Command line automation: Makefiles

Mikhail Dozmorov

Summer 2018

GNU make

- You will almost certainly have to re-run an analysis more than once, possibly with new or changed data
- **GNU Make** is a tool which controls the workflow of generating target/result files from the dependencies (source files)
  - Target/result files may be text files, standalone programs, packages

Capabilities of Make

- Make is for more than just a tool for compiling software
- The path from raw data to final results
- Automates/documents a workflow
- Intelligently handles the dependencies among data files, code
- Accounts for the updates in data, code
- Re-runs only the necessary code, based on what has changed

Makefile structure

Makefile contains recipes in the form of:

```
target: dependencies
    <code>
```

- `target` - the outcome
- `dependencies` - the necessary parts to build the outcome
- code - outlines the rules to build target using dependencies
- All code/commands must be **tab-indented**
- Dependencies, if more than one, must be **space-separated**

Makefile example

Save in Makefile text file, no extension

```
# An example of obtaining counts and types of the cytobands
all:    cytoband_counts.txt cytoband_types.txt

# Download the raw data
cytoBand.txt.gz:
    wget http://hgdownload.cse.ucsc.edu/goldenPath/hg19/database/$@

# Obtain counts of the cytobands
cytoband_counts.txt:    cytoBand.txt.gz
    zcat < $< | cut -f1 | sort | uniq -c | awk '{OFS="\t"} {print $$2,$$1}'

# Obtain types of the cytobands
cytoband_types.txt: cytoBand.txt.gz
    zcat < $< | cut -f5 | sort | uniq -c | awk '{OFS="\t"} {print $$2,$$1}'

clean:
    rm *.gz
```

How to use make

- If you name your make file `Makefile`, then just go into the directory containing that file and type `make` - it'll run the first recipe
- If you name your make file `something.else`, then type `make -f something.else`
- By default, `make` builds the first target listed in the `Makefile`. Generally, the first target generates all other targets

```
all: target1 target2 target3
```

- To build a specific target, type `make target`. For example, `make cytoband_counts.txt`

Typical Makefile recipes

- `clean` – commands to clean up the working directory from temporary files
- `test` – runs a series of tests
- `install` – installs a software

Typical software installation steps using Makefiles

```
./configure
make
make install
```

Makefile variables

- A variable is a name defined in a makefile to represent a string of text, called the variable's value. Variables are used to simplify recipes
- Defining internal Makefile variable

```
DB = "/home/genomerunner/db_2.00_06.10.2014""
```

- Using a variable

```
${DB} or $(DB)
```

https://www.gnu.org/software/make/manual/make.html#Using-Variables

Using shell variables in Make

- Shell variables, e.g. `$HOME`, need to be prefixed by `$`
- Within shell variable use: `awk '{print $0}'`
- Within Makefile variable use: `awk '{print $$0}'`
- Capturing output of shell commands into a variable: `TXT_FILES = $$(shell find . -type f -name "*.txt")`

TIP!

- The content of a Makefile runs in its own shell environment. The default shell environment is `/bin/sh`. To set shell environment to bash, use `SHELL=/bin/bash`
- Why bother? Variable `$SECONDS` exists in `bash`, but not in `sh`. Other syntax incompatibilities, e.g., `if-else-fi` syntax

Automatic variables

Makefile contains recipes in the form of:

```
target: dependencies
    <code>
```

- `$<` - the name of the first dependency
- `$@` - the name of the target of the rule
- `$?` - the names of all the dependencies
- `$(<F)` - the file part of the first dependency

Example:

```
COMPILER = g++                  # Define compiler
COMPILER_FLAGS=-c -Wall # Define flags

hello.o: hello.c hello.h    # Recipe
         $(COMPILER) $(COMPILER_FLAGS) $< -o $@
```

https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

Patterns

- A pattern rule allows "wildcard" matching between the target and the dependencies. The % wildcard is similar to the * wildcard in bash

Existing files: `module0_induction.Rmd`, `module1_basics.Rmd`, `module2_managingR.Rmd`

Makefile recipe:

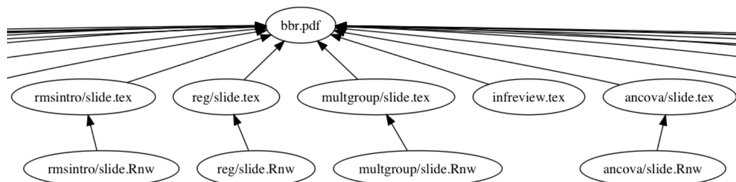
```
%.html:     %.Rmd
        echo $(@)
        ./compile_slides $(basename $(@))
```

Results: `module0_induction.html`, `module1_basics.html`, `module2_managingR.html`

Visualize Makefiles

There are tools to look at Makefiles, like `makefile2dot`

```
python makefile2dot.py < Makefile | dot -Tpng > out.png && ope
```



https://github.com/vak/makefile2dot

Debugging Makefiles

- Problem: Make is doing something strange
- Solution: Keep is simple. Use `make -n -d` (`-n`, or `--dry-run` doesn't run anything and `-d` turns on debugging information)

More automation

- `snakemake` - workflow management system similar to make but uses Python syntax
- `Sequana` - a set of Snakemake NGS pipelines
- `Drake` - Data workflow tool, like a "Make for data"
- `CWL` (Common Workflow Language) - a specification for describing analysis workflows and tools
- `Nextflow` - Data-driven computational pipelines

https://snakemake.readthedocs.io/en/stable/

http://sequana.readthedocs.io, https://github.com/sequana/sequana

https://github.com/Factual/drake, short tutorial
https://www.datascienceatthecommandline.com/chapter-6-managing-your-data-workflow.html

http://www.commonwl.org/, https://github.com/common-workflow-language/common-workflow-language,
https://figshare.com/articles/Common_Workflow_Language_draft_3/3115156/2

https://www.nextflow.io/