**Python入门之你必须了解的实用技巧**

本文主要记录 Python 中一些常用技巧，所描述的是告诉你怎么写才是更好? 如果你并不熟悉Python语法，希望你能在下面代码片段中看到Python的简单、优雅; 如果你象我这样，对 Python 有兴趣或并正在学习，我相信下面的技巧并不会让你失望; 如果你已经是一名 Pythoner ，那么很乐于你分享你的经验和技巧。

**Python 禅道**

这是Python的指导原则，但有不同诠释。

如果您使用的一种编程语言是以小品喜剧艺术团命名的，你最好有幽默感。

美丽优于丑陋。

明确优于含蓄。

简单比复杂好。

平倘优于嵌套。

稀疏比密集更好。

特殊情况不能特殊到打破规则。

错误不应该默默传递。

......

**代码风格: 提高可读性**

Programs must be written for people to read, and only incidentally for machines to execute.

—Abelson & Sussman, Structure and Interpretation of Computer Programs

**PEP 8: Python 代码风格指南**

值得阅读:

http://www.python.org/dev/peps/pep-0008/

**空格(行)使用 (1)**

◆ 使用 4 个空格缩进。

◆ 不要使用制表符。

◆ 不要将制表符和空格混合使用。

◆ IDEL和Emacs的Python的都支持 spaces模式。

◆ 每个函数之间应该有一个空行。

◆ 每一个 Class 之间应该有两个空行。

**空格(行)使用 (2)**

◆ 在使用 字典(dict), 列表(list), 元组(tuple), 参数(argument)列表时， 应在 "," 前添加一个空格, 并且使用字典(dict)时，在 ":" 号后添加空格，而不是在前面添加。

◆ 在括号之前或参数之前不添加空格。

◆ 在文档注释中前后应该没有空格。

Python代码

1. **def** make\_squares(key, value=0):
2. """Return a dictionary and a list..."""
3. d = {key: value}
4. l = [key, value]
5. **return** d, l

**命名**

◆ joined\_lower 可以是 函数名, 方法名, 属性名

◆ joined\_lower or ALL\_CAPS 是常量

◆ StudlyCaps 类名

◆ camelCase 只有在预先制定好的命名规范使用

◆ 属性: interface, \_internal, \_\_private

◆ 但尽量避免\_\_private形式。下面两个链接解释了 为什么python中没有 private声明?

http://stackoverflow.com/questions/70528/why-are-pythons-private-methods-not-actually-private

http://stackoverflow.com/questions/1641219/does-python-have-private-variables-in-classes

**较长代码行**

保持一行代码在 80 个字符长度。

在括号内使用隐含的行延续：

Python代码

1. **def** \_\_init\_\_(self, first, second, third,
2. fourth, fifth, sixth):
3. output = (first + second + third
4. + fourth + fifth + sixth)

或者在需要换行的位置使用 \ 来延续行：

Python代码

1. VeryLong.left\_hand\_side \
2. = even\_longer.right\_hand\_side()

另外，使用反斜杠是有风险的，如果你添加一个空格在反斜杠后面，它就出错了。此外，它使代码难看。

**较长字符串**

将相邻的字符串进行连接的做法：

Python代码

1. >>> **print** 'o' 'n' "e"
2. one

虽然字符之间的空格不是必需的，但是这样有助于可读性。

Python代码

1. >>> **print** 't' r'\/\/' """o"""
2. t\/\/o

用一个 “r“ 开头的字符串是一个“raw“的字符串(类似java中的转义符)。上面的反斜杠就会当成普通字符串处理。他们对正则表达式和Windows文件系统路径非常有用。

注意：使用字符串变量名无法通过以上方式进行连接。

Python代码

1. >>> a = 'three'
2. >>> b = 'four'
3. >>> a b
4. File "<stdin>", line 1
5. a b
6. ^
7. SyntaxError: invalid syntax

这是因为自动连接是由Python解析器/编译器来处理的，因为其无法在编译时对变量值进行"翻译"，所以就这种必须在运行时使用“+“运算符来连接变量。

**复合语句**

Good：

Python代码

1. **if** foo == 'blah':
2. do\_something()
3. do\_one()
4. do\_two()
5. do\_three()

