Introduction to Snakemake

Dillon Barker

2021-10-14

Lesson 1

1. Why Snakemake?
2. Introducing Workflows
3. Workflow Syntax
4. Running Snakemake
5. Assignment 1

Preparation for Assignment 1:

```
sbatch -c 1 --mem 4G --wrap \
"conda create -n smk-lesson-1 -c conda-forge -c bioconda prokka pirate snakemake"
```

Why Snakemake?

Automation

Reproducibility

Others' Snakefiles

Front-loading your effort.

Modest investment at the beginning of a project yields a hands-off tool for performing routine analyses.

Why Snakemake?

Automation

Reproducibility

Others' Snakefiles

Guarantee that the same inputs will give the same outputs.

Altering any input will make Snakemake re-evalutate the outputs.

Built-in version tracking.

Why Snakemake?

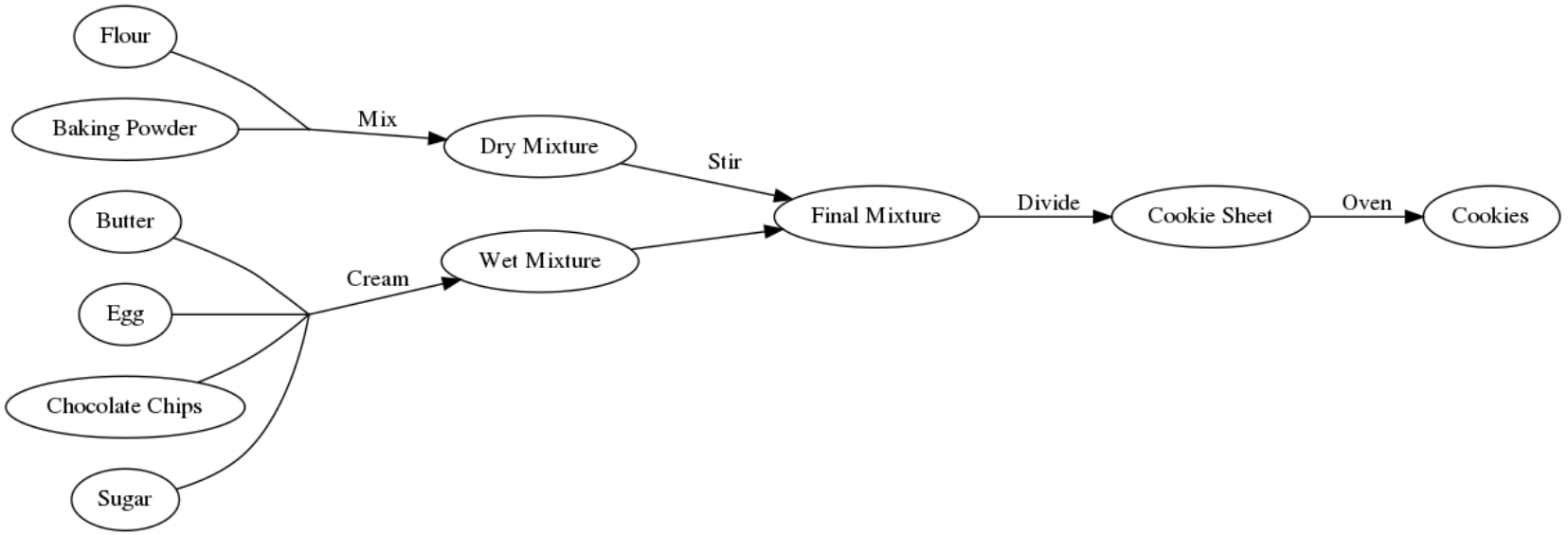
Automation

Reproducibility

Others' Snakefiles

Understand and modify the tools others have created for you.

