Data Science and Advanced Programming — Lecture 3c

Programming Style

Simon Scheidegger
Department of Economics, University of Lausanne, Switzerland

September 29th, 2025 | 12:30 - 16:00 | Internef 263

Python Style Guide

Google Style Guides: http://google.github.io/styleguide/

- Every major open-source software project has its own style guide: a set of conventions (sometimes arbitrary) about how to write code for that project.
- It is much easier to understand a large code base when all the code in it is in a consistent style.
- Programming style is commonly used to enhance readability.

PEP 8 — Style Guide for Python Code

- Many Python programmers tend to follow a commonly agreed style guide known as **PEP8 (Python Enhancement Proposal)**.
- There are tools designed to automate PEP8 compliance.
- → `https://www.python.org/dev/peps/pep-0008/`
- *Beautiful is better than ugly*
- *Explicit is better than implicit*
- *Simple is better than complex*
- *Complex is better than complicated*
- *Readability counts*

A Foolish Consistency is the Hobgoblin of Little Minds

Guido van Rossum — benevolent dictator of Python:

> *One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code.*
>
> *"Readability counts".*

Code Layout

- Use 4 spaces per indentation level.
- Continuation lines should align wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a hanging indent.
  - When using a hanging indent the following should be considered: - there should be no arguments on the first line
  - further indentation should be used to clearly distinguish itself as a continuation line.

Yes

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
# Add 4 spaces (an extra level of indentation) to distinguish
arguments from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

No

```
# Arguments on first line forbidden when not using vertical alignment.
foo = long_function_name(var_one, var_two,
    var_three, var_four)
# Further indentation required as indentation is not distinguishable.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Optional

The 4-space rule is optional for continuation lines.

```
# Hanging indents *may* be indented to other than 4 spaces.
foo = long_function_name (
    var_one, var_two,
    var_three, var_four)
```

if-statement

- When the conditional part of an if-statement is long enough to require that it be written across multiple lines, it's worth noting that the combination of a two character keyword (i.e. if), plus a single space, plus an opening parenthesis creates a natural 4 space indent for the subsequent lines of the multiline conditional.
- This can produce a visual conflict with the indented suite of code nested inside the if-statement, which would also naturally be indented to 4 spaces.
- This PEP takes no explicit position on how (or whether) to further visually distinguish such conditional lines from the nested suite inside the if-statement.

if-statement

- When the conditional part of an if-statement is long enough to require that it be written across multiple lines, it's worth noting that the combination of a two character keyword (i.e. if), plus a single space, plus an opening parenthesis creates a natural 4 space indent for the subsequent lines of the multiline conditional.
- This can produce a visual conflict with the indented suite of code nested inside the if-statement, which would also naturally be indented to 4 spaces.
- This PEP takes no explicit position on how (or whether) to further visually distinguish such conditional lines from the nested suite inside the if-statement.

if-statement

- When the conditional part of an if-statement is long enough to require that it be written across multiple lines, it's worth noting that the combination of a two character keyword (i.e. if), plus a single space, plus an opening parenthesis creates a natural 4 space indent for the subsequent lines of the multiline conditional.
- This can produce a visual conflict with the indented suite of code nested inside the if-statement, which would also naturally be indented to 4 spaces.
- This PEP takes no explicit position on how (or whether) to further visually distinguish such conditional lines from the nested suite inside the if-statement.

No Extra Indentation

```
# No extra indentation.
if (this_is_one_thing and
    that_is_another_thing) :
    do_something( )

# Add a comment, which will provide some distinction in editors
# supporting syntax highlighting.
if (this_is_one_thing and
    that_is_another_thing):
    # since both conditions are true, we can frobnicate.
    do_something( )

