flake8-aaa Documentation

Release 0.17.0

James Cooke

Oct 30, 2023

CONTENTS

CHAPTER

ONE

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 active versions of Python 3 as listed on the python.org downloads page.

See also…

See *full list of previously supported Python versions* for links to the last supported packages and documentation.

1.1.2 Flake8

Requires Flake8 version 3 and later. All integration tests run with the latest version of Flake8 for the active version of Python.

We use the newer plugin system implemented in Flake8 v3.

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

```
flake8 --version
```

```
6.1.0 (flake8-aaa: 0.17.0, mccabe: 0.7.0, pycodestyle: 2.11.1, pyflakes: 3.1.0) CPython␣
↪3.11.6 on Linux
```

The `flake8-aaa: 0.17.0` 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.

Black version `23.1.0` changed how it managed blank lines by default. Set *"large" Act block style option or configuration* when running via Flake8 for best compatibility with Black:

```
flake8 --aaa-act-block-style=large
```

See also Black formatted example tests in Flake8-AAA's test suite.

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.

1.1.7 Previous Python versions

The following versions of Python are no longer supported:

Python 3.7

Python 3.7 was supported up to `v0.15.0`

- v0.15.0 on PyPI
- v0.15.0 Documentation
- Github v0.15.0 tag

Python 3.6

Python 3.6 was supported up to `v0.12.1`

- v0.12.1 on PyPI
- v0.12.1 Documentation
- Github v0.12.1 tag

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

1.2 Error codes

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

- `E303` "too many blank lines"
- `E702` "Multiple statements on one line"

1.2.1 AAA01: no Act block found in test

An Act block is usually a line like `result =` or a check that an exception is raised. When Flake8-AAA raises `AAA01` it could not find an Act block in the indicated test function.

Problematic code

```
def test_some_text() -> None:
    some = 'some'
    text = 'text'

    some_text = f'{some}_{text}'

    assert some_text == 'some_text'
```

```
from pytest import raises

def test() -> None:
    with raises(IndexError):
        list()[0]
```

Correct code 1

Use `result =` assignment to indicate the action in the test:

```
def test_some_text() -> None:
    some = 'some'
    text = 'text'

    result = f'{some}_{some}'

    assert result == 'some_text'
```

Ensure all Pytest context managers are in the `pytest` namespace - use `pytest.raises()` not just `raises()`:

```
import pytest

def test() -> None:
    with pytest.raises(IndexError):
        list()[0]
```

Correct code 2

Alternatively, mark your Act block with the `# act` hint to indicate the action in the test. This can be useful for scenarios where a result can not be assigned, such as tests on functions that return `None`.

```
def test_some_text() -> None:
    some = 'some'
    text = 'text'

    some_text = f'{some}_{text}'  # act

    assert some_text == 'some_text'
```

```
from pytest import raises

def test() -> None:
    with raises(IndexError):
        list()[0]  # act
```

Rationale

The Act block carries out a single action on an object so it's important that Flake8-AAA can clearly distinguish which line or lines make up the Act block in every test.

Flake8-AAA recognises code blocks wrapped in Pytest context managers like `pytest.raises()` as Act blocks.

It also recognises unittest's `assertRaises()` blocks as Act blocks.

1.2.2 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.

1.2.3 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.

Prerequisites

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

If test code is formatted with Black, then it's best to set *"large" Act block style*.

Problematic code

```
def test_simple(hello_world_path: pathlib.Path) -> None:
    with open(hello_world_path) as f:
        result = f.read()

    assert result == 'Hello World!\n'
```

Correct code

Since the `open()` context manager is part of the Arrange block, create space between it and the `result =` Act block.

```
def test_simple(hello_world_path: pathlib.Path) -> None:
    with open(hello_world_path) as f:

        result = f.read()

    assert result == 'Hello World!\n'
```

Alternatively, if you want the context manager to be treated as part of the Act block, the *"large" Act block style* as mentioned above.

Rationale

This blank line creates separation between the test's Arrange and Act blocks and makes the Act block easy to spot.

1.2.4 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.

1.2.5 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.

Problematic code

```
def test_a() -> None:
    x = 3

    y = 4

    result = x**2 + y**2

    assert result == 25
```

```
def test_b() -> None:
    nothing = None

    with pytest.raises(AttributeError):

        nothing.get_something()
```

Correct code

Remove the blank lines.

```
def test_a() -> None:
    x = 3
    y = 4

    result = x**2 + y**2
```

(continues on next page)

(continued from previous page)

```
    assert result == 25
```

```
def test_b() -> None:
    nothing = None

    with pytest.raises(AttributeError):
        nothing.get_something()
```

Rationale

Blank lines are essential for dividing up a test. There will usually be just two blank lines in each test - one above and one below the Act block. They serve to separate the Act block from the rest of the test.

