Managing the Execution of Project Workflows

Snakemake

Programming Practices for Research in Economics

Economics@Zurich

Fall 2017

Some Reflection

- What have we done so far:
  1. Use the command line to pass instructions to computer as text
  2. Version control with git
  3. Programming and Data Analysis with Python and R
- That is essentially **all** the ingredients we need to do the coding tasks within a paper or research project
- How do we manage the workflow of the project?
  - What code is excecuted when?
  - What inputs does a given script need to run?

Solutions for Managing Project Workflows

1. Keep it all in your head
   - **a bad idea!**
2. Writing a long `README` detailing your workflow
   - painful! rarely maintained simulataneously with code
3. A "master file"
   - potentially keeps track of the order things should run
4. A bash script
   - order of execution will matter, dependencies must be specified
   - always runs all scripts
     - inefficient!
5. A build tool, like `Snakemake`
   - order of execution matters
   - keeps track of dependencies, runs partial builds

Today's plan

- Turn a set of scripts into a reproducible bash pipeline
- Identify the problems of using a bash pipeline
- Turn to `Snakemake` as our weapon of choice
- Build up our familarity with `Snakemake` *together*
  - mix an interactive tutorial with small exercises
  - start simple
  - end up with a relatively detailed project workflow
- Translate a bash pipeline into a `Snakemake` workflow *independently*
  - Guided, simple instructions
  - help nearby from fellow learners and instructors

Aside - Directory structure

- Keep inputs and outputs separate
- Separate codes and data into logical subdirectories
- Call scripts from working directory

```
|- ROOT, `.`
|    | - src
|          |- data
|          |- analysis
|          |- model_specs
|    | - out
|          |- data
|          |- analysis
|          |- figures
```

Aside - Setting Up Packages

- Want our code to run on anyone's machine
  - Now, and in months, years after it is hopefully published.
- One way to ensure this?
  - Set up an 'environment' that freezes our packages (`Python`)
  - Use `packrat` for `R`
  - `Stata` - *harder to do*, have to install all `ssc` installed packages to a library inside our ROOT and ensure these are used.
- Alternative? check out `Docker`, but I don't know how to use it

Aside - Windows Users

- I've tried to get many different workflow management tools to work inside Cygwin or other Unix emulators
  - It usually breaks something
- We will use `Powershell` which is a native windows command line interface
  - Main differences:
    - Replace / with \\ everywhere
    - Some different command line functions, we will be clear when it matters

Aside - Setting up Packages II

- For this session:
  - Linux/Mac Users, in a terminal:

    `source set-env.sh`
  - Windows Users, in *Powershell:*

    `set-env.bat`

Aside - Setting up Packages III

What just happened?

- We created a separate `environment` that has specified versions of the packages we need to guarantee our code will run
- Everytime you go to work on this project, type the above to re-enter that environment.
- End of Project? upload the requirements to your repo and everyone can be sure your code will run

Example Project: Testing Zipf's Law

Project: Zipf's Law

- **Zipf's Law**: The most frequently-occurring word occurs approximately twice as often as the second most frequent word.
- Our workflow:
  1. Count occurence of words in multiple texts
  2. Plot the most occuring words for each text in a histogram
  3. 'Test' Zipf's Law
  4. Integrate results into a manuscript

A Simple Directory Structure

```
|- src
|    | - data
|         |- abyss.txt
|         |- isles.txt
|         |- last.txt
|         |- LICENSE_TEXTS.md
|         |- sierra.txt
|    | - analysis
|         |- plotcount.py
|         |- wordcount.py
|         |- zipf_test.py
```

- Ignore (4) - integrating into a paper for now. . .

What the Scripts Do

- `wordcount.py` counts the occurence of works in a text

```
$ python src/analysis/wordcount.py src/data/isles.txt \
      out/analysis/isles.dat
```

- `plotcount.py` plots a histogram of the word occurences

```
$ python src/analysis/plotcount.py out/analysis/isles.dat \
      out/figures/isles.png
```

- `zipf_test.py` tests Zipf's Law on for a set of texts

```
$ python src/analysis/zipf_test.py \
      out/analysis/abyss.dat  \
      out/analysis/isles.dat \
      > out/final/results.txt
```

A Bad Workflow Management Technique

It's (probably) all in the mind

Logic:

1. We know what the scripts do
2. We remember what our data looks like and what we have
3. Manually invoke these scripts as we need them

seems inefficient...

