snakemake

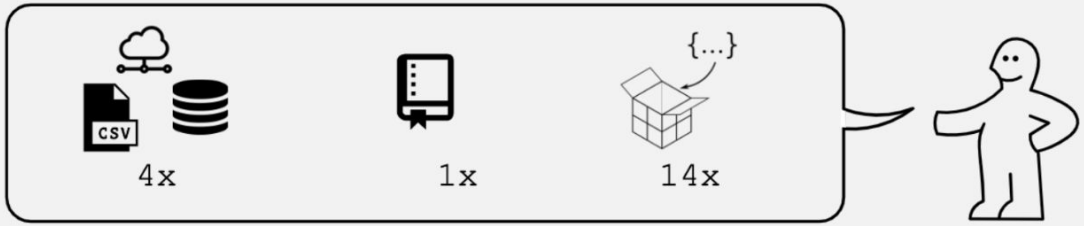
Valeriia (Lera) Lukashenko, Nikhef

HSF DAWG meeting, 08.12.21

KÖMPENDIUM

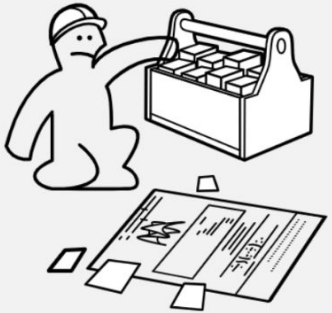
1.

2.



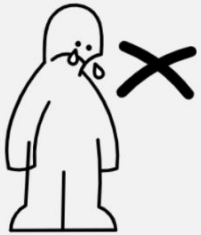
3.

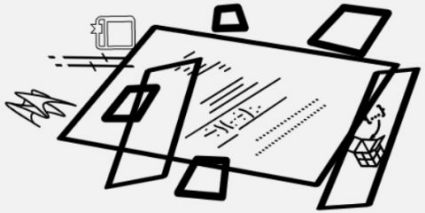
4.



5.

6.





by Karthik Ram

documentation here

Scalability

Readability

snakemake

Portability

Modularization

Transparency

Why to use  ?

Separation from your analysis code	Easy to add to your analysis
Built on top of python	Easy to learn + make your own python wrappers
Can be run anywhere	Supports batch jobs, easy parallelization
Preservation is easier	Supports docker, conda and allows to tag versions
Community	Forums, supports, stackexchange
Utilities	CWL tools, XRootD, paramspace

HTCondor
High Throughput Computing

CONDA

COMMON WORKFLOW LANGUAGE

Your first rule

```
rule my_first_rule:
    input: "myinput1", "myinput2"
    output: "my_output.root"
    shell: "Some command {input} > {output}"
```

Rules live in the file called `Snakefile`

Execute your first rule locally

```
$ snakemake --cores 1 my_first_rule
```

OR

```
$ snakemake -j 1 my_first_rule
```

Rules live in the file called `Snakefile`

Execute your first rule locally via file name

```
$ snakemake --cores 1 my_output.root
```

Rules live in the file called `Snakefile`

Special execution

Test with dry run

```
$ snakemake --cores 1 my_first_rule -n
```

Force reruning everything

```
$ snakemake --cores 1 my_first_rule --forceall
```

OR

```
$ snakemake --cores 1 my_first_rule -F
```

Your first rule can easily talk to python

```
rule my_first_rule:
    input: "myinput1", "myinput2"
    output: "myoutput"
    run:
        text = []
        for f_in in input:
            with open(f_in, "r") as f:
                text.append(f.readlines()[0])
        with open(output[0], "w") as f:
            f.writelines(text)
```

Your rules can easily be connected

```
rule my_first_rule:
    input: "myinput1", "myinput2"
    output: "myoutput"
    shell:  "Some command {input} > {output}"

rule my_second_rule:
    input: "myoutput"
    output: "myoutput2"
    shell:  "Some command {input} > {output}"
```

Your rules can easily be connected

```
$ snakemake --cores 1 my_second_rule
```

Snakemake reruns all the dependent rules when inputs change.

