Flake8 – a tool for PEP8 style guide enforcement

http://flake8.pycqa.org/en/latest/

install on a particular version of Python

```
python3 -m pip install flake8
flake8 project_name
```

```
(forge_dev) mo-2122b:forge mo$ flake8 ../forge
../forge/build/lib/mdf_forge/toolbox.py:17:18: E261 at least two spaces before inline comment
../forge/build/lib/mdf_forge/toolbox.py:17:19: E262 inline comment should start with '# ’
../forge/build/lib/mdf_forge/toolbox.py:34:1: E266 too many leading '#' for block comment
../forge/build/lib/mdf_forge/toolbox.py:137:92: W291 trailing whitespace
../forge/build/lib/mdf_forge/toolbox.py:240:33: E128 continuation line under-indented for visual indent
../forge/build/lib/mdf_forge/toolbox.py:254:1: E303 too many blank lines (3)
../forge/build/lib/mdf_forge/toolbox.py:258:1: E302 expected 2 blank lines, found 3
../forge/build/lib/mdf_forge/toolbox.py:281:80: E251 unexpected spaces around keyword / parameter equals
../forge/build/lib/mdf_forge/toolbox.py:329:1: E266 too many leading '#' for block comment
../forge/build/lib/mdf_forge/toolbox.py:493:46: E712 comparison to False should be 'if cond is False:' or 'if not cond:’
../forge/build/lib/mdf_forge/toolbox.py:497:10: E261 at least two spaces before inline comment
../forge/build/lib/mdf_forge/toolbox.py:501:9: E265 block comment should start with '# ’
../forge/build/lib/mdf_forge/toolbox.py:505:51: E712 comparison to False should be 'if cond is False:' or 'if not cond:’
../forge/build/lib/mdf_forge/toolbox.py:690:1: W391 blank line at end of file
../forge/mdf_forge/forge.py:425:56: E251 unexpected spaces around keyword / parameter equals
../forge/mdf_forge/forge.py:432:101: E501 line too long (108 > 100 characters)
.........
```

Indentation, line-length & code wrapping
- Always use 4 spaces for indentation (don't use tabs)
- Write in ASCII in Python 2 and UTF-8 in Python 3
- Max line-length: 72 characters
- Always indent wrapped code for readablility

Imports
- Don't use wildcards
- Try to use absolute imports over relative ones
- When using relative imports, be explicit (with .)
- Don't import multiple packages per line

Whitespace and newlines
- 2 blank lines before top-level function and class definitions
- 1 blank line before class method definitions
- Use blank lines in functions sparingly
- Avoid extraneous whitespace
- Don't use whitespace to line up assignment operators (=, :)
- Spaces around = for assignment
- No spaces around = for default parameter values
- Spaces around mathematical operators, but group them sensibly
- Multiple statements on the same line are discouraged

Comments
- Keep comments up to date - incorrect comments are worse than no comments

Comments (cont.)
- Write in whole sentences
- Try to write in "Strunk & White" English
- Use inline comments sparingly & avoid obvious comments
- Each line of block comments should start with "# "
- Paragraphs in block comments should be separated by a line with a single "#"
- All public functions, classes and methods should have docstrings
- Docstrings should start and end with """
- Docstring one-liners can be all on the same line
- In docstrings, list each argument on a separate line
- Docstrings should have a blank line before the final """

Naming conventions
- Class names in CapWords
- Method, function and variables names in lowercase_with_underscores
- Private methods and properties start with __double_underscore
- "Protected" methods and properties start with _single_underscore
- If you need to use a reserved word, add a _ to the end (e.g. class_)
- Always use self for the first argument to instance methods
- Always use cls for the first argument to class methods
- Never declare functions using lambda (f = lambda x: 2*x)

```
../forge/tests/test_forge.py:557:5: F841 local variable 'res3' is assigned to but never used
../forge/tests/test_forge.py:560:20: E712 comparison to True should be 'if cond is True:' or 'if cond:'
```

```python
def test_forge_match_years(capfd):
    # One year of data/results
    f1 = forge.Forge()
    years1 = ["2015"]
    res1 = f1.match_years(years1).search(limit=10)
    assert res1 != []
    assert check_field(res1, "mdf.year", 2015) == 0

    # Multiple years
    f2 = forge.Forge()
    years2 = [2015, "2011"]
    # match_all=False (2011 OR 2015)
    res2 = f2.match_years(years2, inclusive=False).search()
    assert check_field(res2, "mdf.year", 2011) == 2

    # Wrong input
    f3 = forge.Forge()
    years3 = ["20x5"]
    res3 = f3.match_years(years3, inclusive=False).search() # noqa
    out, err = capfd.readouterr()
    msg_err = "Year is not a valid input" in out
    assert msg_err == True # noqa
```

I’ve used few `noqa` comments ignoring errors in tests.

