999999) * 1001 + i
            except Exception:
                result = (result % 7777777) + i * 333
        return result
```

注意，哈希计算由于列表的某些成员可能不可用以及算术溢出的可能性而变得复杂。

此外，必须始终如此，如果 `o1 == o2` （即 `o1.__eq__(o2) is True` ）则 `hash(o1) == hash(o2)` （即 `o1.__hash__() == o2.__hash__()` ），无论对象是否在字典中。 如果你不能满足这些限制，字典和其他基于 hash 的结构将会出错。

对于 `ListWrapper` 的情况，只要包装器位于字典中那么被包装的列表就不能更改以避免发生意外。除非你准备好认真考虑相关要求以及未能正确满足这些要求的后果否则请不要这样做。 你已经收到警告了。

为什么 list.sort() 没有返回排序列表?

在性能很重要的情况下，仅仅为了排序而复制一份列表将是一种浪费。因此， `list.sort()` 对列表进行了适当的排序。为了提醒您这一事实，它不会返回已排序的列表。这样，当您需要排序的副本，但也需要保留未排序的版本时，就不会意外地覆盖列表。

如果要返回新列表，请使用内置 `sorted()` 函数。此函数从提供的可迭代列表中创建新列表，对其进行排序并返回。例如，下面是如何迭代遍历字典并按keys排序:

```
for key in sorted(mydict):
    ...  # 对 mydict[key] 做点什么
```

如何在Python中指定和实施接口规范?

由C++和Java等语言提供的模块接口规范描述了模块的方法和函数的原型。许多人认为接口规范的编译时强制执行有助于构建大型程序。

Python 2.6添加了一个 `abc` 模块，允许定义抽象基类 (ABCs)。然后可以使用 `isinstance()` 和 `issubclass()` 来检查实例或类是否实现了特定的ABC。 `collections.abc` 模块定义了一组有用的 ABCs 例如 `Iterable` ， `Container` , 和 `MutableMapping`

对于 Python，接口规范的许多好处可以通过组件的适当测试规程来获得。

一个好的模块测试套件既可以提供回归测试，也可以作为模块接口规范和一组示例。许多Python模块可以作为脚本运行，以提供简单的"自我测试"。即使是使用复杂外部接口的模块，也常常可以使用外部接口的简单"桩代码（stub）"模拟进行隔离测试。可以使用 `doctest` 和 `unittest` 模块或第三方测试框架来构造详尽的测试套件，以运行模块中的每一行代码。

适当的测试规程能像完善的接口规范一样帮助在 Python 构建大型的复杂应用程序。 事实上，它能做得更好因为接口规范无法测试程序的某些属性。 例如，`list.append()` 方法被期望向某个内部列表的末尾添加新元素；接口规范无法测试你的 `list.append()` 实现是否真的能正确执行该操作，但在测试套件中检查该属性却是很容易的。

编写测试套件非常有用，并且你可能希望将你的代码设计为易于测试。 一种日益流行的技术是面向测试的开发，它要求在编写任何实际代码之前首先编写测试套件的各个部分。 当然 Python 也允许你采用更粗率的方式，不必编写任何测试用例。

为什么没有goto?

在 1970 年代人们了解到不受限制的 goto 可能导致混乱得像"意大利面"那样难以理解和修改的代码。 在高级语言中，它也是不必要的，只需能够执行分支（在 Python 中是使用 `if` 语句和 `or`, `and` 以及 `if`/`else` 表达式）和循环（使用 `while` 和 `for` 语句，还可能包含 `continue` 和 `break` 语句）就足够了。

人们还可以使用异常来提供甚至能跨函数调用的"结构化 goto"。 许多人觉得异常可以方便地模拟 C, Fortran 和其他语言中所有合理使用的 `go` 或 `goto` 结构:

```
class label(Exception): pass  # 声明label

try:
    ...
    if condition: raise label()  # 跳转到 label
    ...
except label:  # 跳转到哪里
    pass
...
```

这并不允许跳到一个循环的中间，但这通常被视为是对 `goto` 的滥用。 应当谨慎使用。

为什么原始字符串（r-strings）不能以反斜杠结尾?

更准确地说，它们不能以奇数个反斜杠结束：结尾处的不成对反斜杠会转义结束引号字符，留下未结束的字符串。

原始字符串的设计是为了方便想要执行自己的反斜杠转义处理的处理器(主要是正则表达式引擎)创建输入。此类处理器将不匹配的尾随反斜杠视为错误，因此原始字符串不允许这样做。反过来，允许

通过使用引号字符转义反斜杠转义字符串。当r-string用于它们的预期目的时，这些规则工作的很好。

如果您正在尝试构建Windows路径名，请注意所有Windows系统调用都使用正斜杠:

