Introduction to Snakemake

DAVID KOPPSTEIN

Before we get started

Check out the repository:

```
git clone https://github.com/dkoppstein/ngsschool-snakemake-tutorial
```

Install Miniconda3 and Snakemake according to the instructions, e.g. on Linux:

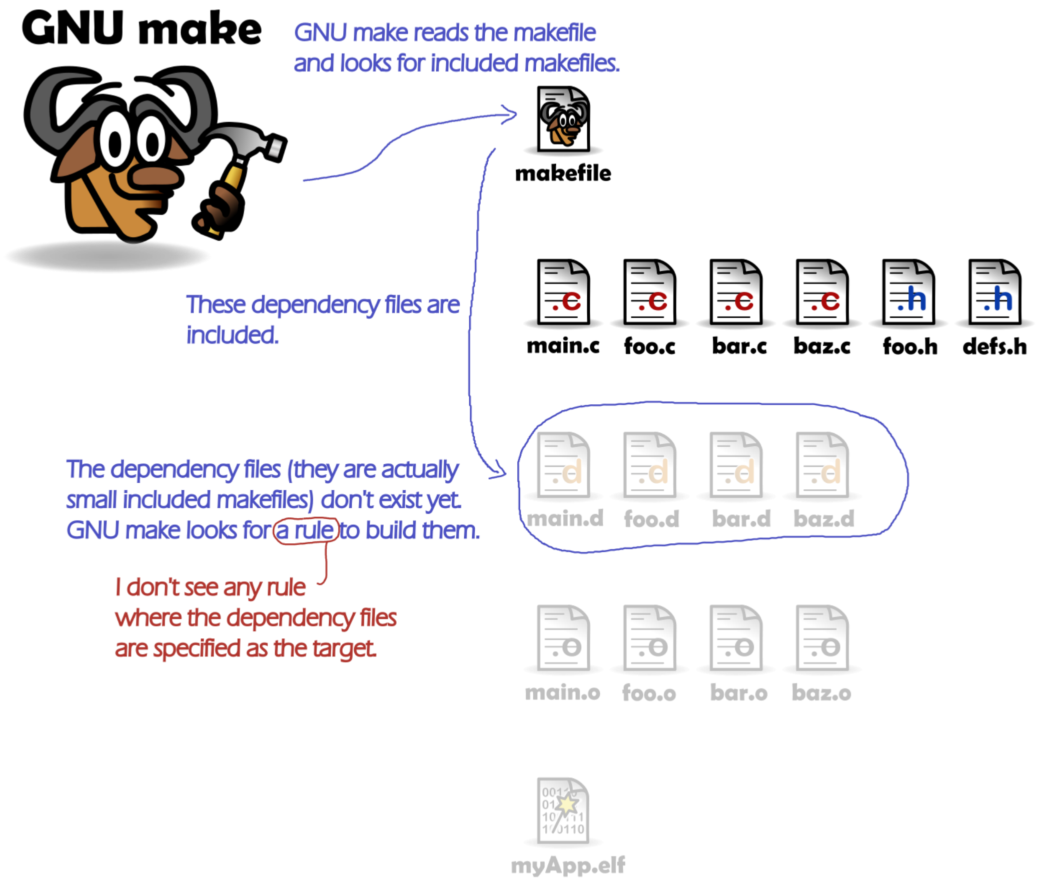
```
curl -O https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh

sh Miniconda3-latest-Linux-x86_64.sh

conda config --add channels defaults

conda config --add channels bioconda

conda config --add channels conda-forge

conda create –n ngstutorial –c bioconda snakemake
```

Follow along by looking at/executing the Snakefiles in green: e.g. exercises/exercise1/Snakefile

What is Snakemake?

- Software to create **reproducible** and **scalable** workflows
- Large and **active bioinformatics community** (3 citations per day!) due to **ease of use** and **interoperability**
- Easy to prototype workflows due to **embedded Python scripting**
- **Feature-rich and configurable**: Over 80 different command-line options and many ways to configure rules
- Designed by **Dr. Johannes Köster** (also lead developer of Bioconda)

Inspired by GNU Make


GNU make
GNU make reads the makefile and looks for included makefiles.
makefile
These dependency files are included.
main.c
foo.c
bar.c
baz.c
foo.h
defs.h
The dependency files (they are actually small included makefiles) don't exist yet. GNU make looks for a rule to build them.
main.d
foo.d
bar.d
baz.d
I don't see any rule where the dependency files are specified as the target.
main.o
foo.o
bar.o
baz.o
myApp.elf

Inspired by GNU Make

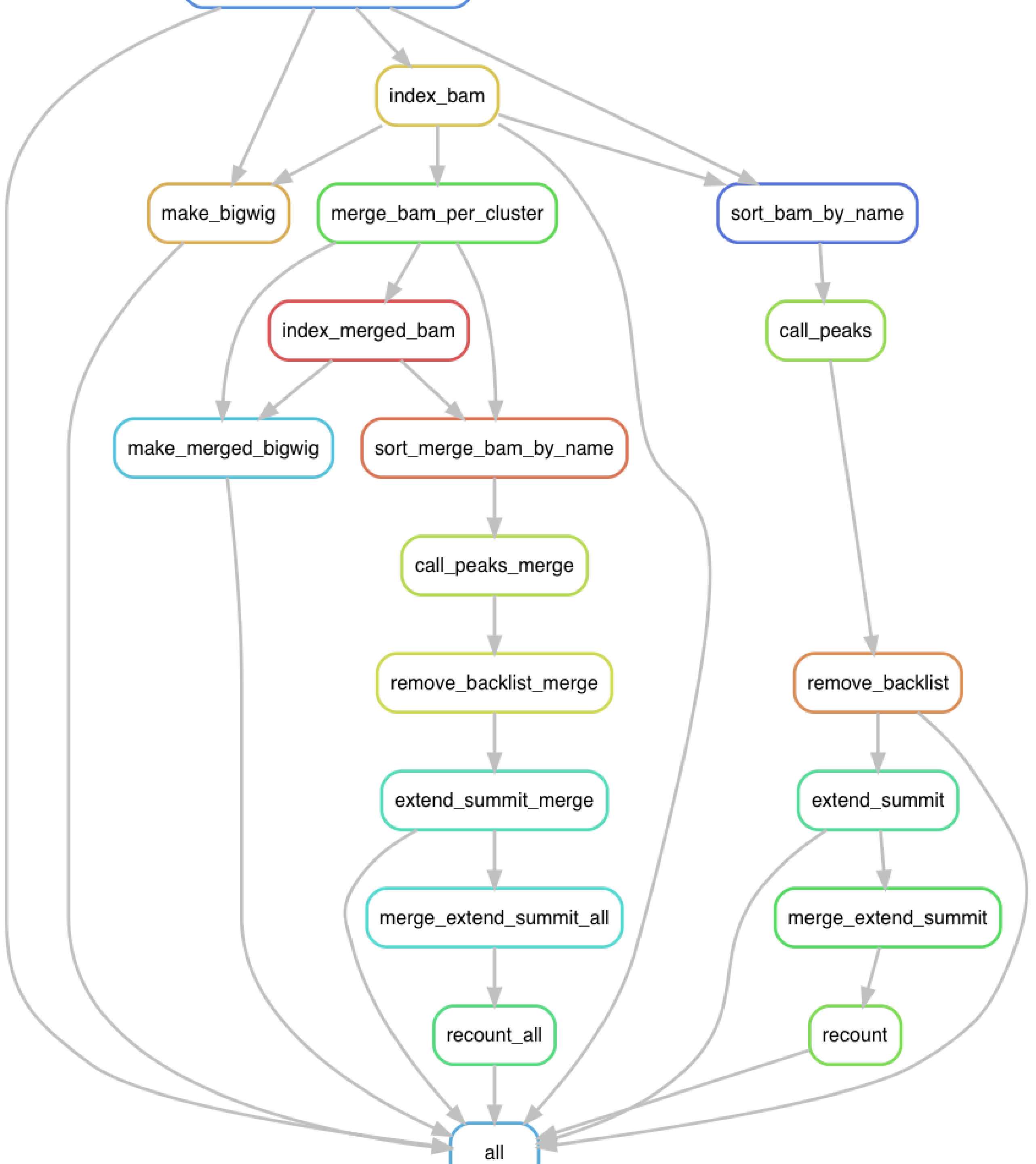
GNU Make uses a declarative **domain-specific language (DSL)**

Create some output file… …from some input files…

```
hellomake: hellomake.c hellofunc.c
	gcc -o $@ $< -I.
```

…using the following shell command

($@ and $< mean „output“ and „input“ files respectively)

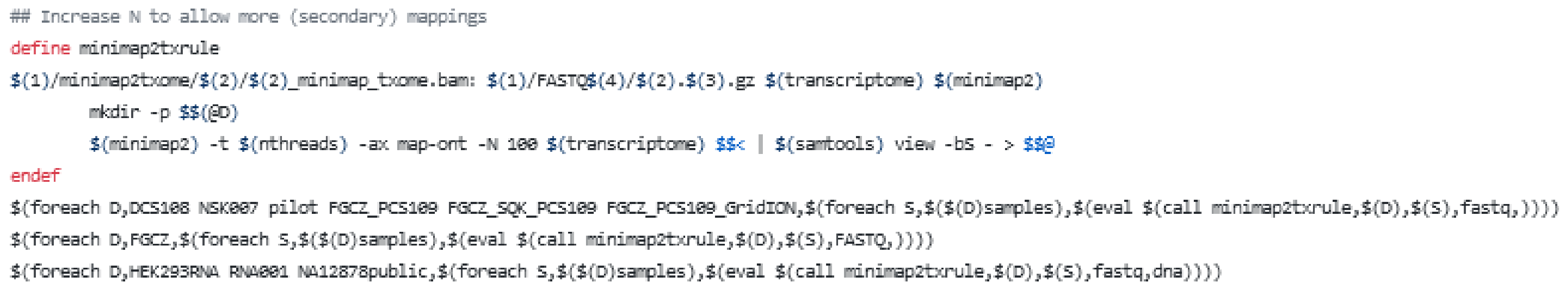


From a set of rules, one can generate complex yet reproducible workflows

Problems with Make

GNU Make

- **Not a full-fledged programming language**: Only simple functions available for storing variables, writing for loops, etc.
- Difficult to read – verbose
- Cryptic debugging messages
- Limited support for cluster and cloud execution



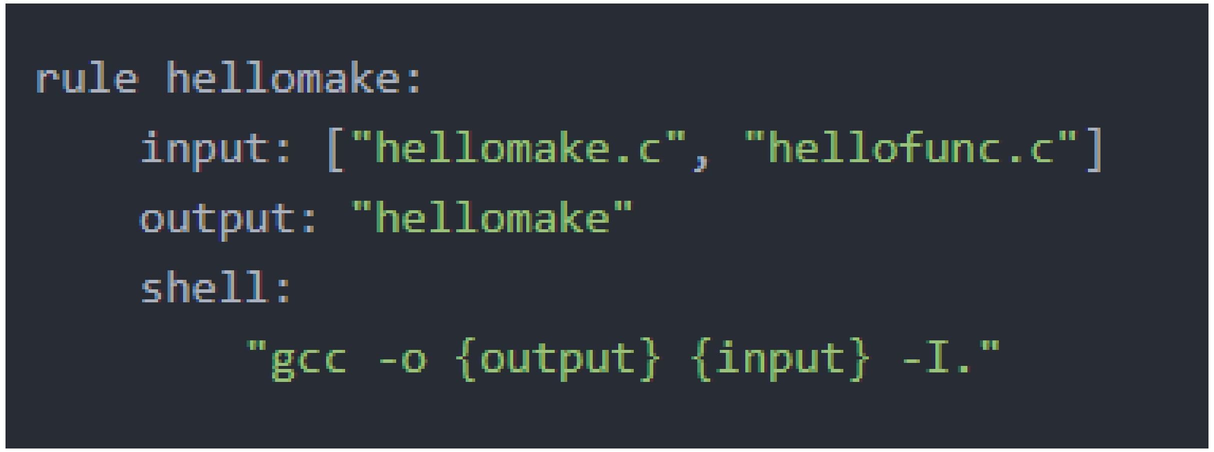
```
## Increase N to allow more (secondary) mappings
define minimap2txrule
$(1)/minimap2txome/$(2)/$(2)_minimap_txome.bam: $(1)/FASTQ$(4)/$(2).$(3).gz $(transcriptome) $(minimap2)
	mkdir -p $$(@D)
	$(minimap2) -t $(nthreads) -ax map-ont -N 100 $(transcriptome) $$< | $(samtools) view -bS - > $$@
endef
$(foreach D,DCS108 NSK007 pilot FGCZ_PCS109 FGCZ_SQK_PCS109 FGCZ_PCS109_GridION,$(foreach S,$($(D)samples),$(eval $(call minimap2txrule,$(D),$(S),fastq,))))
$(foreach D,FGCZ,$(foreach S,$($(D)samples),$(eval $(call minimap2txrule,$(D),$(S),FASTQ,))))
$(foreach D,HEK293RNA RNA001 NA12878public,$(foreach S,$($(D)samples),$(eval $(call minimap2txrule,$(D),$(S),fastq,dna))))
```

https://github.com/csoneson/NativeRNAseqComplexTranscriptome/blob/master/Makefile

