PEP Index > PEP 257 -- Docstring Conventions

PEP:	257
Title:	Docstring Conventions
Version:	a7b2b2d5b1d4
Last-Modified:	**2014-02-28 09:53:33 -0800 (Fri, 28 Feb 2014)**
Author:	David Goodger <goodger at python.org>, Guido van Rossum <guido at python.org>
Discussions-To:	**doc-sig at python.org**
Status:	Active
Type:	Informational
Content-Type:	**text/x-rst**
Created:	29-May-2001
Post-History:	13-Jun-2001

Contents

Abstract

This PEP documents the semantics and conventions associated with Python docstrings.

Rationale

The aim of this PEP is to standardize the high-level structure of docstrings: what they should contain, and how to say it (without touching on any markup syntax within docstrings). The PEP contains conventions, not laws or syntax.

> "A universal convention supplies all of maintainability, clarity, consistency, and a foundation for good programming habits too. What it doesn't do is insist that you follow it against your will. That's Python!"
>
> —Tim Peters on comp.lang.python, 2001-06-16

If you violate these conventions, the worst you'll get is some dirty looks. But some software (such as the **Docutils [4]** docstring processing system **[1] [2]**) will be aware of the conventions, so following them will get you the best results.

Specification

What is a Docstring?

A docstring is a string literal that occurs as the first statement in a module, function, class, or method

definition. Such a docstring becomes the `__doc__` special attribute of that object.

All modules should normally have docstrings, and all functions and classes exported by a module should also have docstrings. Public methods (including the `__init__` constructor) should also have docstrings. A package may be documented in the module docstring of the `__init__.py` file in the package directory.

String literals occurring elsewhere in Python code may also act as documentation. They are not recognized by the Python bytecode compiler and are not accessible as runtime object attributes (i.e. not assigned to `__doc__`), but two types of extra docstrings may be extracted by software tools:

1. String literals occurring immediately after a simple assignment at the top level of a module, class, or `__init__` method are called "attribute docstrings".
2. String literals occurring immediately after another docstring are called "additional docstrings".

Please see **PEP 258**, "Docutils Design Specification" **[2]**, for a detailed description of attribute and additional docstrings.

XXX Mention docstrings of 2.2 properties.

For consistency, always use `"""triple double quotes"""` around docstrings. Use `r"""raw triple double quotes"""` if you use any backslashes in your docstrings. For Unicode docstrings, use `u"""Unicode triple-quoted strings"""`.

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
    ...
```

Notes:

- Triple quotes are used even though the string fits on one line. This makes it easy to later expand it.
- The closing quotes are on the same line as the opening quotes. This looks better for one-liners.
- There's no blank line either before or after the docstring.
- The docstring is a phrase ending in a period. It prescribes the function or method's effect as a command ("Do this", "Return that"), not as a description; e.g. don't write "Returns the pathname ...".
- The one-line docstring should NOT be a "signature" reiterating the function/method parameters (which can be obtained by introspection). Don't do:

  ```
  def function(a, b):
      """function(a, b) -> list"""
  ```

  This type of docstring is only appropriate for C functions (such as built-ins), where introspection

is not possible. However, the nature of the *return value* cannot be determined by introspection, so it should be mentioned. The preferred form for such a docstring would be something like:

