Python Coding Standard Guidelines

CONTENTS

TABLES

1 DOCUMENT DETAILS

1.1 Document History

Version	Author		Reviewer		Approver	
	Name	**Date (DD-MM-YYYY)**	**Name**	**Date (DD-MMM-YYYY)**	**Name**	**Date (DD-MM-YYYY)**
Baseline 1.0	Mumtaz Banu	25-Nov-2016	Prayag Vakharia	25-Nov-2016	Bijal Chudgar	8-Dec-2016

Version	Description of Change
Baseline 1.0	Created initial version

Table 1: Document History

1.2 Definition, Acronyms and Abbreviations

Definition/Acronym/Abbreviation	Description
PEP	Python Enhancement Proposal, a standard for documentation within the Python community**.**

Table 2: Definition, Acronyms and Abbreviations

1.3 References

No.	Document	Version	Remarks
1.	https://www.python.org/dev/peps/pep-0008	-	-
2**.**	https://en.wikibooks.org/wiki/Python_Programming/Source_Documentation_and_Comments		
3	http://eikke.com/python-gotcha/		
4	http://docs.ckan.org/en/latest/contributing/python.html		

5	https://developer.lsst.io/docs/rst_styleguide.html#rst-formatting-guideline		
6	https://developer.lsst.io/coding/python_style_guide.html#comments		

Table 3: References

2. INTRODUCTION

2.1. Purpose of the Document

This document, primarily through reference to established Python "community" documents, specifies standards and best practices for coding

2.2. Scope of the Document

Coding conventions make your code easier to read and debug. Thus, for both our and your benefit, we would like you to use the conventions outlined in this document for your python code.

This document assumes a working knowledge of the Python programming language.

3. GUIDANCE

3.1. Naming Convention

3.1.1. General

- Avoid using names that are too general or too wordy. Strike a good balance between the two.
- lower_with_underscores : Uses only lower case letters and connects multiple words with underscores.
- UPPER_WITH_UNDERSCORES : Uses only upper case letters and connects multiple words with underscores.
- CapitalWords : Capitalize the beginning of each letter in a word; no underscores.
- Bad:data_structure,my_list,info_map, dictionary_for_the_purpose_of_storing_data_representing_word_definitions
- Good: user_profile, menu_options, word_definitions
- Avoid naming things as “O”, “l”, or “I” to create confusion. In some fonts, these characters are indistinguishable from the numerals one `1` and zero `0`. When tempted to use `l`, use `L` instead.
- When using CamelCase names, capitalize all letters of an abbreviation (e.g. HTTPServer)

3.1.2. Packages

- Package names should be all lower case
- When multiple words are needed, an underscore should separate them
- It is usually preferable to stick to 1 word names.
  Ex. mypackage or my_pyhton_package

3.1.3. Modules

- Module names should be all lower case
- When multiple words are needed, an underscore should separate them
- It is usually preferable to stick to 1 word names
  Ex. hello.py or hello_world.py

3.1.4. Classes

- Class names should follow the UpperCaseCamelCase convention
- Python’s built-in classes, however are typically lowercase words
- Exception classes should end in “Error” .
  Ex. MyClass

3.1.5. Global (module-level) Variables

- Global variables should be all lowercase
- Words in a global variable name should be separated by an underscore
  Ex. my_global_var

3.1.6. Instance Variables

- Instance variable names should be all lower case
- Words in an instance variable name should be separated by an underscore
- Non-public instance variables should begin with a single underscore
- If an instance name needs to be mangled, two underscores may begin its name

```
class Sample(object):
    x = 100
  _a = 1
  __b = 11
  def __init__(self, value):
  self.y = value
  self._c = 'private'
  self.__d = 'more private'
  z = 300
```

In this example:

- x is class variable,
- _a is private class variable (by naming convention),
- __b is private class variable (mangled by interpreter),
- y is instance variable,
- _c is private instance variable (by naming convention),
- __d is private instance variable (mangled by interpreter),
- z is local variable within scope of __init__ method.

