Snakemake for reproducible analyses

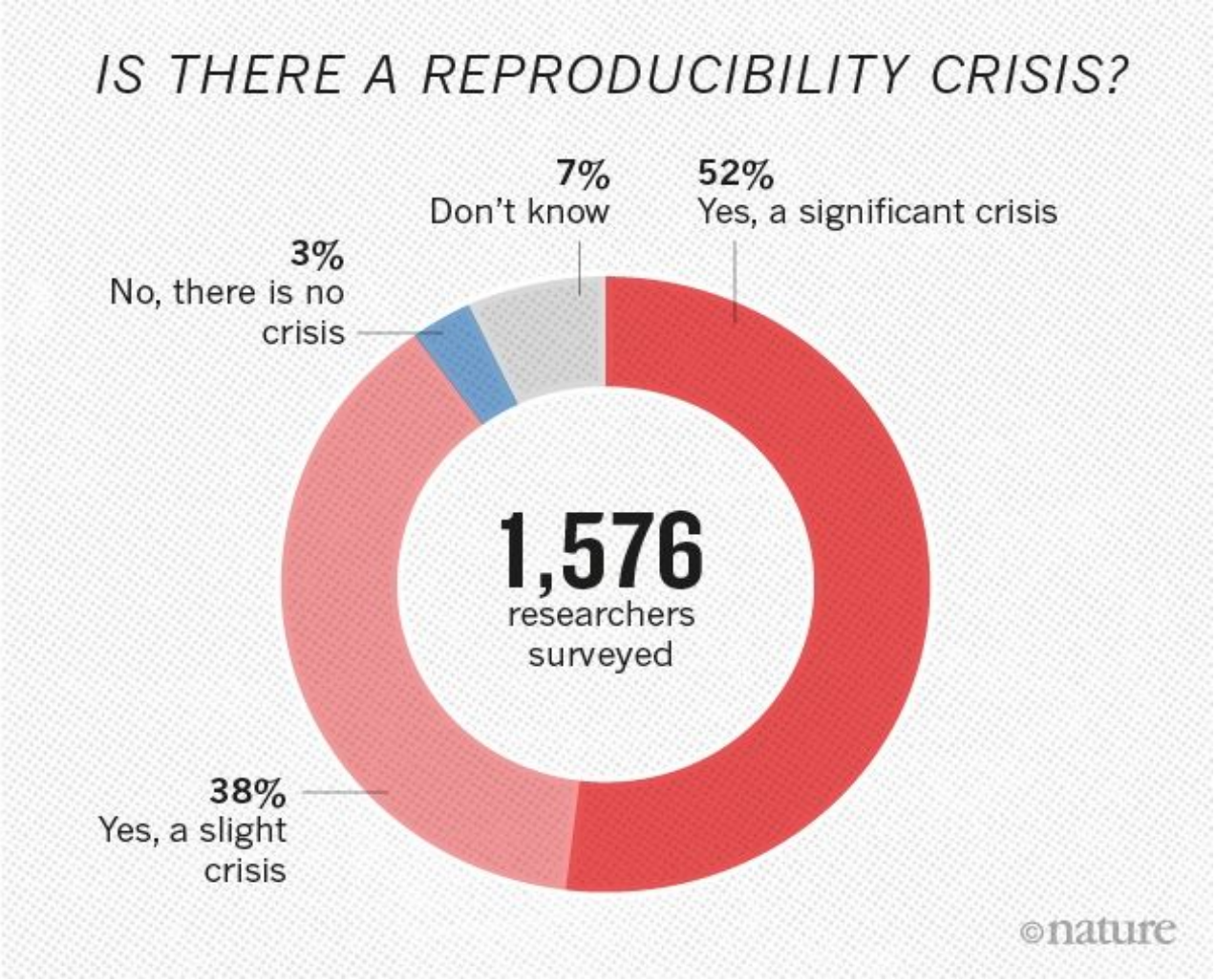
UNIL 2025

Marina Brasó-Vives[1], Alexandre Laverré[1] & Thibault Latrille[2]

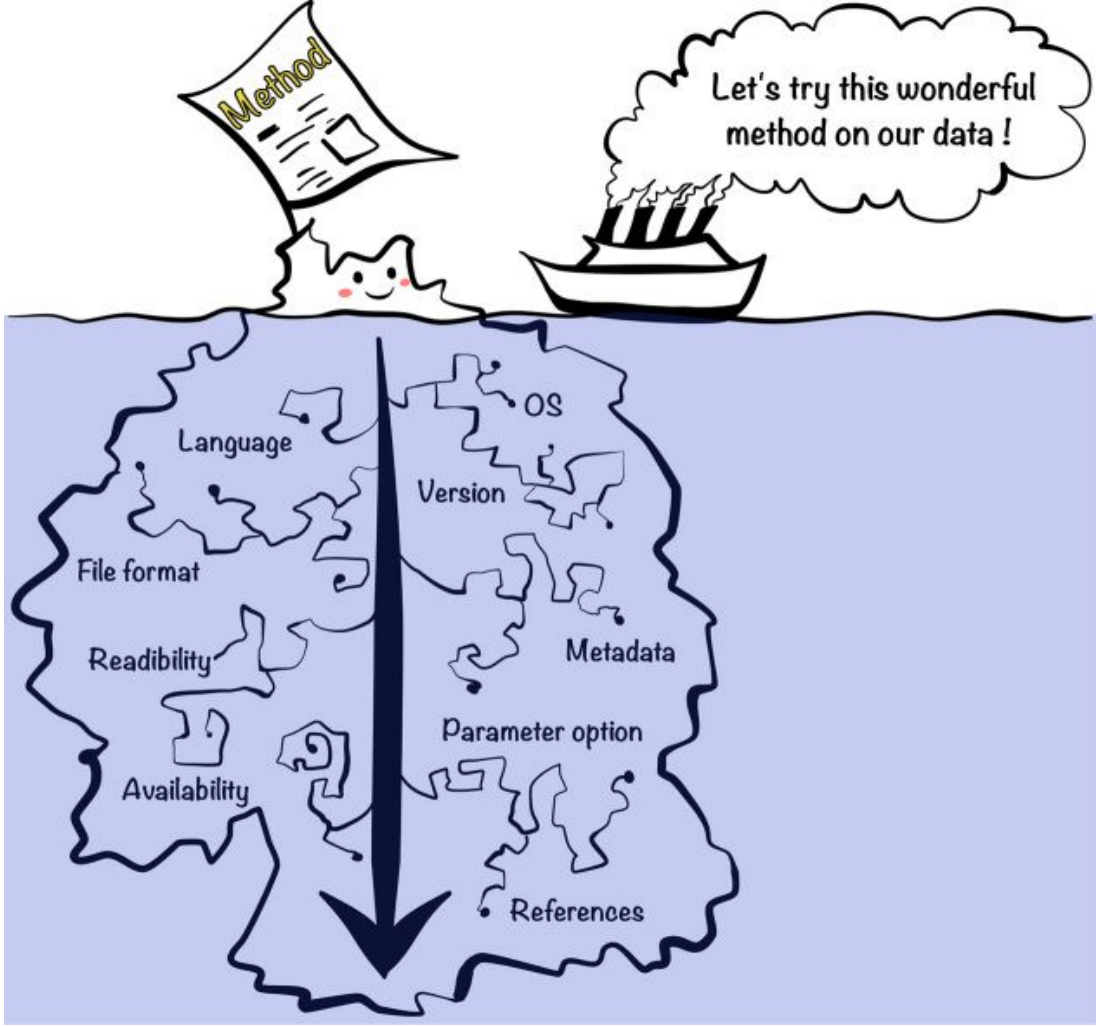
Workshop created with love by Romain Feron in 2022

1: UNIL, DEE, SIB. 2: UNIL, DBC

Marina.BrasoVives@unil.ch; Alexandre.Laverre@unil.ch ; Thibault.Latrille@unil.ch

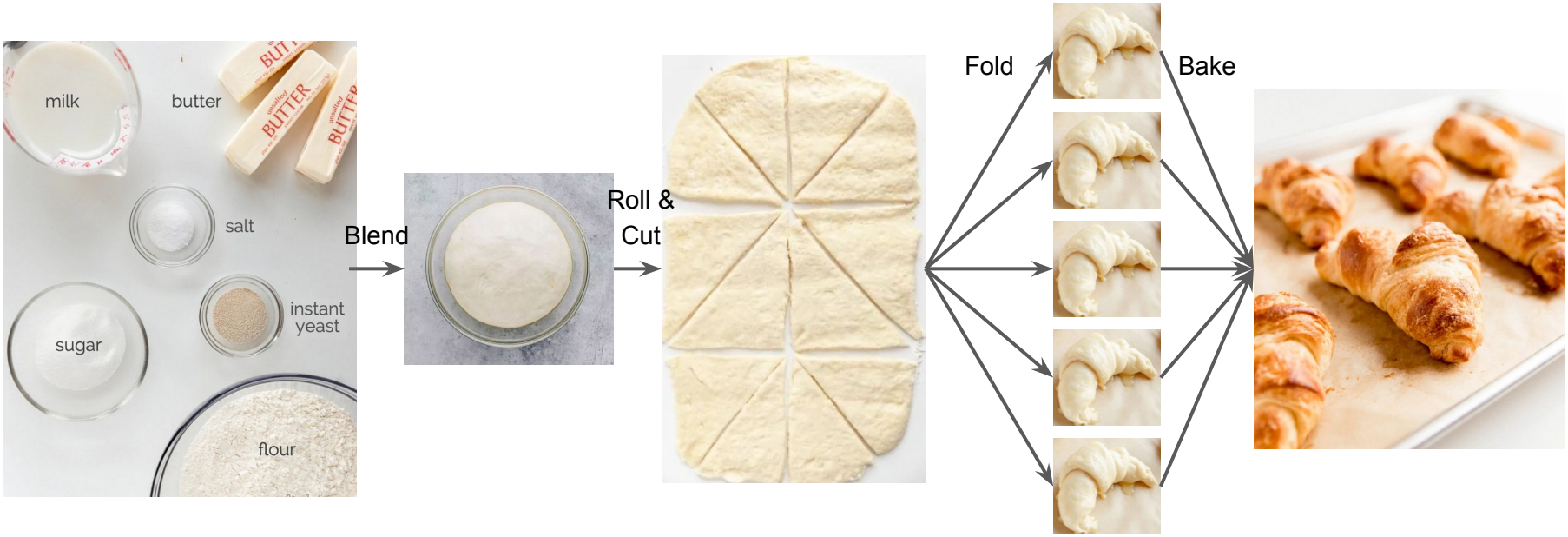


https://www.nature.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970



Kim YM, Poline JB, Dumas G. Experimenting with reproducibility: a case study of robustness in bioinformatics. *Gigascience*. 2018;7(7):giy077.