Enter Snakemake

…from some input files…

Create some output file…



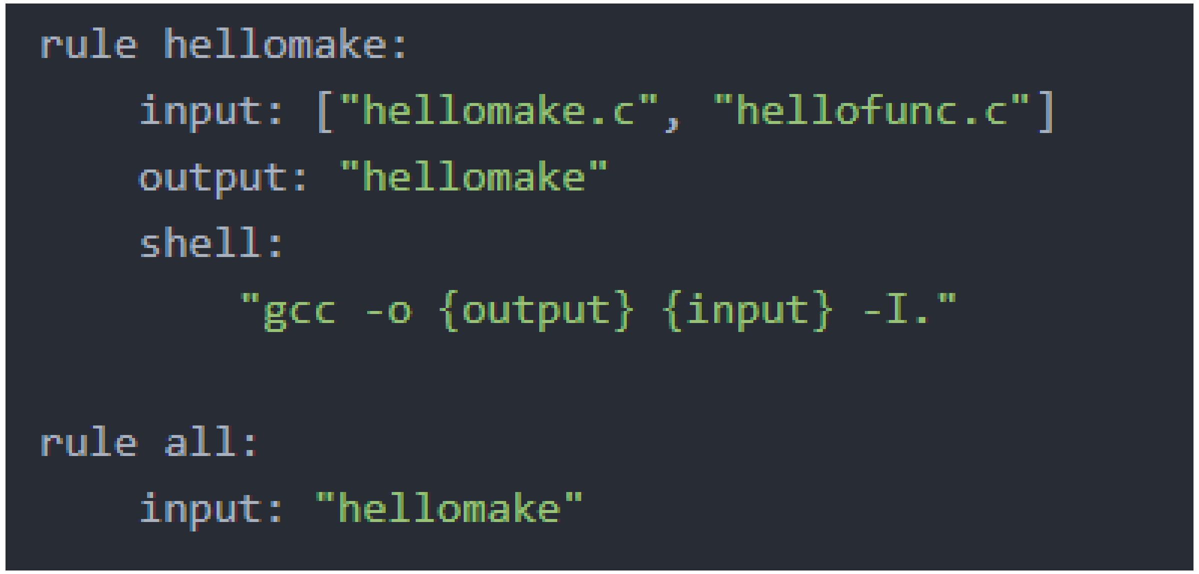
```
rule hellomake:
    input: ["hellomake.c", "hellofunc.c"]
    output: "hellomake"
    shell:
        "gcc -o {output} {input} -I."
```

…using the following shell command

({input} and {output} are wildcards for the input and output files, and are parsed with Python string formatting rules)

Declarative language

Tell Snakemake what to create,
and it will find a way to make it for you!



```
rule hellomake:
    input: ["hellomake.c", "hellofunc.c"]
    output: "hellomake"
    shell:
        "gcc -o {output} {input} -I."

rule all:
    input: "hellomake"
```

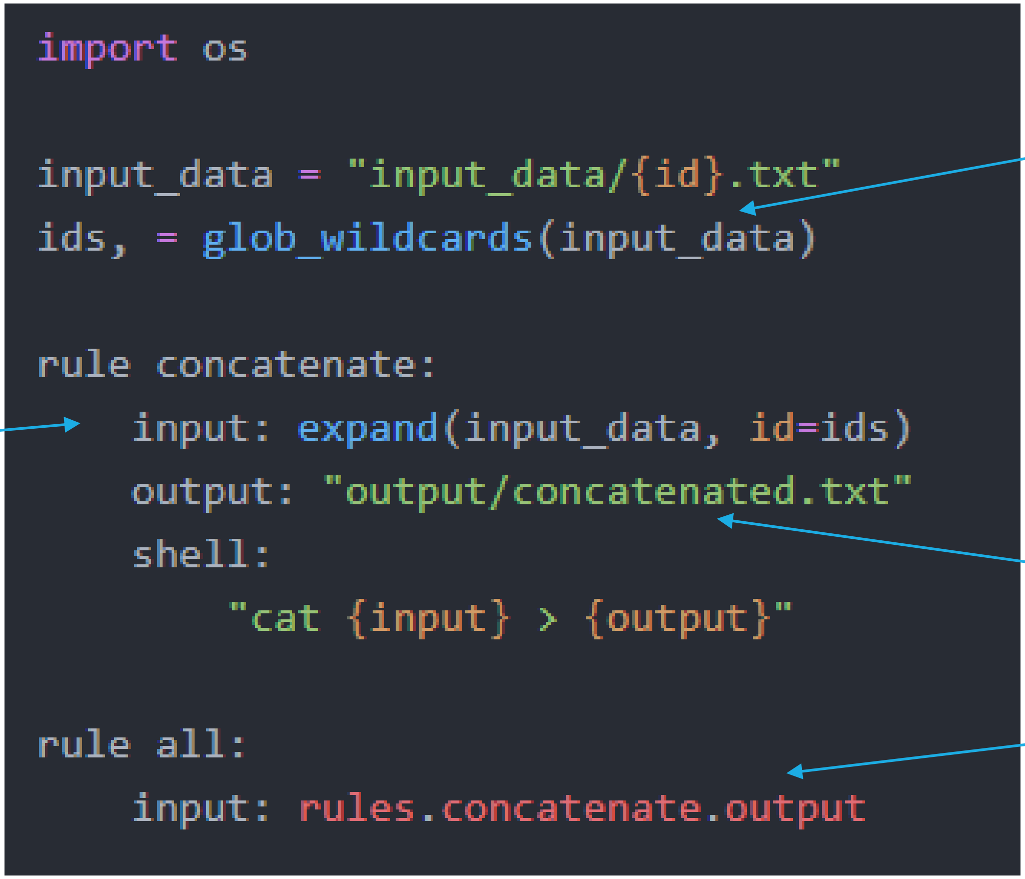
On-the-fly coding in Python

exercises/exercise1/Snakefile

Outside of rules, we can use arbitrary Python!

expand: a special Snakemake function to create a list of strings from a pattern

(returns [„input_data/file1.txt“, „input_data/file2.txt“])



```
import os

input_data = "input_data/{id}.txt"
ids, = glob_wildcards(input_data)

rule concatenate:
    input: expand(input_data, id=ids)
    output: "output/concatenated.txt"
    shell:
        "cat {input} > {output}"

rule all:
    input: rules.concatenate.output
```

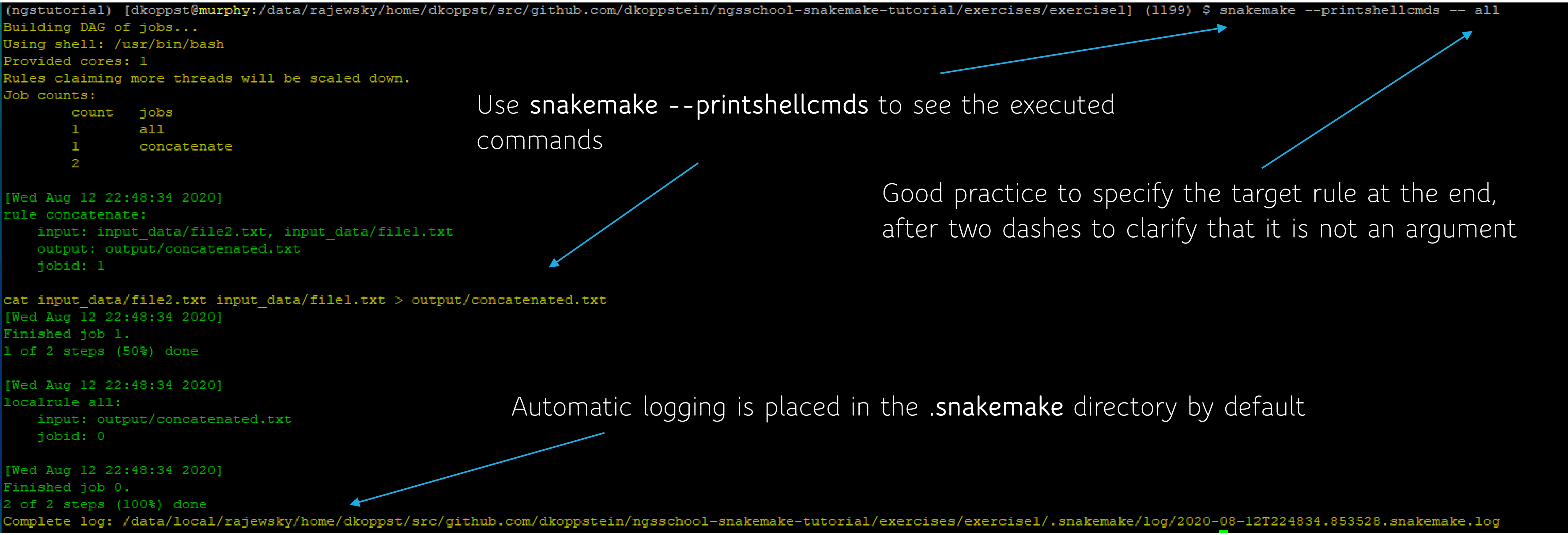
glob_wildcards: a special Snakemake function to look for all files matching a certain pattern (returns [„file1“, „file2“])

Input and output files are just strings, and can be manipulated as such.

You can also access the output of previous rules programmatically.

Example execution of a Snakefile

snakemake --printshellcmds -- all



```
(ngstutorial) [dkoppst@murphy:/data/rajewsky/home/dkoppst/src/github.com/dkoppstein/ngsschool-snakemake-tutorial/exercises/exercise1] (1199) $ snakemake --printshellcmds -- all
Building DAG of jobs...
Using shell: /usr/bin/bash
Provided cores: 1
Rules claiming more threads will be scaled down.
Job counts:
        count   jobs
        1       all
        1       concatenate
        2

[Wed Aug 12 22:48:34 2020]
rule concatenate:
    input: input_data/file2.txt, input_data/file1.txt
    output: output/concatenated.txt
    jobid: 1

cat input_data/file2.txt input_data/file1.txt > output/concatenated.txt
[Wed Aug 12 22:48:34 2020]
Finished job 1.
1 of 2 steps (50%) done

[Wed Aug 12 22:48:34 2020]
localrule all:
    input: output/concatenated.txt
    jobid: 0

[Wed Aug 12 22:48:34 2020]
Finished job 0.
2 of 2 steps (100%) done
Complete log: /data/local/rajewsky/home/dkoppst/src/github.com/dkoppstein/ngsschool-snakemake-tutorial/exercises/exercise1/.snakemake/log/2020-08-12T224834.853528.snakemake.log
```

Use **snakemake --printshellcmds** to see the executed commands

Good practice to specify the target rule at the end, after two dashes to clarify that it is not an argument

