GOOD PRACTICES FOR RESEARCH SOFTWARE DEVELOPMENT

Workshop about Good Practices for Research Software Development,
Hamburg, 19.02.2024

Tobias Schlauch <Tobias.Schlauch@DLR.de>
Institute for Software Technology
German Aerospace Center (DLR)
http://www.dlr.de/sc

HIFIS

DLR

Good Practices for Research Software Development
Why Should I Care?

- **It is for you!**
  - To easily come back to your code 6 months later
  - To enhance trust in your code and results produced with it
  - To enhance chance to reuse (parts of) your code

For me:
Sometimes even after 1 week ;)

- **It helps others!**
  - To get a better understanding of what you did
  - To easier collaborate on your code
  - To reproduce results based on your code

Good Practices for Research Software Development
Help to Enhance Reproducibility!

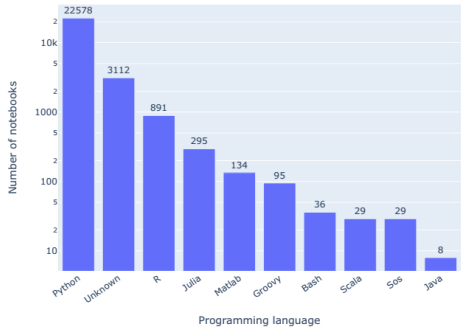


Figure 9. Programming languages of the notebooks. "Unknown" means the language kernel used was not indicated in a standard fashion.

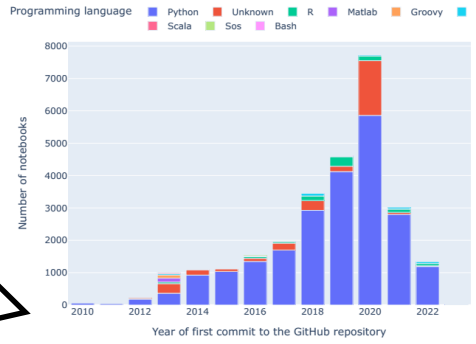


Figure 10. Relative proportion of the most frequent programming languages used in the notebooks per year. This analysis includes only programming languages with more than 7 notebooks. In 2023, we observed only 21 Python notebooks, and no other programming languages had more than 7 notebooks.

languages, as detailed in Table 1: while some (notably Matlab and Julia) showed marked increases (albeit at low absolute numbers rel-

Python versions

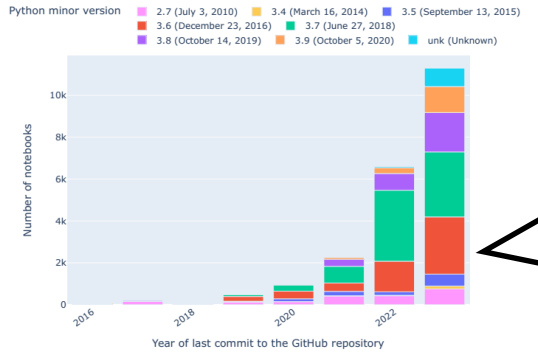


Figure 11. Python notebooks by minor Python version by year of last commit to the GitHub repository containing the notebook. In the legend, the sunset dates for each version are given.

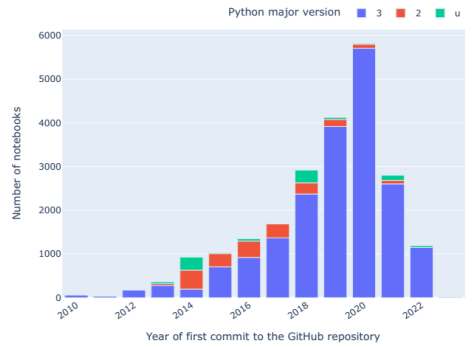


Figure 12. Python notebooks by major Python version by year of first commit to the notebook's GitHub repository.

"Where is the analysis code and how should I run it?"

"How have they calculated this statistics?"

Source: Sheeba Samuel, Daniel Mietchen, "Computational reproducibility of Jupyter notebooks from biomedical publications", https://doi.org/10.48550/arXiv.2308.07333,

Good Practices for Research Software Development
What Recommendations Exist?