How to make croissants? (reproducibly)



- Use a comprehensive recipe (a.k.a. workflow management system)

Workflow management systems

- Purpose : implement reproducible, portable, and scalable data analyses
- Two “parts”:
  - Workflow definition language ⇒ implement the workflow
  - Workflow execution system ⇒ run the workflow in variable environments
- Multiple systems exist. Most popular ones are:
  - **NextFlow**: dataflow “top-down” approach, implemented in Groovy (Java)
  - **Snakemake**: make-like “bottom-up” approach resolving dependencies, implemented in Python

Workflow management systems

- Purpose : implement reproducible, portable, and scalable data analyses
- Two “parts”:
  - Workflow definition language ⇒ implement the workflow
  - Workflow execution system ⇒ run the workflow in variable environments
- Multiple systems exist. Most popular ones are:
  - **NextFlow**: dataflow “top-down” approach, implemented in Groovy (Java)
  - **Snakemake**: make-like “bottom-up” approach resolving dependencies, implemented in Python

Overview of Snakemake's features

- User-friendly language: superset of **Python**
- Can be easily executed on local machines, HPCs, and clouds
- Handles dependencies with **Conda** (package manager)
- With Snakemake and conda installed, you can:
  - Download a workflow (*e.g.* from Github)
  - Run Snakemake
  - Automatically reproduce all the results

Structure of this workshop

- 13:30 - 14:00 -- Introduction and presentation of basic concepts
- 14:00 - 15:00 -- Exercises
- 15:00 - 15:20 -- Break
- 15:20 - 15:45 -- Presentation of advanced concepts
- 15:45 - 16:45 -- Exercises
- 16:45 - 17:00 -- Final remarks and conclusion

- Questions are welcome anytime!

Basic concepts

Basic concepts

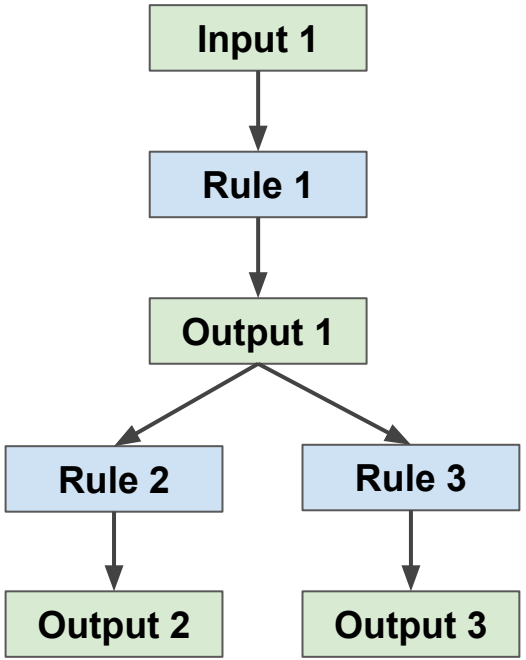
- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- Avoid hard-coding file names
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

Basic concepts

- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- Avoid hard-coding file names
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

Workflow structure

- **Workflow**:
  - Collection of **interdependent** rules to generate **specific outputs**
- **Rule**:
  - Basic workflow unit
  - **Template (recipe)** to produce an **output** (1 or more files)
  - *Can* use an **input**
  - Generates **jobs** when executed
- **Job**:
  - Single **execution** of a rule (apply the recipe to specific data)
  - Successful if **all outputs are present** and **no error**



Workflow structure

Workflow **definition**

```
workflow_dir/
└── Snakefile
```

- Rules are defined in a file called **Snakefile**
- Snakefile is located at the root of the workflow directory
- Paths in Snakefile are relative to the directory containing Snakefile

Workflow structure

Workflow

- Rule
- Sna
- Path

```
rule first_step:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'

rule second_step:
    input:
        'results/first_step.txt'
    output:
        'results/second_step.txt'
    shell:
        'cat {input} | grep “snakemake” > {output}'
```

Workflow structure

Workflow **definition**

```
workflow_dir/
└── Snakefile
```

- Rules are defined in a file called **Snakefile**
- Snakefile is located at the root of the workflow directory
- Paths in Snakefile are relative to the directory containing Snakefile

Workflow **execution**

- Command “`snakemake --cores 1 <output>`” executed from the workflow directory
- Read the rules defined in **Snakefile**
- Computes all jobs necessary to generate <output>

Basic concepts

- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- Avoid hard-coding file names
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

Defining simple rules

- Keyword
- Rule name
- Directives
- File
- Shell command
- Comment

Rule:

- Defined with the **keyword rule**
- User-defined **name**
- Comprised of several **directives**

```
rule first_step:
    output:
        'results/first_step.txt'
    shell:
        'echo "snakemake" > {output}'
```

Defining simple rules

- Keyword
- Rule name
- Directives
- File
- Shell command
- Comment

Rule:

- Defined with the **keyword rule**
- User-defined **name**
- Comprised of several **directives**
- Directives have **values:**
  - Instruction (commands)
  - File names
  - Numeric values...

```
rule first_step:
    output:
        'results/first_step.txt'
    shell:
        'echo "snakemake" > {output}'
```

Here, **the value** is an **instruction:**
“How to generate the output ?”

Defining simple rules

- Keyword
- Rule name
- Directives
- File
- Shell command
- Comment

- Once defined, **directive** values can be accessed in the **shell directive**
  - Here, we use the value of “output”
- If part of a path does not exist, it will be created automatically
  - Here, the “results” directory is created

```
rule first_step:
    output:
        'results/first_step.txt'
    shell:
        'echo "snakemake" > {output}'
```

Value from the directive 'output'

Defining simple rules

Adding directives

- Keyword
- Rule name
- Directives
- File
- Shell command
- Comment

- Most rules use an input
- If the input file doesn't exist, jobs cannot be executed

```
rule first_step:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

Value from directive 'input'

Value from directive 'output'

Defining simple rules
Commenting

- Keyword
- Rule name
- Directives
- File
- Shell command
- Comment

- For clarity and reproducibility, it's important to document your code!

```
rule first_step:
    '''
    Copy input file to output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

Basic concepts

- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- Avoid hard-coding file names
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

Executing workflows

Snakefile

```
rule first_step:
    '''
    Copy input file to output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

Executing workflows

Snakefile

```
rule first_step:
    '''
    Copy input file to output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

```
snakemake --cores 1 <output>
```

Executing workflows

Snakefile

```
rule first_step:
    '''
    Copy input file to output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

```
snakemake --cores 1 results/first_step.txt
```

Executing workflows

Snakefile

```
rule first_step:
    '''
    Copy input file to output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

```
snakemake --cores 1 results/first_step.txt
```

Target = output that you want to generate

Executing workflows

Snakefile

```
rule first_step:
    '''
    Copy input file to output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

```
snakemake --cores 1 results/first_step.txt
```

This command will generate a **job**
= application of the rule **first_step**

Executing workflows

Snakefile

```
rule first_step:
    '''
    Copy input file to output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

Before execution:

```
workflow_dir/
├── data/
│   └── first_step.tsv
└── Snakefile
```

Executing workflows

Before execution:

```
workflow_dir/
├── data/
│   └── first_step.tsv
└── Snakefile
```

Snakefile

```
rule first_step:
    '''
    Copy input file to output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

```
snakemake --cores 1 results/first_step.txt
```

Executing workflows

Snakefile

```
rule first_step:
    '''
    Copy input file to output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

```
snakemake --cores 1 results/first_step.txt
```

Before execution:

```
workflow_dir/
├── data/
│   └── first_step.tsv
└── Snakefile
```

After execution:

```
workflow_dir/
├── data
│   └── first_step.tsv
├── results
│   └── first_step.txt
└── Snakefile
```

Basic concepts

- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- Avoid hard-coding file names
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

