Organising your scientific code

Writing reproducible software

ASPIRE

Evert Rol – 2023-07-04

https://github.com/evertrol/aspire-organise-code

ANTON PANNEKOEK INSTITUTE

Why organise your code

Reasons why code can get disorganised

Applies to code (code bases) in general, but scientific code can get disorganised, because

- more interest on results (and performance: results fast)
    - focus is on paper / thesis; the software is not the primary goal
- numerous quick scripts may be cobbled together to produce some form of a pipeline
- coding / software experience is very variable amongst the people writing (and using) it
- maintenance: 2-4 years for people in temporary positions
    - code can be fine at the start, but other packages, libraries, compilers etc go forward and get updated, making them incompatible with the older code
    - sometimes older packages or libraries may not be available anymore; thus the code needs to be maintained so it can be updated to use newer packages or libraries

Why organise your code - FAIR

FAIR principles: **F**indable, **A**ccessible, **I**nteroperable, **R**eusable

Also (originally) for scientific data management, but also applicable to research software

Helps making the software (and therefore the science) *reproducible*

Why organise your code - FAIR

Findable

- Unique and persistent identifier
- Rich metadata (purpose, authors/contact, citation information)
  - Metadata next to the software, but also in the software publication (if any)
- Stored centrally (e.g. Zenodo); there is no central astronomy software repository or search engine (while there is for articles), but often through publication
  - Good metadata (keywords) can show your software at places like https://github.com/topics/astrophysical-simulation
- Get a DOI: **D**igital **O**bject **I**dentifier

Why organise your code - FAIR

Findable - DOI

- Get a DOI: **D**igital **O**bject **I**dentifier
  `10.5281/zenodo.807293`
  - Zenodo, FigShare and other repositories (but not GitHub and the like) can provide a DOI
    - One DOI for the latest version; continuous updates
    - One DOI for every stored version, so that older results can be matched to an older version of the software

! You can get a DOI before you store / archive your software
This way, you can include the DOI with the stored software

Aside: GitHub, GitLab & similar are for developing software.
Zenodo, FigShare & similar are for archiving (scientific) software (and data/results); these tend to be endpoints

Why organise your code - FAIR

Accessible

- Standard ways of obtaining the software.
- Standard tools for installing a package or library
  - Might be only available upon request, but should still have a standard way to install (just with a manual "download" part)
- No proprietary protocols

Why organise your code - FAIR

Interoperable

- Input and output uses common standards
  - FITS, HDF5, JSON, CSV file formats
- Common software installation techniques

Why organise your code - FAIR

Reusable

- proper licence
  (no licence would technically mean it can't be used.)
  - Stick to standard licences
- standard organisation of software
- proper details about the dependencies

FAIR software

Reproducible and maintainable software

- Findable: D.O.I., metadata (keywords)
    - Stick to standard licences
- Accessible: standard organisation and installation of the software
- Interoperable: standard file formats for data interchange
- Reusable: Licence, standard project structure, list dependencies

Setting up your machine - OS

It is not essential to have this software installed, but you will miss out on any interactivity that way.

We'll be using a Unix-like system (Linux, macOS), with a terminal. On Linux and macOS, you should already be set up, although I prefer iTerm2 instead of the standard Terminal on macOS

For Windows, a solution provided by Microsoft is WSL (Windows Subsystem for Linux), together with the Windows Terminal:
https://learn.microsoft.com/en-us/windows/wsl/install
- on the command prompt or PowerShell, type `wsl --install` and follow the prompts. Pick Ubuntu as operating system (OS); 10 GB (or even 5) should be enough (you can always increase this later).
- the Windows Terminal can be installed through the windows store. Once installed, you can select Ubuntu, and set a default application. See also https://learn.microsoft.com/en-us/windows/terminal/install

On Linux, you can use your default package manager to install some software. The naming depends on the flavour and version; look for a build-essential or build-utils package, which will install a number of packages, including C & related compilers:

```
sudo apt install build-essential
```

If you are running Ubuntu via WSL, you can use these Linux commands as well.

On macOS, I recommend installing Homebrew as package manager for Unix-style utilities. You'll first need to install build utilities through Apple's XCode, with `xcode-select --install` . Then install Homebrew from https://brew.sh/ .

Setting up your machine - editor

A text/code editor is for editing text or code; not for layout (not a word processor).
A good code editor will help with code suggestions.

One popular editor is Visual Studio Code. It works on Windows, macOS and Linux.
It is actually an IDE, Integrated Development Environment, and can do more than just helping with code editing. For now, use it mainly as a code editor.

Other editors common on Unix-like systems are `vi` / vim and `emacs`. Both are not very beginner friendly, but are very extensible, have a long history (late 1970s) and are so common on Unix-like system that you are likely to bump into them. Friendlier, simpler alternatives that are often found are the `pico` and `nano` editor.

For VS Code with WSL, you can install the WSL extension, and point that to the WSL OS you are using: https://learn.microsoft.com/en-us/windows/wsl/tutorials/wsl-vscode . (Or install the remote development extension pack, which includes the WSL extension and provides other practical utilities as well.)

Emergency editor exits

- VI: `<escape>:q!`
- Emacs: `<control>-g <control>-g` (to stop whatever you might be doing), then `<control>-x <control>-c`, then answer the prompts at the bottom.

Setting up your machine - Python

Python is ubiquitous in Astronomy & Astrophysics, even if it's not the fastest programming language to use. But it is very easy to start writing software in it.