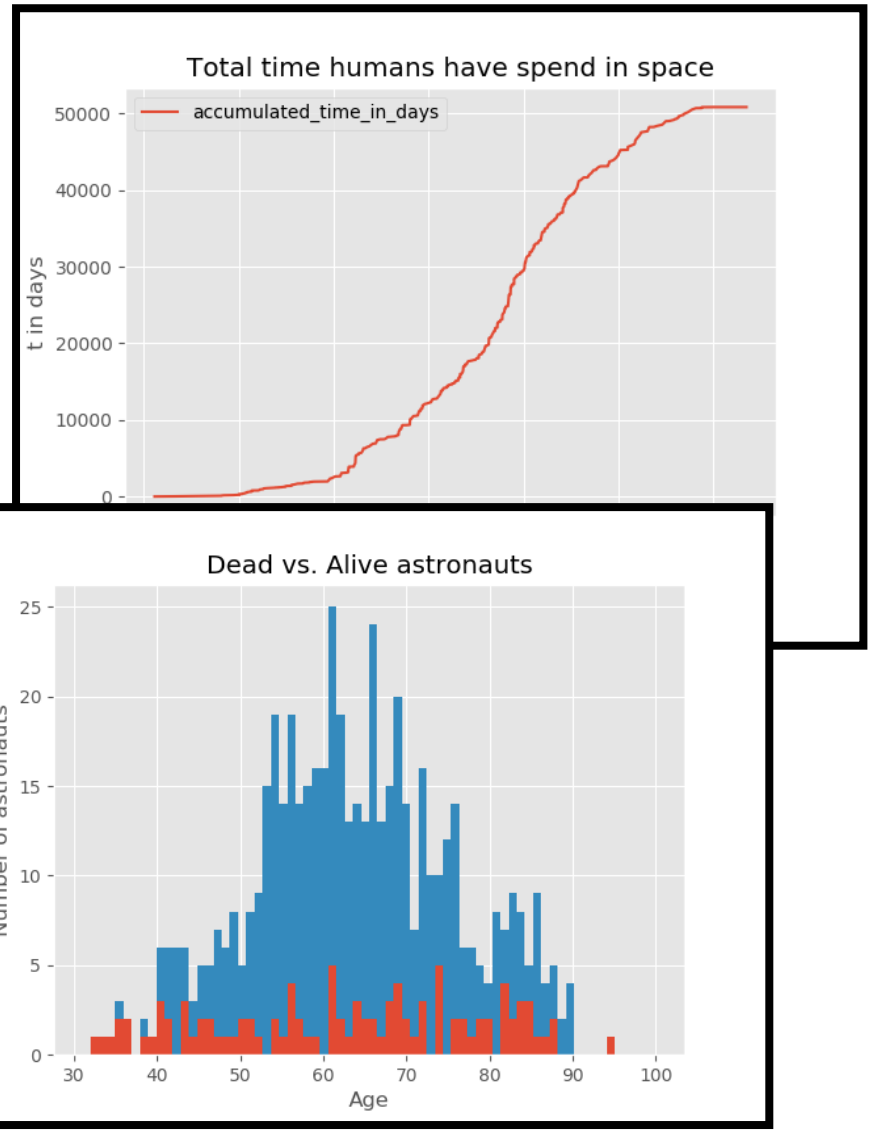
- Model Policy on Sustainable Software at the Helmholtz Centers
- DLR Software Engineering Guidelines
- Materials of the workshop Foundations of Research Software Publication
- Your research domain, journal specific, … recommendations?

➢ **There are many recommendations available! But how do I know exactly what to do …?**

OVERVIEW ABOUT TYPICAL GOOD PRACTICES

Example: Astronaut Analysis

- **Astronauts Analysis is a data publication consisting of:**
  - Data set
  - Analysis script written in Python using pandas and matplotlib
  - Result plots
- **Scenario:**
  - I created it on my own as part of my job.
  - I want to publish it with my research paper.
  - I want to make its reuse as easy as possible and make it available under an open source license.



