pytest-workflow Documentation

Release 1.1.2

Leiden University Medical Center

Mar 07, 2019

Contents

Table of contents

CHAPTER 1

Introduction

Writing workflows is hard. Testing if they are correct is even harder. Testing with bash scripts or other code has some flaws. Is this bug in the pipeline or in my test-framework? Pytest-workflow aims to make testing as simple as possible so you can focus on debugging your pipeline.

CHAPTER 2

Installation

Pytest-workflow is tested on python 3.5, 3.6 and 3.7. Python 2 is not supported.

In a virtual environment:

- Create a new python3 virtual environment.
- Make sure your virtual environment is activated.
- Install using pip `pip install pytest-workflow`

On Ubuntu or Debian:

- This requires the `python3` and `python3-pip` packages to be installed.
- Installing
  - system-wide: `sudo python3 -m pip install pytest-workflow`
  - for your user only (no sudo needed): `python3 -m pip install --user pytest-workflow`
- `pytest` can now be run with `python3 -m pytest`.

NOTE: Running plain `pytest` on Ubuntu or Debian outside of a virtual environment will not work with `pytest-workflow` because this will start the python2 version of `pytest`. This is because python2 is the default python on any distribution released before January 1st 2020.

Pytest-workflow is also available as a conda package on bioconda. To install with conda:

- Set up conda to use the bioconda channel
- `conda install pytest-workflow`

CHAPTER 3

Writing tests with pytest-workflow

3.1 Getting started

In order to write tests that are discoverable by the plugin you need to complete the following steps.

- Create a `tests` directory in the root of your repository.
- Create your test yaml files in the `tests` directory. The files need to start with `test` and have a `.yml` or `.yaml` extension.

Below is an example of a YAML file that defines a test:

```
- name: Touch a file
  command: touch test.file
  files:
    - path: test.file
```

This will run `touch test.file` and check afterwards if a file with path: `test.file` is present. It will also check if the `command` has exited with exit code `0`, which is the only default test that is run. Testing workflows that exit with another exit code is also possible.

3.2 Test options

```
- name: moo file                        # The name of the workflow (required)
  command: bash moo_workflow.sh         # The command to execute the workflow (required)
  files:                                # A list of files to check (optional)
    - path: "moo.txt"                   # File path. (Required for each file)
      contains:                         # A list of strings that should be in the file␣
↪(optional)
        - "moo"
      must_not_contain:                 # A list of strings that should NOT be in the␣
↪file (optional)
```

(continues on next page)

(continued from previous page)

```
        - "Cock a doodle doo"
      md5sum: e583af1f8b00b53cda87ae9ead880224   # Md5sum of the file (optional)

- name: simple echo                  # A second workflow. Notice the starting `-`␣
↪which means
  command: "echo moo"                # that workflow items are in a list. You can add␣
↪as much workflows as you want
  files:
    - path: "moo.txt"
      should_exist: false            # Whether a file should be there or not.␣
↪(optional, if not given defaults to true)
  stdout:                            # Options for testing stdout (optional)
    contains:                        # List of strings which should be in stdout␣
↪(optional)
      - "moo"
    must_not_contain:                # List of strings that should NOT be in stout␣
↪(optional)
      - "Cock a doodle doo"

- name: mission impossible           # Also failing workflows can be tested
  tags:                              # A list of tags that can be used to select␣
↪which test
    - should fail                    # is run with pytest using the `--tag` flag.
  command: bash impossible.sh
  exit_code: 2                       # What the exit code should be (optional, if not␣
↪given defaults to 0)
  files:
    - path: "fail.log"               # Multiple files can be tested for each workflow
    - path: "TomCruise.txt.gz"       # Gzipped files can also be searched, provided␣
↪their extension is '.gz'
      contains: "starring"
  stderr:                            # Options for testing stderr (optional)
    contains:                        # A list of strings which should be in stderr␣
↪(optional)
      - "BSOD error, please contact the IT crowd"
    must_not_contain:                # A list of strings which should NOT be in␣
↪stderr (optional)
      - "Mission accomplished!"
```

The above YAML file contains all the possible options for a workflow test.

3.3 Writing custom tests

Pytest-workflow provides a way to run custom tests on files produced by a workflow.

```
import pathlib
import pytest

@pytest.mark.workflow(name='files containing numbers')
def test_div_by_three(workflow_dir):
    number_file = workflow_dir / pathlib.Path("123.txt")

    with number_file.open('rt') as file_h:
        number_file_content = file_h.read()
```

(continues on next page)

(continued from previous page)

```
    assert int(number_file_content) % 3 == 0
```

The `@pytest.mark.workflow(name='files containing numbers')` marks the test as belonging to a workflow named 'files containing numbers'. The mark can also be written without the explicit `name` key as `@pytest.mark.workflow('files containing nummbers')`. This test will only run if the workflow 'files containing numbers' has run.

`workflow_dir` is a fixture. It does not work without a `pytest.mark.workflow('workflow_name')` mark. This is a pathlib.Path object that points to the folder where the named workflow was executed. This allows writing of advanced python tests for each file produced by the workflow.

