# 代码的整洁之道

书写代码时，能够有意识保持代码的整洁，养成一种代码书写的习惯，本身就是一种智慧。

### 1. 代码的整洁之道

每一篇代码都应该是一篇蕴含着作者解决某一问题逻辑思考的“散文”，即使不懂代码的读者也能够行云流水般的阅读，从而有个大概的认知，知道作者想要解决什么问题，和大概是怎么解决的。因此有必要提及代码的整洁之道，梳理一些关键事项，以备参考。

* **变量的命名应能反映变量本身的意义**

代码书写时为了图一时方便，往往不注重变量名的命名，而是随意的起名，例如代码行中充斥着*x、y、i、j、k*等单个变量名，及*jh,ik*等不反映变量名意义的字母组合。日后需要返回来应用已书写的代码或者迁移代码于新项目时，无法快速理解变量的意义，使得代码可读性差，从而会浪费更多时间再度理解，这也为他人迁移应用该代码形成了一定的阻碍。

在变量名命名时，建议的形式包括三种：

一是单个词，例如markers=['.','+','o','^','x','s']，这是应用matplotlib图表库打印图表时，定义图表标记类型所定义的类型列表，其名字能够反映所要表述变量的意义。

二是字母组合，其一示例为landmarkPts=[Point(coordi[0],coordi[1]) for coordi in np.stack((landmarks[0], landmarks[1]), axis=-1)]中的*landmarkPts*，或者不引入缩写为*landmarkPoints*，可以直接理解该变量名为地标点，形式为字母组合时后一字母首字母大写（Camel-case）；其二是，*landmark\_pts*或者*landmark\_points*形式，即多个字母组合时中间以短下划线隔离，之后字母的首字母不需大写（underscore-separated，Snake-case）。

* **给出关键的注释**

“烂笔头”的重要性无需质疑，虽然好的变量名能够一定程度上解释语句的意义，但是解决问题的逻辑通常需要给与注释（使用# 或者``` note ```），尤其解决问题时思维碰撞时的关键想法。例如对函数功能的注释：

def recursive\_factorial(n):  
 '''  
 阶乘计算。  
  
 Parameters  
 ----------  
 n : int  
 阶乘计算的末尾值.  
  
 Returns  
 -------  
 int  
 阶乘计算结果.  
  
 '''  
 if n == 1:  
 return 1  
 else:  
 return n \* recursive\_factorial(n - 1)

及对关键语句的解释：

idx = tempDf[tempDf['cluster'] == -1].index # 删除字段“cluster”值为-1对应的行，即未形成聚类的独立OSM点数据  
 tempDf.drop(idx,inplace=True)

如果以上示例中不给出注释，则很难立刻理解出这些代码书写的意图，往往不得不花费较大精力来推断，甚至不得不阅读整段代码。当然，要以对函数和变量适合的命名为先，再注释补充。不应该是草草的命名后用大量的注释来解释，本末倒置。

* **尽量保持定义的一个函数仅作一件事情，及最小的迁移代价**

每一段函数代码都是解决某一问题的一种方法，很多时候在同一项目、或不同项目中，这种方法会不断被重复调用，因此该函数应该能够以最少的代价来迁移，无需或者很少的改动代码就可以调用。

一是，一个函数最好仅做一件事情，保持代码具有更好的易读性和可迁移性。这个并不难以理解，如果该函数可以同时做多件事情，当其它项目仅需要该函数的部分功能时，事情就会变得复杂起来。

二是，有时在单独的函数内部包含了全局变量，迁移后会提示全局变量未定义的情况。因此建议函数定义时，尽量通过函数参数传递变量。

三是，在单独函数命名，及函数内变量命名时，尽量保持名称的通用性，避免迁移后的名字只能反映当初项目的内容，例如,*def ffill(arr):*函数*ffill()*表示向前填充数组中的空数据。如果函数名及参数名改为具体的def *landmarks\_ffill(landmarks\_arr)*，迁移后的变量可能不是*landmarks*，就会容易引起歧义，造成代码阅读上的干扰。

