Reproducible research - Preparing code to be usable by you and others in the future

Have you ever spent days **trying to repeat the results from few weeks or months ago**? Or you have to do paper revisions, but you just can’t get the results to match up? It’s unpleasant for both you and science.

In this lesson we will explore different methods and tools for better reproducibility in research software and data. We will demonstrate how version control, workflows, containers, and package managers can be used to **record reproducible environments and computational steps** for our future selves and others.

Learning outcomes

By the end of this lesson, learners should:

- Be able to apply well organized directory structure for their project
- Understand that code can have dependencies, and know how to document them
- Be able to document computational steps, and have an idea when it can be useful
- Know about use cases for containers

Prerequisites

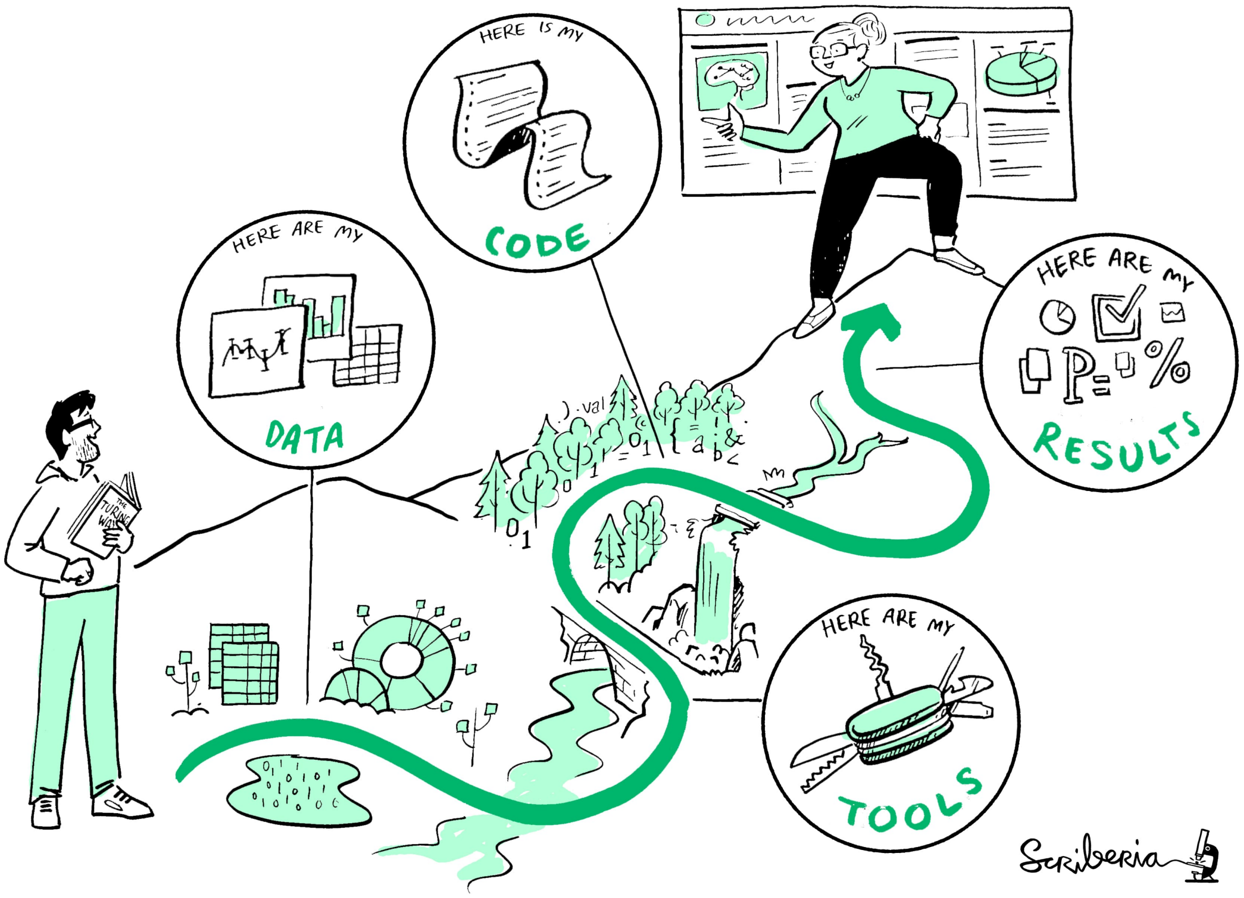
You need to install Git, Python, and Snakemake (part of CodeRefinery Conda environment).

If you wish to follow in the terminal and are new to the command line, we recorded a short shell crash course.

Introduction - How it all connects

Instructor note

- 10 min teaching/discussion
- 0 min exercises



[The Turing Way project illustration by Scriberia. Used under a CC-BY 4.0 licence. DOI: https://zenodo.org/records/13882307]

This workshop is all about reproducibility - from a computational perspective

This section connects the steps above to the CodeRefinery workshop lessons.

“Here is my code”

- **Version control with git** with focus on collaboration
- **Social coding**: What can you do to get credit for your code and to allow reuse
- **Documentation**: How to let others or future you know about your thoughts and how to use your code
- **Jupyter Notebooks**: A tool to write and share executable notebooks and data visualization
- **Automated testing**: Preventing yourself and others from breaking your functioning code
- **Modular code development**: Making reusing parts of your code easier

“Here are my tools”

This lesson on general **Reproducibility**: Preparing code to be usable by you and others in the future

This includes organizing your projects on your own computer and recording your computational steps, dependencies and computing environment.

We will also mention a few tools and platforms for sharing data (**“Here is my data”**) and research outputs (**“Here are my results”**) in the **social coding** lesson, but they are not the focus of this workshop.

Small steps towards reproducible research

If this is all new to you, it may feel quite overwhelming.

Our recommendation: Don’t worry! Focus on “good enough” instead of perfect.

To start, pick one topic that seems reasonable to implement for your current project. Something that helps YOU right now. This may be something you may have to implement due to requirements from your funders or the journal where you want to publish your research. Use their requirements as a checklist and find tools that feel comfortable for you.

A great way to see what are the really important things to implement is to meet with a colleague, exchange codes and try to run each others code. Every question your colleague has to ask from you about your code gives a hint on where you may need to improve.

Keeping a “log book” while working on your own code also serves as a great basis for making your code more reproducible. Can you use any of the tools and techniques learned in this workshop to share parts of your log book with others to help them run your code?