Recommendations from the Workshop “Foundations of Research Software Publication”

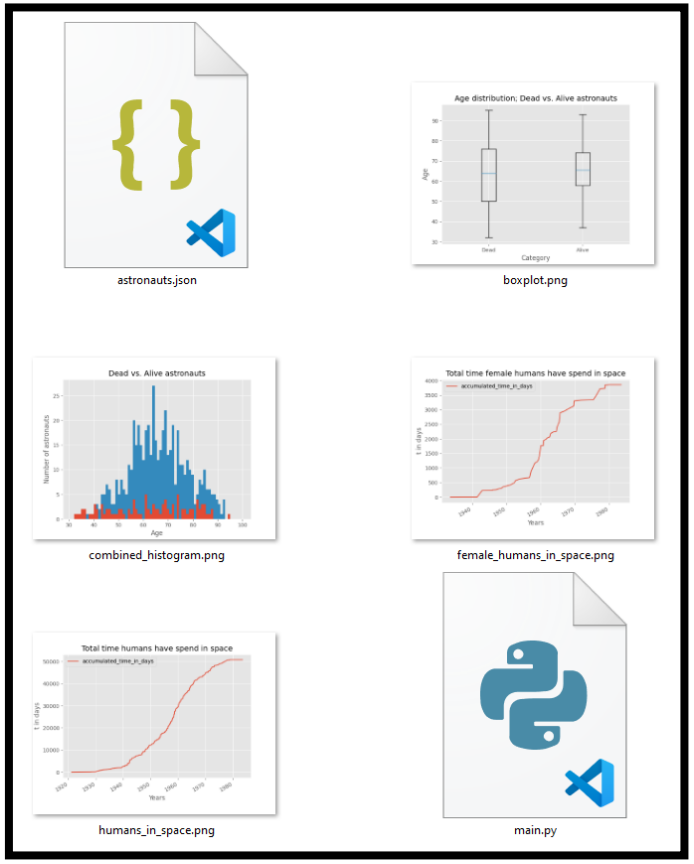
- Step 1: Put your code under version control
- Step 2: Make sure that your code is in a sharable state
- Step 3: Add essential documentation
- Step 4: Add a license
- Step 5: Make your code citable
- Step 6: Release your code

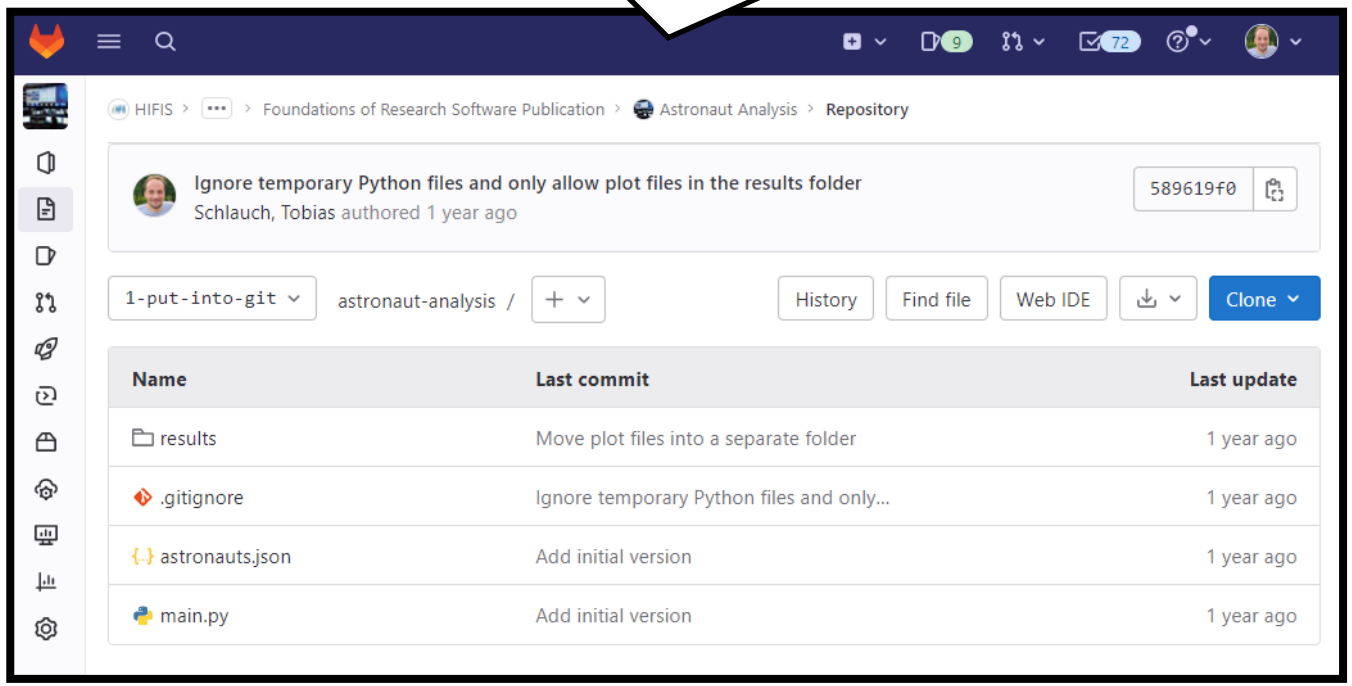
Essential aspects which you should try to already address for “internal” software!

