Snakemake

Raymond Wan

July 14, 2018

Outline

Overview

Snakemake is:

- A workflow definition language, similar to Make.
- Based on Python.
- Developed for bioinformatics[1], but general enough for other tasks.

[1] The application of computer science, statistics, etc. to problems in the life sciences.

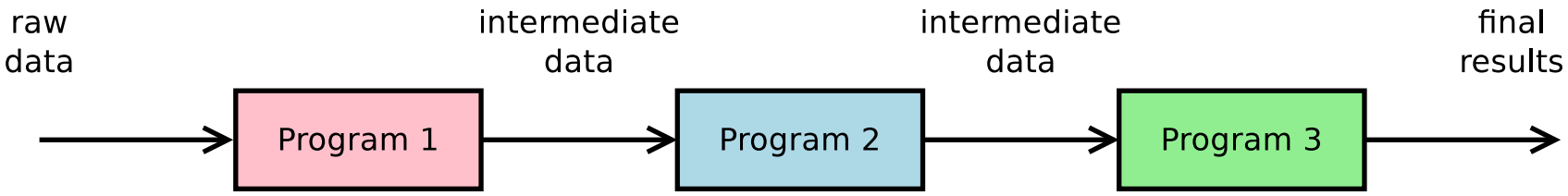
Disclaimer

In terms of scripts, I'm primarily a Perl programmer and have had to learn Python the past year because of Snakemake. Thus, I don't claim to be an expert of either Python or Snakemake. I will cover what I've needed to know (so far).

Hopefully this talk will give you enough information to decide if Snakemake is useful for your work.

Pipelines

Like other areas of data analysis, bioinformaticians often chain a series of programs together so that the output of one program becomes the input of another:



Make

For longer than what some of us are willing to admit ☺, we may have used Make to build programs and libraries from source code.

Files with the filename `Makefile` are used with rules of this form:

```
target:   dependencies
    system command(s)
```

Make

Thus, a rule in a Makefile might look like this:

```
helloworld:   helloworld.c
        gcc helloworld.c -o helloworld
```

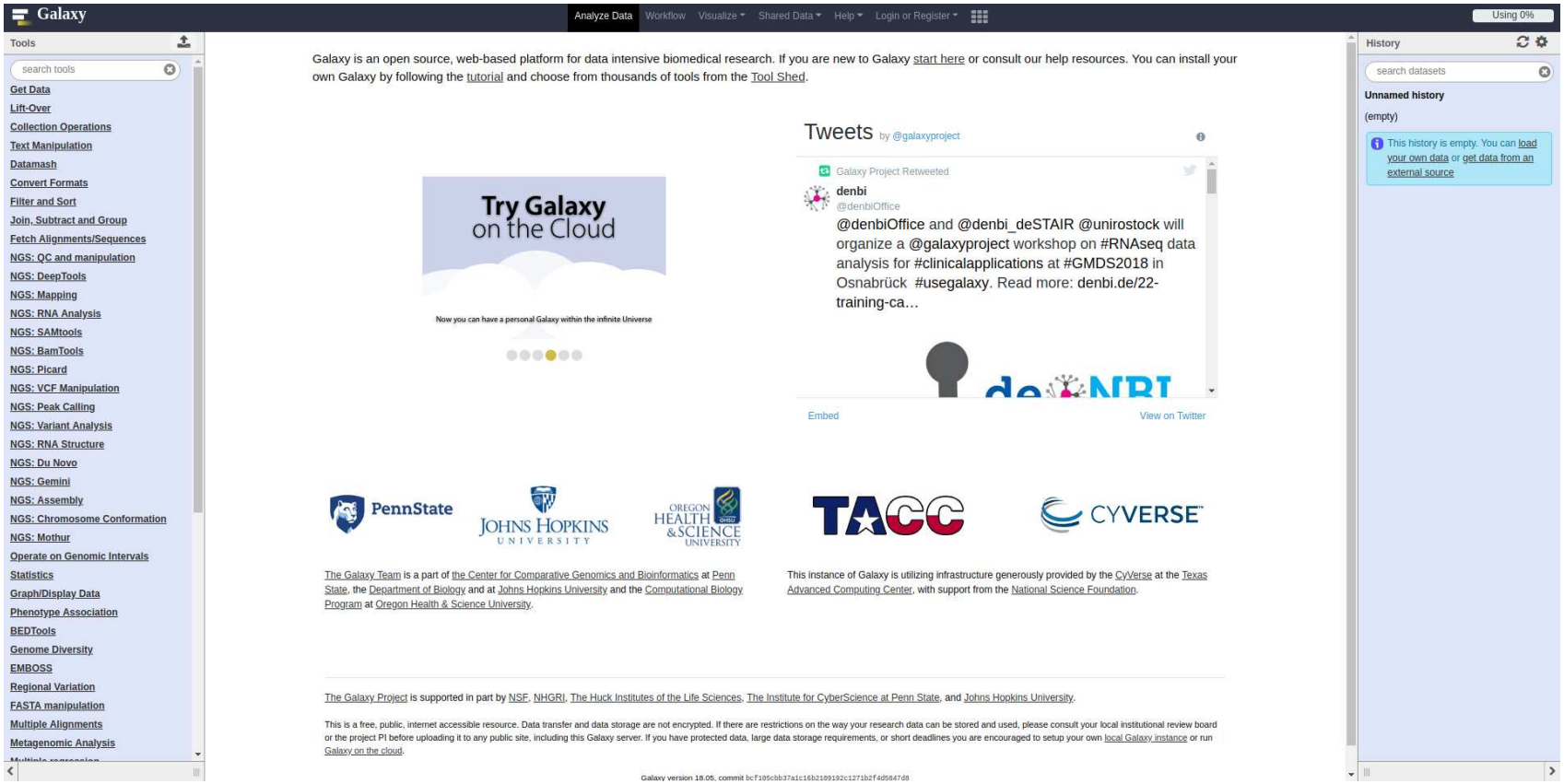
When compiling a program with a Makefile, you might be using one compiler to compile source files to object files, linking them, and maybe even doing this in parallel.