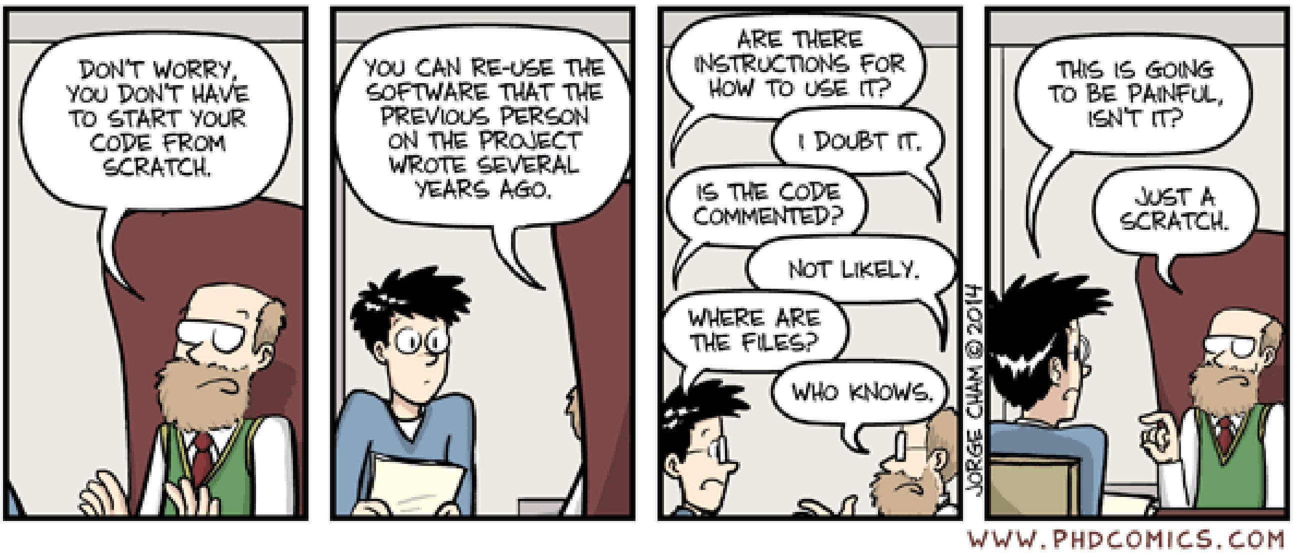
Motivation

ⓘ Objectives

- Understand why we are talking about reproducibility in this workshop

Instructor note

- 10 min teaching/discussion

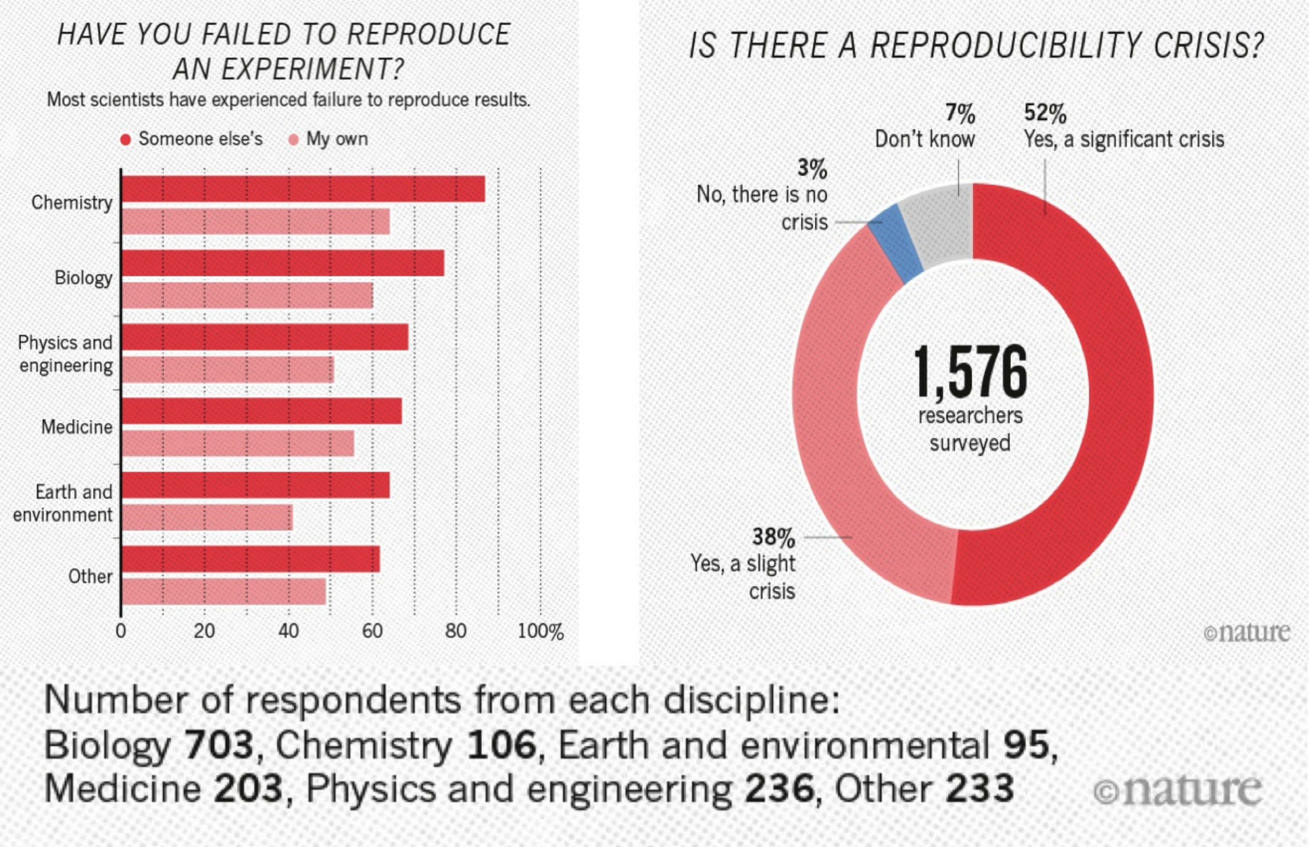

DON'T WORRY, YOU DON'T HAVE TO START YOUR CODE FROM SCRATCH.
YOU CAN RE-USE THE SOFTWARE THAT THE PREVIOUS PERSON ON THE PROJECT WROTE SEVERAL YEARS AGO.
ARE THERE INSTRUCTIONS FOR HOW TO USE IT?
I DOUBT IT.
IS THE CODE COMMENTED?
NOT LIKELY.
WHERE ARE THE FILES?
WHO KNOWS.
JORGE CHAM © 2014
THIS IS GOING TO BE PAINFUL, ISN'T IT?
JUST A SCRATCH.
WWW.PHDCOMICS.COM

❗ A scary anecdote

- A group of researchers obtain great results and submit their work to a high-profile journal.
- Reviewers ask for new figures and additional analysis.
- The researchers start working on revisions and generate modified figures, but find inconsistencies with old figures.
- The researchers can't find some of the data they used to generate the original results, and can't figure out which parameters they used when running their analyses.
- The manuscript is still languishing in the drawer …

Why talking about reproducible research?

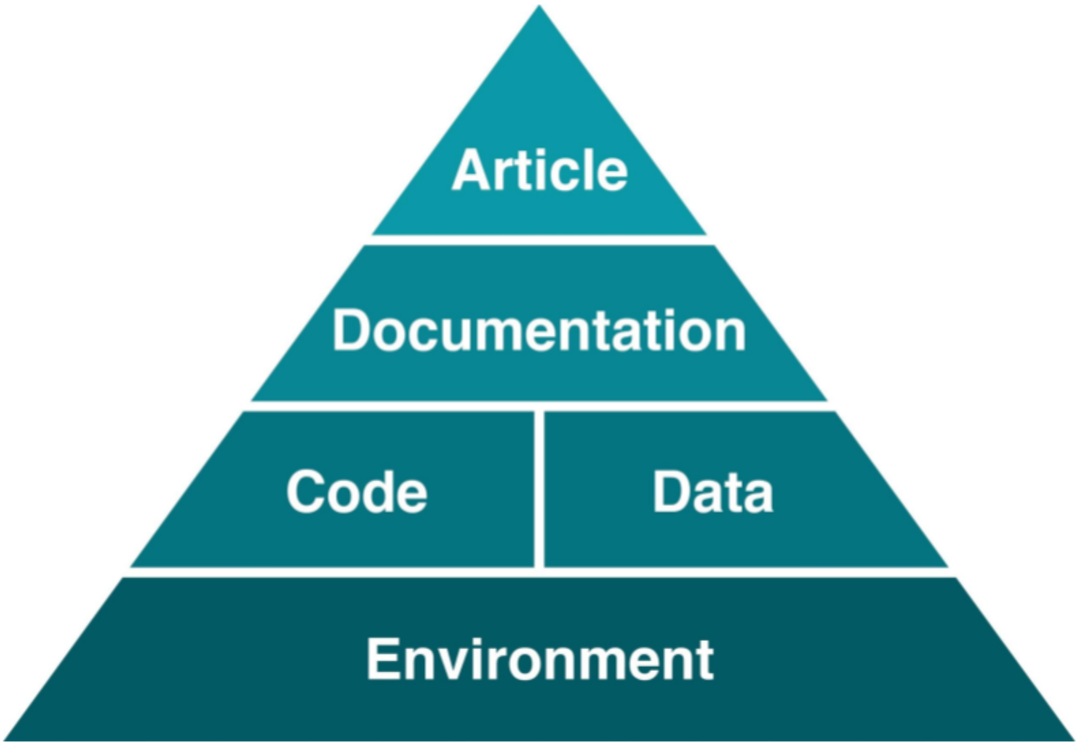
A 2016 survey in Nature revealed that irreproducible experiments are a problem across all domains of science:



This study is now few years old but the highlighted problem did not get smaller.

