Methods and Tools

Interactive Lectures

All lectures in the course will be interactive

They contain running code, as well as theory!

- Presented and discussed in frontal lectures...
- ...You can download PDFs
- ...But you will also be able to make changes and experiment

From a software perspective, the workshorses of this approach are:

- Jupyter notebooks for the presentation & interaction
- Poetry dependency and virtual environment manager

You can read more about poetry in the online documentation

If you don't like poetry, a `requirements.txt` file is also included in each lecture

Our Setup

We will often work with this development setup

Every lecture will be structured as follows:

```
data              <-- datasets
notebooks         <-- notebooks and code
pdfs              <-- PDF notes
LICENSE           <-- license file
README.md         <-- usage instructions
requirements.txt <-- dependencies, in classical format
pyproject.toml    <-- main poetry configuration file
poetry.lock       <-- specific package versions, for poetry
```

Our Setup

The notebook folder in turn will be structured as:

```
notebook1.pynb
notebook2.pynb
...
util     <-- module
assets   <-- images and such
rise.css <-- for the "slide" mode
```

Our Setup

The notebook folder in turn will be structured as:

```
notebook1.pynb
notebook2.pynb
...
util +-- __init__.py
     +-- XYZ.py       <-- submodule
     +-- YZX.py       <-- submodule
     +-- ...
font
rise.css
```

The most important part: we'll use modules besides notebooks

Our Setup

Working with modules provides some advantages:

We do not need to keep all our code in the notebooks. We can:

- Share functions between cells
- Share functions between notebooks
- IDEs can offer more functionality if they recognize a module

...But also a significant disadvantage:

- Python modules are compiled first when loaded...
- ...The loaded version is not updated when the source changes

This is very inconvenient at development time

Our Setup

We can circumvent this thanks to Jupyter "magic" extensions

The first one is the "autoreload" extension

```
In [1]: %load_ext autoreload
        %autoreload 2
```

- `load ext` will enable the extension
- `autoreload 2` will reload all modules before code execution

This is inefficient, but convenient during development

- Together with the use of volumes (in docker-compose)...
- ...This allows us to update the code without re-building the docker image