代码风格指南

Kenneth Reitz

2025 年 9 月 6 日

目录

如果你问 Python 程序员他们最喜欢 Python 的哪一点，他们通常会提到它的高可读性。事实上，高可读性是 Python 语言设计的核心，遵循了代码被阅读的频率远高于它被编写的公认事实。

Python 代码高可读性的一个原因是它相对完整的代码风格指南和 “Pythonic” 惯用表达。

当一个资深的 Python 开发者（Pythonista）说某段代码不 “Pythonic” 时，他们通常意味着这些代码行没有遵循通用指南，并且无法以被认为是最优（即：最可读）的方式表达其意图。

在一些边缘情况下，Python 社区尚未就如何在 Python 代码中表达意图的最佳方式达成一致，但这种情况很少见。

1 通用概念

1.1 显式代码

虽然 Python 可以实现各种 “黑魔法”，但最显式和直接的方式是首选。

不好的示例：

```
def make_complex(*args):
    x, y = args
    return dict(**locals())
```

好的示例：

```
def make_complex(x, y):
    return {'x': x, 'y': y}
```

在上面的好示例中，`x` 和 `y` 从调用者那里显式接收，并返回一个显式的字典。开发者通过阅读函数的首行和末行就能清楚知道该做什么，而坏示例则不然。

1.2 一行一个语句

虽然某些复合语句（如列表推导式）因其简洁和表达力强而被允许和欣赏，但在一行代码中包含两个不相关的语句是不好的做法。

不好的示例：

```
print('one'); print('two')

if x == 1: print('one')

if <complex comparison> and <other complex comparison>:
    # do something
```

好的示例：

```
print('one')
print('two')

if x == 1:
    print('one')

cond1 = <complex comparison>
cond2 = <other complex comparison>
if cond1 and cond2:
    # do something
```

1.3 函数参数

参数可以通过四种不同方式传递给函数。

1. **位置参数**是必需的，没有默认值。它们是最简单的参数形式，适用于那些完全属于函数含义且顺序自然的少数参数。例如，在 `send(message, recipient)` 或 `point(x, y)` 中，函数用户可以轻松记住这两个函数需要两个参数以及它们的顺序。

   在这两种情况下，可以在调用函数时使用参数名称，并且可以改变参数的顺序，例如调用 `send(recipient='World', message='Hello')` 和 `point(y=2, x=1)`，但这会降低可读性且显得过于冗长，相比之下更直接的调用方式 `send('Hello', 'World')` 和 `point(1, 2)` 更好。

2. **关键字参数**不是必需的，有默认值。它们常用于函数的可选参数。当函数有超过两三个位置参数时，其签名更难记住，使用带默认值的关键字参数会很有帮助。例如，一个更完整的 `send` 函数可以定义为 `send(message, to, cc=None, bcc=None)`。这里的 `cc` 和 `bcc` 是可选的，如果未传递其他值，则评估为 `None`。

   调用带关键字参数的函数在 Python 中可以有多种方式；例如，可以按照定义中的参数顺序不显式命名参数，如 `send('Hello', 'World', 'Cthulhu`

', 'God')，将密件抄送发送给 God。也可以以其他顺序命名参数，如 send('Hello again', 'World', bcc='God', cc='Cthulhu')。但除非有充分理由，否则应避免这两种方式，最好遵循最接近函数定义的语法：send('Hello', 'World', cc='Cthulhu', bcc='God')。

作为补充，遵循 YAGNI 原则（You Aren't Gonna Need It），删除一个“以防万一”添加但似乎从未使用的可选参数（及其函数内的逻辑）通常比在需要时添加一个新的可选参数及其逻辑更困难。

3. **任意参数列表**是传递参数给函数的第三种方式。如果函数的意图通过一个可扩展数量的位置参数签名能更好地表达，可以使用 *args 结构。在函数体内，args 将是一个包含所有剩余位置参数的元组。例如，send(message, *args) 可以将每个接收者作为参数调用：send('Hello', 'God', 'Mom', 'Cthulhu')，在函数体内 args 将等于 ('God', 'Mom', 'Cthulhu')。

   然而，这种结构有一些缺点，应谨慎使用。如果函数接收多个同性质的参数，通常更清楚的做法是定义为一个接受单个参数（该参数为列表或任何序列）的函数。在这里，如果 send 有多个接收者，最好明确定义为：send(message, recipients) 并调用 send('Hello', ['God', 'Mom', 'Cthulhu'])。这样，函数用户可以预先将接收者列表作为列表操作，并且可以传递任何序列，包括无法像其他序列那样解包的迭代器。

