
flake8-aaa Documentation

Release 0.8.0

James Cooke

Feb 27, 2020

Contents

1	Flake8-AAA	3
1.1	Installation and usage	3
1.2	Arrange Act Assert	3
1.3	Compatibility	4
1.4	Resources	4

A linter for Python tests.

Flake8-AAA enforces simple formatting of your test suite making it more consistent and easier to grok, especially across teams.

1.1 Installation and usage

Flake8-AAA is a Flake8 plugin.

Install `flake8-aaa` with `pip`, which will also install `flake8`:

```
$ pip install flake8-aaa
```

Invoke Flake8 on your test suite, in this case in the `tests` directory:

```
$ flake8 tests
```

Errors returned by Flake8-AAA have the AAA code, for example:

```
tests/block/test_init.py:14:1: AAA02 multiple Act blocks found in test
```

1.2 Arrange Act Assert

Tests are linted against the [Arrange Act Assert](#) pattern.

TL;DR following the AAA pattern means tests are laid out like this:

```
def test():  
    """  
    __docstring__
```

(continues on next page)

(continued from previous page)

```
"""
<ARRANGE block> # set up of the system under test (SUT)

<ACT block> # perform a single action on the SUT

<ASSERT block> # check that the SUT changed as expected
```

For example:

```
def test(tmpdir):
    """
    Checker is able to parse provided file at load time
    """
    target_file = tmpdir.join('test.py')
    target_file.write('assert 1 + 2 == 3\n')
    tree = ast.parse(target_file.read())
    checker = Checker(tree, ['assert 1 + 2 == 3\n'], target_file.strpath)

    result = checker.load()

    assert result is None
    assert len(checker.tree.body) == 1
    assert type(checker.tree.body[0]) == ast.Assert
    assert len(checker.ast_tokens.tokens) == 8
```

More examples are in our test suite's “good” files.

1.3 Compatibility

- Pytest and unittest supported.
- Compatible with Black and yapf formatted code.
- Compatible with type-annotated code, checked with mypy.
- Current release works with the latest versions of Python 3 (3.6, 3.7 and 3.8). Older releases have support for older Pythons.

See the [Compatibility list](#) for more info.

1.4 Resources

- [Documentation on ReadTheDocs](#)
- [Package on PyPI](#)
- [Source code on GitHub](#)
- [Licensed on MIT](#)
- [Changelog](#)

1.4.1 Compatibility list

The stable version of Flake8-AAA is compatible with the following software.

Future versions will maintain this compatibility, unless an item is deprecated.

Python

Works with Python 3.

We fully test against the latest versions of Python 3 - currently that's 3.6, 3.7 and 3.8.

Python 3.5

Python 3.5 is supported up to v0.7.2

- [PyPI](#)
- [Documentation](#)
- [Github v0.4.0 tag](#)

Python 2

Python 2 is supported up to v0.4.0

- [PyPI](#)
- [Documentation](#)
- [Github v0.4.0 tag](#)

Flake8

Works with Flake8 version 3 and later.

We use the newer plugin system implemented in Flake8 v3. This dependency is not specified in `setup.py` because users may only want to use the command line interface.

Check that Flake8-AAA was installed correctly by asking `flake8` for its version signature:

```
$ flake8 --version
3.7.8 (aaa: 0.7.0, mccabe: 0.6.1, pycodestyle: 2.5.0, pyflakes: 2.1.1) CPython 3.6.7
↳ on Linux
```

The `aaa: 0.7.0` part of that output tells you Flake8 found this plugin.

Black

Tests formatted with latest Black version will pass.

The coding style used by Black can be viewed as a strict subset of PEP8.

Given that the AAA pattern is PEP8 compatible it makes sense that Flake8-AAA must work with test code that is formatted with PEP8 compatible formatters. Therefore we'll maintain Flake8-AAA to pass with AAA-style tests formatted with Black.

This compatibility is pinned by the test examples in the [examples/good/black directory](#) - we assert that these tests pass the latest version of Black's formatting and Flake8-AAA's linting.

Pytest

Pytest is fully supported.

To pin this compatibility we use the latest version of Pytest in the Flake8-AAA test suite and lint that test suite with Flake8-AAA (aka. dog fooding).

Unittest

Python unittest style is supported.

To pin this compatibility we include unittest-style tests in the [examples/good directory](#) -

1.4.2 Test discovery

The Flake8-AAA plugin is triggered for files that look to it like test modules - anything that does not look like a test module is skipped.

The following rules are applied by Flake8-AAA when discovering tests:

- The module's filename must start with "test_" and have been collected for linting by Flake8.
- Every function in the module that has a name that starts with "test" is checked.
- Test functions can be class methods.
- Test functions that contain only comments, docstrings or `pass` are skipped.

These rules are aimed to mirror pytest's default collection strategy as closely as possible.