Multiple inputs

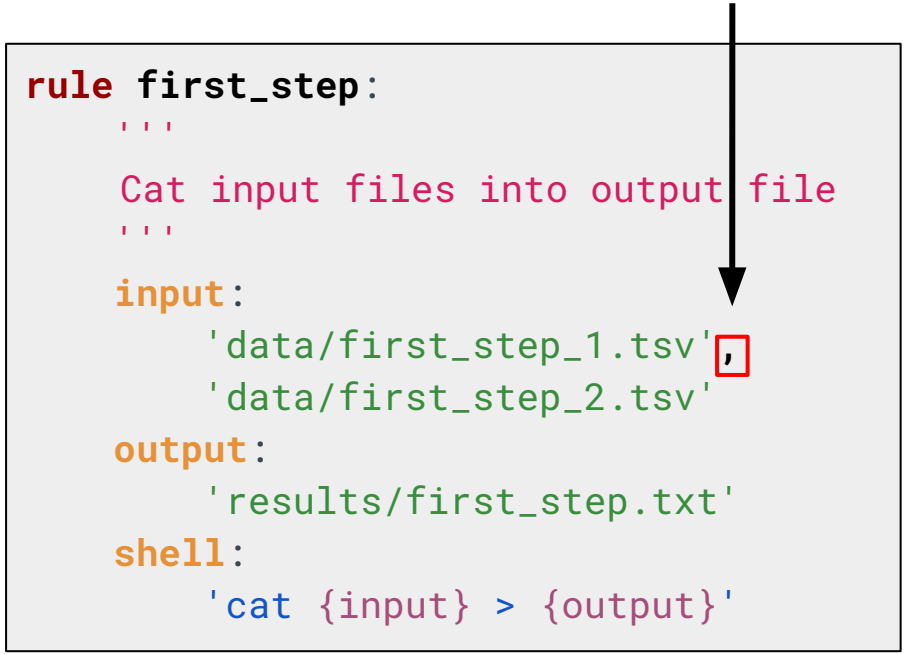
- Rules can use more than one input
- Don’t forget the comma!

```
rule first_step:
    '''
    Cat input files into output file
    '''
    input:
        'data/first_step_1.tsv',
        'data/first_step_2.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cat {input} > {output}'
```

Multiple inputs

- Rules can use more than one input
- Don’t forget the comma!

Input directive values are concatenated



```
rule first_step:
    '''
    Cat input files into output file
    '''
    input:
        'data/first_step_1.tsv',
        'data/first_step_2.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cat {input} > {output}'
```

```
cat data/first_step_1.tsv data/first_step_2.tsv > results/first_step.txt
```

Multiple inputs

- Rules can use more than one input
- Don’t forget the comma!
- Inputs can be accessed by their positional index: *input[n]*

```
rule first_step:
    '''
    Cat input files into output file
    '''
    input:
        'data/first_step_1.tsv',
        'data/first_step_2.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cat {input[0]} > {output};'
        'cat {input[1]} >> {output}'
```

Commands are concatenated

Multiple inputs

- Inputs can be named for clarity
- Named input can be accessed by their names: *input.input_name*

```
rule first_step:
    '''
    Cat input files into output file
    '''
    input:
        input_1 = 'data/first_step_1.tsv',
        input_2 = 'data/first_step_2.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cat {input.input_1} > {output};'
        'cat {input.input_2} >> {output}'
```

Multiple outputs

Outputs work just like inputs

- Multiple output separated by ‘,’
- Outputs can be named
- Can be accessed by positional index or by name
- All output files will be generated by Snakemake

```
rule first_step:
    '''
    Cat input file 1 into output file 1 and
    input file 2 into output file 2
    '''
    input:
        input_1 = 'data/first_step_1.tsv',
        input_2 = 'data/first_step_2.tsv'
    output:
        output_1 = 'results/first_step_1.txt',
        output_2 = 'results/first_step_2.txt'
    shell:
        'cat {input.input_1} > {output.output_1};'
        'cat {input.input_2} >> {output.output_2}'
```

```
snakemake --cores 1 results/first_step_1.txt
```

↳ results/first_step_1.txt, results/first_step_2.txt

Basic concepts

- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- Avoid hard-coding file names
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

Rules dependencies

```
rule first_step:
    '''
    Copy input file into output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in input to output
    '''
    input:
        'results/first_step.txt'
    output:
        'results/second_step.txt'
    shell:
        'cat {input} | grep “snakemake” > {output}'
```

Rules dependencies

```
rule first_step:
    '''
    Copy input file into output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in input to output
    '''
    input:
        'results/first_step.txt'
    output:
        'results/second_step.txt'
    shell:
        'cat {input} | grep “snakemake” > {output}'
```

```
snakemake --cores 1 results/second_step.txt
```

Rules dependencies

```
rule first_step:
    '''
    Copy input file into output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in input to output
    '''
    input:
        'results/first_step.txt'
    output:
        'results/second_step.txt'
    shell:
        'cat {input} | grep “snakemake” > {output}'
```

```
--cores 1 results/second_step.txt
```

- Does the input of rule **second_step** exist ?
  ---> NO

-

Rules dependencies

```
rule first_step:
    '''
    Copy input file into output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in input to output
    '''
    input:
        'results/first_step.txt'
    output:
        'results/second_step.txt'
    shell:
        'cat {input} | grep “snakemake” > {output}'
```

```
snakemake --cores 1 results/second_step.txt
```

- Does the input of rule **second_step** exist ?
  ---> NO

- Look for a rule generating that input
  ---> rule **first_step**

Rules dependencies

```
rule first_step:
    '''
    Copy input file into output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in input to output
    '''
    input:
        'results/first_step.txt'
    output:
        'results/second_step.txt'
    shell:
        'cat {input} | grep “snakemake” > {output}'
```

```
snakemake --cores 1 results/second_step.txt
```

- Does the input of rule **second_step** exist ?
  ---> NO
- Look for a rule generating that input
  ---> rule **first_step**
- Does the input of rule **first_step** exist?
  ---> YES

Rules dependencies

```
rule first_step:
    '''
    Copy input file into output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in input to output
    '''
    input:
        'results/first_step.txt'
    output:
        'results/second_step.txt'
    shell:
        'cat {input} | grep “snakemake” > {output}'
```

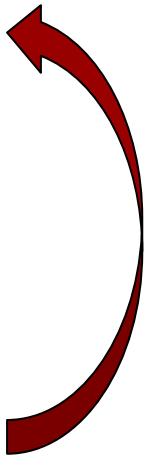
```
snakemake --cores 1 results/second_step.txt
```

- Does the input of rule **second_step** exist ?
  ---> NO
- Look for a rule generating that input
  ---> rule **first_step**
- Does the input of rule **first_step** exist?
  ---> YES
- All good, execute the workflow

Rules dependencies

```
rule first_step:
    '''
    Copy input file into output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in input to output
    '''
    input:
        'results/first_step.txt'
    output:
        'results/second_step.txt'
    shell:
        'cat {input} | grep “snakemake” > {output}'
```



Dependency between rule *second_step* and rule *first_step*.

Rules dependencies

```
rule first_step:
    '''
    Copy input file into output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in input to output
    '''
    input:
        'results/first_step.txt'
    output:
        'results/second_step.txt'
    shell:
        'cat {input} | grep “snakemake” > {output}'
```

- Core concept of Snakemake: resolving input/output dependencies
- For each job: determine if input exists, otherwise look for rule that generates it
- Snakemake computes a **Directed Acyclic Graph (DAG)** resolving all dependencies

Basic concepts

- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- Avoid hard-coding file names
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

Wildcards: Snakemake "variables"

“Hardcoded” input and output files

```
rule first_step:
    '''
    Copy tsv into txt
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

Wildcards: Snakemake "variables"

“Hardcoded” input and output files

```
rule first_step:
    '''
    Copy tsv into txt
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

“General” input and output files with wildcards

```
rule first_step:
    '''
    Copy sample tsv into txt
    '''
    input:
        'data/{sample}.tsv'
    output:
        'results/{sample}.txt'
    shell:
        'cp {input} {output}'
```

Wildcards: inferred from output

```
rule first_step:
    '''
    Copy sample tsv into txt
    '''
    input:
        'data/{sample}.tsv'
    output:
        'results/{sample}.txt'
    shell:
        'cp {input} {output}'
```

Wildcards are “resolved” from the output and propagated to other directives

```
snakemake --cores 1 results/first_step.txt
```

Snakemake interpretation:

`{sample} = "first_step"`

Wildcards in workflows

- A workflow can use multiple wildcards
- A single rule can use multiple (different) wildcards

```
rule first_step:
    '''
    Copy sample_treatment tsv into sample_treatment txt
    '''
    input:
       'data/{sample}_{treatment}.tsv'
    output:
       'results/{sample}_{treatment}.txt'
    shell:
        'echo {wildcards.sample};'
        'cp {input} {output}'
```

Wildcard values can be accessed in ‘shell’

Wildcards in workflows

- A workflow can use multiple wildcards
- A single rule can use multiple (different) wildcards
- Input and output files do not have to share the same wildcards

```
rule first_step:
    '''
    Copy sample tsv into sample_treatment txt
    '''
    input:
       'data/{sample}.tsv'
    output:
       'results/{sample}_{treatment}.txt'
    shell:
        'echo {wildcards.sample};'
        'grep “{wildcards.treatment}” {input} > {output}'
```

```
snakemake --cores 1 results/sample1_control.txt
```

→ Input: **data/sample1.tsv**

Wildcards in workflows

- **All files generated by a rule need to have the same wildcards!**

```
rule first_step:
    '''
    Cat sample_treatment tsv into sample_treatment txt &
    keep info lines in sample info.txt
    '''
    input:
        'data/{sample}_{treatment}.tsv'
    output:
        'results/{sample}_{treatment}.txt',
        'results/{sample}_info.txt'
    shell:
        'cat {input} > {output[0]};'
        'grep "info" {input} > {output[1]}'
```

!!!

```
snakemake --cores 1 results/sample1_control.txt
```

Output:
- results/sample1_control.txt
- results/sample1_info.txt

```
snakemake --cores 1 results/sample1_1day.txt
```

Output:
- results/sample1_1day.txt
- results/sample1_info.txt

```
snakemake --cores 1 results/sample1_info.txt
```

Output: **???????**

Wildcards in workflows

- A workflow can use multiple wildcards
- A single rule can use multiple (different) wildcards
- Input and output files do not have to share the same wildcards
- **All files generated by a rule need to have the same wildcards!**

```
rule first_step:
    '''
    Copy sample_treatment tsv into sample_treatment txt
    '''
    input:
       'data/{sample}_{treatment}.tsv'
    output:
       'results/{sample}_{treatment}.txt'
    shell:
        'echo {wildcards.sample};'
        'cp {input} {output}'
```

Basic concepts

- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- Avoid hard-coding file names
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

Avoid hard-coding file names

```
rule first_step:
    '''
    Copy input file into output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in input to output
    '''
    input:
        'results/first_step.txt'
    output:
        'results/second_step.txt'
    shell:
        'cat {input} | grep “snakemake” > {output}'
```

In concatenated rules, filenames are written (hardcoded) more than once

Avoid hard-coding file names

```
rule first_step:
    '''
    Copy input file into output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in input to output
    '''
    input:
        'results/first_step.txt'
    output:
        'results/second_step.txt'
    shell:
        'cat {input} | grep “snakemake” > {output}'
```

In concatenated rules, filenames are written (hardcoded) more than once

What happens if we decide to change one??