1) It wants me to use `is` to compare singletons. So the statement `assert x == True` should be written `assert x is True` instead, in accordance with PEP8.
2) `res3` is indeed not used but I think it is needed for code style compatibility.

Options

--count	Print the total number of errors.
--diff	Use the unified diff provided on standard in to only check the modified files and report errors included in the diff.
--exclude=<patterns>	Provide a comma-separated list of glob patterns to exclude from checks.
--filename=<patterns>	Provide a comma-separate list of glob patterns to include for checks.
--format=<format>	Select the formatter used to display errors to the user.
--hang-closing	Toggle whether pycode style should enforce matching the indentation of the opening bracket's line.
--ignore=<errors>	Specify a list of codes to ignore. The list is expected to be comma-separated,
--max-line-length=<n>	Set the maximum length that any line (with some exceptions) may be. Exceptions include lines that are either strings or comments which are entirely URLs. This defaults to: 79
--select=<errors>	Specify the list of error codes you wish Flake8 to report.
--disable-noqa	Report all errors, even if it is on the same line as a # NOQA comment. # NOQA can be used to silence messages on specific lines. Sometimes, users will want to see what errors are being silenced without editing the file. This option allows you to see all the warnings, etc. reported.
--show-source	Print the source code generating the error/warning in question.
--statistics	Count the number of occurrences of each error/warning code and print a report.
--exit-zero	Force Flake8 to use the exit status code 0 even if there are errors.
--jobs=<n>	Specify the number of subprocesses that Flake8 will use to run checks in parallel.
--output-file=<path>	Redirect all output to the specified file.
--tee	Also print output to stdout if output-file has been configured.
--config=<config>	Provide a path to a config file that will be the only config file read and used. This will cause Flake8 to ignore all other config files that exist.
--isolated	Ignore any config files and use Flake8 as if there were no config files found.
--builtins=<builtins>	Provide a custom list of builtin functions, objects, names, etc.
--doctests	Enable PyFlakes syntax checking of doctests in docstrings.
--benchmark	Collect and print benchmarks for this run of Flake8.

Error / Violation Codes

```
../forge/tests/test_forge.py:557:5: F841 local variable 'res3' is assigned to but never used
../forge/tests/test_forge.py:560:20: E712 comparison to True should be 'if cond is True:' or 'if cond:'
```

Ignoring Violations (here codes starting with E1, etc.)

```
flake8 --ignore=E1,E23,W503 path/to/files/
```

List all of the errors we want to ignore in the code: `# noqa: E731,E123`

Using Plugins For Fun and Profit

Flake8 is useful on its own but a lot of its popularity is due to its extensibility. Our community has developed plugins that augment Flake8’s behavior. The developers of these plugins often have some style they wish to enforce.

Python Enhancement Proposals (PEP) 8 compliance

Style Guide for Python Code

“**A foolish consistency is the hobgoblin of little minds**, adored by little statesmen and philosophers and divines. With consistency a great soul has simply nothing to do. He may as well concern himself with his shadow on the wall. Speak what you think now in hard words, and to-morrow speak what to-morrow thinks in hard words again, though it contradict every thing you said to-day. — 'Ah, so you shall be sure to be misunderstood.' — Is it so bad, then, to be misunderstood? Pythagoras was misunderstood, and Socrates, and Jesus, and Luther, and Copernicus, and Galileo, and Newton, and every pure and wise spirit that ever took flesh. To be great is to be misunderstood.”

Ralph Waldo Emerson, Self-Reliance

The guidelines are intended to improve the readability of code and make it consistent across the wide spectrum of Python code.

However, know when to be inconsistent -- sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment.