Levels of reproducibility

A published article is like the top of a pyramid. It rests on multiple levels, each contributing to its reproducibility.



[Steeves, Vicky (2017) in “Reproducibility Librarianship,” Collaborative Librarianship: Vol. 9: Iss. 2, Article 4. Available at: https://digitalcommons.du.edu/collaborativelibrarianship/vol9/iss2/4]

This also means that you can think about it from the beginning of your research life cycle!

💬 Discuss in collaborative document or with your team members

```
- What are your experiences re-running or adjusting a script or a figure you
  created few months ago?
  - ...
  - ...
  - (share your experience)

- Have you continued working from a previous student's
  script/code/plot/notebook? What were the biggest challenges?
  - ...
  - ...
  - (share your experience, but constructively)
```

❗ Keypoints

- Without reproducibility in scientific computing, everyone would have to start a new project / code from scratch.

Organizing your projects

❗ Objectives

- Understand how to organize research projects
- Get an overview of tools for collaborative and version controlled manuscripts

Instructor note

- 10 min teaching incl. discussions

One of the first steps to make your work reproducible is to organize your projects well. Let’s go over some of the basic things which people have found to work (and not to work).

Directory structure for projects

- Project files in a **single directory**
- **Different projects** should have **separate directories**
- Use **consistent and informative directory structure**

  - Avoid spaces in directory and file names – use `-`, `_` or CamelCase instead (nicer for computers to handle)
- If you need to separate public/private directories,
  - put them separately in public and private Git repositories, or
  - use `.gitignore` to exclude the private information from being tracked
- Add a **README file** to describe the project and instructions on reproducing the results
- If you want to use the **same code in multiple projects**, host it on GitHub (or similar) and clone it into each of your project directories

A project directory can look something like this:

```
project_name/
├── README.md             # overview of the project
├── data/                 # data files used in the project
│   ├── README.md         # describes where data came from
│   └── sub-directory/       # may contain subdirectories
├── processed_data/       # intermediate files from the analysis
├── manuscript/           # manuscript describing the results
├── results/              # results of the analysis (data, tables, figures)
├── src/                  # contains all code in the project
│   ├── LICENSE           # license for your code
│   ├── requirements.txt  # software requirements and dependencies
│   └── ...
└── doc/                  # documentation for your project
    ├── index.rst
    └── ...
```

Tracking source code, data, and results

- All code is version controlled and goes in the `src/` or `source/` directory
- Include appropriate LICENSE file and information on software requirements
- You can also version control data files or input files under `data/`
  - If data files are too large (or sensitive) to track, untrack them using `.gitignore`
- Intermediate files from the analysis are kept in `processed_data/`
- Consider using Git tags to mark specific versions of results (version submitted to a journal, dissertation version, poster version, etc.):

  ```
  $ git tag -a thesis-submitted -m "this is the submitted version of my thesis"
  ```

Check the Git-intro lesson for a reminder.

Some tools and templates

- R devtools
- Python cookiecutter template
- Reproducible research template by the Turing Way

More tools and templates in Heidi Seibolds blog.

Excursion: Reproducible publications

Discussion on collaborative writing of academic papers

💬 Discuss in the collaborative document:

```
- How do you collaborate on writing academic papers?
  - ...
  - ...
  - (share your experience)

- How do you handle collaborative issues e.g. conflicting changes?
  - ...
  - ...
  - (share your experience)
```

-> Consider using **version control for manuscripts** as well. It may help you when keeping track of edits + if you sync it online then you don't have to worry about losing your work.

Version control does not have to mean git, but could also mean using "tracking changes" in tools like Word, Google Docs, or Overleaf (find links below).

Tools for collaborative writing and version control of manuscripts

Git **can** be used to collaborate on manuscripts written in, e.g., LaTeX and other text-based formats. However it might not always be the most convenient. Other tools exist to make the process more enjoyable:

You can **collaboratively gather notes** using self-hosted or public instances of tools like HedgeDoc and Etherpad or use online options like HackMD, Google Docs or the Microsoft online tools for easy and efficient collaboration.

To format your notes into a manuscript, you can use Word-like online editors or tools like Overleaf (LaTeX) or Typst (markdown). Most of the tools in this section even provide a git integration.

Manubot offers another way to turn your written word into a fully rendered manuscript using GitHub.

Executable manuscripts

You may also want to consider writing an executable manuscript using tools like Jupyter Notebooks hosted on Binder, Quarto, Authorea or Observable, to name a few.

Resources on research compendia

- About research compendia at the Turing Way
- “Research compendia”: a set of good practices for reproducible data analysis in R, but much is transferable to other languages.
- rrtools: instructions, templates, and functions for writing a reproducible article or report with R.
- …

Keypoints

- An organized project directory structure helps with reproducibility.
- Also think about version control for writing your academic manuscripts.

Recording computational steps

Objectives

- Understand why and when a workflow management tool can be useful

❓ Questions

- You have some steps that need to be run to do your work. How do you actually run them? Does it rely on your own memory and work, or is it reproducible? **How do you communicate the steps** for future you and others?
- How can we create a reproducible workflow?

Instructor note

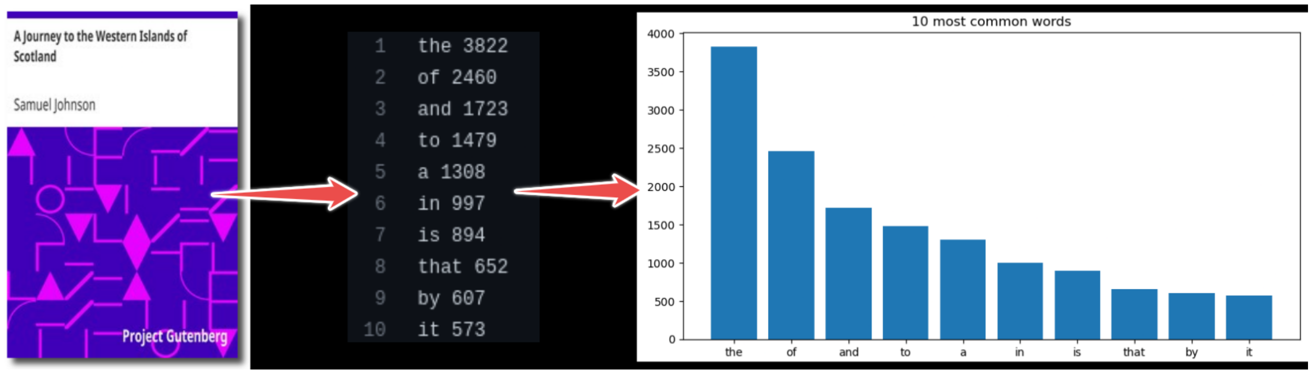
- 5 min teaching
- 15 min exercise/demo

Several steps from input data to result

The following material is partly derived from a HPC Carpentry lesson.