- We have to change ALL of them!
- It can lead to errors

Avoid hard-coding file names

```
rule first_step:
    '''
    Copy input file into output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in input to output
    '''
    input:
        rules.first_step.output
    output:
        'results/second_step.txt'
    shell:
        'cat {input} | grep “snakemake” > {output}'
```

In concatenated rules, filenames are written (hardcoded) more than once

What happens if we decide to change one??

We have to change ALL of them!

It can lead to errors

Solution:

Reference the output of the previous rule

Avoid hard-coding file names

```
rule first_step:
    '''
    Copy tsv to txt &
    keep info lines in info txt
    '''
    input:
        'data/first_step.tsv'
    output:
        outTXT = 'results/first_step.txt',
        outINFO = 'results/first_info.txt'
    shell:
        'cp {input} {output.outTXT};'
        'grep “info” {input} > {output.outINFO}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in outTXT from first_step to output
    '''
    input:
        rules.first_step.output.outTXT
    output:
        'results/second_step.txt'
    shell:
        'grep “snakemake” {input} > {output}'
```

What if we have multiple output files?

Avoid hard-coding file names

```
rule first_step:
    '''
    Copy sample_treatment tsv into sample_treatment txt
    '''
    input:
        'data/{sample}_{treatment}.tsv'
    output:
        sample_treatmentTXT = 'results/{sample}_{treatment}.txt',
        sample_treatmentINFO = 'results/{sample}_{treatment}_info.txt'
    shell:
        'cp {input} {output.sample_treatmentTXT};'
        'grep “info” {input} > {output.sample_treatmentINFO}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in sample_treatmentTXT from first_step to output
    '''
    input:
        rules.first_step.output.sample_treatmentTXT
    output:
        'results/{sample}_{treatment}_step.txt'
    shell:
        'grep “snakemake” {input} > {output}'
```

What if we have wildcards?

Basic concepts

- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- Avoid hard-coding file names
- The "expand" syntax
- Workflow execution
- Non-file rule parameters
- Executing Python / R code

The “expand” syntax

```
rule first_step:
    '''
    Copy input files into output file
    '''
    input:
        'data/A.tsv',
        'data/B.tsv',
        'data/C.tsv',
        'data/D.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cat {input} > {output}'
```

The “expand” syntax

expand() is only a generator of file names from a pattern

```
rule first_step:
    '''
    Copy input files into output file
    '''
    input:
        'data/A.tsv',
        'data/B.tsv',
        'data/C.tsv',
        'data/D.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cat {input} > {output}'
```

```
rule first_step:
    '''
    Copy sample input files into output file
    '''
    input:
        expand('data/{sample}.tsv', sample=['A','B','C','D'])
    output:
        'results/first_step.txt'
    shell:
        'cat {input} > {output}'
```

The “expand” syntax

expand() is only a generator of file names from a pattern

```
samples = [‘A’, ‘B’]
replicates = [1, 2]

rule first_step:
    '''
    Copy {sample}_{replicate} TSV files into output file
    '''
    input:
        expand(‘data/{sample}_{replicate}.tsv’, sample=samples, replicate=replicates)
    output:
        'results/first_step.txt'
    shell:
        'cat {input} > {output}'
```

The “expand” syntax

expand() is only a generator of file names from a pattern

```
samples = [‘A’, ‘B’]
replicates = [1, 2]

rule first_step:
    '''
    Copy {sample}_{replicate} TSV files into output file
    '''
    input:
        expand(‘data/{sample}_{replicate}.tsv’, sample=samples, replicate=replicates)
    output:
        'results/first_step.txt'
    shell:
        'cat {input} > {output}'
```

data/A_1.tsv
data/A_2.tsv
data/B_1.tsv
data/B_2.tsv

→ Expands a wildcard expression to a series of wildcard values.

The “expand” syntax

The wildcards defined in expand are
INDEPENDENT from any other wildcard in the rule

```
samples = [‘A’, ‘B’]
replicates = [1, 2]

rule first_step:
    '''
    Copy {sample}_{replicate} TSV files into output file
    '''
    input:
        expand(‘data/{sample}_{replicate}.tsv’, sample=samples, replicate=replicates)
    output:
        'results/first_step.txt'
    shell:
        'cat {input} > {output}'
```

The “expand” syntax

The wildcards defined in expand are
INDEPENDENT from any other wildcard in the rule

```
samples = [‘A’, ‘B’]

rule first_step:
    '''
    Copy {sample}_{replicate} TSV files into sample TXT file
    '''
    input:
        expand(‘data/{sample}.tsv’, sample=samples)
    output:
        'results/{sample}.txt'
    shell:
        'cat {input} > {output}'
```

In this case, the value of the {sample} wildcard will NOT be propagated to the input

Basic concepts

- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- Avoid hard-coding file names
- The "expand" syntax
- Workflow execution
- Non-file rule parameters
- Executing Python / R code

Execution

- Specify a target

```
snakemake --cores 1 results/second_step.txt
```

```
rule first_step:
    '''
    Copy input file into output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in input to output
    '''
    input:
        'results/first_step.txt'
    output:
        'results/second_step.txt'
    shell:
        'cat {input} | grep “snakemake” > {output}'
```

Execution

- Specify a target

```
snakemake --cores 1 results/second_step.txt
```

- Without target: snakemake will use the **output** of the **first rule** found in the Snakefile as a target

```
snakemake --cores 1
```

=

```
snakemake --cores 1 results/first_step.txt
```

```
rule first_step:
    '''
    Copy input file into output file
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'

rule second_step:
    '''
    Copy the lines containing “snakemake” in input to output
    '''
    input:
        'results/first_step.txt'
    output:
        'results/second_step.txt'
    shell:
        'cat {input} | grep “snakemake” > {output}'
```

Execution

- To avoid specifying a target, we usually create a dummy first rule calling all your desired output:

```
rule all:
    '''
    Define all desired output
    '''
    input:
       last='results/last_step.txt'
```

→ Snakemake will resolve all the dependencies needed to obtain these files

```
snakemake --cores 1
```

=

```
snakemake --cores 1 results/last_step.txt
```

Execution

- Important! First rule **cannot have wildcards** (impossible to resolve)

```
rule all:
    '''
    Define all desired output
    '''
    input:
       last='results/last_step_{sample}.txt'
```

- But you can use it to defined them for the rest of your workflow:

```
rule all:
    '''
    Define all desired output
    '''
    input:
       last=expand(results/last_step_{sample}.txt', sample=['A,'B','C','D'])
```

Execution

- By default, existing outputs are not generated again if input is unchanged

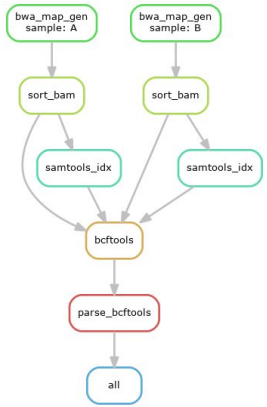
```
snakemake --cores 1 --force <target> / --forceall
```

- Dry-run: see what snakemake would do, without actually doing it

```
snakemake --cores 1 --dry-run <target>
```

- Visualize the DAG:

```
snakemake --cores 1 --dag <target> | dot -Tpng > dag.png
```



Basic concepts

- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- Avoid hard-coding file names
- The "expand" syntax
- Workflow execution
- Non-file rule parameters
- Executing Python / R code

Non-file rule parameters

```
rule first_step:
    '''
    Copy the first 5 lines of TSV into TXT
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    shell:
        'head -n 5 {input} > {output}'
```

Stuck with only the first **5** lines of the input

Non-file rule parameters

```
rule first_step:
    '''
    Copy the first 5 lines of TSV into TXT
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    params:
        5
    shell:
        'head -n {params} {input} > {output}'
```

- Directive **params**

Non-file rule parameters

```
rule first_step:
    '''
    Copy the first 5 lines of TSV into TXT
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    params:
        5
    shell:
        'head -n {params} {input} > {output}'
```

- Directive **params**
- Accessible in shell

Non-file rule parameters

```
rule first_step:
    '''
    Copy the first n_lines lines of TSV into TXT
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    params:
        n_lines = 5
    shell:
        'head -n {params.n_lines} {input} > {output}'
```

