Pipelines and Workflows
Software Engineering for Scientists

https://github.com/swe4s/lectures/tree/master/src/workflow/snakemake

The process of transforming data into a scientific result can involve many steps. These pipelines can be fragile and difficult to reproduce if executed manually. Workflow languages make this process more robust.

Install `snakemake`

```
$ conda activate swe4s
(swe4s) $ conda install -c conda-forge -c bioconda snakemake graphviz
```

Rules

A `snakemake` workflow is specified in a text file named `Snakefile` that contains a series of rules. Each rule is a small step in the process that takes a set of input files and runs a shell command to produce a set of output files.

From the `Snakefile`, `snakemake` will determine the dependencies between the rules by matching input files and output files and will execute the rules in an order that satisfies those dependencies. For example, if `rule A` has `foo.txt` as an input file, then `rule A` cannot execute until `foo.txt` exists. If `rule B` has `foo.txt` as an output file, then `rule B` may need to run before `rule A`. By tieing dependencies to files and not rules, `snakemake` can prevent wasted execution. If `foot.txt` already exists, then `rule B` does not need to run.

The input files, output files, and shell commands are specified in `input:`, `output:`, and `shell:` directives. The input and output directives contain Python lists of strings that resolve to paths to files. The shell directive contains a Python string that resolves to a bash command.

In the simple workflow below, a `plot.py` takes `foo.txt` as input and crates `foo.png`.

`Snakefile`

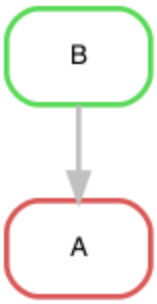
```
rule all:
  input:  "foo.png"

rule A:
  input:  "foo.txt"
  Output: "foo.png"
  shell:
    "python plot.py foo.txt foo.png"

rule B:
  output: "foo.txt"
  shell:
    "seq 1 100 > foo.txt"
```

The dependency graph can be visualized

```
$ snakemake --dag | dot -Tpng > dag.png
```



The steps that would be run in a workflow can also be inspected. When `--dryrun` is given none of the shell commands are actually exected.

```
$ snakemake --dryrun --printshellcmds
```

```
rule B:
    output: foo.txt
    jobid: 1

seq 1 100 > foo.txt

rule A:
    input: foo.txt
    output: foo.png
    jobid: 0

python plot.py foo.txt foo.png
Job counts:
      count jobs
      1     A
      1     B
      2
```

Once you are ready to execute the pipeline just run the `snakemake` command. In the output, yellow and green are good and red is bad.

```
$ snakemake

Provided cores: 1
Rules claiming more threads will be scaled down.
Job counts:
      count jobs
      1     A
      1     B
      2

rule B:
    output: foo.txt
    jobid: 1

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

rule A:
    input: foo.txt
    output: foo.png
    jobid: 0

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

If we look at the dependency graph again, we can see that it has been updated to show the rules whose dependencies have been met (output files exist) and do not need to be run.

```
$ snakemake --dag | dot -Tpng > dag.png
```

Specifying the runtime environment

`snakemake` rules also have a `conda:` directive that specifies an environment for the commands in the `shell:` directive. To use this feature, simply provide the path to a conda environment configuration file in the `conda:` directive. These files are in the `.yml` format and have a `channels:` section where you can give specific channels if you need packages that are not in the default channel, and a `dependencies:` section where you give the package names. While it is not required, specifying the version for these packages is highly recommended.

`Snakefile`

```
rule A:
  input:
    "foo.txt"
  output:
    "foo.png"
  conda:
    "env.yml"
  shell:
    "python plot.py foo.txt foo.png"
```

`env.yml`

```
channels:
  - conda-forge
  - bioconda
dependencies:
  - python=3.6.9
  - matplotlib=3.1.1
```

With the conda: directive and the conda environment file, pass the `--use-conda` flag to `snakemake`

```
$ snakemake --use-conda
```

Example workflow

Suppose we want to look at a histogram of the fires in different countries. First, we need to download the file containing the data.

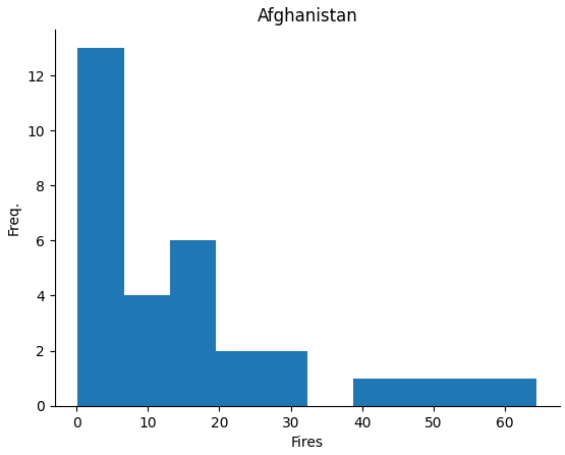
```
$ wget -O Argrofood_co2_emission.csv
'https://docs.google.com/uc?export=download&id=1Wytf3ryf9EtOwaloms8HEzLG0yjtRqxr'
```

Next, we need to extract the fire data for a given country and store it in a file.