4. **任意关键字参数字典**是传递参数给函数的最后一种方式。如果函数需要一系列未确定的命名参数，可以使用 **kwargs 结构。在函数体内，kwargs 将是一个包含所有未被其他关键字参数捕获的命名参数的字典。

   与任意参数列表的情况一样，出于类似原因需要谨慎使用这些强大技术：只有在确实需要时才使用它们，如果更简单、更清晰的结构足以表达函数意图，则不应使用。

编写函数的程序员需要决定哪些参数是位置参数，哪些是可选关键字参数，以及是否使用任意参数传递的高级技术。如果明智地遵循上述建议，就可以编写出易于阅读（名称和参数无需额外解释）、易于更改（添加新的关键字参数不会破坏其他代码部分）的 Python 函数。

1.4 避免魔法

Python 为黑客提供了强大的工具，包含丰富的钩子和工具，几乎可以实现任何类型的复杂技巧。例如，可以实现以下操作：

- 改变对象的创建和实例化方式

- 改变 Python 解释器导入模块的方式
- 甚至可以（如果需要，也推荐）在 Python 中嵌入 C 语言例程。

然而，这些选项有很多缺点，最好始终使用最直接的方式实现目标。主要缺点是使用这些结构会大大降低可读性。许多代码分析工具，如 pylint 或 pyflakes，将无法解析这种 “魔法” 代码。

我们认为，Python 开发者应该了解这些几乎无限的可能性，因为这让人相信不会遇到无法解决的问题。然而，了解如何以及尤其是什么时候不使用它们非常重要。

就像功夫大师一样，Pythonista 知道如何用一根手指杀人，但从不真正这样做。

1.5 我们都是负责任的用户

如上所述，Python 允许许多技巧，其中一些可能具有危险性。一个很好的例子是任何客户端代码都可以覆盖对象的属性和方法：Python 中没有 `private` 关键字。这种哲学与高度防御性的语言（如 Java）截然不同，Java 提供了许多机制来防止任何误用，这一哲学通过一句格言表达：“我们都是负责任的用户”。

这并不意味着，例如，没有属性被认为是私有的，或者在 Python 中无法实现适当的封装。相反，Python 社区更倾向于依靠一系列约定，表明这些元素不应被直接访问，而不是在代码和他人之间竖起坚固的壁垒。

私有属性和实现细节的主要约定是给所有 “内部” 元素添加下划线前缀。如果客户端代码违反此规则并访问这些标记的元素，那么如果代码被修改后出现任何不当行为或问题，责任在于客户端代码。

鼓励慷慨使用此约定：任何不打算被客户端代码使用的方法或属性都应以下划线开头。这将保证更好的职责分离和更容易修改现有代码；将私有属性公开化总是可能的，但将公开属性变为私有可能是一个更困难的操作。

1.6 返回值

当函数复杂度增加时，在函数体内使用多个返回语句并不罕见。然而，为了保持清晰的意图和可持续的可读性水平，最好避免从多个输出点返回有意义的值。

函数返回值有两种主要情况：函数正常处理后返回的结果，以及指示错误输入参数或其他原因导致函数无法完成计算或任务的错误情况。

如果你不想为第二种情况抛出异常，那么可能需要返回一个值（如 `None` 或 `False`）来表示函数无法正确执行。在这种情况下，最好在检测到不正确上下文时尽早返回。这

将有助于扁平化函数结构：返回错误语句后的所有代码都可以假设条件已满足，以便进一步计算函数的主要结果。拥有多个这样的返回语句通常是必要的。

然而，当函数在其正常流程中有多个主要退出点时，调试返回结果会变得困难，因此保持单一退出点可能更可取。这也有助于分解一些代码路径，多个退出点可能表明需要进行这样的重构。

```
def complex_function(a, b, c):
    if not a:
        return None  # Raising an exception might be better
    if not b:
        return None  # Raising an exception might be better
    # Some complex code trying to compute x from a, b and c
    # Resist temptation to return x if succeeded
    if not x:
        # Some Plan-B computation of x
    return x  # One single exit point for the returned value x will
       help
              # when maintaining the code.
```

1.7 惯用表达

编程惯用表达，简单来说，就是一种写代码的方式。关于编程惯用表达的讨论在 c2[1] 和 Stack Overflow[2] 上有很多。

惯用的 Python 代码通常被称为 Pythonic。

虽然通常有且最好只有一种显而易见的方式来实现某件事，但对于 Python 初学者来说，编写惯用 Python 代码的方式可能并不显而易见。因此，好的惯用表达必须有意识地学习。