Python may already be installed: try `python --version`
Version 3.11 is current, but 3.10, 3.9 or even 3.8 are also fine.

If not, use your system's package manager. (On older systems, you may need to install the python3 package, not python. Check the version after installation.)

I don't recommend installing the package from the official Python website, but through your system's package manager (or Homebrew). If you use the WSL, use that system's package manager

For VS Code, there also exists a Python extension. Once installed, set the Python interpreter in VS Code through the "command palette" (`<control>-<shift>-P` or `<F1>`), then search for "interpreter" or "python".

When you (accidentally or not) end up on the Python prompt, `>>>`, you can exit this with `<control>-d` (`<control>-z` under Windows, but not the WSL) or `exit()`

Setting up your machine - Python

Python is ubiquitous in Astronomy & Astrophysics, even if it's not the fastest programming language to use. But it is very easy to start writing software in it.

Try install a few Python packages:

```
python -m pip install matplotlib jupyterlab
```

Note: no sudo: Python packages can be installed as a normal user; system packages usually require sudo (except for Homebrew).
If Pip can't be found, install the `python-pip` package first.

Try starting the Jupyter Lab environment from the command line: `python -m jupyter lab`
A browser tab should open, and you can test Jupyter by typing

```
import numpy as np
(1 + np.sqrt(5)) / 2
```

in the empty cell, then press `<shift>-<enter>` to execute that cell.

Close the tab (no need to save this) and use `<control>-c` (and confirm with y) to exit the Jupyter server on the command line.

I prefer using `python -m <a-python-package-or-utility>`: that ensures I am using the right Python with the correct utility. Instead of simply `pip` or `jupyter lab`, which may be part of a different Python installation.

Setting up your machine - Git

Git may already be installed: try `git --version`

If not, use your system's package manager. The package should be simply called `git`.

Setting up your machine

Why this focus on the command line, terminal and shell? (And what are they?)

You can't always use a GUI (graphical user interface); in particular remote HPC (high-performance computing) systems are accessible only by command line.

The command line is also more reproducible: entering commands is always the same (scriptable), while clicking a button is harder to record, and may not always be the same / in the same position.

Many astronomical softwares are primarily written on and for Linux.

- Command line: where you are typing commands
- Shell: what interprets your commands (before passing them to the operating system). For example, wildcard expansion: `*.txt`
- Terminal: application that provides an interface to the shell and command line. Refers to old-style terminals.

Project structure

Don't reinvent the wheel.

Look at what is commonly done, what is the standard for the language you are using. Many languages have a standard way of organising the software.

Easy to navigate and recognisable for other people.

- C & C++ will have src/ and include/ subdirectories.
- Python will have a <package-name>/ subdirectory
- Many projects have separate docs/ and tests/ subdirectories.

Some newer languages have standard tools that already setup a good structure for you. E.g., in Python you can use Poetry, Rust has Cargo. The older languages (C/C++/Fortran) have some tools, but most of it has often been done manually, and the structure evolved over several decades of use.

Tools

Use standard tools and configuration files related to the language

For example, so-called build tools and files:

- C / Fortran - `make` + Makefiles
- C++ - `cmake` + CMakefiles (other tools exist)
- Python - `pip` / pyproject.toml / (setup.py) / `poetry`
- Rust - `cargo`

If you have multiple languages, pick the most appropriate language (e.g., C extension for Python can be build using setup.py)

Or organise the languages separately in their own subdirectories, with their own build and installation tools. And possibly an overall build script in the base project directory.

Metadata

Provide the following files (other than the code). Usually in the main directory

- a README file
- a licence file
- authors & contact info (often in project metadata file)
- citation file, if applicable (`CITATION.cff`)
- a metadata / organisation / build file (for example, `pyproject.toml` for Python projects)

Metadata - README

- (single line with version info, DOI, documentation link, testing status)
- title
- short paragraph what the software does
- brief example of how the software works. If too long, put in separate documentation.
- brief installation guide. Pointer to a separate installation guide (with troubleshooting) if necessary, such as an INSTALL file
    - this does, in general, not require listing all dependencies: most of the time, a separate, project-related file exists for that. C/C++/Fortran are somewhat the exception, so in that case, list the dependencies in the README or INSTALL file
- licence name, pointer to licence file
- possible reference required / optional; pointer to a citation file. Provide a D.O.I.!
- how to contribute (if applicable)
- references to software / methods used

Metadata - Licence

Without a license, the software actually can't be used!

- Open source: pick a default open source
  - MIT / 2-clause BSD license: very permissive (only provide credits to original authors)
  - GPL: credits, and contribute back when changing the code
  - Apache 2: safeguards from (accidental) patents by contributors
  - public domain: it's anyone's and everyone's software! (SQLite does this)
- Closed source license: check with your institute
  - This is only important once you make the software available upon request (you should still get a D.O.I. for it)
- Be aware whether you need to include licenses for (parts of) other software that your software contains.
  This is, however, often not the case: packages / libraries are generally fine to use without license. Ditto for compilers, interpreters, OS etc. Generally, only when you specifically include (parts of) software directly in your code.

Metadata - CITATION.CFF file

Humand and machine readable file for citations.

