UNIL | Université de Lausanne

SIB

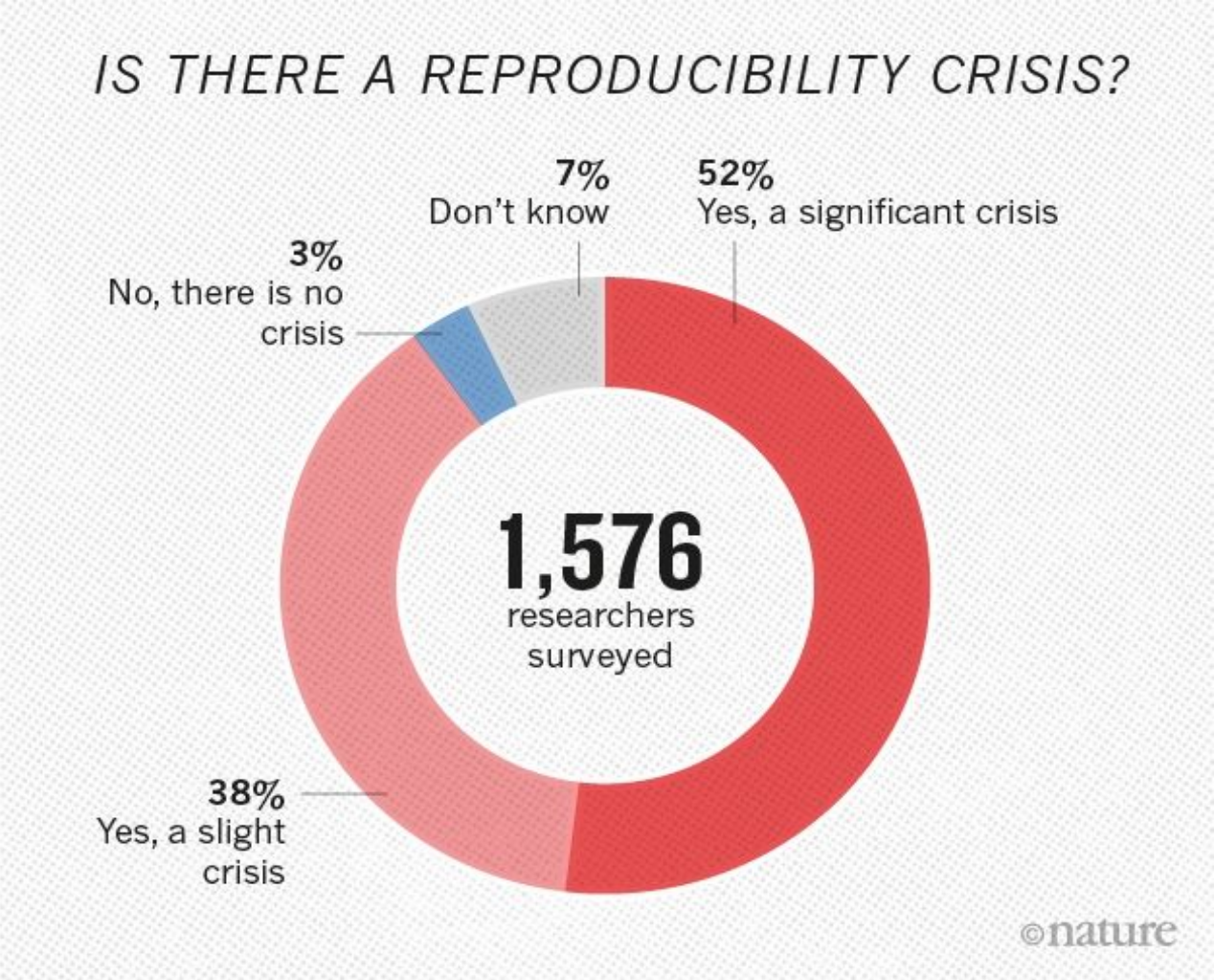
Snakemake for reproducible analyses

SIB Days 2022

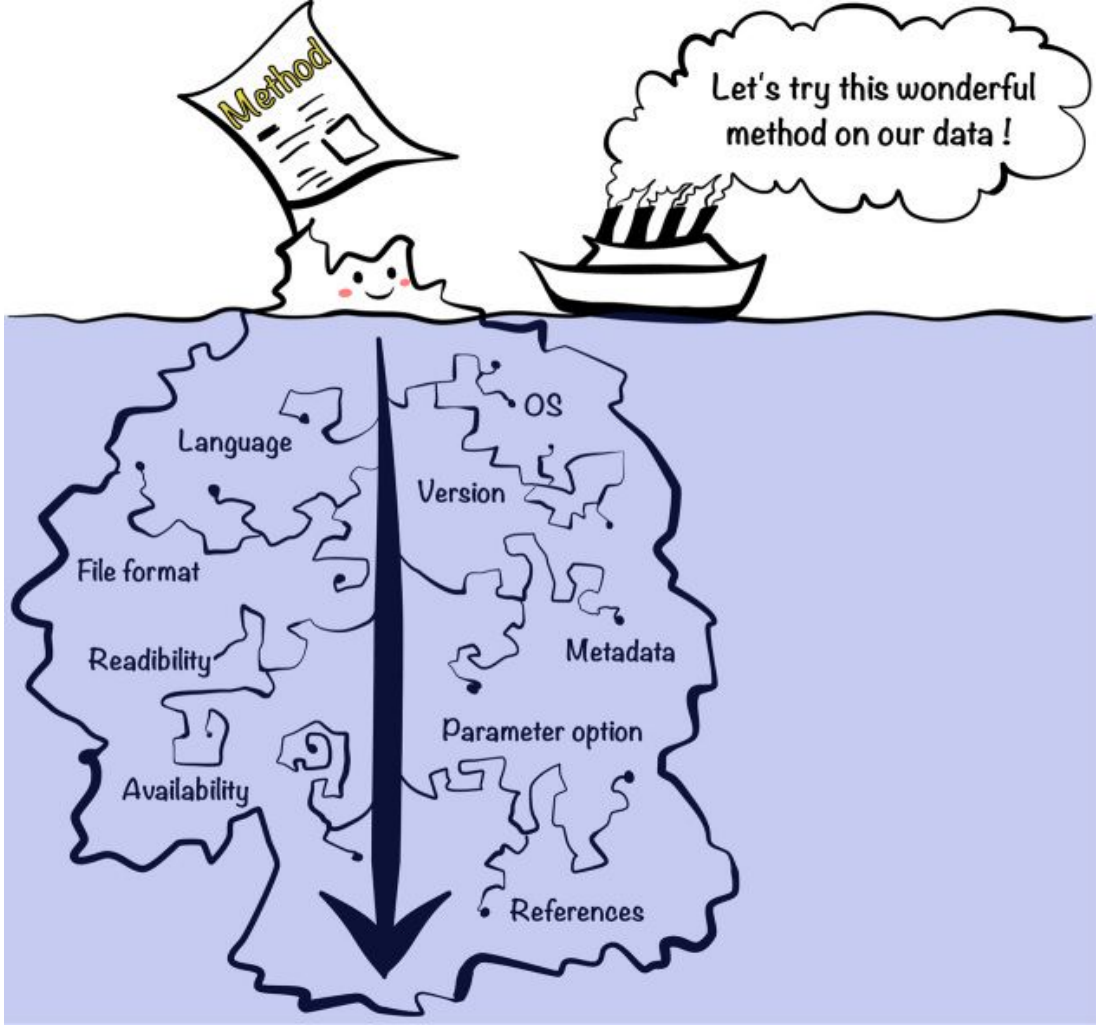
Romain Feron[1] & Antonin Thiébaut[1]

1: UNIL, DEE, SIB

romain.feron@unil.ch; antonin.thiebaut@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.