Your rules can easily be connected

```
rule my_first_rule:
    input:
        input_file = "myinput1",
        another_input = "myinput2"
    output:
        output_file = "myoutput"
    shell:  "Some command input.input_file input.another_input > output.output_file"

rule my_second_rule:
    input:
        input_file = rules.my_first_rule.output.output_file
    output:
        output_file = "myoutput2"
    shell:  "Some command {input.input_file} > output.output_file"
```

Your first rule can be generalized with wildcards

```
rule my_first_rule:
    input:
        input_file  =  "/path/to/my/input/file_{year}.root",
        script = "my_script.py"
    output:
        output_file = "myoutput_{year}.file"
    shell:
        "python {input.script} --flag1 {input.input_file} --flag2
 {output.output_file}"
```

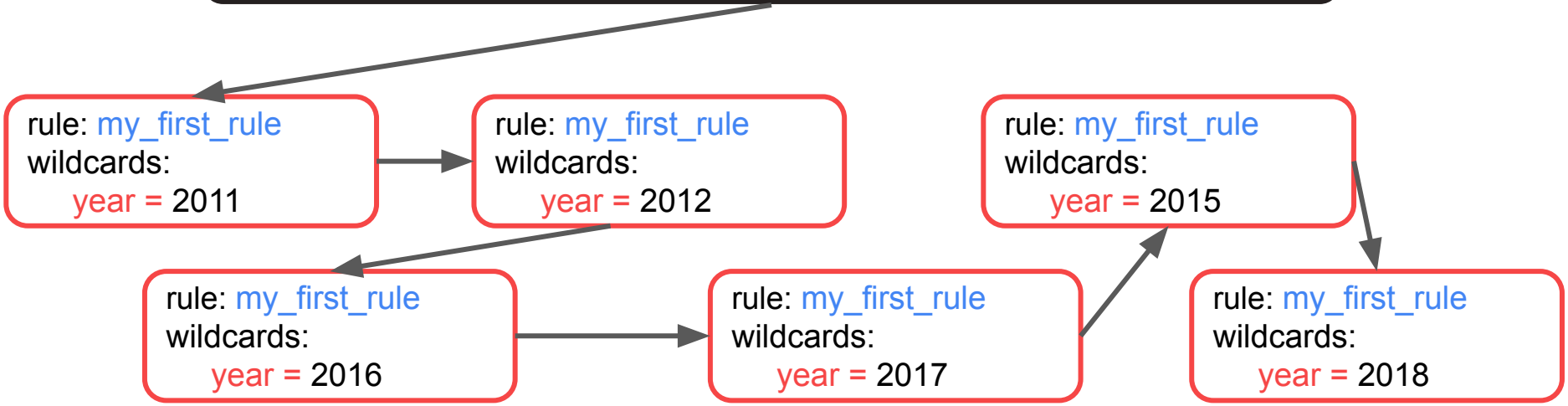
Your first rule can be generalized with wildcards

```
rule my_first_rule_all_years:
    input:
        expand(rules.my_first_rule.output.output_file, year = [
"2011", "2012", "2015", "2016", "2017", "2018"])
```

```
expand(“name_{year}”, year = ['2011', '2012']) == [“name_2011”, “name_2012”]
```

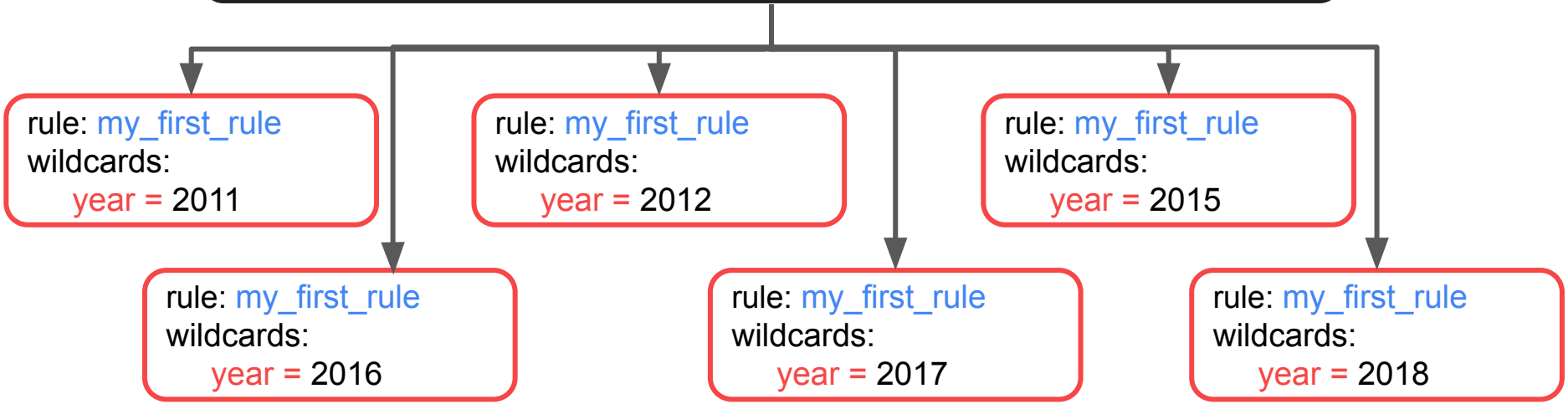
Execute your first rules

```
$ snakemake --cores 1 my_first_rule_all_years
```