Only needed if there is an actual citation to the software. Otherwise, the software (repository) itself is the citation

Example:

```
cff-version: 1.2.0
message: "If you use this software, please cite it as below."
authors:
  - family-names: Druskat
    given-names: Stephan
    orcid: https://orcid.org/1234-5678-9101-1121
title: "My Research Software"
version: 2.0.4
doi: 10.5281/zenodo.1234
date-released: 2021-08-11
```

See also https://citation-file-format.github.io/

Metadata - notes

README, INSTALL and LICENSE are often capitalised to have them stand out and put them first when running `ls`. On Windows, this may not work, since it is case insensitive (but should work in the WSL).

Often, appending `.md` or `.rst` extension, such as `README.md`, make the file instantly readable on sites like github.com or gitlab.com.
In that case, you can also perform some layout with Markdown (.md) or reStructured Text (reST, .rst). Try to refrain from doing too much markup: the files should be readable as plain text, without any markup (both Markdown and reST already do this anyway, in particular Markdown). Plain text is still far more portable than e.g. PDFs or Word ssdocuments.

Documentation

The README is the first place for documentation. This can contain a pointer to a separate documentation file or directory

If there is separate documentation

- Find a language appropriate documentation tool (e.g., Sphinx is popular for Python, but also a few other languages).
- When writing with Markdown or reST, as with the README file, GitHub or GitLab can automatically format it for you (including links)
- If lengthy documentation, you may consider using a website like https://readthedocs.org/ .

Documentation should / can include

- Installation guide (may refer to a separate INSTALL document)
- Introduction / quickstart
- Details on the configuration or functionality
    - Very project type dependent:
        - Is it a library for use elsewhere? Then document the functions properly
        - Is it an executable to be used with input parameters? Document the parameters properly
- Examples. A few good examples often tell more than lots of details.

Configuration files

Configuration files allow flexibility for using your code, and can organise this in subsections

There are various formats: don't invent your own! Use a well-known format.

My favourite choices

- TOML (Tom's Obvious, Minimal Language): subsections and flexible divisions, fairly strict formatting (to prevent user errors), various data types (integers, floating point numbers, strings, date-time), boolean, arrays, key-value pairs, comments
- Use the language itself. It can be overly flexible, so it's up to the users of your software to restrict themselves. You have the data types directly in the right language, but you'll have to perform checks yourself (instead of a library or package)

I don't recommend YAML, which is otherwise fairly popular, because it's too flexible, and can have ambiguous results (read up on the *Norway problem*).
JSON is more strict, but unreadable for humans.

Plain text files work, but they can grow into a self-invented wheel with all kind of extra complications.

Coding style

Stick to a well-known and generally accepted coding style

Use formatters.
You may not fully like the result, but it will make the code more uniform, both within your project and compared to the rest of the world.

Example formatters:

- C & C++: clang-formatter
- Python: black
- Julia: JuliaFormatter
- Rust: rustfmt

Coding style

Warnings are often ignored, because they're "just that": warnings; they're not errors.

Warnings can only be ignored if they are completely understood: why they can be ignored. If necessary, write a comment about it.
Otherwise, *fix the warning* (that does not mean "sweeping it under the carpet": that would make the potential problem worse).

Compilers have warning flags, such as `-Wall` and `-Wextra`. Use them (`-Weverything` is fun to try once, but it's really too much).

Other languages often have so-called **linters**, which are code checkers.

These work not just for mistakes, but also for potential mistakes and stylistic conventions.

Example: i, j and k are often used as loop variables for counters. A 4th loop counter could be called l. But l and i look similar (depending on the font), so l is a bad choice for a variable name.
In fact, the actual bad choice is four nested loops: better turn part of that into a function (the function calling overhead will be minimal compared to the nested outer loop). A linter will tell you about that.

Python has, among others, flake8 and pylint utilities. You can install them `python -m pip install flake8 pylint`, and use them on Python code.
Pylint will be a bit over the top with its warnings and tips; just pick up the good suggestions. With some reasonable effort, you can get Flake8 to not report anything, which generally is good.

Also be weary about deprecation warnings: these warn about functionality in a library or package that you are using, but that may disappear in a next version. Unless you require backward compatibility to an old version, usually it's worth addressing these warnings early, often by using a newer function of that library or package.

Why the trouble?

If you are just writing a small piece of software for yourself, or you and one other person, you can basically do whatever you like.

But if you are writing something that might be used by someone else later, in particular when you have left the project, a good project structure, with clear documentation and so on, becomes vital.
Think of a research group, where a master or PhD student or a PostDoc writes software, but then leaves after a number of years, while newer students have to work with the software and extend it.

Most of the steps are small steps, fairly easy to do on a regular basis, and there are lots of tools that can help you with (formatting tools, tools to set up an initial project structure).

Version control software - Git

Version control software keeps track of (small) changes to your software.
Technically, you'll have lots of small versions of your software.

Versioning software also makes it possible to create "branches" of your software, with specific, more experimental, features to your code.

Git is the most popular these days, but others exists, such as Subversion (svn) and Concurrent Versions System (CVS).

Git is used in many places, but can be hard to learn at first: it has a steep learning curve.