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 a Github or Gitlab repository)
  - Run Snakemake
  - Automatically reproduce all the results

Structure of this workshop

- Short lecture (~30 min), then focus on practice
- Exercises are loosely based on the official tutorial: genomics workflow
- You will implement exercises on your local computer
- All the information is on the workshop's wiki
- Reference implementation for all exercises is in the Github repository
- Questions are welcome anytime!

Basic concepts

Basic concepts

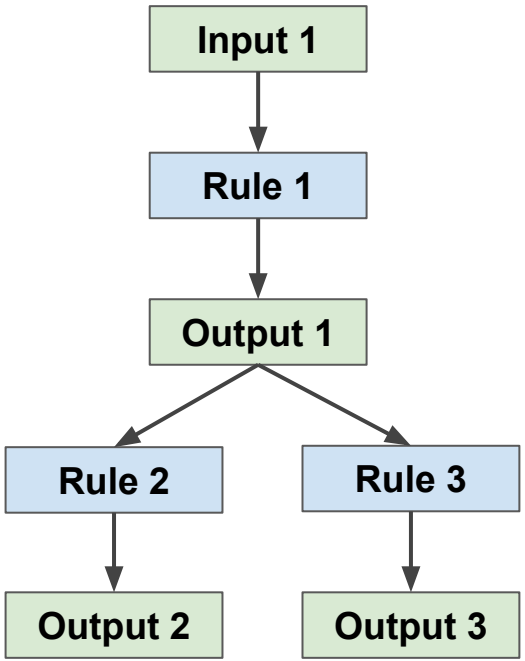
- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

Basic concepts

- Workflow structure
- Defining simple rules
- Workflow execution for simple rules
- Multiple inputs/outputs
- Rules dependencies
- Wildcards
- 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

Workflov

- Wor
- Sna

```
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
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

Defining simple rules

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

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

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

- 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

- 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'

Basic concepts

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

Executing workflows

Snakefile

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

Executing workflows

Snakefile

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

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

Executing workflows

Snakefile

```
rule first_step:
    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:
    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:
    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:
    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

Snakefile

```
rule first_step:
    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
```

Executing workflows

Snakefile

```
rule first_step:
    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
- 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:
    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:
    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:
    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

- 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:
    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}
        '''
```

Multiple inputs

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

```
rule first_step:
    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 need to be generated or the job will fail

```
rule first_step:
    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
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

Rules dependencies

```
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}'
```

Rules dependencies

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

```
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}'
```

Rules dependencies

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

```
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}'
```

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

Rules dependencies

```
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}'
```

```
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:
    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}'
```

```
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:
    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}'
```

```
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:
    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}'
```

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

Rules dependencies

```
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}'
```

- 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
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

Wildcards: Snakemake "variables"

“Hardcoded” input and output files

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

“General” input and output files with wildcards

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

Wildcards: inferred from output

```
rule first_step:
    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:
    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:
    input:
        'data/{sample}.tsv'
    output:
        'results/{sample}_{treatment}.txt'
    shell:
        'echo {wildcards.sample};'
        'cp {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:
    input:
        'data/{sample}_{treatment}.tsv'
    output:
        'results/{sample}_{treatment}.txt',
        'results/{sample}_info.txt'
    shell:
        'echo {wildcards.sample} > {output[0]};'
        'cat {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:
    input:
        'data/{sample}_{treatment}.tsv'
    output:
       'results/{sample}_{treatment}.txt'
    shell:
        'echo {wildcards.sample};'
        'cp {input} {output}'
```

Two different jobs cannot generate the same output

Basic concepts

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

Execution

- If no target is specified, 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
```

- If using this execution method, first rule **cannot have wildcards** (impossible to resolve)

```
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}'
```

Execution

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

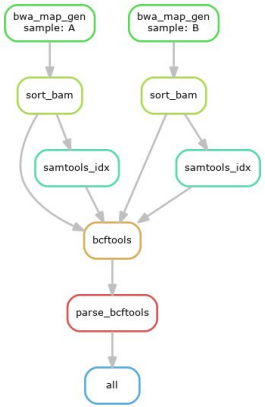
```
snakemake --cores 1 --force <target> / --forceall
```

- 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
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

The “expand” syntax

