code_style Documentation

Lukas Turcani

Oct 14, 2019

Style Guides

CHAPTER 1

Python Style Guidelines

Style should follow PEP8 and PEP257 in cases where these guidelines do not state a different preference.

Table of Contents

1.1 Line length.

Line length should be at most 71 visible characters, so that it is 72 with newlines. This ensures that all lines obey PEP8 and that all lines are the same length.

1.2 Numbers.

Numbers should always use separators.

```
number1 = 1_000
number2 = 100_000
number3 = 1_000_000
```

1.3 Lists.

`list`, `set`, `dict` objects should be defined on a single line, if they fit.

```
some_list = [1, 34, 423, 12, 4, 10]
some_set = {1, 45, 233, 549}
some_dict = {'a': 10, 'b': 20, 'c': 10_000}
```

They should be defined on a line by line basis otherwise, with opening and closing brackets on their own line. A comma should be used after the last element too.

```
some_list = [
    some_very_long_list_element,
    some_other_very_long_list_element,
    some_even_very_very_long_list_element,
    and_so_on,
]

some_set = {
    some_very_long_set_element,
    some_other_very_long_set_element,
    some_even_very_very_long_set_element,
    and_so_on,
}

some_dict = {
    'some_long_key': some_very_long_set_element,
    'another_long_key': some_other_very_long_set_element,
    'key': some_even_very_very_long_set_element,
    'also_a_key': and_so_on,
}
```

1.4 Comprehensions.

Comprehensions should be done on a single line, if they fit.

```
list1 = [2*x for x in range(20)]
list2 = [2*x for x in range(20) if 2*x % 4 == 0]
set1 = {2*x for x in range(20)}
dict1 = {key: value for key, value in zip(range(10), (10, 20))}
```

If the the comprehension does not fit on a single line, try placing the opening and closing brackets on separate lines.

```
some_very_long_variable_name = [
    some_element for some_element in some_long_container_name
]

some_other_very_long_variable_name = {
    key: value for key, value in zip(range(10), range(10, 20))
}
```

If the comprehension still does not fit, split it so that each Python keyword begins on a new line, with the exception of `for` and `in` which should be placed on the same line, if they fit.

```
some_very_long_variable_name = [
    some_very_long_function_name(some_very_long_element_name)
    for some_very_long_element_name in some_long_container_name
    if some_very_long_element_name == 20
    and some_very_long_element_name % 2 == 1
]

some_other_very_long_variable_name = {
    key_name: value_name
    for key_name, value_name in zip(range(10), range(10, 20))
}
```

1.5 Function calls.

Function calls, with 3 or fewer parameters, may be done on a single line without any parameter names.

```
some_variable = some_function(1, 2, 3)
```

They can also be done with parameter names, if they fit.

```
some_variable = some_function(param1=12, param2=32, param3=21)
```

Functions with more than 3 parameters must be called with each parameter specified by name and on a separate line.

```
some_variable = some_function(
    param_name1=1,
    param_name2=2,
    param_name3=3,
    param_name4='4th',
)
```

This keeps everything readable and ensures that parameters are not sensitive to order.

1.6 Function and method definitions.

Functions and methods should be defined so that they are on a single line, if they fit.

```
def some_function(param1, param2, param3, keyword_param=10):
    return 12
```

If the function or method does not fit on a single line it should be split so that each parameter is on a separate line. The closing `):` is at the same indentation as the `def` keyword.

```
def some_function(
    some_very_long_parameter_name1,
    some_very_long_parameter_name2,
    some_keyword_parameter=12,
):
    return 12
```

1.7 Strings.

Strings should use ', unless ' needs to be a character in the string.

```
some_string = 'abcdefg'
```

1.8 Multi-line strings.

Opening and closing brackets should be on separate lines and spaces should be at the end of the line, not the start.

```
some_multiline_string = (
    'this is a '
    'multiline '
    'string.'
)
```

1.9 Docstrings.

Docstrings should open and close with three double quotes “”“. The first line of a docstring should always be on the line below the opening “”“. The closing “”“ always needs to be on its own line and preceded by an empty line. The docstring must be followed by an empty line.

```
def some_fn():
    """
    Do something.

    """

    foo()
    return bar(12)
```

1.10 Docstring markup.

Docstrings should use numpy style formatting. An explanation of the markup is provided here, and this is considered required reading.

