The interactive tutorial

We will start by using `Snakefile` and the data in `sample` (see below for folder structure):

```
.
├── README.md
├── Snakefile
├── config
│   ├── config.yaml
│   ├── samples.csv
│   └── trim_plotting.mplstyle
├── environment.yml
├── results
├── runs
│   ├── pep_m_c
│   └── pep_um
├── sample
│   └── data
└── workflow
    ├── Snakefile_to_demo
    ├── rules
    └── scripts
```

A very simple example

Open `Snakefile` in the parent directory.

Note that it contains the following line — the details are not important, this is just to make all the scripts run.

```
configfile: "config/config.yaml"
```

Let's try using Snakemake to clean a single file (this step is to mimic the gromacs processing step).

```
rule clean_data:
    input:
        "sample/data/1-backbone_rmsd_dirty.xvg"
    output:
        "sample/results/1-backbone_rmsd.xvg"
    shell:
        "awk 'NR>1' {input} > {output}"
```

It's good to try a dry run before running the actual Snakemake rule:

```
snakemake -np # dry run
```

After that, try running the Snakemake rule:

```
snakemake -c1 # actual run with one core
```

All well and good, we could have done this easily using Bash, no need for all the bells and whistles of Snakemake. Now, let's add a new rule below it. The `script` directive here to call a python script to do all the hard plotting work — we'll look at how to make Snakemake interact with Python scripts in more detail later.

```
rule clean_data:
    input:
        "sample/data/1-backbone_rmsd_dirty.xvg"
    output:
        "sample/results/1-backbone_rmsd.xvg"
    shell:
        "awk 'NR>1' {input} > {output}"


rule plot_data:
    input:
        "sample/results/1-backbone_rmsd.xvg",
    output:
        "sample/results/1-backbone_rmsd.png",
    script:
        "workflow/scripts/plot_time_series_single.py"
```

Try a dry run again with `snakemake -np` ...wait, why does nothing happen? If no rule/rule output is provided, Snakemake runs the first rule by default. To get snakemake to run the plot_data rule, specify the desired target/output file:

```
snakemake sample/results/1-backbone_rmsd.png -np # dry run
snakemake sample/results/1-backbone_rmsd.png -c1 # actual run
```

Running `snakemake plot_data` will also work in this instance, but this can be a bad idea, as we will soon see.

Now, let's have a look at the Python script. Script paths are provided relative to the Snakefile (see File paths and Snakemake), and within the script, a global `snakemake` object is provided. From this object, the properties of the rule e.g. `input`, `output`, `params` etc. are available.

```python
...
time_file = snakemake.input[0]
...
...
fig.savefig(snakemake.output[0], dpi=600)
```

Note the use of `snakemake.input[0]` etc., as `input` is a list of input files.

Other languages like R, Julia and Rust are also supported, but I've personally never tried them.

An example with wildcards

What happens when there is more than one file to analyse? This is where wildcards come in, and the Snakefile now looks like this:

```
rule clean_data:
    input:
        "sample/data/{i}-backbone_rmsd_dirty.xvg",
    output:
        "sample/results/{i}-backbone_rmsd.xvg",
    shell:
        "awk 'NR>1' {input} > {output}"



rule plot_data:
    input:
        "sample/results/{i}-backbone_rmsd.xvg",
    output:
        "sample/results/{i}-backbone_rmsd.png",
    script:
        "workflow/scripts/plot_time_series_single.py"
```

Now, trying to run `snakemake -np` results in:

```
WorkflowError:
Target rules may not contain wildcards. Please specify concrete files or a rule
without wildcards at the command line, or have a rule without wildcards at the very
top of your workflow (e.g. the typical "rule all" which just collects all results
you want to generate in the end).
```

So, let's specify the output clearly (`{1..2}` is bash brace expansion):

```
snakemake sample/results/{1..2}-backbone_rmsd.png -np
```

Here, Snakemake only offers to make `2-backbone_rmsd.png`, since there are no changes to the input file(s) needed to generate `1-backbone_rmsd.png`. This is a nice thing about Snakemake, it helps to keep track of what work has already been done. As of Snakemake >=7.8.0, Snakemake automatically reruns jobs if parameter, code, input file set or software stack changes.