Galaxy

Not surprisingly, bioinformaticians have developed their own systems (too numerous to list them all) which ultimately do almost the same thing (from a computing perspective).

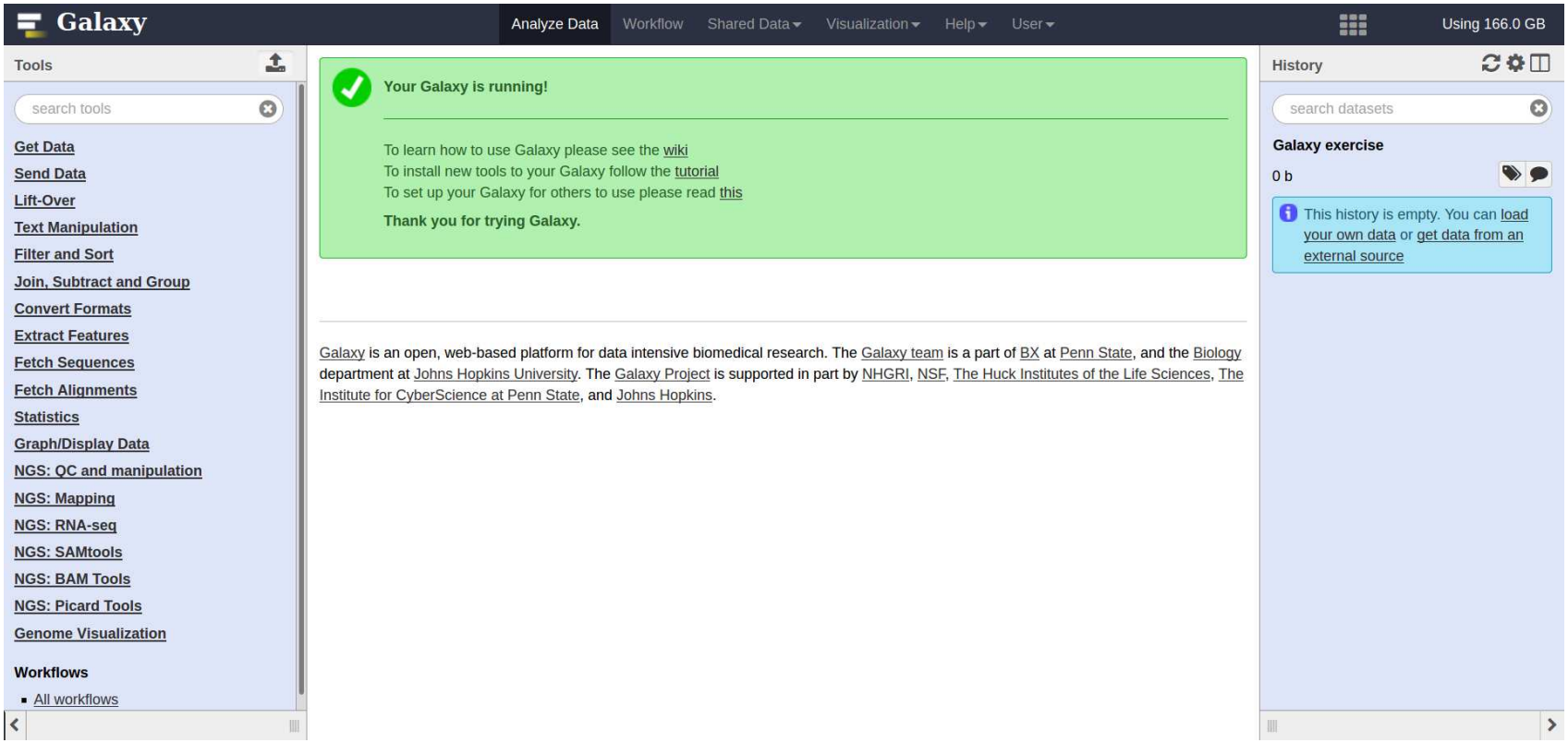
Galaxy (which predates Snakemake) is a web-based system for constructing such workflows.