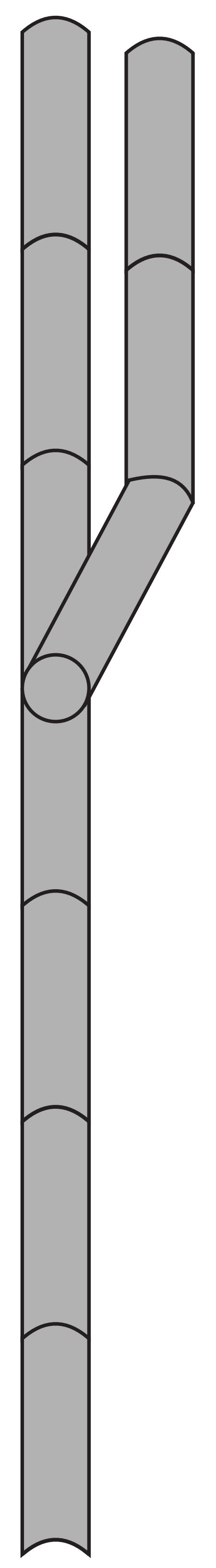
Git

Git works with so-called commits. Each commit records a change in your code (or any file you track with Git).

Think of commits as a series of pipe segments: each segment contains a *change* in the tracked files when compared to the previous segment. Starting from the beginning, you can then always reconstruct the current files, or anything in between.

The most recent commit is identified as HEAD (think the head of a snake of pipe segments); the first commit does not have a specific name, and is often simply called first commit or initial commit.

Git allows you to compare changes between commits, between HEAD and the current state of your files, list the history of your project, create branches of your software for, for example, adding new software, and merging those branches back into the main branch.



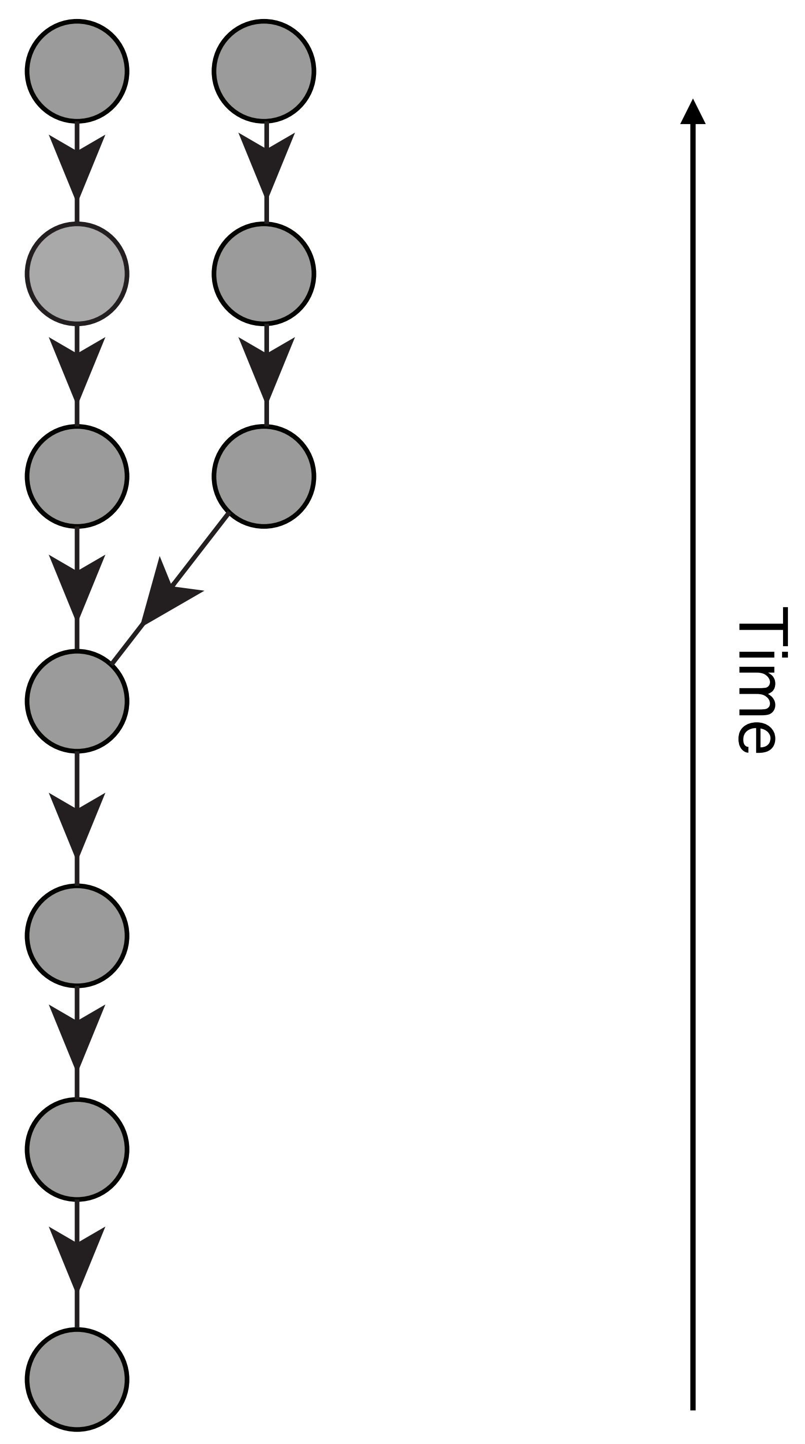
Git

Git works with so-called commits. Each commit records a change in your code (or any file you track with Git).

Think of commits as a series of pipe segments: each segment contains a *change* in the tracked files when compared to the previous segment. Starting from the beginning, you can then always reconstruct the current files, or anything in between.