```
$ python get_data.py Argrofood_co2_emission.csv Afghanistan Afghanistan.txt
```

Last we want to make a histogram of those counts. `hist.py` takes a file with one number per line and creates a histogram of that data.

```
$ python hist.py Afghanistan.txt Afghanistan Afghanistan.png
```



`snakemake` specification

A simple rule for running `get_data.py` would be

```
COUNTRY="Afghanistan"

rule all:
  input:
    COUNTRY + '.txt'

rule country_fire_counts:
  input:
    'Argrofood_co2_emission.csv'
  output:
    COUNTRY + '.txt'
  Shell:
    "python get_data.py {input} " + COUNTRY + " {output}"
```

Here we take advantage of variables. `COUTNRY` is a standard python variable, and it is concatenated with several directives so that the gene name does not need to be hardcoded in the rules. There are two other variables, `{input}` and `{output}` that resolve to the values of the input and output directives.

NOTE: `snakemake` abuses the strings wrapped in curly brackets syntax. In some instances they are variables and in others they are wildcards.

The issue with this rule is that it can only create one country fire counts file, but to satisfy the dependencies for the second graph `get_gene_counts.py` must run twice (for SDHB and MUTYH). To make the rule work for any number of genes we need to move from a scalar `GENE` value to a `GENES` array, let the output directive resolve to a list of files (one for each element in `GENES`), and let the shell directive run a command for each gene in `GENE`.

The input and output directives need to resolve to a list of files. To make this easier, `snakemake` provides the `expand()` function that takes a string containing variables (strings wrapped in curly brackets) and the values of those variables and returns all possible version of that string. For example,

```
GENES=['SDHB', 'MUTYH']
expand('{gene}_counts.txt', gene=GENES)
```

yields

```
['SDHB_counts.txt', 'MUTYH_counts.txt']
```

In this case, the expand function allows the number of files in the output directive to match the size of the GENES array

The shell directive executes bash commands, and a `for` loop is the most straightforward way of controlling how many times (and with what parameters) a command is executed. Constructing the shell directive can be confusing because a Python string is encoding a bash command, and the two languages have different variable syntaxes (bash variables begin with a `$`). To further complicate matters, Python variables can also be encoded as strings wrapped in curly brackets (e.g., `{GENES}` and `{input}`). We will color code variables in the next few examples to clarify which strings are variables and what type they are. Bash variables will be blue and Python variables will be red.

```
GENES = ["SDHB", "MUTYH"]

rule all:
  input:
    expand('{gene}_counts.txt', gene=GENES)

rule gene_sample_counts:
  input:
    'GTEx_Analysis_2017-06-05_v8_RNASeQCv1.1.9_gene_reads.acmg_59.gct.gz'
  output:
    expand('{gene}_counts.txt', gene=GENES)
  shell:
    'for gene in {GENES}; do ' \
    +  'python get_gene_counts.py {input} $gene $gene\_counts.txt;' \
    + 'done'
```

The same can be done for `get_tissue_samples.py`

```
TISSUES = ["Blood", "Brain"]

rule tissue_samples:
  input:
    'GTEx_Analysis_v8_Annotations_SampleAttributesDS.txt'
  output:
    expand("{tissue}_samples.txt", tissue=TISSUES)
  shell:
    'for tissue in {TISSUES}; do' \
    +  'python get_tissue_samples.py {input} $tissue $tissue\_samples.txt;' \
    + 'done'
```

The last step in the workflow is to run the `hist.py` script, which takes a series of gene/tissue pairs and an output file name and produces a plot. The input parameters can be easily encoded as an array of gene/tissue pairs, and the output file name can just be a simple joining of the pairs. In the example below the array of pairs `[ ['SDHB','Blood'], ['SDHB','Brain'] ]` will have the output file name `SDHB-Blood_SDHB-Brain.png`.

```
TISSUE_GENE_PAIRS = [ ['SDHB','Blood'], ['SDHB','Brain'] ]

rule tissue_samples:
  input:
    expand('{gene}_counts.txt', gene=GENES),
    expand("{tissue}_samples.txt", tissue=TISSUES)
  output:
    '_'.join( ['-'.join(p) for p in TISSUE_GENE_PAIRS]) + '.png'
  shell:
    'python hist.py ' \
     + ' '.join([' '.join(p) for p in TISSUE_GENE_PAIRS]) \
     + ' {output}'
```

The final step is to add rules for getting the data. These rules will not have an `input:` directive because they do not depend on anything.

```
rule gene_data:
  output:
    "GTEx_Analysis_2017-06-05_v8_RNASeQCv1.1.9_gene_reads.acmg_59.gct.gz"
  Shell:
    "wget https://github.com/swe4s/lectures/raw/master/data_integration/gtex/GTEx_Analy
sis_2017-06-05_v8_RNASeQCv1.1.9_gene_reads.acmg_59.gct.gz"

rule sample_tissue_data:
  output:
     "GTEx_Analysis_v8_Annotations_SampleAttributesDS.txt"
  Shell:
    "wget https://storage.googleapis.com/gtex_analysis_v8/annotations/GTEx_Analysis_v8_
_Annotations_SampleAttributesDS.txt"
```