show your work! for open science

Test cases

Giovanni Picogna

Universitäts-Sternwarte, Ludwig-Maximilians-Universität München, Scheinerstr. 1, München, 81679, Bayern, Germany
e-mail: picogna@usm.lmu.de

Received 14 July 2025 / Accepted 14 July 2025

ABSTRACT

Aims. I provide some easy to follow instructions to set-up your first show your work! project
Methods. This is not at all a complete guide, for which I point the readers at the well written documentation on https://show-your.work/
Results.

Key words. open science

1. Installation

show your work! requires Python <3.11, >=3.8 in the current development version. We can start by creating a conda environment with python 3.10

```
conda create -n python310 python=3.10
anaconda
```

activate it

```
conda activate python310
```

and install the development version from github

```
pip install git+https://github.com/
showyourwork/showyourwork
```

2. First paper

2.1. Local build

In order to set up your first paper you need to run

```
showyourwork setup ${GITHUB_USER}/${GITHUB-
REPO}
```

and create the related repo on github (follow the instruction on screen - you can add the caching support on Zenodo and the Overleaf connection also in a second step) A working `environment.yml` for the conda environment is:

```
channels:
  - conda-forge
  - defaults
dependencies:
  - numpy=1.21
  - pip=25.1.1
  - python=3.10
  - pip:
    - matplotlib==3.10
```

You can then build first your paper locally

```
showyourwork build
```

and check the resulting ms.pdf file in the root directory for the final result.

2.2. Remote build on github

In order to build it on github with github actions you need first to include/substitute in the files `build.yml` and `build-pull-request.yml` under the directory `.github/workflows` the following lines:

```
- name: Build the article PDF
  id: build
  with:
    showyourwork-spec: git+https://github.
com/showyourwork/showyourwork
  uses: showyourwork/showyourwork-
action@main
  env:
    SANDBOX_TOKEN: ${{ secrets.
SANDBOX_TOKEN }}
    OVERLEAF_TOKEN: ${{ secrets.
OVERLEAF_TOKEN }}
```

then you can `git add` the changed files, commit it and push it to the remote repository.

3. Adding a figure

3.1. Simple figure

show your work! does not expect you to provide directly a figure (in fact it gives an error if you do). It expects instead a script to generate the plot through the `script` command and,

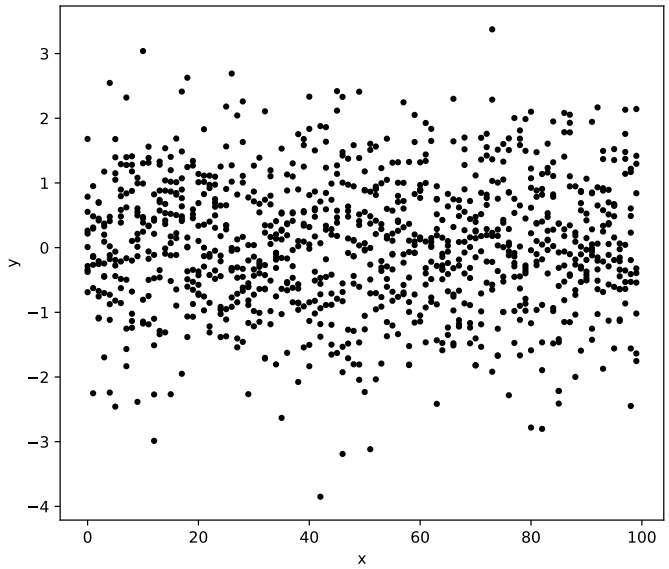

Fig. 1: Plot showing a bunch of random numbers.

after running it, it looks for the figure obtained with it. Fig. 1 is an example. The script `randomnumbers.py` is placed in the `src/scripts/` directory and called in the first line of the figure definition. After that, the figure is included normally from the predefined path.

```
\begin{figure}
    \script{random_numbers.py}
    \begin{centering}
        \includegraphics[width=\linewidth]{
    figures/random_numbers.pdf}
        \caption{
            Plot showing a bunch of random
    numbers.
        }
        \label{fig:random_numbers}
    \end{centering}
\end{figure}
```