* **复杂的程序要学会使用类和分文件——系统的思维**

代码是可以在不知不觉中变得复杂起来，少则千行，多则万行。那么所有代码位于一个文件（.py）中，包含有数十个单独的函数，代码的管理就会成为问题。因此学会应用类的面向对象编程（Object-oriented programming，OOP）来组织相关属性和方法，并用多个子包（Subpackage）和分文件（Module，模块）来切分代码，在相关文件中只是引用，例如import driverlessCityProject\_spatialPointsPattern\_association\_basic as basic，调入模块并取别名为basic来使用代码，那么代码的结构就会比较清晰，也避免了处于同一文件中，不容易查找的弊病；同时，方便后续代码打包，发布到[PyPI](https://pypi.org/)①，分享工具。

类和分文件只是系统思维表现的一种手段，对于项目所有代码的合理组织，确定数据的流动走向、前后关联层级的配置、数据库的使用、整体结构的把握，才能够保证代码的稳健性（robustness）。

《代码的整洁之道》(Clean Code: A Handbook of Agile Software Craftsmanship)[1]这本阐述代码哲理的书，不管是对刚刚进入代码领域，还是早已浸淫代码多年的程序员，通读或偶尔翻阅，都会受益良多。

### 2. 代码的风格

参考引用：[Google 开源项目风格指南](https://zh-google-styleguide.readthedocs.io/en/latest/google-python-styleguide/python_style_rules/" \l "section-2)②，并根据该书情况稍作调整。

* 风格是为了代码具有更好的可读性；
* 风格同一模块和同一函数保持一致，同一项目保持一致，同风格指南或共识性规范尽量保持一致；
* 缩进风格统一，示例：

# 同开始分界符(左括号)对齐（Aligned with opening delimiter）  
foo = long\_function\_name(var\_one, var\_two,  
 var\_three, var\_four)  
  
# 续行多缩进一级以同其他代码区别  
def long\_function\_name(  
 var\_one, var\_two, var\_three,  
 var\_four):  
 print(var\_one)  
  
# 悬挂缩进需要多缩进一级（space hanging indent; nothing on first line）  
foo = long\_function\_name(  
 var\_one, var\_two,  
 var\_three, var\_four)  
# Aligned with opening delimiter in a dictionary  
foo = {  
 long\_dictionary\_key: value1 +  
 value2,   
 ...  
}

* 行长度，示例

# Python会将圆括号，中括号和花括号中的行隐式的连接起来  
Yes: foo\_bar(self, width, height, color='black', design=None, x='foo',  
 emphasis=None, highlight=0)  
  
 if (width == 0 and height == 0 and  
 color == 'red' and emphasis == 'strong'):  
  
 # 如果一个文本字符串在一行放不下，可以使用圆括号来实现隐式行连接   
 x = ('This will build a very long long '  
 'long long long long long long string')   
  
 # 在注释中，如果必要，将长的URL放在一行上  
 # <http://www.example.com/us/developer/documentation/api/content/v2.0/csv_file_name_extension_full_specification.html>

* 序列元素尾部逗号

Yes: golomb3 = [0, 1, 3]  
Yes: golomb4 = [  
 0,  
 1,  
 4,  
 6,  
 ]  
  
No: golomb4 = [  
 0,  
 1,  
 4,  
 6  
 ]

* 空行: 顶级定义之间空两行，方法定义之间空一行

顶级定义之间空两行，比如函数或者类定义。方法定义，类定义与第一个方法之间，都应该空一行。函数或方法中，根据具体情况确定空一行，还空两行。

* 空格

括号内不要有空格。

Yes: spam(ham[1], {eggs: 2}, [])  
  
No: spam( ham[ 1 ], { eggs: 2 }, [ ] )

不要在逗号、分号、冒号前面加空格，但应该在它们后面加（除了在行尾）。

Yes: if x == 4:  
 print(x, y)  
 x, y = y, x  
  
No: if x == 4 :  
 print(x , y)  
 x , y = y , x

参数列表，索引或切片的左括号前不应加空格。

Yes: spam(1)  
  
no: spam (1)  
  
Yes: dict['key'] = list[index]  
  
No: dict ['key'] = list [index]

在二元操作符两边都加上一个空格，比如赋值（=）, 比较（==, <, >, !=, <>, <=, >=, in, not in, is, is not），布尔（and, or, not）。 至于算术操作符两边的空格该如何使用，需要具体情况判断，不过两侧务必要保持一致。

Yes: x == 1  
  
No: x<1

当 = 用于指示关键字参数或默认参数值时，不要在其两侧使用空格。 但若存在类型注释的时候，需要在 = 周围使用空格.

