SNAKEMAKE

A framework for reproducible data analysis

WHAT IS SNAKEMAKE?

A rule based pipeline generator
Documentation of your processing steps
A scheduler for tasks
A utility resolve program dependencies
A utility for reproducibility and portability
A way of effectively using computing clusters

HOW DOES IT WORK?

RTFM!

Really - there is too much functionality to put it in one presentation and **the documentation** is very good!

HOW DOES IT WORK?

One defines rules which describe a specific processing step in the pipeline
The rules are combined into a snakefile which is called by snakemake
From the input and outputs of each rule a DAG is reconstructed
The DAG is then used to compute each rule in the right order

RULE

Rules are defined in yaml style

It can define different settings

- Name of the rule
- Inputs
- Outputs
- Processing
- Logfiles
- Additional parameters
- Resource allocations
- Specific environments to use

SNAKEFILE

The rules are combined to one snakefile - the complete description of the pipeline

At the same time, sanity checks are done:

- All initial input files available?
- Are unavailable inputs defined as outputs of other rules?
- What files have already been produced?

OTHER NOTABLE FEATURES

Snakemake is python based: everywhere within the pipeline one can run python commands
The use of **wildcards** makes rules highly reusable and expandable
Integration of *R*, *Python*, *RMarkdown*, *Jupyter*, *Julia* scripts into the pipeline
Config files can be used to customize the execution of snakemake on several levels

THE RULE

```
rule NAME:
    input: "path/to/inputfile"
    output: "path/to/outputfile"
    shell: "somecommand {input} {output}"
```

THE SNAKEFILE

Its "simply" a bunch of rules pasted together.

```
rule all:
    input: "mapped.bam"

rule trim:
    input:
        "raw.read_fwd.fastq",
        "raw.read_rev.fastq",
        "adapter.fa"
    output:
        "trimmed.read_fwd.fastq",
        "trimmed.read_fwd_UP.fastq",
        "trimmed.read_rev.fastq",
        "trimmed.read_rev_UP.fastq"
    shell:
        """trimmmatic PE {input} {output} \
        ILLUMINACLIP:{input.adapter}:2:30:10 \
        LEADING:3 \
        TRAILING:3 \
```

```
19          SLIDINGWINDOW:4:20 MINLEN:36 \
20          -threads 1"""
21
22  rule map:
23      input:
24          fwd = "trimmed.read_fwd.fastq",
25          rev = "trimmed.read_rev.fastq",
26      output:
27          "mapped.bam"
28      shell:
29          """bowtie2 -x reference -1 {input.fwd} \
30          -2 {input.rev} -p 1 | \
31          samtools view -Sb > {output}"""
```

The first rule defines **target files** for the workflow

RUN SNAKEMAKE

Have all necessary input files correctly linked in the snakefile
Have the snakefile in the current directory (or in a subdirectroy workflow) and call:

```
snakemake --cores 2
```

- -n will run it in dry mode (sanity checks and counting of jobs only)
- -q will reduce the verbosity

WILDCARDS

```
rule map:
    input:
        fwd = "trimmed/trimmed.{sample}_fwd.fastq",
        rev = "trimmed/trimmed.{sample}_rev.fastq",
    output:
        temp("results/{sample}.bam")
    shell:
        """bowtie2 -x reference -1 {input.fwd} \
        -2 {input.rev} -p 1 | \
        samtools view -Sb > {output}"""
```

Wildcards can be used for pattern matching in file names

Snakemake will defines wildcards by target files of the workflow - using `.*` extension if not properly defined elswere

EXPAND

Make use of wildcards in the first rule for definition of what files to include in the workflow

```
1 rule all:
2     input:
3         expand("results/{samples}.bam",
4         samples = ["sample1","sample2","sample3"])
```

