

# Pocoo Styleguide

The Pocoo styleguide is the styleguide for all Pocoo Projects, including Flask. This styleguide is a requirement for Patches to Flask and a recommendation for Flask extensions.

In general the Pocoo Styleguide closely follows [PEP 8](#) with some small differences and extensions.

## General Layout

Indentation:

4 real spaces. No tabs, no exceptions.

Maximum line length:

79 characters with a soft limit for 84 if absolutely necessary. Try to avoid too nested code by cleverly placing *break*, *continue* and *return* statements.

Continuing long statements:

To continue a statement you can use backslashes in which case you should align the next line with the last dot or equal sign, or indent four spaces:

```
this_is_a_very_long(function_call, 'with many parameters') \
    .that_returns_an_object_with_an_attribute


MyModel.query.filter(MyModel.scalar > 120) \
    .order_by(MyModel.name.desc()) \
    .limit(10)
```

If you break in a statement with parentheses or braces, align to the braces:

```
this_is_a_very_long(function_call, 'with many parameters',
                    23, 42, 'and even more')
```

For lists or tuples with many items, break immediately after the opening brace:

```
items = [
    'this is the first', 'set of items', 'with more items',
    'to come in this line', 'like this'
]
```

 v: 1.1.x ▼

Blank lines:

Top level functions and classes are separated by two lines, everything else by one. Do not use too many blank lines to separate logical segments in code. Example:

```
def hello(name):
    print 'Hello %s!' % name

def goodbye(name):
    print 'See you %s.' % name

class MyClass(object):
    """This is a simple docstring"""

    def __init__(self, name):
        self.name = name

    def get_annoying_name(self):
        return self.name.upper() + '!!!!111'
```

## Expressions and Statements

General whitespace rules:

- No whitespace for unary operators that are not words (e.g.: `-`, `~` etc.) as well on the inner side of parentheses.
- Whitespace is placed between binary operators.

Good:

```
exp = -1.05
value = (item_value / item_count) * offset / exp
value = my_list[index]
value = my_dict['key']
```

Bad:

```
exp = - 1.05
value = ( item_value / item_count ) * offset / exp
value = (item_value/item_count)*offset/exp
value=( item_value/item_count ) * offset/exp
value = my_list[ index ]
value = my_dict ['key']
```

 v: 1.1.x ▼

Yoda statements are a no-go:

Never compare constant with variable, always variable with constant:

Good:

```
if method == 'md5':  
    pass
```

Bad:

```
if 'md5' == method:  
    pass
```

Comparisons:

- against arbitrary types: `==` and `!=`
- against singletons with `is` and `is not` (eg: `foo is not None`)
- never compare something with `True` or `False` (for example never do `foo == False`, do `not foo` instead)

Negated containment checks:

use `foo not in bar` instead of `not foo in bar`

Instance checks:

`isinstance(a, C)` instead of `type(A) is C`, but try to avoid instance checks in general. Check for features.

## Naming Conventions

- Class names: `CamelCase`, with acronyms kept uppercase (`HTTPWriter` and not `HttpWriter`)
- Variable names: `lowercase_with_underscores`
- Method and function names: `lowercase_with_underscores`
- Constants: `UPPERCASE_WITH_UNDERSCORES`
- precompiled regular expressions: `name_re`

Protected members are prefixed with a single underscore. Double underscores are reserved for mixin classes.

On classes with keywords, trailing underscores are appended. Clashes with builtins are allowed and **must not** be resolved by appending an underline to the variable name. If the function needs to access a shadowed builtin, rebind the builtin to a different name instead.

Function and method arguments:

- class methods: `cls` as first parameter

- instance methods: `self` as first parameter
- lambdas for properties might have the first parameter replaced with `x` like in `display_name = property(lambda x: x.real_name or x.username)`

## Docstrings

Docstring conventions:

All docstrings are formatted with reStructuredText as understood by Sphinx. Depending on the number of lines in the docstring, they are laid out differently. If it's just one line, the closing triple quote is on the same line as the opening, otherwise the text is on the same line as the opening quote and the triple quote that closes the string on its own line:

```
def foo():
    """This is a simple docstring"""

def bar():
    """This is a longer docstring with so much information in there
    that it spans three lines. In this case the closing triple quote
    is on its own line.
    """
```

Module header:

The module header consists of a utf-8 encoding declaration (if non ASCII letters are used, but it is recommended all the time) and a standard docstring:

```
# -*- coding: utf-8 -*-
"""
    package.module
    ~~~~~

    A brief description goes here.

    :copyright: (c) YEAR by AUTHOR.
    :license: LICENSE_NAME, see LICENSE_FILE for more details.
"""
```

Please keep in mind that proper copyrights and license files are a requirement for approved Flask extensions.

## Comments

Rules for comments are similar to docstrings. Both are formatted with reStructuredText. If a comment is used to document an attribute, put a colon after the opening pound sign (#):

```
class User(object):  
    #: the name of the user as unicode string  
    name = Column(String)  
    #: the sha1 hash of the password + inline salt  
    pw_hash = Column(String)
```