make

Lecture 15

Dr. Colin Rundel

make

- build tool for the creation of software / libraries / documents by specifying dependencies
  - Almost any process that has files as input and outputs can be automated via make
- Originally created by Stuart Feldman in 1976 at Bell Labs
- Almost universally available (all flavors of unix / linux / MacOS / Windows via RTools)
- Dependencies are specified using a text-based `Makefile` with a simple syntax

Makefile

A `Makefile` provides a list of target files along, their dependencies, and the steps necessary to generate each of the targets from the dependencies.

```
target1: depend1 depend2 depend3 ...
    step1
    step2
    step3
    ...

depend1: depend4 depend5
    step1
    step2
    ...
```

In the above example `target*` and `depend*` are all just files (given by a relative or absolute path).

Makefile (basic example)

```
paper.html: paper.Rmd fig1/fig.png fig2/fig.png
    Rscript -e "rmarkdown::render('paper.Rmd')"

fig1/fig.png: fig1/fig.R
    Rscript fig1/fig.R

fig2/fig.png: fig2/fig.R
    Rscript fig2/fig.R
```

Smart Building

Because the `Makefile` specifies the dependency structure `make` knows when a file has changed (by examining the file's modification timestamp) and only runs the steps that depend on the file(s) that have changed.

- After running `make` the first time, I edit `paper.Rmd`, what steps run if I run `make` again?
- What about editing `fig1/fig.R`?

Variables

Like R or other language we can define variables

```
1 R_OPTS=--no-save --no-restore --no-site-file --no-init-file -
2
3 fig1/fig.png: fig1/fig.R
4     cd fig1;Rscript $(R_OPTS) fig.R
```

Special Targets

By default if you run `make` without arguments it will attempt to build the *first* target in the `Makefile` (whose name does not start with a `.`). By convention we often include an `all` target which explicitly specifies how to build everything within the project.

`all` is an example of what is called a phony target - because there is no file named `all` in the directory. Other common phony targets:

- `clean` - remove any files created by the Makefile, restores to the original state
- `install` - for software packages, installs the compiled programs / libraries / header files

Optionally, we specify all phony targets by including a line with `.PHONY` as the target and the phony targets as dependencies, i.e.:

```
1 .PHONY: all clean install
```

Builtin / Automatic Variables

- `$@` - the file name of the target
- `$<` - the name of the first dependency
- `$^` - the names of all dependencies
- `$(@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

Pattern Rules

Often we want to build several files in the same way, in these cases we can use `%` as a special wildcard character to match both targets and dependencies.

So we can go from

```
fig1/fig.png: fig1/fig.R
    cd fig1;Rscript fig.R

fig2/fig.png: fig2/fig.R
    cd fig2;Rscript fig.R
```

to

```
fig%/fig.png: fig%/fig.R
    cd $(<D);Rscript $(<F)
```

Makefile (fancier example)

```
all: paper.html

paper.html: paper.Rmd fig1/fig.png fig2/fig.png
	Rscript -e "library(rmarkdown);render('paper.Rmd')"

Fig%/fig.png: Fig%/fig.R
	cd $(<D);Rscript $(<F)

clean:
	rm -f paper.html
	rm -f Fig*/*.png

.PHONY: all clean
```

Live Demo
HW4 Makefile

HW4 Makefile

```
all: hw4.html

hw4.html: hw4.qmd data/lq.rds data/dennys.rds
	quarto render hw4.qmd

data/lq.rds: parse_lq.R data/lq/*.html
	Rscript parse_lq.R

data/lq/*.html: get_lq.R
	Rscript get_lq.R

data/dennys.rds: parse_dennys.R data/dennys/*.html
	Rscript parse_dennys.R

data/dennys/*.html: get_dennys.R
	Rscript get_dennys.R

clean:
	rm -f hw4.html
	rm -rf data/
```


targets
github.com/ropensci/targets

Why targets?

`make` is great, but has some limitations for data analysis pipelines:

- File-based dependencies only (what about R objects in memory?)
- Limited insight into what changed and why
- Limited support for parallel execution
- Syntax is shell based

The `targets` package provides a modern R-native alternative:

- Define pipelines in R code
- Track both files and R objects
- Automatic parallel execution
- Better debugging and visualization
- Integration with R ecosystem (Quarto, R Markdown, etc.)

Getting started with targets

```
# Create a new targets project
use_targets()

# Explore your pipeline
tar_visnetwork()
tar_manifest()

# Run the pipeline
tar_make()

# Check what needs updating
tar_outdated()

# Access a target from storage
tar_read()

# To reset / clean-up
tar_invalidate()
tar_prune()
tar_destroy()
```

Basic targets workflow

A `targets` pipeline is defined in a `_targets.R` file:

```
library(targets)

tar_option_set(packages = c("tibble", "dplyr"))

list(
  tar_target(file, "data.csv", format = "file"),
  tar_target(raw_data, read.csv(file)),
  tar_target(clean_data, raw_data |> filter(!is.na(value))),
  tar_target(summary, summarize(clean_data, mean = mean(value)))
)
```

Run the pipeline:

```
tar_visnetwork()
tar_make()
tar_read(summary)
```

HW4 pipeline with targets

```
# _targets.R
library(targets)

list(
  # Get La Quinta data
  tar_target(lq_html_files, {
    source("get_lq.R")
    list.files("data/lq", pattern = "*.html", full.names = TRUE)
  }),

  tar_target(lq_data, {
    source("parse_lq.R")
    readRDS("data/lq.rds")
  }, depend_on = lq_html_files),

  # Get Denny's data
  tar_target(dennys_html_files, {
    source("get_dennys.R")
    list.files("data/dennys", pattern = "*.html", full.names = TRUE)
  }),
```