`samples` can be an arbitrary python list

USING PYTHON

It would be tedious to define the list of samples in every snakemake project - use python to read csv-files or similar

```
1 import pandas as pd
2 meta = pd.read.csv("resources/meta.csv")
3 samples = meta.ID.to_list()
```

Python can be used anywhere in the snakemake file

INCLUDE EXTERNAL CODE

Snakemake can **include external code** - may it be python or rules using the include command

```
include: "rules/tximport.smk"
# NOTE: the path is relative to the snakefile
```

EXECUTION OF CUSTOM CODE

The `shell:` statment is for running programs available in your $PATH
Use `run:` to execute python code directly
Use `script:` to execute a script and parse rule variables
Use `wrapper:` to execute pedefined commands

EXAMPLE FOR `run:`

```
rule NAME:
    input: "path/to/inputfile", "path/to/other/inputfile"
    output: "path/to/outputfile", somename = "path/to/another/outputfile"
    run:
        for f in input:
            ...
            with open(output[0], "w") as out:
                out.write(...)
        with open(output.somename, "w") as out:
            out.write(...)
```

USING SCRIPTS

```
input:
    in = "infile"
output: "outfile"
script: "path/to/script" # relative to file of the rule
```

Arguments are parsed to **language specific objects**:

- **R**: `snakemake@input[["in"]]` or `snakemake@input[[1]]`
- **Python**: `snakemake.input["in"]` or `snakemake.input[0]`
- **Julia**: `snakemake.input["in"]` or `snakemake.input[1]` `(???)`

USING WRAPPERS

```
input: "mapping.bam"
output: "mapping_sort.bam"
threads: 2
params: "-m 4G"
wrapper:  ""0.2.0/bio/samtools/sort"
```

Wrappers are predefined and provided as a repository at **github: snakemake/snakemake-wrappers/tree/master**

OTHER DECLARATIONS IN RULES

`log:` - used to define **log files**

`conda:` - defines yaml/json file with **conda dependencies** needed to execute rule

`threads:` - defines the maximum **number of cores** for the rule (default = 1) - scaled down if needed

`params:` - defines **additional parameters** which should be parsed to the rule execution

`resources:` - defines **allocation of resources** for the rule - handy for cluster execution

FILE MARKUPS

Output files can be marked for specific handling

`temp()` - marks temporary files, which are deleted after dependending jobs have finished

`protected()` - marks write protected files, usually for long lasting tasks

`directory()` - marks directories as output

`report()` - marks files to be integrated in the report

USEFUL TAGS FOR SNAKEMAKE EXECUTION

`--dryrun | -n` - do sanity checks and report job counts without executing the rules

`--cores | -N` - how many cores are provided to snakemake

`--use-conda` - tells snakemake to download and use conda envrionments

USEFUL TAGS FOR SNAKEMAKE EXECUTION

`--report` - generates a self contained html reporting the current state of the workflow

`--profile` - specifies which profile (predefined options to execute)

`--cluster` - tells snakemake to switch to cluster execution mode

BEST PRACTICSES

Run `snakemake --lint` befor publishing **(here)**
Follow this **directory tree**:

```
.
├── config           # config files
├── logs             # logfiles for each job here
├── resources        # files to start the workflow with
├── results          # resulting data files
|   ├── reports      # reports here
|   └── plots        # plots here
└── workflow         # files to define the workflow
    ├── envs         # files for conda environments
    ├── notebooks    # files for Rmarkdown/Jupyter
    ├── report       # rst files for descriptions in
reports
    ├── rules        # files for rules
    └── scripts      # files for R/Python/Julia
```

EXERCISES

1. Find the snakemake file in the tutorial_snakemake directory. Open it in an editor and try to understand what is being done. Investigate the inputs!
2. Run the pipeline! That might take several minutes.
3. Generate a report of the current pipeline!
4. Write your own rule: plot a histogram of only the tumor samples.
5. Generate a new report and include the plot!
6. Write your own rule: generate a fastqc report for each sample - before and after trimming!