Yes: def complex(real, imag=0.0): return magic(r=real, i=imag)  
Yes: def complex(real, imag: float = 0.0): return Magic(r=real, i=imag)  
  
No: def complex(real, imag = 0.0): return magic(r = real, i = imag)  
No: def complex(real, imag: float=0.0): return Magic(r = real, i = imag)

不要用空格来垂直对齐多行间的标记，因为这会成为维护的负担(适用于:, #, =等):

Yes:  
 foo = 1000 # comment  
 long\_name = 2 # comment that should not be aligned  
  
 dictionary = {  
 "foo": 1,  
 "long\_name": 2,  
 }  
  
No:  
 foo = 1000 # comment  
 long\_name = 2 # comment that should not be aligned  
  
 dictionary = {  
 "foo" : 1,  
 "long\_name": 2,  
 }

* 注释

函数部分注释按[Spyder](https://www.spyder-ide.org/)③提供的方法，例如：

def recursive\_factorial(n):  
 '''  
 阶乘计算。  
  
 Parameters  
 ----------  
 n : int  
 阶乘计算的末尾值.  
  
 Returns  
 -------  
 int  
 阶乘计算结果.  
  
 '''  
 if n==1:  
 return 1  
 else:  
 return n\*recursive\_factorial(n-1)

#之后空一格

Yes:  
 # the next element is i+1  
  
No:  
 #the next element is i+1

无需注释模块中的所有函数:

1. 公共的API需要注释；
2. 在代码的安全性，清晰性和灵活性上进行权衡是否注释；
3. 对于容易出现类型相关的错误的代码进行注释；
4. 难以理解的代码请进行注释；
5. 若代码中的类型已经稳定，可以进行注释。对于一份成熟的代码，多数情况下，即使注释了所有的函数，也不会丧失太多的灵活性。

* 字符串

% 格式化字符前后空一格。

Yes:  
 x = '%s, %s!' % (imperative, expletive)  
  
Noe:  
 x = '%s, %s!'%(imperative, expletive)

避免在循环中用+和+=操作符来累加字符串。由于字符串是不可变的，这样做会创建不必要的临时对象，并且导致二次方而不是线性的运行时间。作为替代方案，可以将每个子串加入列表，然后在循环结束后用 .join 连接列表。（也可以将每个子串写入一个 cStringIO.StringIO 缓存中）

Yes: items = ['<table>']  
 for last\_name, first\_name in employee\_list:  
 items.append('<tr><td>%s, %s</td></tr>' % (last\_name, first\_name))  
 items.append('</table>')  
 employee\_table = ''.join(items)  
  
No: employee\_table = '<table>'  
 for last\_name, first\_name in employee\_list:  
 employee\_table += '<tr><td>%s, %s</td></tr>' % (last\_name, first\_name)  
 employee\_table += '</table>'

在同一个文件中，保持使用字符串引号的一致性。使用单引号’或者双引号”之一用以引用字符串，并在同一文件中沿用。在字符串内可以使用另外一种引号，以避免在字符串中使用。

Yes:  
 Python('Why are you hiding your eyes?')  
 Gollum("I'm scared of lint errors.")  
 Narrator('"Good!" thought a happy Python reviewer.')  
  
No:  
 Python("Why are you hiding your eyes?")  
 Gollum('The lint. It burns. It burns us.')  
 Gollum("Always the great lint. Watching. Watching.")

为多行字符串使用三重双引号"""而非三重单引号'。当且仅当项目中使用单引号’来引用字符串时，才可能会使用三重'为非文档字符串的多行字符串来标识引用。文档字符串必须使用三重双引号"。多行字符串不应随着代码其他部分缩进的调整而发生位置移动。如果需要避免在字符串中嵌入额外的空间，可以使用串联的单行字符串或者使用 textwrap.dedent() 来删除每行多余的空间。

No:  
long\_string = """This is pretty ugly.  
Don't do this.  
"""  
  
Yes:  
long\_string = """This is fine if your use case can accept  
 extraneous leading spaces."""  
  