Step 1: Put Your Code Under Version Control
Where Should I Store My Code?

Minimum: Use a local Git repository + backup

Recommended: Use a code collaboration platform





Step 1: Put Your Code Under Version Control
What Belongs in the Repository?

- **Everything to make a usable version of your code** such as:
  - Source code, documentation, build scripts, test cases, configuration files, input data, …
- **Avoid adding generated files** such as:
  - Third-party libraries, generated binaries, ...
- **How to handle large (data) files?**
  - Available could be git-lfs, git-annex, Datalad or your research data management publication repository
- **Please note:**
  - Details depend on the “product” that you manage in the Git repository
  - **.gitignore files** helps you to control what goes into your repository. See also https://gitignore.io/ for templates.

Step 1: Put Your Code Under Version Control
Key Points

- Version control helps you to keep track of changes and is the basis for collaboration with others.
- Make sure to add all relevant files (or link them properly) to the source code repository.
- **.gitignore files** helps you to specify things that you do not want to share.
- Know your version control system properly.

Step 2: Make Sure That Your Code Is in a Sharable State
General Hints

- Make sure others can run your code:
  - No dependencies on internal resources (servers, storage, licensed software, ...)
  - No absolute paths
  - Clearly state dependencies + provide required build / installation scripts (e.g.: pip-tools, poetry) => crucial aspect of reproducibility
- Organize files in a suitable directory structure (e.g.: Python Application Layouts, Good Data Practices)
- Do not share sensitive data such as passwords, user accounts, SSH keys, internal IP addresses, etc. (e.g.: gitleaks)
- Orientate on standards of your domain / community

Step 2: Make Sure That Your Code Is in a Sharable State
Improve Your Code Style and Structure

- Strive for understandable code:
  - Apply a code style – consistency is more important than convenience (e.g.: PEP8)
  - Use a consistent and light code layout
  - Structure your code in suitable "building blocks" such as functions
  - Use specific and appropriate names for all artifacts
  - Provide sufficient level of code comments
- Read code of others for inspiration
- Try to do pair programming and reviews (even if it is with your rubber duck)

Step 2: Make Sure That Your Code Is in a Sharable State
Think About Testing and Automation

- Small tests are done easily but already show effect:
  - Code linters and checkers help to find poor code snippets and help to enforce coding styles (e.g.: flake8, black)
  - Automated tests work as an executable documentation (e.g.: pytest)
- Tests offer a good starting point for your automation efforts!

Step 2: Make Sure That Your Code Is in a Sharable State
Example After Step 2

astronaut-analysis.py

```python
""" This script analysis the astronaut data set and creates different plots as result. """


from datetime import date

import pandas as pd
import matplotlib.pyplot as plt


_ASTRONAUT_DATA_FILE = "../data/astronauts.json"


##
# Data preparation functions
##
def prepare_data_set(df):
    df = rename_columns(df)
    df = df.set_index("astronaut_id")

    # Set pandas dtypes for columns with date or time
    df = df.dropna(subset=["time_in_space"])
    df["time_in_space"] = df["time_in_space"].astype(int)
    df["time_in_space"] = pd.to_timedelta(df["time_in_space"], unit="m")
    df["birthdate"] = pd.to_datetime(df["birthdate"])
    df["date_of_death"] = pd.to_datetime(df["date_of_death"])
    df.sort_values("birthdate", inplace=True)

    # Calculate extra columns from the original data
    df["time_in_space_D"] = df["time_in_space"].astype("timedelta64[D]")
    df["alive"] = df["date_of_death"].apply(is_alive)
    df["age"] = df["birthdate"].apply(calculate_age)
    df["died_with_age"] = df.apply(died_with_age, axis=1)
    return df


def rename_columns(df):
    """
    The original column naming in the data set is not useful
    for programming with pandas. So we rename it.
    """
```

- **Applied PEP8 code style**
- **Cleaned up the code**
- **Added basic testing and more ☺**