The script is generating and plotting some random numbers using the `numpy random` package

```
import matplotlib.pyplot as plt
import numpy as np
import paths

# Generate some data
random_numbers = np.random.randn(100, 10)

# Plot and save
fig = plt.figure(figsize=(7, 6))
plt.plot(random_numbers, 'k.')
plt.xlabel("x")
plt.ylabel("y")
fig.savefig(paths.figures / "random_numbers.pdf
    ", bbox_inches="tight", dpi=300)
```

3.2. Static figure

It is possible that some figures are not generated from a script, i.e. an hand drawn figure or a figure from another paper. In that case `showyouwork!` provides a static directory under `src/static` where any static content can be added. In this case the figure is defined normally in LaTeX, but the location of the figure is always within the figures folder (show your work! is copying it under the wood from the static directory to the LaTeXfigures directory).

Fig. 2: USM Logo.

```
\begin{figure}
    \begin{centering}
        \includegraphics[width=\linewidth]{
    figures/usmlogo.pdf}
        \caption{
            USM Logo.
        }
        \label{fig:usm_logo}
    \end{centering}
\end{figure}
```

4. Zenodo integration

4.1. Static datasets

If your workflow depends on data that cannot be programmatically generated (e.g. data collected from a telescope), that data should be made available to anyone trying to reproduce your results. Instead of committing the dataset directly to the repository, one can archive it on an online open-access file-hosting service (like Zenodo, for which show your work! does all the communicating back-and-forth for you). All you need to do is specify the ID of the (public) archive and some information about the files your workflow needs in the `showyourwork.yml` config file.

4.2. Dynamic datasets

show your work! can cache the results of intermediate steps in your pipeline on Zenodo or Zenodo Sandbox. This is useful

for workflows that need running expensive simulations, etc., that a reader may not want to rerun. It's also useful for builds on GitHub Actions, which has limited compute resources.

The way show your work! deals with these cases is to cache these lengthy computations on Zenodo alongside a record of all the inputs that went into generating the cached output. If, on subsequent runs of the workflow, the inputs remain unchanged, show your work! will simply download the cached results from Zenodo, maintaining the guarantee that the output you get follows deterministically from the given inputs.

4.2.1. Set-up the Zenodo integration

By default, caching takes place on Zenodo Sandbox, which is equal to Zenodo in all respects except that records on that service are temporary. This makes it perfect for caching intermediate results during the development cycle, where things are likely to change a lot and you may not want to assign a static, permanent DOI with any particular dataset.

show your work! needs access to an API token to communicate with Zenodo Sandbox. You can generate it by clicking here. Name the token something informative and make sure to give it `deposit:actions` and `deposit:write` permissions. Copy the token and store it somewhere secure.

Then, on your local computer, create an environment variable called `$SANDBOX_TOKEN` with value equal to the API token you just generated. You can either do this manually: `export SANDBOX_TOKEN=YYYYYY` or by adding that line to your shell config file (.bashrc, .zshrc, etc.) and re-starting your session. In order for show your work! to have access to Zenodo Sandbox when running on GitHub Actions, you'll also have to provide this value as a secret with name `SANDBOX_TOKEN`.

If you've done all that, the next time you create a new article repository using `showyourwork setup`, pass the `-cache` option and show your work! will automatically create a Zenodo Sandbox draft deposit which it will use to cache your intermediate results. Note that you can also manually create a draft deposit by running `showyourwork cache create` after you created your article repository.

4.2.2. Intermediate results

Earlier, we mentioned that the Zenodo integration allows users to cache intermediate results in their workflow. But what is an *intermediate result*? The standard procedure for generating figures using show your work! is to define a figure script that generates the figure output. There is no intermediate step. We simply go from figure script to figure output.