- Directive **params**
- Accessible in shell
- Parameters can be named (and they should)

Basic concepts

- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- Avoid hard-coding file names
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

The 'run' directive

```
rule first_step:
    '''
    Copy the first n_lines lines of TSV into TXT
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    params:
        n_lines = 5
    run:
        input_file = open(input[0])
        output_file = open(output[0], 'w')
        for i in range(params.n_lines):
            output_file.write(input_file.readline())
```

- Execute Python code directly from a Snakefile with **run**
- Replaces **shell**

The 'run' directive

```
rule first_step:
    '''
    Copy the first n_lines lines of TSV into TXT
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    params:
        n_lines = 5
    run:
        input_file = open(input[0])
        output_file = open(output[0], 'w')
        for i in range(params.n_lines):
            output_file.write(input_file.readline())
```

- Execute Python code directly from a Snakefile with **run**
- Replaces **shell**
- Directive values can be accessed like in **shell**

The 'script' directive

Snakefile

```
rule first_step:
    '''
    Copy the first n_lines lines of TSV into TXT
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    params:
        lines = 5
    script:
        'first_step.py'
```

first_step.py

```
# Retrieve information from Snakemake
input_file = open(snakemake.input[0])
output_file = open(snakemake.output[0], 'w')
n_lines = snakemake.params.lines

# Process file
for i in range(n_lines):
    output_file.write(input_file.readline())
```

- Call an external Python script from Snakemake with **script**
- Directives and values can be accessed from a **snakemake** Python object

The 'script' directive

Snakefile

```
rule first_step:
    '''
    Copy the first n_lines lines of TSV into TXT
    '''
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    params:
        lines = 5
    script:
        'first_step.R'
```

first_step.R

```
library(readr)

# Retrieve information from Snakemake
input_file_path <- snakemake@input[[1]]
output_file_path <- snakemake@output[[1]]
n_lines <- snakemake@params$lines[1]

# Open input file
data <- read_delim(input_file_path, '\t', n_max = n_lines)
```

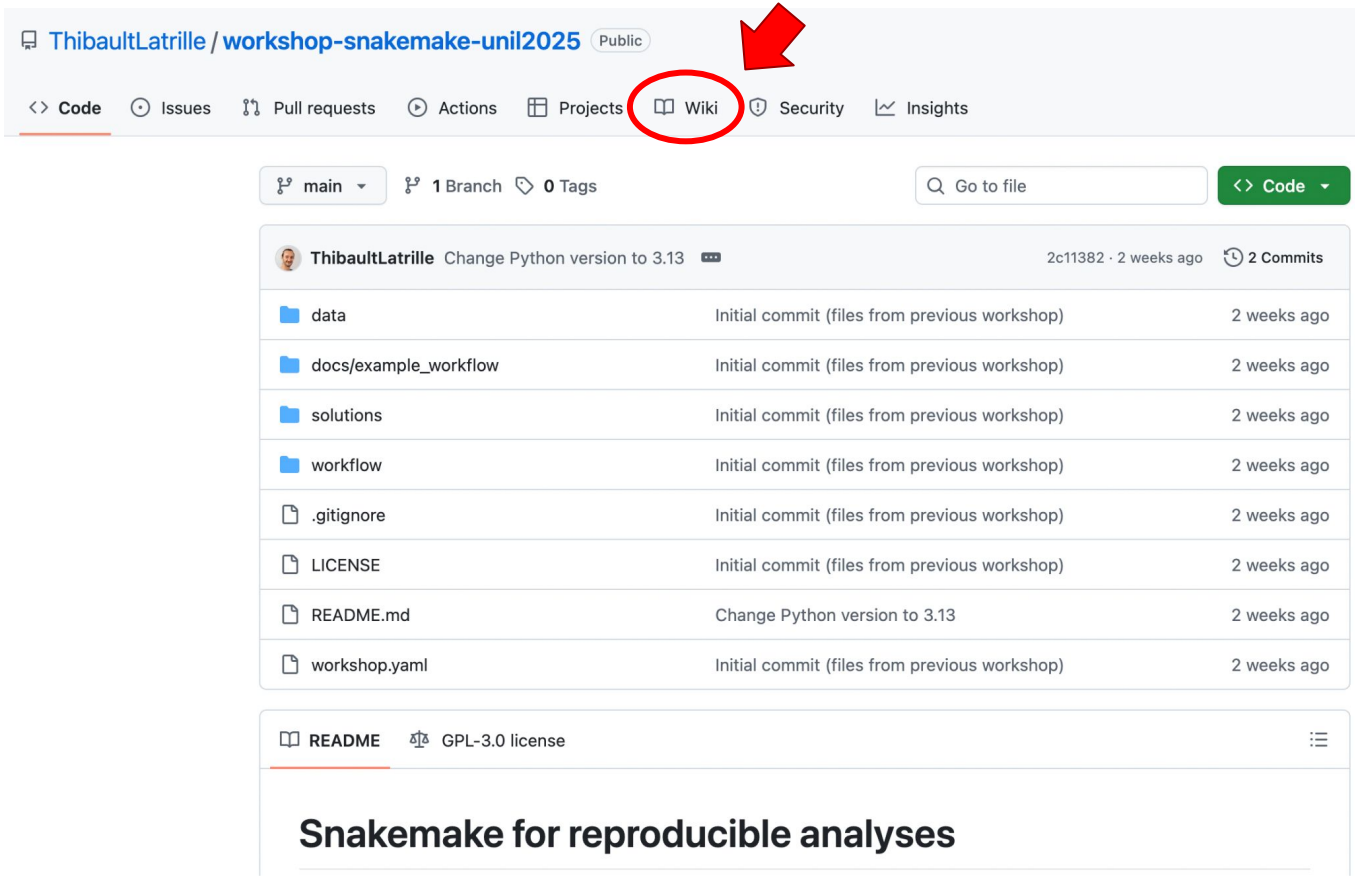
- Call an external Python script from Snakemake
- Directives and values can be accessed from a **snakemake** Python object
- Other supported languages:
  - R
  - Julia
  - Rust

Basic concepts

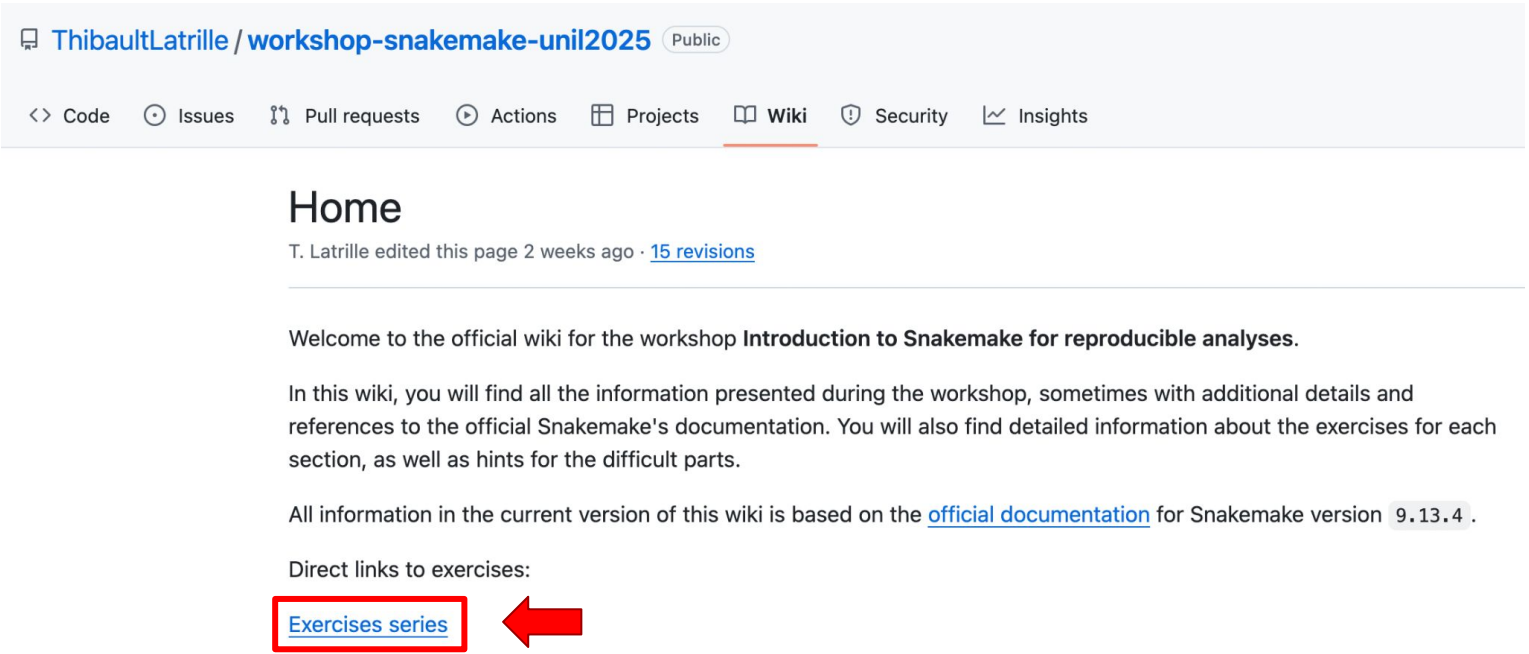
- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- The "expand" syntax
- Workflow execution
- Non-file rule parameters
- Executing Python code

