Make and Makefiles for workflows

Hannah Holland-Moritz

April 20, 2020

Getting Started

Materials

You don't have to participate at home, but if you want to participate at home, you will need:

1. To download the make_example file from github (link in Zoom chat)
2. Move the file to a place that you will be able to easily navigate to with the terminal, for example your Desktop
3. Open your terminal and navigate to that location.

For example `cd ~/Desktop`

4. Use the terminal to unzip the example file, and then move inside of it.

```
unzip make_example.zip
cd ~/make_example
```

Resources/Links/Inspiration:

Today's presentation is heavily inspired by:

1. Software Carpentry's lesson in makefile
   - https://swcarpentry.github.io/make-novice/
2. Karl Broman's "minimal make" tutorial
   - https://kbroman.org/minimal_make

Why you should care

Why should we be using makefiles?

1. More efficient use of your time and energy. You can change your analysis pipeline and rerun only the parts of the pipeline that are affected by the change.

Why should we be using makefiles?

1. More efficient use of your time and energy. You can change your analysis pipeline and rerun only the parts of the pipeline that are affected by the change.
2. Reproducible, shareable analysis with minimal brain input

Why should we be using makefiles?

1. More efficient use of your time and energy. You can change your analysis pipeline and rerun only the parts of the pipeline that are affected by the change.
2. Reproducible, shareable analysis with minimal brain input
3. Self-documenting pipeline

Challenges of `make`

1. Weird rules and idiosyncrasies

Challenges of `make`

1. Weird rules and idiosyncrasies
2. Can have cryptic error messages

Challenges of `make`

1. Weird rules and idiosyncrasies
2. Can have cryptic error messages
3. Requires a bit of knowledge about the unix shell

Today's goals:

1. Learn how to run `make`.

Today's goals:

1. Learn how to run `make`.
2. Learn how to write Makefiles

Today's goals:

1. Learn how to run `make`.
2. Learn how to write Makefiles
3. Apply that knowledge in a simple R pipeline.

Introduction to `make`

make is a program that executes instructions in a makefile

- make looks for a file called "Makefile" and runs all of the instructions in the makefile
- By default it assumes your makefile is called "Makefile", but you can specify different file name if you want using the `-f` flag

How to run make:
```
[you@localhost make_example]$ make
[you@localhost make_example]$ make -f mymake.txt
```
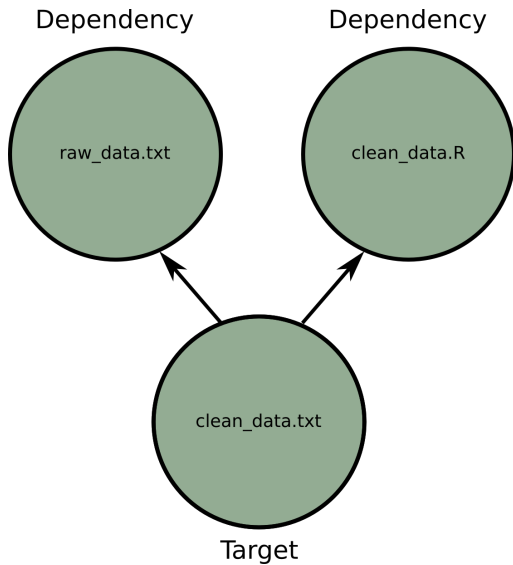
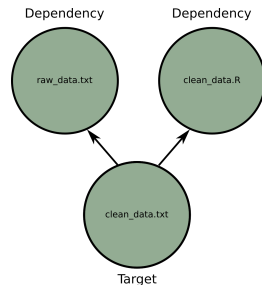How to write makefiles

Anatomy of a makefile

Anatomy

```
# Makefile structure
target: dependency
    action
```

- *target*: output files that you are trying to make
- *dependency*: the files (including scripts and data), that your outputs depend upon
- *action*: a command that can be run in a terminal that describes what to do with the dependencies.

