Introduction to Snakemake

Dillon Barker

2021-10-20

class: inverse, center, middle name: lesson2

Lesson 2

2021-10-20

Lesson 2

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

```
from pathlib import Path
samples = [p.stem for p in Path("genomes").glob("*.fasta")]

rule all:
    input: "pangenome/PIRATE.gene_families.tsv"

rule annotate:
    input: "genomes/{sample}.fasta"
    output: "annotations/{sample}/{sample}.gff"
    threads: 8
    shell:
        "prokka --force --cpus {threads} "
        "--prefix {wildcards.sample} --outdir annotations/{wildcards.sample} "
        "{input}"

rule symlink_gffS:
    input: "annotations/{sample}/{sample}.gff"
    output: "gffs/{sample}.gff"
    threads: 1
    shell: "ln -sr {input} {output}"

rule pangenome:
    input: expand("gffs/{sample}.gff", sample=samples)
    output: "pangenome/PIRATE.gene_families.tsv"
    threads: 8
    shell: "PIRATE --input gffs/ --output pangenome/ --nucl --threads {threads}"
```

Conda Integration

- Snakemake can manage `conda` directly
    - No need to manually build or activate conda environments like in Lesson 1

Conda directive

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

- Available through two methods
    - `--config` passes arguments directly via command line
    - `--configfile` points to a YAML file that provides values

`--config key="value" number=5` is equivalent to `--configfile config.yaml` where...

```yaml
# config.yaml
key: "value"
number: 5
```

- Python `dict` available within the Snakefile
    - Access as `config["key"]` inside the workflow

Configuration via:

`--config` flag:

- ↓ effort
- ↑ flexible
- ↓ reproducible

YAML file:

- ↑ effort
- ↓ flexible
- ↑ reproducible

Params

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

```
rule annotate:
    input: "genomes/{sample}.fasta"
    output: "annotations/{sample}/{sample}.gff"
    threads: 8
    params: outdir="annotations/{sample}"
    shell:
        "prokka --force --cpus {threads} "
        "--prefix {wildcards.sample} --outdir {params.outdir} "
        "{input}"
```

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 --force --prefix {wildcards.sample} "
        "--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`

Local Rules

- Not every job is worth submitting as its own job to the cluster
    - Undemanding jobs, like symlinking files or the `all` rule
- Rules can be marked as **local**
    - Run in the same process as `snakemake`
- List rule names in `localrules` directive

Local Rules

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

localrules: all, symlink_fastas

rule all:
    input: expand("genomes/{sample}.fasta", sample=samples)

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 symlink_fastas:
    input: "assemblies/{sample}/contigs.fasta"
    output: "genomes/{sample}.fasta"
    shell: "ln -sr {input} {output}"
```

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 - Building On 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. Provides a GBK file to prokka's `--proteins` argument via `--config` or `--configfile`
5. Change your symlinking rule to be local

Assignment 2 Hints

pandas for easily reading and writing 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]
selected_columnA.to_csv(output[0], header=False)
```

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_lines_to_list())`

```python
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
```