# Add some extra indentation on the conditional continuation line.
if (this_is_one_thing
        and that_is_another_thing):
    do_something( )
```

Brackets I

- The closing brace/bracket/parenthesis on multi-line constructs may either line up under the first non-whitespace character of the last line of list, as in:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

Brackets II

- or it may be lined up under the first character of the line that starts the multiline construct, as in:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

Tabs or Spaces?

- Spaces are the preferred indentation method.
- Tabs should be used solely to remain consistent with code that is already indented with tabs.
- Python 3 disallows mixing the use of tabs and spaces for indentation.
- Python 2 code indented with a mixture of tabs and spaces should be converted to using spaces exclusively.
- → When invoking the Python 2 command line interpreter with the -t option, it issues warnings about code that illegally mixes tabs and spaces. When using -tt these warnings become errors. These options are highly recommended!

Tabs or Spaces?

- Spaces are the preferred indentation method.
- Tabs should be used solely to remain consistent with code that is already indented with tabs.
- Python 3 disallows mixing the use of tabs and spaces for indentation.
- Python 2 code indented with a mixture of tabs and spaces should be converted to using spaces exclusively.
- → When invoking the Python 2 command line interpreter with the -t option, it issues warnings about code that illegally mixes tabs and spaces. When using -tt these warnings become errors. These options are highly recommended!

Tabs or Spaces?

- Spaces are the preferred indentation method.
- Tabs should be used solely to remain consistent with code that is already indented with tabs.
- Python 3 disallows mixing the use of tabs and spaces for indentation.
- Python 2 code indented with a mixture of tabs and spaces should be converted to using spaces exclusively.
- → When invoking the Python 2 command line interpreter with the -t option, it issues warnings about code that illegally mixes tabs and spaces. When using -tt these warnings become errors. These options are highly recommended!

Tabs or Spaces?

- Spaces are the preferred indentation method.
- Tabs should be used solely to remain consistent with code that is already indented with tabs.
- Python 3 disallows mixing the use of tabs and spaces for indentation.
- Python 2 code indented with a mixture of tabs and spaces should be converted to using spaces exclusively.
- $\rightarrow$ When invoking the Python 2 command line interpreter with the -t option, it issues warnings about code that illegally mixes tabs and spaces. When using -tt these warnings become errors. These options are highly recommended!

Maximum Line Length I

- Limit all lines to a maximum of 79 characters.
- For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters (from FORTRAN).
- The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces.
- Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation.

Maximum Line Length II

- Backslashes may still be appropriate at times.
- For example, long, multiple with-statements cannot use implicit continuation, so backslashes are acceptable:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
    open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

Should a Line Break Before or After a Binary Operator?

- Following the tradition from mathematics usually results in more readable code:

```
# Yes: easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

- In Python code, it is permissible to break before or after a binary operator, as long as the convention is consistent locally.

Blank Lines

- Surround top-level function and class definitions with two blank lines.
- Method definitions inside a class are surrounded by a single blank line.
- Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).
- Use blank lines in functions, sparingly, to indicate logical sections.
- Python accepts the control-L (i.e. ^L) form feed character as whitespace.

Imports

- Imports should usually be on separate lines: `Yes: import os`
  `import sys`
  `No: import sys, os`
- It's okay to say this though: `from subprocess import Popen, PIPE`
- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

Imports — Order I

- Imports should be grouped in the following order:
  1. Standard library imports.
  2. Related third party imports.
  3. Local application/library specific imports.
- You should put a blank line between each group of imports.
- Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on `sys.path`):

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

Imports — Order II

- However, explicit relative imports are an acceptable alternative to absolute imports, especially when dealing with complex package layouts where using absolute imports would be unnecessarily verbose:

```
from . import sibling
from . sibling import example
```

- Standard library code should avoid complex package layouts and always use absolute imports.
- Implicit relative imports should never be used and have been removed in Python 3.

Module Level Dunder Names

- Module level "dunders" (i.e. names with two leading and two trailing underscores) such as __all__, __ __author__, version__, etc. should be placed after the module docstring but before any import statements except from ___future__ imports.
- Python mandates that future-imports must appear in the module before any other code except docstrings:

```
"""This is the example module.
This module does stuff.
"""
from __future__ import barry_as_FLUFL
__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'Cardinal Biggles'
import os
import sys
```

String Quotes

- In Python, single-quoted strings and double-quoted strings are the same.
- This PEP8 does not make a recommendation for this. Pick a rule and stick to it. When a string contains single or double quote characters, however, use the other one to avoid backslashes in the string. It improves readability.
- For triple-quoted strings, always use double quote characters

Whitespace in Expressions and Statements

- Avoid extraneous whitespace in the following situations:
- Immediately inside parentheses, brackets or braces.
- Between a trailing comma and a following close parenthesis.
- Immediately before a comma, semicolon, or colon:

Whitespace

- However, in a slice the colon acts like a binary operator, and should have equal amounts on either side (treating it as the operator with the lowest priority).
- In an extended slice, both colons must have the same amount of spacing applied. Exception: when a slice parameter is omitted, the space is omitted.

```
# Yes:
ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]
ham[lower:upper], ham[lower:upper:], ham[lower::step]
ham[lower+offset : upper+offset]
ham[: upper_fn(x) : step_fn(x)], ham[:: step_fn(x)]
ham[lower + offset : upper + offset]
# No:
ham[lower + offset:upper + offset]
ham[1: 9], ham[1 :9], ham[1:9 :3]
ham[lower : : upper]
ham[ : upper]
```

Whitespace — other Recommendations

- Avoid trailing whitespace anywhere. Because it's usually invisible, it can be confusing: e.g. a backslash followed by a space and a newline does not count as a line continuation marker.
- Some editors don't preserve it and many projects (like CPython itself) have pre-commit hooks that reject it.
- Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (`+=`, `-=` etc.), comparisons (`==`, `>`, `<`, `!=`,`<>`,`<=`, `>=` , `in`, `not in`, `is`, `is not`), Booleans (`and`, `or`, `not`).
- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies).
- Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

Whitespace — other Recommendations

```
# Yes:
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)

# No:
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

Comments

- Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!
- Comments should be complete sentences. The first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).
- Block comments generally consist of one or more paragraphs built out of complete sentences, with each sentence ending in a period.
- You should use two spaces after a sentence-ending period in multi- sentence comments, except after the final sentence.
- Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

Block Comments

- Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code.
- Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).
- Paragraphs inside a block comment are separated by a line containing a single #.

Inline Comments

- Use inline comments sparingly.
- An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.
- Inline comments are unnecessary and in fact distracting if they state the obvious.
- Don't do this:

Documentation Strings

- Write docstrings for all public modules, functions, classes, and methods.
- Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the def line.
- Note that most importantly, the "'m" that ends a multiline docstring should be on a line by itself:

```
"""Return a foobang
Optional plotz says to frobnicate the bizbaz first.
"""
```

Naming Conventions

- The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent – nevertheless, here are the currently recommended naming standards.
- New modules and packages (including third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

Overriding Principle

- Names that are visible to the user as public parts of the API (Application Programming Interface) should follow conventions that reflect usage rather than implementation.

Descriptive: Naming Styles

- There are a lot of different naming styles. It helps to be able to recognize what naming style is being used, independently from what they are used for.
- The following naming styles are commonly distinguished:

- `b` (single lowercase letter)
- `B` (single uppercase letter)
- `lowercase`
- `lower_case_with_underscores`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (or CapWords, or CamelCase – so named because of the bumpy look of its letters). This is also sometimes known as StudlyCaps.
- Note: When using acronyms in CapWords, capitalize all the letters of the acronym. Thus `HTTPServerError` is better than `HttpServerError`.
- `mixedCase` (differs from `CapitalizedWords` by initial lowercase character!)
- `Capitalized_Words_With_Underscores` (ugly!)

Prescriptive: Naming Conventions

Names to Avoid

- Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.
- In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead.

Package and Module Names

- Modules should have short, all-lowercase names.
- Underscores can be used in the module name if it improves readability.
  Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.
- When an extension module written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. `_socket`).

Class Names

- Class names should normally use the CapWords convention.
- The naming convention for functions may be used instead in cases where the interface is documented and used primarily as a callable.
- Note that there is a separate convention for builtin names: most builtin names are single words (or two words run together), with the CapWords convention used only for exception names and builtin constants.

Type Variable Names

- Names of type variables should normally use CapWords preferring short names: T, AnyStr, Num.
- It is recommended to add suffixes `_co` or `_contra` to the variables used to declare covariant or contravariant behavior correspondingly:

```
from typing import TypeVar
VT_co = TypeVar('VT_co', covariant=True)
KT_contra = TypeVar('KT_contra', contravariant=True)
```

Programming Recommendations I

- Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, Cython, Psyco, and such).
- For example, do not rely on CPython's efficient implementation of inplace string concatenation for statements in the form `a+=b` or `a=a+b`.
- This optimization is fragile even in CPython (it only works for some types) and isn't present at all in implementations that don't use refcounting.
- In performance-sensitive parts of the library, the `.join()` form should be used instead.
- This will ensure that concatenation occurs in linear time across various implementations.

Programming Recommendations II

- Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier.

```
# Yes:
def f(x): return 2*x
# No:
f = lambda x: 2*x
```

- The first form means that the name of the resulting function object is specifically ' `f` ' instead of the generic '`<lambda>`'.
- This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement (i.e., that it can be embedded inside a larger expression).

You got the idea...