Anatomy of a makefile

Makefile

```
# Makefile structure
clean_data.txt: clean_data.R raw_data.txt
    R CMD BATCH clean_data.R
```

Important Syntax:

The second line **must** begin with a TAB, not spaces.

Action lines must begin with tab

```
# Makefile structure
clean_data.txt: clean_data.R raw_data.txt
    R CMD BATCH clean_data.R # This line starts with TAB
```

Important Syntax:

The first line must be on one line from the computer's perspective. But you can break it up into multiple human-readable lines with line breaks (backslashes) for easy reading.

Without a line break

```
clean_data.txt: clean_data.R raw_data.txt
    R CMD BATCH clean_data.R
```

With a line break

```
clean_data.txt: clean_data.R \
raw_data.txt
    R CMD BATCH clean_data.R
```

Important Syntax:

The action line must be a command that can be run in the terminal. You can use as many actions as necessary to complete your task.

- To run an R script in the terminal use either `R CMD BATCH myscript.R` or `Rscript myscript.R`.

Action lines must be executable in the terminal

```
# Makefile structure
clean_data.txt : clean_data.R raw_data.txt
    R CMD BATCH clean_data.R # executable in the terminal
```

In summary:

1. Begin each action with a TAB

In summary:

1. Begin each action with a TAB
2. Use line breaks if you have to break up the first line.

In summary:

1. Begin each action with a TAB
2. Use line breaks if you have to break up the first line.
3. Construct actions so that they can be run in a terminal.

Your turn: Write a makefile called "Makefile" for the following workflow

- Save your makefile in the folder called "make_example"
- R scripts can be run with `R CMD BATCH myscript.R`
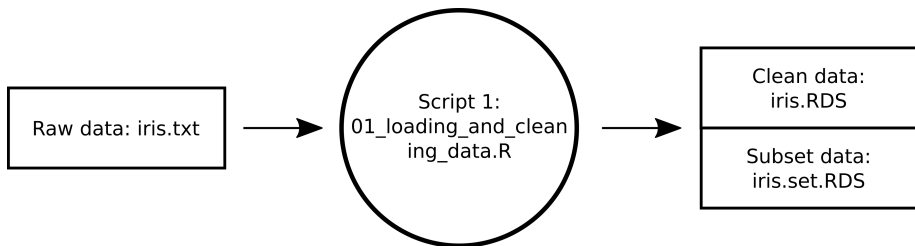

```
# Makefile structure
target: dependency
    action
```

Your turn: Writing a makefile

Makefile_00

```
# Cleaning Data =========================================

# Files from 01_loading_data_and_cleaning_data.R
iris.RDS \
iris.set.RDS: 01_loading_and_cleaning_data.R iris.txt
    R CMD BATCH 01_loading_and_cleaning_data.R
```

Your turn: Running your makefile

Makefiles only run a step if the dependencies have been updated (i.e. if their time stamps have changed).

The first time you run

```
[you@localhost make_example]$ make
R CMD BATCH 01_loading_and_cleaning_data.R
```

The first time you run

```
[you@localhost make_example]$ make
R CMD BATCH 01_loading_and_cleaning_data.R
```

The second time you run

```
[you@localhost make_example]$ make
make: 'iris.RDS' is up to date.
```

After a timestamp has been updated

```
[you@localhost make_example]$ touch iris.txt
[you@localhost make_example]$ make
R CMD BATCH 01_loading_and_cleaning_data.R
```

Errors

Make has cryptic errors, so let's go through some of them now.

If you accidentally forget a line break or use spaces instead of a tab for the second line you'll see the following error:

Missing seperator error

```
[you@localhost make_example]$ make
Makefile:5: *** missing separator.  Stop.
```

If your action line is not able to run, you'll get a much more cryptic error usually with "Command not found".