Execute your first rules in parallel

```
$ snakemake --cores 6 my_first_rule_all_years
```



Your first rule can easily be generalized with wildcards and python

```
def inputs_provider(wildcards):
    if wildcards.year in ["2011", "2012"]:
        return "/path/to/my/input/file_Run1.root"
    elif wildcards.year in ["2015", "2016", "2017", "2018"]:
        return "/path/to/my/input/file_Run2.root"

rule my_first_rule:
    input:
        input_file  = inputs_provider,
        script = "my_script.py"
    output:
        output_file = "myoutput_{year}.file"
    shell: "Some command input > output"
```

You can use XRootD in snakemake

```
from snakemake.remote.XRootD import RemoteProvider as
XRootDRemoteProvider

XRootD = XRootDRemoteProvider(stay_on_remote=True)

rule my_first_rule:
    input: XRootD.remote("root://my_path/file.root")
    output: "myoutput"
    shell: "Some command {input} > {output}"
```

Pretty much everything you need can be specified

```
rule my_first_rule:
    input: "myinput1", "myinput2"
    output: "myoutput"
    params:
         value = 2.0
    threads: 8
    log: "my_log_file"
    conda:  "my_env.yml"
    shell: "Some command {input} {params.value} > {output}
  | tee {log}"
```

There are also useful utils: [snakemake.utils]

Pretty much everything you need can be specified

```
rule my_first_rule:
    input: "myinput1", "myinput2"
    output: "myoutput"
    params:
         value = 2.0
    threads: 8
    log: "my_log_file"
    container: "docker://my_docker_container"
    shell: "Some command {input} {params.value} > {output}
 | tee {log}"
```

There are also useful utils: [snakemake.utils]

You can use multiple Snakefiles

```
$ cat Snakefile

include: "analysis/selection/Snakefile"
include: "analysis/mass_fit/Snakefile"
include: "analysis/acceptance/Snakefile"
```

You can separate configuration from rules definition

```
$ cat config.yaml

years: [2011, 2012, 2015, 2016, 2017, 2018]
```

You can separate configuration from rules definition

```
$ cat analysis/utils/helpers.smk

configfile: "analysis/config.yaml"
include: "analysis/selection/Snakefile"
include: "analysis/mass_fit/Snakefile"
include: "analysis/acceptance/Snakefile"
```

You can separate configuration from the rules definition

```
rule my_first_rule_all_years:
    input:
        expand(rules.my_first_rule.output.output_file, year =
config["years"])
```

You can separate your python wrappers

```
$ cat analysis/utils/helpers.smk

import os
output_dir = "/path/to/output/"

def output_path(path):
    return os.path.join(output_dir, path)
```

You can separate your python wrappers

```
$ cat analysis/Snakefile

configfile: "analysis/config.yaml"
include: "analysis/utils/helpers.smk"
include: "analysis/selection/Snakefile"
include: "analysis/mass_fit/Snakefile"
include: "analysis/acceptance/Snakefile"
```

You can separate your python wrappers

```
rule my_first_rule:
    input: "myinput1", "myinput2"
    output: output_path("my_output.root")
    shell: "Some command {input} > {output}"
```