In this episode, we will use an example project which finds most frequent words in books and plots the result from those statistics. In this example we wish to:

1. Analyze word frequencies using code/count.py for 4 books (they are all in the data directory).
2. Plot a histogram using plot/plot.py.



```
the 3822
of 2460
and 1723
to 1479
a 1308
in 997
is 894
that 652
by 607
it 573
```

Example (for one book only):

```
$ python code/count.py data/isles.txt > statistics/isles.data
$ python code/plot.py --data-file statistics/isles.data --plot-file plot/isles.png
```

Another way to analyze the data would be via a graphical user interface (GUI), where you can for example drag and drop files and click buttons to do the different processing steps.

Both of the above (single line commands and simple graphical interfaces) are tricky in terms of reproducibility. We currently have two steps and 4 books. But **imagine having 4 steps and 500 books**. How could we deal with this?

As a first idea we could express the workflow with a script. The repository includes such script called `run_all.sh`.

We can run it with:

```
$ bash run_all.sh
```

This is **imperative style**: we tell the script to run these steps in precisely this order, as we would run them manually, one after another.

Discussion

- What are the advantages of this solution compared to processing all one by one?
- Is the scripted solution reproducible?
- Imagine adding more steps to the analysis and imagine the steps being time consuming. What problems do you anticipate with a scripted solution?

✓ Solution

The advantage of this solution compared to processing one by one is more automation: We can generate all. This is not only easier, it is also less error-prone.

Yes, the scripted solution can be reproducible. But could you easily run it, e.g., on a Windows computer?

If we had more steps and once steps start to be time-consuming, a limitation of a scripted solution is that it tries to run all steps always. Rerunning only part of the steps or only part of the input data requires us to outcomment or change lines in our script in between runs which can again become tedious and error-prone.

Workflow tools

Sometimes it may be helpful to go from imperative to declarative style. Rather than saying “do this and then that” we describe dependencies between steps, but we let the tool figure out the order of steps to produce results.

Example workflow tool: Snakemake

Snakemake (inspired by GNU Make) is one of many tools to create reproducible and scalable data analysis workflows. Workflows are described via a human readable, Python based language. Snakemake workflows scale seamlessly from laptop to cluster or cloud, without the need to modify the workflow definition.

A demo

⚙ Preparation

The exercise (below) and pre-exercise discussion uses the word-count repository (https://github.com/coderefinery/word-count) which we need to clone to work on it.

If you want to do this exercise on your own, you can do so either on your own computer (follow the instructions in the bottom right panel on the CodeRefinery installation instruction page), or the Binder cloud service:

On your own computer:

- Install the necessary tools
- Activate the CodeRefinery Conda environment with `conda activate coderefinery`.
- Clone the word-count repository:

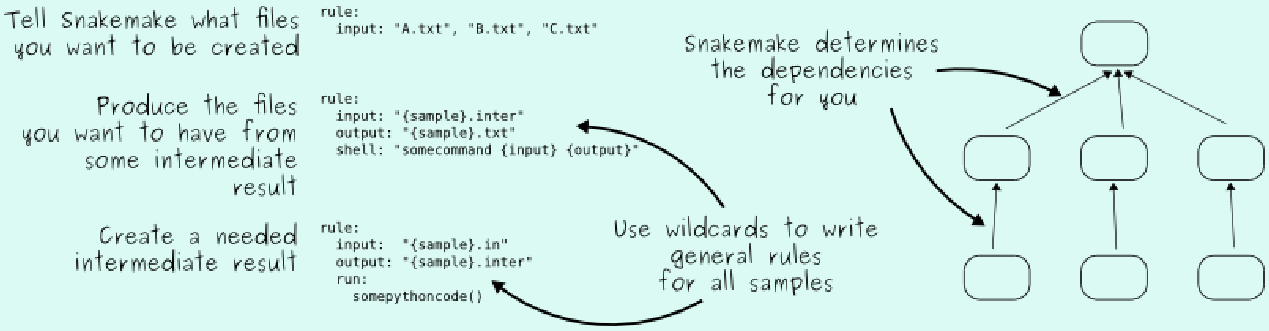
```
$ git clone https://github.com/coderefinery/word-count.git
```

On Binder:

We can also use the cloud service Binder to make sure we all have the same computing environment. This is interesting from a reproducible research point of view and it's explained further in the Jupyter lesson how this is even possible.

- Go to https://github.com/coderefinery/word-count and click on the "launch binder" badge in the README.
- Once it gets started, you can open a new **Terminal** from the Launcher or via **File > New > Terminal**.

✍️ Workflow-1: Workflow solution using Snakemake


Tell Snakemake what files you want to be created
Produce the files you want to have from some intermediate result
Create a needed intermediate result
Snakemake determines the dependencies for you
Use wildcards to write general rules for all samples

```
rule:
  input: "A.txt", "B.txt", "C.txt"
```

```
rule:
  input: "{sample}.inter"
  output: "{sample}.txt"
  shell: "somecommand {input} {output}"
```

```
rule:
  input:  "{sample}.in"
  output: "{sample}.inter"
  run:
    somepythoncode()
```

Somebody wrote a Snakemake solution in the Snakefile:

```
# a list of all the books we are analyzing
DATA = glob_wildcards('data/{book}.txt').book

rule all:
    input:
        expand('statistics/{book}.data', book=DATA),
        expand('plot/{book}.png', book=DATA)

# count words in one of our books
rule count_words:
    input:
        script='code/count.py',
        book='data/{file}.txt'
    output: 'statistics/{file}.data'
    shell: 'python {input.script} {input.book} > {output}'

# create a plot for each book
rule make_plot:
    input:
        script='code/plot.py',
        book='statistics/{file}.data'
    output: 'plot/{file}.png'
    shell: 'python {input.script} --data-file {input.book} --plot-file {output}'
```

We can see that Snakemake uses **declarative style**: Snakefiles contain rules that relate targets (`output`) to dependencies (`input`) and commands (`shell`).

Steps:

1. Clone the example to your computer: `$ git clone https://github.com/coderefinery/word-count.git`
2. Study the Snakefile. How does it know what to do first and what to do then?
3. Try to run it. Since version 5.11 one needs to specify number of cores (or jobs) using `-j`, `--jobs` or `--cores`:

   ```
   $ snakemake --delete-all-output -j 1
   $ snakemake -j 1
   ```

   The `--delete-all-output` part makes sure that we remove all generated files before we start.

4. Try running `snakemake` again and observe that and discuss why it refused to rerun all steps:

   ```
   $ snakemake -j 1

   Building DAG of jobs...
   Nothing to be done (all requested files are present and up to date).
   ```

5. Make a tiny modification to the plot.py script and run `$ snakemake -j 1` again and observe how it will only re-run the plot steps.
6. Make a tiny modification to one of the books and run `$ snakemake -j 1` again and observe how it only regenerates files for this book.
7. Discuss possible advantages compared to a scripted solution.
8. **Question for R developers**: Imagine you want to rewrite the two Python scripts and use R instead. Which lines in the Snakefile would you have to modify so that it uses your R code?
9. If you make changes to the Snakefile, validate it using `$ snakemake --lint`.

✔ Solution

- 2: Start with “all” and look what it depends on. Now search for rules that have these as output. Look for their inputs and search where they are produced. In other words, search backwards and build a graph of dependencies. This is what Snakemake does.
- 4: It can see that outputs are newer than inputs. It will only regenerate outputs if they are not there or if the inputs or scripts have changed.