The examples below also explain how the markup is used but it is not a substitute for reading the linked documents.

```python
def some_fn(param1, param2, param3, param4, param5=12):
    """
    Do something.

    There are two general rules. Use a period in front of a name
    to create a hyperlink to where that name is documented,
    for example :class:`.SomeType`. Use a ~ at  the start of a
    name to remove any preceding names in the compiled
    documentation, for example
    :class:`~.module.submodule.SomeType` will only display
    "SomeType" in the compiled documentation. The modules are
    added in front of the name to resolve any ambiguity in
    name resolution, for example if two classes in your library
    have the same name but are found in different submodules.
    Periods should only be used in front of names which are part
    of your own library.

    When literal code values are specified they should be between
    two backticks, for example if I was to say that the default
    value of `param5` is ``12``. Argument names, such as
    `param5` or `param1` are surrounded by a single backtick.

    Simple code expressions, such as ``UserDefinedType(12) + 12``
    are also surrounded by two backticks. When multi-line or
    complicated code expressions are to be described, they
    should be performed in a code block.

    .. code-block:: python

        # This is a multi-line code example.
        variable_name = 'one two three'
        for i in range(10):
            print(i**2)

    Sometimes, when talking about an attribute in a class, we
    want to make it clear to the reader what attribute we are
    referring as well as the class, so the
    :attr:`.SomeTypeName.attr_name` syntax is used. However, as we
    continue to refer to the attribute, it is
    unnecessary to continue stating the class explicitly in the
    compiled documentation, so the :attr:`~.SomeTypeName.attr_name`
    syntax can be used. The ~ can be added to
    remove the class name from the compiled documentation. Use
    the ~ when appropriate to maximize the readability of the
    compiled documentation.

    Parameters
    ----------
    param1 : :class:`int`
        When the type being described is a builtin type, such as
        :class:`int` or :class:`str`, it should have the form
        the :class:`type_name`.

    param2 : :class:`.UserDefinedType`
        When the type being described refers to a user defined
        type within the same library, make sure the type name
        is preceded by a period.
```

(continues on next page)

(continued from previous page)

```
    param3 : :class:`other.UserDefinedType`
        When the type being described refers to a user defined
        type in another library, state the name of the library
        and name the type. If the type is in a submodule of the
        library DO NOT name the submodules. For example, we may
        want to document the use of a :class:`numpy.ndarray`.

    param4 : :class:`~.mod.submod.subsubmod.UserDefinedType`
        Sometimes there may be two user defined types in the
        same library with the same name but defined in different
        submodules of the library. If we want to refer to form a
        hyperlink, we have to add the the necessary submodule
        names in front of the user defined type's name, so that
        it is unambiguous which type is being referred to. The
        ~ means that the submodule names will be removed from the
        compiled documentation. This means that the ~ can be
        removed or kept, depending on what is less confusing to
        the reader.

    param5 : :class:`int`, optional
        When the parameter is optional, it should be stated as
        optional after the type declaration.

    Returns
    -------
    :class:`int`
        Returns a number.

    """

    return 12

class SomeType:
    """
    An example type.

    Note that class docstrings should not document the attributes
    defined in a parent class, unless there is something different
    about them.

    Attributes
    ----------
    alpha : :class:`int`
        Attributes defined within this class are referred to by
        the :attr:`alpha` or :attr:`beta` syntax.

    beta : :class:`str`
        Attributes of a different class are referred to by
        preceding with a period and the type name, for example
        :attr:`.SomeOtherType.attribute_name`.

    """

    def __init__(self):
        self.alpha = 1
        self.beta = 'b'
```

CHAPTER 2

C++ Style Guidelines

2.1 Function Declarations

Check the sidebar for style guides for the different languages.