Yes:  
long\_string = ("And this is fine if you cannot accept\n" +  
 "extraneous leading spaces.")  
  
Yes:  
long\_string = ("And this too is fine if you cannot accept\n"  
 "extraneous leading spaces.")  
  
Yes:  
import textwrap  
  
long\_string = textwrap.dedent("""\  
 This is also fine, because textwrap.dedent()  
 will collapse common leading spaces in each line.""")

* 文件和sockets

除文件外，sockets或其他类似文件的对象在没有必要的情况下打开，会有许多副作用，例如：

1. 它们可能会消耗有限的系统资源，如文件描述符。如果这些资源在使用后没有及时归还系统，那么用于处理这些对象的代码会将资源消耗殆尽；
2. 持有文件将会阻止对于文件的其他诸如移动、删除之类的操作；
3. 仅仅是从逻辑上关闭文件和sockets，那么它们仍然可能会被其共享的程序在无意中进行读或者写操作。只有当它们真正被关闭后，对于它们尝试进行读或者写操作将会抛出异常，并使得问题快速显现出来。

而且，幻想当文件对象析构时，文件和sockets会自动关闭，试图将文件对象的生命周期和文件的状态绑定在一起的想法，都是不现实的。因为有如下原因：

1. 没有任何方法可以确保运行环境会真正的执行文件的析构。不同的Python实现采用不同的内存管理技术，比如延时垃圾处理机制。延时垃圾处理机制可能会导致对象生命周期被任意无限制的延长；
2. 对于文件意外的引用，会导致对于文件的持有时间超出预期（比如对于异常的跟踪，包含有全局变量等）。

推荐使用 with语句 以管理文件：

with open("hello.txt") as hello\_file:  
 for line in hello\_file:  
 print line

对于不支持使用with语句的类似文件的对象，使用contextlib.closing():

import contextlib  
  
with contextlib.closing(urllib.urlopen("http://www.python.org/")) as front\_page:  
 for line in front\_page:  
 print line

* 导入格式 采用部分Google 开源项目风格指南

导入应该按照从最通用到最不通用的顺序分组:

from \_\_future\_\_ import absolute\_import # \_\_future\_\_ 导入  
import sys # 标准库导入  
import tensorflow as tf # 第三方库导入  
from otherproject.ai import mind # 本地代码子包导入

每种分组中，应该根据每个模块的完整包路径按字典序排序，忽略大小写。

import collections  
import queue  
import sys  
  
from absl import app  
from absl import flags  
import bs4  
import cryptography  
import tensorflow as tf  
  
from book.genres import scifi  
from myproject.backend import huxley  
from myproject.backend.hgwells import time\_machine  
from myproject.backend.state\_machine import main\_loop  
from otherproject.ai import body  
from otherproject.ai import mind  
from otherproject.ai import soul  
  
# Older style code may have these imports down here instead:  
#from myproject.backend.hgwells import time\_machine  
#from myproject.backend.state\_machine import main\_loop

导入代码统一位置，1，位于模块文件开始行；2，位于函数内开始行。一次性在模块开始行调入，可以避免函数内重复调入。但如果导入的模块体量较大，且存在库之间的冲突，如果一次性在模块开始行导入，代码无法顺利运行，此时在函数内开始行调入较为合理。因此具体调入位置，根据具体情况比较确定。

* 命名

模块名写法: *module\_name* ；包名写法: *package\_name* ；类名: *ClassName* ；方法名: *method\_name* ；异常名: *ExceptionName* ；函数名: *function\_name* ；全局常量名: *GLOBAL\_CONSTANT\_NAME* ；全局变量名: *global\_var\_name* ；实例名: *instance\_var\_name* ；函数参数名: *function\_parameter\_name* ；局部变量名: *local\_var\_name* 。函数名，变量名和文件名应该是描述性的，尽量避免缩写，特别要避免使用非项目人员不清楚难以理解的缩写，不要通过删除单词中的字母来进行缩写。始终使用 .py作为文件后缀名，不要用破折号。