Shell Scripting

Managing a Workflow with a Shell Script

It's (almost) all written down

Logic:

1. We know what the scripts do
2. Write down all the commands we need to execute in a single script
3. Call shell script to execute entire workflow each time we change something

we are making progress...

Writing a Shell Script

```
# runPipeline.sh
# Contributor: Lachlan Deer (@ldeer)
# See 'README.md' for explanation of
#   what this file does.
# Expected Usage:
#   $ bash runPipeline.sh

## Count Words of texts
python src/analysis/wordcount.py src/data/isles.txt \
    out/analysis/isles.dat
<...>
## Plot word count distributions
python src/analysis/plotcount.py out/analysis/isles.dat \
    out/figures/isles.png
<...>
```

Writing a Shell Script (cont.)

```
## Test Zip's Law
python src/analysis/zipf_test.py \
        out/analysis/abyss.dat  \
        out/analysis/isles.dat \
        > out/final/results.txt
```

Execute the script with

```
$ bash runPipeline.sh
```

Advantages of a Shell script

This shell script solves several problems in computational reproducibility:

1. It explicitly documents our pipeline, making communication with colleagues (and our future selves) more efficient.
2. It allows us to type a single command, `bash runPipeline.sh`, to reproduce the full analysis.
3. It prevents us from *repeating* typos or mistakes. You might not get it right the first time, but once you fix something it'll stay fixed.

Disadvantages of a Shell script

- We shouldn't need to recreate all the files each time - only the ones that would be updated / created with our changes
- Commenting out lines, and subsequently uncommenting them, can be a hassle and source of errors in complicated pipelines.

Want an executable *description* of our pipeline that allows software to do the tricky part for us: figuring out what steps need to be rerun.

Workflow Management with Snakemake

What we need

We want to be able to

1. Keep track of our analysis pipeline
2. Specify inputs and outputs explicitly
3. Keep track of complicated file dependencies
4. Only execute code that will update/create new outputs when scripts are modified
5. Be concise, without resorting to over abstraction
6. Not have to learn *yet another faddish programming language*
7. Use a tool designed to manage research workflows, not adapted from other purposes
8. Interface with (some types of) computing clusters and cloud computing environments

Today's Solution

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

ABSTRACT

Summary: Snakemake is a workflow engine that provides a readable Python-based workflow definition language and a powerful execution environment that scales from single-core workstations to compute clusters without modifying the workflow. It is the first system to support the use of automatically inferred multiple named wildcards (or variables) in input and output filenames.

Availability: http://snakemake.googlecode.com.

Contact: johannes.koester@uni-due.de

Received on May 14, 2012; revised on June 28, 2012; accepted on July 28, 2012

1 INTRODUCTION

custom server processes running on the cluster nodes. Finally, Snakemake is the first system to support file name inference with multiple named wildcards in rules.

2 SNAKEMAKE LANGUAGE

A workflow is defined in a 'Snakefile' through a domain-specific language that is close to standard Python syntax. It consists of rules that denote how to create output files from input files. The workflow is implied by dependencies between the rules that arise from one rule needing an output file of another as an input file.

A rule definition specifies (i) a name, (ii) any number of input and output files and (iii) either a shell command or Python code that creates the output from the input. Input and output files

Figure 1: Snakemake

Key Ideas of Snakemake

1. Define `rules` that ultimately execute a script from a shell
2. Within a rule specify (multiple) inputs and outputs
3. Only execute a `rule` if an input (i.e a dependency) changes
4. Encapsulates DRY principle - one rule can be used to run a single script on multiple input combinations

Let's see it in action...