Galaxy



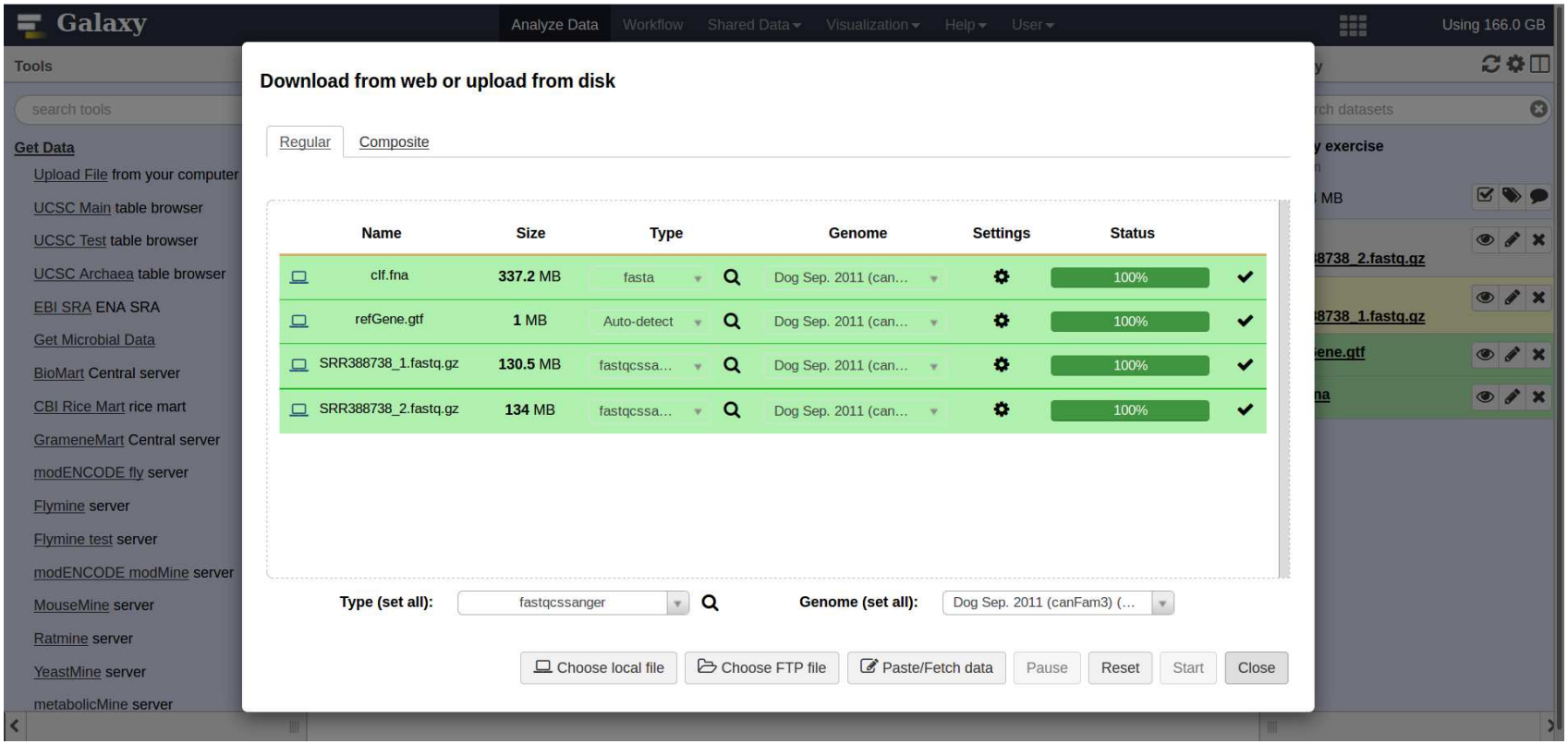
(Figure source: https://usegalaxy.org/.)

Galaxy



(Figure source: Local installation.)

Galaxy


Galaxy
Analyze Data
Workflow
Shared Data
Visualization
Help
User
Using 166.0 GB
Tools
search tools
Get Data
Upload File from your computer
UCSC Main table browser
UCSC Test table browser
UCSC Archaea table browser
EBI SRA ENA SRA
Get Microbial Data
BioMart Central server
CBI Rice Mart rice mart
GrameneMart Central server
modENCODE fly server
Flymine server
Flymine test server
modENCODE modMine server
MouseMine server
Ratmine server
YeastMine server
metabolicMine server
Download from web or upload from disk
Regular
Composite
Name
Size
Type
Genome
Settings
Status
clf.fna
337.2 MB
fasta
Dog Sep. 2011 (can...
100%
refGene.gtf
1 MB
Auto-detect
Dog Sep. 2011 (can...
100%
SRR388738_1.fastq.gz
130.5 MB
fastqcssa...
Dog Sep. 2011 (can...
100%
SRR388738_2.fastq.gz
134 MB
fastqcssa...
Dog Sep. 2011 (can...
100%
Type (set all):
fastqcssanger
Genome (set all):
Dog Sep. 2011 (canFam3) (...
Choose local file
Choose FTP file
Paste/Fetch data
Pause
Reset
Start
Close

Galaxy

Galaxy | Analyze Data | Workflow | Shared Data | Visualization | Help | User | Using 166.0 GB

Tools

- Map with Bowtie for Illumina
- Map with BWA for Illumina
- Map with BWA for SOLiD
- Megablast compare short reads against htgs, nt, and wgs databases
- Parse blast XML output
- Map with BWA-MEM - map medium and long reads (> 100 bp) against reference genome
- Map with BWA - map short reads (< 100 bp) against reference genome
- Bowtie2 - map reads against reference genome
- TopHat Gapped-read mapper for RNA-seq data

NGS: RNA-seq
NGS: SAMtools
NGS: BAM Tools
NGS: Picard Tools
Genome Visualization

Workflows
- All workflows

TopHat Gapped-read mapper for RNA-seq data (Galaxy Tool Version 2.1.0) — Options

Is this single-end or paired-end data?
Paired-end (as individual datasets)

RNA-Seq FASTQ file, forward reads
3: SRR388738_1.fastq
Must have Sanger-scaled quality values with ASCII offset 33

RNA-Seq FASTQ file, reverse reads
4: SRR388738_2.fastq
Must have Sanger-scaled quality values with ASCII offset 33

Mean Inner Distance between Mate Pairs
300
-r/--mate-inner-dist; This is the expected (mean) inner distance between mate pairs. For, example, for paired end runs with fragments selected at 300bp, where each end is 50bp, you should set -r to be 200. The default is 50bp.

