Coding Styles for Python

Prof. Dr. Rüdiger Weis

TFH Berlin

The Zen of Python

```
Python 2.4.2 (#2, Sep 30 2005, 21:19:01)
[GCC 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu8)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
```

The Zen of Python (II)

```
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Style Guide for Python Code

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

```
PEP:  8
Title:  Style Guide for Python Code
Version:  43264
Last-Modified:  2006-03-23 21:13:19 +0100 (Thu, 23 Mar 2006)
Author:  Guido van Rossum <guido at python.org>,
         Barry Warsaw <barry at python.org>
Status:  Active
Type:  Informational
Created:  05-Jul-2001
Post-History:  05-Jul-2001

This document gives coding conventions for the Python code
comprising the standard library in the main Python distribution.
```

Readability counts

Readability counts

"One of Guido's key insights is that code is read much more often than it is written."

Code Lay-out

- Use 4 spaces per indentation level.
- Never mix tabs and spaces.
- Limit all lines to a maximum of 79 characters.
- For flowing long blocks of text (docstrings or comments), limiting the length to 72 characters is recommended.
- Code in the core Python distribution should aways use the ASCII or Latin-1 encoding (a.k.a. ISO-8859-1).

Blank Lines

- Separate top-level function and class definitions with two blank lines.
- Method definitions inside a class are separated 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.

Imports

- Imports should usually be on separate lines
- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.
- 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.
- Always use the absolute package path for all imports.

Whitespace in Expressions and Statements

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces.

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

- Immediately before a comma, semicolon, or colon:

```
Yes: if x == 4: print x, y; x, y = y, x
No:  if x == 4 : print x , y ; x , y = y , x
```

- Immediately before the open parenthesis that starts the argument list of a function call:

```
Yes: spam(1)
No:  spam (1)
```

Whitespace in Expressions and Statements (II)

Avoid extraneous whitespace in the following situations:

- Immediately before the open parenthesis that starts an indexing or slicing:

```
Yes: dict['key'] = list[index]
No:  dict ['key'] = list [index]
```

Whitespace in Expressions and Statements (III)

Avoid extraneous whitespace in the following situations:

- More than one space around an assignment (or other) operator to align it with another.

```
Yes:
    x = 1
    y = 2
    long_variable = 3
No:
    x             = 1
    y             = 2
    long_variable = 3
```

Whitespace in Expressions and Statements (IV)

- 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).

Whitespace in Expressions and Statements (V)

- Use spaces around arithmetic operators:

```
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)
```

Whitespace in Expressions and Statements (VI)

- Don't use spaces around the '=' sign when used to indicate a keyword argument or a default parameter value.

Yes:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

No:

```
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

Whitespace in Expressions and Statements (VII)

- Compound statements (multiple statements on the same line) are generally discouraged.

```
Yes:
    if foo == 'blah':
        do_blah_thing()
    do_one()
    do_two()
    do_three()
Rather not:
    if foo == 'blah': do_blah_thing()
    do_one(); do_two(); do_three()
```

Whitespace in Expressions and Statements (VIII)

- While sometimes it's okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

```
Definitely not:
    if foo == 'blah': do_blah_thing()
    else: do_non_blah_thing()
    try: something()
    finally: cleanup()
    do_one(); do_two(); do_three(long, argument,
                                 list, like, this)
      if foo == 'blah': one(); two(); three()
```

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. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).
- Use inline comments sparingly.

Documentation Strings

Conventions for writing good documentation strings (a.k.a. "docstrings") are immortalized in PEP 257 [3].

- Write docstrings for all public modules, functions, classes, and methods.
- PEP 257 describes good docstring conventions. Note that most importantly, the """ that ends a multiline docstring should be on a line by itself, and preferably preceded by a blank line.
- For one liner docstrings, it's okay to keep the closing """ on the same line.

Version Bookkeeping

If you have to have Subversion, CVS, or RCS crud in your source file, do it as follows.

```
__version__ = "$Revision: 43264 $"
# $Source$
```

These lines should be included after the module's docstring, before any other code, separated by a blank line above and below.

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.

Descriptive: Naming Styles

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)
- mixedCase (differs from CapitalizedWords by initial lowercase character!)
- Capitalized_Words_With_Underscores

Python Conventions

In addition, the following special forms using leading or trailing underscores are recognized :

- **_single_leading_underscore**:
  weak "internal use" indicator.
  - E.g. "from M import *" does not import objects whose name starts with an underscore.
- **single _trailing _underscore _**:
  used by convention to avoid conflicts with Python keyword, e.g.

  ```
  Tkinter.Toplevel(master, class_='ClassName')
  ```

Python Conventions (II)

In addition, the following special forms using leading or trailing underscores are recognized :

- **__double_leading_underscore**:
  when naming a class attribute, invokes name mangling (inside class FooBar, __boo becomes _FooBar__boo; see below).
- **__double_leading_and_trailing_underscore__**:
  "magic" objects or attributes that live in user-controlled namespaces. E.g. __init__, __import__ or __file__.
  - *Never invent such names; only use them as documented!*

Names to Avoid

- Never use the characters
  - l (lowercase letter el),
  - O (uppercase letter oh)
  - 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.

Module Names

`modulename`

Modules should have short, lowercase names, without underscores.

- 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).
- Like modules, Python packages should have short, all-lowercase names, without underscores.

Class Names

`KlassenName`

class names use the CapWords convention.

- Classes for internal use have a leading underscore in addition.

  `_KlassenName`

- For exception classes you should use the suffix "Error" on your exception names.

Function Names

`funtionsname`

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

Function and method arguments

- Always use 'self' for the first argument to instance methods.
- Always use 'cls' for the first argument to class methods.

Method Names and Instance Variables

`methodename`

Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.

- Use one leading underscore only for non-public methods and instance variables.
  To avoid name clashes with subclasses, use two leading underscores to invoke Python's name mangling rules.

Name Mangeling

`__mangle_me`

Python mangles names using two leading underscores with the class name: if class Foo has an attribute named __a, it cannot be accessed by Foo.__a.

- *An insistent user could still gain access by calling Foo._Foo__a.*
- Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

Public Attributes

- Public attributes should have no leading underscores.
- If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling.
- For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods.

Designing for inheritance

- If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores.
  - This invokes Python's name mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name.

References

1. PEP 7, Style Guide for C Code, van Rossum
2. `http://www.python.org/doc/essays/styleguide.html`
3. PEP 257, Docstring Conventions, Goodger, van Rossum
4. `http://www.wikipedia.com/wiki/CamelCase`
5. Barry's GNU Mailman style guide
   `http://barry.warsaw.us/software/STYLEGUIDE.txt`
6. PEP 20, The Zen of Python
7. PEP 328, Imports: Multi-Line and Absolute/Relative