The most recent commit is identified as HEAD (think the head of a snake of pipe segments); the first commit does not have a specific name, and is often simply called first commit or initial commit.

Git allows you to compare changes between commits, between HEAD and the current state of your files, list the history of your project, create branches of your software for, for example, adding new software, and merging those branches back into the main branch.



Git - exponential decay

$$\frac{dy}{dt} = -\alpha \cdot y(t)$$

$$y(t) = y_0 \cdot e^{-\alpha \cdot t}$$

Forward difference

$$\frac{dy}{dt} = \frac{y(t+dt) - y(t)}{dt}$$

$$y(t+dt) = y(t) - \alpha \cdot y(t) \cdot dt$$

```
y_next = y - alpha * y * dt
```

Central difference

$$\frac{dy}{dt} = \frac{y(t+dt) - y(t-dt)}{2 \cdot dt}$$

$$y(t+dt) = y(t-dt) - \alpha \cdot y(t) \cdot 2 \cdot dt$$

```
y_next = y_prev - 2 * alpha * y * dt
```

Git - practical

```
mkdir myproject
cd myproject
git init
git status
```

create a (code) file

```
git add <file.extension>
git status
git commit
```

You may get stuck in Vi: `<escape>:!q` to exit

Configure an editor for Git messages:

VS Code: `git config --global core.editor "code --wait"`

or

Emacs: `git config --global core.editor "emacs -nw"`

```
cat ~/.gitconfig
```

https://tinyurl.com/expdecay1a

Try again: git commit

Enter a relevant short message (maximum of 50 to 60 characters) on the first line.

Leave a blank line, then enter a more informative message (can be several paragraphs). (Ideally, format the paragraphs to be a maximum of 80 characters or so wide. That'll make it easier to read later.) Save, and exit the editor.

```
git status
git log
```

Edit the file with the code

```
git status
git diff
git add <file.extension>
git status
git commit
git log
```

https://tinyurl.com/expdecay1b

Git - practical

Note: Git will show difference for whitespace (space, newline, tab) only.

This means that trailing whitespace (spaces at the end of a line, empty lines at the end of a file) will show a difference.

But such trailing whitespace does not add anything to the code: no functionality, no readability.

Always clean your code from trailing whitespace!

https://tinyurl.com/expdecay1a

Try again: git commit

Enter a relevant short message (maximum of 50 to 60 characters) on the first line.

Leave a blank line, then enter a more informative message (can be several paragraphs). (Ideally, format the paragraphs to be a maximum of 80 characters or so wide. That'll make it easier to read later.) Save, and exit the editor.

```
git status
git log
```

Edit the file with the code

```
git status
git diff
git add <file.extension>
git status
git commit
git log
```

https://tinyurl.com/expdecay1b

Git - practical

git log can get rather verbose. You can use the `--short` option, or even more options, like

```
git log --pretty=format:%h%x09%ad%x09%s --date=short --graph --decorate --color
```

Create a Git alias:

```
git config --global alias.slog "log --pretty=format:%h%x09%ad%x09%s --date=short --graph --decorate --color"
git slog
```

The hexadecimal string you can see in the log output, and its shortened form in the slog output, is the commit hash: a unique identifier for that particular commit. This way, you can always refer to and find a specific commit anywhere in your code.

You can often use a shortened form, for example the first four, five or six characters. As long as it is unique for the project, Git will find the correct commit.

HEAD is a special indicator: it is where you are on the current branch.

Git - commit message

A good commit message has the following:

- short line, 50-60 characters. Make it an active sentence, starting with a verb. (Mentally prepend "this commit will" in front of the sentence.)
  This is a practice that is done in many places, so better keep that. Messages such as "README", "update" or "oops" don't really mean anything: did you add a README or change it, what was updated, what was the mistake? Make it specific.
  Examples:
  - `Fix bug by preventing a negative array index`
  - `Update the README with a macOS specific installation section`
- Details on why and how in the following paragraphs. You could include references to websites etc here as well (I do that when I have a bug fix and I found a solution on the internet).

The very first commit message is a bit of an odd one, since there is no clear change. Often, "Initial commit" is used, which is not in line with the above suggestions, but works and is often used.

Something like `git log --short` can provide a very nice overview this way.

Git - commit message

Randall Munroe - https://xkcd.com/

A good commit message has the following:

- short line, 50-60 characters. Make it an active sentence of the sentence.)
  This is a practice that is done in many places, so better really mean anything: did you add a README or change Examples:
  - `Fix bug by preventing a negative array inde`
  - `Update the README with a macOS specific ins`
- Details on why and how in the following paragraphs. Yo when I have a bug fix and I found a solution on the inte

The very first commit message is a bit of an odd one, sin not in line with the above suggestions, but works and is often used.

Something like `git log --short` can provide a very nice overview this way.

Versions

Semantic versioning

Major.minor.bugfix

`1.0(.0)`, `1.0.1`, `1.1(.0)`, `1.2(.0)`, `1.2.1`, `1.2.2`

`2.0(.0)`, `2.0.1`, `2.0.2`

Versions with the same major number are compatible between each other!

Before version 1:

`0.1`, `0.2`, `0.3`, ... (also 0.10 and beyond)

Compatibility is more loose.

Sometimes `alpha`, `beta` or `rc` (release candidate) is appended to a major release

Alternative: by date

20230704, 20230724, 20230820, 20231001 etc

Doesn't show compatibility between versions!

Don't use this.