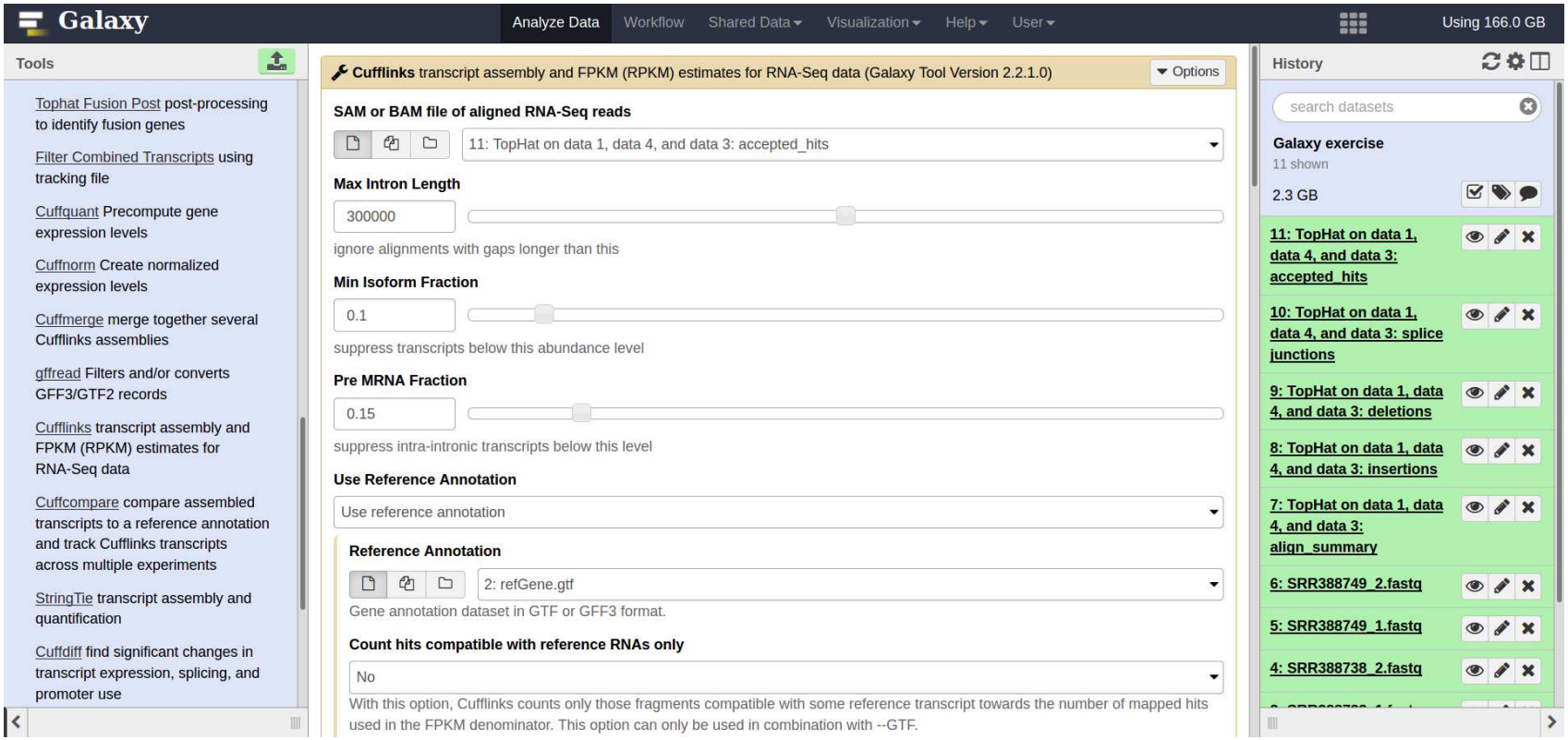
Std. Dev for Distance between Mate Pairs
20
--mate-std-dev; The standard deviation for the distribution on inner distances between mate pairs. The default is 20bp.

Report discordant pair alignments?
Yes
--no-discordant

Use a built in reference genome or own from your history
Use a built-in genome

History
search datasets
Galaxy exercise
6 shown
2.3 GB
- 6: SRR388749_2.fastq
- 5: SRR388749_1.fastq
- 4: SRR388738_2.fastq
- 3: SRR388738_1.fastq
- 2: refGene.gtf
- 1: clf.fna

Galaxy



While it's conceivable to generalize Galaxy for non-bioinformaticians, it's probably not worth the effort...

Snakemake

Published in 2012 in the context of bioinformatics:

BIOINFORMATICS APPLICATION NOTE

Vol. 28 no. 19 2012, pages 2520–2522
doi:10.1093/bioinformatics/bts480

Genome analysis

Advance Access publication August 20, 2012

Snakemake—a scalable bioinformatics workflow engine

Johannes Köster[1,2,*] and Sven Rahmann[1]

[1]Genome Informatics, Institute of Human Genetics, University of Duisburg-Essen and [2]Paediatric Oncology, University Childrens Hospital, 45147 Essen, Germany

Associate Editor: Alfonso Valencia

This (unfortunately) undersells it since users who want to do what Make does, would appreciate its Python-based syntax.