```
rule first_step:
    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

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

The “expand” syntax

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

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

The “expand” syntax

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

rule first_step:
    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:
    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']
replicates = (1, 2)

rule first_step:
    input:
        expand('data/{sample}_{replicate}.tsv', sample=samples, replicate=replicates)
    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
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

Non-file rule parameters

```
rule first_step:
    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:
    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:
    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:
    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
- Workflow execution
- The "expand" syntax
- Non-file rule parameters
- Executing Python / R code

The 'run' directive

```
rule first_step:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    params:
        lines = 5
    run:
        input_file = open(input[0])
        output_file = open(output[0], ‘w’)
        for i in range(params.lines):
            output_file.write(input_file.readline())
```

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

The 'run' directive

```
rule first_step:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    params:
        lines = 5
    run:
        input_file = open(input[0])
        output_file = open(output[0], 'w')
        for i in range(params.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:
    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:
    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)
```

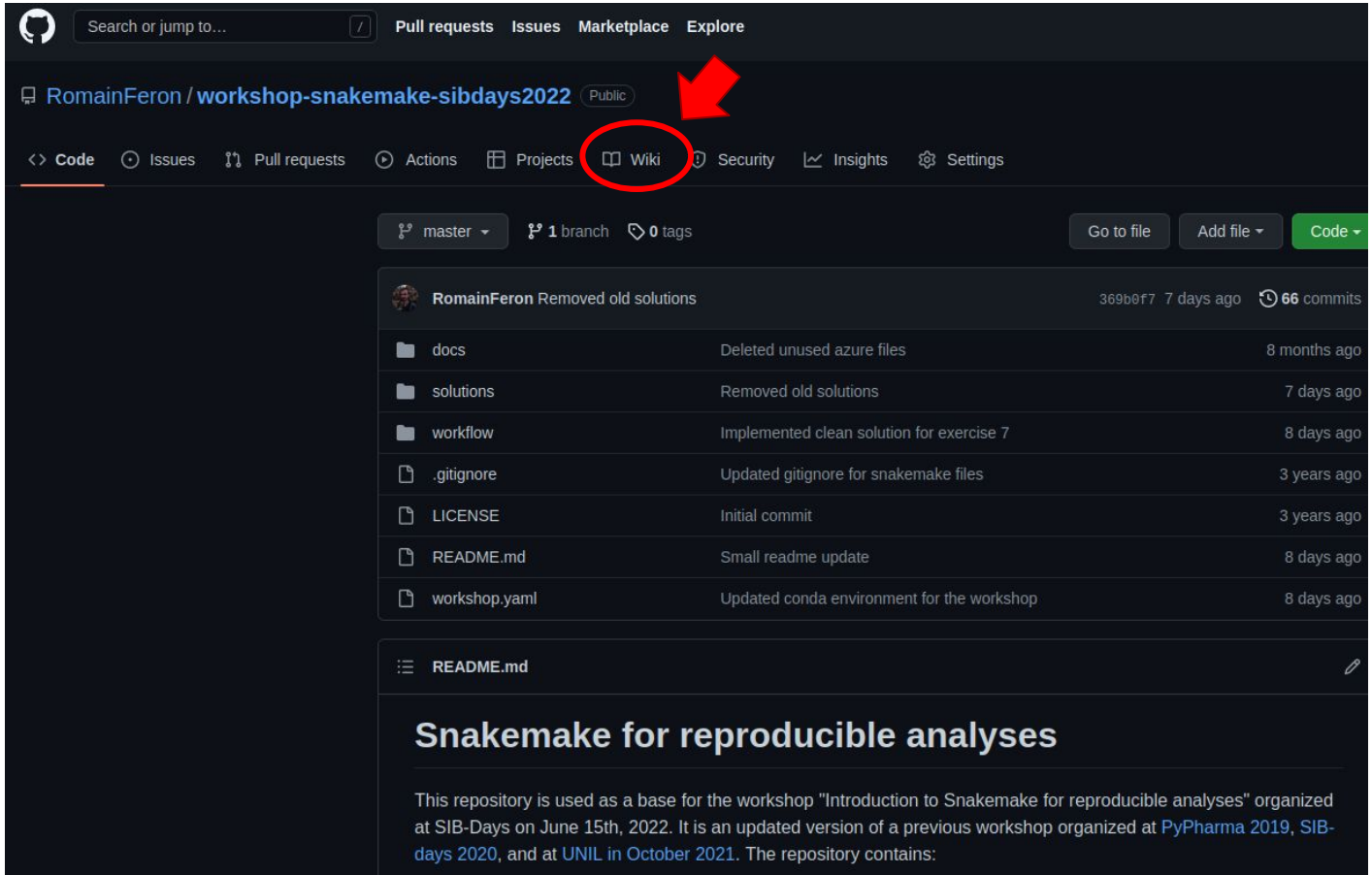