Release date of a version will also be in the metadata with the version number

```
git tag v0.1
git tag
```

If you want to tag a previous commit

```
git tag v0.1 <commit-hash>
```

Dependencies & versions

Dependencies are packages, libraries, compilers, language versions, that your project uses.

Over time, these dependencies may get updated, and not work with your project anymore.

Technically, for any version of your software you release, a list of dependency versions should be given. This can be hard, or at least annoying. Tools exist to help.

You can start with very strict dependencies for your current software version: every package, library etc has only one specific version. Then, try and widen the range for each package, by checking which versions are currently supported of that package. For example, in July 2023, Python (the language), versions 3.8, 3.9, 3.10 and 3.11 are supported (leave a few months of margin on each side). Start with coding for 3.11, then try and test for older versions.

Using newer versions of dependencies during development can work better, since these will become more established over time, while older versions get abandoned as your project matures.

Once you have a released a specific version of your software (1.0 or later), the dependencies should remain fixed for that version, even if they go out of date; the latter then also means your software needs to be updated to a new version (perhaps only to fix the dependencies)

Dependencies & versions

Example dependencies

```
requires-python = ">=3.8"
dependencies = [
    "numpy>=1.20,!=1.24.0",
    "pandas>=1.2",
    "matplotlib>=3.3,!=3.6.1",
]
```

Fix "upper limits" of your dependencies!

Especially for the major version part.

```
requires-python = ">=3.8,<=3.11"
dependencies = [
    "numpy>=1.20,!=1.24.0,<=1.25",
    "pandas>=1.2,<2",
    "matplotlib>=3.3,!=3.6.1,<4",
]
```

Format depends on installation/building tool used

Can't always specify it like this. Write down as plain text where necessary.

Git - branches

```
git status
git slog
git branch
git branch feature/scipy
git branch
git checkout feature/scipy
git slog
```

Edit the code

```
git status
git diff
git add <file(s)>
git commit
git slog
git checkout main
git slog
```

The checkout command is very useful, and often used, in Git.

You can use it to switch between branches, but also switch to a specific commit in the past.

Be careful: any change you make after such a checkout (to a past commit) will make a mess of your Git history.

You'll get a warning when you switch to a commit in the past, about being in a "detached" HEAD state. If, for some reason, you want to add a commit there, it is better to do that with a new branch from that commit.

Better yet, if you find out too late you have forgotten something, add it as a separate commit at the top of the chain (that is, at the HEAD). That way, the normal order is preserved, and you don't need to alter the past.

https://tinyurl.com/expdecay1c

Git - merging

You can continue on the main branch, for bug fixes and other changes, while we develop our new feature on the other branch.

You can only switch between branches if the status of the current branch is clean!

Once finished, You can merge the new feature branch into the main branch:

```
git checkout main
git merge feature/scipy
```

Similarly, small changes made into the main branch, such as a bugfix, can be merged in the feature branch while that is still being developed:

<add bugfix in main branch>

```
git commit
git log
```

Note the hash of the last commit

```
git checkout feature/scipy
git cherry-pick <commit-hash>
git checkout main
```

Merge conflicts!

Merges are a common thing in Git, but they can become tricky: you can end up with so-called merge conflicts, where you'll need to help Git manually decide (by editing the file(s)) what to merge and what not (or not yet).

For now, fingers crossed and assume the merges work.

If there is time, we may have a look at a merge conflict and how to resolve it.

https://tinyurl.com/expdecay1c

Git - don't panic

Git can become problematic, especially when merging things.
Sometimes, it may even look like you have lost code!
But most of the time, it will still be there, somewhat hidden.

(Virtual) environments

Virtual environments "shield" you from the standard installation provided by the operating system.

They are useful for testing, and for setting up an independent project

Example: you need to install a set of very specific packages for a data reduction pipeline. A virtual environment can help preventing conflicts between these packages and globally installed ones.

Example: you create a software package with specific dependencies. Develop and test inside a virtual environment, so those dependencies don't conflict with global dependencies.

(Virtual) environments

Python venv

Python's built-in virtual environment provide an environment only useful for Python packages. Which already helps for a lot of software.

Note that it does not allow you to select different versions of Python; only packages.

Use as follows:

```
pip list

python -m venv ~/venvs/testenv

source ~/venvs/testenv/bin/activate

pip list

pip install matplotlib==3.4

pip list

deactivate

pip list
```

(Virtual) environments

Conda / Anaconda

Conda goes a step further. It handles more dependencies (libraries) than just Python packages, and can also install multiple Python versions.

It also tries to prevent problems / conflicts between various libraries you may want to install.

If you want to use Conda, use the miniconda installer from https://docs.conda.io/en/latest/miniconda.html

If you use Conda, always work in a specific environment. Try to avoid using the Conda <base> environment (see your prompt, or run `conda env list` and look at the environment with an asterisk, *)

Conda is very popular, since it provides more options, such as libraries, and multiple Python versions, but care needs to be taken by using the correct environment (and activating or deactivating it properly).
For a straightforward set of Python packages, however, Python's built-in virtual env is absolutely fine.

The difference between Conda and Anaconda

Anaconda is a full, consistent set of Python packages (mostly for sciences), gathered together and provided through the Anaconda company (profit through services)

Conda is the package manager, and can also use other sources of packages. It allows more freedom, while Anaconda is more curated (but can be more limited).
Nowadays, there is also Mamba, as a faster alternative to Conda.

