Jupyter notebooks maintenance tips and tricks

Manon Marchand, Stefania Amodeo, Mark Allen

Jupyter notebooks maintenance tips and tricks

Manon Marchand[1], Stefania Amodeo[1], and Mark Allen[1]

[1],*Université de Strasbourg, CNRS, Observatoire astronomique de Strasbourg, UMR 7550, F-67000 Strasbourg, France.;* manon.marchand@astro.unistra.fr

Abstract. Published computational notebooks are rarely reproducible, mainly because software development is not an easy and well documented task for non professional developers. In this communication we provide a comprehensive list of re-usable workflows and good practice tips that we hope can help maintainers of repositories of computational notebooks. This framework is illustrated in a re-usable demonstration repository developed within the ESCAPE and EOSC projects.

1. Introduction

Mixing together code and text in a nice human readable interface makes a good recipe for sharing our work (Knuth 1984). This might explain the growing popularity of computational notebooks since the release of Jupyter notebooks in 2013, an open source alternative to the previously existing ones. More than 10M public notebooks are currently hosted on GitHub,[1] a cloud hosting service for git repositories.

Sharing computational notebooks could be an extremely good news for reproducible data analysis (Piccolo & Frampton 2016) or computational tutorials (Perkel 2018), but in reality, a pioneering study by (Pimentel et al. 2019) showed that most of the notebooks on GitHub are not reproducible. They harvested and studied a sample of 1M notebooks. Among this subset, only 24% could be executed without errors and 4% reproduced the results provided by the authors!

While for research notebooks a solution is to provide a container in which everything works (Piccolo & Frampton 2016; Clyburne-Sherin et al. 2019), this approach cannot be adopted for tutorials or showcase notebooks. Indeed, we want the newcomers and learners to be able to extract whatever function or workflow they find useful in the notebooks and apply them seamlessly in their own research. This cannot be done with outdated modules or language versions and calls for a maintenance workflow.

Here, we compile a list of tips and tricks for repositories of Python notebooks that we illustrate with the Euro-VO/ESCAPE tutorials repository (Marchand et al. 2022). We focus on reducing the most frequent errors measured in the study of (Pimentel et al. 2019), `ImportError`, `NameError`, and `ModuleNotFoundError`, clearly highlighting the need of declaring build and dependencies and of basic testing of the software execution. Our workflows are deployed GitHub, but are applicable to local git repositories

[1]see daily GitHub API query of the nbestimate project `https://github.com/parente/nbestimate`

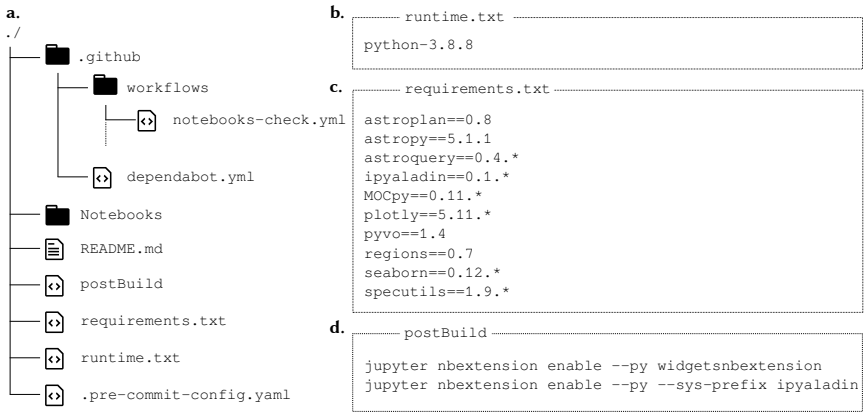

Figure 1. **Providing environment specifications. (a.)** File structure of a repository hosted on GitHub with **(b.)** a runtime.txt file providing a python version, **(c.)** a requirements.txt file with a list of dependencies, and **(d.)** a postBluid file containing bash commands to be executed once the environment is set up.

and other hosting platforms with the exception of the dependency bot and the pytest check that uses the GitHub actions and would need adaptation for other tools.

2. Providing enough information to build virtual environments

The minimal set of information needed to run a notebook is a compatible Python version, a declared set of dependencies, and any specific module or widget installation steps (Wilson et al. 2017). Since notebooks format does not encode the version requirements, this information is often lacking (Wang et al. 2021). Moreover, environment specifications are not only useful to know how to execute the notebooks on a personal machine, but also for automatic containerization or environment building tools such as repo2docker or the virtual science analysis platforms that are currently developed by different scientific projects.[2] Different conventions exist for human and machine readable environment files, with all of them requiring to add specific files in the root directory (Fig 1a.). In this project, we chose to use a combination of `runtime.txt` to provide a compatible Python version, `requirement.txt` to declare dependencies and `postBuild` to specify bash commands to be executed once the virtual environment is set, as illustrated in figure 1b., c. and d. The most important point in these three files is that the `requirements.txt` specifies versions explicitly, with the exception of the minor release wildcard * to allow for bug fixes but not for API modifications.