Outline

Rules

A Snakemake workflow consists of a Snakemake file (`Snakefile`) that consists of a set of rules.

Each rule specifies a series of directives, which includes at least:

1. input files,
2. output files, and
3. how to generate the output files from the input files

In many ways, this is similar to Make and Galaxy.

Sample Rule

Save the following in `Snakefile`:

```
rule HelloWorld:
    input: "Download/iris.data"
    output: "HelloWorld/iris.txt"
    shell:
        """
        ./helloworld {input} >{output}
        """
```

Note: Variables are enclosed in curly braces.

Sample Rule

Save the following in `Snakefile`:

```
rule HelloWorld:
    input: "Download/iris.data"
    output: "HelloWorld/iris.txt"
    shell:
        """
        ./helloworld {input} >{output}
        """
```

Note: Variables are enclosed in curly braces.

Sample Rule

Save the following in `Snakefile`:

```
rule HelloWorld:
    input: "Download/iris.data"
    output: "HelloWorld/iris.txt"
    shell:
        """
        ./helloworld {input} >{output}
        """
```

Note: Variables are enclosed in curly braces.

Sample Rule

Save the following in `Snakefile`:

```
rule HelloWorld:
    input: "Download/iris.data"
    output: "HelloWorld/iris.txt"
    shell:
        """
        ./helloworld {input} >{output}
        """
```

Note: Variables are enclosed in curly braces.

Sample Rule

Save the following in `Snakefile`:

```
rule HelloWorld:
    input: "Download/iris.data"
    output: "HelloWorld/iris.txt"
    shell:
        """
        ./helloworld {input} >{output}
        """
```

Note: Variables are enclosed in curly braces.

Sample Rule

Save the following in `Snakefile`:

```
rule HelloWorld:
    input: "Download/iris.data"
    output: "HelloWorld/iris.txt"
    shell:
        """
        ./helloworld {input} >{output}   ←
        """
```

Note: Variables are enclosed in curly braces.

Sample Rule

```
$ snakemake HelloWorld/iris.txt
```

```
Building DAG of jobs...
Provided cores: 1
Rules claiming more threads will be scaled down.
Job counts:
count jobs
1 HelloWorld
1

rule HelloWorld:
input: Download/iris.data
output: HelloWorld/iris.txt
jobid: 0

Finished job 0.
1 of 1 steps (100%) done
```

Sample Rule

If the input doesn't exist, then the command will not work:

```
$ snakemake HelloWorld/not-exist.txt
```

Building DAG of jobs...
MissingInputException in line 1 of Snakefile.py:
Missing input files for rule HelloWorld:
Download/not-exist.data

Not surprisingly, the input directive is a precondition for the rule to run.

Variable Substitution

The interesting part of Snakemake (IMHO) occurs during variable substitution; commands can then be generalized.

Variable Substitution

```
rule HelloWorld:
    input: "Download/{dataset}.data"
    output: "HelloWorld/{dataset}.txt"    ←
    shell:
        """
        ./helloworld {input} >{output}
        """
```

Variable Substitution

```
rule HelloWorld:
    input: "Download/{dataset}.data"      ←
    output: "HelloWorld/{dataset}.txt"
    shell:
        """
        ./helloworld {input} >{output}
        """
```

Variable Substitution

For example:

- `$ snakemake HelloWorld/abalone.txt`
- `$ snakemake HelloWorld/iris.txt`

will both work as expected.

Top-level Rule

Instead of typing `snakemake` followed by the output to be generated every time, like Make, there is a top-level rule called `all`.

It is special in that it has no output defined.

Top-level Rule

```
rule all:
    input: "HelloWorld/iris.txt"

rule HelloWorld:
    input: "Download/{dataset}.data"
    output: "HelloWorld/{dataset}.txt"    ←
    shell:
        """
        ./helloworld {input} >{output}
        """
```

Note: The `all` rule must come first.

Top-level Rule

```
rule all:
    input: "HelloWorld/iris.txt"

rule HelloWorld:
    input: "Download/{dataset}.data"        ←
    output: "HelloWorld/{dataset}.txt"
    shell:
        """
        ./helloworld {input} >{output}
        """
```

Note: The `all` rule must come first.

Top-level Rule

```
rule all:
    input: "HelloWorld/iris.txt"

rule HelloWorld:
    input: "Download/{dataset}.data"
    output: "HelloWorld/{dataset}.txt"
    shell:
        """
        ./helloworld {input} >{output}
        """
```

Note: The `all` rule must come first.