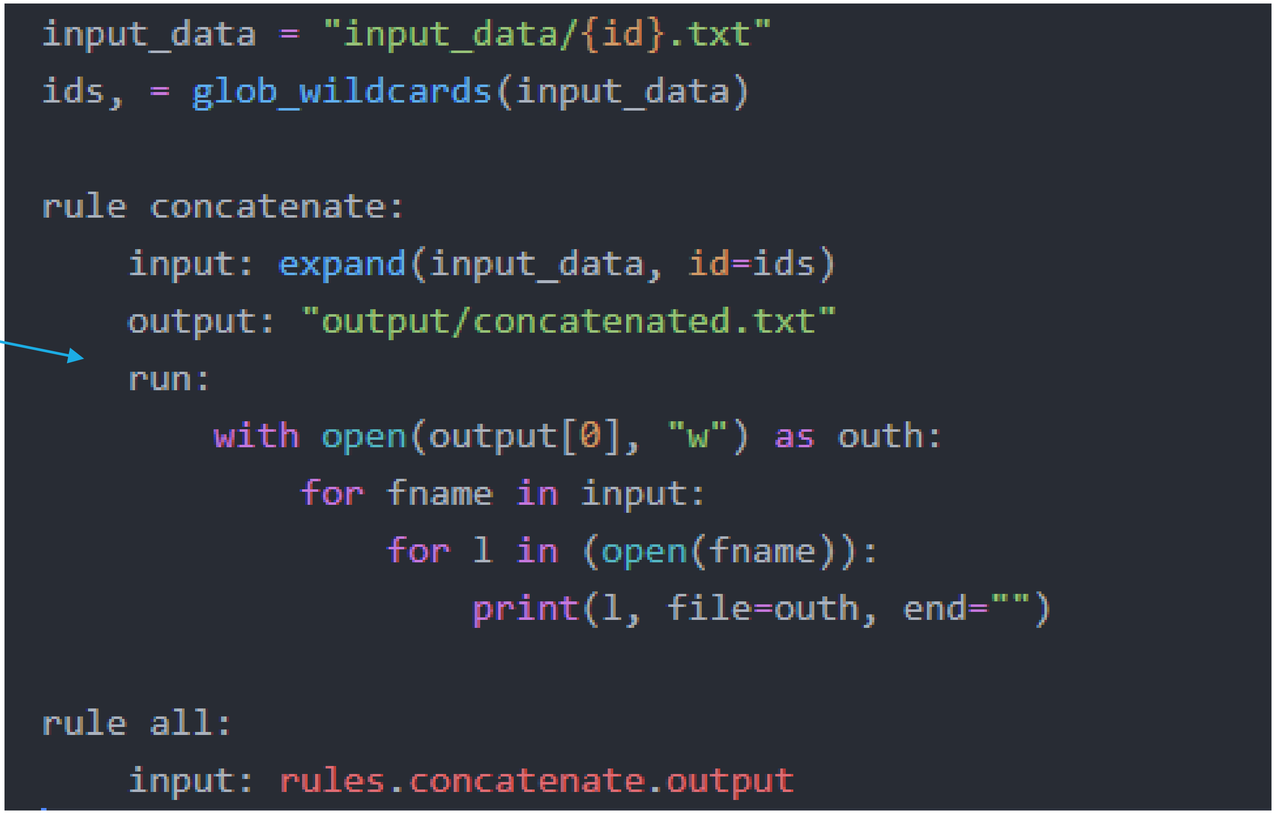
Automatic logging is placed in the **.snakemake** directory by default

On-the-fly coding in Python in rules

exercises/exercise2/Snakefile

We can use the „**run**“ keyword to drop into Python within the rule itself!

Here, the special variables „**output**“ and „**input**“ are passed to the Python script by Snakemake as a list of strings.



```
input_data = "input_data/{id}.txt"
ids, = glob_wildcards(input_data)

rule concatenate:
    input: expand(input_data, id=ids)
    output: "output/concatenated.txt"
    run:
        with open(output[0], "w") as outh:
            for fname in input:
                for l in (open(fname)):
                    print(l, file=outh, end="")

rule all:
    input: rules.concatenate.output
```

Parallel processing with wildcards

What if we want to process many files in parallel, rather than concatenating them all at once?

Then, creating a generic rule for processing them, and combining them at the end, is ideal.

Parallel processing with wildcards

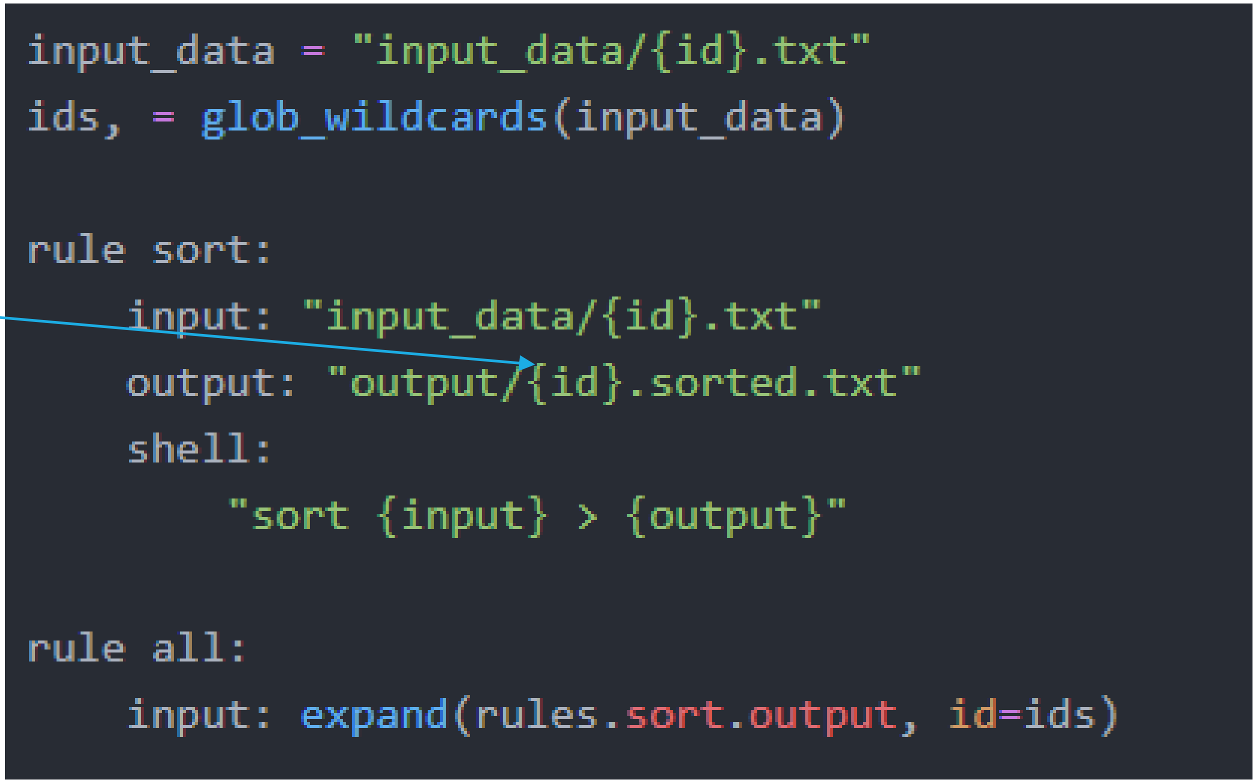
exercises/exercise3/Snakefile

Here, the {id} wildcard matches any string that contains .

Wildcards must be consistent between input and output files.

Advanced tip: we can also restrict the wildcards using regular expressions:
For example, we can use {id,[A-Za-z0-9]+}.txt to constrain the ID to alphanumeric characters.

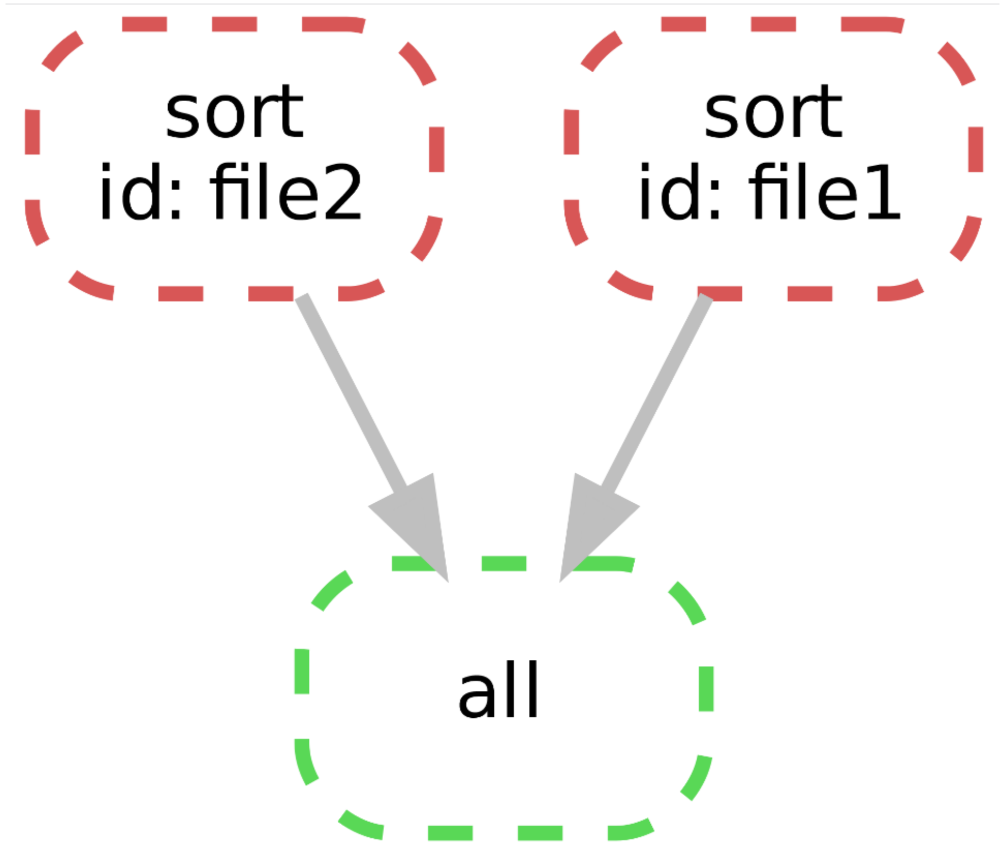
We can also use the keyword **wildcard_constraints** either per-rule or globally.



```
input_data = "input_data/{id}.txt"
ids, = glob_wildcards(input_data)

rule sort:
    input: "input_data/{id}.txt"
    output: "output/{id}.sorted.txt"
    shell:
        "sort {input} > {output}"

rule all:
    input: expand(rules.sort.output, id=ids)
```

Visualizing the workflow

```
snakemake --dag -- all | dot -Tpdf > dag.pdf
```



Parametrizing rules

exercises/exercise4/Snakefile

Often in bioinformatics, one would like to run a generic rule differently based on the type of sample being used

Or, you might want to run the same rule several times, using different sets of parameters.

Here, we discuss two new ideas:

1. Specifying input files using **input functions**
2. Specifying parameters using the **params** directive

GOAL 1: Map file1 to human index, file2 to mouse index

GOAL 2: Subsample 1e3 or 1e4 reads for both samples

Use **input functions** to determine the **input** of a rule based on the matching **wildcard of the output**
Input functions take a single variable: the Wildcards object