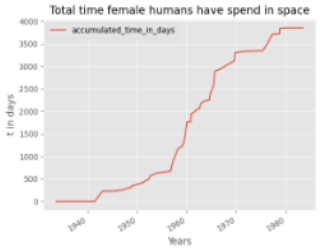
Step 2: Make Sure That Your Code Is in a Sharable State
Key Points

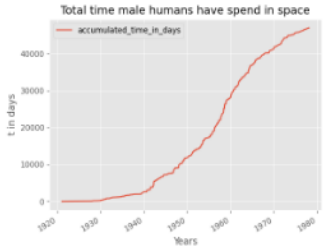
- Make sure that others can (re-)use your code
- Do not store secrets in your code repository
- Strive for understandable code
- Start introducing basic test automation

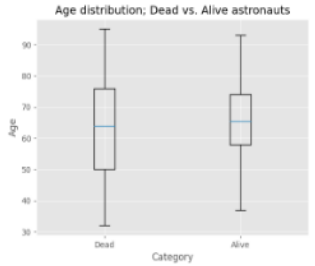
Step 3: Add Documentation

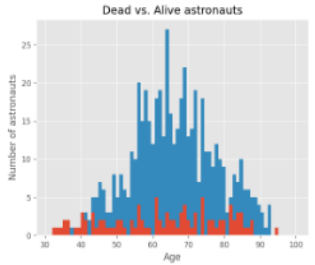
Astronaut Analysis

This analysis is based on publicly available astronauts data from Wikidata. In this context, we investigated aspects such as time humans spent in space as well as the age distribution of the astronauts.









The repository is organized as follows:

- data: Contains the astronauts data set retrieved from Wikidata
- code: Contains the astronaut analysis script
- results: Contains the resulting analysis plots

Astronaut Data

The data set has been generated using the following SPARQL query [1] (retrieval date: 2018-10-25).

You can also analyze a recent version of the astronaut data by replacing the data set and re-running the analysis script:

- Run the SPARQL query
- Download the resulting data formatted as JSON
- Replace the file `data/astronauts.json`
- Run the analysis script

Astronaut Analysis Script

The script requires Python >= 3.8 and uses the libraries pandas as well as matplotlib.

> The script has been successfully tested on Windows 10 and Linux with Python 3.8.

Typical Structure:

- **Software name**
- **Purpose**
- **Install**
- **Usage**
- **Contributing**
- **Citation Hint**
- **License**