Top-level Rule

```
$ snakemake
```

```
Building DAG of jobs...
Provided cores: 1
Rules claiming more threads will be scaled down.
Job counts:
count jobs
1 HelloWorld
1 all
2

rule HelloWorld:
input: Download/iris.data
output: HelloWorld/iris.txt
jobid: 1
wildcards: dataset=iris

Finished job 1.
1 of 2 steps (50%) done

localrule all:
input: HelloWorld/iris.txt
jobid: 0

Finished job 0.
2 of 2 steps (100%) done
```

Multiple Arguments

```
rule HelloWorld:
    input: "Download/iris.data",
           "Download/abalone.data"
    output: "HelloWorld/iris.txt"
    shell:
        """
        ## Assuming helloworld takes two arguments
        ./helloworld {input} >{output}                    ←
        ./helloworld {input[0]} {input[1]} >{output}
        """
```

Multiple Arguments

```
rule HelloWorld:
    input: "Download/iris.data",
           "Download/abalone.data"
    output: "HelloWorld/iris.txt"
    shell:
        """
        ## Assuming helloworld takes two arguments
        ./helloworld {input} >{output}
        ./helloworld {input[0]} {input[1]} >{output}    ←
        """
```

Named Inputs/Outputs

```
rule HelloWorld:
    input:
        data1 = "Download/iris.data",
        data2 = "Download/abalone.data"
    output: "HelloWorld/iris.txt"
    shell:
        """
        ./helloworld {input.data2} >{output}    ←
        """
```

Parameters

Within a rule, you might need access to non-file parameters for some reason. This is where the params directive is needed:

```
rule HelloWorld:
    input: "Download/{dataset}.data"
    output: "HelloWorld/{dataset}.txt"
    params:
        count="42",
        dataset="{dataset}"
    shell:
        """
        ./helloworld -count {params.count} -label
            {params.dataset} {input} >{output}
        """
```

Parameters

Within a rule, you might need access to non-file parameters for some reason. This is where the params directive is needed:

```
rule HelloWorld:
    input: "Download/{dataset}.data"
    output: "HelloWorld/{dataset}.txt"
    params:
        count="42",                                  ←
        dataset="{dataset}"                          ←
    shell:
        """
        ./helloworld -count {params.count} -label ↵
            {params.dataset} {input} >{output}
        """
```

Parameters

Within a rule, you might need access to non-file parameters for some reason. This is where the params directive is needed:

```
rule HelloWorld:
    input: "Download/{dataset}.data"
    output: "HelloWorld/{dataset}.txt"
    params:
        count="42",
        dataset="{dataset}"
    shell:
        """
        ./helloworld -count {params.count} -label ↵   ←
            {params.dataset} {input} >{output}        ←
        """
```

Functions for Inputs

Instead of listing the inputs within a rule, a Python function that returns a list can be used. (Finally, some Python... ☺)

Functions cannot be defined for outputs, though.

Functions for Inputs

```
OUTPUT_DIR = "Download/"

def Get_Inputs (foo):
    my_inputs = []
    my_inputs.append (OUTPUT_DIR + "iris.data")
    my_inputs.append (OUTPUT_DIR + "abalone.data")
    print ("Data set: ", foo.dataset, "\n")
    return my_inputs

rule HelloWorld:
    input:
        Get_Inputs
    output: "HelloWorld/{dataset}.txt"
    shell:
        """
        ./helloworld {input[1]} >{output}
        """
```

←

Functions for Inputs

```
OUTPUT_DIR = "Download/"

def Get_Inputs (foo):
    my_inputs = []
    my_inputs.append (OUTPUT_DIR + "iris.data")
    my_inputs.append (OUTPUT_DIR + "abalone.data")
    print ("Data set: ", foo.dataset, "\n")
    return my_inputs

rule HelloWorld:
    input:
        Get_Inputs
    output: "HelloWorld/{dataset}.txt"
    shell:
        """
        ./helloworld {input[1]} >{output}
        """
```

Functions for Inputs

```
OUTPUT_DIR = "Download/"

def Get_Inputs (foo):
    my_inputs = []
    my_inputs.append (OUTPUT_DIR + "iris.data")
    my_inputs.append (OUTPUT_DIR + "abalone.data")
    print ("Data set: ", foo.dataset, "\n")
    return my_inputs

rule HelloWorld:
    input:
        Get_Inputs                                  ←
    output: "HelloWorld/{dataset}.txt"
    shell:
        """
        ./helloworld {input[1]} >{output}
        """
```

Functions for Inputs

```
OUTPUT_DIR = "Download/"

def Get_Inputs (foo):
    my_inputs = []
    my_inputs.append (OUTPUT_DIR + "iris.data")
    my_inputs.append (OUTPUT_DIR + "abalone.data")
    print ("Data set: ", foo.dataset, "\n")
    return my_inputs

rule HelloWorld:
    input:
        Get_Inputs
    output: "HelloWorld/{dataset}.txt"
    shell:
        """
        ./helloworld {input[1]} >{output}
        """
```

Functions for Inputs

```
OUTPUT_DIR = "Download/"

def Get_Inputs (foo):
    my_inputs = []
    my_inputs.append (OUTPUT_DIR + "iris.data")
    my_inputs.append (OUTPUT_DIR + "abalone.data")
    print ("Data set: ", foo.dataset, "\n")
    return my_inputs

rule HelloWorld:
    input:
        Get_Inputs
    output: "HelloWorld/{dataset}.txt"
    shell:
        """
        ./helloworld {input[1]} >{output}
        """
```

Functions for Inputs

```
OUTPUT_DIR = "Download/"

def Get_Inputs (foo):
    my_inputs = []
    my_inputs.append (OUTPUT_DIR + "iris.data")        ←
    my_inputs.append (OUTPUT_DIR + "abalone.data")     ←
    print ("Data set: ", foo.dataset, "\n")
    return my_inputs

rule HelloWorld:
    input:
        Get_Inputs
    output: "HelloWorld/{dataset}.txt"
    shell:
        """
        ./helloworld {input[1]} >{output}
        """
```