以下是一些常见的 Python 惯用表达：

1.7.1 解包

如果你知道列表或元组的长度，可以通过解包为其元素分配名称。例如，因为 `enumerate()` 会为列表中的每个项目提供一个包含两个元素的元组：

```
for index, item in enumerate(some_list):
    # 使用 index 和 item 做一些事情
```

[1] http://c2.com/cgi/wiki?ProgrammingIdiom

[2] https://stackoverflow.com/questions/302459/what-is-a-programming-idiom

你也可以用它来交换变量：

```
a, b = b, a
```

嵌套解包也有效：

```
a, (b, c) = 1, (2, 3)
```

在 Python 3 中，PEP 3132[3] 引入了一种新的扩展解包方法：

```
a, *rest = [1, 2, 3]
# a = 1, rest = [2, 3]
a, *middle, c = [1, 2, 3, 4]
# a = 1, middle = [2, 3], c = 4
```

1.7.2 创建忽略变量

如果需要分配某些值（例如在解包时[4]）但不需要使用该变量，使用 `__`：

```
filename = 'foobar.txt'
basename, __, ext = filename.rpartition('.')
```

注意：

许多 Python 风格指南推荐使用单个下划线 `_` 表示丢弃变量，而不是这里推荐的双下划线 `__`。问题在于，`_` 通常被用作 `gettext()` 函数的别名，也在交互式提示符中用于保存最后一次操作的值。使用双下划线同样清晰且几乎同样方便，同时消除了意外干扰这些其他用例的风险。

1.7.3 创建长度为 N 的相同元素列表

使用 Python 列表的 `*` 运算符：

```
four_nones = [None] * 4
```

1.7.4 创建长度为 N 的列表的列表

因为列表是可变的，`*` 运算符（如上）会创建一个包含 N 个指向同一列表的引用的列表，这可能不是你想要的。相反，使用列表推导式：

[3] https://www.python.org/dev/peps/pep-3132

[4] https://docs.python-guide.org/writing/style/#unpacking-ref

```
four_lists = [[] for __ in range(4)]
```

1.7.5 从列表创建字符串

创建字符串的常见惯用方法是使用空字符串上的 `str.join()` 方法。

```
letters = ['s', 'p', 'a', 'm']
word = ''.join(letters)
```

这会将变量 `word` 的值设置为 'spam'。此惯用方法可应用于列表和元组。

1.7.6 在集合中搜索项目

有时我们需要在集合中搜索内容。来看两个选项：列表和集合。

以下是示例代码：

```
s = set(['s', 'p', 'a', 'm'])
l = ['s', 'p', 'a', 'm']

def lookup_set(s):
    return 's' in s

def lookup_list(l):
    return 's' in l
```

尽管这两个函数看起来相同，但因为 `lookup_set` 利用了 Python 集合是哈希表的特性，两者的查找性能差异很大。要确定一个项目是否在列表中，Python 必须逐一检查每个项目直到找到匹配的项目。这对于长列表来说非常耗时。而在集合中，项目的哈希值会告诉 Python 在集合中的哪个位置查找匹配项目。因此，即使集合很大，搜索也能很快完成。字典的搜索方式相同。有关更多信息，请参见 StackOverflow 页面[5]。有关各种常见操作在这些数据结构上所需时间的详细信息，请参见此页面[6]。

由于这些性能差异，在以下情况下通常建议使用集合或字典而不是列表：

- 集合包含大量项目
- 你将反复在集合中搜索项目
- 集合中没有重复项目

[5] https://stackoverflow.com/questions/513882/python-list-vs-dict-for-look-up-table

[6] https://wiki.python.org/moin/TimeComplexity?

对于小型集合或不经常搜索的集合，设置哈希表所需的时间和内存通常会超过通过提高搜索速度节省的时间。

1.8 Python 之禅

也称为 PEP 20[7]，是 Python 设计的指导原则。

```
>>> import this
The Zen of Python, by Tim Peters

优美胜于丑陋。
显式胜于隐式。
简单胜于复杂。
复杂胜于繁琐。
扁平胜于嵌套。
稀疏胜于密集。
可读性很重要。
特例不足以打破规则。
尽管实用性胜过纯粹性。
错误不应悄无声息地通过。
除非明确地沉默。
面对模棱两可，拒绝猜测的诱惑。
应该有一种——最好只有一种——显而易见的方式来实现。
尽管这种方式一开始可能不明显，除非你是荷兰人。
现在做总比不做好。
尽管不做往往比立刻做好。
如果实现难以解释，那是个坏主意。
如果实现易于解释，那可能是个好主意。
命名空间是个绝妙的主意——让我们多用它！
```

