Example Google Style Python Docstrings

See also

Example NumPy Style Python Docstrings

Download: example_google.py

```python
# -*- coding: utf-8 -*-
"""Example Google style docstrings.

This module demonstrates documentation as specified by the `Google Python
Style Guide`_. Docstrings may extend over multiple lines. Sections are created
with a section header and a colon followed by a block of indented text.

Example:
    Examples can be given using either the ``Example`` or ``Examples``
    sections. Sections support any reStructuredText formatting, including
    literal blocks::

        $ python example_google.py

Section breaks are created by resuming unindented text. Section breaks
are also implicitly created anytime a new section starts.

Attributes:
    module_level_variable1 (int): Module level variables may be documented in
        either the ``Attributes`` section of the module docstring, or in an
        inline docstring immediately following the variable.

        Either form is acceptable, but the two should not be mixed. Choose
        one convention to document module level variables and be consistent
        with it.

Todo:
    * For module TODOs
    * You have to also use ``sphinx.ext.todo`` extension

.. _Google Python Style Guide:
   http://google.github.io/styleguide/pyguide.html

"""

module_level_variable1 = 12345

module_level_variable2 = 98765
"""int: Module level variable documented inline.

The docstring may span multiple lines. The type may optionally be specified
on the first line, separated by a colon.
"""


def function_with_types_in_docstring(param1, param2):
    """Example function with types documented in the docstring.

    `PEP 484`_ type annotations are supported. If attribute, parameter, and
    return types are annotated according to `PEP 484`_, they do not need to be
    included in the docstring:

    Args:
        param1 (int): The first parameter.
        param2 (str): The second parameter.

    Returns:
        bool: The return value. True for success, False otherwise.

    .. _PEP 484:
        https://www.python.org/dev/peps/pep-0484/

    """


def function_with_pep484_type_annotations(param1: int, param2: str) -> bool:
    """Example function with PEP 484 type annotations.

    Args:
        param1: The first parameter.
        param2: The second parameter.
    Returns:
        The return value. True for success, False otherwise.

    """


def module_level_function(param1, param2=None, *args, **kwargs):
    """This is an example of a module level function.

    Function parameters should be documented in the ``Args`` section. The name
    of each parameter is required. The type and description of each parameter
    is optional, but should be included if not obvious.

    If \*args or \*\*kwargs are accepted,
    they should be listed as ``*args`` and ``**kwargs``.

    The format for a parameter is::

        name (type): description
            The description may span multiple lines. Following
            lines should be indented. The "(type)" is optional.

            Multiple paragraphs are supported in parameter
            descriptions.

    Args:
        param1 (int): The first parameter.
        param2 (:obj:`str`, optional): The second parameter. Defaults to None.
            Second line of description should be indented.
        *args: Variable length argument list.
        **kwargs: Arbitrary keyword arguments.

    Returns:
        bool: True if successful, False otherwise.

        The return type is optional and may be specified at the beginning of
        the ``Returns`` section followed by a colon.

        The ``Returns`` section may span multiple lines and paragraphs.
        Following lines should be indented to match the first line.

        The ``Returns`` section supports any reStructuredText formatting,
        including literal blocks::

            {
                'param1': param1,
                'param2': param2
            }

    Raises:
        AttributeError: The ``Raises`` section is a list of all exceptions
            that are relevant to the interface.
        ValueError: If `param2` is equal to `param1`.

    """
    if param1 == param2:
        raise ValueError('param1 may not be equal to param2')
    return True


def example_generator(n):
    """Generators have a ``Yields`` section instead of a ``Returns`` section.

    Args:
        n (int): The upper limit of the range to generate, from 0 to `n` - 1.

    Yields:
        int: The next number in the range of 0 to `n` - 1.

    Examples:
        Examples should be written in doctest format, and should illustrate how
        to use the function.

        >>> print([i for i in example_generator(4)])
        [0, 1, 2, 3]

    """
    for i in range(n):
        yield i


class ExampleError(Exception):
    """Exceptions are documented in the same way as classes.

    The __init__ method may be documented in either the class level
    docstring, or as a docstring on the __init__ method itself.

    Either form is acceptable, but the two should not be mixed. Choose one
    convention to document the __init__ method and be consistent with it.

    Note:
        Do not include the `self` parameter in the ``Args`` section.

    Args:
        msg (str): Human readable string describing the exception.
        code (:obj:`int`, optional): Error code.

    Attributes:
        msg (str): Human readable string describing the exception.
        code (int): Exception error code.

    """

    def __init__(self, msg, code):
        self.msg = msg
        self.code = code


class ExampleClass(object):
    """The summary line for a class docstring should fit on one line.

    If the class has public attributes, they may be documented here
    in an ``Attributes`` section and follow the same formatting as a
    function's ``Args`` section. Alternatively, attributes may be documented
    inline with the attribute's declaration (see __init__ method below).

    Properties created with the ``@property`` decorator should be documented
    in the property's getter method.

    Attributes:
        attr1 (str): Description of `attr1`.
        attr2 (:obj:`int`, optional): Description of `attr2`.

    """

    def __init__(self, param1, param2, param3):
        """Example of docstring on the __init__ method.

        The __init__ method may be documented in either the class level
        docstring, or as a docstring on the __init__ method itself.

        Either form is acceptable, but the two should not be mixed. Choose one
        convention to document the __init__ method and be consistent with it.

        Note:
            Do not include the `self` parameter in the ``Args`` section.

        Args:
            param1 (str): Description of `param1`.
            param2 (:obj:`int`, optional): Description of `param2`. Multiple
                lines are supported.
            param3 (:obj:`list` of :obj:`str`): Description of `param3`.
        """
        self.attr1 = param1
        self.attr2 = param2
        self.attr3 = param3  #: Doc comment *inline* with attribute

        #: list of str: Doc comment *before* attribute, with type specified
        self.attr4 = ['attr4']

        self.attr5 = None
        """str: Docstring *after* attribute, with type specified."""

    @property
    def readonly_property(self):
        """str: Properties should be documented in their getter method."""
        return 'readonly_property'

    @property
    def readwrite_property(self):
        """:obj:`list` of :obj:`str`: Properties with both a getter and setter
        should only be documented in their getter method.

        If the setter method contains notable behavior, it should be
        mentioned here.
        """
        return ['readwrite_property']

    @readwrite_property.setter
    def readwrite_property(self, value):
        value

    def example_method(self, param1, param2):
        """Class methods are similar to regular functions.

        Note:
            Do not include the `self` parameter in the ``Args`` section.

        Args:
            param1: The first parameter.
            param2: The second parameter.

        Returns:
            True if successful, False otherwise.

        """
        return True

    def __special__(self):
        """By default special members with docstrings are not included.

        Special members are any methods or attributes that start with and
        end with a double underscore. Any special member with a docstring
        will be included in the output, if
        ``napoleon_include_special_with_doc`` is set to True.

        This behavior can be enabled by changing the following setting in
        Sphinx's conf.py::

            napoleon_include_special_with_doc = True

        """
        pass

    def __special_without_docstring__(self):
        pass

    def _private(self):
        """By default private members are not included.

        Private members are any methods or attributes that start with an
        underscore and are *not* special. By default they are not included
        in the output.
        This behavior can be changed such that private members *are* included
        by changing the following setting in Sphinx's conf.py::

            napoleon_include_private_with_doc = True

        """
        pass

    def _private_without_docstring(self):
        pass
```