(Virtual) environments

Containers

Conda are nearly a all-in-one operating system (OS). They run inside the current OS, but can run another OS.

The basics are fairly straightforward to do: you start with a good default container (a minimal Ubuntu container), and add the packages you want, then install further software you want.

A user gets the whole container *image*, and uses the main container software to run that image, and the code inside of it. No changes to the user's directory or the OS are made, other than the installation of the container software and the image(s).

The most well-known container software is **Docker**. It's a single application (macOS, Windows, Linux), and many base containers exists.
A disadvantage is that Docker requires root (OS administrative) permissions, which is less safe to unsafe on multi-users computers.

Singularity is another container system, similar to Docker. It is slowly getting popular in HPC, since it does not require root to run, and is thus a lot safer on such systems, while automatically providing a full setup for someone to use.

Note that container systems, while shielded from the host OS, are not necessarily slower. They can be just as fast as more normal programs, but simply do not have access to files and directories on the host system.

Creating a container is not really necessary, but can be a convenience for users of your software.
You will need to take care of input and output data between the host system and the container.

(Virtual) environments

Virtual machine / cloud

Virtual machines are similar to containers: you have a complete OS to be able to setup.

You can also control the "machine" itself better, by tweaking its memory, for example.

Cloud machines are also virtual machines, running remotely, with various "hardware" settings to tweak. Generally, these costs money, but you often pay only for time / CPU used.

Virtual machines and cloud machines are nice for testing, but generally not very feasible for using for a single software project.

Cloud machines can be useful if you want to run something that doesn't require a lot of compute power (for that, use HPC), but needs to run for quite some time.

Testing

Testing your software is obviously important.

Most testing will be done by running your software, and see if the output for a specific input agrees with your expectations.

More importantly, however, is what your code should do with unexpected inputs, and bad intermediate results.
Ideally, it should stop. Either with an error message, or just simply crash.
Continuing with bad input or bad intermediate results may lead to incorrect output, but which could be assumed to be correct.

You should try and test for such cases: provide incorrect input, and verify that your software stops early.

For testing for incorrect intermediate input, you may want to create unit tests

Testing - unit tests

Unit tests are small pieces of standalone code, that use functions from your software, to test these functions with a various (difficult / tricky) input.

Usually, these unit tests are organised together (for example, in a tests/ directory), so that can be quickly run together.

Input could consists of things such as an infinite value (+infinity, -infinity), a non-number value (NaN), an empty string, an empty array or list. For most of these cases, the function being tested should return an appropriate output, or crash (perhaps with an exception and proper error message; but at least stop, not simply continue and pretend there was no problem).

The input can also be more normal numbers, to verify that the algorithm of the function being tested is correct in the code. Testing for inputs of 0 and 1, for example, can help with that.

When all run one after the other, unit tests normally just provide a "failed" or "passed" status. If you provide good names for the tests, it becomes easy to see where it fails (so not `myfunction`, but `test_myfunction_handles_badvalues` ; that unit test should test how myfunction handles inputs such as infinity or NaN.

Unit tests are normally handled by a testing framework. Such a framework makes it easier to run the tests, handle testing crashes, varying the input parameters etc. For Python, `pytest` is popular. For C & C++, many frameworks exist; I use `gtest`. Search around for your language used what an appropriate unit test framework is.

Testing can make good use of virtual environments: test in isolation

Testing - unit tests

Unit tests are small pieces of standalone code, that use functions from your software, to test these functions with a various (difficult / tricky) input.

Usually, these unit tests a

Input could consists of thi an empty array or
list. For most of these cas xception and
proper error message; bu

The input can also be mor de. Testing for
inputs of 0 and 1, for exar

When all run one after the es for the tests, it
becomes easy to see whei ld test how
myfunction handles inputs

!

There is a lot you can unit test

Be careful not got stuck on writing unit tests

Write them for a few essential pieces of code,

or when you encounter an error.

Unit tests are normally handled by a testing framework. Such a framework makes it easier to run the tests, handle testing crashes, varying the input parameters etc. For Python, `pytest` is popular. For C & C++, many frameworks exist; I use `gtest`. Search around for your language used what an appropriate unit test framework is.

Testing can make good use of virtual environments: test in isolation

Testing

Continuous integration

Automatically perform (unit) tests

- when there is a change
- for multiple (virtual) environments

Often integrated with Git hosting services

- potentially for every commit a series of tests is run,
- tests are run in the cloud, for a variety of environments, including OSes
- tests are set up with a configuration file, with a "matrix" of dependencies
    - configuration file part of your repository

Various CI services:

GitHub Actions, GitLab CI/CD, Jenkins, CircleCI

Some dependent on repository hosting service

Some free, some restricted free

Example:

```
name: Python package

on: [push]

jobs:
  build:

    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, macos-latest, windows-latest]
        python-version: ["3.7", "3.8", "3.9", "3.10", "3.11"]
        exclude:
          - os: macos-latest
            python-version: "3.7"
          - os: windows-latest
            python-version: "3.7"

    steps:
      - uses: actions/checkout@v3
      - name: Set up Python ${{ matrix.python-version }}
        uses: actions/setup-python@v4
        with:
          python-version: ${{ matrix.python-version }}
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install ruff pytest
          if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
     - name: Test with pytest
        run: |
          pytest
```