However, if our figure script involves some expensive simulation, every time we change anything in that script, show your work! will attempt to re-run the entire computation when asked to build your article. This is good for reproducibility but it is extremely wasteful in cases where we wish to tweak some aspect of the plot, like the color of a line.

To avoid this, we can split our script into two: one that runs the simulation and saves the results, and one that loads the results and plots them.

4.2.3. A DustPy simulation

Consider a workflow that needs to run a DustPy simulation to generate a figure. We would like to streamline our workflow by decoupling the plotting step from the simulation step. We can do this by introducing two scripts in the `src/scripts` directory. A first one `simulation.py` that runs and saves the result of the simulation.

Listing 1: simulations.py

```python
from dustpy import Simulation, constants
import paths

sim = Simulation()
sim.initialize()

sim.writer.datadir = paths.data
sim.writer.overwrite = True

sim.run()
```

And a second one `figure.py` that loads the results and plot the figure:

Listing 2: figure.py

```python
import numpy as np
import matplotlib.pyplot as plt
import paths
from dustpy import hdf5writer

cm_to_au = 6.684587e-14

#Load the data
writer = hdf5writer()
writer.datadir = paths.data
data = writer.read.output(21)

# Plot the results
fig, ax = plt.subplots(2, layout='tight')
ax[0].semilogy(data.grid.r*cm_to_au, data.gas.
    Sigma)
ax[0].set_title('Final gas distribution')
ax[0].set_xlabel('R [au]')
ax[0].set_ylabel(r'$\\Sigma_g$ [g/cm$^{-2}$]')
ax[0].set_ylim(1.e-2,1e3)
ax[0].set_xlim(0,400)
ax[1].semilogy(data.grid.r*cm_to_au, data.dust.
    Sigma)
ax[1].set_title('Final dust distribution')
ax[1].set_xlabel('R [au]')
ax[1].set_ylabel(r'$\\Sigma_D$ [g/cm$^{-2}$]')
ax[1].set_ylim(1.e-4,1e1)
ax[1].set_xlim(0,400)
fig.savefig(paths.figures / "figure.pdf")
```

which is then called by LaTeXto plot the figure on the paper

```latex
\begin{figure}
    \script{figure.py}
    \begin{centering}
        \includegraphics[width=\linewidth]{
    figures/figure.pdf}
        \caption{
            Plot showing the result of a DustPy
     simulation.
        }
        \label{fig:simulation_dustpy}
    \end{centering}
\end{figure}
```

Our workflow is now separable: changes to `figure.py` will not result in the re-execution of the simulation, as they are merely plotting changes. The simulation will only be re-executed if we change something in `simulation.py`, like the input arguments to our simulation function.

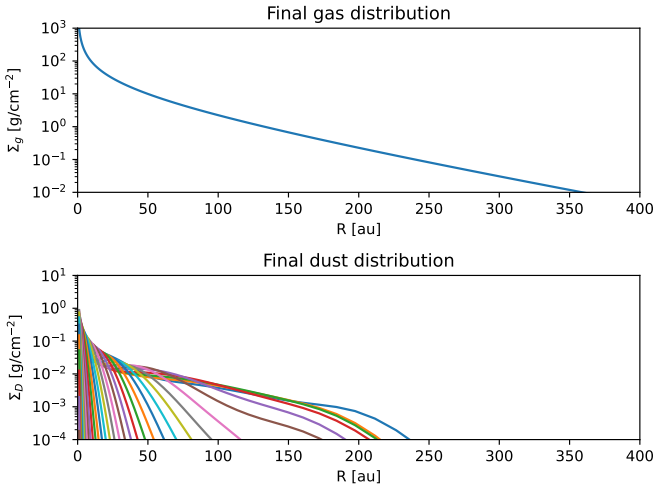

Fig. 3: Plot showing the result of a DustPy simulation.

