6: GNU Make

`bit.ly/2018rr`

Here, we'll talk about automating the entire workflow for a project using the tool GNU Make.

GNU Make

- Automation the full project
- Document dependencies
- Only re-run things that need to be re-run

GNU Make is an old tool that was originally for automating the compilation of large, complex programs. But it's useful much more generally, though it does have some quirks.

In addition to automating, you'll be documenting the dependencies among the steps, and since the dependencies are defined, only stuff that needs to be re-run will be.

Automate the process (GNU Make)

```
R/analysis.html: R/analysis.Rmd Data/cleandata.csv
    cd R;R -e "rmarkdown::render('analysis.Rmd')"

Data/cleandata.csv: R/prepData.R RawData/rawdata.csv
    cd R;R CMD BATCH prepData.R

RawData/rawdata.csv: Python/xls2csv.py RawData/rawdata.xls
    Python/xls2csv.py RawData/rawdata.xls > RawData/rawdata.csv
```

GNU Make is an old (and rather quirky) tool for automating the process of building computer programs. But it's useful much more broadly, and I find it valuable for automating the full process of data file manipulation, data cleaning, and analysis.

In addition to automating a complex process, it also documents the process, including the dependencies among data files and scripts.

Automation with GNU Make

- `Make` is for more than just compiling software
- The essence of what we're trying to do
- Automates a workflow
- Documents the workflow
- Documents the dependencies among data files, code
- Re-runs only the necessary code, based on what has changed

People usually think of Make as a tool for automating the compilation of software, but it can be used much more generally.

To me, Make is the essential tool for reproducible research: automation plus the documentation of dependencies and workflows.

Fancier example

```
FIG_DIR = Figs

mypaper.pdf: mypaper.tex $(FIG_DIR)/fig1.pdf $(FIG_DIR)/fig2.pdf
    pdflatex mypaper

# One line for both figures
$(FIG_DIR)/%.pdf: R/%.R
    cd R;R CMD BATCH $(<F)

# Use "make clean" to remove the PDFs
clean:
    rm *.pdf Figs/*.pdf
```

As I said, you can get really fancy with GNU Make.

Use variables for directory names or compiler flags. (This example is not a good one.)

Use pattern rules and automatic variables to avoid repeating yourself. With `%`, we have one line covering both `fig1.pdf` and `fig2.pdf`. The `$(<F)` is the file part of the first dependency. More on this later.

Look at the manual for Make and the many online tutorials, such as the one from Software Carpentry, or `http://kbroman.org/minimal_make`.

Installing Make

- On Macs, Make should be installed. Type "`make --version`" to check.
- On Windows, probably the easiest is to install Rtools, which includes Make.

  `cran.r-project.org/bin/windows/Rtools`

Installation of these sorts of command-line tools on Windows can be a bit difficult.

How do you use Make?

- If you name your make file `Makefile`, then just go into the directory containing that file and type `make`
- If you name your make file `something.else`, then type `make -f something.else`
- Actually, the commands above will build the first target listed in the make file. So I'll often include something like the following.

  ```
  all:  target1 target2 target3
  ```

  Then typing `make all` (or just `make`, if `all` is listed first in the file) will build all of those things.
- To be build a specific target, type `make target`. For example, `make Figs/fig1.pdf`

Details on the use.

Variables

- Define a variable like
  `R_OPTS=--vanilla`
- Use it with a `$` and `()` or `{}`, for example:
  `R CMD BATCH $(R_OPTS) fig1.R`

Variables are useful shorthand, for bits of code that you want to use repeatedly.

Automatic variables

There are a bunch of automatic variables that you can use to save yourself a lot of typing.

Here are the ones I use most:

`$@`	the file name of the target
`$<`	the name of the first dependency
`$^`	the names of all dependencys
`$(@D)`	the directory part of the target
`$(@F)`	the file part of the target
`$(<D)`	the directory part of the first dependency
`$(<F)`	the file part of the first dependency

You'll see on the next slide how automatic variables get used.

Pattern rules

Pattern rules are like wildcards for file names: if a bunch of files are to be built the same way, you can use the symbol % as a wildcard.

For example, if you have two figures `fig1.pdf` and `fig2.pdf` that are to be built by `fig1.R` and `fig2.R`, respectively, you might do:

```
Figs/%.pdf: R/%.R
    cd $(<D);R CMD BATCH $(<F)
```

The two figures’ file names will need to be be spelled out somewhere, for example as dependencies.

Pattern rules can greatly reduce the length of your `Makefile`, but they can also be rather frustrating. I recommend keeping things really simple initially and then move to pattern rules later, after you’ve been working with Make for a while.

Resources

- `kbroman.org/minimal_make`
- `bost.ocks.org/mike/make`
- `robjhyndman.com/hyndsight/makefiles`
- Search github with `filename:Makefile`
  - `R CMD BATCH filename:Makefile`
  - `filename:Makefile user:yihui`

A lot of people are using Make; you can search for their files on GitHub. And there are some good introductory tutorials out there.

Activity

Go back to your R Markdown documents from this morning.

- Write a `Makefile` to produce different types of outputs from your various Rmd files.
- Add `make all` and `make clean` as targets
- What happens if you run `make all` twice?

An activity on Make.