```
def function(a, b):
    """Do X and return a list."""
```

(Of course "Do X" should be replaced by a useful description!)

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).

Insert a blank line before and after all docstrings (one-line or multi-line) that document a class -- generally speaking, the class's methods are separated from each other by a single blank line, and the docstring needs to be offset from the first method by a blank line; for symmetry, put a blank line between the class header and the docstring. Docstrings documenting functions or methods generally don't have this requirement, unless the function or method's body is written as a number of blank-line separated sections -- in this case, treat the docstring as another section, and precede it with a blank line.

The docstring of a script (a stand-alone program) should be usable as its "usage" message, printed when the script is invoked with incorrect or missing arguments (or perhaps with a "-h" option, for "help"). Such a docstring should document the script's function and command line syntax, environment variables, and files. Usage messages can be fairly elaborate (several screens full) and should be sufficient for a new user to use the command properly, as well as a complete quick reference to all options and arguments for the sophisticated user.

The docstring for a module should generally list the classes, exceptions and functions (and any other objects) that are exported by the module, with a one-line summary of each. (These summaries generally give less detail than the summary line in the object's docstring.) The docstring for a package (i.e., the docstring of the package's `__init__.py` module) should also list the modules and subpackages exported by the package.

The docstring for a function or method 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. It should be documented whether keyword arguments are part of the interface.

The docstring for a class should summarize its behavior and list the public methods and instance variables. If the class is intended to be subclassed, and has an additional interface for subclasses, this interface should be listed separately (in the docstring). The class constructor should be documented in the docstring for its `__init__` method. Individual methods should be documented by their own docstring.

If a class subclasses another class and its behavior is mostly inherited from that class, its docstring should mention this and summarize the differences. Use the verb "override" to indicate that a subclass method replaces a superclass method and does not call the superclass method; use the verb "extend" to indicate that a subclass method calls the superclass method (in addition to its own behavior).

Do not use the Emacs convention of mentioning the arguments of functions or methods in upper case in running text. Python is case sensitive and the argument names can be used for keyword arguments,

so the docstring should document the correct argument names. It is best to list each argument on a separate line. For example:

```
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
    ...
```

Unless the entire docstring fits on a line, place the closing quotes on a line by themselves. This way, Emacs' `fill-paragraph` command can be used on it.

Handling Docstring Indentation

Docstring processing tools will strip a uniform amount of indentation from the second and further lines of the docstring, equal to the minimum indentation of all non-blank lines after the first line. Any indentation in the first line of the docstring (i.e., up to the first newline) is insignificant and removed. Relative indentation of later lines in the docstring is retained. Blank lines should be removed from the beginning and end of the docstring.

Since code is much more precise than words, here is an implementation of the algorithm:

```
def trim(docstring):
    if not docstring:
        return ''
    # Convert tabs to spaces (following the normal Python rules)
    # and split into a list of lines:
    lines = docstring.expandtabs().splitlines()
    # Determine minimum indentation (first line doesn't count):
    indent = sys.maxint
    for line in lines[1:]:
        stripped = line.lstrip()
        if stripped:
            indent = min(indent, len(line) - len(stripped))
    # Remove indentation (first line is special):
    trimmed = [lines[0].strip()]
    if indent < sys.maxint:
        for line in lines[1:]:
            trimmed.append(line[indent:].rstrip())
    # Strip off trailing and leading blank lines:
    while trimmed and not trimmed[-1]:
        trimmed.pop()
    while trimmed and not trimmed[0]:
        trimmed.pop(0)
    # Return a single string:
    return '\n'.join(trimmed)
```

The docstring in this example contains two newline characters and is therefore 3 lines long. The first and last lines are blank:

```
def foo():
    """
    This is the second line of the docstring.
```

```
    """
```

To illustrate:

```
>>> print repr(foo.__doc__)
'\n    This is the second line of the docstring.\n    '
>>> foo.__doc__.splitlines()
['', '    This is the second line of the docstring.', '    ']
>>> trim(foo.__doc__)
'This is the second line of the docstring.'
```

Once trimmed, these docstrings are equivalent:

```
def foo():
    """A multi-line
    docstring.
    """

def bar():
    """
    A multi-line
    docstring.
    """
```

References and Footnotes

[1] **PEP 256**, Docstring Processing System Framework, Goodger (**http://www.python.org/dev/peps/pep-0256/**)

[2] ***(1, 2)*** **PEP 258**, Docutils Design Specification, Goodger (**http://www.python.org/dev/peps/pep-0258/**)

[3] Guido van Rossum, Python's creator and Benevolent Dictator For Life.

[4] http://docutils.sourceforge.net/

[5] (http://www.python.org/dev/peps/pep-0008/)

[6] http://www.python.org/sigs/doc-sig/

Copyright

Acknowledgements

The "Specification" text comes mostly verbatim from the **Python Style Guide [5]** essay by Guido van Rossum.

This document borrows ideas from the archives of the Python **Doc-SIG [6]**. Thanks to all members past and present.