Functions for Inputs

```
OUTPUT_DIR = "Download/"

def Get_Inputs (foo):
    my_inputs = []
    my_inputs.append (OUTPUT_DIR + "iris.data")
    my_inputs.append (OUTPUT_DIR + "abalone.data")
    print ("Data set: ", foo.dataset, "\n")
    return my_inputs

rule HelloWorld:
    input:
        Get_Inputs
    output: "HelloWorld/{dataset}.txt"
    shell:
        """
        ./helloworld {input[1]} >{output}
        """
```

Functions for Inputs

```
OUTPUT_DIR = "Download/"

def Get_Inputs (foo):
    my_inputs = []
    my_inputs.append (OUTPUT_DIR + "iris.data")
    my_inputs.append (OUTPUT_DIR + "abalone.data")
    print ("Data set: ", foo.dataset, "\n")
    return my_inputs                                    ←

rule HelloWorld:
    input:
        Get_Inputs
    output: "HelloWorld/{dataset}.txt"
    shell:
        """
        ./helloworld {input[1]} >{output}
        """
```

Functions for Inputs

```
OUTPUT_DIR = "Download/"

def Get_Inputs (foo):
    my_inputs = []
    my_inputs.append (OUTPUT_DIR + "iris.data")
    my_inputs.append (OUTPUT_DIR + "abalone.data")
    print ("Data set: ", foo.dataset, "\n")
    return my_inputs

rule HelloWorld:
    input:
        Get_Inputs
    output: "HelloWorld/{dataset}.txt"
    shell:
        """
        ./helloworld {input[1]} >{output}
        """
```

←

Functions for Inputs

Here, we have:

- Definition of a simple function, with arguments from the rule passed in. (Also called wildcards in the documentation.)
- Use of a global variable.

Generating Output with Python

The shell directive in a rule states "how to generate the output files from the input files". The shell commands can be replaced with the run directive followed by Python code.

Suppose we wanted to count the number of lines in a file:

```
rule Use_WC:
    input: "Download/{dataset}.data"
    output: "Use_WC/{dataset}.txt"
    shell:
        """
        TMP_FILE=`mktemp`                    ←
        wc -l {input} >{output}
        rm -f ${{TMP_FILE}}
        """
```

Generating Output with Python

The shell directive in a rule states "how to generate the output files from the input files". The shell commands can be replaced with the run directive followed by Python code.

Suppose we wanted to count the number of lines in a file:

```
rule Use_WC:
    input: "Download/{dataset}.data"
    output: "Use_WC/{dataset}.txt"
    shell:
        """
        TMP_FILE=`mktemp`
        wc -l {input} >{output}
        rm -f ${{TMP_FILE}}
        """
```

←

Generating Output with Python

This rule will achieve the same thing:

```
rule Use_Python:
  input:
    input_fn = "Download/{dataset}.data"
  output:
    output_fn = "Use_Python/{dataset}.txt"
  run:
    in_fn = input.input_fn
    in_fp = open (in_fn, 'r')
    line_count = 0

    line = in_fp.readline ()
    while line != '':
      line_count += 1
      line = in_fp.readline ()
```

Generating Output with Python

This rule will achieve the same thing:

```
rule Use_Python:
    input:
        input_fn = "Download/{dataset}.data"        ←
    output:
        output_fn = "Use_Python/{dataset}.txt"
    run:
        in_fn = input.input_fn                      ←
        in_fp = open (in_fn, 'r')
        line_count = 0

        line = in_fp.readline ()
        while line != '':
            line_count += 1
            line = in_fp.readline ()
```

Generating Output with Python

```
in_fp.close ()

## Write out the value
out_fn = output.output_fn
out_fp = open (out_fn, 'w')
out_fp.write (str (line_count) + " " + in_fn + "\n")
out_fp.close ()
```

←

Outline

Including Files

The `Snakefile` can be organized better using the include directive:

include: "accessors.py"

Constraints

If a file specification has a mix of variables (i.e., `{dataset}-{count}`, then Snakemake cannot parse the filename `iris-2-42`. One solution is to employ constraints on variables:

```
wildcard_constraints:
    count = "\d+"
```

Python regular expressions are allowed. Thus, it might even be better to say `dataset` cannot have hyphens and use underscores instead.

Configuration File

Allows the specification of a YAML file, which is then read in as a global dictionary. If `config.yaml` looks like this:

```
datasets:
  iris:
    numcols: 5
```

Read it in like this in your `Snakefile`:

```
configfile: "config.yaml"
```

And then access it like this:

```
curr_sample = "iris"
numcols = config['datasets'][curr_sample]['numcols']
```

Suggestions

- Suggest each rule write to a different directory to avoid the possibility of a circular dependency.
- If a shell command in a rule returns a non-zero value, the rule fails and (for safety reasons) all outputs are deleted since they are inconsistent.
- Log files for programs should not be specified in the output directive.

Running Snakemake

- After running `Snakemake`, a "hidden" directory called `.snakemake` is created which keeps track of its progress. If you need to reset things, then just erase this directory.
- If a previous run was interrupted, then use the `--unlock` argument to remove the lock on the working directory.

Running Snakemake

In fact, Snakemake has many useful command-line arguments:

- `--configfile` Explicitly specify the configuration file.
- `--cores` Set the number of cores to use.
- `--printshellcmds` / `-p` Print shell commands as they are executed.
- `--dag` Output a directed acyclic graph in the dot language.
- `--dryrun` Perform a dry run.
- `--list` List rules in the Snakefile.
- `--forceall` / `-F` Force the execution of a rule and all subsequent rules.