Step 3: Add Documentation
General Hints

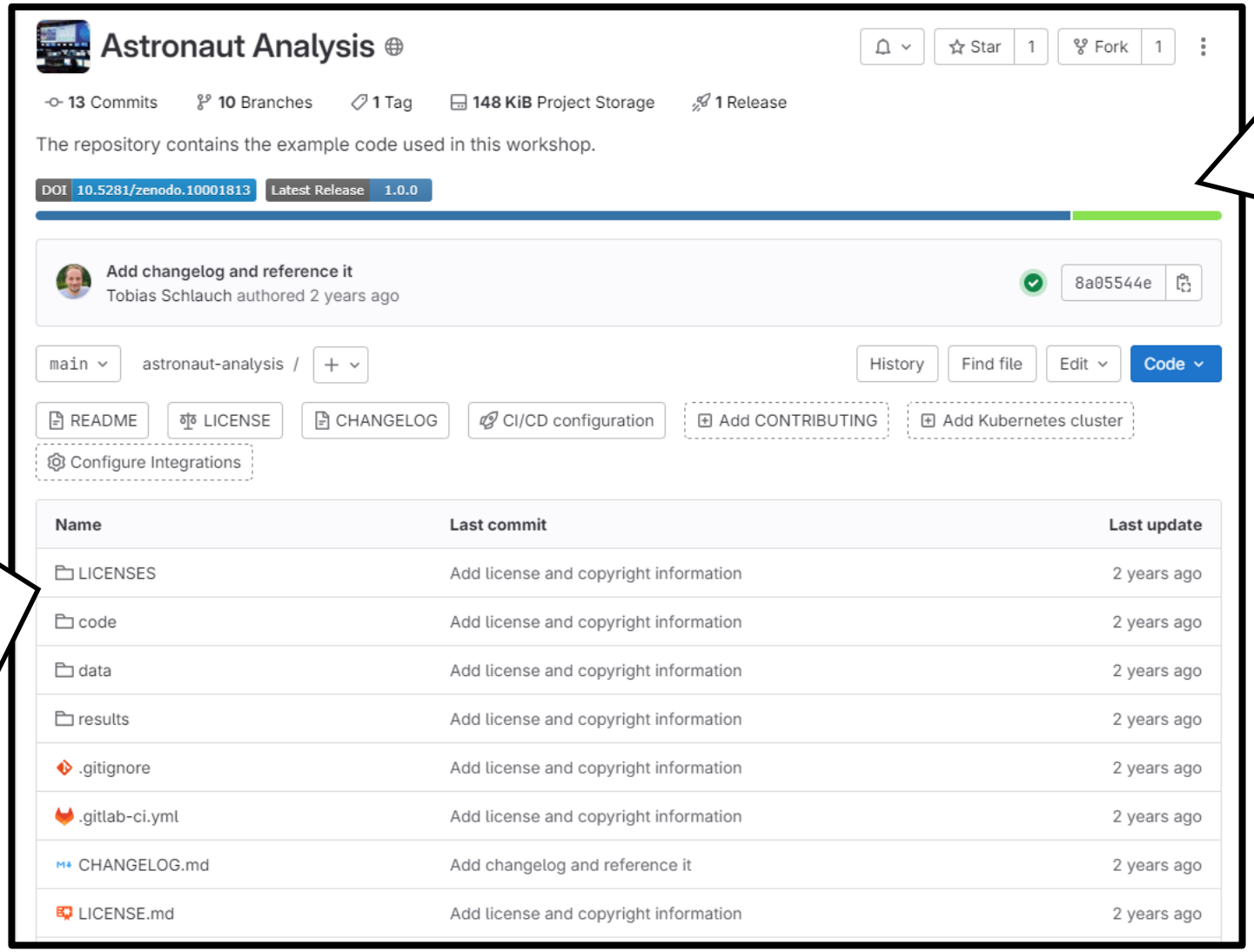
- **Mind your target groups:**
  - **Typical perspectives:** Users, contributors
  - **Users:** Installation / usage instructions, tutorials, support channels, …
  - **Contributors:** Contribution guidelines, technical overview, …
- **Think about adding typical documentation files** such as:
  - `README` (project front page), `CONTRIBUTING` (contributions guidelines), `CODE_OF_CONDUCT` (communication rules), `LICENSE` (license information), `CHANGELOG` (major changes), `CITATION` (citation metadata)
- **Please note:**
  - Markdown or another markup language is quite often used to write documentation
  - Usually, you will need additional documentation, for example, in a `docs` directory (e.g.: Sphinx, MkDocs)

Step 3: Add Documentation
Key Points

- Provide documentation for relevant target groups
- Add a **README** file as a minimum documentation to your source code repository

