Documentation and tests

Jules KOZOLINSKY

2/11/16

Contents

Chapter 1

Documentation

1.1 Convention

This is only a brief summary of the PEP 8 – Style Guide for Python Code.

1.1.1 Code lay-out

Indentation

Use 4 spaces per indentation level. You can convert tabs to (four) spaces in the options of your favorite editor (e.g. for Emacs or Vim)
See "Yes and No" examples in following link : Indentation Rules

Maximum Line Length

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. Long lines can be broken over multiple lines by wrapping expressions in parentheses but backslashes may still be appropriate at times.

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).

1.1.2 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
  ```

- 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]
```

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

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

1.1.3 Documentation Strings

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition.
All modules should normally have docstrings, and all functions and classes exported by a module should also have docstrings.
There are two forms of docstrings: one-liners and multi-line docstrings.

One-line Docstrings

One-liners are for really obvious cases. They should really fit on one line. For example:

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
    ...
```

Multi-line Docstrings

Multi-line docstrings consist of a summary line just like a one-line docstring, followed by a blank line, followed by a more elaborate description. The summary line may be used by automatic indexing tools; it is important that it fits on one line and is separated from the rest of the docstring by a blank line. The summary line may be on the same line as the opening quotes or on the next line. The entire docstring is indented the same as the quotes at its first line (see example below).

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
    ...
```

Docstring for a function or method It should summarize its behavior and document its arguments, return value(s), side effects, exceptions raised, and restrictions on when it can be called (all if applicable). Optional arguments should be indicated.

Docstring for a class It should summarize its behavior and list the public methods and instance variables.

1.1.4 Naming Conventions

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

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.

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.

Class Names

Class names should normally use the CapWords convention.

Type variable names

Names of type variables introduced in PEP 484 should normally use CapWords preferring short names: `T` , `AnyStr` , `Num`

Exception Names

Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix "Error" on your exception names (if the exception actually is an error).

Global Variable Names and Functions

Function names and global variable 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

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.

Constants

Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include `MAX_OVERFLOW` and `TOTAL` .

1.1.5 Enforce type annotations on public methods

1.2 Doxygen

Automatically done with Travis, you don't need to read this in detail.
Doxygen is a standard tool for generating documentation from **annotated** Python sources.
It can generate an on-line documentation browser in HTML and an off-line reference manual in LaTeX from a set of documented source files.
Doxygen is developed under Mac OS X and Linux, but is set-up to be highly portable. As a result, it runs on most other Unix flavors as well. Furthermore, executables for Windows are available.

1.2.1 Installation

Usefull informations in following link : *Installation procedure for Linux and Windows*. (I use *homebrew* for MacOS)

1.2.2 Documenting the code

Please use docstrings (""") to comment all of your files, fonctions, classes and methods. (See subsection 1.1.3 about docstrings conventions). Comments with # will not appear in the documentation.
For example :

```
"""Documentation for this module.

More details.
"""
# just use the # for small comments (it will not appear in the doc)

def func():
    """Documentation for a function.

    More details.
    """
    pass
class PyClass:
    """Documentation for a class.

    More details.
    """

    def __init__(self):
        """The constructor."""
        self.mem_var = 0

    def py_method(self):
        """Documentation for a method."""
        pass

    def py_method_2(self):
        """Documentation for a method."""
        pass
```

1.2.3 Compile and create the documentation

The first time you use Doxygen, you have to create a configuration file :

```
doxygen -g
```

To run doxygen, you simply have to execute :

```
doxygen Doxyfile
```

1.2.4 Read the documentation

You can either compile the LaTeX documentation (using the Makefile in the *latex* folder) or use the on-line HTML documentation (open *index.html*)

1.2.5 Edit the *.gitignore* file

Add the following lines to the *.gitignore* file :

```
/latex
/html
Doxyfile
```

Chapter 2

Tests

2.1 Testing Tools

2.1.1 Static Analysis with Pylint

(Outil recommandé par le CERN)
Pylint is a tool that checks for errors in Python code, tries to enforce a coding standard and looks for bad code smells. This is similar but nevertheless different from what pychecker provides, especially since pychecker explicitly does not bother with coding style. Pylint will display a number of messages as it analyzes the code, as well as some statistics about the number of warnings and errors found in different files. The messages are classified under various categories such as errors and warnings.

2.1.2 Design by Contracts with PyContracts

A permanent change to Python to support Design by Contracts was proposed in PEP-316, but deferred.

Programming contracts extends the language to include invariant expressions for classes and modules, and pre- and post-condition expressions for functions and methods.
These expressions (contracts) are similar to assertions: they must be true or the program is stopped, and run-time checking of the contracts is typically only enabled while debugging. Contracts are higher-level than straight assertions and are typically included in documentation.

Use **PyContracts**.

2.1.3 Unit testing infrastructure with unittest

Recommandation
Development Tools of the Python Standard Library

- doctest — Test interactive Python examples
- unittest — Unit testing framework
- unittest.mock — mock object library

2.1.4 Fuzz Testing with Hypothesis

Fuzz testing or fuzzing is a software testing technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer program. The program is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks. Fuzzing is a form of random testing commonly used to test for security problems in software or computer systems.
Use Hypothesis.

2.1.5 Code Coverage Tool with Coverage.py

Coverage.py is a tool for measuring code coverage of Python programs. It monitors your program, noting which parts of the code have been executed, then analyzes the source to identify code that could have been executed but was not.

2.1.6 Stress

- Flood a server with requests
- Execution with constrained resources (memory, disk)
- Create latency (network)

2.1.7 GUI Testing Tools

TestText

2.2 Procédures facilitant les test

2.2.1 Hooks on commit

Les hooks permettent de lancer automatiquement ma suite de tests avant de commiter.

2.2.2 Continuous integration

Jenkins, Travis CI

References

PEP-0008
PEP-0257