有关良好 Python 风格的示例，请参见 Python 用户组的这些幻灯片[8]。

1.9 PEP 8

PEP 8[9] 是 Python 的实际代码风格指南。高质量、易读的 PEP 8 版本也可在 http://pep8.org/ 找到。

强烈推荐阅读此文档。整个 Python 社区都尽力遵循其中列出的指南。一些项目可能会偶尔偏离它，而其他项目可能会修改其建议。

[7] https://www.python.org/dev/peps/pep-0020

[8] https://github.com/hblanks/zen-of-python-by-example

[9] https://www.python.org/dev/peps/pep-0008

尽管如此，将你的 Python 代码符合 PEP 8 通常是个好主意，有助于在与其他开发者合作项目时保持代码一致性。有一个命令行程序 pycodestyle[10]（以前称为 pep8），可以检查你的代码是否符合要求。通过在终端运行以下命令安装它：

```
$ pip install pycodestyle
```

然后在文件或一系列文件上运行它以获取任何违规的报告：

```
$ pycodestyle optparse.py
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
optparse.py:222:34: W602 deprecated form of raising exception
optparse.py:347:31: E211 whitespace before '('
optparse.py:357:17: E201 whitespace after '{'
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use 'in'
```

1.10 自动格式化

有几个自动格式化工具可以重新格式化你的代码，以符合 PEP 8。

1.10.1 autopep8

程序 autopep8 可用于自动将代码重新格式化为 PEP 8 风格。安装程序：

```
$ pip install autopep8
```

使用以下命令原地格式化文件：

```
$ autopep8 --in-place optparse.py
```

不使用 --in-place 标志将使程序直接将修改后的代码输出到控制台以供审查。--aggressive 标志将执行更大幅度的更改，并可多次应用以获得更大效果。

1.10.2 yapf

虽然 autopep8 专注于解决 PEP 8 违规问题，yapf 试图在符合 PEP 8 的基础上改善代码格式。此格式化工具旨在提供与编写 PEP 8 合规代码的程序员一样美观的代

[10] https://github.com/PyCQA/pycodestyle

码。安装：

```
$ pip install yapf
```

使用以下命令自动格式化文件：

```
$ yapf --in-place optparse.py
```

与 autopep8 类似，不使用 --in-place 标志将输出差异以供审查，然后再应用更改。

1.10.3 black

自动格式化工具 black 提供了一种固执且确定性的代码库重新格式化方式。其主要重点在于提供统一的代码风格，用户无需配置。因此，black 的用户可以完全忘记格式化问题。此外，由于其确定性方法，可保证仅包含相关更改的最小 git 差异。安装工具：

```
$ pip install black
```

格式化 Python 文件：

```
$ black optparse.py
```

添加 --diff 标志将提供代码修改以供审查，而不直接应用。

1.11 约定

以下是你应该遵循的一些约定，以使代码更易阅读。

1.11.1 检查变量是否等于常量

无需显式比较值与 True、None 或 0 —— 你可以直接在 if 语句中使用它。有关被视为假的值列表，请参见 Truth Value Testing[11]。

不好的示例：

```
if attr == True:
    print('True!')
```

[11] http://docs.python.org/library/stdtypes.html#truth-value-testing

```
if attr == None:
    print('attr is None!')
```

好的示例：

```
# Just check the value
if attr:
    print('attr is truthy!')

# or check for the opposite
if not attr:
    print('attr is falsey!')

# or, since None is considered false, explicitly check for it
if attr is None:
    print('attr is None!')
```

1.11.2 访问字典元素

不要使用 `dict.has_key()` 方法[12]。相反，使用 `x in d` 语法，或将默认参数传递给 `dict.get()`。

不好的示例：

```
d = {'hello': 'world'}
if d.has_key('hello'):
    print(d['hello'])    # prints 'world'
else:
    print('default_value')
```

好的示例：

```
d = {'hello': 'world'}

print(d.get('hello', 'default_value')) # prints 'world'
print(d.get('thingy', 'default_value')) # prints 'default_value'

# Or:
if 'hello' in d:
    print(d['hello'])
```

[12]译者注：`dict.has_key()` 方法在 Python 2 中存在，但在 Python 3 中已被移除。使用 `x in d` 是更现代且兼容的方式，确保代码在 Python 3 及未来版本中有效。

1.11.3 短小的方式操作列表

列表推导式[13]提供了一种强大而简洁的方式来处理列表。

生成器表达式[14]几乎遵循与列表推导式相同的语法，但返回的是生成器而不是列表。

创建新列表需要更多工作并使用更多内存。如果只是要遍历新列表，优先使用迭代器。

不好的示例：

```
# needlessly allocates a list of all (gpa, name) entires in memory
valedictorian = max([(student.gpa, student.name) for student in
    graduates])
```