Now, make the second plot:

```
snakemake sample/results/{1..2}-backbone_rmsd.png -c1
```

And that's how to run a basic Snakemake workflow! There's plenty more than can be done, so now I'll use the other files in the repo to show a more complicated workflow.

The demo tutorial (just to run)

Snakemake searches for Snakefiles in the following order:

```
Snakefile, snakefile, workflow/Snakefile, workflow/snakemake
```

To provide a Snakefile in a different location, use `snakemake -s`. Here, another Snakefile is present at `workflow/Snakefile`, and to see what it does, use:

```
snakemake -s workflow/Snakefile -np
```

In this Snakefile, there is only a single rule called `rule all`. This is a standard way of collating the desired targets into a single rule, to make it easier to run Snakemake. The wildcards needed are filled in using the the IDS and FOLDERS variables. To keep the Snakefile clean, the other rules needed to execute the workflow are placed in the `rules` folder.

```
import pandas as pd


configfile: "config/config.yaml"


IDS = [str(i) for i in config["run_ids"]]
samples = pd.read_csv("config/samples.csv", index_col="folder")
FOLDERS = samples.index.tolist()


# Load rules
include: "rules/protein_rms.smk"
include: "rules/peptide_rms.smk"


rule all:
    input:
        protein_rmsd_single=expand(
            "results/{folder}/protein/{i}-backbone_rmsd.png", folder=FOLDERS, i=IDS
        ),
        protein_rmsd_multi=expand(
            "results/{folder}/protein/backbone_rmsd_all.png", folder=FOLDERS
        ),
        peptide_rmsd_single=expand(
            "results/{folder}/peptide/{i}-peptide_backbone_rmsd.png",
            folder=FOLDERS,
            i=IDS,
        ),
        peptide_rmsd_multi=expand(
            "results/{folder}/peptide/peptide_backbone_rmsd_all.png", folder=FOLDERS
        ),
```

Let's have a look at the rules in `protein.smk`:

```
rule clean_protein_rmsd:
    input:
        "runs/{folder}/{i}-backbone_rmsd_dirty.xvg",
    output:
        "results/{folder}/protein/data/{i}-backbone_rmsd.xvg",
    shell:
        "awk 'NR>1' {input} > {output}"


rule plot_protein_rmsd_single:
    input:
        rules.clean_protein_rmsd.output,
    output:
        "results/{folder}/protein/{i}-backbone_rmsd.png",
    params:
        ylabel="RMSD (Å)",
        time_unit="ns",
        ymax=5,
    script:
        "../scripts/plot_time_series_single.py"


def get_rmsd_xvgs(wildcards):
    return [
        os.path.join(
            "results", wildcards.folder, "protein/data", run + "-backbone_rmsd.xvg"
        )
        for run in IDS
    ]


rule plot_protein_rmsd_all:
    """
    Specify time units (usually ns)
    """
    input:
        get_rmsd_xvgs,
    output:
        "results/{folder}/protein/backbone_rmsd_all.png",
    params:
        ylabel="RMSD (Å)",
        time_unit="ns",
        ymax=5,
    script:
        "../scripts/plot_time_series_multi.py"
```

The first two rules are similar to the rules from the interactive tutorial, but generalised to work on files in different folders. This is useful when processing multiple datasets, e.g. different sets of MD runs in this case. The `params` directive is used to configure the Python script used for plotting.

Before the third rule, an input function is used in order to collate multiple files to be used to make a single plot. This is a way to get around the "wildcards in the input need to be present in the output" restriction in Snakemake.

The `peptide.smk` file is similar, but with different target folders and params.

With this, it is now easy to run all the analysis for protein and peptide backbone RMSD, and this process can even be done in parallel.