应该避免的名称：

1. 单字符名称，除了计数器和迭代器，作为 try/except 中异常声明的 e，作为 with 语句中文件句柄的 f；
2. 包/模块名中的连字符(-)；
3. 双下划线开头并结尾的名称(Python保留, 例如\_\_init\_\_)。

命名约定：

1. 所谓”内部(Internal)”表示仅模块内可用，或者, 在类内是保护或私有的；
2. 用单下划线(\_)开头表示模块变量或函数是protected的（使用from module import \*时不会包含）；
3. 用双下划线(\_\_)开头的实例变量或方法表示类内私有；
4. 将相关的类和顶级函数放在同一个模块里，不像Java，没必要限制一个类一个模块；
5. 对类名使用大写字母开头的单词（如*CapWords*, 即Pascal风格），但是模块名应该用小写加下划线的方式（如*lower\_with\_under.py*）。尽管已经有很多现存的模块使用类似于*CapWords.py*这样的命名，但现在已经不鼓励这样做，因为如果模块名碰巧和类名一致，会让人困扰。

文件名：

所有Python脚本文件都应该以 .py 为后缀名且不包含 -。若是需要一个无后缀名的可执行文件，可以使用软联接或者包含 exec "$0.py" "$@" 的bash脚本。

Python之父Guido推荐的规范:

| Type | Public | Internal

|----------------------------|--------------------|--------------------

| Modules | lower\_with\_under | \_lower\_with\_under

| Packages | lower\_with\_under |   
| Classes | CapWords | \_CapWords   
| Exceptions | CapWords |   
| Functions | lower\_with\_under() | \_lower\_with\_under()   
| Global/Class Constants | CAPS\_WITH\_UNDER | \_CAPS\_WITH\_UNDER   
| Global/Class Variables | lower\_with\_under | \_lower\_with\_under   
| Instance Variables | lower\_with\_under | \_lower\_with\_under (protected) or \_\_lower\_with\_under (private)

| Method Names | lower\_with\_under() | \_lower\_with\_under() (protected) or \_\_lower\_with\_under() (private)

| Function/Method Parameters | lower\_with\_under |   
| Local Variables | lower\_with\_under |

* Main

即使是一个打算被用作脚本的文件，也应该是可导入的。并且简单的导入不应该导致这个脚本的主功能（main functionality）被执行，这是一种副作用。主功能应该放在一个main()函数中。

在Python中，pydoc及单元测试要求模块必须是可导入的。代码应该在执行主程序前总是检查 if \_\_name\_\_ == '\_\_main\_\_' ，这样当模块被导入时主程序就不会被执行。

若使用absl模块, 则使用 app.run :

from absl import app  
...  
  
def main(argv):  
 # process non-flag arguments  
 ...  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(main)

否则，使用：

def main():  
 ...  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

所有的顶级代码在模块导入时都会被执行。要小心不要去调用函数，创建对象，或者执行那些不应该在使用pydoc时执行的操作。

* 类型注释（略）

类型注释具体解释查看[Google 开源项目风格指南](https://zh-google-styleguide.readthedocs.io/en/latest/google-python-styleguide/python_style_rules/" \l "section-17)。

注释（Notes）：

① Python Package Idnex, PyPI，为Python编程语言的软件库，可以安装发布的Python库（包）或者发布包文件（<https://pypi.org/>）。

② Google 开源项目风格指南（<https://zh-google-styleguide.readthedocs.io/en/latest/google-python-styleguide/python_style_rules/#section-2>）；同时少量参考“Python PEP-8编码风格指南中文版”（<https://alvin.red/2017/10/07/python-pep-8/>）。

③ Spyder，免费开源的Python交互式解释器，由科学家、工程师和数据分析师设计并为其服务。具有科学软件包的数据分析探索，交互执行，深度检验和数据可视化等能力，包括高级编辑、分析，调试和剖析等功能。（<https://www.spyder-ide.org/>）。

参考文献（References）:

[1] Robert C. Martin.Clean Code: A Handbook of Agile Software Craftsmanship[M]. U.S. Prentice Hall. August 11, 2008.