- 7: It only generates steps and outputs that are missing or outdated. The workflow does not run everything every time. In other words if you notice a problem or update information "half way" in the analysis, it will only re-run what needs to be re-run. Nothing more, nothing less. Another advantage is that it can distribute tasks to multiple cores, off-load work to supercomputers, offers more fine-grained control over environments, and more.
- 8: Probably only the two lines containing "shell".

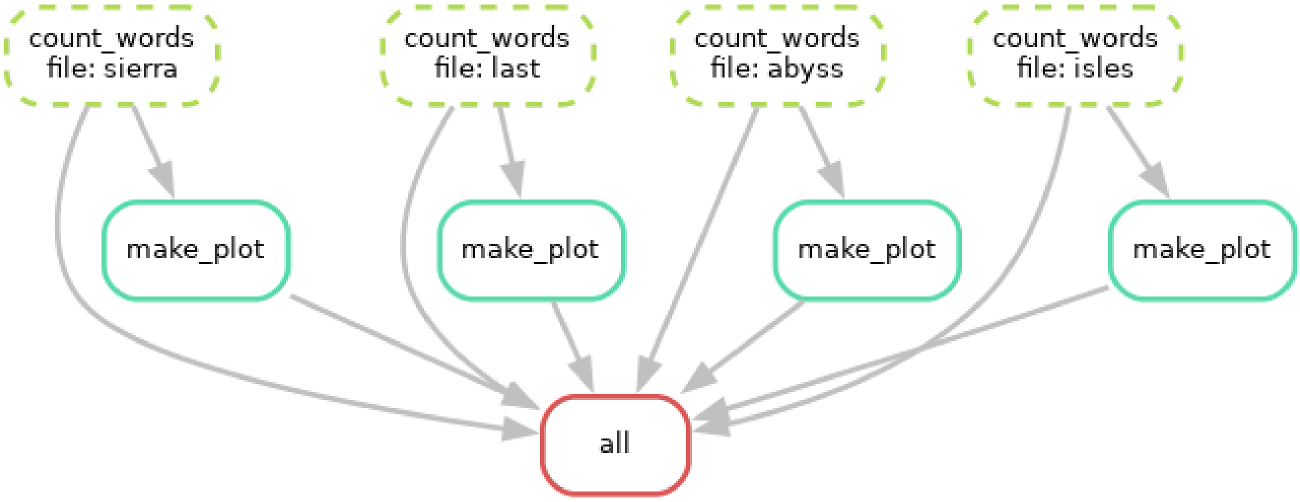
Visualizing the workflow

We can visualize the directed acyclic graph (DAG) of our current Snakefile using the `--dag` option, which will output the DAG in `dot` language.

Note: This requires the Graphviz software, which can be installed by `conda install graphviz`.

```
$ snakemake -j 1 --dag | dot -Tpng > dag.png
```

Rules that have yet to be completed are indicated with solid outlines, while already completed rules are indicated with dashed outlines.



Why Snakemake?

- Gentle **learning curve**.
- Free, open-source, and **installs easily** via conda or pip.
- **Cross-platform** (Windows, MacOS, Linux) and compatible with all High Performance Computing (HPC) schedulers: same workflow works without modification and scales appropriately whether on a laptop or cluster.
- If several workflow steps are independent of each other, and you have multiple cores available, Snakemake can run them **in parallel**.

- It is possible to define **isolated software environments** per rule, e.g. by adding `conda: 'environment.yml'` to a rule.
- It is also possible to run workflows in Docker or Apptainer **containers**, e.g. by adding `container: 'docker://some-org/some-tool#2.3.1'` to a rule.
- Heavily used in bioinformatics, but is **completely general**.
- Nice functionality for archiving the workflow, see: the official Snakemake documentation

Tools like Snakemake help us with **reproducibility** by supporting us with **automation**, **scalability** and **portability** of our workflows.

Similar tools

- Make
- Nextflow
- Task
- Common Workflow Language
- Many specialized frameworks exist.
- Book on building reproducible analytical pipelines with R
- {targets} R package - make-like pipeline tool for R

Keypoints

- Computational steps can be recorded in many ways.
- Workflow tools can help if there are many steps to be executed and/or many datasets to be processed.

Recording dependencies

Objectives

- Understand what dependency management tools can be useful for
- Discuss environment/requirements files in the context of reusability and reproducibility

❓ Questions

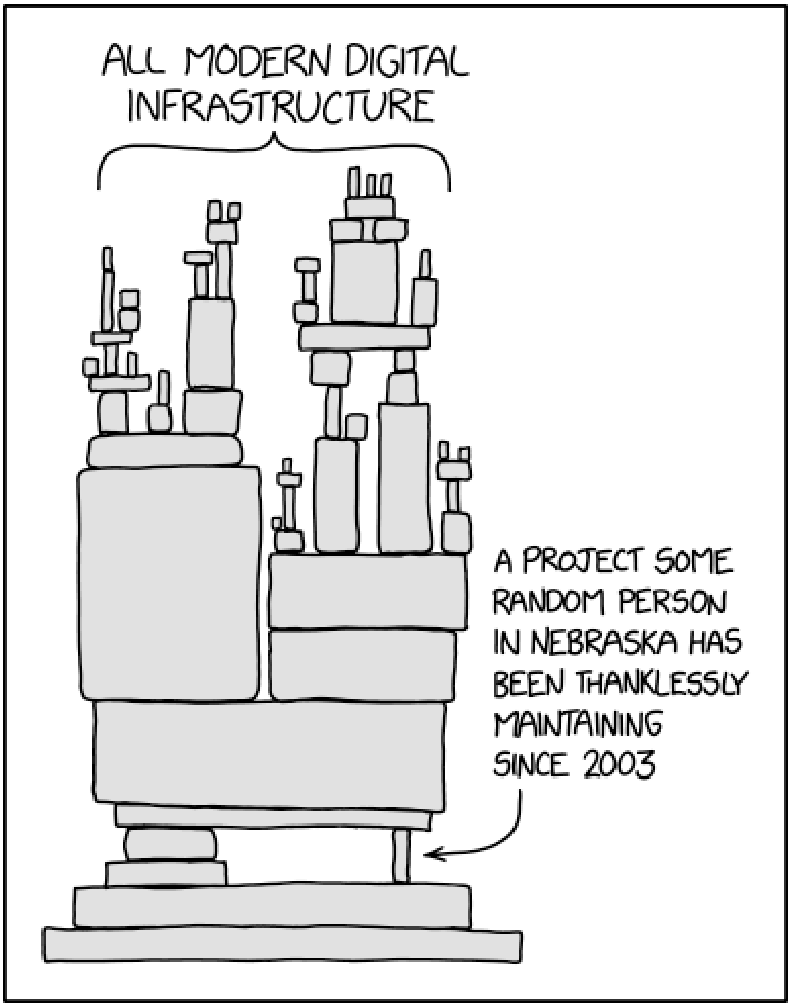
- How can we communicate different versions of software dependencies?

Instructor note

- 10 min teaching
- 10 min exercise/demo

Our codes often depend on other codes that in turn depend on other codes …

- **Reproducibility**: We can version-control our code with Git but how should we version-control dependencies? How can we capture and communicate dependencies?
- **Dependency hell**: Different codes on the same environment can have conflicting dependencies.


ALL MODERN DIGITAL INFRASTRUCTURE
A PROJECT SOME RANDOM PERSON IN NEBRASKA HAS BEEN THANKLESSLY MAINTAINING SINCE 2003

From xkcd - dependency. Another image that might be familiar to some of you working with Python can be found on xkcd - superfund.