In order to get this all to work, we need to tell show your work! that:

1. the script `figure.py` has a dependency called `data0021.hdf5`
2. the dependency `data0021.hdf5` can be generated by running the script `simulation.py`.

We accomplish this by

1. editing the config file `showyourwork.yml`

```
dependencies:
  src/scripts/figure.py:
    - src/data/data0021.hdf5
```

2. adding a custom rule to our `Snakefile`

```
rule simulation:
    output:
        "src/data/data0021.hdf5"
    script:
        "src/scripts/simulation.py"
```

4.2.4. Caching the intermediate result

The workflow above is now separable, but we are still not caching anything. If we commit and push it to GitHub, the runner will still have to execute `simulation.py` in order to generate `simulation.dat`. The same goes for third-party users who have cloned your repository. Adding caching functionality can be done by adding a single line to the `Snakefile`:

```
rule simulation:
    output:
        "src/data/data0021.hdf5"
    cache:
        True
    script:
        "src/scripts/simulation.py"
```

which tells show your work! to cache the output of that rule (`simulation.dat`). Normally, if we were just running this in a regular Snakemake pipeline, this would result in the data file getting cached in some local hidden folder. The next time you run your workflow, Snakemake will check to see if any of the inputs to the simulation rule changed and, if not, it will restore `simulation.dat` from the cache (if it is needed).

show your work! builds on this functionality by also caching the file `simulation.dat` on Zenodo Sandbox, allowing the results to be restored on any computer running your workflow (as long as they have the correct `SANDBOX_TOKEN`). This means that, provided you have run your workflow locally first, the runner on GitHub Actions will never have to execute `simulation.py`, as it can just download the result from Zenodo Sandbox. Recall that this procedure still guarantees that you will get the same result as if you had run your entire simulation (provided your workflow is deterministic), since a cache is only restored if none of the upstream inputs to a rule have changed.

The cached files (and the hashes of the rule inputs) are stored in a Zenodo Sandbox deposit draft with concept ID specified in your zenodo.yml config file. If you navigate to Zenodo Sandbox in your browser and log in, you should see a draft with a title like Data for user/repo [main], where user/repo is your repository slug and main is the current branch. At any given time, you can only have one draft per deposit, so if you change any of the inputs to your rule (e.g., if you change the file simulation.py), the draft will get overwritten with a new version of the cache. Note, also, that drafts are private: only users with access to your account can see their files.

If you switch branches, or if you set up a repository without caching functionality and would like to add it, you can create a new Zenodo Sandbox deposit for the current branch by running

```
showyourwork cache create
```

4.2.5. Snakemake

Under the hood, show your work! is essentially a wrapper around Snakemake. The code builds the article PDF by parsing the `ishowyourwork.yml` config file and the `ms.tex` manuscript to build the computational graph for the workflow, identifying which scripts it needs to execute and which datasets it needs to download to produce all the figures in the article.

If your article consists only of text and figures that can be generated by running lightweight scripts, you probably don't need to worry about any of this. But for certain use cases, it can be convenient to extend or even override some of the show your work! functionality by defining custom Snakemake rules.

Every show your work! article repository is instantiated with a blank `Snakefile` at the repository root. This file gets included at the start of the main (build) step of the workflow and it can be used to define custom rules or to run custom python code during the workflow.

Snakefiles are, at their core, Python scripts with a little extra functionality (i.e. any valid Python script is also a valid Snakefile) However, the main thing you probably want to use the Snakefile for is to define custom rules for your workflow. Snakefile rules tell Snakemake how to generate an output file from given input files, much like rules in a classic `Makefile`. Snakemake rules usually look something like this:

Listing 3: Snakemake

```
rule simulation:
    input:
        "dataset1.dat",
        "dataset2.dat"
    output:
        "results.dat"
    conda:
```

```
        "environment.yml"
    params:
        seed=42,
        iterations=1000,
        mode="fast"
    script:
        "src/scripts/run_simulation.py"
```