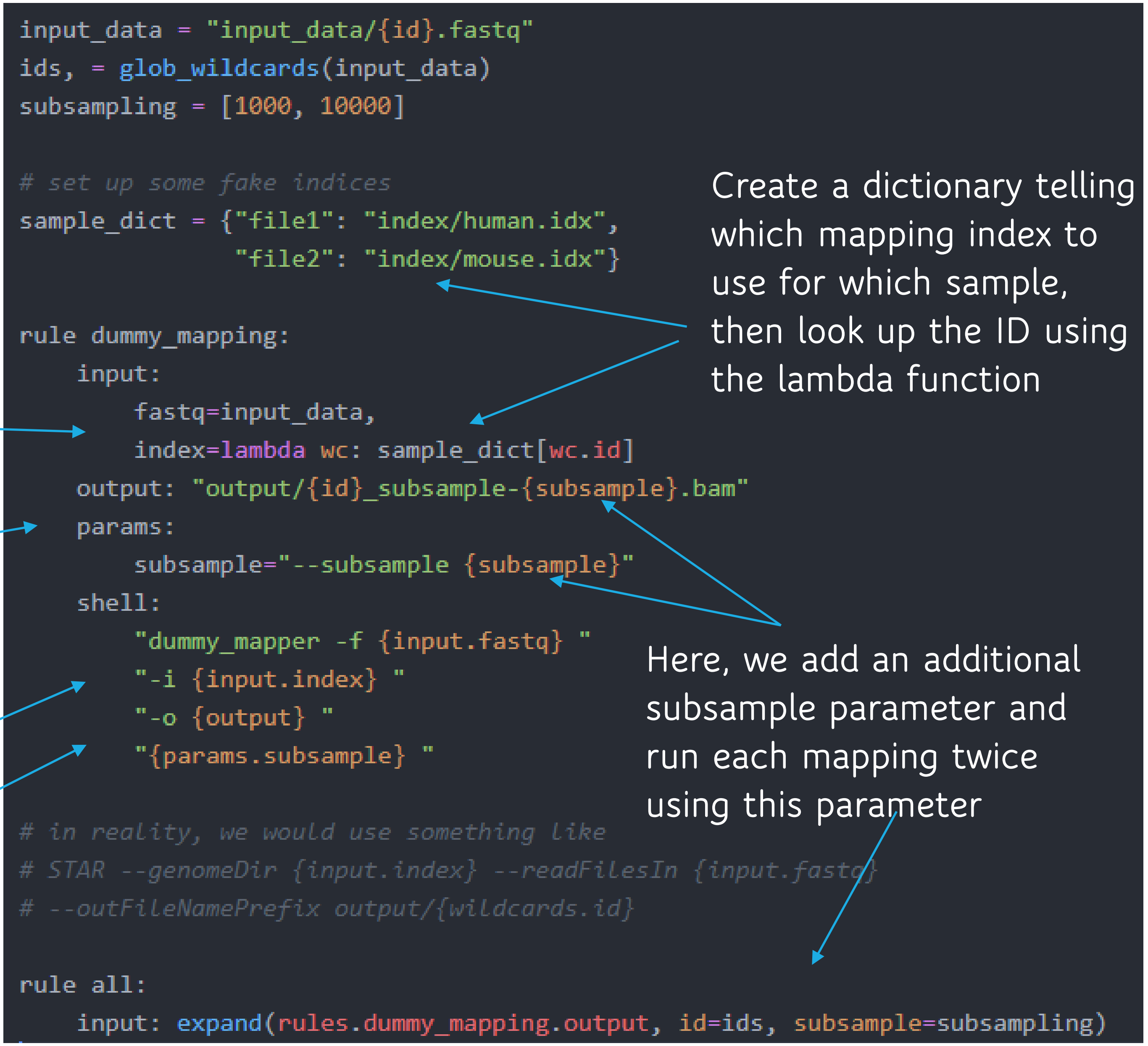
Use the **params** keyword to add additional non-file parameters to the rule

We can **name files** and refer to them in the shell script as e.g. **{input.index}**

The **params** object also gets passed to the **shell** directive and can be accessed in a similar way

Lexical scoping is useful!

exercises/exercise4/Snakefile



```
input_data = "input_data/{id}.fastq"
ids, = glob_wildcards(input_data)
subsampling = [1000, 10000]

# set up some fake indices
sample_dict = {"file1": "index/human.idx",
               "file2": "index/mouse.idx"}

rule dummy_mapping:
    input:
        fastq=input_data,
        index=lambda wc: sample_dict[wc.id]
    output: "output/{id}_subsample-{subsample}.bam"
    params:
        subsample="--subsample {subsample}"
    shell:
        "dummy_mapper -f {input.fastq} "
        "-i {input.index} "
        "-o {output} "
        "{params.subsample} "

# in reality, we would use something like
# STAR --genomeDir {input.index} --readFilesIn {input.fastq}
# --outFileNamePrefix output/{wildcards.id}

rule all:
    input: expand(rules.dummy_mapping.output, id=ids, subsample=subsampling)
```

Create a dictionary telling which mapping index to use for which sample, then look up the ID using the lambda function

Here, we add an additional subsample parameter and run each mapping twice using this parameter

Side note: Lexical scoping and string formatting

How do these variables actually get expanded?

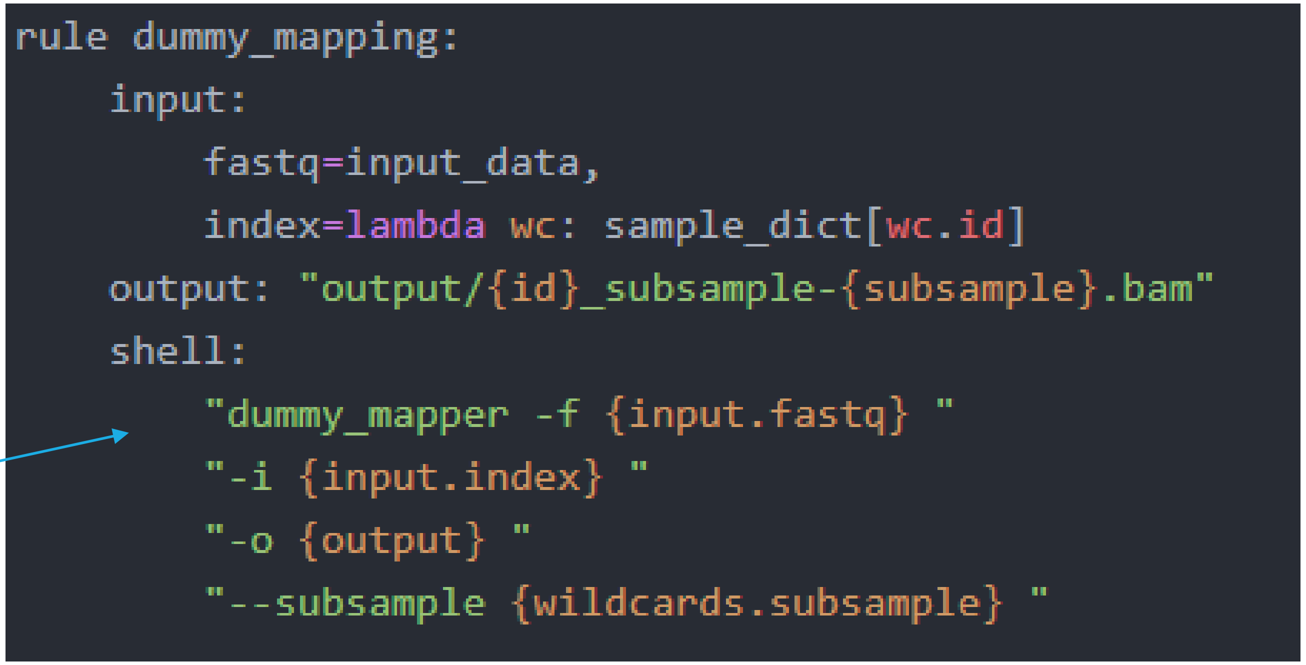
Snakemake creates the following "special" variables as objects:

input, output, params, wildcards, log, threads, resources, config

These are then passed to the **shell** directive and expanded using the .format(**vars) string formatting function

This is useful! You can also refer any variable within the scope of the rule, including external variables defined

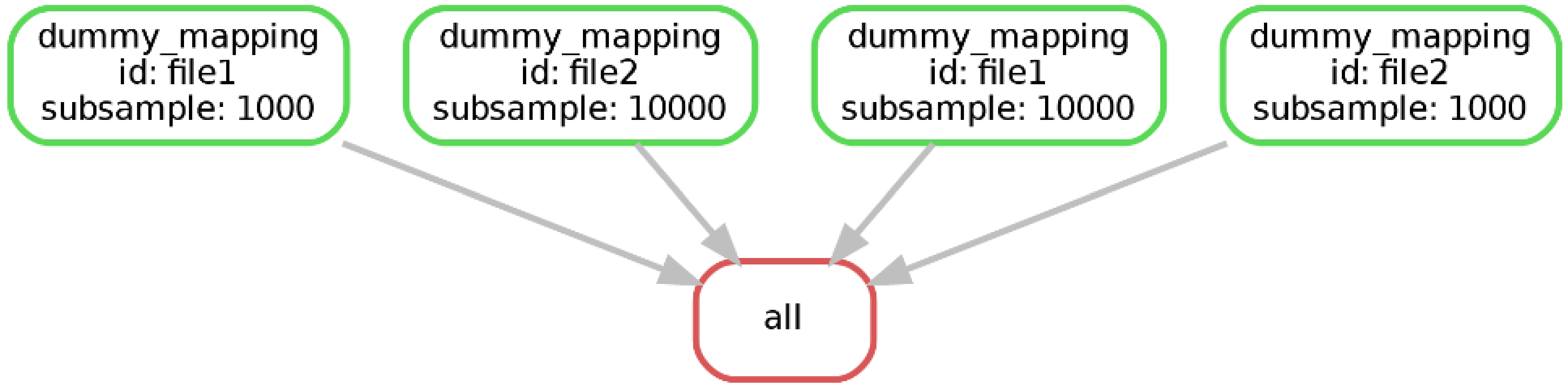
exercises/exercise4/Snakefile



```
rule dummy_mapping:
    input:
        fastq=input_data,
        index=lambda wc: sample_dict[wc.id]
    output: "output/{id}_subsample-{subsample}.bam"
    shell:
        "dummy_mapper -f {input.fastq} "
        "-i {input.index} "
        "-o {output} "
        "--subsample {wildcards.subsample} "
```

For example, {input[0]} refers to the first file in input, {external_variable} refers to some external variable, etc.

Parametrizing rules



Parametrizing rules: config files

exercises/exercise5/Snakefile

```
# use a config file to store our metadata
configfile: "config.yaml"

rule dummy_mapping:
    input:
        fastq=input_data,
        index=lambda wc: config["sample_index"][wc.id]
```

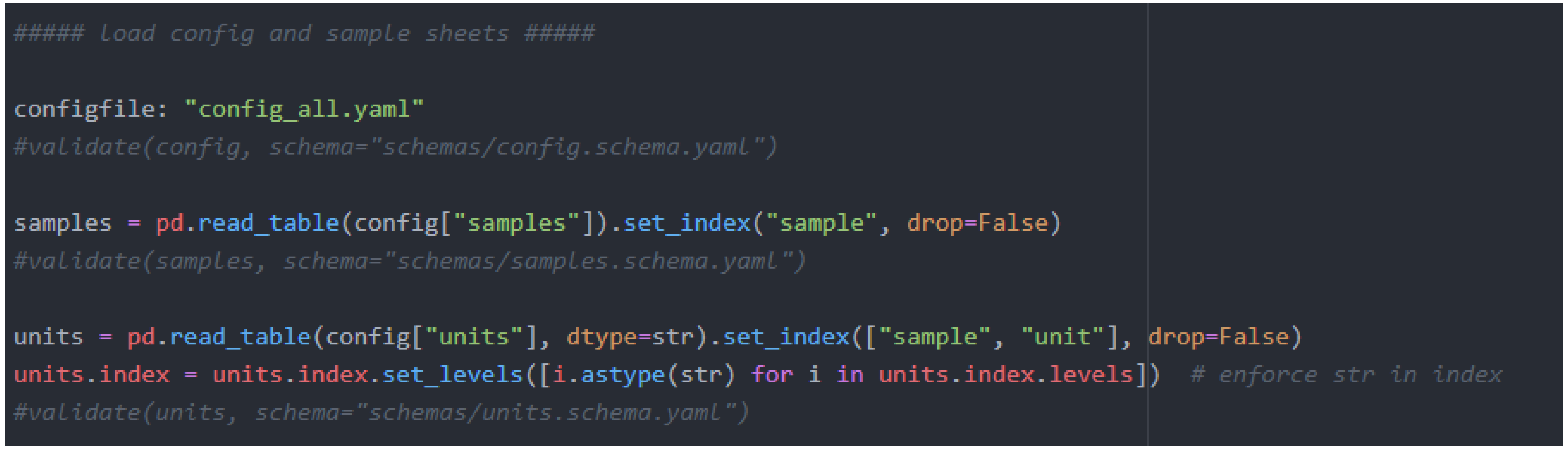
exercises/exercise5/config.yaml

```
sample_index:
  file1: index/human.idx
  file2: index/mouse.idx
```

Alternatively, specify on command line with `snakemake --configfile config.yaml`
Or per-variable with `snakemake --config „my_var=foo“`

Parametrizing rules

Best practice: Store all sample information in a „tidy" CSV file, then load it with pandas



```
##### load config and sample sheets #####

configfile: "config_all.yaml"
#validate(config, schema="schemas/config.schema.yaml")

samples = pd.read_table(config["samples"]).set_index("sample", drop=False)
#validate(samples, schema="schemas/samples.schema.yaml")

units = pd.read_table(config["units"], dtype=str).set_index(["sample", "unit"], drop=False)
units.index = units.index.set_levels([i.astype(str) for i in units.index.levels])  # enforce str in index
#validate(units, schema="schemas/units.schema.yaml")
```

https://github.com/snakemake-workflows/rna-seq-star-deseq2

Custom scripts: R, Julia, and Python

It is also possible to create custom R, Julia, and Python scripts in case a **run:** statement gets too complicated.