Hands on !
Exercises series 1

Hands on - Exercises



Hands on - Exercises series



Advanced concepts

Advanced concepts

- Config files
- Advanced directives
- Wildcard constraints
- Rule order
- Functions as input
- Modularization
- Automatic software deployment with Conda
- Running snakemake on clusters and cloud
- Workflow organization guidelines

Advanced concepts

- Config files
- Advanced directives
- Wildcard constraints
- Rule order
- Functions as input
- Modularization
- Automatic software deployment with Conda
- Running snakemake on clusters and cloud
- Workflow organization guidelines

Specifying parameters with a config file

config.yaml

```
lines_number: 5
samples:
    - sample1
    - sample2
resources:
    threads: 4
    memory: 4G
```

Snakefile

```
configfile: 'config.yaml'

rule first_step:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    params:
        lines = config['lines_number']
    shell:
        'head -n {params.lines} {input} > {output}'
```

- The config file is parsed into a ‘config’ dictionary that can be accessed inside rule definitions

Specifying parameters with a config file

- Snakemake has 2 ways to read information from a config file :
  - Specify the file at runtime
  - Add a line at the top of Snakefile

```
snakemake --cores 1 --configfile config.yaml
```

```
configfile: 'config.yaml'
```

- Config files are either YAML (preferred) or JSON files

- Lists of parameters become list:

```
config['samples'] = ['sample1', 'sample2']
```

- Lists of named parameters become dictionaries:

```
config['resources'] = {threads:4, memory:'4G'}
```

```yaml
lines_number: 5
samples:
    - sample1
    - sample2
resources:
    threads: 4
    memory: 4G
```

Advanced concepts

- Config files
- Advanced directives
- Wildcard constraints
- Rule order
- Functions as input
- Modularization
- Automatic software deployment with Conda
- Running snakemake on clusters and cloud
- Workflow organization guidelines

Advanced directives

threads

- The ‘threads’ directive specifies the number of threads to allocate to each job spawned by a rule. Syntax : “threads: <number_of_threads>”.

```
rule first_step:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    threads: 4
    shell:
        'command --threads {threads} {input} > {output}'
```

- In local mode, the total number of threads allocated to Snakemake is constrained by the execution parameter “--cores”

Advanced directives

log

- The ‘log’ directive specifies the path to a log file for a rule
- Logs still need to be handled manually for each command, but now Snakemake automatically creates the directory in the log file path
- **Every output files** of a rule must have the **same wildcards**!

```
rule first_step:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    log:
        'logs/first_step.log'
    shell:
        'command {input} > {output} 2> {log}'
```

Advanced directives

log

- The ‘log’ directive specifies the path to a log file for a rule
- Logs still need to be handled manually for each command, but now Snakemake automatically creates the directory in the log file path
- **Every output files** of a rule must have the **same wildcards**!

```
rule first_step:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    log:
        err = 'logs/first_step.err',
        out = 'logs/first_step.out'
    shell:
        'command {input} -o {output} > {log.out} 2> {log.err}'
```

Advanced directives

benchmark

- Snakemake can measure runtime and memory usage for a given rule
- The ‘benchmark’ directive specifies the path to save a benchmark file.

```
rule first_step:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    benchmark:
        'benchmarks/first_step.txt'
    shell:
        'command {input} > {output}'
```

s	h:m:s	max_rss	avg_rss	max_vms	avg_vms	io_in	io_out	mean_load
12.354	0:00:12	420.15MB	415.32MB	1.54GB	1.51GB	5.2MB	48.6MB	0.87

Advanced directives
benchmark

- The ‘benchmark’ directive specifies the path to a benchmark results file for a rule. Syntax : “benchmark : <path/to/benchmark/file.txt>”
- Snakemake will automatically measure runtime and memory usage for the rule and save it to the file
- Snakemake can repeat measurements with the syntax “benchmark : repeat(<path/to/benchmark/file.txt>, N)”
- Benchmark files must have the **same wildcards** as the **output** !
- It’s a best to regroup benchmarks in a “benchmarks” folder in your workflow

Advanced directives

message

- The ‘message’ directive print a custom message to the terminal when a rule starts
- It can helps to monitor the pipeline, or for debugging → printed also in log files
- Can be used to print dynamic information (wildcards, in-rule parameters, variables …)

```
rule filter_data:
    message:
        'Running filtering step (X={params.X}) on {wildcards.sample} for {species}...'
    input:
        'data/{sample}.tsv'
    output:
        'results/{sample}_filtered.txt'
    params:
        X=10
    shell:
        'command {params.X} {input} > {output}'
```

Advanced directives

priority

- ‘priority’ controls which rules run first among those that are ready (it can’t override dependencies)
- With large workflow, and limited resources, it can be strategic to obtain certain files earlier:
  - Sanity check to avoid a late crash (file existence, quality control steps…)
  - Start “bottleneck” or heavy steps sooner (indexing, model training…)
  - Run small, fast rules to prevent cluster overload by parallel large jobs

- Higher priority → scheduled earlier
- Values are arbitrary
- Default = 0

```
rule validate_config:
    priority: 200

rule bottleneck_step:
    priority: 100

rule small_rule:
    priority: 50
```

Advanced concepts

- Config files
- Advanced directives
- Wildcard constraints
- Rule order
- Functions as input
- Modularization
- Automatic software deployment with Conda
- Running snakemake on clusters and cloud
- Workflow organization guidelines

Wildcard constraints

- Sometimes it is useful to constrain the values a wildcard can have.
- This can be achieved by adding a regular expression that describes the set of allowed wildcard values.

```
rule filter_data:
    input:
        'data/{sample,[A-Za-z0-9]+}.tsv'
    output:
        'results/{sample,[A-Za-z0-9]+}.txt'
    shell:
        'command {input} > {output}'
```

Wildcard constraints

- Sometimes it is useful to constrain the values a wildcard can have.
- This can be achieved by adding a regular expression that describes the set of allowed wildcard values.

```
rule filter_data:
    input:
        'data/{sample}.tsv'
    output:
        'results/{sample}.txt'
    wildcard_constraints:
        sample = '[A-Za-z0-9]+'
    shell:
        'command {input} > {output}'
```

Wildcard constraints

- Sometimes it is useful to constrain the values a wildcard can have.
- This can be achieved by adding a regular expression that describes the set of allowed wildcard values.
- You can also define global wildcard constraints that apply to all rules

```
wildcard_constraints:
    sample = '[A-Za-z0-9]+'

rule filter_data:
    input:
        'data/{sample}.tsv'
    output:
        'results/{sample}.txt'
    shell:
        'command {input} > {output}'
```

Advanced concepts

- Config files
- Advanced directives
- Wildcard constraints
- Rule order
- Functions as input
- Modularization
- Automatic software deployment with Conda
- Running snakemake on clusters and cloud
- Workflow organization guidelines

Rule order

- If several rules can produce the same output file: `AmbiguousRuleException`
  Snakemake need you to decide which one to execute

- Can be useful when:
  - Several methods can be used to obtain a file
  - Use of both generic and specific rules

```
rule any_chromosome:
    output:'results/{chr}.vcf

rule specific_chromosome:
    output:'results/chrM.vcf'

ruleorder: specific_chromosome > any_chromosome
```

Rule order

- If several rules can produce the same output file: `AmbiguousRuleException`
  Snakemake need you to decide which one to execute

- Can be useful when:
  - Several methods can be used to obtain a file

```
rule download_genome:
    output:'results/{species}.fasta'

rule build_genome:
    output:'results/{species}.fasta'

if config["genome"] == "public":
    ruleorder: download_genome > build_genome
else:
    ruleorder: build_genome > download_genome
```

Advanced concepts

- Config files
- Advanced directives
- Wildcard constraints
- Rule order
- Functions as input
- Modularization
- Automatic software deployment with Conda
- Running snakemake on clusters and cloud
- Workflow organization guidelines

Using functions as input

- Situation : input files depend on wildcards in a non-trivial way
- Input functions are Python functions that take “wildcards” as single argument and return a file or list of files.
- Define function above the rule, then use syntax “`input` : <function_name>”

Using functions as input

```
def get_fasta(wildcards):
    clade = wildcards.clade
    if clade == 'primates':
        return 'data/Primates_241sp_filtered.fasta
    else:
        return 'data/HRA_190_c55_s2.fasta


rule filter_fasta:
    input:
        get_fasta
    output:
        'results/{clade}.fasta
    shell:
        'command {input} > {output}'
```

Using functions as input

- Situation : input files depend on wildcards in a non-trivial way
- Input functions are Python functions that take “wildcards” as single argument and return a file or list of files.
- Define function above the rule, then use syntax “`input` : <function_name>”
- Functions are evaluated before executing the workflow.
- You can write any python code inside the function.
- You can use function for `params` directive too (when parameters depends on wildcards in a non-trivial way)

Advanced concepts

- Config files
- Advanced directives
- Wildcard constraints
- Rule order
- Functions as input
- Modularization
- Automatic software deployment with Conda
- Running snakemake on clusters and cloud
- Workflow organization guidelines

Modularization: subfiles

- If you have a complex workflow with many rules, you can organize them in different Snakefiles (.smk)
- Different Snakefiles could contain meaningful groups of rules

  For example: VariantCalling.smk (from fastq to vcf)

Modularization: subfiles

- If you have a complex workflow with many rules, you can organize them in different Snakefiles (.smk)
- Different Snakefiles could contain meaningful groups of rules

  For example: VariantCalling.smk (from fastq to vcf)

- You still need a “main Snakefile” with the “first rule” (normally *rule all*)

Snakefile

```
include: 'first_steps.smk'

rule all:
    input:
        'results/first_step.txt'
```

first_steps.smk

```
rule first_step:
    input:
        'results/first_step.txt'
    output:
        'results/first_step.txt'
    shell:
        'cp {input} {output}'
```