You can reuse your rules

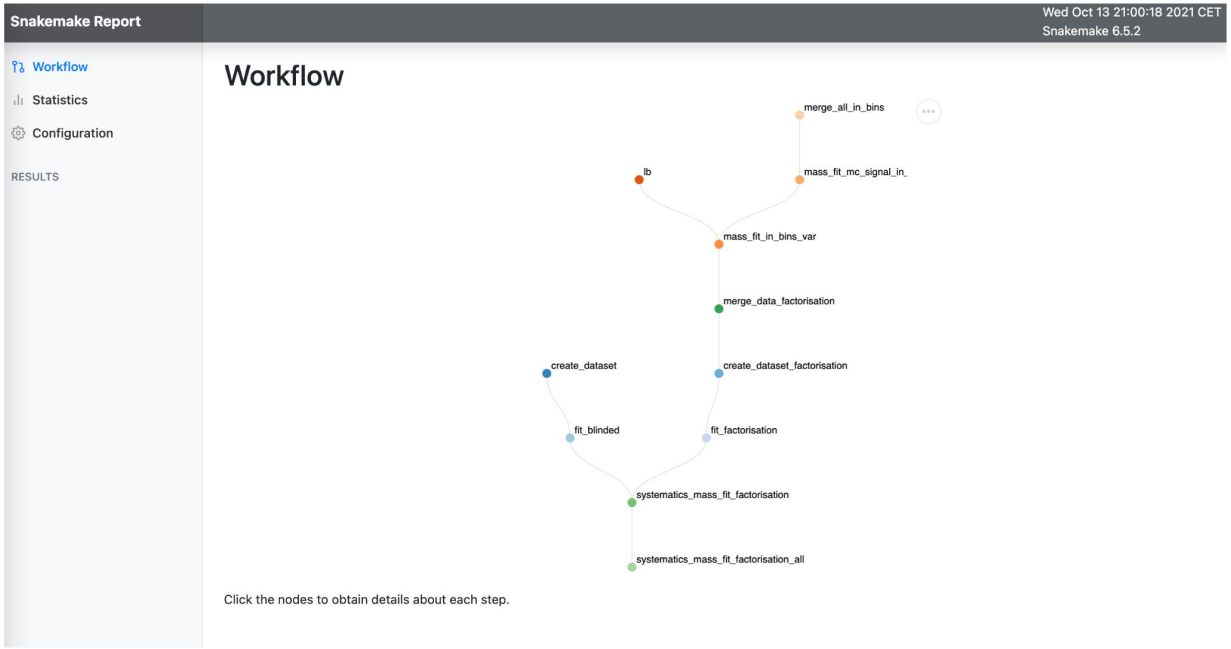
```
from snakemake.utils import min_version
min_version("6.0")

module other_workflow:
    snakefile:
        "other_workflow/Snakefile"

 use rule * from other_workflow as other_* with:
    params:
        value = 6
    output: "other_output"
```

You can build reports

```
$ snakemake --cores 1 my_first_rule --report report.html
```



You can build reports

```
$ snakemake --cores 1 my_first_rule --report report.html
```