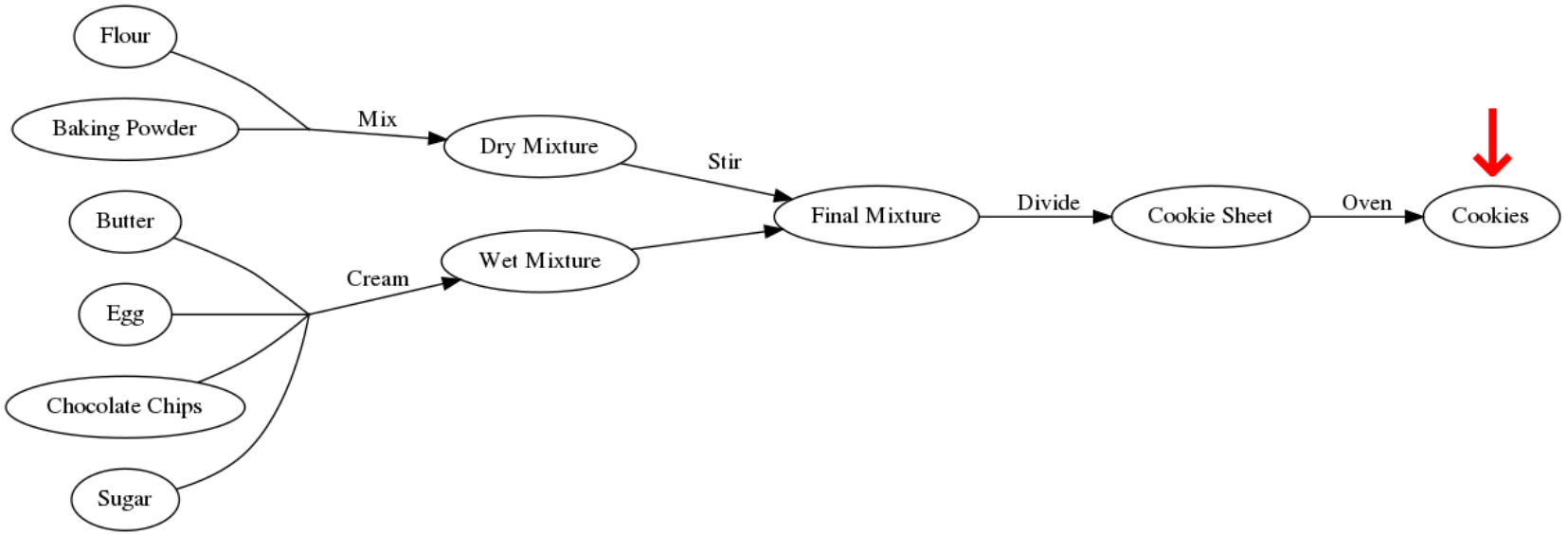
Baking with Graphs



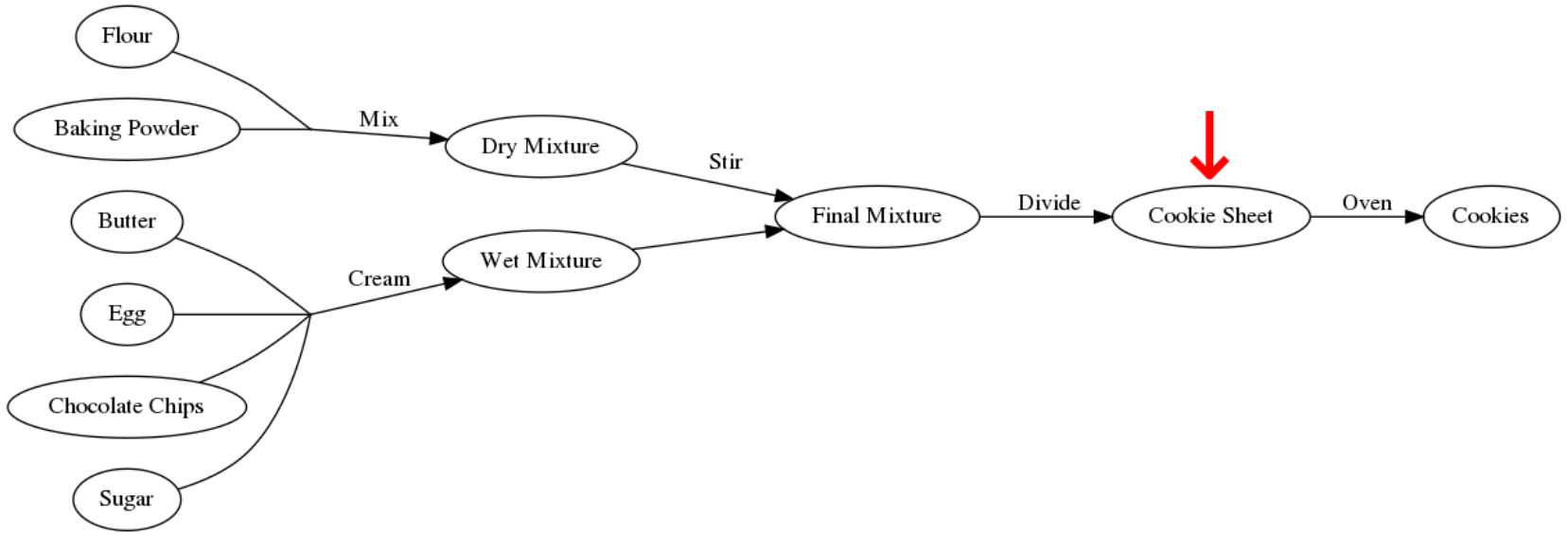
Thinking Backwards

- Snakemake figures out how to achieve the desired result:
  - starts at the final product
  - works backwards until it finds what it needs
- A collection of dependencies not a sequence of instructions
- You tell it how to convert each input to each output