Modularization: subfiles

Snakefile

```
include: 'VariantCalling.smk'
include: 'PopulationSubstructure.smk'
include: 'GeneticDiversity.smk'
include: 'GeneExpression.smk'
include: 'ManuscriptFigures.smk'

rule all:
    input:
        'results/Figure1.pdf',
        'results/Figure2.pdf',
        'results/Figure3.pdf',
        'results/Figure4.pdf',
        'results/Table1.tbl',
        'results/SupplementaryFigures.pdf'
```

```
my_repository/
├── data
│   └── ...
├── results
│   └── ...
└── workflow
    └──rules
        └── VariantCalling.smk
        └── PopulationSubstructure.smk
        └── GeneticDiversity.smk
        └── GeneExpression.smk
        └── ManuscriptFigures.smk
    └── Snakefile
```

Advanced concepts

- Config files
- Advanced directives
- Wildcard constraints
- Rule order
- Functions as input
- Modularization
- Automatic software deployment with Conda
- Running snakemake on clusters and cloud
- Workflow organization guidelines

Automatic deployment of software with Conda

VariantCalling.smk

```
rule SAM2SortedBAM:
    '''
    Sort SAM and convert it to BAM
    '''
    input:
        sam = 'results/{sample}.sam'
    output:
        bam = 'results/{sample}_sorted.bam'
    shell:
        'samtools sort -O bam {sam} > {bam}'
```

Automatic deployment of software with Conda

VariantCalling.smk

```
rule SAM2SortedBAM:
    '''
    Sort SAM and convert it to BAM
    '''
    input:
        sam = 'results/{sample}.sam'
    output:
        bam = 'results/{sample}_sorted.bam'
    shell:
        'samtools sort -O bam {sam} > {bam}'
```

Automatic deployment of software with Conda

VariantCalling.smk

```
rule SAM2SortedBAM:
    '''
    Sort SAM and convert it to BAM
    '''
    input:
        sam = 'results/{sample}.sam'
    output:
        bam = 'results/{sample}_sorted.bam'
    shell:
        'samtools sort -O bam {sam} > {bam}'
```

Snakemake provides Conda integration!

Automatic deployment of software with Conda

VariantCalling.smk

```
rule SAM2SortedBAM:
    '''
    Sort SAM and convert it to BAM
    '''
    input:
        sam = 'results/{sample}.sam'
    output:
        bam = 'results/{sample}_sorted.bam'
    shell:
        'samtools sort -O bam {sam} > {bam}'
```

Snakemake manages software for you!

Automatic deployment of software with Conda

VariantCalling.smk

```
rule SAM2SortedBAM:
    '''
    Sort SAM and convert it to BAM
    '''
    input:
        sam = 'results/{sample}.sam'
    output:
        bam = 'results/{sample}_sorted.bam'
    conda:
        '../envs/VariantCalling.yaml'
    shell:
        'samtools sort -O bam {sam} > {bam}'
```

VariantCalling.yaml

YAML file

```
name: VariantCalling
channels:
  - conda-forge
  - bioconda
dependencies:
  - samtools=1.17
```

Conda: open-source package and environment manager (Windows, macOS, linux)

Automatic deployment of software with Conda

VariantCalling.smk

```
rule SAM2SortedBAM:
    '''
    Sort SAM and convert it to BAM
    '''
    input:
        sam = 'results/{sample}.sam'
    output:
        bam = 'results/{sample}_sorted.bam'
    conda:
        '../envs/VariantCalling.yaml'
    shell:
        'samtools sort -O bam {sam} > {bam}'
```

VariantCalling.yaml

```
name: VariantCalling
channels:
  - conda-forge
  - bioconda
dependencies:
  - samtools=1.17
  - bwa=0.7.17
  - picard=2.26.4
  - gatk4=4.2.4.0
  - vcftools=0.1.16
```

Conda: open-source package and environment manager (Windows, macOS, linux)

Channels: repository of software, packaged and maintained

- Conda-forge: lots of general software, often used
- Bioconda: specifically for bioinformatics software

Automatic deployment of software with Conda

VariantCalling.smk

```
rule SAM2SortedBAM:
    '''
    Sort SAM and convert it to BAM
    '''
    input:
        sam = 'results/{sample}.sam'
    output:
        bam = 'results/{sample}_sorted.bam'
    conda:
        '../envs/VariantCalling.yaml'
    shell:
        'samtools sort -O bam {sam} > {bam}'
```

VariantCalling.yaml

```
name: VariantCalling
channels:
  - conda-forge
  - bioconda
dependencies:
  - samtools=1.17
  - bwa=0.7.17
  - picard=2.26.4
  - gatk4=4.2.4.0
  - vcftools=0.1.16
```

Execution parameter “--use-conda”

```
snakemake --cores 1 –use-conda
```

Automatic deployment of software with Conda

```
my_repository/
├── data
│   └── ...
├── results
│   └── ...
└── workflow
    └──envs
       └── VariantCalling.yaml
    └──rules
       └── VariantCalling.smk
    └── Snakefile
```

Automatic deployment of software with Conda

```
my_repository/
├── data
│   └── ...
├── results
│   └── ...
└── workflow
    ├──envs
    │   ├── VariantCalling.yaml
    │   ├── PopulationSubstructure.yaml
    │   └── ManuscriptFigures.yaml
    ├──rules
    │   ├── VariantCalling.smk
    │   ├── PopulationSubstructure.smk
    │   └── ManuscriptFigures.smk
    └── Snakefile
```

Automatic deployment of software with Containers

VariantCalling.smk

```
rule SAM2SortedBAM:
    '''
    Sort SAM and convert it to BAM
    '''
    input:
        sam = 'results/{sample}.sam'
    output:
        bam = 'results/{sample}_sorted.bam'
    container:
        'docker://biocontainers/samtools:v1.9-4-deb_cv1'
    shell:
        'samtools sort -O bam {sam} > {bam}'
```

Execution parameter “--use-singularity”

```
snakemake --cores 1 --use-singularity
```

Advanced concepts

- Config files
- Advanced directives
- Wildcard constraints
- Rule order
- Functions as input
- Modularization
- Automatic software deployment with Conda
- Running snakemake on clusters and cloud
- Workflow organization guidelines

Running snakemake on clusters and cloud

- Snakemake can make use of a scheduler to run jobs on a cluster (SLURM, SGE, …)

Running snakemake on clusters and cloud

- Snakemake can make use of a scheduler to run jobs on a cluster (SLURM, SGE, …)
- General syntax:

```
snakemake --cluster <submit_command>
```

(qsub, sbatch …)

Running snakemake on clusters and cloud

- Snakemake can make use of a scheduler to run jobs on a cluster (SLURM, SGE, …)
- For Curnagl cluster (SLURM) at Unil. :

```
snakemake --jobs 30 --cluster "sbatch --partition=cpu --nodes=1 --job-name={params.name}
                               --cpus-per-task={params.threads} --mem={params.mem} --time={params.time}
                               --output=./logs/.slurm/%x.out --error=./logs/.slurm/%x.err"
```

  - Specify the maximum number of jobs to submit with **“-j / --jobs”**
  - Inside --cluster you can add any SLURM commands (see wiki)
  - You can specify “localrules” in your Snakefile
  - You can use the command **screen** to have Snakemake running on the background

Running snakemake on clusters and cloud

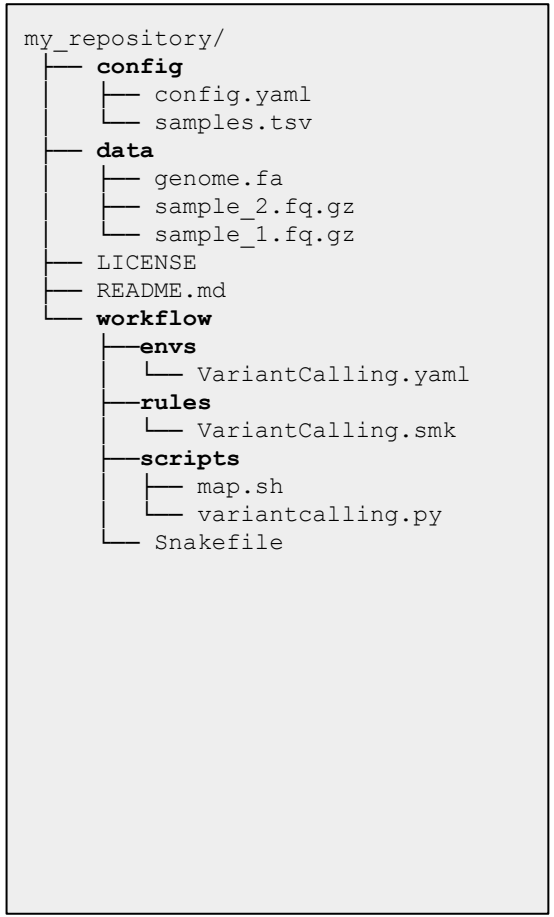
- Snakemake can make use of a scheduler to run jobs on a cluster (SLURM, SGE, …)
- Also, built-in support for Cloud computing (Amazon, Kubernetes,..)