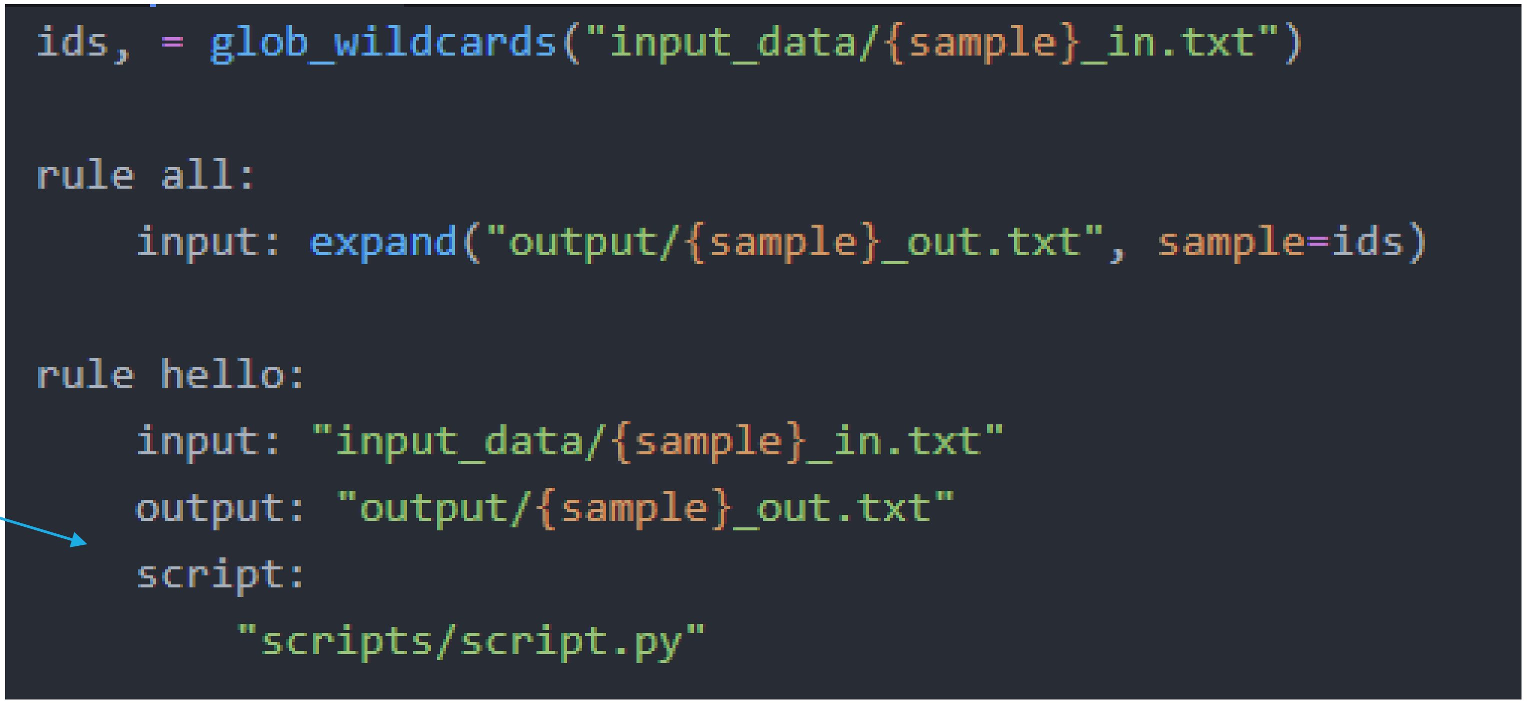
When using the **script**: directive, a special variable called **snakemake** is made available within the script.

snakemake has the same variables embedded within it as are available from the **shell** and **run** directives: **input, output, params, wildcards, log, threads, resources, config**

For example, we can use **snakemake.input[0]** to get the first file in Python.
In R, we would use **snakemake@input[[1]]**.

Custom scripts: R, Julia, and Python

exercises/exercise6/Snakefile

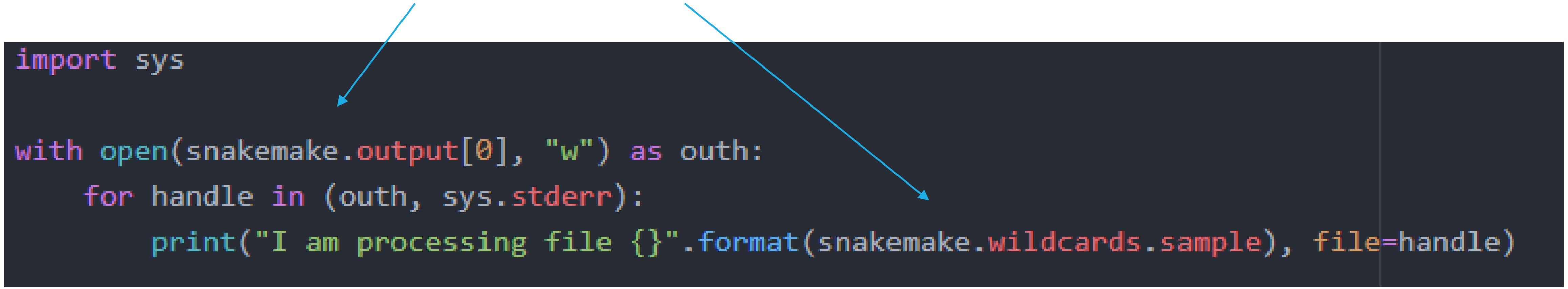


```
ids, = glob_wildcards("input_data/{sample}_in.txt")

rule all:
    input: expand("output/{sample}_out.txt", sample=ids)

rule hello:
    input: "input_data/{sample}_in.txt"
    output: "output/{sample}_out.txt"
    script:
        "scripts/script.py"
```

Use the **script** directive to give a path to a script that will contain the special **snakemake** variable with all its parameters (input, output, etc.)

Custom scripts: R, Julia, and Python

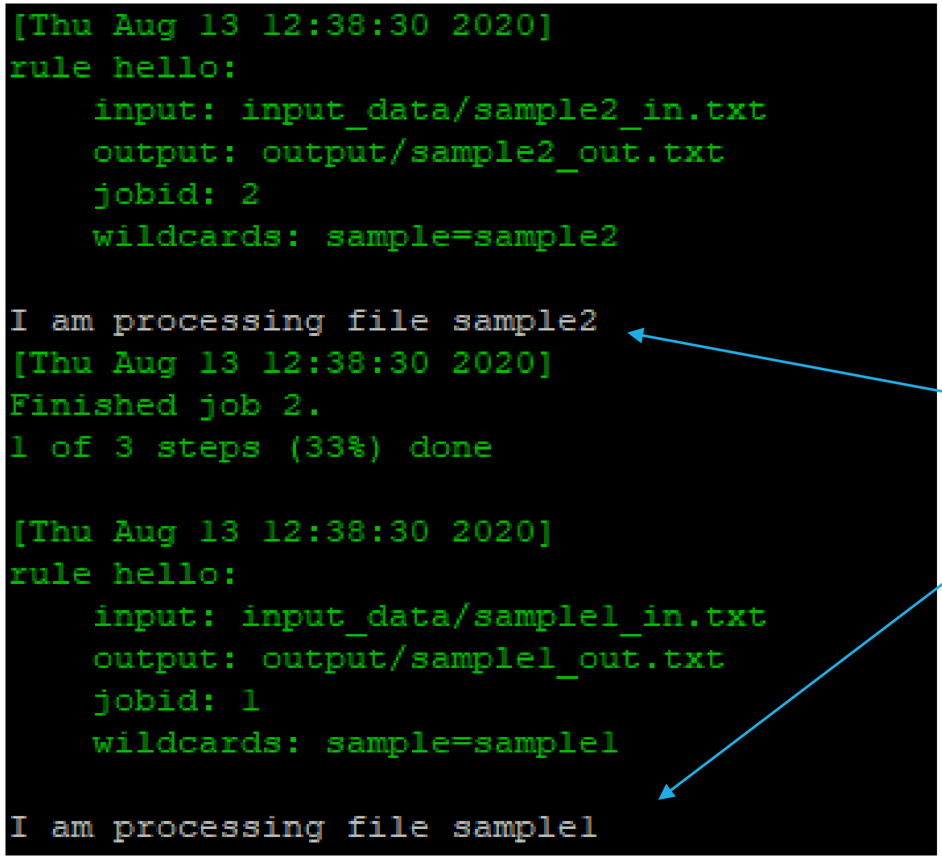
With the **snakemake** object, we can access both the **output** variable as well as the **wildcards** variable, which itself contains the **sample** wildcard for this particular rule



```python
import sys

with open(snakemake.output[0], "w") as outh:
    for handle in (outh, sys.stderr):
        print("I am processing file {}".format(snakemake.wildcards.sample), file=handle)
```

exercises/exercise6/scripts/script.py

Custom scripts: R, Julia, and Python



```
[Thu Aug 13 12:38:30 2020]
rule hello:
    input: input_data/sample2_in.txt
    output: output/sample2_out.txt
    jobid: 2
    wildcards: sample=sample2

I am processing file sample2
[Thu Aug 13 12:38:30 2020]
Finished job 2.
1 of 3 steps (33%) done

[Thu Aug 13 12:38:30 2020]
rule hello:
    input: input_data/sample1_in.txt
    output: output/sample1_out.txt
    jobid: 1
    wildcards: sample=sample1

I am processing file sample1
```

The script correctly prints the **wildcard.sample** variable that was matched from the given **snakemake** object

Reproducible environments

Specifying the computational environment is critical for reproducible workflows.

Since Johannes Köster is also the lead developer of Bioconda, **Snakemake** works exceptionally well with **conda**.
(NB: mamba is also a new alternative that is faster than conda)

Conda is usually sufficient for **prototyping**; for production, **Singularity** or **Docker** are preferred

We can also combine them by running Conda in a Singularity environment!

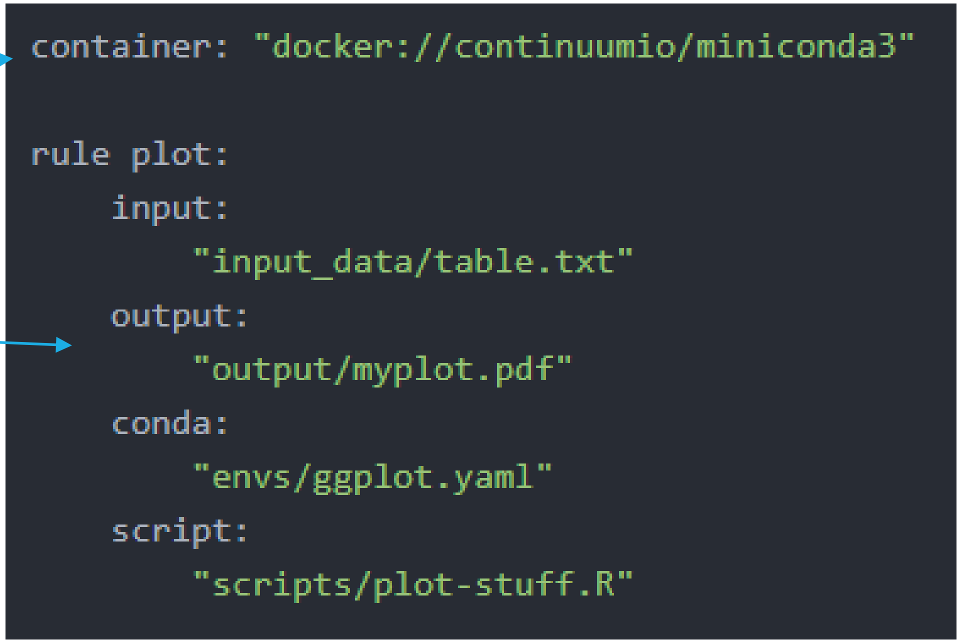
Reproducible environments

Run the entire Snakefile in a Singularity container

Run this rule in a particular conda environment within that container

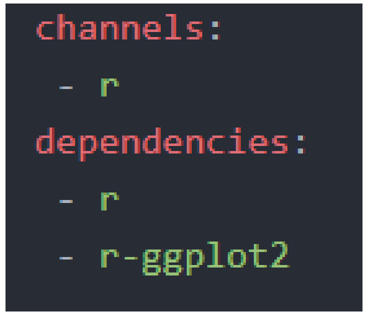
(Can also run containers on a per-rule basis if needed)

exercises/exercise7/Snakefile



```
container: "docker://continuumio/miniconda3"

rule plot:
    input:
        "input_data/table.txt"
    output:
        "output/myplot.pdf"
    conda:
        "envs/ggplot.yaml"
    script:
        "scripts/plot-stuff.R"
```

exercises/exercise7/envs/ggplot.yaml



```
channels:
  - r
dependencies:
  - r
  - r-ggplot2
```