If you find that Flake8-AAA is giving false positives (you have checks that you expected to fail, but they did not), then you should check that the plugin did not ignore or skip those tests which you expected to fail.

Note: Flake8-AAA does not check doctests.

Processing

For each test found, Flake8-AAA runs the following processes, most of which can be found in `Function.check_all()`.

Check for no-op

Skip test if it is considered "no-op" (`pass`, docstring, etc).

Mark blank lines

Mark all lines in the test that have no characters and are not part of a string. For example, the following snippet contains only one blank line (line 3 - in the middle of the list), the second at line 9 is part of a string and therefore not counted:

```
assert result == [
    1,
    2,
]
# Check on output
assert str(result) == "[
1,
2,
]"
```

Find Act node

Search the test for the Act node which will indicate the existence of an Act block. There are four recognised types of Act node:

marked_act Action is marked with Marked with # act comment:

```
do_thing() # act
```

pytest_raises Action is wrapped in `pytest.raises` context manager:

```
with pytest.raises(ValueError):
    do_thing()
```

result_assignment Simple `result = action`:

```
result = do_thing()
```

unittest_raises Action is wrapped in `unittest's assertRaises` context manager:

```
with self.assertRaises(ValueError):
    do_thing()
```

Build Act block

The Act node is used to build the Act block for the test. The Act block contains all parent nodes of the Act node up to the test function. For example:

```
def test():
    with mock.patch('thing.thinger') as mock_thinger: # < Act block first line
        with mock.patch('other_thing.thinger'):
            with pytest.raises(ValueError): # < Act node
                do_thing() # < Act block last line

    assert mock_thinger.call_count == 0
```

Build Arrange and Assert blocks

The Arrange block is created with all nodes in the test function that have a line number before the start of the Act block.

The Assert block is created with all nodes in the test function that have a line number after the end of the Act block.

Line-wise analysis

Each block updates a list of line markers for the test and line-wise analysis occurs to ensure that there is one and only one blank line between each block.

1.4.3 Rules and error codes

The rules applied by Flake8-AAA are from the [Arrange Act Assert pattern for Python developers](#).

Note: The rules applied by Flake8-AAA are only a subset of the rules and guidelines of the Arrange Act Assert pattern itself. Please see the [published guidelines for the pattern](#) and read these rules in the context of the definition there.

Note: Flake8-AAA works best with the following Flake8 rules enabled:

- E303 “too many blank lines”
 - E702 “Multiple statements on one line”
-

AAA01: no Act block found in test

An Act block is usually a line like `result =` or a check that an exception is raised. Flake8-AAA could not find an Act block in the indicated test function.

Resolution

Add an Act block to the test or mark a line that should be considered the action.

Even if the result of a test action is `None`, assign that result and pin it with a test:

```
result = action()
assert result is None
```

If you can not assign a `result`, then mark the end of the line considered the Act block with `# act` (case insensitive):

```
data['new_key'] = 1 # act
```

Code blocks wrapped in `pytest.raises()` and `unittest.assertRaises()` context managers are recognised as Act blocks.

AAA02: multiple Act blocks found in test

There must be one and only one Act block in every test but Flake8-AAA found more than one potential Act block. This error is usually triggered when a test contains more than one `result = statement` or more than one line marked `# act`. Multiple Act blocks create ambiguity and raise this error code.

Resolution

Split the failing test into multiple tests. Where there is complicated or reused set-up code then apply the DRY principle and extract the reused code into one or more fixtures.

AAA03: expected 1 blank line before Act block, found none

For tests that have an Arrange block, there must be a blank line between the Arrange and Act blocks, but Flake8-AAA could not find one.

This blank line creates separation between the arrangement and the action and makes the Act block easy to spot.

This rule works best with `pycodestyle`'s E303 rule enabled because it ensures that there are not multiple blank lines between the blocks.

Resolution

Add a blank line before the Act block.

AAA04: expected 1 blank line before Assert block, found none

For tests that have an Assert block, there must be a blank line between the Act and Assert blocks, but Flake8-AAA could not find one.

This blank line creates separation between the action and the assertions and makes the Act block easy to spot.

As with rule AAA03, this rule works best with E303 enabled.

Resolution

Add a blank line before the Assert block.

AAA05: blank line in block

The only blank lines in the test must be around the Act block making it easy to spot. Flake8-AAA found additional blank lines which break up the block's layout.

Resolution

Remove the blank line.

AAA99: collision when marking this line as NEW_CODE, was already OLD_CODE

This is an error code that is raised when Flake8 tries to mark a single line as occupied by two different types of block. It *should* never happen. The values for `NEW_CODE` and `OLD_CODE` are from the list of *Line markers*.

Resolution

Please open a [new issue](#) containing the output for the failing test as generated by the *Command line* tool.