Execution error

```
[you@localhost make_example]$ make
make: 01_loading_and_cleaning_data.R: Command not found
make: *** [Makefile:9: iris.RDS] Error 127
```

Other errors

Any other cryptic errors, may be the result of a bug in your script. You can check the output file (for R, it's .Rout) to search for clues.

Errors Summary:

1. "missing seperator" $->$ check your tabs, linebreaks, and spaces

Errors Summary:

1. "missing seperator" –> check your tabs, linebreaks, and spaces
2. "Command not found" –> Is your action line something that can be run in the terminal?

Errors Summary:

1. "missing seperator" $->$ check your tabs, linebreaks, and spaces
2. "Command not found" $->$ Is your action line something that can be run in the terminal?
3. Other error $->$ Check your script for bugs

Dealing with Multiple targets

Dealing with Multiple targets
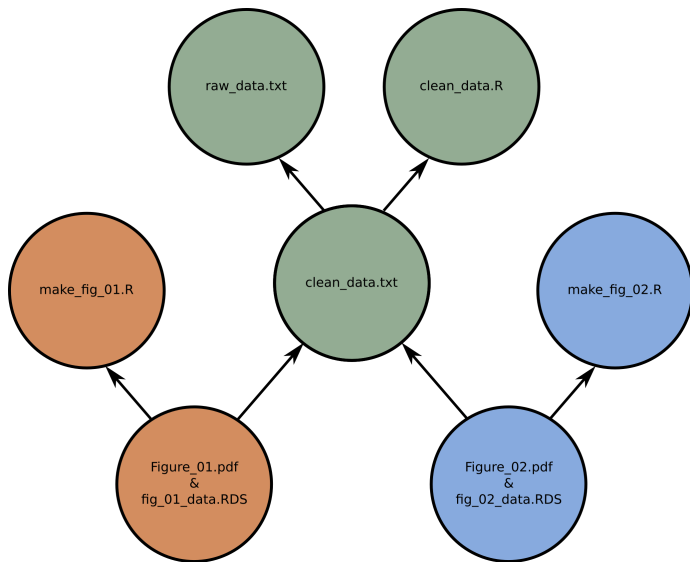
Makefile

```
# Makefile structure
clean_data.txt: clean_data.R raw_data.txt
    R CMD BATCH clean_data.R

Figure_01.pdf fig_01_data.RDS: make_fig_01.R clean_data.txt
    R CMD BATCH make_fig_01.R

Figure_02.pdf fig_02_data.RDS: make_fig_02.R clean_data.txt
    R CMD BATCH make_fig_02.R
```

- By default, make will only run the first target and any of its dependencies so you have to specify what you want to create.

Makefile

```
# Makefile structure
clean_data.txt: clean_data.R raw_data.txt
    R CMD BATCH clean_data.R

Figure_01.pdf fig_01_data.RDS: make_fig_01.R clean_data.txt
    R CMD BATCH make_fig_01.R

Figure_02.pdf fig_02_data.RDS: make_fig_02.R clean_data.txt
    R CMD BATCH make_fig_02.R
```

To create Figure_01.pdf:

```
[you@localhost make_example]$ make Figure_01.pdf
```

"Phony" targets are a convenient way to group targets together

Since we don't want to have to type a new command for every figure in our workflow, we can create a target of targets (also called a "Phony" target) to create all of our figures at one time.