In this example, we have defined a rule called simulation, which tells Snakemake how to produce the output file results.dat. Specifically, this file can be generated by running the script `src/scripts/run_simulation.py` in an isolated conda environment with specs given in `environment.yml`. The rule also tells Snakemake that the files `dataset1.dat` and `dataset2.dat` are dependencies of `results.dat`, meaning that the rule cannot be executed if those files are not present (and there is no other Snakemake rule capable of generating them) and that whenever either of those two files is modified, this rule will be re-executed the next time the workflow runs in order to keep `results.dat` up to date with its inputs. Finally, the rule specifies three parameters `params`, which can be accessed within the script via the `snakemake.params` dictionary (e.g., `snakemake.params["seed"]`). Note that there is no need to explicitly import snakemake within `run_simulation.py`, as it gets automagically inserted into the namespace.

We can change our previous example to include some input parameters directly from Snakemake

Listing 4: Snakemake DustPy

```
rule simulation:
    output:
        "src/data/data{group}.hdf5"
    conda:
        "environment.yml"
    cache:
        True
    params:
        v_frag = 1.,
        rho_dust = 2.,
        alpha = 5.e-4,
        stellar_mass = 0.1,
        dust_to_gas_ratio = 0.01,
        disk_mass = 0.1
    script:
        "src/scripts/simulation.py"
```

Listing 5: simulations.py

```
from dustpy import Simulation, constants
import paths

sim = Simulation()
sim.initialize()

sim.dust.v.frag = snakemake.params["v_frag"]
sim.dust.rhos = snakemake.params["rho_dust"]
sim.gas.alpha = snakemake.params["alpha"]
sim.star.M = snakemake.params["stellar_mass"]*
    constants.M_sun
sim.dust.eps = snakemake.params["
    dust_to_gas_ratio"]
sim.gas.Mdisk = snakemake.params["disk_mass"]*
    constants.M_sun

sim.writer.datadir = paths.data
sim.writer.overwrite = True

sim.run()
```

The argument to the script key must be a Python script. If your script is in a different language, you can instead pass the shell key and provide a string containing the shell command Snakemake should execute to produce the output file, e.g., `jupyter execute notebook.ipynb`. If you do that, remember to include the script (`notebook.ipynb`) as an explicit input to your rule so that Snakemake can track dependencies properly.

Input files and parameters can be provided as functions, adding another layer of flexibility to your workflow. Rules can also be declared within for loops, if statements, etc. For the full list of features, please refer to the Snakemake documentation.

Another use case for custom rules is the definition of dynamic variables in the LaTeXmanuscript. For example, say I have a script called `age_of_universe.py` that infers the age of the universe from some cosmological dataset:

Listing 6: Python age of universe

```
import paths
from my_awesome_code import get_age_of_universe

# Load the data
dataset = paths.data / "planck.dat"

# Compute the age
age = get_age_of_universe(dataset)

# Write it to disk
with open(paths.output / "age_of_universe.txt",
     "w") as f:
    print(f"{age:.3f}", file=f)
```

I would like to report this age in the text of my article, but I want to avoid having to re-type it in every time I make changes to my workflow that affect this quantity. We can easily automate this by defining a custom Snakemake rule:

Listing 7: Snakefile age of universe

```
rule age_of_universe:
    input:
        "src/data/planck.dat"
    output:
        "src/tex/output/age_of_universe.txt"
    script:
        "src/scripts/age_of_universe.py"
```

And in the LaTeXfile:

Listing 8: TeX age of universe

```
Based on a detailed analysis of Planck
    observations of the cosmic microwave
    background, we have determined the age of
    the universe to be \variable{output/age_of_
    universe.txt} Gyr.
```

This functionality can easily be adapted to automatically populate tables in your article or anything else that can be generated programmatically from your workflow. Note that show your work! automatically parses calls to `variable` statements and adds their arguments as explicit dependencies of the manuscript, so that any changes to these files will trigger a re-run of the compile step.

5. Conclusions

Acknowledgements.