Our First SnakeFile

- Want to count words from the text `abyss.txt`, save output in `abyss.dat`
- Create a file `Snakefile` in our working directory and enter the following:

```
rule abyss:
    input: "src/data/abyss.txt"
    output:
        "out/analysis/abyss.dat"
    shell:
        "python src/analysis/wordcount.py \
            src/data/abyss.txt \
            out/analysis/abyss.dat"
```

- Execute by typing `snakemake` into your terminal and pressing `Return`

Dependencies and Updating Output I

- Try and re-run our `Snakefile`, we get the following:

```
Nothing to be done.
```

- Why?

What's Happening?

When asked to build a target:

1. `Snakemake` checks the 'last modification time' of both the target and its dependencies.
2. If any dependency has been updated since the target, then the actions are re-run to update the target.

- Snakemake knows to only rebuild the files that, either directly or indirectly, depend on the file that changed.
- This is called an **incremental build**

What's Going On

- Snakemake has an option `summary` that documents important information for us:

```
$ snakemake --summary

output_file                         date
out/analysis/abyss.dat  Thu May 18 09:36:50 2017

rule    version log-file(s) status  plan
abyss   -                   ok      no update
```

What's Going On II

- Now let's remove `abyss.dat`, and check the summary output

```
$ rm -rf out/analysis/abyss.dat
$ snakemake --summary

output_file                 date
out/analysis/abyss.dat      -

 rule   version   log-file(s) status   plan
abyss    -                   missing update pending
```

Looking at Snakemake's Execution Plan

- Snakemake has a `dry-run` option that allows us to see what operations it will perform without executing them

```
$ snakemake -n

rule abyss:
    input: src/data/abyss.txt
    output: out/analysis/abyss.dat
    jobid: 1

Job counts:
    count   jobs
    1   abyss
```

- Re-run `snakemake`

What's Going On III

- Now let's update an input `abyss.txt`, and check the summary output

```
$ touch src/data/abyss.txt
$ snakemake --summary

output_file date
out/analysis/abyss.dat  Thu May 18 09:52:20 2017

rule    version log-file(s) status
abyss   -                   updated input files

plan
update pending
```

Your Turn

Challenge: Create a New Rule

- Construct a rule, called 'isles' that counts the words from `isles.txt` and saves the output in `isles.dat`
- Place the rule above the one we made for `abyss`
- Time: 5 mins

Creating a 'Clean' Rule

- Sometimes we want to clean out files that were generated by scripts.

```
rule clean:
    shell:
        "rm -f out/analysis/*.dat"
```

- Execute the rule with `snakemake clean`
- Try and re-run our analysis with the command `snakemake`

Creating 'Target Rules'

- At the top of our `Snakefile` we can write a rule that will ensure all our outputs are created
- How?
  - Rule with no outputs or shell commands
  - Inputs are the files we want created.

```
rule processedData:
    input:
        isles = "out/analysis/isles.dat"
```

Your Turn

Challenge: Update our Target Rule

- Update the target rule so that all our outputs can be constructed with one call to `snakemake`
- Time: 5 mins

Automatic Variables

Where we are now

- `Snakefile` has a lot of duplication.
- For example, the names of text files and data files are repeated in the input and output as well as the call to the shell.
- Snakefiles are a form of code and, in any code, repeated content can lead to problems
- e.g. we rename a data file in one part of the `Snakefile` but forget to rename it elsewhere.
- Violates one of our early stated principles: DRY!
- Let us set about removing some of the repetition from our Snakefile.

Automatic Variables

- Snakemake has 'macros' that allow us to directly use the inputs and outputs we specify in the shell command
- Unsuprisingly, they are called {input} and {output}

```
rule abyss:
    input: "src/data/abyss.txt"
    output:
        "out/analysis/abyss.dat"
    shell:
        "python src/analysis/wordcount.py \
            {input} {output}"
```

Your Turn

Challenge: Update our Target Rule

- Update the rule `isles` using the input and output macros
- Create a new rule `last` that counts the words of the text `last.txt` using input and output macros
- Time: 5 mins

The expand Function

- The processedData rule also has some duplication, all the inputs follow a similar pattern
- We can remove duplication by using snakemake's expand function to reduce this

```
TEXTS = ["abyss", "isles", "last"]

rule processedData:
    input: expand("out/analysis/{iText}.dat",
                  iText = TEXTS)
```