- Call an external Python script from Snakemake
- Directives and values can be accessed from a **snakemake** Python object
- Other supported languages:
  - R
  - Julia
  - Rust

Basic concepts

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

Hands on !
Exercises series 1

Hands on - Exercises



Hands on - Exercises series 1

RomainFeron / **workshop-snakemake-sibdays2022** Public

Pin

Code | Issues | Pull requests | Actions | Projects | **Wiki** | Security | Insights | Settings

Home

Edit | New Pa

Romain Feron edited this page 8 days ago · 13 revisions

Welcome to the official wiki for the workshop **Introduction to Snakemake for reproducible analyses** which is taking place at SIB-Days on June 15th 2022.

In this wiki, you will find all the information presented during the workshop, sometimes with additional details and references to the official Snakemake's documentation. You will also find detailed information about the exercises for each section, as well as hints for the difficult parts.

All information in the current version of this wiki is based on the official documentation for Snakemake version `7.8.1`.

Direct links to exercises:

Exercises series

+ Add a custom footer

Pages 22

Exercises Series

Basic concepts

- Defining rules
- Rule dependencies
- Wildcards
- Executing workflows
- The expand syntax
- Non-file rule parameters
- Executing Python code

Advanced concepts

- Config files
- Functions as input and output for rules
- Advanced directives: threads, log, and

Advanced concepts

Advanced concepts

- Config files
- Advanced directives
- Functions as input and output for rules
- Modularization
- Automatic software deployment with Conda
- Workflow organization guidelines

Advanced concepts

- Config files
- Advanced directives
- Functions as input and output for rules
- Modularization
- Automatic software deployment with Conda
- Workflow organization guidelines

Specifying parameters with a config file

config.yaml