Bad：

Python代码

1. **if** foo == 'blah': do\_something()
2. do\_one(); do\_two(); do\_three()

**文档注释(Docstrings) & 注释**

文档注释 = 用于解释如何使用代码

文档注释公约：http://www.python.org/dev/peps/pep-0257/

注释 = 为什么 (理由) & 代码如何工作的如：

Python代码

1. # !!! BUG: ...
2. # !!! FIX: This is a hack
3. # ??? Why is this here?

注释对于任何语言开发者来说已经最基本的东西了，这里就不详细说了.

**交换变量**

在其它语言的交换变量的做法一般是:

Java代码

1. temp = a
2. a = b
3. b = temp

Python的做法：

Python代码

1. b, a = a, b

也许你见到过这样的情况，但是你知道它是如何工作的吗?

◆ 首先，逗号是元组构造语法。

◆ 等号的右边是定义一个元组 (tuple packing).

◆ 其左边为一个目标元组 (tuple unpacking)).

右边的元组根据名称被 unpacked 到左边的无组。

更多关于 unpacked例子：

Python代码

1. >>> info =['David', 'Pythonista', '+1250']
2. >>> name, title, phone = info
3. >>> name
4. 'Davids'
5. >>> title
6. 'Pythonista'
7. >>> phone
8. '+1250'

在结构化的数据上使用循环：

info 是在上面定义的一个 list . 所以下面的 people 有两个项, 两个项都是分别都拥有三个项的 list.

Python代码

1. >>> people = [info, ['Guido', 'BDFL', 'unlisted']]
2. >>> **for** (name, title, phone) **in** people:
3. ...     **print** name, phone
4. ...
5. David +1250
6. Guido unlisted

以上循环中，people中的两个项(list item)，都已经被 unpacked 到 (name, title, phone) 无组中。

可以任意嵌套(只要左右两边的结构一定要能够匹配得上):

Python代码

1. >>> david, (gname, gtitle, gphone) = people
2. >>> gname
3. 'Guido'
4. >>> gtitle
5. 'BDFL'
6. >>> gphone
7. 'unlisted'
8. >>> david
9. ['David', 'Pythonista', '+1250']

**更多关于 Tuples**

我们看到的是元组通过逗号构造，而不是括号。例如：

Python代码

1. >>> 1,
2. (1,)

Python的解释器会为你显示括号，所以建议你使用括号：

Python代码

1. >>> (1,)
2. (1,)

千万不要忘记逗号!

Python代码

1. >>> (1)
2. 1

在只有一个元素的元组，尾随逗号是必须的，在2 + 元素的元组，尾随逗号是可选的。 如果创建一个 0或空元组，一对括号是快捷的语法：

Python代码

1. >>> ()
2. ()
3. >>> tuple()
4. ()

一个常见的​​错误当你并不想要一个无组，却无意的添加了一个逗号，很容易造成你在代码中的错误，如：

Python代码

1. >>> value = 1,
2. >>> value # is a tuple, not a int
3. (1,)

所以，当你发现一个元组时，赶紧去找一下那个,号吧。

**关于 "\_"**

是一个非常有用的功能，但是却很少有人知道。

当你在交互式模式下(如 IDEL)计算一个表达式或调用一个函数后，其结果必然是一个临时名称，\_(下划线)：

Python代码

1. >>> 1 + 1
2. 2
3. >>> \_
4. 2

在 \_ 中存储最后输出的值。

当输出的结果是 None 或没有任何输出时，而 \_ 的值并不会改变，仍然保存上一次的值。这就是方便所在。

当然，这只能交互式的模式中使用，在模块中不能支持。

这在交互式模式中是非常有用的，当你在过程中没有保存计算结果 或 你想看最后一步的执行的输出结果：

Python代码

1. >>> **import** math
2. >>> math.pi / 3
3. 1.0471975511965976
4. >>> angle = \_
5. >>> math.cos(angle)
6. 0.50000000000000011
7. >>> \_
8. 0.50000000000000011

**创建String: 从列表中创建**

开始定义一个 string 列表：

Python代码

1. colors = ['red', 'blue', 'green', 'yellow']

当我们需要将上面的列表连接成一个字符串。尤其当 list 是一个很大的列表时....

