flake8-aaa Documentation

Release 0.11.1

James Cooke

Dec 30, 2020

Contents

1 Overview **1**
- 1.1 Compatibility list . . . 1
- 1.2 Test discovery and analysis . . . 3
- 1.3 Rules and error codes . . . 4
- 1.4 Controlling Flake8-AAA . . . 7
- 1.5 Release checklist . . . 9

CHAPTER 1

Overview

Flake8-AAA is a Flake8 plugin that checks Python tests follow the Arrange-Act-Assert pattern.

Please see the README on GitHub for a general introduction to this project and AAA.

Continue here for more detail about using Flake8-AAA.

1.1 Compatibility list

Flake8-AAA is compatible with the following software. Future versions will maintain this compatibility as closely as possible.

1.1.1 Python

Works with Python 3.

Flake8-AAA is fully compatible and tested against the latest versions of Python 3. Currently that's 3.6, 3.7 and 3.8.

The following versions of Python are no longer supported:

Python 3.5

Python 3.5 was supported up to `v0.7.2`

- v0.7.2 on PyPI
- v0.7.2 Documentation
- Github v0.7.2 tag

Python 2

Python 2 was supported up to `v0.4.0`

- v0.4.0 on PyPI
- v0.4.0 Documentation
- Github v0.4.0 tag

1.1.2 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
3.8.2 (aaa: 0.10.0, mccabe: 0.6.1, pycodestyle: 2.6.0, pyflakes: 2.2.0) CPython 3.6.
↪10 on Linux
```

The `aaa: 0.10.0` part of that output tells you Flake8 found this plugin.

1.1.3 Yapf

Yapf is used to format Flake8-AAA code and tests. It is the primary formatter focused on for compatibility.

1.1.4 Black

Flake8-AAA is compatible with tests formatted with Black.

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

The AAA pattern is PEP8 compatible so it makes sense that Flake8-AAA should work with PEP8 compatible formatters.

This compatibility is pinned by the test examples in the examples/good/black directory. These tests are formatted with the latest version of Black in default mode. They are then checked to pass Flake8-AAA's linting.

1.1.5 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).

1.1.6 Unittest

Python unittest style is supported.

To pin this compatibility we include unittest-style tests in the examples/good directory -

1.2 Test discovery and analysis

When running as a Flake8 plugin, Flake8-AAA filters the Python code passed to it by Flake8. It finds code that looks like test code and then checks that code matches the AAA pattern. When all checks pass, then no error is raised.

1.2.1 Filtering

First, the filename is checked. It must either `test.py`, `tests.py` or start with `test_`. For those files that match, every function that has a name that starts with "test" is checked. This includes class methods.

Test functions and methods that contain only comments, docstrings or `pass` are skipped.

The aim of this process is to mirror Pytest's default collection strategy as closely as possible. It also aims to work with popular testing tutorials such as Django's Writing your first Django app which states:

> Put the following in the `tests.py` file in the polls application

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.

1.2.2 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 snipped 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,
]"""
```

Mark comments

All lines that are # comment lines are marked.

```
# This line is considered a comment line

result = item.act()  # But not this line
```

This process relies on analysing the tokens that make up the test.

Find the Act block

There are four recognised types of Act block:

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 `result =` action:

```
result = do_thing()
```

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

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

Flake8-AAA searches each test function for lines that look like Act blocks. It will raise an error when a function does not have exactly 1 Act block.

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

Finally a line-by-line analysis of the test function is carried out to ensure that:

- No blocks contain extra blank lines.
- There is a single blank line above and below the Act block.

1.3 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"

1.3.1 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.

1.3.2 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.

1.3.3 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.

1.3.4 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.

1.3.5 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.

1.3.6 AAA06: comment in Act block

Problematic code

```
def test() -> None:
    shopping = ['apples', 'bananas', 'cabbages']

    # Reverse shopping list operates in place
    shopping.reverse()  # act

    assert shopping == ['cabbages', 'bananas', 'apples']
```

Correct code

```
def test() -> None:
    """
    Reverse shopping list operates in place
    """
    shopping = ['apples', 'bananas', 'cabbages']
```

(continues on next page)

(continued from previous page)

```
    shopping.reverse()  # act

    assert shopping == ['cabbages', 'bananas', 'apples']
```

Rationale

The Act block carries out a single action on an object. It is the focus of each test. Therefore any comments on this single action are really comments on the test itself and so should be moved to the test docstring.

By placing these important comments in the docstring we can:

- Make it easier to keep the Act block simple.
- Help to distinguish the Act block from the rest of the test.
- Improve the documentation of tests because any important comments and notes are lifted to the top of the test.

Exceptions

Inline comments used to pass information to linters are OK:

- Marking the Act block:

```
shopping.reverse()  # act
```

- Marking lines in the action for linting reasons:

```
result = shopping.reverse()  # type: ignore
```

1.3.7 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 Controlling Flake8-AAA

1.4.1 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
```

1.4.2 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
```

Only one file can be passed to the command line at a time. So to test all files in a test suite, `find` should be used:

```
$ find tests -name '*.py' | xargs -n 1 python -m flake8_aaa
```

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.

CMT Line is a # comment.

DEF Test function definition.

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

1.5 Release checklist

The following tasks need to be completed for each release of Flake8-AAA. They are mainly for the maintainer's use.

1.5.1 Versioning

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

- Create a branch for the new release. Usually called something like `bump-vx.y.z`.
- Run `./bump_version.sh [x.y.z]`.
- Commit changes and push `bump-vx.y.z` branch for testing.

1.5.2 Documentation

Now is a good time to build and check the documentation locally:

```
$ make doc
$ firefox docs/_build/html/index.html
```

Ensure that command line output examples are up to date. They can be updated using the output of the `cmd` and `cmdbad` `tox` environments.

1.5.3 Merge

- When branch `bump-vx.y.z` is green, then merge it to `master`. All pull requests are "squash merged".
- Update master locally and ensure that you remain on master for the rest of the process.

1.5.4 Test PyPI

- Test that a build can be shipped to test PyPI with `make testpypi`.
- After successful push, check the TestPyPI page.

1.5.5 Tag and push

- Tag the repo with `make tag`. Add a short message describing the key feature of this release.
- 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.

1.5.6 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.