Create conda environment YAML files using
conda env export > env.yaml

Run with: snakemake --use-singularity --use-conda
(NB: need to have singularity installed separately; for now, run this just using --use-conda)

Reproducible environments

We can also compartmentalize software using **wrappers**, which also define the execution environment using conda

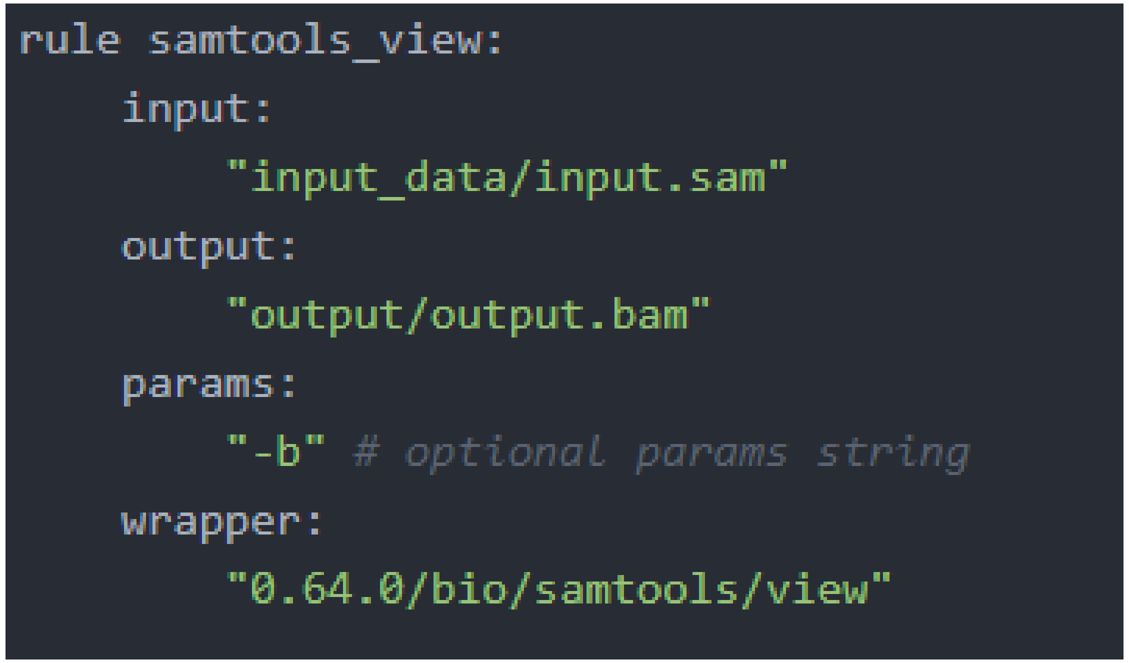
A repository of wrappers for commonly-used bioinformatics tools lives in

https://snakemake-wrappers.readthedocs.io/

Wrappers are easy to create – add your own!

Reproducible environments

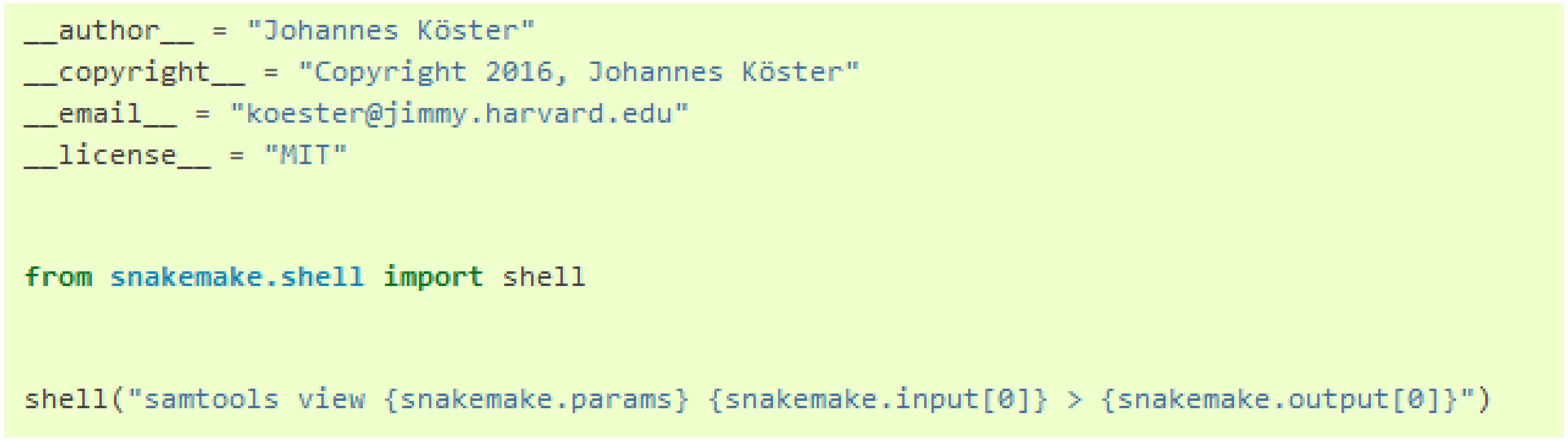
exercises/exercise8/Snakefile



```
rule samtools_view:
    input:
        "input_data/input.sam"
    output:
        "output/output.bam"
    params:
        "-b" # optional params string
    wrapper:
        "0.64.0/bio/samtools/view"
```

Be sure to use the `--use-conda` option when using wrappers!

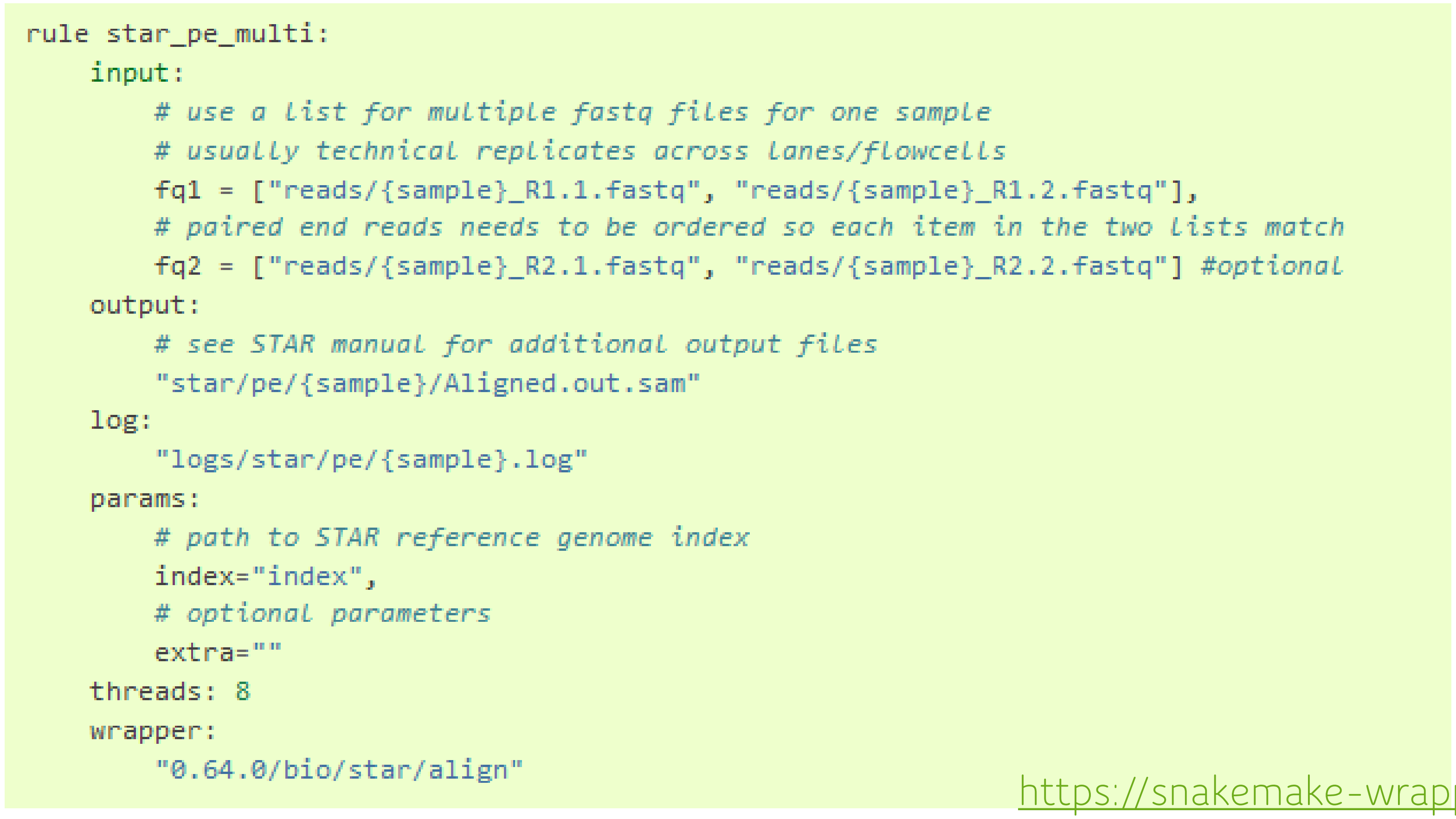
All wrapper code is available on the snakemake wrappers website



```
__author__ = "Johannes Köster"
__copyright__ = "Copyright 2016, Johannes Köster"
__email__ = "koester@jimmy.harvard.edu"
__license__ = "MIT"


from snakemake.shell import shell


shell("samtools view {snakemake.params} {snakemake.input[0]} > {snakemake.output[0]}")
```

https://snakemake-wrappers.readthedocs.io/

More advanced wrappers



```
rule star_pe_multi:
    input:
        # use a list for multiple fastq files for one sample
        # usually technical replicates across lanes/flowcells
        fq1 = ["reads/{sample}_R1.1.fastq", "reads/{sample}_R1.2.fastq"],
        # paired end reads needs to be ordered so each item in the two lists match
        fq2 = ["reads/{sample}_R2.1.fastq", "reads/{sample}_R2.2.fastq"] #optional
    output:
        # see STAR manual for additional output files
        "star/pe/{sample}/Aligned.out.sam"
    log:
        "logs/star/pe/{sample}.log"
    params:
        # path to STAR reference genome index
        index="index",
        # optional parameters
        extra=""
    threads: 8
    wrapper:
        "0.64.0/bio/star/align"
```

https://snakemake-wrappers.readthedocs.io/

Parallelizing your workflow: Locally

There are several options for parallelizing your workflow.
On a single node with many cores, you can use

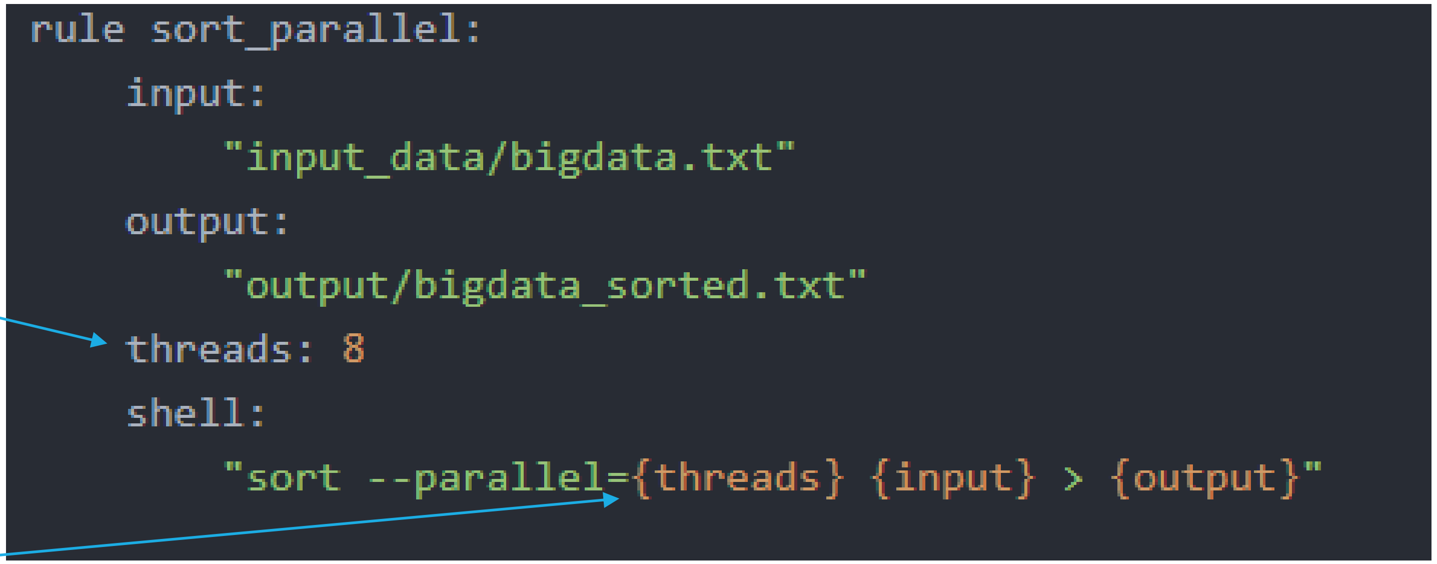
```
snakemake -j $NUM_THREADS
```

Each rule is assumed to take 1 thread by default; you can use the **threads:** keyword to specify how many threads each rule takes.