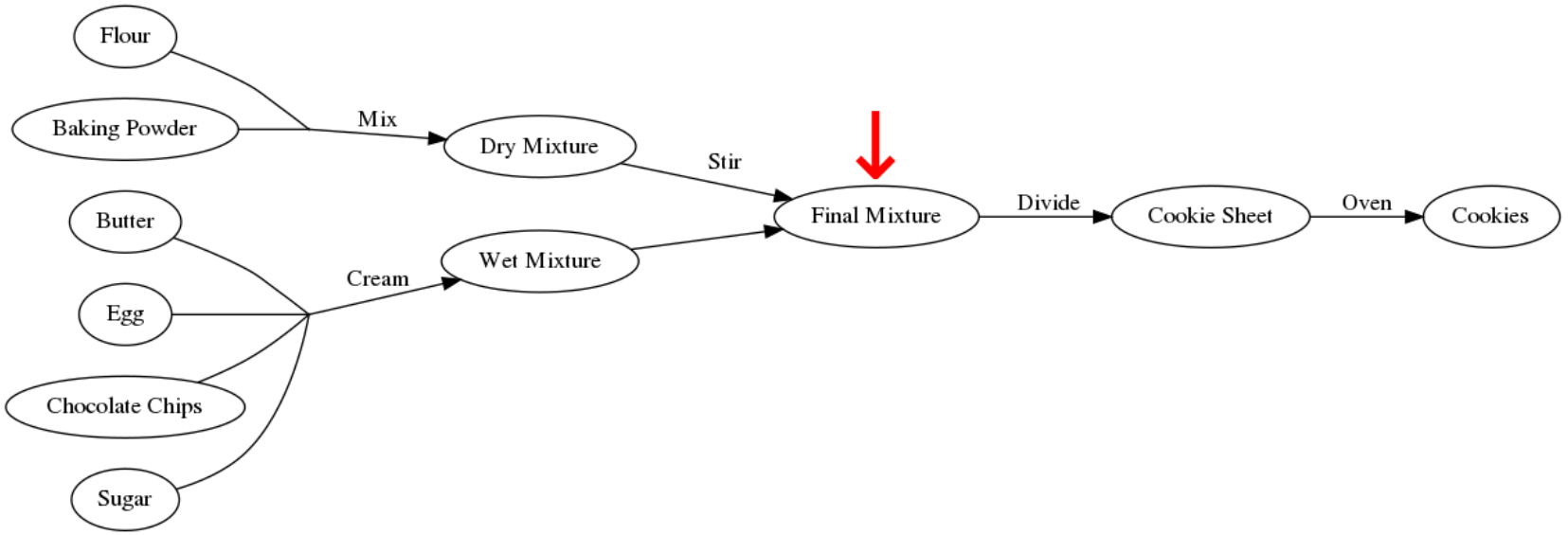
Thinking Backwards



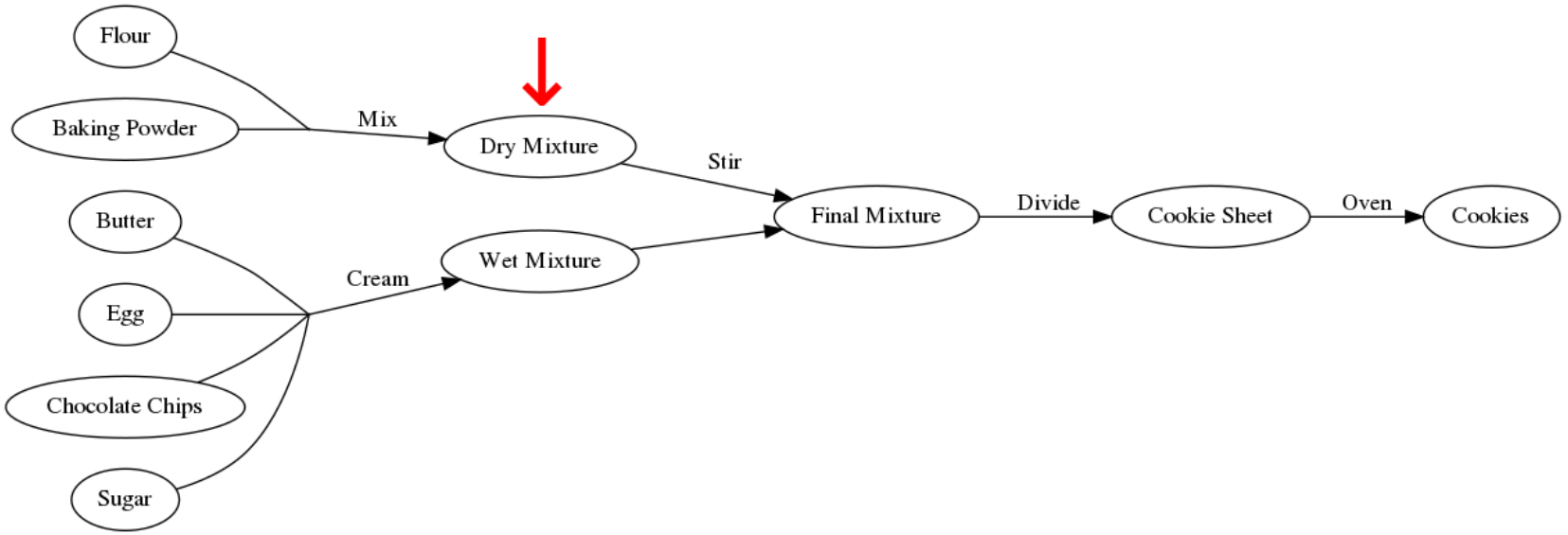
Thinking Backwards



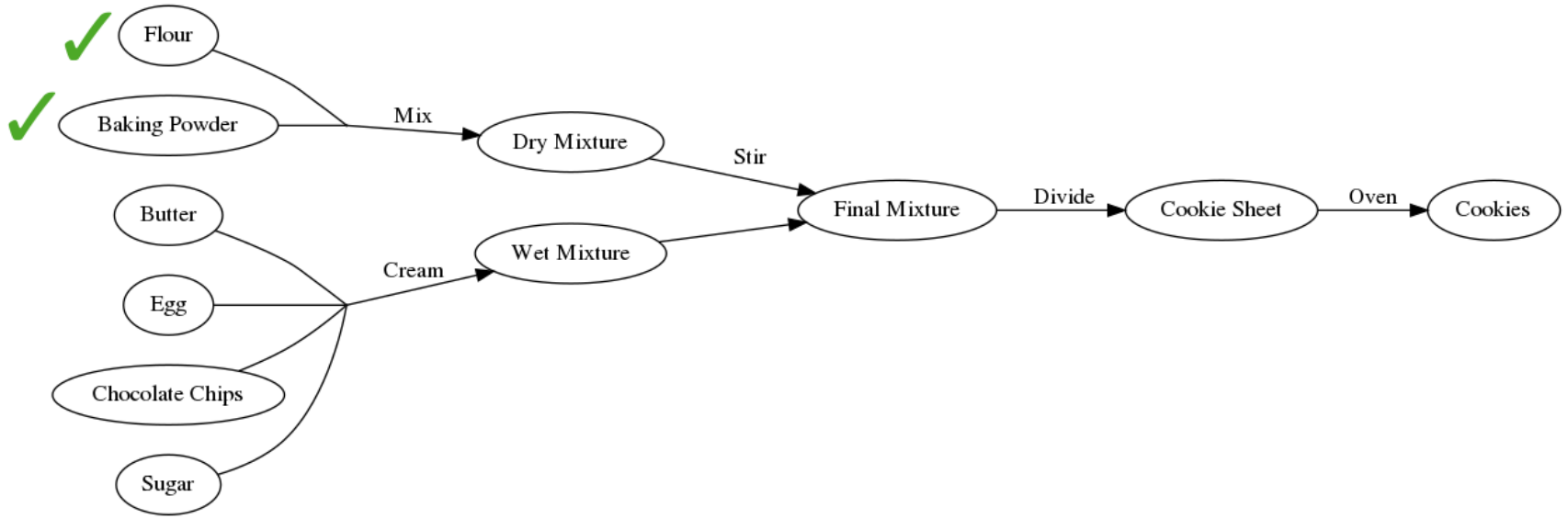
Thinking Backwards



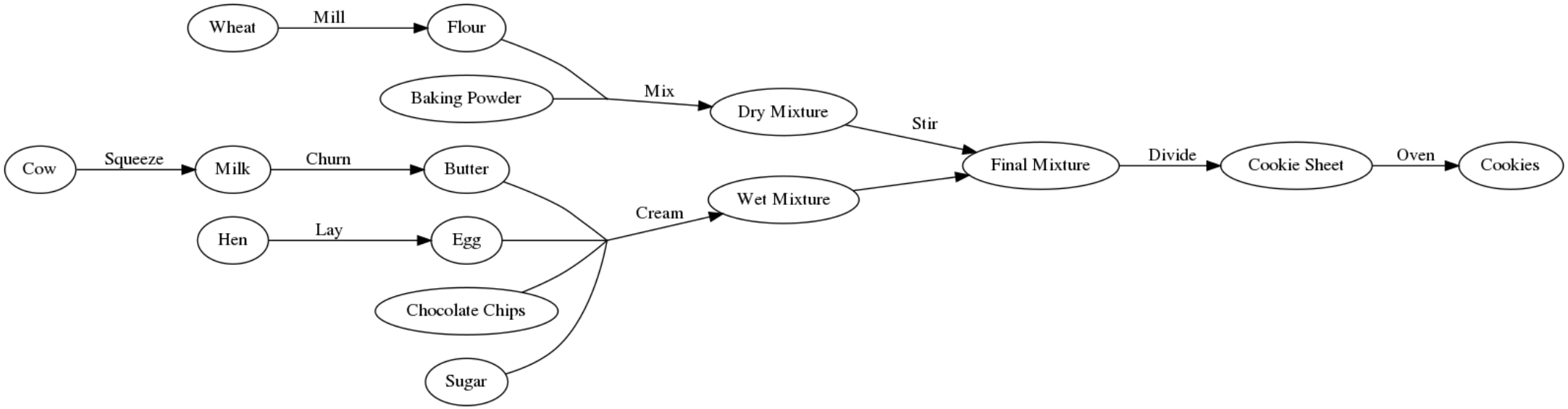
Thinking Backwards



Thinking Backwards



Thinking Backwards



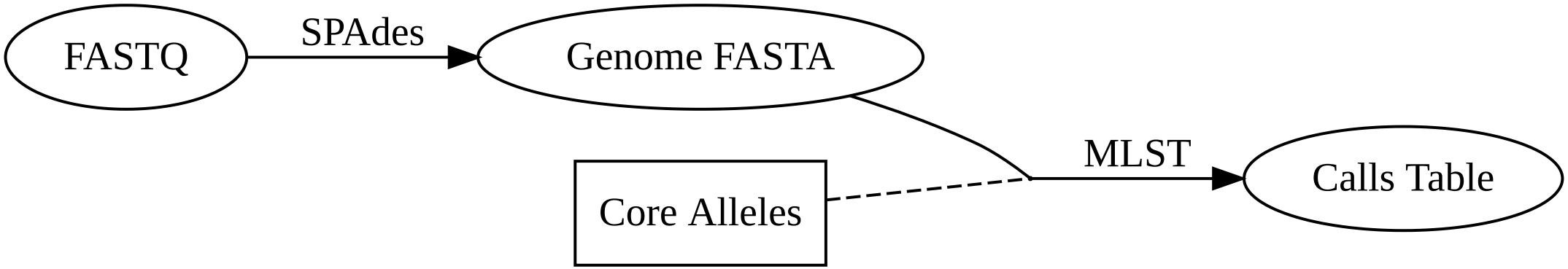
Rules for Baking

```
rule all:
    input: "cookies"

rule bake_cookies:
    input: "pan/dough.blobs"
    output: "cookies"
    shell: "oven -i {input} -o {output} --temp 350 --time 15"

rule apportion_dough_blobs:
    input: "bowls/final.mix"
    output: "pan/dough.blobs"
    shell: "scoop -n 24 {input} > {output}"

rule combine_bowls:
    input: wet="bowls/wet.mix", dry="bowls/dry.mix"
    output: "bowls/final.mix"
    shell: "mixer {input.wet} {input.dry} > {output}"
```

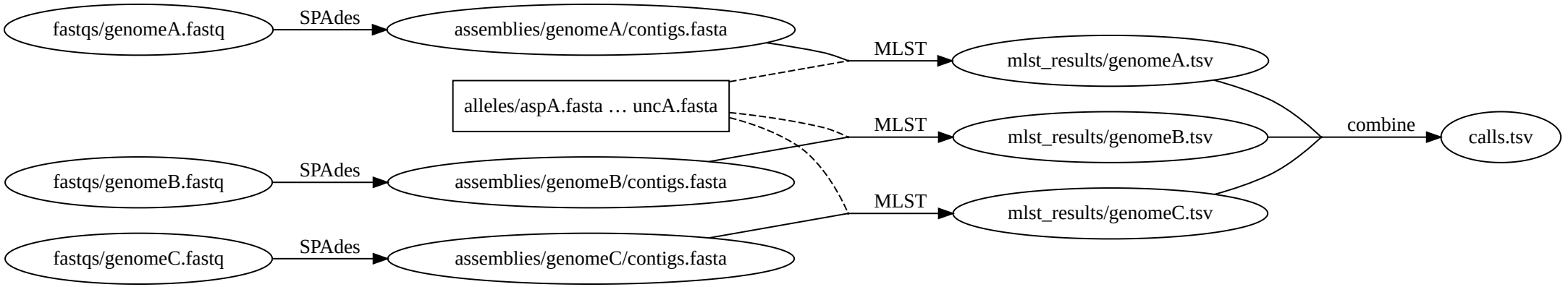
Rules for Baking

- The `all` rule is the final target, and written first
- Each rule specifies its `input` and `output`
- The `shell` command defines how the `input` becomes the `output`
- Snakemake matches up `input` and `output` for all the rules until it can generate `all`