Advanced concepts

- Config files
- Advanced directives
- Wildcard constraints
- Rule order
- Functions as input
- Modularization
- Automatic software deployment with Conda
- Running snakemake on clusters and cloud
- Workflow organization guidelines

How to organize your workflow: best practices

- A **(git) repository** should contain a single workflow
- Use **Conda environments** when possible
- Break out large workflow into **modules** with extension ".smk"
- Specify parameters in a **config file** located in a 'config' folder
- Use a **sample sheet** located in the 'config' folder to store sample metadata



Configuration files (modified by the user)

Conda environment files

Snakemake modules (rules)

Scripts

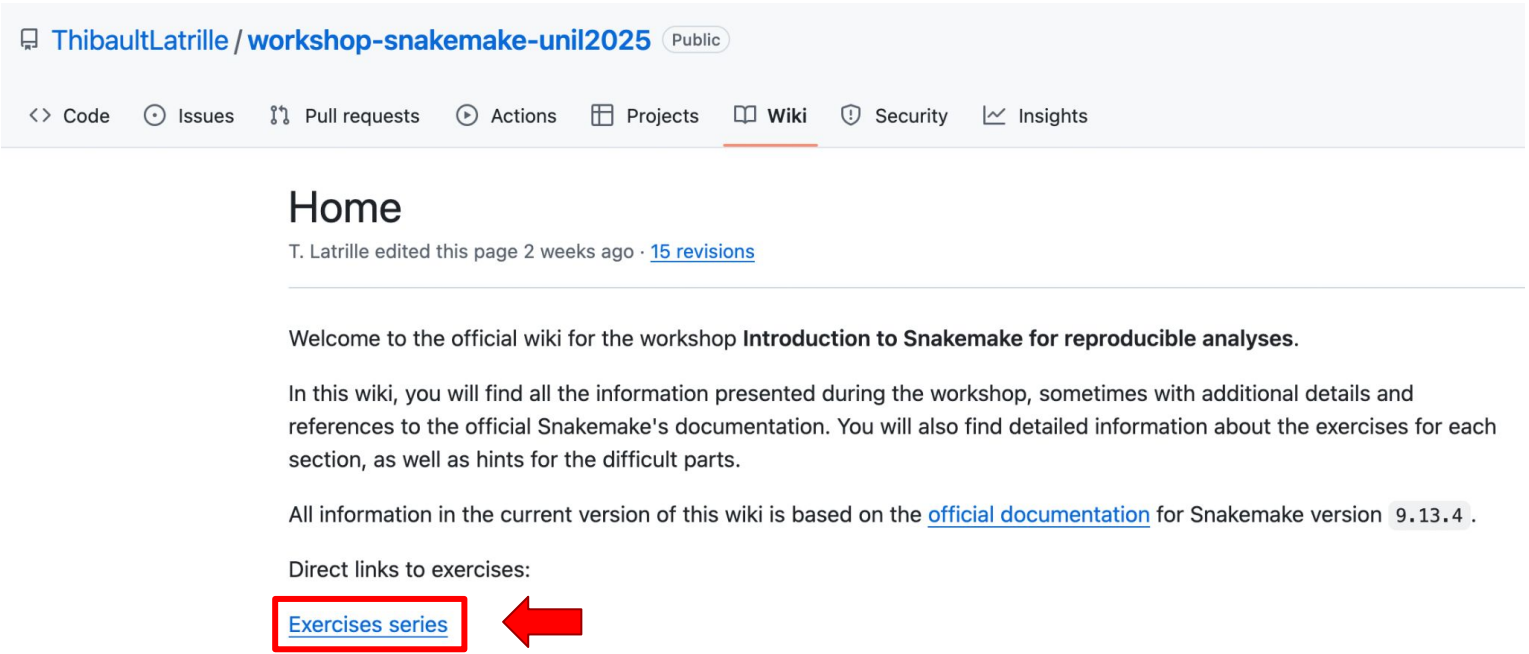
Actual workflow implementation


my_repository/
config
config.yaml
samples.tsv
data
genome.fa
sample_2.fq.gz
sample_1.fq.gz
LICENSE
README.md
workflow
envs
VariantCalling.yaml
rules
VariantCalling.smk
scripts
map.sh
variantcalling.py
Snakefile
Execution
my_repository/
benchmarks
sample_1.txt
sample_2.txt
config
config.yaml
samples.tsv
data
genome.fa
sample_2.fq.gz
sample_1.fq.gz
LICENSE
logs
sample_1.txt
sample_2.txt
README.md
results
sample_1.bam
sample_2.bam
variants.vcf
workflow
envs
VariantCalling.yaml
rules
VariantCalling.smk
scripts
map.sh
variantcalling.py
Snakefile
Benchmarks
Log files
Final results files

```
my_repository/
├── config
│   ├── config.yaml
│   └── samples.tsv
├── data
│   └── ...
├── LICENSE
├── README.md
└── workflow
    ├──envs
    │   ├── VariantCalling.yaml
    │   ├── PopulationSubstructure.yaml
    │   └── ManuscriptFigures.yaml
    ├──rules
    │   ├── VariantCalling.smk
    │   ├── PopulationSubstructure.smk
    │   └── ManuscriptFigures.smk
    ├──scripts
    │   ├── VariantCalling
    │   ├── PopulationSubstructure
    │   └── ManuscriptFigures
    └── Snakefile
```


```
my_repository/
├── benchmarks
│   ├── VariantCalling
│   ├── PopulationSubstructure
│   └── ManuscriptFigures
├── config
│   ├── config.yaml
│   └── samples.tsv
├── data
│   └── ...
├── LICENSE
├── logs
│   ├── VariantCalling
│   ├── PopulationSubstructure
│   └── ManuscriptFigures
├── README.md
├── results
│   ├── VariantCalling
│   ├── PopulationSubstructure
│   └── ManuscriptFigures
└── workflow
    ├──envs
    │   ├── VariantCalling.yaml
    │   ├── PopulationSubstructure.yaml
    │   └── ManuscriptFigures.yaml
    ├──rules
    │   ├── VariantCalling.smk
    │   ├── PopulationSubstructure.smk
    │   └── ManuscriptFigures.smk
    ├──scripts
    │   ├── VariantCalling
    │   ├── PopulationSubstructure
    │   └── ManuscriptFigures
    └── Snakefile
```

Hands on - Exercises series



Concluding remarks

- **Reproducibility** :
  - Workflow ⇒ steps clearly defined, commands saved
  - Conda integration ⇒ perfect handling of software installation and versions
  - Self-contained workflow archive ⇒ other people can easily reproduce your analyses (with almost no programming knowledge)

- **Practical use** :
  - Once workflow is build, can be applied to any number of samples
  - Snakemake does a lot for you !
    - Create directory structure
    - Check job completion, restart if needed
    - Fully handles parallelization of jobs
    - Easy handling of logs and benchmarks
  - Portability and scalability: run on the cloud, on HPCs, and on any UNIX machine

Additional advanced concepts

Special output types

- Outputs can be “decorated” with specific properties
- **Temporary** : “temp(‘path/to/file.txt’)” ⇒ deleted when not required by future jobs
- **Protected** : “protected(‘path/to/file.txt’)” ⇒ cannot be overwritten after job ends
- **Ancient** : “ancient(‘path/to/file.txt’)” ⇒ file will not be re-created when running the pipeline
- **Directory** : “directory(‘path/to/directory’)” ⇒ the output is a directory instead of a file (try to avoid that)

Reproducibility: official wrappers

- Wrappers are scripts that integrate popular software with Snakemake. Main point: you don’t need to write the command yourself

- Wrappers available for many popular tools in the **official wrapper repository** (community-based effort)

```
rule run_tool_wrapper:
    input:
        'data/input.tsv'
    output:
        'results/output.txt'
    wrapper:
        '0.40.2/bio/tool'
```

Instructions for each tool are in the official repository: parameter names, inputs and outputs ...

Reproducibility: official wrappers

- Wrappers are scripts that integrate popular software with Snakemake. Main point: you don’t need to write the command yourself
- Wrappers available for many popular tools in the **official wrapper repository** (community-based effort)
- Wrappers are automatically downloaded and deploy a conda environment when running the workflow. Versions ⇒ increased reproducibility
- Best way to run software when available. Be careful, sometimes their implementation can be “rigid”, and you may have to write your own rule

Execution profiles

- Execution profiles are like presets of runtime parameter values (‘-j <N>’, ‘--use-conda’ …)
- Profile ⇒ directory *~/.config/snakemake/<profile_name>/* (on Linux). Minimum : config.yaml with syntax <runtime_option>: <value>
- Profiles can be extended a lot, especially for HPC environments: scripts to submit jobs and check job status ⇒ advanced customization
- Collection of official profiles on Github. Custom profile for Slurm developed by us

Data-dependent conditional execution

- Situation : rule has variable or unpredictable output (splitting file, clustering, different file types …)
- Solution: checkpoints ⇒ DAG is re-evaluated when output is required
- Syntax : “checkpoint” instead of “rule”, then input function with :

```
checkpoints.<checkpoint_name>.get(**wildcards).output
```

- Since DAG is re-evaluated, you won’t see the whole pipeline at the beginning of a run (no full DAG graph for instance)

Data-dependent conditional execution

```
checkpoint variable_output_rule:
    input:
        'data/{sample}.txt'
    output:
        directory('results/{sample}')
    shell:
        # Split input in files of length 1000 lines starting
        # with the prefix {output}/
        'split {input} {output}/'

def collect_input(wildcards):
    checkpoint = checkpoints.variable_output_rule.get(**wildcards).output[0]
    full_output = expand('results/{sample}/{i}.txt', sample=wildcards.sample,
                         i=glob_wildcards(os.path.join(checkpoint, '{i}.txt')).i)
    return full_output

rule aggregate:
    input:
        collect_input
    output:
        'results/{sample}.txt'
    shell:
        'cat {input} > {output}'
```

Variable output rule

Input function

Rule using checkpoint output