好的示例：

```
valedictorian = max((student.gpa, student.name) for student in
    graduates)
```

当你确实需要创建第二个列表时使用列表推导式，例如如果你需要多次使用结果。

如果你的逻辑对于短列表推导式或生成器表达式来说过于复杂，考虑使用生成器函数而不是返回列表。

好的示例：

```
def make_batches(items, batch_size):
    """
    >>> list(make_batches([1, 2, 3, 4, 5], batch_size=3))
    [[1, 2, 3], [4, 5]]
    """
    current_batch = []
    for item in items:
        current_batch.append(item)
        if len(current_batch) == batch_size:
            yield current_batch
            current_batch = []
    yield current_batch
```

永远不要仅为了其副作用而使用列表推导式。

不好的示例：

[13] http://docs.python.org/tutorial/datastructures.html#list-comprehensions

[14] http://docs.python.org/tutorial/classes.html#generator-expressions

```
[print(x) for x in sequence]
```

好的示例：

```
for x in sequence:
    print(x)
```

1.11.4 过滤列表

不好的示例：

永远不要在遍历列表时从中删除项目。

```
# Filter elements greater than 4
a = [3, 4, 5]
for i in a:
    if i > 4:
        a.remove(i)
```

不要多次遍历列表。

```
while i in a:
    a.remove(i)
```

好的示例：

使用列表推导式或生成器表达式。

```
# comprehensions create a new list object
filtered_values = [value for value in sequence if value != x]

# generators don't create another list
filtered_values = (value for value in sequence if value != x)
```

1.11.5 修改原始列表的可能副作用

如果有其他变量引用原始列表，修改原始列表可能有风险。但如果你确实想这样做，可以使用切片赋值。

```
# replace the contents of the original list
sequence[::] = [value for value in sequence if value != x]
```

1.11.6 修改列表中的值

不好的示例：

记住，赋值永远不会创建新对象。如果两个或多个变量引用同一列表，改变其中一个会改变所有。

```
# Add three to all list members.
a = [3, 4, 5]
b = a                        # a and b refer to the same list object

for i in range(len(a)):
    a[i] += 3                # b[i] also changes
```

好的示例：

创建一个新列表对象并保持原始列表不变更安全。

```
a = [3, 4, 5]
b = a

# assign the variable "a" to a new list without changing "b"
a = [i + 3 for i in a]
```

使用 `enumerate()` 跟踪列表中的位置。

```
a = [3, 4, 5]
for i, item in enumerate(a):
    print(i, item)
# prints
# 0 3
# 1 4
# 2 5
```

`enumerate()` 函数比手动处理计数器具有更好的可读性。此外，它对迭代器进行了更好的优化。

1.11.7 从文件中读取

使用 `with open` 语法从文件读取。这将为你自动关闭文件。

不好的示例：

```
f = open('file.txt')
a = f.read()
```

```
print(a)
f.close()
```

好的示例：

```
with open('file.txt') as f:
    for line in f:
        print(line)
```

with 语句更好，因为它确保即使在 with 块内引发异常，文件也始终会被关闭。

1.11.8 行延续

当逻辑代码行超过接受的长度限制时，需要将其拆分为多个物理行。如果行的最后一个字符是反斜杠，Python 解释器会将连续的行连接起来。在某些情况下这很有用，但通常应避免，因为它很脆弱：反斜杠后添加空格将破坏代码，并可能产生意外结果。

更好的解决方案是在元素周围使用括号。行末未闭合的括号将使 Python 解释器连接下一行，直到括号闭合为止。花括号和大括号也是如此。

不好的示例：

```
my_very_big_string = """For a long time I used to go to bed early.
   Sometimes, \
    when I had put out my candle, my eyes would close so quickly
       that I had not even \
    time to say “I’ m going to sleep.” """

from some.deep.module.inside.a.module import a_nice_function,
   another_nice_function, \
    yet_another_nice_function
```

好的示例：

```
my_very_big_string = (
    "For a long time I used to go to bed early. Sometimes, "
    "when I had put out my candle, my eyes would close so quickly "
    "that I had not even time to say “I’ m going to sleep.” "
)

from some.deep.module.inside.a.module import (
    a_nice_function, another_nice_function,
       yet_another_nice_function)
```

然而，通常情况下，必须拆分长逻辑行表明你试图同时做太多事情，这可能会损害

可读性。