💬 **Kitchen analogy**

- Software <-> recipe
- Data <-> ingredients
- Libraries <-> pots/tools

Cooking recipe in an unfamiliar language [Midjourney, CC-BY-NC 4.0]

When we create recipes, we often use tools created by others (libraries) [Midjourney, CC-BY-NC 4.0]

Dependency and environment management

Tools like **Conda, Anaconda, pip, virtualenv, Pipenv, pyenv, Poetry, renv** and files to record dependencies like **requirements.txt** and **environment.yml** try to solve the following problems:

- **Defining a specific set of dependencies**
- **Installing those dependencies** mostly automatically
- **Recording the versions** for all dependencies
- **Isolate environments**
  - On your computer for projects, so they can use different software
  - Isolate environments on computers with many users (and allow self-installations)
- Using **different package versions** per project (also, e.g., Python/R versions)

- Provide tools and services to **share packages**

Isolated environments are also useful because they help you make sure that you know your dependencies!

If things go wrong, you can delete and re-create - much better than debugging. The more often you re-create your environment, the more reproducible it is.

Exercise / Demo

✍ Dependencies-1: Time-capsule of dependencies

Situation: 5 students (A, B, C, D, E) wrote a code that depends on a couple of libraries. They uploaded their projects to GitHub. We now travel 3 years into the future and find their GitHub repositories and try to re-run their code before adapting it.

Answer in the collaborative document:

- Which version do you expect to be easiest to re-run? Why?
- What problems do you anticipate in each solution?

Conda Python virtualenv R Matlab

A: You find a couple of library imports across the code but that's it.

B: The README file lists which libraries were used but does not mention any versions.

C: You find a `environment.yml` file with:

```
name: student-project
channels:
  - conda-forge
dependencies:
  - scipy
  - numpy
  - sympy
  - click
  - python
  - pip
  - pip:
    - git+https://github.com/someuser/someproject.git@master
    - git+https://github.com/anotheruser/anotherproject.git@master
```

D: You find a `environment.yml` file with:

```
name: student-project
channels:
  - conda-forge
dependencies:
  - scipy=1.3.1
  - numpy=1.16.4
  - sympy=1.4
  - click=7.0
  - python=3.8
  - pip
  - pip:
    - git+https://github.com/someuser/someproject.git@d7b2c7e
    - git+https://github.com/anotheruser/anotherproject.git@sometag
```

E: You find a `environment.yml` file with:

```
name: student-project
channels:
  - conda-forge
dependencies:
  - scipy=1.3.1
  - numpy=1.16.4
  - sympy=1.4
  - click=7.0
  - python=3.8
  - someproject=1.2.3
  - anotherproject=2.3.4
```

✔ Solution

A: It will be tedious to collect the dependencies one by one. And after the tedious process you will still not know which versions they have used.

B: If there is no standard file to look for and look at, it might become very difficult to create the software environment required to run the software. At least we know the list of libraries, but we don’t know the versions.

C: Having a standard file listing dependencies is definitely better than nothing. However, if the versions are not specified, you or someone else might run into problems with dependencies, deprecated features, changes in package APIs, etc.

D and **E**: In both of these cases exact versions of all dependencies are specified and one can recreate the software environment required for the project. One problem with the dependencies that come from GitHub is that they might have disappeared (what if their authors deleted these repositories?).

E is slightly preferable because version numbers are easier to understand than Git commit hashes or Git tags.

✍ Dependencies-2: Create a time-capsule for the future

Now we will demo creating our own time-capsule and share it with the future world. If we asked you now which dependencies your project is using, what would you answer? How would you find out? And how would you communicate this information?

Conda Python virtualenv R Matlab

We start from an existing conda environment. Try this either with your own project or inside the "coderefinery" conda environment. For demonstration purposes, you can also create an environment with:

```
$ conda env create -f myenv.yml
```

Where the file `myenv.yml` could have some python libraries with unspecified versions:

```
name: myenv
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.10
  - numpy
  - pandas
  - seaborn
```

After creating the environment we can activate it with

```
conda activate myenv
```

Now we can freeze the environment into a new YAML file with:

```
$ conda env export > environment.yml
```

Have a look at the generated file and discuss what you see.

✔ Some things to note

- Can you find all packages you installed directly? Which versions were installed?
- What other packages were installed? -> Dependencies of dependencies
- Besides the version you can also see the build channel
  - Sometimes the build includes an operating system or an architecture
  - Using this environment file might therefore not work/ not result in an identical setup on other computers

In the future — or on a different computer — we can re-create this environment with:

```
$ conda env create -f environment.yml
```

You may use `conda` or `mamba` interchangeably for this step; mamba may solve the dependencies a bit faster.

What happens instead when you run the following command?

```
$ conda env export --from-history > environment_fromhistory.yml
```

✔ Some things to note

- Everything is listed as you installed it; with or without specified versions
- Using this environment file a few days/weeks later will likely not result in the same environment
- This can be a good starting point for a reproducible environment as you may add your current version numbers to it (check for example with `conda list | grep "packagename"`)

In daily use you may not always use an environment.yml file to create the full environment, but create a base environment and then add new packages with `conda install packagename` as you go. Also those packages will be listed in the environment files created with either of the approaches above.

More information: https://docs.conda.io/en/latest/ and https://github.com/mamba-org/mamba

Keypoints

- Recording dependencies with versions can make it easier for the next person to execute your code.
- There are many tools to record dependencies and separate environments.

Recording environments

Objectives

- Understand what containers are and what they are useful for
- Discuss container definitions files in the context of reusability and reproducibility

Instructor note

- 10 min teaching/discussion
- 10 min demo

What is a container?

Imagine if you didn’t have to install things yourself, but instead you could get a computer with the exact software for a task pre-installed. Containers effectively do that, with various advantages and disadvantages. They are **like an entire operating system with software installed, all in one file**.

From reddit.

💬 Kitchen analogy

- Our codes/scripts <-> cooking recipes
- Container definition files <-> like a blueprint to build a kitchen with all utensils in which the recipe can be prepared.
- Container images <-> showroom kitchens
- Containers <-> a real connected kitchen

Just for fun: which operating systems do the following example kitchens represent?

1 2 3

[Midjourney, CC-BY-NC 4.0]

From definition files to container images to containers

- Containers can be built to bundle *all the necessary ingredients* (data, code, environment, operating system).
- A container image is like a piece of paper with all the operating system on it. When you run it, a transparent sheet is placed on top to form a container. The container runs and writes only on that transparent sheet (and what other mounts have been layered on top). When you are done, the transparent sheet is thrown away. This can be repeated as often as you want, and base is always the same.
- Definition files (e.g., Dockerfile or Singularity definition file) are text files that contain a series of instructions to build container images.

You may have use for containers in different ways

- **Installing a certain software is tricky**, or not supported for your operating system? - Check if an image is available and run the software from a container instead!
- You want to make sure your colleagues are using the **same environment** for running your code? - Provide them an image of your container!
  - If this does not work, because they are using a different architecture than you do? - Provide a definition file for them to **build the image suitable for their computers**. This does not create the exact environment you have, but in most cases a similar enough one.

The container recipe

Here is an example of a Singularity definition file (reference):

```
Bootstrap: docker
From: ubuntu:24.04

%post
    apt-get -y update
    apt-get -y install fortune cowsay lolcat

%environment
    export LC_ALL=C
    export PATH=/usr/games:$PATH

%runscript
    fortune | cowsay | lolcat
```

Popular container implementations:

- Docker
- Singularity (popular on high-performance computing systems)
- Apptainer (popular on high-performance computing systems, fork of Singularity)
- podman

They are to some extent interoperable:

- podman is very close to Docker
- Docker images can be converted to Singularity/Apptainer images
- Singularity Python can convert Dockerfiles to Singularity definition files

Pros and cons of containers

Containers are popular for a reason - they solve a number of important problems:

- Allow for seamlessly **moving workflows across different platforms**.
- Can solve the **"works on my machine"** situation.
- For software with many dependencies, in turn with its own dependencies, containers offer possibly the only way to preserve the computational experiment for **future reproducibility**.
- A mechanism to "send the computer to the data" when the **dataset is too large** to transfer.
- **Installing software into a file** instead of into your computer (removing a file is often easier than uninstalling software if you suddenly regret an installation).