NB: make sure multithread rules use the **threads** parameter accurately, or you can end up running more threads than expected!

Parallelizing your workflow: Locally

exercises/exercise9/Snakefile



```
rule sort_parallel:
    input:
        "input_data/bigdata.txt"
    output:
        "output/bigdata_sorted.txt"
    threads: 8
    shell:
        "sort --parallel={threads} {input} > {output}"
```

Use **threads**: to specify how many cores a rule takes

The **threads** keyword is accessible in the shell directive through lexical scoping.

NB: If **-j** is less than **threads**, **threads** will be reduced to **-j**.

Parallelizing your workflow: On the cluster

There are several options for parallelizing your workflow on the cluster.

Use the --cluster to provide specific keywords when submitting

For example, to run on the Max Cluster at MDC in Berlin, I use:

--cluster “qsub -V -l h_stack={cluster.h_stack} -l h_vmem={cluster.MEM} -m abe -M david.koppstein@mdc-berlin.de -b y -pe smp {threads} –cwd --q {cluster.q}” **--cluster-config** max-config.yaml

{**threads**} corresponds to the **threads**: directive for each rule

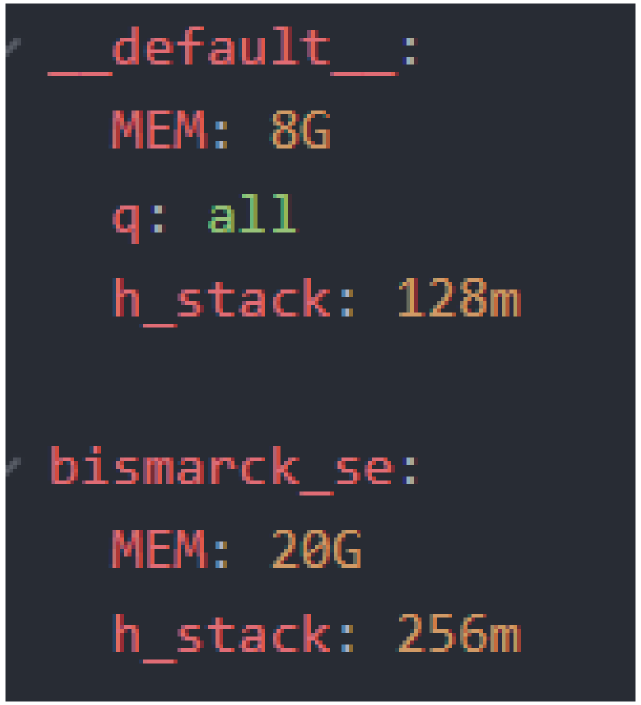
Parallelizing your workflow: On the cluster

The variables {cluster.h_stack}, etc. are defined the cluster configuration file (here, **max-config.yaml**)

By default, rules take resources corresponding to those annotated in **__default__**

For specific rules that require more resources, you can annotate them specifically using the **rule name** (e.g. bismarck_se; attributes not explicitly stated are inherited from __default__)

max-config.yaml



```
__default__:
  MEM: 8G
  q: all
  h_stack: 128m

bismarck_se:
  MEM: 20G
  h_stack: 256m
```

Parallelizing your workflow: On the cluster

Other options:

--drmaa for robust execution and sending Ctrl-C to process (if you have drmaa bindings installed)

Use in the same way as --cluster

--profile [profile] for seamless submission of jobs to specific commonly-used architectures, i.e. SLURM

Check out https://github.com/Snakemake-Profiles/doc for more details on profiles, and feel free to contribute your own architecture!

Parallelizing your workflow: In the cloud

Cloud computing with Snakemake has improved a lot in the last two years!

Google Cloud

(with **--google-life-sciences** option)

Kubernetes: generic cloud execution
(with **--kubernetes** option)

aws

(with **--tibanna** option)

See https://snakemake.readthedocs.io/en/stable/executing/cloud.html for more details

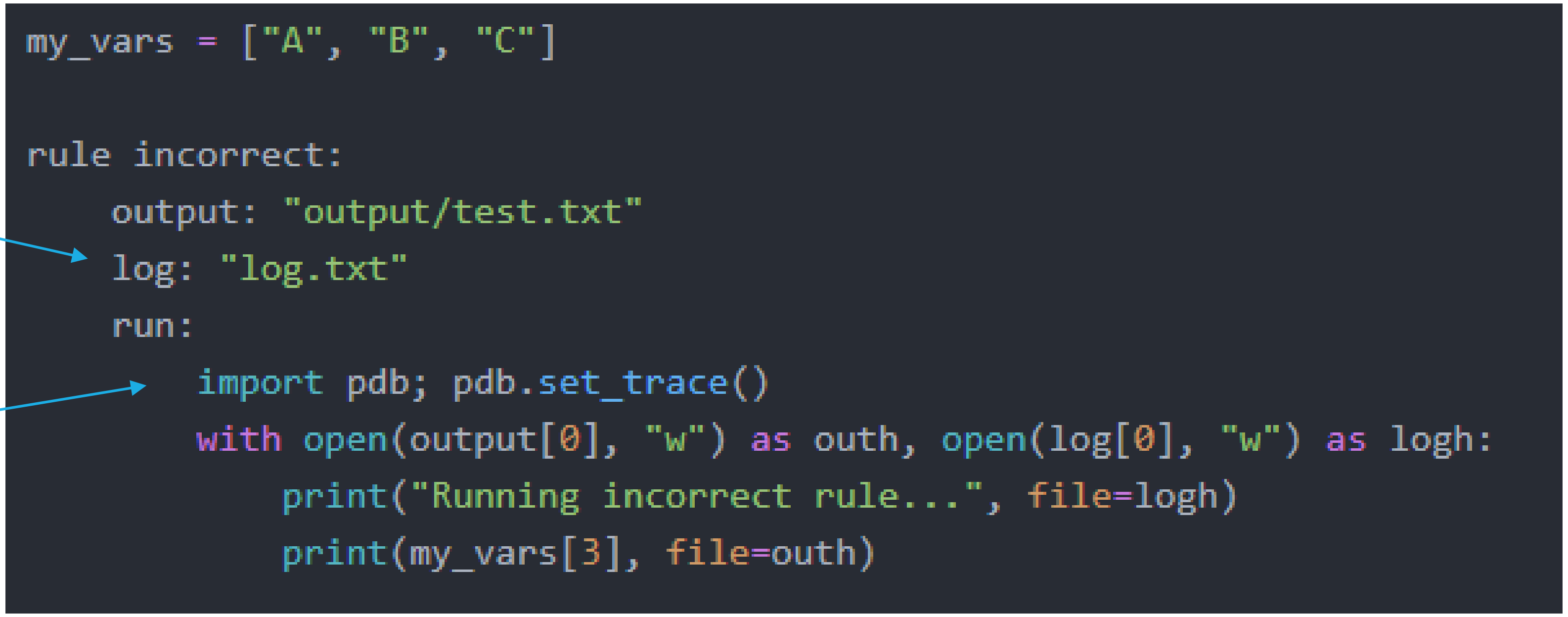
See https://github.com/snakemake/snakemake-tutorials for in-depth Google Life Sciences tutorials

Debugging your Snakefile

exercises/exercise10/Snakefile_incorrect

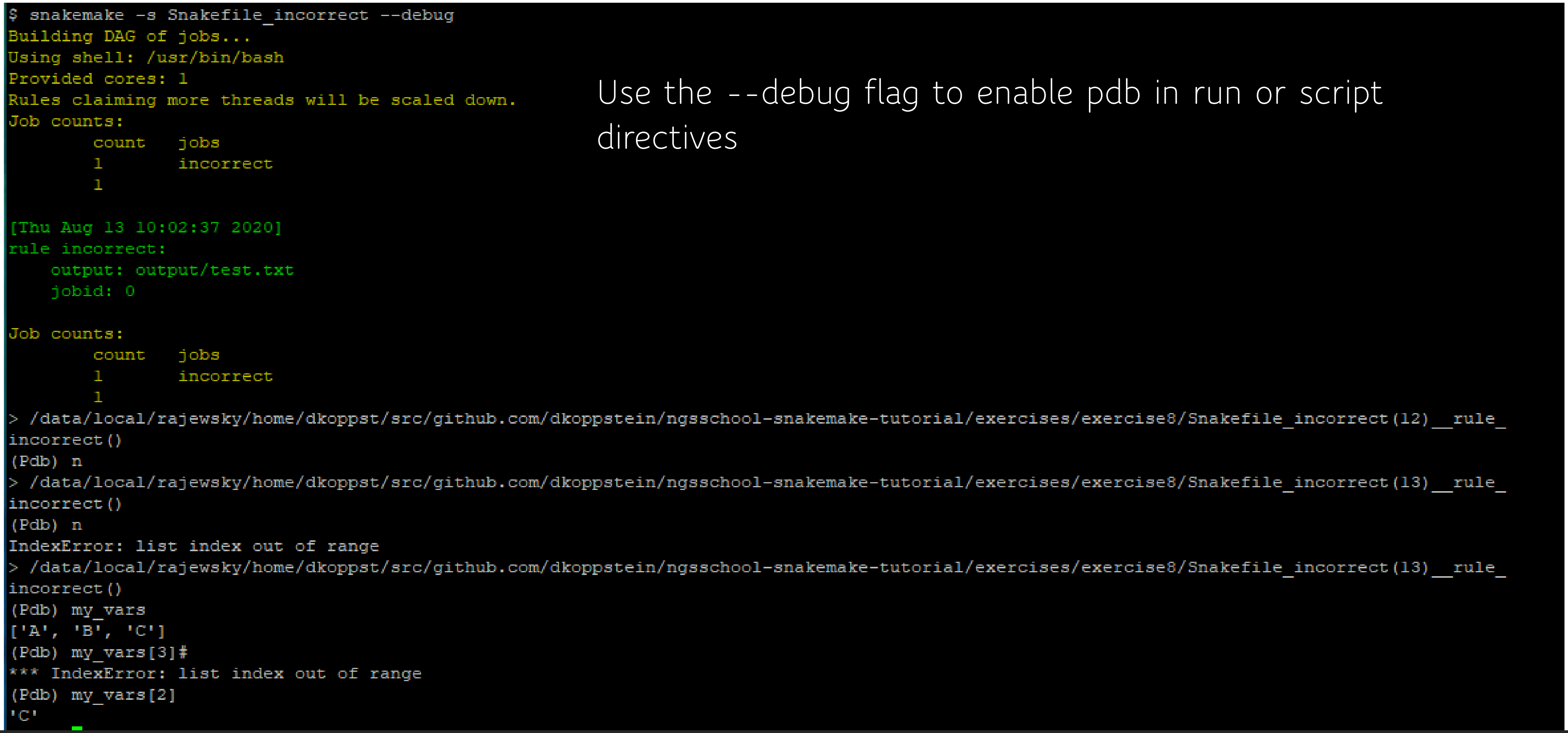
Use the **log**: keyword to capture STDERR automatically, or you can print to it explicitly (as shown here)