Rule mass_fit_mc_signal_in_bins_var

Jobs 4

Input files

- at 0x7f8015511c20>
- mass_fit/fit_mc.py

Output files

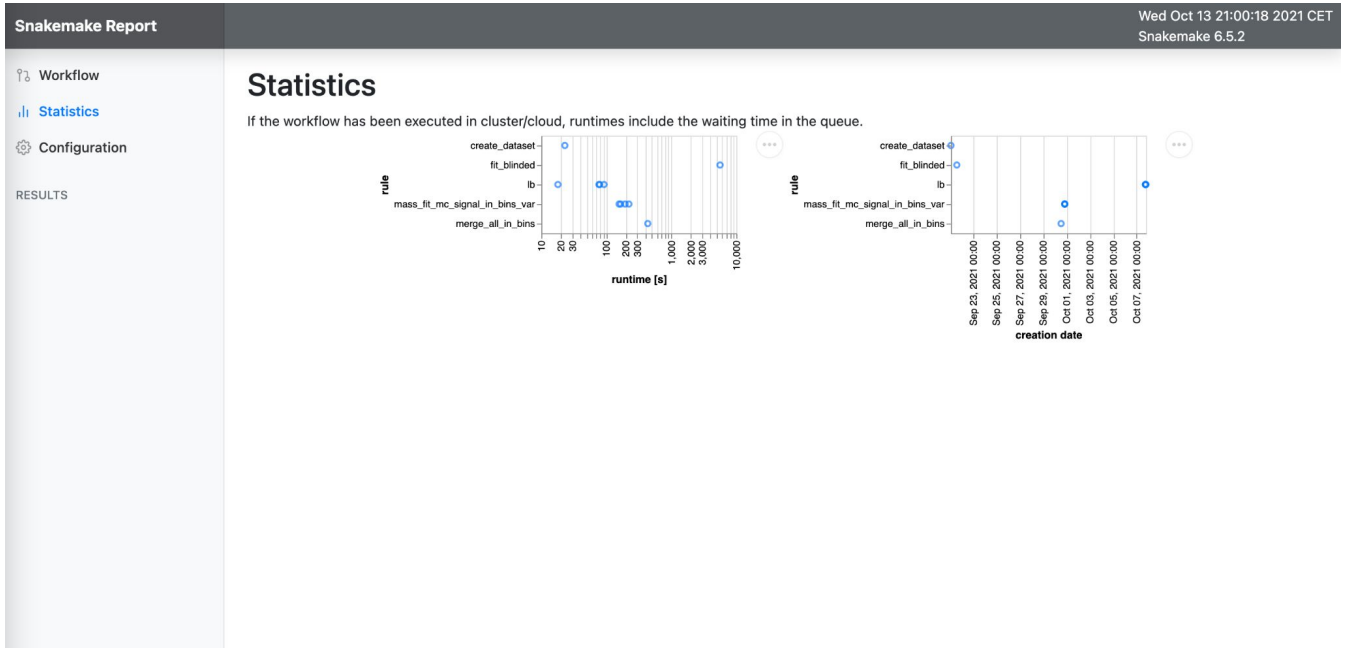
- /project/bfys/valukash/BsJpsiphi/p2vv/scripts/run2/output/systematics/mass_fit_factorisation_in_bins/mass_fit_mc_signal/{mc_config}/{model}/{year,\d{4}}/{variable,[A-Za-z0-9]+}/{bin,[0-9]+}/figs
- /project/bfys/valukash/BsJpsiphi/p2vv/scripts/run2/output/systematics/mass_fit_factorisation/mass_fit_mc_signal_in_bins/{mc_config}/{model}/{year,\d{4}}/{variable,[A-Za-z0-9]+}/{bin,[0-9]+}/pars

Code

```
1 mkdir {output.plot_dir} | python {input.script} --ntuple-in {input.data_in}                    --ntuple-cuts "{params.ntuple_cut}"
```

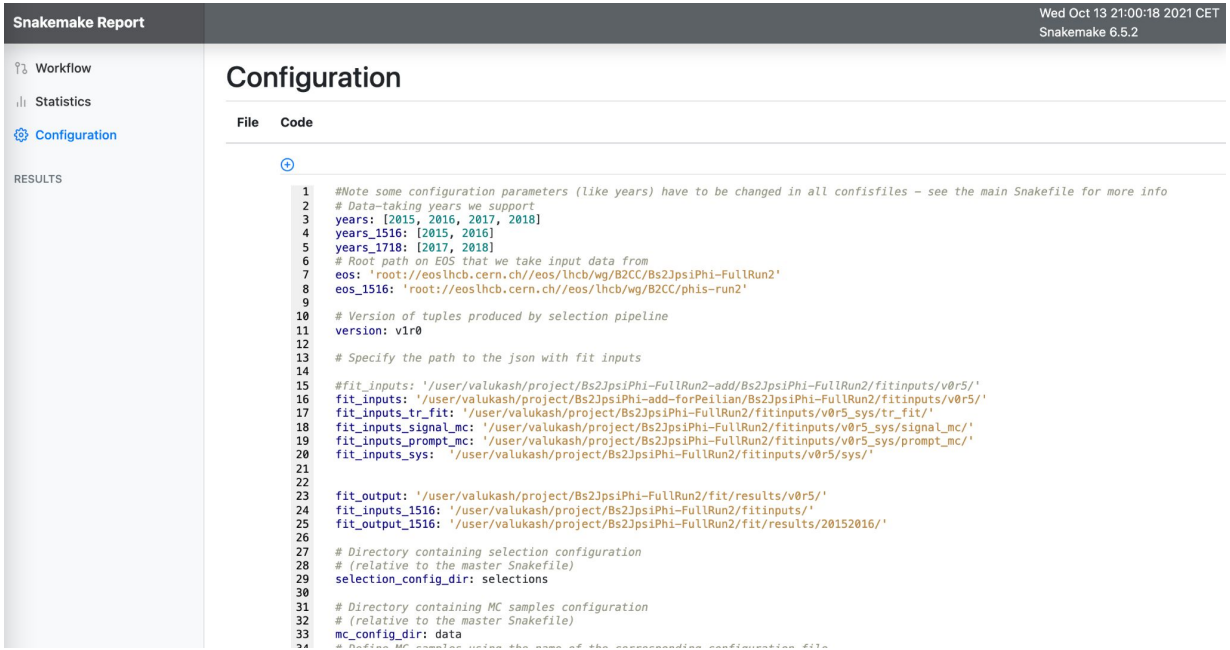
You can build reports

```
$ snakemake --cores 1 my_first_rule --report report.html
```