```
f = open("/mydir/file.txt")  # 效果很好!
```

如果您正在尝试为DOS命令构建路径名，请尝试以下示例

```
dir = r"\this\is\my\dos\dir" "\\"
dir = r"\this\is\my\dos\dir\ "[:-1]
dir = "\\this\\is\\my\\dos\\dir\\"
```

为什么Python没有属性赋值的“with”语句？

Python 有一种 `with` 语句能将一个代码块的执行包装起来，在进入和退出该代码块时调用特定的代码。 某些语言具有类似这样的结构:

```
with obj:
    a = 1               # 相当于 to obj.a = 1
    total = total + 1   # obj.total = obj.total + 1
```

在Python中，这样的结构是不明确的。

其他语言，如ObjectPascal、Delphi和C++ 使用静态类型，因此可以毫不含糊地知道分配给什么成员。这是静态类型的要点 -- 编译器 *总是* 在编译时知道每个变量的作用域。

Python使用动态类型。事先不可能知道在运行时引用哪个属性。可以动态地在对象中添加或删除成员属性。这使得无法通过简单的阅读就知道引用的是什么属性：局部属性、全局属性还是成员属性?

例如，采用以下不完整的代码段:

```
def foo(a):
    with a:
        print(x)
```

该代码段假设 `a` 必须有一个名为 `x` 的成员属性。 然而，Python 中没有什么能告诉解释器这一点。举例来说，如果 `a` 是一个整数那么会发生什么？ 如果有一个名为 `x` 的全局变量，它是否会在 `with` 代码块内被使用？ 如你所见，Python 的动态特性使得这样的选择更为困难。

然而，`with` 及类似语言特性的主要好处（减少代码量）在 Python 中可以通过赋值轻松地实现。 而不是使用:

```
function(args).mydict[index][index].a = 21
function(args).mydict[index][index].b = 42
function(args).mydict[index][index].c = 63
```

写成这样:

```
ref = function(args).mydict[index][index]
ref.a = 21
ref.b = 42
ref.c = 63
```

这也具有提高执行速度的附带效果，因为 Python 在运行时解析名称绑定，而第二个版本只需要执行一次解析。

引入可以进一步减小代码量的类似提议，例如使用“前导点号”，出于明白胜于隐晦的理由而被拒绝了(参见 https://mail.python.org/pipermail/python-ideas/2016-May/040070.html)。

生成器为什么不支持 with 语句?

由于技术原因，直接作为上下文管理器使用的生成器将无法正常工作。 最常见的情况下，当一个生成器被用作迭代器运行到完成时，不需要手动关闭。 如果需要，请在 `with` 语句中将它包装为 `contextlib.closing(generator)`。

为什么 if/while/def/class语句需要冒号?

冒号主要用于增强可读性(ABC语言实验的结果之一)。考虑一下这个:

```
if a == b
    print(a)
```

与

```
if a == b:
    print(a)
```

注意第二种方法稍微容易一些。请进一步注意，在这个FAQ解答的示例中，冒号是如何设置的；这是英语中的标准用法。

另一个次要原因是冒号使带有语法突出显示的编辑器更容易工作；他们可以寻找冒号来决定何时需要增加缩进，而不必对程序文本进行更精细的解析。

为什么Python在列表和元组的末尾允许使用逗号?

Python 允许您在列表，元组和字典的末尾添加一个尾随逗号:

```
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A":[1, 5],
    "B":[6, 7], # 最后的逗号是可选的，但风格很好
}
```

有几个理由允许这样做。

如果列表，元组或字典的字面值分布在多行中，则更容易添加更多元素，因为不必记住在上一行中添加逗号。这些行也可以重新排序，而不会产生语法错误。

不小心省略逗号会导致难以诊断的错误。例如:

```
x = [
  "fee",
  "fie"
  "foo",
  "fum"
]
```

这个列表看起来有四个元素，但实际上包含三个 : "fee", "fiefoo" 和 "fum" 。总是加上逗号可以避免这个错误的来源。

允许尾随逗号也可以使编程代码更容易生成。