However, containers may also have some drawbacks:

- Can be used to hide away software installation problems and thereby **discourage good software development practices**.
- Instead of "works on my machine" problem: **"works only in this container"** problem?

- They can be **difficult to modify**.
- Container **images can become large**.

❗ Danger

Use only **official and trusted images**! Not all images can be trusted! There have been examples of contaminated images, so investigate before using images blindly. Apply the same caution as when installing software packages from untrusted package repositories.

Where can one share or find images?

- Docker Hub
- Quay
- GitHub Container Registry
- GitLab Container Registry
- GitHub/GitLab release artifacts
- Zenodo

Exercises

✍️ Containers-1: Time travel

Scenario: A researcher has written and published their research code which requires a number of libraries and system dependencies. They ran their code on a Linux computer (Ubuntu). One very nice thing they did was to publish also a container image with all dependencies included, as well as the definition file (below) to create the container image.

Now we travel 3 years into the future and want to reuse their work and adapt it for our data. The container registry where they uploaded the container image however no longer exists. But luckily we still have the definition file (below)! From this we should be able to create a new container image.

- Can you anticipate problems using the definitions file 3 years after its creation? Which possible problems can you point out?
- Discuss possible take-aways for creating more reusable containers.

Python project using virtual environment R project using renv

```
 1  Bootstrap: docker
 2  From: ubuntu:latest
 3
 4  %post
 5      # Set environment variables
 6      export VIRTUAL_ENV=/app/venv
 7
 8      # Install system dependencies and Python 3
 9      apt-get update && \
10      apt-get install -y --no-install-recommends \
11          gcc \
12          libgomp1 \
13          python3 \
14          python3-venv \
15          python3-distutils \
16          python3-pip && \
17      apt-get clean && \
18      rm -rf /var/lib/apt/lists/*
19
20      # Set up the virtual environment
21      python3 -m venv $VIRTUAL_ENV
22      . $VIRTUAL_ENV/bin/activate
23
24      # Install Python libraries
25      pip install --no-cache-dir --upgrade pip && \
26      pip install --no-cache-dir -r /app/requirements.txt
27
28  %files
29      # Copy project files
30      ./requirements.txt /app/requirements.txt
31      ./app.py /app/app.py
32      # Copy data
33      /home/myself/data /app/data
34      # Workaround to fix dependency on fancylib
35      /home/myself/fancylib /usr/lib/fancylib
36
37  %environment
38      # Set the environment variables
39      export LANG=C.UTF-8 LC_ALL=C.UTF-8
40      export VIRTUAL_ENV=/app/venv
41
42  %runscript
43      # Activate the virtual environment
44      . $VIRTUAL_ENV/bin/activate
45      # Run the application
46      python /app/app.py
```

✓ Solution

- Line 2: “ubuntu:latest” will mean something different 3 years into the future.
- Lines 11-12: The compiler gcc and the library libgomp1 will have evolved.
- Line 30: The container uses requirements.txt to build the virtual environment but we don’t see here what libraries the code depends on.
- Line 33: Data is copied in from the hard disk of the person who created it. Hopefully we can find the data somewhere.

- Line 35: The library fancylib has been built outside the container and copied in but we don’t see here how it was done.
- The Python version will be different and hopefully the code still runs.
- Singularity/Apptainer will have also evolved by then. Hopefully this definition file still works.
- No contact address to ask more questions about this file.
- (Can you find more? Please contribute more points.)

✍ (optional) Containers-2: Installing the impossible.

When you are missing privileges for installing certain software tools, containers can come handy. Here we build a Singularity/Apptainer container for installing the `cowsay` and `lolcat` Linux programs.

1. Make sure you have apptainer installed:

```
$ apptainer --version
```

2. Make sure you set the apptainer cache and temporary folders.

```
$ mkdir ./cache/
$ mkdir ./temp/
$ export APPTAINER_CACHEDIR="./cache/"
$ export APPTAINER_TMPDIR="./temp/"
```

3. Build the container from the container recipe file introduced above.

```
apptainer build cowsay.sif cowsay.def
```

4. Let’s test the container by entering into it with a shell terminal:

```
$ apptainer shell cowsay.sif
```

5. We can verify the installation.

```
$ cowsay "Hello world!"|lolcat
```

✍ (optional) Containers-3: Explore two really useful Docker images

You can try the below if you have Docker installed. If you have Singularity/Apptainer and not Docker, the goal of the exercise can be to run the Docker containers through Singularity/Apptainer.

1. Run a specific version of *Rstudio*:

   ```
   $ docker run --rm -p 8787:8787 -e PASSWORD=yourpasswordhere rocker/rstudio
   ```

   Then open your browser to http://localhost:8787 with login rstudio and password "yourpasswordhere" used in the previous command.

   If you want to try an older version you can check the tags at https://hub.docker.com/r/rocker/rstudio/tags and run for example:

   ```
   $ docker run --rm -p 8787:8787 -e PASSWORD=yourpasswordhere rocker/rstudio:3.3
   ```

2. Run a specific version of *Anaconda3* from https://hub.docker.com/r/continuumio/anaconda3:

   ```
   $ docker run -i -t continuumio/anaconda3 /bin/bash
   ```

Resources for further learning

- Carpentries incubator lesson on Docker
- Carpentries incubator lesson on Singularity/Apptainer

Keypoints

- Containers can be helpful if complex setups are needed to run a specific software.
- They can also be helpful for prototyping without "messing up" your own computing environment, or for running software that requires a different operating system than your own.

Where to go from here

Objectives

- Understand when tools discussed in this episode can be useful

Instructor note

- 10 min teaching/discussion

This episode presents a lot of different tools and opportunities for your research software project. However, you will not always need all of them. As with so many things, it again depends on your project.

Workflow tools will maybe make sense in the future

- In many cases, it is probably not needed
- You will want to consider workflow tools:
  - When processing many files with many steps
  - Steps or files may change
  - Your main script, connecting your steps, gets very long
  - You are still collecting your input data
  - …

Containers seem amazing, but do I have use for them?

- Maybe not yet, but knowing that you can …

  - Run Linux tools on your Windows computer
  - Run different versions of same software on your computer
  - Follow the "easy installation instructions" for an operating system that is not your own
  - Get a fully configured environment instead of only installing a tool
  - Share your setup and configurations with others

  … can be very beneficial :)

Important for every project

- A Clear directory/file structure for your project.
- Record your workflow and write it down in a script file.
- Create a dependency list and keep it updated, optimally in an environment file.
- At least consider the possibility that someone, maybe you, may want to reproduce your work:
  - Can you do something (small) to make it easier?
  - If you have ideas, but no time: add an issue to your repository; maybe someone else wants to help.

Further reading

- The Turing Way handbook to reproducible, ethical and collaborative data science
- Reproducible research policies and software/data management in scientific computing journals: a survey, discussion, and perspectives

- …

➡ See also

Do you want to practice your reproducibility skills and get inspired by working with other people’s code/data? Join a ReproHack event!

❗ Keypoints

- Not everything in this lesson might be useful right now, but it is good to know that these things exist if you ever get in a situation that would require such solutions.
- Caring about reproducibility makes work easier for the next person working on the project - and that might be you in a few years!

List of exercises

Full list

This is a list of all exercises and solutions in this lesson, mainly as a reference for helpers and instructors. This list is automatically generated from all of the other pages in the lesson. Any single teaching event will probably cover only a subset of these, depending on their interests.