You can build reports

```
$ snakemake --cores 1 my_first_rule --report report.html
```



Execute your first rule on a cluster

```
$ snakemake -j 10 --cluster "qsub -q <queue>" my_first_rule
```

and HTCondor Software Suite

You need to install: snakemake-condor-profile

```
$ snakemake -j 10 my_first_jobs --profile HTCondor
```

and reana finally there!

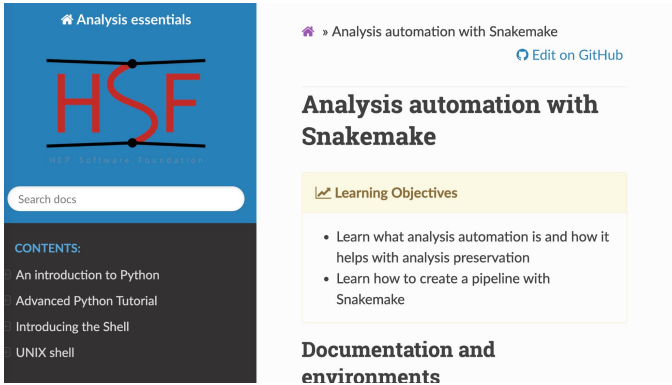
November 30, 2021

#snakemake | #cluster | #client

Support for running Snakemake workflows

We are thrilled to announce the support for running Snakemake workflows on REANA reproducible analysis platform, starting from the REANA 0.8.0 release. Snakemake joins CWL and Yadage as another complete workflow definition language that REANA users can use to run their analysis workflows.

in



@

STARTERKIT
LHCb
Estd. 2015

Analysis Essentials snakemake lesson

in

```
[(base) [valukash@lxplus753 ~]$ lb-conda default
[bash-5.0$ snakemake --help
 usage: snakemake [-h] [--dry-run] [--profile PROFILE
                  [--set-resources RULE:RESOURCE=VALU
                  [--preemptible-rules PREEMPTIBLE_RU
```

Part of the [lb-conda]() environment

A

Analysis Workflow Template

Project ID: 115835

Star 0 | Fork 0

18 Commits | 1 Branch | 0 Tags | 184 KB Files | 290 KB Storage

This repository demonstrates best practices for designing a user-analysis workflow. It contains an example, how to use snakemake and the CI. It can serve as a template for new analyses.

master | analysis-workflow-template / | History | Find file | Web IDE | Clone

Merge branch 'add-independent-paramspace-workflow' into 'master'
Sebastian Neubert authored 3 weeks ago

9cc78897

README | CI/CD configuration | No license. All rights reserved

Name	Last commit	Last update
scripts	rearrange and small improvement	6 months ago
snakefiles	rearrange and small improvement	6 months ago

[LHCb snakemake template]()

is an easy to learn and read, but powerful workflow manager

is a community with a good user-support

makes my life 1000 times easier and probably will make yours

Backup with installation

installation

```
$ python3 -m pip install --user snakemake
```

OR

```
$ conda install -n base -c conda-forge mamba
$ conda activate base
$ mamba create -c conda-forge -c bioconda -n snakemake snakemake
```