You can use **pdb** (Python debugger) to step through rules and examine variables at each step



```
my_vars = ["A", "B", "C"]

rule incorrect:
    output: "output/test.txt"
    log: "log.txt"
    run:
        import pdb; pdb.set_trace()
        with open(output[0], "w") as outh, open(log[0], "w") as logh:
            print("Running incorrect rule...", file=logh)
            print(my_vars[3], file=outh)
```

Debugging your Snakefile

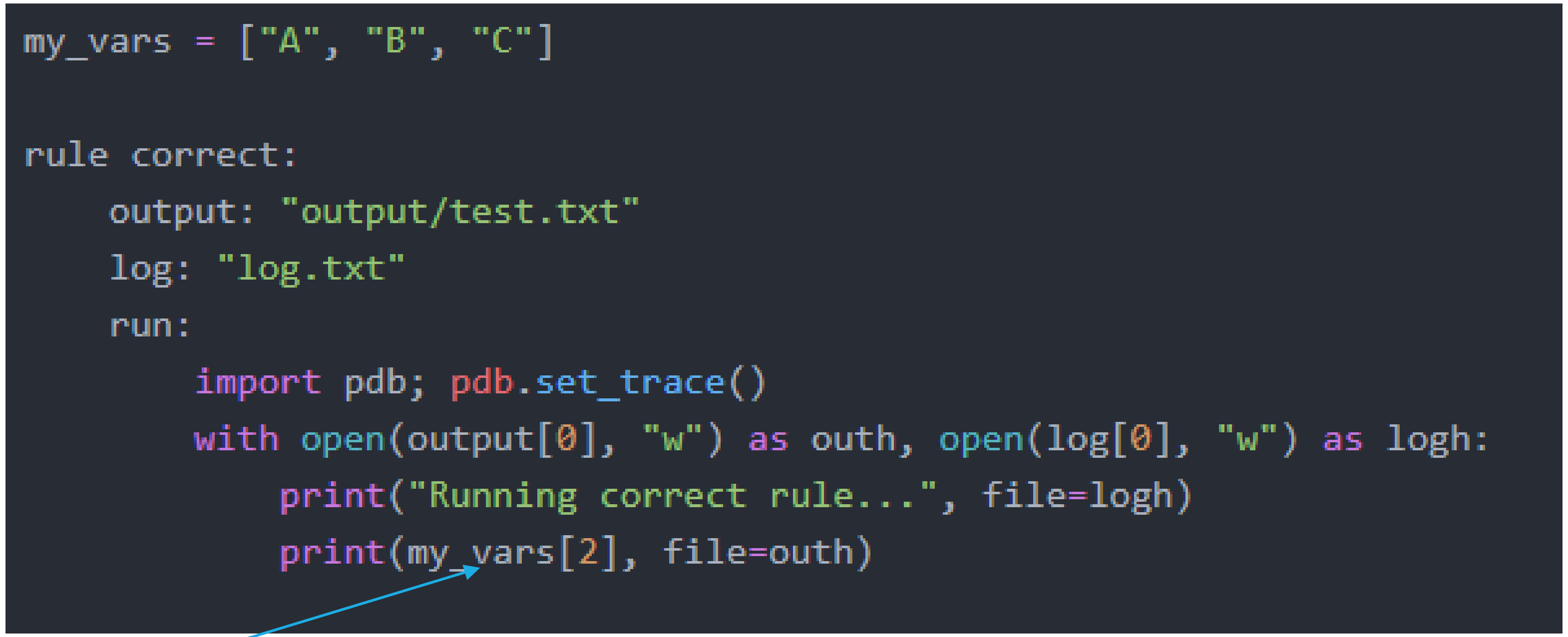


Use the --debug flag to enable pdb in run or script directives

```
$ snakemake -s Snakefile_incorrect --debug
Building DAG of jobs...
Using shell: /usr/bin/bash
Provided cores: 1
Rules claiming more threads will be scaled down.
Job counts:
        count   jobs
        1       incorrect
        1

[Thu Aug 13 10:02:37 2020]
rule incorrect:
    output: output/test.txt
    jobid: 0

Job counts:
        count   jobs
        1       incorrect
        1
> /data/local/rajewsky/home/dkoppst/src/github.com/dkoppstein/ngsschool-snakemake-tutorial/exercises/exercise8/Snakefile_incorrect(12)__rule_
incorrect()
(Pdb) n
> /data/local/rajewsky/home/dkoppst/src/github.com/dkoppstein/ngsschool-snakemake-tutorial/exercises/exercise8/Snakefile_incorrect(13)__rule_
incorrect()
(Pdb) n
IndexError: list index out of range
> /data/local/rajewsky/home/dkoppst/src/github.com/dkoppstein/ngsschool-snakemake-tutorial/exercises/exercise8/Snakefile_incorrect(13)__rule_
incorrect()
(Pdb) my_vars
['A', 'B', 'C']
(Pdb) my_vars[3]#
*** IndexError: list index out of range
(Pdb) my_vars[2]
'C'
```

Debugging your Snakefile

exercises/exercise10/Snakefile



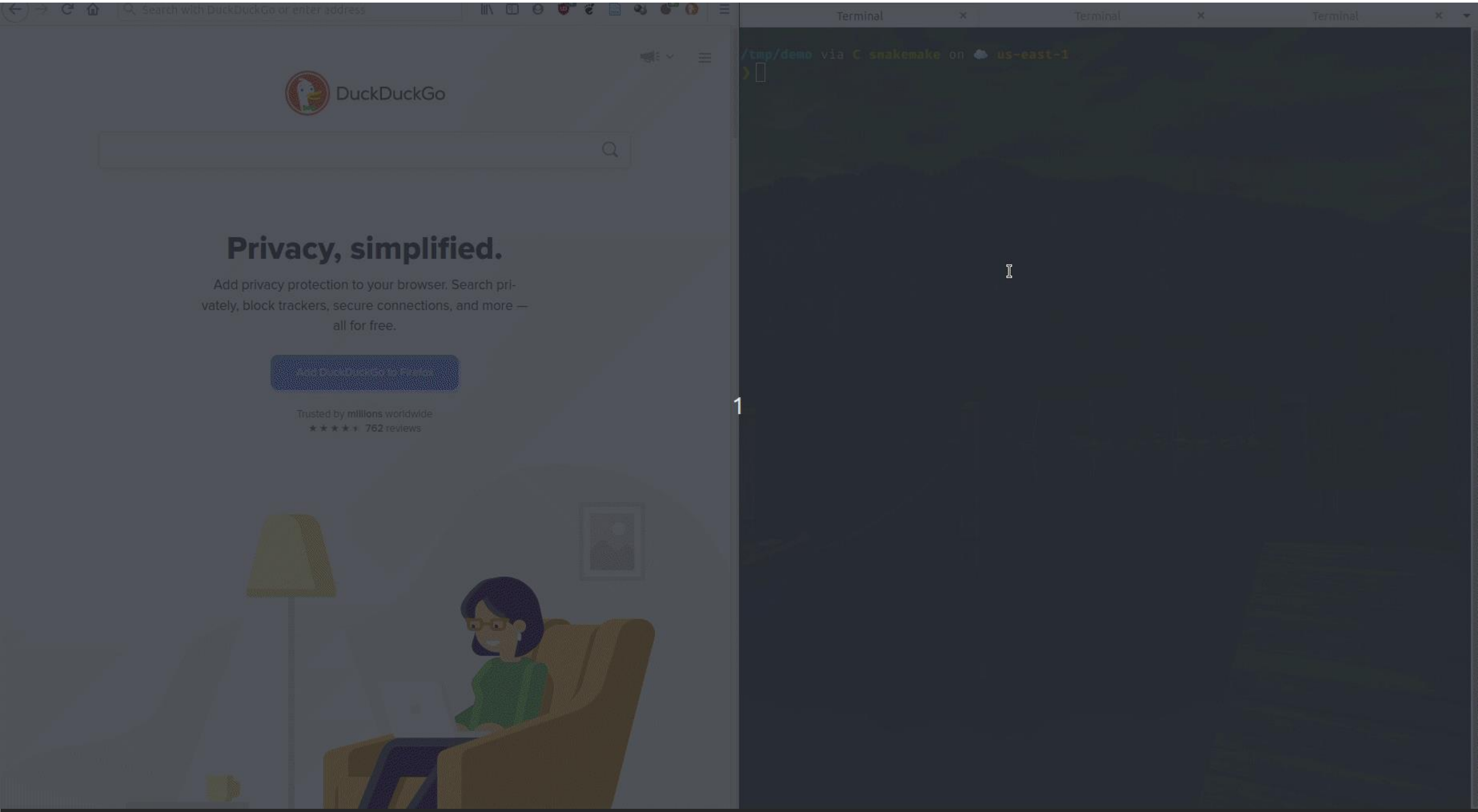
```
my_vars = ["A", "B", "C"]

rule correct:
    output: "output/test.txt"
    log: "log.txt"
    run:
        import pdb; pdb.set_trace()
        with open(output[0], "w") as outh, open(log[0], "w") as logh:
            print("Running correct rule...", file=logh)
            print(my_vars[2], file=outh)
```

Fixed!

NB: the **--verbose** option is also very useful for debugging execution of the Snakefiles themselves

Jupyter Notebook Integration

- Similar to scripts, it is now possible to integrate Jupyter Notebooks into the pipeline itself!
- This enables rapid and reproducible production of reports for a particular pipeline
- In this way, you can write an Ipython Notebook to analyze one set of output, and then parallelize it to apply it to all the outputs!
- Use `--edit-notebook [TARGET]` to create a Jupyter notebook on the fly for a specific output file.
- If your Jupyter notebook is on a remote server, use the `--notebook-listen [IP:PORT]` option to specify a particular port when using SSH port forwarding.
- See exercises/exercise11/Snakefile



Privacy, simplified.

Add privacy protection to your browser. Search privately, block trackers, secure connections, and more — all for free.

Add DuckDuckGo to Firefox

Trusted by millions worldwide
★★★★★ 762 reviews

```
/tmp/demo via C snakemake on ☁ us-east-1
❯
```

1

Creating reports

Use e.g. **reports: „workflow.rst“** keyword with the **--report** commandline option to create HTML reports on the fly

Reports contain captions using the „Restructured Text“ format (commonly used in Python documentation, similar to Markdown)

Output files to include in the report are marked using the **report()** function

See https://koesterlab.github.io/resources/report.html

See https://snakemake.readthedocs.io/en/stable/snakefiles/reporting.html for further details

Best practices workflows

See https://github.com/snakemake-workflows/rna-seq-star-deseq2

Useful keywords and functions

- You can use the **priority:** keyword or **--prioritize TARGET** on the command line to increase the priority of a file (for example, if your PI is bugging you about some particular figure).
- Use the **temp()** function around files that you do not need, and they will be automatically deleted when no rules depend on them anymore.
- Use **protected()** around important files to protect them from accidental deletion.
- Use **directory()** to indicate the creation of a directory (by default, Snakemake only recognizes files)
- The **remote()** function allows you to download files via S3, Google, Dropbox, https, ftp, etc.
- Use the **include:** keyword to include sub-Snakefiles and better organize your code.

Useful command line arguments

- Use **--forceall** to rerun an analysis from scratch.
- Use **--keep-going** to prevent your workflow from stopping if a step fails.
- Use **--until** or **--omit-from** to stop the pipeline at a certain step.
- Use **--delete-all-output** to remove all files generated by the workflow.
- Use **--lint** to make your code pretty
- Use **--rulegraph | dot –Tpdf > ruledag.pdf** to print an overview of the workflow
- Use **--archive** to create a gzipped tarball of the entire workflow, including Conda packages, to upload to e.g. Zenodo

- And many, many more!

Other useful features

- Between workflow caching
- Tracking of parameter and code changes between runs
- Piping between jobs
- Conditional execution based on output
- Shadow directories for per-rule execution
- Various cluster submission features (specifying file latency, bulk submission of jobs, etc.)

Useful resources

- Snakemake wrappers (https://snakemake-wrappers.readthedocs.io/)
- Snakemake on Stackoverflow (https://stackoverflow.com/questions/tagged/snakemake)
- Snakemake Github: Submit issues here!
  https://github.com/snakemake/snakemake
- Google Cloud Snakemake tutorials:
  https://github.com/snakemake/snakemake-tutorials
- Bioconda (https://bioconda.github.io/)
- Cluster profiles (https://github.com/Snakemake-Profiles)

Acknowledgments

- Johannes Köster
- The Snakemake Team
- The Bioconda Team
- The Rajewsky Lab and Max Delbrück Center
- Kasia and NGS School