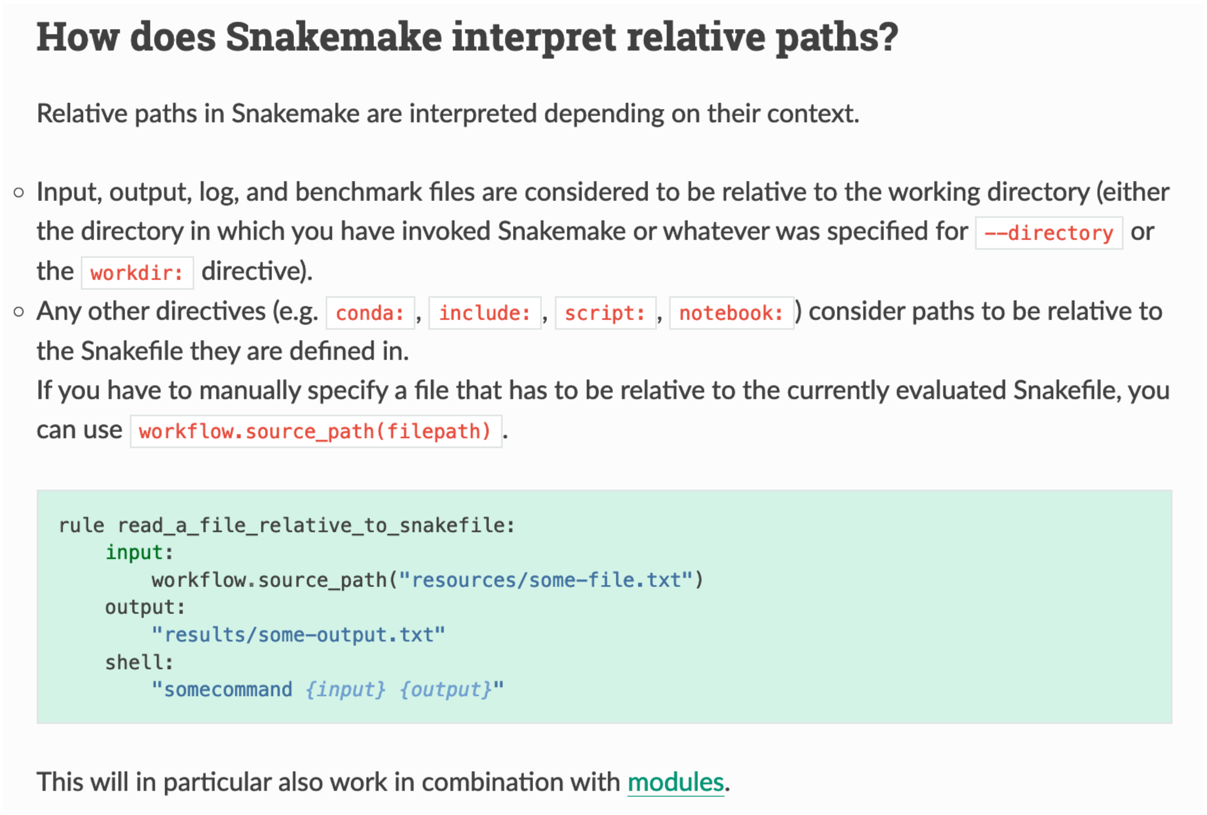
```
snakemake -s workflow/Snakefile -c4 # to run on four cores
```

And we're done with plotting all the data!

Finally, let's say we don't like the ylim used for the peptides plot. Changing the `ylim` parameter and rerunning Snakemake will fix that.

File paths and Snakemake

Taken from the Snakemake FAQs:
https://snakemake.readthedocs.io/en/stable/project_info/faq.html#how-does-snakemake-interpret-relative-paths



How does Snakemake interpret relative paths?

Relative paths in Snakemake are interpreted depending on their context.

- Input, output, log, and benchmark files are considered to be relative to the working directory (either the directory in which you have invoked Snakemake or whatever was specified for `--directory` or the `workdir:` directive).
- Any other directives (e.g. `conda:`, `include:`, `script:`, `notebook:`) consider paths to be relative to the Snakefile they are defined in.

If you have to manually specify a file that has to be relative to the currently evaluated Snakefile, you can use `workflow.source_path(filepath)`.

```
rule read_a_file_relative_to_snakefile:
    input:
        workflow.source_path("resources/some-file.txt")
    output:
        "results/some-output.txt"
    shell:
        "somecommand {input} {output}"
```

This will in particular also work in combination with modules.