Guido van Rossum

Indentation, line-length & code wrapping

- Always use 4 spaces for indentation (don't use tabs)
- Write in ASCII in Python 2 and UTF-8 in Python 3
- Max line-length: 72 characters
- Always indent wrapped code for readablility

Imports

- Don't use wildcards
- Try to use absolute imports over relative ones
- When using relative imports, be explicit (with .)
- Don't import multiple packages per line

Whitespace and newlines

- 2 blank lines before top-level function and class definitions
- 1 blank line before class method definitions
- Use blank lines in functions sparingly
- Avoid extraneous whitespace
- Don't use whitespace to line up assignment operators (=, :)
- Spaces around = for assignment
- No spaces around = for default parameter values
- Spaces around mathematical operators, but group them sensibly
- Multiple statements on the same line are discouraged

Comments

- Keep comments up to date - incorrect comments are worse than no comments

Comments (cont.)

- Write in whole sentences
- Try to write in "Strunk & White" English
- Use inline comments sparingly & avoid obvious comments
- Each line of block comments should start with "# "
- Paragraphs in block comments should be separated by a line with a single "#"
- All public functions, classes and methods should have docstrings
- Docstrings should start and end with """
- Docstring one-liners can be all on the same line
- In docstrings, list each argument on a separate line
- Docstrings should have a blank line before the final """

Naming conventions

- Class names in CapWords
- Method, function and variables names in lowercase_with_underscores
- Private methods and properties start with __double_underscore
- "Protected" methods and properties start with _single_underscore
- If you need to use a reserved word, add a _ to the end (e.g. class_)
- Always use self for the first argument to instance methods
- Always use cls for the first argument to class methods
- Never declare functions using lambda (f = lambda x: 2*x)

PEP-8 cheatsheet - http://wwd.ca/blog/2009/07/09/pep-8-cheatsheet/

```python
#! /usr/bin/env python
# -*- coding: utf-8 -*-
"""This module's docstring summary line.

This is a multi-line docstring. Paragraphs are separated with blank lines.
Lines conform to 79-column limit.

Module and packages names should be short, lower_case_with_underscores.

See http://www.python.org/dev/peps/pep-0008/ for more PEP-8 details and
http://wwd.ca/blog/2009/07/09/pep-8-cheatsheet/ for an up-to-date version
of this cheatsheet.
"""
import os
import sys

import some_third_party_lib
import some_other_third_party_lib

import your_local_stuff
import more_local_stuff
import dont_import_two, modules_in_one_line    # IMPORTANT!

_a_global_var = 2   # so it won't get imported by 'from foo import *'
_b_global_var = 3

A_CONSTANT = 'ugh.'

# 2 empty lines between top-level funcs + classes
def some_function():
    pass


class FooBar(object):
    """Write docstrings for ALL public classes, funcs and methods.

    Class and exception names are CapWords.
    """

    a = 2
    b = 4
    _internal_variable = 3
    class_ = 'foo'      # trailing underscore to avoid conflict with builtin

    # this will trigger name mangling to further discourage use from outside
    # this is also very useful if you intend your class to be subclassed, and
    # the children might also use the same var name for something else; e.g.
    # for simple variables like 'a' above. Name mangling will ensure that
    # *your* a and the children's a will not collide.
    __internal_var = 4

    # NEVER use double leading and trailing underscores for your own names
    __nooooooodontdoit__ = 0

    # don't call anything:
    l = 1
    O = 2
    I = 3

    # some examples of how to wrap code to conform to 79-columns limit:
    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if width == 0 and height == 0 and \
           color == 'red' and emphasis == 'strong' or \
           highlight > 100:
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                             (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)

        # empty lines within method to enhance readability; no set rule
        short_foo_dict = {'looooooooooooooooooooong_element_name': 'cat',
                          'other_element': 'dog'}

        long_foo_dict_with_many_elements = {
            'foo': 'cat',
            'bar': 'dog'
        }

    # 1 empty line between in-class def'ns
    def foo_method(self, x, y=None):
        """Method and function names are lower_case_with_underscores.

        Always use self as first arg.
        """
        if x == 4:          # x is blue <== USEFUL 1-liner comment
            x, y = y, x     # inverse x and y <== USELESS COMMENT
        dict['key'] = dict[index] = {x: 2, 'cat': 'not a dog'}
        c = (a + b) * (a - b)

    @classmethod
    def bar(cls):
        """Use cls!"""
        pass

# a 79-char ruler:
#234567891123456789212345678931234567894123456789512345678961234567897123456789
```