Baking → Bioinformatics



Multiple Samples



Multiple Samples

- Recall that rules are only executed if their inputs update or outputs are missing
- Lets you run jobs without worrying you'll waste effort

MLST Example from Above

1. If you have 1000 genomes and run the above MLST rule, it will calculate 1000 calls, and build the calls table
2. If you add 1 more genome, and rerun snakemake, only a single MLST will run, and the calls table is rebuilt

Wildcards

- We can match every file with particular naming pattern with wildcards
- In a rule, wrap a variable name with curly braces
  - *e.g.* `{sample}`
- Rule will be applied in parallel to each file matching the rule
- In `shell` block, you can access these when preceded by `wildcards`
  - *e.g.* `{wildcards.sample}`

Rules for Multiple Samples

```
from pathlib import Path
sample_names = [fq.stem for fq in Path("fastqs").glob("*")]

rule all:
    input: "calls.tsv"

rule assemble:
    input:
        fwd="fastqs/{sample}/{sample}_1.fastq", rev="fastqs/{sample}/{sample}_2.fastq"
    output: "assemblies/{sample}/contigs.fasta"
    shell: "spades -1 {input.fwd} -2 {input.rev} -o assemblies/{wildcards.sample}"

rule get_sequence_types:
    input: "assemblies/{sample}/contigs.fasta"
    output: "mlst_results/{sample}.tsv"
    shell: "mlst --scheme campylobacter {input} > {output}"

rule combine_mlst_results:
    input: expand("mlst_results/{sample}.tsv", sample=sample_names)
    output: "calls.tsv"
    shell: "cat {input} > {output}"
```

Threads

- Many (but not all!) bioinformatics tools use multiple CPU threads
- `threads` directive defaults to `1`
  - Accessible in the `shell` block, similar to `input` and `output`
    - `{threads}`

```
rule annotate_genome:
    input: "genomes/{sample}.fasta"
    output: "annotations/{sample}/{sample}.gff"
    threads: 8
    shell: "prokka --cpus {threads} -o annotations/{wildcards.sample} {input}"
```

Caveats and Assumptions

Directory Structure

- Snakefiles themselves can live anywhere
  - I keep mine in `~/snakefiles/`
- Workflows tightly coupled to any directory structure described within
- Will implicitly create any directories it needs
  - No need for `mkdir`

Directory Structure

This rule...

```
rule annotate_genome:
    input: "genomes/{sample}.fasta"
    output: "annotations/{sample}/{sample}.gff"
    threads: 8
    shell: "prokka --cpus {threads} -o annotations/{wildcards.sample} {input}"
```

requires this structure:

```
analysis/
├── annotations/
│   ├── isolateA.fasta
│   ├── isolateB.fasta
│   └── isolateC.fasta
└── genomes/
    ├── isolateA.fasta
    ├── isolateB.fasta
    └── isolateC.fasta
```

Caveats and Assumptions

Independent Jobs

- Failure of any job will abort all other jobs
  - Override with `--keep-going`
  - Dependent jobs will still await all inputs
  - *e.g.* if stiring the dry cookie mixture fails, the wet mixture still gets made, but nothing goes in the oven

Running Snakemake (Basic)

The **basic invocation** of Snakemake:

```
snakemake --jobs <number of parallel jobs> -s <path to your Snakefile> -d <work directory>
```

Example populated with real values:

```
snakemake --jobs 5 -s ~/snakefiles/assemble.smk -d ~/Projects/cj_population_study
```

Running Snakemake on Waffles

- Snakemake can be run on HPCs like **Waffles**
  - *Must* be combined with Slurm
  - *Don't run it on the head node!*
- Two parts:
  1. Tell Snakemake how to submit jobs with `--cluster`
  2. Submit Snakemake itself as a Slurm job

Running Snakemake on Waffles

The `--cluster` argument:

- Create a template command to pass to **Slurm**
- May access Snakemake special variables like `{threads}`
  - **More on this later**

```
--cluster 'sbatch -c {threads} --mem 12G --partition NMLResearch '
```

Running Snakemake on Waffles

Submitting the Snakemake job to Slurm:

```
sbatch -c 1 --mem 4G --wrap "snakemake --jobs 5 -s
~/snakefiles/assemble.smk -d ~/Projects/cj_population_study --cluster
'sbatch -c {threads} --mem 12G --partition NMLResearch '"
```

Assignment 1

Write a Snakemake workflow that does the following:

1. Run Prokka on each genome
2. Symlink GFF annotations into `gffs/`
3. Build a pangenome with PIRATE

```
conda activate smk-lesson-1
```

Lesson 2

Lesson 2

1. Assignment 1 Answers
2. Conda Integration
3. Params and Threads
4. Mixing in Python
5. Assignment 2

Assignment 1

Conda Integration

- Snakemake can manage conda directly
- No need to manually build or activate conda environments