Instructor guide

Detailed day schedule

Some example schedules for this lesson:

2024 edition plan (times in EET, Helsinki time), **no exercises**, just demos:

- 09:50 - 10:00 Soft start and icebreaker question
  - Page: collaborative notes document
  - Give more space to the icebreaker and see what people are writing and talk about our own experiences
- 10:00 - 10:03 Collab document intro
- 10:03 - 10:05 Learning outcomes: https://coderefinery.github.io/reproducible-research/
- 10:05 - 10:10 Overview of CR and how it all fits together
  - Page: https://coderefinery.github.io/reproducible-research/intro
  - Learning outcomes from index
- 10:10 - 10:20 Reproducible research, Motivation
  - Exercise in notes doc with the discussions in bottom of motivation page
  - Page: https://coderefinery.github.io/reproducible-research/motivation/
- 10:20 - 10:30 Organizing your projects
  - Copy the discussion on the notes and if we have time we can highlight some answers
  - Page: https://coderefinery.github.io/reproducible-research/organizing-projects/

- 10:30 - 10:35 ask in collab document and discuss
    - https://coderefinery.github.io/reproducible-research/organizing-projects/#discussion-on-reproducibility
        - Are you using version control for academic papers?
            - ...
            - ...
        - How do you handle collaborative issues e.g. conflicting changes?
            - ...
            - ...
- 10:35 - 10:55 Recording computational steps
    - Page: https://coderefinery.github.io/reproducible-research/workflow-management/
- 10:55 - 11:05 Real break
- 11:05 - 11:25 Recording dependencies
    - https://coderefinery.github.io/reproducible-research/dependencies/#exercises
        - ask first one in collab doc and discuss on stream
        - show difference between created env from env file vs exported env file on stream
- 11:25 - 11:30 ask in collaborative document
    - Are you using any dependency and/or environment management tool in your work?
        - No: o
            - why not?
                - ..
                - ..
        - Yes: o
            - which?
                - ..
                - ..
    - Have you heard about or been in contact with containers (docker, singularity, podman) in your work? How did you come across them?
        - No: o
        - Yes:
            - ..
            - ..
            - ..
- 11:30 - 11:50 Recording environments
    - The first contact with containers is often: Take this and run this command and then when you need to share/build.
    - Discuss setup issues, permissions if docker wants root, bandwidth, etc
    - Pros and cons of containers
    - Demo of two pre-made containers e.g. expand the R studio optional exercise?
- 11:50 - 12.00 Wrapup
    - where to go from here: idea would be to give it more practical focus: what to do with these tools? Project level reproducibility. Time-scales of what changes (short time changes of code, long time years changes of OS-s, libraries).
    - Bring your code session advertisement
    - Material + recording available

- 12:00 - long break starts

This is the planned schedule for the workshop in September 2023 (2 hours and 5 minutes including 10 min break) ; note that for this workshop, sharing code and data was moved to social coding lesson:

- 08:50 - 09:00 Soft start and icebreaker question
- 09:00 - 09:10 Overview of CR and how it all fits together
- 09:10 - 09:20 Reproducible research, Motivation
- 09:20 - 09:27 Organizing your projects
- 09:27 - 09:35 Recording computational steps - discussion
- 09:35 - 10:00 Snakemake exercise (25 min)
- 10:00 - 10:10 Break
- 10:10 - 10:15 Summary of workflows and the exercise
- 10:15 - 10:30 Recording dependencies
- 10:30 - 10:40 Recording environments
- 10:40 - 11:00 Container-1 exercise (20 min)
- 11:00 - 11.05 Wrapup

This was the schedule at workshop in March 2023 (2 hours and 15 minutes including 2x 10 min break):

- 08:50 - 09:00 Soft start and icebreaker question
- 09:00 - 09:10 Interview with an invited guest
- 09:10 - 09:20 Motivation
- 09:20 - 09:30 Organizing your projects
- 09:30 - 10:00 Recording dependencies
  - discussion (5 min)
  - exercise (20 min)
  - discussion (5 min)
- 10:00 - 10:10 Break
- 10:10 - 10:40 Recording computational steps
  - discussion (5 min)
  - exercise (20 min)
  - discussion (5 min)
- 10:40 - 10:50 Recording environments
  - an exercise exists but is typically not done as part of a standard workshop
- 10:50 - 11:05 Sharing code and data
  - demo (15 min)
- 11:05 - 11:15 Break

Why we teach this lesson

Reproducibility in research is something that publishers, funding agencies, universities, research leaders and the general public worries about and much is being written about it. It is also something that researchers care deeply about - this lesson is typically one of the most popular lessons in the pre-workshop survey.

Even though most PhD students, postdocs and researchers (i.e. typical workshop participants) know about the importance of reproducibility in research, they often lack both a general overview of what different aspects there are to reproducibility, and the knowledge of specific tools that can be used for improving reproducibility.

Many participants may not adhere to good practices when organizing their projects, and the “Organizing your projects” episode is meant to encourage participants to structure their projects better. This may be obvious to some participants but it doesn’t harm to preach to the choir.

Even though many participants know that code can have many dependencies (e.g. they may have experienced difficulties in getting other people’s code to run), they often don’t know or use good practices when it comes to recording dependencies. Most participants also don’t use isolated environments for different projects and don’t know why that can be important. The episode “Recording dependencies” tries to convey the importance of recording dependencies accurately for your projects, and shows how tools like conda can be used both as a package and software environment manager.

Many participants have heard about containers and find them interesting, but lack an understanding of how they work or how they can be used. The episode “Recording environments” introduces the concept of containers, and the optional episode “Creating and sharing a container image” goes into details.

Many participants use complicated series of computational steps in their research without realizing that this work falls into the category of “scientific workflows”, and that there actually exist tools that help make such workflows reproducible. The episode “Recording computational steps” introduces the concept of scientific workflows, discusses various ways of managing workflows with varying degrees of reproducibility, and shows how tools like Snakemake can be used to both simplify workflows and make them more reproducible.

How to teach this lesson

How to start

Everyone knows that scientific results need to be reproducible, but not everyone is using appropriate tools to ensure this. Here we’re going to get to know tools which help with preserving the provenance of data and reproducibility on different levels, ranging from workflow automation to software environment (containers).

Focus on concepts, and when to use which tool

Try to explain better what the different tools are useful for, but don’t go into details. In this lesson we are not trying to gain expertise in the various tools and master the details but rather we want to give an overview and show that many tools exist and try to give participant the right feel for which set of tools to approach for which type of problem.

Typical pitfalls

Indentation in Snakefiles

- the body of a rule and the body of an input keyword need to be indented, but the number of spaces doesn't matter This works:

```
rule all:
    input:
        expand('statistics/{book}.data', book=DATA),
        expand('plot/{book}.png', book=DATA)
```

but this doesn't work:

```
rule all:
    input:
    expand('statistics/{book}.data', book=DATA),
    expand('plot/{book}.png', book=DATA)
```

nor this:

```
rule all:
input:
    expand('statistics/{book}.data', book=DATA),
    expand('plot/{book}.png', book=DATA)
```

Field reports

2022 September

We used the strategy "absolutely minimal introductions, most time for exercise". Overall, it was probably the right thing to do since there is so little time and so much to cover.

There wasn't enough time for the conda exercise (we could give only 7 minutes), but also I wonder how engaging it is. We should look at how to optimize the start of that episode.

The Snakemake episode went reasonably well. Our goal was 5 minutes intro, long exercise, 5 minutes outro. The intro was actually a bit longer, and there was the comment that we didn't really explain what Snakemake was before it started (though we tried). The start of this episode should get particular focus in the future, since this is the main exercise of the day.