Outline

Small Example

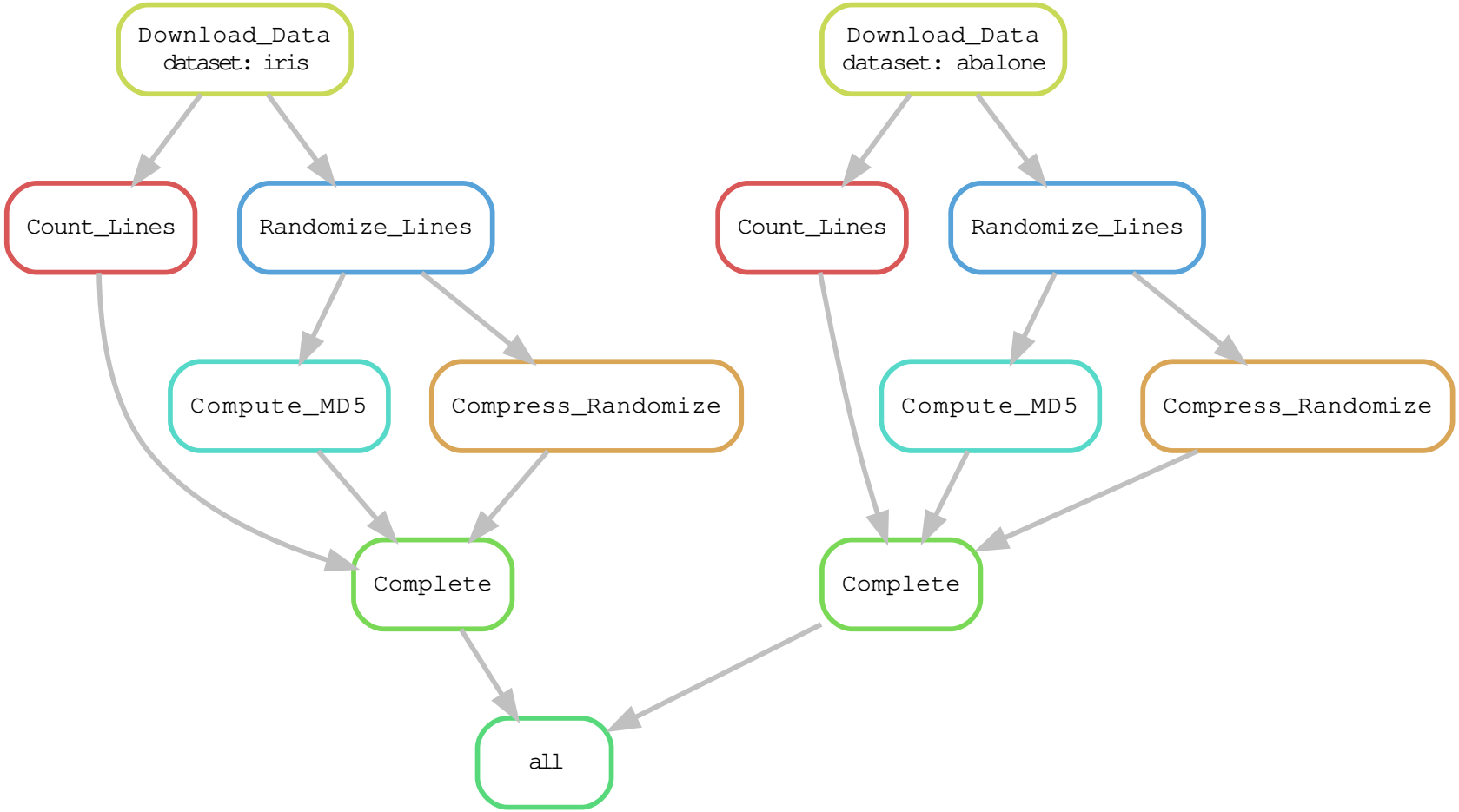
A small example has been made available on GitHub: http://www.github.com/rwanwork/Snakemake-Example/.

Performs simple tasks:

1. Download an example from the University of California of Irvine's Machine Learning Repository.
2. Count the number of lines in the data file.
3. Randomize the lines of the data file.
4. Compress the permuted data file.
5. Calculate the MD5 message digest of the permuted data file.

Small Example

The dependency graph with two data sets looks like this:



Other Works

Systems such as Biopipe (Hoon *et al.*, 2003), Taverna (Oinn *et al.*, 2004), Galaxy (Goecks *et al.*, 2010), GeneProf (Halbritter *et al.*, 2011) or PegaSys (Shah *et al.*, 2004) are easy to learn and use through their graphical user interface. Others such as Ruffus (Goodstadt, 2010), Pwrake (Tanaka and Tatebe, 2010), GXP Make (Taura *et al.*, 2010) and Bpipe (Sadedin *et al.*, 2012) use text-based definition of workflows, which can be advantageous: workflows can be edited without a graphical environment (e.g. directly on a remote server); and developers can collaborate on them through source code management tools. Similar to Pwrake and GXP Make, Snakemake is inspired by the build system GNU Make (Stallman and McGrath, 1991). They all infer the

(Figure source: From the Snakemake paper.)

Further Information

More information is available from the developer:

Documentation https://snakemake.readthedocs.io/en/stable/

Manuscript In Bioinformatics journal[2]

Source https://bitbucket.org/snakemake/snakemake.git

[2] https://academic.oup.com/bioinformatics/article/28/19/2520/290322