3.1.7. Methods

- Method names should be all lower case
- Words in an method name should be separated by an underscore
- Non-public method should begin with a single underscore
- If a method name needs to be mangled, two underscores may begin its name
  Ex . def displayCount(self):
  print "Total Employee %d" % Employee.empCount

3.1.8. Method Arguments

- Instance methods should have their first argument named ‘self’.
- Class methods should also have their first argument named ‘self .
  Ex. class Employee:

```
    empCount = 0

    def __init__(self, name, salary): #Instance method arguments
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):  #class method argument and you can add other argument as well
        print "Total Employee %d" % Employee.empCount

    def displayEmployee(self): #class method argument
        print "Name : ", self.name,  ", Salary: ", self.salary
```

3.1.9. Functions

- Function names should be all lower case
- Words in a function name should be separated by an underscore.
  Ex. def sum(a,b):
  return a + b
  or
  def sum_of_int(a,b)

```
        retutn a + b
```

3.1.10. Constants

- Constant names must be fully capitalized
- Words in a constant name should be separated by an underscore
  Ex. MY_CONSTANT_VAR

3.2. File Naming and Organization

From a file system perspective, a module is a file ending with .py and a package is a folder containing modules and (nested) packages again. Python recognizes a folder as a package if it contains a __init__.py file.

A file structure like that

```
some/
   __init__.py
   foofoo.py
   thing/
      __init__.py
      barbar.py
```

defines the package some, which has a module foofoo and a nested package thing, which again has a module barbar. However, when using packages and modules, you don't really distinguish these two types:

```
import some

some.dothis() # dothis is defined in 'some/__init__.py'

import some.foofoo # <- module
import some.thing   # <- package
```

Please follow PEP8 when selecting naming your packages/modules (i.e. use lower-case names).

3.3. Formatting and Indentation

3.3.1. Whitespace

Indentation:

• Lines should be indented with a multiple of 4 spaces depending on indent level
• Hanging indents, code that is a continuation of the line above, should be indented to the next level.
• Use spaces exclusively, no tabs!

Blank Lines

• Leave 2 blank lines between class definitions and module-level functions
• Leave 1 blank line between methods in a class
• Use blank lines as needed in functions, methods, and modules to visually split up logical blocks of code.

Spaces In Code

- Surround binary operators with a space on each side
- Do not include spaces around '=' when used to indicate a default argument or keyword argument
- In all other cases, extraneous spaces are frowned upon

3.3.2. Imports

- Imports should occur at the top of the module, after any module docstring
- Standard imports (those beginning with "import") should each be on a separate line. Do not combine imports into one line.
- "From ... import ..."-style imports may be combined together on one line if possible
- Wildcard imports should only be used when absolutely necessary, otherwise only import the modules to be used

3.3.3. Code Blocks

- Do not use parenthesis in the condition for a code block header unless it would be otherwise appropriate to use parenthesis around that condition
- Do not use any single line code blocks. Even those with a single statement in the body should occupy multiple lines.

3.4 Comment and Documentation

3.4.1 Comment Guidelines

The following guidelines are from PEP 8, written by Guido van Rossum.

General:

- 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!).
- If a comment is short, the period at the end can be omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period.
- You should use two spaces after a sentence-ending period.
- When writing English, Strunk and White applies.
- 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.

Inline Comment

- 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:

```
x = x + 1  # Increment x
```

But sometimes, this is useful:

```
x = x + 1  # Compensate for border
```

3.4.2. Documentation

But what if you just want to know how to use a function, class, or method? You could add comments before the function, but comments are inside the code, so you would have to pull up a text editor and view them that way. But you can't pull up comments from a C extension, so that is less than ideal. You could always write a separate text file with how to call the functions, but that would mean that you would have to remember to update that file. If only there was a mechanism for being able to embed the documentation and get at it easily...

Fortunately, Python has such a capability. Documentation strings (or docstrings) are used to create easily-accessible documentation. You can add a docstring to a function, class, or module by adding a string as the first indented statement. For example:

```
#!/usr/bin/env python
# docstringexample.py

"""Example of using documentation strings."""

class Knight:
    """
    An example class.

    Call spam to get bacon.
    """

    def spam(eggs="bacon"):
        """Prints the argument given."""
        print(eggs)
```

The convention is to use triple-quoted strings, because it makes it easier to add more documentation spanning multiple lines.

To access the documentation, you can use the `help` function inside a Python shell with the object you want help on, or you can use the `pydoc` command from your system's shell. If we were in the directory where docstringexample.py lives, one could enter `pydoc docstringexample` to get documentation on that module.