不要这样做：

Python代码

1. result = ''
2. **for** s **in** colors:
3. result += s

这种方式效率非常低下的，它有可怕的内存使用问题，至于为什么，如果你是 javaer 的话，其中的 string 连接，我想你并不陌生。

相反，你应该这样做：

Python代码

1. result = ''.join(colors)

当你只有几十或几百个string项连接时，它们效率上并不会太大的差别。但你要在养成写高效代码的习惯，因为当字符串数千时，join 比起 for 连接性能会能有所提升。

如果你需要使用一个函数来生成一个字符串列表，同样可以使用：

Python代码

1. result = ''.join(fn(i) **for** i **in** items)

尽可能的使用

Good:

Python代码

1. **for** key **in** d:
2. **print** key

◆ 使用 in 一般情况下是非常快的。

◆ 这种方式也适用于其它的容器对象(如 list，tuple 和 set)。

◆ in 是操作符(正如上面所看到的)。

Bad:

Python代码

1. **for** key **in** d.keys():
2. **print** key

保持与上面的一致性，使用 use key in dict 方式，而不是 dict.has\_key():

Python代码

1. # do this:
2. **if** key **in** d:
3. ...do something with d[key]
5. # not this:
6. **if** d.has\_key(key):
7. ...do something with d[key]

**字典中的 get 函数**

我们经常需要在字典中初始化数据：

以下是不好的实现方法：

Python代码

1. navs = {}
2. for (portfolio, equity, position) in data:
3. if portfolio not in navs:
4. navs[portfolio] = 0
5. navs[portfolio] += position \* prices[equity]

使用dict.get(key, default) 删除 if 判断代码：

Python代码

1. navs = {}
2. for (portfolio, equity, position) in data:
3. navs[portfolio] = (navs.get(portfolio, 0)
4. + position \* prices[equity])

这种方式更为直接。

**字典中的 setdefault 函数 (1)**

当我们要初始化一个可变字典的值。每个字典的值将是一个列表。下面是不好的做法：

初始化可变字典的值：

Python代码

1. equities = {}
2. for (portfolio, equity) in data:
3. if portfolio in equities:
4. equities[portfolio].append(equity)
5. else:
6. equities[portfolio] = [equity]

通过 dict.setdefault(key, default) 使这段代码工作的更好:

Python代码

1. equities = {}
2. for (portfolio, equity) in data:
3. equities.setdefault(portfolio, []).append(
4. equity)

dict.setdefault()等同于“ get, or set & get“ 或"如果没有，就设置"; 如果你的字典Key是复杂的计算或long类型，使用 setdefault 是特别有效的。

**字典中的 setdefault 函数 (2)**

在我们看到的setdefault字典方法也可以作为一个独立的语句使用：

Python代码

1. avs = {}
2. for (portfolio, equity, position) in data:
3. navs.setdefault(portfolio, 0)
4. navs[portfolio] += position \* prices[equity]

我们在这里忽略了字典的setdefault方法返回的默认值。我们正利用的setdefault中的作用，仅仅只是在dict中没有 key 的值的时候才会设置。

**创建 & 分割字典**

如果你有两份 list 对象，希望通过这两个对象构建一个 dict 对象。

Python代码

1. given = ['John', 'Eric', 'Terry', 'Michael']
2. family = ['Cleese', 'Idle', 'Gilliam', 'Palin']
3. pythons = dict(zip(given, family))
4. >>> pprint.pprint(pythons)
5. {'John': 'Cleese',
6. 'Michael': 'Palin',
7. 'Eric': 'Idle',
8. 'Terry': 'Gilliam'}