Conda directive

```
rule annotate_genome:
    input: "genomes/{sample}.fasta"
    output: "annotations/{sample}/{sample}.gff"
    conda: "envs/prokka.yaml"
    shell: "prokka --cpus {threads} -o annotations/{wildcards.sample} {input}"
```

Conda YAML files

- Placed **relative to the Snakefile**, *not* the project directory

```
# annotate.smk
rule annotate_genome:
    input: "genomes/{sample}.fasta"
    output: "annotations/{sample}/{sample}.gff"
    conda: "envs/prokka.yaml"
    shell: "prokka --cpus {threads} -o annotations/{wildcards.sample} {input}"
```

The above will look for the following directory structure:

```
snakefiles/
├── annotate.smk
└── envs
    └── prokka.yaml
```

Conda YAML files

This YAML file ...

```
name: prokka
channels:
    - conda-forge
    - bioconda
    - defaults
dependencies:
    - prokka
```

... is equivalent to this conda command:

```
conda create -n prokka -c conda-forge -c bioconda -c defaults prokka
```

Using Conda Directives with Snakemake

- Must explicitly tell Snakemake to use Conda

```
snakemake --use-conda <…>
```

- Automatic installation and activation

Config

Configuration is possible through `config`

- Python `dict` available within the Snakefile
- Available through two methods
  - `--config` passes arguments directly via command line
  - `--configfile` points to a YAML file that provides values

`--config "key=value"` is equivalent to `--configfile config.yaml` where...

```
# config.yaml
key: "value"
```

Configuration via:

`--config` flag:

- ↓ effort
- ↑ flexible
- ↓ reproducible

YAML file:

- ↑ effort
- ↓ flexible
- ↑ reproducible

Params

- Non-file parameters may be provided in the `params` directive

Abusing Params to Fine-tune Resources

```
snakemake <…> --cluster 'sbatch -c {threads} --mem {params.mem} --time {params.time} '
```

```
rule annotate_genome:
    input: "genomes/{sample}.fasta"
    output: "annotations/{sample}/{sample}.gff"
    threads: 8
    params:
        time="45:00",
        mem="16G"
    shell: "prokka --cpus {threads} -o annotations/{wildcards.sample} {input}"

rule symlink_gff:
    input: "annotations/{sample}/{sample}.gff"
    output: "gffs/{sample}.gff"
    threads: 1
    params:
        time="01:00",
        mem="100M"
    shell: "ln -sr {input} {output}"
```

Config vs Params

- Params are fairly "fixed"
  - Used primarily to simplify `shell` block
- Config for run-specific information
  - *e.g.* providing a particular host database to `kat` or training file to `chewBBACA`

Mixing in Python

- Python may be mixed in arbitrarily into Snakemake
  - *i.e.* All Python is valid Snakemake
- Two main ways of using Python in Snakemake
  - `run` blocks
  - Python used directly in the Snakemake file

Python → Snakemake, get it?

Run blocks

- `run` blocks can be used in place of `shell` blocks
- Write Python inside the `run` block, rather than Bash in a `shell` block
- May access snakemake values like `input` and `output`

```
rule transpose_table:
    input: "data/results_table.csv"
    output: "data/results_table_transposed.csv"
    run:
        import pandas as pd
        original = pd.read_csv(input[0], header=0)
        transposed = original.transpose()
        transposed.to_csv(output[0], header=False)
```

Directly Using Python in Snakemake

- You can directly use Python in Snakemake
- Particularly useful for handling cases where a rule generates variable output
  - *e.g.* The number of gene FASTAs generated by a pangenome analysis
- Can provide a Python function to `input` instead of a file pattern

- `select_high_quality_genomes` takes a list of FASTAs, then symlinks high-quality ones into `./good_genomes/` and writes a report called `quality_report.txt`
- We don't know in advance which genomes will pass QC, so we need an input function

```
rule quality_filter_genomes:
    input: expand("genomes/{sample}.fasta", sample=samples)
    output: "quality_report.txt"
    shell: "select_high_quality_genomes {input} > {output}"

# input functions need to take parameter `wildcards`
def collect_good_genome_sample_names(wildcards):
    good_genomes = Path("good_genomes/").glob("*.fasta")
    return list(good_genomes)

# use the report as a dummy input to make sure quality_filter_genomes executes
rule run_abricate:
    input: report="quality_report.txt", fastas=collect_good_genome_sample_names
    output: "amr_results.tsv"
    shell: "abricate {input.fastas} > {output}"
```