```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}'
```

Specifying parameters with a config file

- Snakemake has two ways to read information from a config file :
  - Specify the file at runtime with the execution parameter “--configfile”
  - Add a line “configfile: <filename>” at the top of Snakefile
- Config files are either YAML (preferred) or JSON files
- The config file is parsed into a ‘config’ dictionary that can be accessed inside rule definitions
- Lists of parameters become list (*e.g.* ‘samples’), lists of named parameters become dictionaries (*e.g.* ‘resources’)

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

Advanced concepts

- Config files
- Advanced directives
- Functions as input and output for rules
- Modularization
- Automatic software deployment with Conda
- 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. Syntax : “log : <path/to/log/file.log>”. The path can be accessed in “shell” with “{log}”

- Logs still need to be handled manually for each command, but now Snakemake automatically creates the directory in the log file path

```
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. Syntax : “log : <path/to/log/file.log>”. The path can be accessed in “shell” with “{log}”

- Logs still need to be handled manually for each command, but now Snakemake automatically creates the directory in the log file path

- Log files must have the **same wildcards** as the **output** !

- It’s best to regroup logs in a “logs” folder in your workflow

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

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

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 concepts

- Config files
- Advanced directives
- Functions as input and output for rules
- Modularization
- Automatic software deployment with Conda
- Workflow organization guidelines

Using functions as input / output for rules

```
def first_step_input(wildcards):
    sample = wildcards.sample
    if sample == 'sample1':
        return 'data/data1.txt'
    else:
        return 'data/data2.txt'


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

Using functions as input / output for rules

- 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. Can be a lambda expression
- Define function above the rule, then use syntax “input : <function_name>”
- Functions are evaluated before executing the workflow ⇒ can’t list output files !
- Functions can return a dictionary with input names as keys.
  Use “input : unpack(<function_name>) to obtain named inputs

Advanced concepts

- Config files
- Advanced directives
- Functions as input and output for rules
- Modularization
- Automatic software deployment with Conda
- Workflow organization guidelines

Modularization : subfiles

- Simplest way to organize workflow : group rules in different snakefiles and use “include” in main Snakefile. Syntax : “include: <path/to/snakefile.smk>”

first_step.smk

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

Snakefile

```
include: 'first_step.smk'

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

Modularization : subfiles

- Simplest way to organize workflow : group rules in different snakefiles and use “include” in main Snakefile. Syntax : “include: <path/to/snakefile.smk>”

- Default rule is not affected by includes (still first rule in Snakefile)

- If you placed included files in sub-directories, remember to change relative paths (*e.g.* for external script files)

Modularization : modules

- **Modules** : import rules explicitly from another Snakefile, this is similar to the "*from math import log, exp*" statement in python.

my_workflow/Snakefile

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

Snakefile

```
module other_workflow:
    Snakefile:
        'my_workflow/Snakefile'

use rule * from other_workflow

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

Advanced concepts

- Config files
- Advanced directives
- Functions as input and output for rules
- Modularization
- Automatic software deployment with Conda
- Workflow organization guidelines

Automatic deployment of software with Conda

- **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
- Environments can be defined in YAML files
- Great tool to manage software in general

workshop.yaml

```yaml
name: snakemake-workshop
channels:
  - conda-forge
  - bioconda
dependencies:
  - python=3.6.8
  - snakemake=6.9.1
  - graphviz=2.38.0
  - bcftools=1.9
  - samtools=1.9
  - bwa=0.7.17
```

Automatic deployment of software with Conda

- Snakemake provides Conda integration: automatically deploy a conda environment for a rule.
- Directive “conda”, value is the relative path to the environment file: <path/to/environment.yaml>”
- Execution parameter “--use-conda”

Automatic deployment of software with Conda

```
rule first_step:
    input:
        'results/genome.fa'
    output:
        'results/genome.fai'
    conda:
        'envs/indexing.yaml'
    shell:
        'samtools index {input}'
```

name: indexing
channels:
- conda-forge
- bioconda

dependencies:
- samtools=1.13

```
workflow_dir/
  ├── results/
  │    └── genome.fa
  ├── envs/
  │    └── indexing.yaml
  └── Snakefile
```

```
snakemake --cores 1 --use-conda results/genome.fai
```

Advanced concepts

- Config files
- Advanced directives
- Functions as input and output for rules
- Modularization
- Automatic software deployment with Conda
- Workflow organization guidelines

How to organize your workflow : best practices

- A 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
- If you have many samples with information, use a sample sheet located in the ‘config’ folder

```
.
├── config
│   ├── config.yaml
│   └── samples.tsv
├── LICENSE
├── README.md
├── resources
│   └── calling_model.txt
└── workflow
    ├── envs
    │   ├── first_task.yaml
    │   └── second_task.yaml
    ├── rules
    │   ├── first_task.smk
    │   └── second_task.smk
    ├── scripts
    │   └── do_something.py
    └── Snakefile
```

Configuration files (modified by the user)

External files provided with the workflow

Conda environment files

Snakemake modules (rules)

Python / R scripts

Actual workflow implementation