Makefile
```
.PHONY: figures

figures: Figure_01.pdf Figure_02.pdf

# Makefile structure
clean_data.txt: clean_data.R raw_data.txt
    R CMD BATCH clean_data.R
...
```
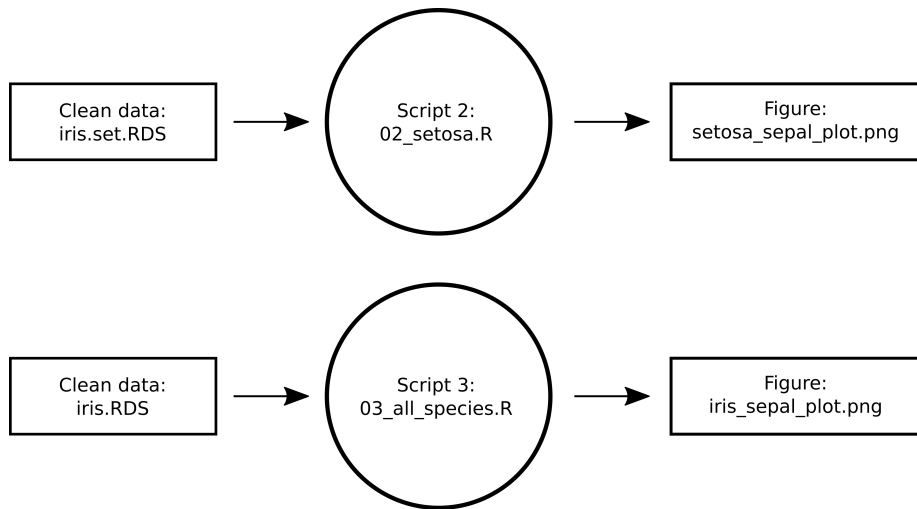
To create our figures:

```
[you@localhost make_example]$ make figures
OR
[you@localhost make_example]$ make
```

Your turn: Phony targets

Add the following steps to your workflow and then create a Phony target for the figures.

Makefile_01

```
.PHONY : figures

# Figures ==========================================
figures: iris_sepal_plot.png \
setosa_sepal_plot.png

# Cleaning Data ====================================
...
# Analyses =========================================

# All species together
iris_sepal_plot.png: 03_all_species.R iris.RDS
    R CMD BATCH 03_all_species.R

# Setosa
setosa_sepal_plot.png: 02_setosa.R iris.set.RDS
    R CMD BATCH 02_setosa.R
```

Your turn: Run your makefile

> ## To create our figures:
>
> ```
> [you@localhost make_example]$ make figures
> OR
> [you@localhost make_example]$ make
> ```

Default targets

Let's say that most often you will be (re)making your figures. You can save yourself some typing by telling make explicitly to make figures the default target. figures will then be the default target regardless of its position in the makefile. For example:
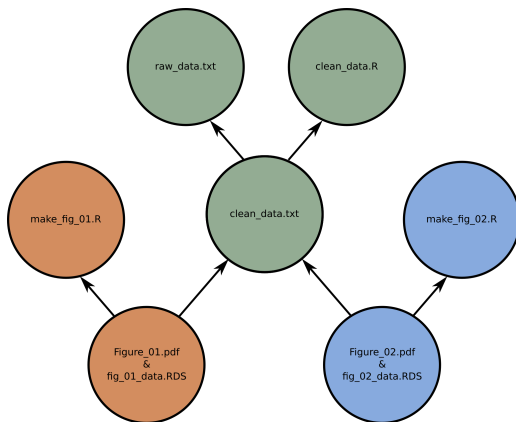
Makefile

```
.PHONY : figures
.DEFAULT_GOAL := figures

# Figures ==========================================
figures: iris_sepal_plot.png \
setosa_sepal_plot.png
...
```

Important!! Your dependencies *must not* be circular

If any of your dependencies are circular (Figure_01 depends on on Figure_02 and vice versa) then your makefile will always run both, regardless of the timestamps and the order it will run them in will be arbitrary.