NOTE: stdout and stderr are available as files in the root of the `workflow_dir` as `log.out` and `log.err` respectively.

CHAPTER 4

Running pytest-workflow

Run `pytest` from an environment with pytest-workflow installed or `python3 -m pytest` if using a system-wide or user-wide installation. Pytest will automatically gather files in the `tests` directory starting with `test` and ending in `.yaml` or `.yml`.

The workflows are run automatically. Each workflow gets its own temporary directory to run. The `stdout` and `stderr` of the workflow command are also saved to this directory. The temporary directories are cleaned up after the tests are completed. If you wish to inspect the output of a failing workflow you can use the `--kwd` or `--keep-workflow-wd` flag to disable cleanup. This will also make sure the logs of the pipeline are not deleted. The `--keep-workflow-wd` flag is highly recommended when debugging pipelines.

If you wish to change the temporary directory in which the workflows are run use `--basetemp <dir>` to change pytest's base temp directory.

WARNING: When a directory is passed to `--basetemp` some of the directory contents will be deleted. For example: if your workflow is named `"my workflow"` then any file or directory named `my_workflow` will be deleted. This makes sure you start with a clean slate if you run pytest again with the same `basetemp` directory. DO NOT use `--basetemp` on directories where none of the contents should be deleted.

To run multiple workflows simultaneously you can use `--workflow-threads <int>` or `--wt <int>` flag. This defines the number of workflows that can be run simultaneously. This will speed up things if you have enough resources to process these workflows simultaneously.

To check the progress of a workflow while it is running you can use `tail -f` on the `stdout` or `stderr` file of the workflow. The locations of these files are reported in the log as soon as a workflow is started.

4.1 Running specific workflows

To run a specific workflow use the `--tag` flag. Each workflow is tagged with its own name and additional tags in the `tags` key of the yaml.

```
- name: moo
  tags:
    - animal
```

(continues on next page)

(continued from previous page)

```
  command: echo moo
- name: cock-a-doodle-doo
  tags:
    - rooster sound
    - animal
  command: echo cock-a-doodle-doo
- name: vroom vroom
  tags:
    - car
  command: echo vroom vroom
```

With the command `pytest --tag moo` only the workflow named 'moo' will be run. With `pytest --tag 'rooster sound'` only the 'cock-a-doodle-doo' workflow will run. Multiple tags can be used like this: `pytest --tag 'rooster sound' --tag animal` This will run all workflows that have both 'rooster sound' and 'animal'.

Internally names and tags are handled the same so if the following tests:

```
- name: hello
  command: echo 'hello'
- name: hello2
  command: echo 'hello2'
  tags:
    - hello
```

are run with `pytest --tag hello` then both `hello` and `hello2` are run.

CHAPTER 5

Examples

5.1 Snakemake example

An example yaml file that could be used to test a snakemake pipeline is listed below.

```
- name: test-dry-run
  command: snakemake -n -r -p -s Snakefile
- name: test-full-run
  command: snakemake -r -p -s Snakefile
  files:
    - "my_output.txt"
  stderr:
    contains:
     - "(100%) done"
```

5.2 WDL with Cromwell example

Below an example yaml file is explained which can be used to test a WDL pipeline run through Cromwell.

One problem with Cromwell is the way it handles relative paths and how it handles the input file:

- Relative paths are written only within the `cromwell-executions` folder. If you want to write outside this folder you need absolute paths. This is fine but for testing your pipeline `pytest-workflow` creates a temporary folder from which the pipeline is run. You don't know beforehand which path this is, but you could use the environment variable `$PWD`.
- However the second problem is that inputs can only be supplied to Cromwell in a json file, not on the command line. So you cannot dynamically choose an output folder. You have to rewrite the input file.

To fix this problem you can write `command` to be a bash script that injects `$PWD` into the inputs.json.

```
- name: My pipeline
  command: >-
    bash -c '
    TEST_JSON=tests/inputs/my_pipeline_test1.json ;
    sed -i "2i\"my_pipeline.output_dir\":\"$PWD/test-output\"," $TEST_JSON ;
    cromwell run -i $TEST_JSON simple.wdl'
  files:
    - path: test-output/moo.txt.gz
      md5sum: 173fd8023240a8016033b33f42db14a2
  stdout:
    contains:
      - "WorkflowSucceededState"
```

`sed -i "2i\"my_pipeline.output_dir\":\"$PWD/test-output\"," $TEST_JSON` inserts `"my_pipeline.output_dir":"</pytest/temporary/dir>/test-output",` on the second line of `$TEST_JSON`. This solves the problem. File paths can now be traced from `test-output` as demonstrated in the example.

CHAPTER 6

Known issues

- `pytest-workflow` does not work well together with `pytest-cov`. This is due to the temporary directory creating nature of `pytest-workflow`. This can be solved by using:

```
coverage run --source=<your_source_here> -m py.test <your_test_dir>
```

  This will work as expected.

CHAPTER 7

Reporting bugs and feature requests

Bugs can be reported and features can be requested on our Github issue tracker.

The aim of this project is to be as user-friendly as possible for writing workflow tests, so all suggestions and bug reports are welcome!