```
.
├── config
│   ├── config.yaml
│   └── samples.tsv
├── data
│   ├── genome.fa
│   ├── sample_2.fq.gz
│   └── sample_1.fq.gz
├── LICENSE
├── README.md
├── resources
│   └── calling_model.txt
└── workflow
    ├── envs
    │   ├── first_task.yaml
    │   └── second_task.yaml
    ├── rules
    │   ├── first_task.smk
    │   └── second_task.smk
    ├── scripts
    │   └── do_something.py
    └── Snakefile
```

```
.
├── config
│   ├── config.yaml
│   └── samples.tsv
├── data
│   ├── genome.fa
│   ├── sample_2.fq.gz
│   └── sample_1.fq.gz
├── LICENSE
├── README.md
├── resources
│   └── calling_model.txt
└── workflow
    ├── envs
    │   ├── first_task.yaml
    │   └── second_task.yaml
    ├── rules
    │   ├── first_task.smk
    │   └── second_task.smk
    ├── scripts
    │   └── do_something.py
    └── Snakefile
```

Execution

```
.
├── 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
├── resources
│   └── calling_model.txt
├── results
│   ├── sample_1.bam
│   ├── sample_2.bam
│   └── variants.vcf
└── workflow
    ├── envs
    │   ├── first_task.yaml
    │   └── second_task.yaml
    ├── rules
    │   ├── first_task.smk
    │   └── second_task.smk
    ├── scripts
    │   └── do_something.py
    └── Snakefile
```

Benchmarks

Log files

Final results files

Hands on - Exercises series 2

Projects | Wiki | Security | Insights | Settings

Home

Edit | New Page

Romain Feron edited this page 19 hours ago · 7 revisions

Welcome to the official wiki for the virtual workshop **Snakemake for reproducible analyses** which is taking place at UNIL on October 14th 2021.

In this wiki, you will find all the information presented during the workshop, sometimes with additional details and references to the official Snakemake's documentation. You will also find detailed information about the exercises for each section, as well as hints for the difficult parts.

All information in the current version of this wiki is based on the official documentation for Snakemake version `6.9.1`.

Direct links to exercises:

- Guidelines
- Series 1
- Series 2

+ Add a custom footer

Pages 23

Exercises: General guidelines

Basic concepts

- Defining rules
- Rule dependencies
- Wildcards
- Executing workflows
- The expand syntax
- Non-file rule parameters
- Executing Python code

Exercises Series 1

Advanced concepts

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

Working with remote inputs

- Snakemake implements remote file access for many protocols
- Idea :
  - Import module for the remote access protocol
  - Initiate remote provider instance in the snakefile’s body
  - Access remote files within a rule

Working with remote inputs

- Snakemake implements remote file access for many protocols
- Idea :
  - Import module for the remote access protocol
  - Initiate remote provider instance in the snakefile's body
  - Access remote files within a rule
- Files are downloaded to a sub-dir of the current working directory
- List of available remote protocols :

- Amazon Storage Service (AWS S3)
- Google Cloud Storage (GS)
- Microsoft Azure Storage
- SFTP
- HTTP(S)
- FTP
- Dropbox
- GenBank / NCBI Entrez
- XRootD
- WebDAV
- GFAL
- GridFTP
- iRODS
- EGA

Running snakemake on clusters and cloud

- Built-in support for Kubernetes / Google cloud and AWS (check doc)
- Snakemake can make use of a scheduler (slurm, SGE, LFS …) to execute jobs on a cluster without changes to the rules (almost true)
- Syntax : “snakemake --cluster <submit_command>” (qsub, sbatch …)
- Advanced syntax : command can take job information from rule definition

```
snakemake --jobs 12 --cluster “sbatch --cpus-per-task={threads}”
```

- Specify the maximum number of jobs to submit with **“-j / --jobs”**

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

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
  - Beautiful DAG in one command, no more powerpoint !