Other oddities of make

Other oddities of `make`: Only one-line actions are allowed

If you need to change directories to run a command or do any other action that normally uses multiple steps, you must do it on one line with line-breaks. For example:

Changing directories

```
# Makefile structure
clean_data.txt : R/clean_data.R raw_data.txt
  cd R;\
    R CMD BATCH clean_data.R

Figures/Figure_01.pdf fig_01_data.RDS: R/make_fig_01.R clean_d
  cd R;\
    R CMD BATCH make_fig_01.R
```

Other oddities of make: The "@" prefix

By default all action lines are printed to the terminal as they are run.
Preface an action line with "@" to print the results of the command rather
than the command itself. For example:

Makefile

```
.PHONY : figures
.DEFAULT_GOAL := figures

figures : Figure_01.pdf Figure_02.pdf
  @echo "Made figures"
...
```

Running makefile

```
[you@localhost make_example]$ make figures
R CMD BATCH clean_data.R
R CMD BATCH make_fig_01.R
R CMD BATCH make_fig_02.R
Made figures
```

Housekeeping and Good Coding Practices

make clean

make clean

Often your workflow produces many files, some of which are temporary and can be removed. It's also fairly common that you might want to force make to rerun all of your analyses from the beginning. This is often taken care of by creating a target called `clean` which removes your unwanted files.

```
make clean
# Makefile Cleaning ====================================
clean: remove-figures remove-R-outputs

remove-figures:
    rm -f *.png
    rm -f *.pdf

remove-R-outputs:
    rm -f *.Rout
    rm -f *.RDS
```

> **To clean our directory:**
>
> ```
> [you@localhost make_example]$ make clean
> rm -f *.png
> rm -f *.pdf
> rm -f *.Rout
> rm -f *.RDS
> Or to only remove R-outputs
> [you@localhost make_example]$ make remove-R-outputs
> rm -f *.Rout
> rm -f *.RDS
> ```

Your turn: Add a clean section to your makefile

Add a section that removes all files ending in ".png", ".pdf", ".Rout", and "*.RDS" to the iris workflow makefile. Don't forget to add the new section to your list of phony targets.

- Don't forget to add clean to your .Phony section after you are finished

Makefile_02

```
.PHONY : clean figures
.DEFAULT_GOAL := figures
```

Your turn: run make clean

Self-documenting makefiles

Self-documenting makefiles: makefiles with a help section

To prevent you from forgetting what each of your phony targets does, you can add a self-generating help section to your makefile.

- Let's say that everytime we add a new section to our makefile, we write a little about it, prefaced by ##.

Documentation

```
.PHONY : clean figures
.DEFAULT_GOAL := figures

# Figures ==============================================
## figures: makes figures for all iris species and for setosa
figures: iris_sepal_plot.png \
setosa_sepal_plot.png

...
# Makefile Cleaning ====================================
## clean: cleans files ending in ".png", ".pdf", ".Rout", and
clean: remove-figures remove-R-outputs
...
```

Self-documenting makefiles

Now we can create a new target called "help", that uses the search-and-replace program `sed` to create documentation. The target `help` will depend on the makefile itself.

Documentation

```
help: Makefile
    @echo "Targets to make:"
    @sed -n 's/^##/    /p' Makefile
    @echo ""
```

Your turn: Add a help section

1. First add the help target to the end of your makefile.
2. Then add lines that begin with ## before each target to create the documentation.

Help target

```
help: Makefile
    @echo "Targets to make:"
    @sed -n 's/^##/    /p' Makefile
    @echo ""
```

Advanced Tips and Tricks

Shortcuts

Makefiles can be made less repetitive and more reliable to changes in file names by using shortcut variables.

Shortucut Variables

- $@ = "the target of the current rule"
- $^ = "all the dependencies of the current rule"
- $< = "the first dependency of the current rule"
- $(@D) = "the directory part of the target"
- $(@F) = "the file part of the target"
- $(<D) = "the directory part of the dependency"
- $(<F) = "the file part of the dependency"

More information

For a more in-depth tutorial on reducing repetition in your makefile see steps 3 - 7 in https://swcarpentry.github.io/make-novice/

Advanced Makefile for an R output

Check out the Makefile for the make_example_advanced folder for an example of a more advanced makefile that deals with directory structures.
Thanks!