Your Turn

Challenge: Create a results rule

- Create a rule `results` at the top of the `Snakefile`, that takes all data we have produced as inputs and tests Zipf's Law, producing the output `out/final/results.txt`
- Your rule should use everything we have learned so far:
  - `{input}`, `{output}`, `expand(<...>)`
- Time: 10 mins

Dependencies

Getting Dependencies Right

- Our data files are a product not only of our text files but the script, `wordcount.py`, that processes the text files and creates the data files.
- A change to `wordcount.py` (e.g. to add a new column of summary data or remove an existing one) results in changes to the `.dat` files it outputs.
- So, let's pretend to edit `wordcount.py`, and re-run `Snakemake`:

```
$ touch ./src/analysis/wordcount.py
$ snakemake
```

Getting Dependencies Right

```
Nothing to be done.
```

- We need to add `wordcount.py` as a dependency of each of our data files

Specifying Multiple Inputs or Outputs:

```
rule abyss:
    input:
        data = "src/data/abyss.txt",
        script = "src/analysis/wordcount.py"
    output:
        "out/analysis/abyss.dat"
    shell:
        "python {input.script} {input.data} {output}"
```

Getting Dependencies Right!

- Intuitively, we want to add `wordcount.py` as dependency for `results`,
- as the final table should be rebuilt as we remake the `.dat` files.
- However, it turns out we don't have to!
- Let's see what happens to `results` when we update `wordcount.py`...

Getting Dependencies Right!

- The whole pipeline is triggered, even the creation of the `results` file!
- Why?
- `results` depends on the `.dat` files.
- The update of `wordcount.py` triggers an update of the `*.dat` files.
- Thus, `snakemake` sees that the dependencies (the `.dat` files) are newer than the target file (`results.txt`) and thus it recreates `results.txt`.
- This is an example of the power of `snakemake`: updating a subset of the files in the pipeline triggers re-running the appropriate downstream steps.

Your Turn

Challenge: Specifying Correct Dependencies

- Go through all of our rules and correctly identify all inputs
- Time: 10 mins

Pattern Rules

Motivation: DRY Principle Redux

- Our `Snakefile` still has repeated content.
- The rules for each `.dat` file are identical apart from the text and data file names.
- We can replace these rules with a single **pattern rule**
  - can be used to build any `.dat` file from a `.txt` file
- We might also want to replace our `TEXTS` dictionary with a short piece of python code to automatically detect all texts in our data directory
  - Then, when more data are added, analysis workflows immediately extend.

Pattern Rules

- Let's create a rule `countWords` that will take any dataset with a given file extension, count the words inside it, and return the result:

```
rule countWords:
    input:
        data = "src/data/{dataset}.txt",
        script = "src/analysis/wordcount.py"
    output:
        "out/analysis/{dataset}.dat"
    shell:
        "python {input.script} {input.data} {output}"
```

Pattern Rules

- The important point here is that `input` and `output` are both using the same macro, `{dataset}`
- Intuitively, this says
  - find any dataset in the `data` directory ...
  - I want to create an output, using it ...
  - according to the specified `shell` command

Executing the Pattern Rule

- Intuitively, one would think this is the right call to execute our pattern rule:

```
$ snakemake countWords

WorkflowError:
Target rules may not contain wildcards.
Please specify concrete files or a rule
without wildcards.
```

What's going wrong?

Executing the Pattern Rule

- `Snakemake` doesn't know how to fill the `{dataset}` wildcard
  - What datasets do you want to run?
- The Target Rule, `processedData` specifies these data sets:

```
$ snakemake processedData

Provided cores: 1
Rules claiming more threads will be scaled down.
Job counts:
    count   jobs
    4   countWords
    1   processedData
    5
```

Executing the Pattern Rule

- Alternatively, calling another rule, which uses the outputs of your pattern rule as inputs will do the job

```
$ snakemake

Provided cores: 1
Rules claiming more threads will be scaled down.
Job counts:
    count   jobs
    4   countWords
    1   results
    5
```