When there are additional blank lines in a test, then the "shape" of the test is broken and it is hard to see where the Act block is at a glance.

1.2.6 AAA06: comment in Act block

Problematic code

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

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

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

```
def test_b() -> None:
    # NOTE: the most interesting thing about this test is this comment
    result = 1 + 1

    assert result == 2
```

Correct code

Use docstrings instead of hash-comments:

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

    shopping.reverse()  # act

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

```
def test_b() -> None:
    """
    NOTE: the most interesting thing about this test is this comment
    """
    result = 1 + 1

    assert result == 2
```

Separate hash-comment line from Act block with a blank line:

```
def test_b() -> None:
    # NOTE: the most interesting thing about this test is this comment

    result = 1 + 1

    assert result == 2
```

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

Directives in the form of inline comments are OK, for example:

- Marking the Act block:

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

- Marking lines in the action for linting reasons:

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

1.2.7 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 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.

Resolution

Please open a new issue containing the output for the failing test as generated by `flake8`.

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.

1.3 Options and configuration

Flake8 can be invoked with `--` options *and* can read values from project configuration files.

All names of Flake8-AAA's options and configuration values are prefixed with "aaa". E.g. `--aaa-act-block-style`.

1.3.1 Act block style

Command line flag
`--aaa-act-block-style`

Configuration option
`aaa_act_block_style`

The Act block style option adjusts how Flake8-AAA builds the Act block from the Act node.

The allowed values are "default" and "large".

Default

In default mode the Act block is the single Act node, best demonstrated by example:

```
result = do_thing()
```

Or…

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

The important feature of default Act blocks is that they do not contain any context managers other than pytest or unittest ones.

```
def test_with():
    a_class = AClass()
    with freeze_time("2021-02-02 12:00:02"):

        result = a_class.action('test')

    assert result == 'test'
```

In the example above, Flake8-AAA considers the `with freeze_time()` context manager to be in the Arrange block. It therefore expects a blank line between it and the `result =` Act block.

Large

Large style Act blocks have been provided to be compatible with Black.

In Large mode the Act block can grow to include context managers that wrap it. For example, referring to the test above, this would be formatted as follows with Large Act blocks:

```
def test_with():
    a_class = AClass()

    with freeze_time("2021-02-02 12:00:02"):
        result = a_class.action('test')

    assert result == 'test'
```

The `result =` result assignment Act block expands to include the `freeze_time()` context manager. In this way, the blank line that divides the Arrange block from the Act block can be *before* the context manager - a format which is compatible with Black.

Note however, the context manager only joined the Act block because the Act node was the **first** line in the context manager's body. If we moved the `AClass()` initialisation inside the context manager, something different would happen:

```
def test_with():
    with freeze_time("2021-02-02 12:00:02"):
        a_class = AClass()

        result = a_class.action('test')

    assert result == 'test'
```

This time the result assignment does *not* consume the context manager. Instead, the `freeze_time()` context manager and the `a_class` initialisation make up the Arrange block, and there's a single blank line between that and the simple result assignment Act block.

1.4 Directives

Flake8-AAA can be controlled using some special directives in the form of comments in your test code.

1.4.1 Explicitly marking blocks

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

See *AAA01: no Act block found in test - Correct code 2*.

1.4.2 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
```

1.5 Test discovery and analysis

Flake8-AAA filters the Python code passed to it by Flake8. It finds lines that looks like test code and then checks those lines match the AAA pattern. When all checks pass no error is raised.

1.5.1 File filtering

First, the filename is checked. It must match one of the following patterns:

- Is called `test.py` or `tests.py`.
- Starts with `test_`, i.e match `test_*.py`
- Ends with `_test.py`, i.e. match `*_test.py`.

For every file that matches the patterns above, Flake8-AAA checks every function and class method whose name starts with "test".

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

Rationale

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.

1.5.2 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.

The "act block style" configuration allows for a "large" style of Act block to be specified, which changes how Act blocks are built in relation to context managers. See … # TODO225 fix this ref

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.

1.6 Release checklist

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

1.6.1 Versioning

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

- Create a branch for the new release. Usually called something like `bump-x.y.z`.
- Run `./bump_version.sh [x.y.z]`.
- Ensure command line output examples in `README.rst` are up to date. Run:

  ```
  make signature
  ```

  Update the version string in the README and compatibility doc.
- Commit changes and push `bump-x.y.z` branch for testing. Use `Bump to x.y.z` as the PR title.

1.6.2 Merge

- When branch `bump-x.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.

1.6.3 Test PyPI

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

1.6.4 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.

1.6.5 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.