同样，如果希望获取两份列表，也是非常简单：

Python代码

1. >>> pythons.keys()
2. ['John', 'Michael', 'Eric', 'Terry']
3. >>> pythons.values()
4. ['Cleese', 'Palin', 'Idle', 'Gilliam']

需要注意的是，上面 list 虽然是有序的，但是 dict 中的 keys 和 values 是无序的，这正是因为 dict 本质就是无序存储的。

**索引 & 项 (1)**

如果你需要一个列表，这里有一个可爱的方式来节省你的输入：

Python代码

1. >>> items = 'zero one two three'.split()
2. >>> print items
3. ['zero', 'one', 'two', 'three']

如果我们需要遍历这个 list ，而且需要 index 和 item：

Python代码

1. - or -
2. i = 0
3. for item in items:      for i in range(len(items)):
4. print i, item               print i, items[i]
5. i += 1

**索引 & 项 (2): enumerate**

通过 enumerate 可以返回 list 中的 (index, item)元组：

Python代码

1. >>> print list(enumerate(items))
2. [(0, 'zero'), (1, 'one'), (2, 'two'), (3, 'three')]

于是，遍历list获取index 及 item 就更加简单了：

Python代码

1. for (index, item) in enumerate(items):
2. print index, item

Python代码

1. # compare:              # compare:
2. index = 0               for i in range(len(items)):
3. for item in items:              print i, items[i]
4. print index, item
5. index += 1

不难看出，使用 enumerate 比起下面两种方式，更加简单，更加容易阅读，这正是我们想要的。

下面是例子是如何通过 enumerate 返回迭代器：

Python代码

1. >>> enumerate(items)
2. <enumerate object at 0x011EA1C0>
3. >>> e = enumerate(items)
4. >>> e.next()
5. (0, 'zero')
6. >>> e.next()
7. (1, 'one')
8. >>> e.next()
9. (2, 'two')
10. >>> e.next()
11. (3, 'three')
12. >>> e.next()
13. Traceback (most recent call last):
14. File "<stdin>", line 1, in ?
15. StopIteration

**默认参数值**

这是对于一个初学者常犯的错误，甚至于一些高级开发人员也会遇到，因为他们并不了解 Python 中的 names.

Python代码

1. def bad\_append(new\_item, a\_list=[]):
2. a\_list.append(new\_item)
3. return a\_list

这里的问题是，a\_list是一个空列表，默认值是在函数定义时进行初始化。因此，每次调用该函数，你会得到不相同的默认值。尝试了好几次：

Python代码

1. >>> print bad\_append('one')
2. ['one']
3. >>> print bad\_append('two')
4. ['one', 'two']

列表是可变对象，你可以改变它们的内容。正确的方式是先获得一个默认的列表(或dict，或sets)并在运行时创建它。

Python代码

1. def good\_append(new\_item, a\_list=None):
2. if a\_list is None:
3. a\_list = []
4. a\_list.append(new\_item)
5. return a\_list

**判断 True 值**

Python代码

1. # do this:        # not this:
2. if x:             if x == True:
3. pass              pass

它的优势在于效率和优雅。

判断一个list：

Python代码

1. # do this:        # not this:
2. if items:         if len(items) != 0:
3. pass              pass
5. # and definitely not this:
6. if items != []:
7. pass

**True 值**

True和False是内置的bool类型的布尔值的实例。谁都只有其中的一个实例。

|  |  |
| --- | --- |
| False | True |
| False (== 0) | True (== 1) |
| "" (empty string) | any string but "" (" ", "anything") |
| 0, 0.0 | any number but 0 (1, 0.1, -1, 3.14) |
| [], (), {}, set() | any non-empty container ([0], (None,), ['']) |
| None | almost any object that's not explicitly False |

**简单比复杂好**

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

—Brian W. Kernighan

**不要重新发明轮子**

在写任何代码之前,

➔ ➔ ➔ ➔

检查python 标准库.http://docs.python.org/library/index.html

检查Python的包索引 (the "Cheese Shop"):

http://cheeseshop.python.org/pypi

Search the web. Google is your friend.

原文：[urlhttp://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html][url]