Keeping dependencies up-to-date. Declaration of dependencies is not sufficient. The notebooks should also use recent modules and libraries to avoid that the tutorials or the examples end up outdated. This would prevent any interested reader from extracting a function or piece of software from the notebooks and applying it on their own research environment. However, this maintenance of up-to date modules is a gar-

[2] see `https://github.com/jupyterhub/repo2docker` and `https://notebooks.egi.eu`

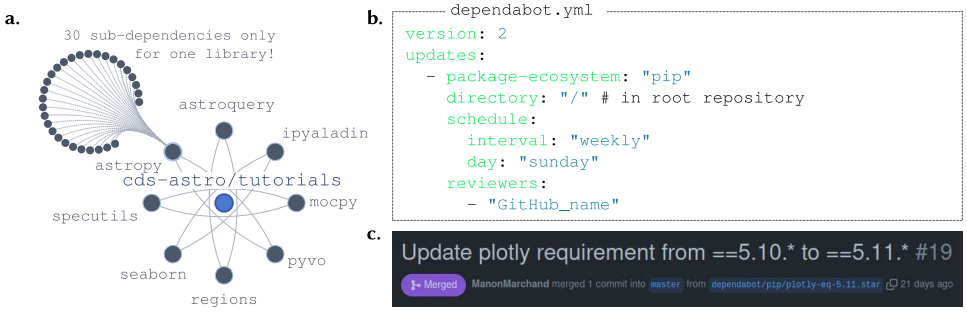

Figure 2. **Dependencies management (a.)** With the exponential number of dependencies of any project, dependency management requires bots that can be configured **(b.)** in an easy workflow **(c.)** to trigger an automatic pull request each time a new compatible dependency is available in a specified package index – such as the pip registry for Python projects.

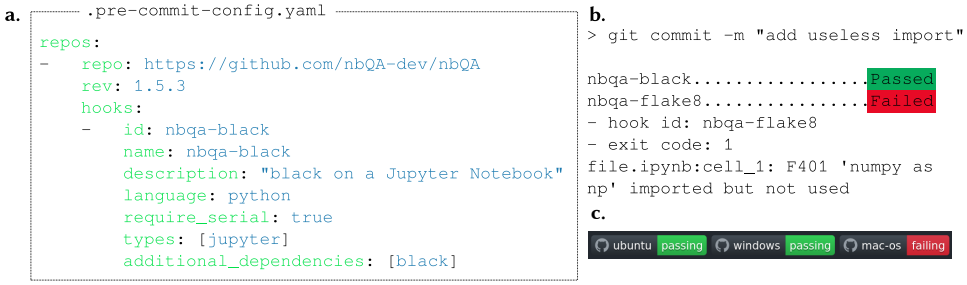

Figure 3. **Testing framework (a.)** Configuration files for a pre-commit running black – a python formater – **(b.)** Example of a commit rejected by flake8 because one of the notebooks imports an unused module. **(c.)** The results of the GitHub actions are automatically updated in the repository README page.

gantuan task due to the large number of dependencies of any python project, see Fig2a. We manage the dependencies with DependaBot,[3] a tool available for GitHub, GitLab and Azure DevOps. It only requires adding a configuration file in the `.github` folder, see fig 2b., and allowing the bot in GitHub settings in the security section. In our case, the bot is instructed to scan every week for a more recent compatible set of versions. This process opens automatic pull requests, as shown in figure 2c. These pull requests also trigger our testing workflows, as described in the following section. It allows to accept in one click if every test is successful or to investigate the needed changes of bug fixes to support the last version of dependencies.

3. Testing framework

Catch syntax and coding errors with git before pushing. Every git repository has a `.git/hooks/` folder that contains scripts executed at every commit, push or any other git event. In our repository, we used pre-commit that can be configured in a

[3] `https://github.com/dependabot`

`.pre-commit-config.yaml` file (see fig 1a. and 3a.) and installed via the pip package manager. We run the Python formatter Black and the code quality checker Flake8 through all the notebooks modified in each commits [4]. Figure 3b. shows a rejection message for a commit with an unused module import in one notebook. This framework should prevent most of the `NameError` and also ensures good code quality and readability.

Multi-platform testing with GitHub actions. Once the commit goes through the pre-commit tests, it can be added to the repository. On GitHub, we can add actions in the `.github/workflows/` folder that we chose to trigger on each push and pull requests, see (Marchand et al. 2022). These tests build environments for Windows, Linux and MacOS and for all currently active python versions by using the environment information provided in the repository as in the precedent section and then runs a pytest command `pytest --nbmake -n=auto` that tests wether all notebooks can be executed and fail if any errors is raised. The results of these tests are automatically displayed on the README page of the project, see fig 3c. This last check should catch all the remaining errors not covered by the pre-commits tests.

4. Summary/Conclusion

This maintenance workflow works as an ensemble, where the information provided to build a running environment for the notebooks is used directly in the tests and where any dependency update is probed by the same tests automatically before being manually accepted. This ensures minimal human interaction while continuously testing that all notebooks are all good and running. This work can be explored and re-used from the Euro-VO/ESCAPE repository of tutorials (Marchand et al. 2022).

Acknowledgments. This work has been supported by the ESCAPE project (the European Science Cluster of Astronomy & Particle Physics ESFRI Research Infrastructures) that has received funding from the European Union's Horizon 2020 research and innovation program under the Grant Agreement n. 824064. The EOSC Future project is co-funded by the European Union Horizon Program call INFRAEOSC-03-2020 - Grant Agreement Number 101017536.

References

Clyburne-Sherin, A., Fei, X., & Green, S. A. 2019, Meta-psychology, 3
Knuth, D. E. 1984, The Computer Journal, 27, 97
Marchand, M. C., et al. 2022. URL `https://zenodo.org/record/7358070`
Perkel, J. M. 2018, Nature, 563, 145
Piccolo, S. R., & Frampton, M. B. 2016, GigaScience, 5. S13742-016-0135-4
Pimentel, J. F., Murta, L., Braganholo, V., & Freire, J. 2019, in 2019 IEEE/ACM 16th international conference on mining software repositories (MSR) (IEEE), 507
Wang, J., Li, L., & Zeller, A. 2021, in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 1622
Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., & Teal, T. K. 2017, PLOS Computational Biology, 13, 1

[4] `https://pre-commit.com`, `https://black.readthedocs.io`, and `https://github.com/pycqa`