How to organise your code - summary

Goal: Maintainability and reproducibility of the code

Try and adhere to the FAIR principles

Follow standard practices for code organisation

- Project (directory) structure
- Standard build tools for the language
- Adhere to warnings, use linters and formatters
- Write appropriate documentation
- Write appropriate unit tests
- Practice good versioning:
    - small commits with clear messages
    - branches for new features / experiments

Metadata

- README file
- Licence
- Citation file (if relevant)
- Get a DOI
- Maintain and update a project version number
- Manage dependencies and their versions

Extra

Provenance

Description of how something (results, data) was produced

Traceability of a result

Provenance is hard

- input parameters should all be kept
- (hidden) settings fixed in the code
- software version
- versions of dependencies

This should be kept together, preferably in a single file.
Such a file should be kept with the results (not with the software)

Storing log files and input configuration files, together with the software DOI, is a first step.
In addition, you can create a shell script that simply executes the steps necessary to run the software.

But you have to guarantee (somehow) that the shell script was indeed run with that software and those input files!

Provenance

Description of how something (results, data) was produced

Traceability of a result

You could also try and store everything in a container.

The container should have a version (it usually has a unique identifier).

Running the container should reproduce your results.

You'll need to handle any input data (from the host machine to the container),
which should of course match the original input data

A container that becomes outdated ("too old") may not be readable anymore.

Notebooks

Notebooks are excellent for

- data exploration: examine how your data is organised, make quick plots
- code exploration: try out different variants of a function, time the variants
- examples: text interspersed with code and figures
- teaching: combination of code exploration and examples

Bonus: various websites (GitHub, GitLab) provide rendered views of notebooks.
That provides directly viewable examples with your code.

Beware of using notebooks too much.

If you start writing large loops, functions or even classes, consider moving these into a module, package or library of their own, and import / include that part in your notebook.

That module/package/library becomes your actual software (with all the necessary metadata), while the notebook is one way to use the software.

Notebooks

Notebooks are not so good for FAIR, reproducible science

Note: somewhat contrary opinion

- They don’t contain a list of dependencies
- The way of working with notebooks is often non-linear
  (always **restart the kernel and rerun all cells**)
- Notebooks are tricky for versioning software
  metadata inside the notebook (execution date and time, for example), continuously
  changes

GitHub / GitLab

Services to collaborate using Git

Create a GitHub account

Follow the prompts on GitHub

Create and upload an ssh key

In a terminal:

```
cd
mkdir .ssh
chmod 0700 .ssh
cd .ssh
ssh-keygen -t ed25519 -f github
# (empty passphrase)
ls
cat github.pub
```

Copy that line

Now go to GitHub and log in

Go to user (top right), then settings

Find "ssh and gpg keys", then "new ssh key"

Past into the key field. For the title, I often use the name of my machine

Back in the terminal, create or edit a `config` file in the `.ssh/` directory. Add the following section:

```
host github
     HostName github.com
     User your-github-username
     IdentityFile ~/.ssh/github
```

GitHub / GitLab

Services to collaborate using Git

Create a new, blanco repository on GitHub (+ button in the top right). Don't add README, Licence or .gitignore files

There will be a few options that GitHub suggests; we pick a variant of the third one:

```
  git remote add origin git@github.com:<username>/<repository-name>.git
  git push origin main
```

Add the remote as shown, then push your existing repository to GitHub.

Refresh the GitHub page.

Clone it somewhere else to test: click the green "code" button on GitHub, then choose the SSH tab. Copy that url, then in a terminal on your machine:

```
git clone <paste GitHub ssh url>
```

```
cd <new-repository-directory>
```

edit / add / commit

Git merge conflicts

To do:

Simulate two conflicting contributions in local repository

- Create a feature branch
- Create a commit
- Go back to main, create another feature
- Create a very similar commit
- Merge one branch
- Now attempt to merge the other branch
- Go into an editor, look at sections between >>>, === and <<<
- Edit as necessary
- Resolve conflict
- Check git log, git status
- Delete feature branches

Git - special files & directories

Preliminary note: files starting their name with a single dot, `.<filename>`, are hidden in a simple listing with `ls`. Use `ls -a` to show these files as well.

- Do **not** remove the `.git` directory! This is where Git stores all the commit details.
- A `.gitignore` file can contain file names and patterns (wildcards: *.bck) to have files ignored by Git. The .gitignore file should be added to your Git repository.
- Files with editor preferences, such as .vscode, do **not** belong to your repository; they belong to users / developers. You can add such files to a .gitignore file.
- Files with a continuous integration setups and other service specific configurations, are often in their own dot-files or -subdirectories, such as `.circle` or `.github`. While they are not part of the repository/software, for practical purposes, you can leave them in.
- You can create so-called **hooks** in a Git repository (they live in `.git/hooks/`. These can perform actions when committing files, or pushing or pulling repositories.
  I use them to check that files don't fail the basic linting steps (clean output) and have no trailing whitespace.
  These files can be simple script; search around if you want to use them.
- Git works on files (changes), not on directories. An empty directory will be ignored by Git.
  If you do want to store an empty directory in your repository, put a special (dot-)file in that directory. For example, `empty/.gitkeep`

References & other resources

- The Turing way: https://the-turing-way.netlify.app/index.html
- FAIR principles: https://www.go-fair.org/fair-principles/
- FAIR checklist: https://fair-software.nl/
- If you do want to create reproducible notebooks: https://github.com/jupyter-guide/ten-rules-jupyter