CHAPTER 8

Contributing

If you feel like this project is missing a certain something, feel free to make a pull request. You can find our Github page here.

CHAPTER 9

Changelog

9.1 version 1.1.2

- Fixed a bug where the program would hang indefinitely after a user input error.

9.2 version 1.1.1

- Added `--kwd` as alias for `--keep-workflow-wd`. Notify the user of deletion of temporary directories and logs.
- Released pytest-workflow as a conda package on bioconda.

9.3 version 1.1.0

- Enabled custom tests on workflow files.

9.4 Version 1.0.0

Lots of small fixes that improve the usability of pytest-workflow are included in version 1.0.0.

- Gzipped files can now also be checked for contents. Files with '.gz' as extension are automatically decompressed.
- `stdout` and `stderr` of workflows are now streamed to a file instead of being kept in memory. This means you can check the progress of a workflow by running `tail -f <stdout or stderr>`. The location of `stdout` and `stderr` is now reported at the start of each worflow. If the `--keep-workflow-wd` is not set the `stdout` and `stderr` files will be deleted with the rest of the workflow files.

- The log reports now when a workflow is starting, instead of when it is added to the queue. This makes it easier to see which workflows are currently running and if you forgot to use the `--workflow-threads` or `--wt` flag.
- Workflow exit code failures now mention the name of the workflow. Previously the generic name "Workflow" was used, which made it harder to figure out which workflows failed.
- When tests of file content fail because the file does not exist, a different error message is given compared to when the file exist, but the content is not there, which makes debugging easier. Also the accompanying "FileNotFound" error stacktrace is now suppressed, which keeps the test output more pleasant.
- When tests of stdout/stderr content or file content fail a more informative error message is given to allow for easier debugging.
- All workflows now get their own folder within the *same* temporary directory. This fixes a bug where if `basetemp` was not set, each workflow would get its own folder in a separate temp directory. For example running workflows 'workflow1' and 'workflow2' would create two temporary folders:

  '/tmp/pytest_workflow_**33mrz5a5**/workflow1' and '/tmp/pytest_workflow_**b8m1wzuf**/workflow2'

  This is now changed to have all workflows in one temporary directory per pytest run:

  '/tmp/pytest_workflow_**33mrz5a5**/workflow1' and '/tmp/pytest_workflow_**33mrz5a5**/workflow2'
- Disallow empty `command` and `name` keys. An empty `command` caused pytest-workflow to hang. Empty names are also disallowed.

9.5 Version 0.4.0

- Added more information to the manual on how to debug pipelines and use `pytest-workflow` outside a virtual environment.
- Reworked code to use `tempfile.mkdtemp` to create a truly unique temporary working directory if the `--basetemp` flag is not used. This replaces the old code which dependeded on pytest internal code which was flagged as deprecated. Also more information was added to the manual about the use of `--basetemp`.
- Added a test case for WDL pipelines run with Cromwell and wrote an example for using WDL+Cromwell in the manual.
- Added `--tag` flag to allow for easier selection of workflows during testing.
- Added a test case for snakemake pipelines and wrote an example for using pytest-workflow with snakemake in the manual.

9.6 Version 0.3.0

- Improved the log output to look nicer and make workflow log paths easier to find in the test output.
- Fixed an error that polluted the log message with a pytest stacktrace when running more than one workflow. Measures are taken in our test framework to detect such issues in the future.
- Added the possibility to run multiple workflows simultaneously with the `--workflow-threads` or `--wt` flag.
- Made code easier to maintain by using stdlib instead of pytest's `py` lib in all of the code.
- Added a schema check to ensure that tests have unique names when whitespace is removed.

9.7 Version 0.2.0

- Cleanup the readme and move advanced usage documentation to our readthedocs page.
- Start using sphinx and readthedocs.org for creating project documentation.
- The temporary directories in which workflows are run are automatically cleaned up at the end of each workflow test. You can disable this behaviour by using the `--keep-workflow-wd` flag, which allows you to inspect the working directory after the workflow tests have run. This is useful for debugging workflows.
- The temporary directories in which workflows are run can now be changed by using the `--basetemp` flag. This is because pytest-workflow now uses the built-in tmpdir capabilities of pytest.
- Save stdout and stderr of each workflow to a file and report their locations to stdout when running `pytest`.
- Comprehensible failure messages were added to make debugging workflows easier.

9.8 Version 0.1.0

- A full set of examples is now provided in the README.
- Our code base is now checked by pylint and bandit as part of our test procedure to ensure that our code adheres to python and security best practices.
- Add functionality to test whether certain strings exist in files, stdout and stderr.
- Enable easy to understand output when using pytest verbose mode (`pytest -v`). The required code refactoring has simplified the code base and made it easier to maintain.
- Enable the checking of non-existing files
- Enable the checking of file md5sums
- Use a schema structure that is easy to use and understand.
- Pytest-workflow now has continuous integration and coverage reporting, so we can detect regressions quickly and only publish well-tested versions.
- Fully parametrized tests enabled by changing code structure.
- Initialized pytest-workflow with option to test if files exist.