Astronaut Analysis Release 1.0.0

License information for code, data, results properly annotated via REUSE

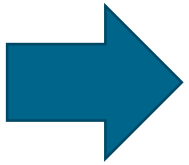


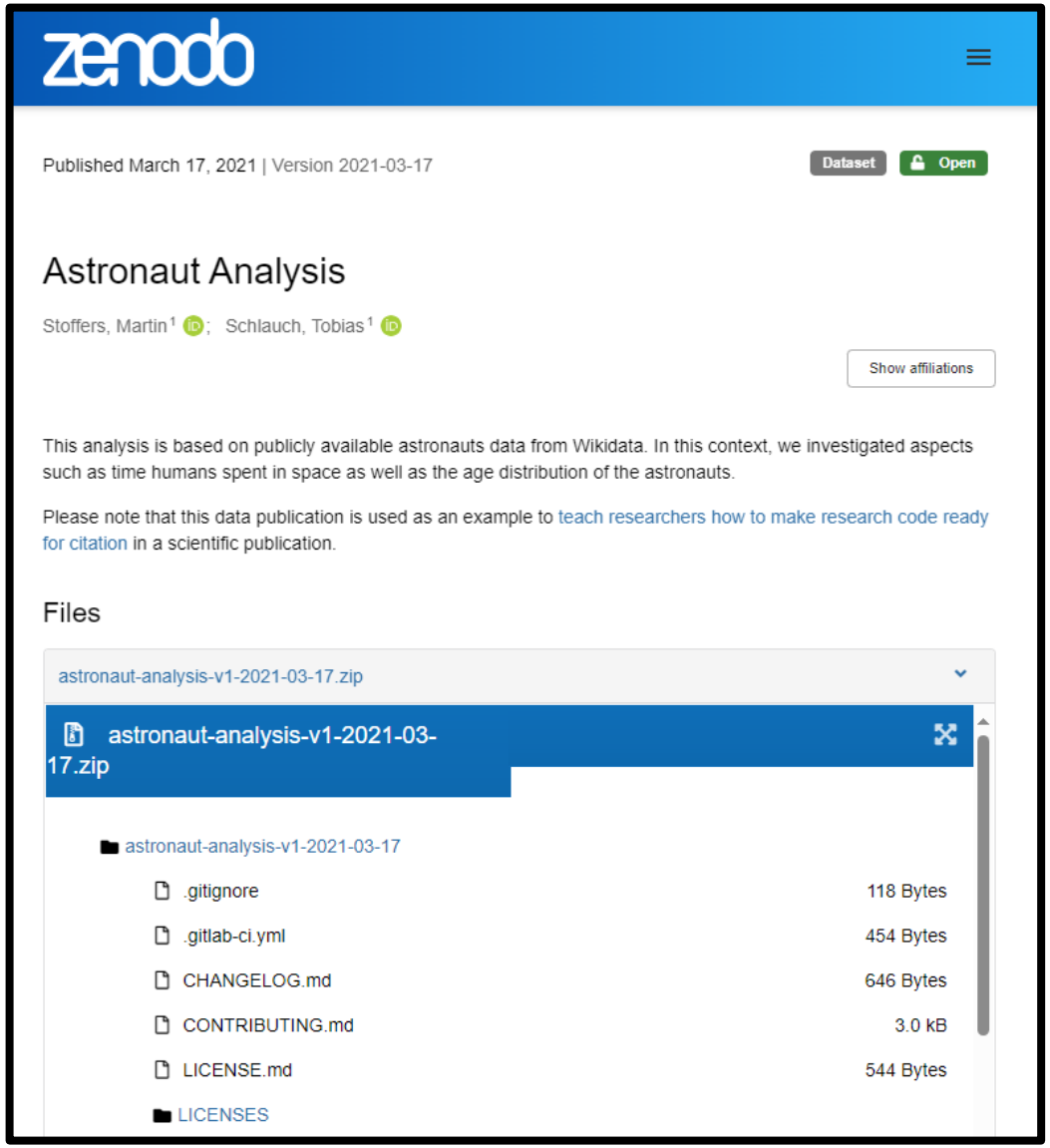
Release 1.0.0 marked as **Git tag** in the repository

Astronaut Analysis Release 1.0.0 (cont.)

DOI 10.5281/zenodo.10001813

```
cff-version: 1.2.0
title: Astronaut Analysis
message: >-
  If you use this dataset, please cite it using the
metadata
  from this file.
type: dataset
authors:
  - given-names: Martin
    family-names: Stoffers
  - given-names: Tobias
    family-names: Schlauch
identifiers:
  - type: doi
    value: 10.5281/zenodo.10001813
```





Citable Release:

- Citation metadata in Citation File Format
- DOI via Zenodo

There Are Many Recommendations Available! But How Do I Know Exactly What to Do …?

- Recommendations are typically made under certain assumptions. I.e., they leave out details and might not fit for your case directly … ☹
- Establishing detailed good practices on a **research group level** could help:
  - Similar tasks and projects make it easier to agree on relevant practices and details
  - Use generic recommendations as a starting point and leave out irrelevant aspects / add required details as needed
- “Executable” templates can help to get everyone better started:
  - Relevant tools: Cookiecutter, Cruft
  - Example: HCDC / Software Templates / Python Package · GitLab (helmholtz.cloud)

SUMMARY

Summary

- **Good practices for research software development are important:**
  - Help you and others to work on code and have trust in results produced with it
  - Enhance chances for research to be reproducible
- **Existing recommendations are made under certain assumptions and need to be tailored to the right context:**
  - Existing guidelines might be too generic
  - Presented recommendations might be in some aspects too detailed or (currently) not relevant for your specific case
- **Research group could be the right level to establish effective good practice!**

Thank you!

What are your Questions?

Email: Tobias.Schlauch@dlr.de
Mastodon: https://norden.social/@schlauch
HIFIS Mattermost: @schlauch

Copyright and License Information