You could hack around with your test to see if you can get it to work while waiting for someone to reply to your issue. If you're able to adjust the test to get it to work, that updated test would also be helpful for debugging.

1.4.4 Controlling Flake8-AAA

In code

Flake8-AAA can be controlled using some special comments in your test code.

Explicitly marking blocks

One can set the act block explicitly using the `# act` comment. This is necessary when there is no assignment possible.

Disabling Flake8-AAA selectively

When invoked via Flake8, Flake8 will filter any errors raised when lines are marked with the `# noqa` syntax. You can turn off all errors from Flake8-AAA by marking a line with `# noqa: AAA` and other Flake8 errors will still be returned.

If you just want to ignore a particular error, then you can use the more specific code and indicate the exact error to be ignored. For example, to ignore the check for a space before the Act block, we can mark the Act block with `# noqa: AAA03`:

```
def test():
    x = 1
    result = x + 1 # noqa: AAA03

    assert result == 2
```

Command line

Flake8-AAA has a simple command line interface to assist with development and debugging. Its goal is to show the state of analysed test functions, which lines are considered to be parts of which blocks and any errors that have been found.

Invocation, output and return value

With Flake8-AAA installed, it can be called as a Python module:

```
$ python -m flake8_aaa [test_file]
```

Where `[test_file]` is the path to a file to be checked.

The return value of the execution is the number of errors found in the file, for example:

```
$ python -m flake8_aaa test_example.py
-----+-----
1 DEF|def test():
2 ARR|   x = 1
3 ARR|   y = 1
   ^ AAA03 expected 1 blank line before Act block, found none
4 ACT|   result = x + y
5 BL |
6 ASS|   assert result == 2
-----+-----
      1 | ERROR
=====+=====
                FAILED with 1 ERROR
$ echo "$?"
1
```

And once the error above is fixed, the return value returns to zero:

```
$ python -m flake8_aaa test_example.py
-----+-----
1 DEF|def test():
2 ARR|   x = 1
3 ARR|   y = 1
4 BL |
5 ACT|   result = x + y
6 BL |
7 ASS|   assert result == 2
-----+-----
      0 | ERRORS
=====+=====
                PASSED!
$ echo "$?"
0
```

noqa and command line

The `# noqa` comment marker works slightly differently when Flake8-AAA is called on the command line rather than invoked through `flake8`. When called on the command line, to skip linting a test function, mark the function definition with `# noqa` on the same line as the `def`.

For example:

```
def test_to_be_ignored( # noqa
    arg_1,
    arg_2,
):
    ...
```

Line markers

Each test found in the passed file is displayed. Each line is annotated with its line number in the file and a marker to show how Flake8-AAA classified that line. Line markers are as follows:

ACT Line is part of the Act Block.

ARR Line is part of an Arrange Block.

ASS Line is part of the Assert Block.

BL Line is considered a blank line for layout purposes.

DEF Test function definition.

??? Unprocessed line. Flake8-AAA has not categorised this line.

1.4.5 Release checklist

Items to be completed before and after each release.

Pre-release steps

Given a new version called `x.y.z`:

- Create a branch for the new release. Usually called something like `bump-vx.y.z`.
- Update `__version__` in `__about__.py` with the new version number `'x.y.z'`.
- Update `CHANGELOG`.
 - Add a new subtitle below `Unreleased` after the note about latest documentation, in the format `x.y.z_`
`yyyy/mm/dd`, where `yyyy/mm/dd` is the reverse formatted date of the day the release is created.
 - Update the `.. _Unreleased:` link at the bottom of the page to compare `vx.y.z...HEAD`.
 - Under the `_Unreleased` link, create a new link for the release `.. _x.y.z: https://[...]/compare/va.b.c...vx.y.z`, where `va.b.c` is the previous release.
- Commit changes and push `bump-vx.y.z` branch for testing.
- Now is a good time to build and check the documentation locally.
 - Ensure that command line output examples are up to date. They can be updated using the output of the `cmd` and `cmdbad` `tox` environments.
- When branch `bump-vx.y.z` is green, then merge it to master.
- Update master locally and ensure that you remain on master for the rest of the process.
- Test that a build can be shipped to test PyPI with `make testpypi`. (Every build runs the full clean test suite locally to ensure that nothing has broken before building)
- After successful push, check the [TestPyPI page](#).
- Then tag the repo with `make tag`. Add a short message about what the key change is.
- Make the new tag public with `git push origin --tags`.
- Build and push to PyPI with `make pypi`.
- After successful push, check the [PyPI page](#).

Post release checks

- Visit the [CHANGELOG](#) and ensure that the new release's comparison link works with the new tag.
- Check the [RTD builds](#) to ensure that the latest documentation version has been picked up and that the `stable` docs are pointed at it.

A new docs release will not have been created for the new tag as per [this issue](#). Click “Build Version:” on the builds page for the new tag to be picked up.