Assignment 2

- Build upon Assignment 1

1. Create conda YAMLs for `prokka` and `pirate`
2. Give appropriate resources to each rule with `params`
3. Write a rule with a `run` block that reads `PIRATE.gene_families.tsv`, finds loci present in 100% of genomes, and writes their names to a text file
    - columns of interest: `gene_family` & `number_genomes`
4. A rule that:
    - Uses an input function reads selected genes from the text file in **Part 3**
    - Symlinks these into a directory called `loci`
        - Either `shell` or `run` at your preference

Assignment 2 Hints

`pandas` for easily reading tabular files

```
import pandas as pd
data_table = pd.read_csv(input[0], sep = "\t")
# select rows from columnA where columnC is greater than 42
selected_rows = data_table["columnC"] > 42
selected_columnA = data_table["columnA"].loc[selected_rows]
```

Creating symlinks from a list of file basenames

```
# list_of_names = ["larry", "moe", "curly"]
import os
for name in list_of_names:
    src = f"originals/{name}.txt"
    dst = f"filtered/{name}.txt"
    os.symlink(src, dst)
```

Assignment 2 Hints

Reading a text file into a list with Python

- Consider combining functions like this with `expand()`
  - `expand("path/to/{sample}.txt", sample=read_list_to_list())`

```
def read_lines_to_list(path: str):
    lines = []
    with open(path, "r") as f:
        for line in f:
            trimmed_line = line.strip()
            lines.append(trimmed_line)
    return lines
```

Reading a text file, but using at as an input function

```
def aggregate_files(path: str):
    lines = []
    with open(path, "r") as f:
        for line in f:
            trimmed_line = line.strip()
            lines.append(trimmed_line)
    lines_with_paths = [f"path/to/{sample}.txt" for sample in lines]
    return lines_with_paths
```

```
rule do_something:
    input: aggregate_files
    output: "somefile.txt"
    shell: "my_program {input} -o {output}"
```

Lesson 3

Lesson 3

1. Assignment 2 Answer
2. Fixing when things go wrong
3. Priorities
4. Script files (Python, R, and Julia)
5. Visualizing Workflows
6. Clinic

Assignment 2

When Things Go Wrong

- Snakemake locks its working directory when running
  - Prevents other snakemake instances from running in the same place
- Snakemake removes the lock when it completes (success *or* failure)
- Lockfile may not be removed when if snakemake crashes or is killed by slurm
  - `scancel --user your_username`

```
Unable to lock working directory.
```

Unlocking a Stale Lock

The Correct Way: `--unlock`

```
sbatch -c 1 --mem 2G --wrap "snakemake --unlock -s path/to/your/workflow.smk -j 1"
```

☠☠☠ Nuclear Option ☠☠☠

```
rm -r ./.snakemake
```

Priorities

- Rule priorities may be set
- These **are not** the same as slurm partitions
- Only determine the priority of execution within the workflow
- Rules default to priority `0`
- Bigger numbers are higher priority

```
rule my_example:
  input: ...
  output: ...
  priority: 50
  shell: ...
```

Script Files

- Can be used like a `run` block, but stored in another file
  - Path is specified relative to the workflow
- Available languages:
  - Python
  - R
  - Julia
- Snakemake will use special variables in the script
  - These scripts won't be general purpose!
  - Tied to snakemake

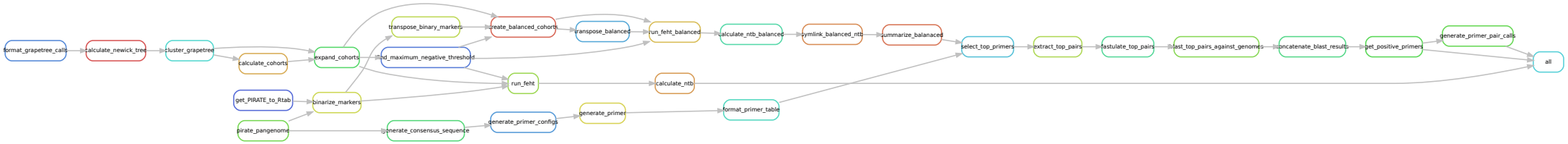
Script Files

```
# workflow.smk
rule scriptfile_example:
    input:
        "path/to/inputfile",
        "path/to/other/inputfile"
    output:
        "path/to/outputfile",
        "path/to/another/outputfile"
    script:
        "scripts/script.py"
```

```
# scripts/script.py
def do_something(data_path, out_path, threads, myparam):
    # python code here

do_something(snakemake.input[0], snakemake.output[0],
             snakemake.threads, snakemake.config["myparam"])
```

Visualizing Workflows

```
# Shows just the abstract graph of rules
snakemake -s your/workflow.smk --rulegraph | dot -Tpng > workflow.png
# OR
# Show every sample's journey through the workflow
snakemake -s your/workflow.smk --dag | dot -Tpng > workflow.png
```



Clinic

Please feel free to ask any questions!