Your turn

Challenge: Creating a Pattern Rule

- Create a Pattern Rule, and a corresponding Target Rule to generate the `.png` plots we had in our bash pipeline.
- Time: 15 mins

Using Patterns in File Names

- In our `snakemake` file we are explicitly writing down which files we want the programs to use as input data

```
TEXTS = ["abyss", "isles", "last"]
```

- But our texts all come from the same directory, and have the same line ending:

```
$ ls src/data/*.txt

src/data/abyss.txt  src/data/isles.txt
src/data/last.txt  src/data/sierra.txt
```

Using Patterns in File Names

- The python package `glob` can match patterns for us
- The python package `os` knows how to operate on our file system
- `Snakemake` is written in python
- A simple solution is apparent . . .

Using Patterns in File Names

- Replace our manually entered text dictionary with

```
import glob, os

# Get a list os all texts in the data folder
TEXTS = [os.path.splitext(
            os.path.basename(textNames)
            )[0]
                for textNames
                    in glob.glob('src/data/*.txt')]
```

- Try running `snakemake` again

Building the Entire Workflow

Where we are:

We have done quite a bit:

1. How to create rules
2. Using macros / wildcards
3. Correct specification of dependencies
4. Writing 'generalizable' rules

How do we ensure our entire workflow runs through from beginning to end?

- Some (all?) of us weren't getting the `.pngs` *and* `results.txt` built with one call to `snakemake`

Adding an `all` Target

- Recall how we are using Pattern Rules?
  - build all inputs with a particular line ending
  - by specifying them as `inputs`
- Generalize this... specify all targets we want built as inputs to a rule,
  - call the rule `all` ...
  - could be anything, we are just following in the footsteps of others
  - place it at the beginning of your workflow

Adding an `all` Target

```
rule all:
    input:
        table = "out/final/results.txt",
        figures = expand("out/figures/{iText}.png",
                         iText = TEXTS)
```

Adding an `all` Target

- Examine the proposed workflow with
  - `snakemake --summary`
  - `snakemake -np`
- Build the workflow if we are happy with what we see

Subworkflows

Keeping Things Tidy

- Throughout the course we have emphasized that short scripts are the preferred way to work
- Our `snakemake` file is starting to get long
  - and in a full-blown research pipeline would be **much** longer
- **Solution:** divide our one `Snakefile` into multiple, separate ones that can talk to each other to construct the entire workflow
  - also means we can build a each subworkflow separately if we like

Setting up a Subworkflow

- Let's take all the rules that do some kind of analysis, and separate them into their own `Snakefile` located in the `src/analysis` subdirectory

```
## Subworkflow: Analysis
<...>
# --- Build Rules --- ##

rule allAnalysis:
    <...>
rule results:
    <...>
<...>
rule countWords:
    <...>
```

Setting up a Subworkflow

- In our `Snakefile` in the working directory we can now refer to the subworkflow:

```
<...>
# --- Sub Workflows --- #
subworkflow analysis:
    workdir: "."
    snakefile: "src/analysis/Snakefile"
# --- Build Rules --- #
rule all:
    input:
        table = analysis("out/final/results.txt"),
        figures = analysis(
                    expand("out/figures/{iText}.png",
                        iText = TEXTS)),
<...>
```

Your Turn

Challenge: Creating a Subworkflow

- Create a new subworkflow called `manuscript` that compiles a short paper, `shortpaper.md`, to a pdf file `shortpaper.pdf`.
- The short paper takes as an input, the figure `abyss.png`
- The correct path to the figure is in the manuscript already, assuming your working directory is the project's root.
- The shell command to build the paper is `pandoc {input.manuscript} --latex-engine=pdflatex -o {output}`
- Add the output of the manuscript workflow
- Time: 15-20 mins

Adding Useful Rules to `Snakefile`

- Now our main `Snakefile` looks quite short (tidy!)
- There are some other useful rules we can build to make our life a little easier
  - Some additional cleaning rules
  - A rule that prints out some help for the user when they forget which rule does what

A Useful `help` rule

```
## help:              provide simple info about each rule
rule help:
    input:
        mainWorkflow = "Snakefile",
        analysisWorkflow = "src/analysis/Snakefile",
        manuscriptWorkflow = "src/paper/Snakefile"
    shell:
     "sed -n 's/^##//p' {input.mainWorkflow} \
                        {input.analysisWorkflow} \
                        {input.manuscriptWorkflow}"
```

Configuration Files

A Final Step

- Sometimes in our workflows there are some references that we are making all the time
  - Always true of calls to subdirectories
  - And these are easy to screw up and cause headaches
- We can set these references as a dictionary in a separate file `config.yaml`, and call them as we need them
  - can also be a `config.json` if you prefer
- Let's set one up!

A Simple `config.yaml`

```yaml
ROOT: "."
src: "src/"
src_analysis: "src/analysis/"
src_paper: "src/paper/"
src_data: "src/data/"

out_analysis: "out/analysis/"
out_figures: "out/figures/"
out_final: "out/final/"
out_paper: "out/paper/"
```

Importing the `config`

- We need to import the config file into each `Snakefile` that we wish to use it.

```
<...>
# --- Importing Configuration Files --- #
configfile: "config.yaml"
<...>
```

- Reference objects in the dictionary using `config["someName"]`
  - Let's do this together in our main `Snakefile` in the working directory

Referencing config

```
# --- Sub Workflows --- #
subworkflow analysis:
    workdir: config["ROOT"]
    snakefile: config["src_analysis"] + "Snakefile"
<...>
## All:    generate all outputs,
##             copy paper to root directory
rule all:
    input:
        table = analysis( config["out_final"]
                          + "results.txt"),
<...>
```

Your Turn

Challenge: Using config files

- Import our `config.yaml` file into each of our subworkflows
- Replace directory references in each workflow with the corresponding reference to the config dictionary
- Time: 10-15 mins

Summary

What we argued

- Correctly specifying a correct workflow that manages an entire build of a research project is non-trivial
- The worst option (and most commonly used) is fatally flawed
- Build Tools can manage the entire workflow of large projects that resemble academic papers and research reports
  - particularly when there are computational steps involved
- `Snakemake` is a relatively easy toolbox to manage our workflow and track file dependencies

What we did

Used `Snakemake` to:

- Divide a workflow into a set of individual rules that specify outputs and inputs
- Correctly specify file dependencies to generate partial builds
- Generalized rules to reduce repitition and minimize error
- Split a research project into manageable subworkflows that are easily combined
- Decorate our example to reflect a `simple` workflow that resemebles how a full-blown project should be managed

What we didn't do

- Examine how to use inputs from the `Snakemake` workflow inside our scripts
  - Specific to Python and R
- How to use `Snakemake` to manage jobs sent to computational clusters
- Examine and use more advanced features of Snakemake
  - That said, we have already built a pretty powerful set of tools in just a few hours
- Compare alternative build tools

Complete Documentation available:
`Snakemake Documentation`
but generally uses an unfamiliar to us example from Bioinformatics

Acknowledgements

- This module is designed after and borrows a lot from:
  - `Effective Programming Practices for Economists`, a course by Hans-Martin von Gaudecker
  - Software Carpentry's `Automation and Make` lesson
- `Snakemake` is software made available under the `MIT` license and was created by Johannes Köster
- The example project from these slides utilizes python scripts and data originally designed for Software Carpentry's `Automation and Make` lesson
- The follow up exercise adapts the `Schelling Model` example from Hans-Martin's module on `Waf`, from which the python scripts were originally borrowed from `QuantEcon.`
- The material from above sources is made available under Attribution based Licenses, as is this course's material.

License

- Material is licensed under a CC-BY-NC-SA license. Further information is available at our `course homepage`
- Suggested Citation:
  Deer, Lachlan; Adrian Etter, Julian Langer, Max Winkler (2017), `Managing the Execution of Project Workflows: Snakemake`, Programming Practices for Research in Economics, University of Zurich.

Programming Practices Team

Programming Practices for Research in Economics was created by

* Lachlan Deer
* Adrian Etter
* Julian Langer
* Max Winkler

at the Department of Economics, University of Zurich in 2016. These slides are from the 2017 edition, conducted by the original course developers.