MONASH University

MONASH BUSINESS SCHOOL

ETC4500/ETC5450

Advanced R programming

Week 7: Reactive programming with targets and renv

Outline

1. Reactive programming
2. Caching
3. targets
4. Reproducible environments

Outline

1 Reactive programming

2 Caching

3 targets

4 Reproducible environments

Regular (imperative) programming

Consider how code is usually evaluated...

```
a <- 1
b <- 2
x <- a + b
x
```

What is `x`?

```
a <- -1
x
```

What is `x` now?

Regular (imperative) programming

Predictable programming

All programming we’ve seen so far evaluates code in sequential order, line by line.

Since `x` was not re-evaluated, its value stays the same even when its inputs have changed.

Reactive programming

Within a reactive programming paradigm, objects *react* to changes in their inputs and automatically update their value!

Reactive programming

Within a reactive programming paradigm, objects *react* to changes in their inputs and automatically update their value!

⚠ Disclaimer

Reactive programming is a broad and diverse paradigm, we'll focus only on the basic concepts and how they apply in shiny applications.

Reactive programming

We can implement *reactivity* with functions & environments.

```r
library(rlang)
react <- function(e) new_function(alist(), expr(eval(!!enexpr(e))))
```

We’ll learn how this function works later (metaprogramming).

Reactive programming is also smarter about ‘*invalidation*’, results are **cached and reused** if the inputs aren’t changed.

Reactive programming

How does reactive programming differ?

```
a <- 1
b <- 2
y <- react(a + b)
y()
```

What is `y`?

```
a <- -1
y()
```

What is `y` now?

Reactive programming

(Un)predictable programming?

Reactive programming can be disorienting!

Reactive objects *invalidate* whenever their inputs change, and so its value will be recalculated and stay up-to-date.

Reactive programming

Your turn!

```r
a <- 1
b <- 2
y <- react(a + b)
y()
```

When was `a + b` evaluated?

How does this differ from ordinary (imperative) code?

Imperative and declarative programming

Imperative programming

- Specific commands are carried out immediately.
- Usually direct and exact instructions.
- e.g. read in data from this file.

Declarative programming

- Specific commands are carried out when needed.
- Expresses higher order goals / constraints.
- e.g. make sure this dataset is up to date every time I see it.

Use cases for reactive programming

❗ Use-less cases

This paradigm is rarely needed or used in R for data analysis.

💡 Useful cases

Reactive programming is useful for developing user applications (including web apps!).

In R, the shiny package uses reactive programming for writing app interactivity.

Outline

1 Reactive programming

2 Caching

3 targets

4 Reproducible environments

Caching: using rds

```r
if (file.exists("results.rds")) {
  res <- readRDS("results.rds")
} else {
  res <- compute_it() # a time-consuming function
  saveRDS(res, "results.rds")
}
```

Caching: using rds

```r
if (file.exists("results.rds")) {
  res <- readRDS("results.rds")
} else {
  res <- compute_it() # a time-consuming function
  saveRDS(res, "results.rds")
}
```

Equivalently…

```r
res <- xfun::cache_rds(
  compute_it(), # a time-consuming function
  file = "results.rds"
)
```

Caching: using rds

```
compute <- function(...) {
  xfun::cache_rds(rnorm(6), file = "results.rds", ...)
}
compute()
```

```
[1]  1.113 -0.163 -0.557 -0.428 -0.444 -0.503
```

```
compute()
```

```
[1]  1.113 -0.163 -0.557 -0.428 -0.444 -0.503
```

```
compute(rerun = TRUE)
```

```
[1] -0.5011  1.0217  0.1034 -1.7602 -0.0269  2.1689
```

```
compute()
```

```
[1] -0.5011  1.0217  0.1034 -1.7602 -0.0269  2.1689
```

Caching downloads

You often want to prevent downloads of the same data multiple times.

```r
download_data <- function(url) {
  dest_folder <- tempdir()
  sanitized_url <- stringr::str_replace_all(url, "/", "_")
  dest_file <- file.path(dest_folder, paste0(sanitized_url, ".rds"))
  if (file.exists(dest_file)) {
    data <- readRDS(dest_file)
  } else {
    data <- read_tsv(url, show_col_types = FALSE)
    saveRDS(data, dest_file)
  }
  data
}
bulldozers <- download_data("https://robjhyndman.com/data/Bulldozers.csv")
```

Caching: memoise

Caching stores results of computations so they can be reused.

```
library(memoise)
sq <- function(x) {
  print("Computing square of 'x'")
  x**2
}
memo_sq <- memoise(sq)
memo_sq(2)
```

```
[1] "Computing square of 'x'"
```

```
[1] 4
```

```
memo_sq(2)
```

```
[1] 4
```

Caching: Rmarkdown

~~~
```{r import-data, cache=TRUE}
d <- read.csv('my-precious.csv')
```


```{r analysis, dependson='import-data', cache=TRUE}
summary(d)
```
~~~

- Requires explicit dependencies or changes not detected.
- Changes to functions or packages not detected.
- Good practice to frequently clear cache to avoid problems.
- targets is a better solution

Caching: Quarto

~~~
```{r}
#| label: import-data
#| cache: true
d <- read.csv('my-precious.csv')
```


```{r}
#| label: analysis
#| dependson: import-data
#| cache: true
summary(d)
```
~~~

- Same problems as Rmarkdown
- targets is a better solution

Outline

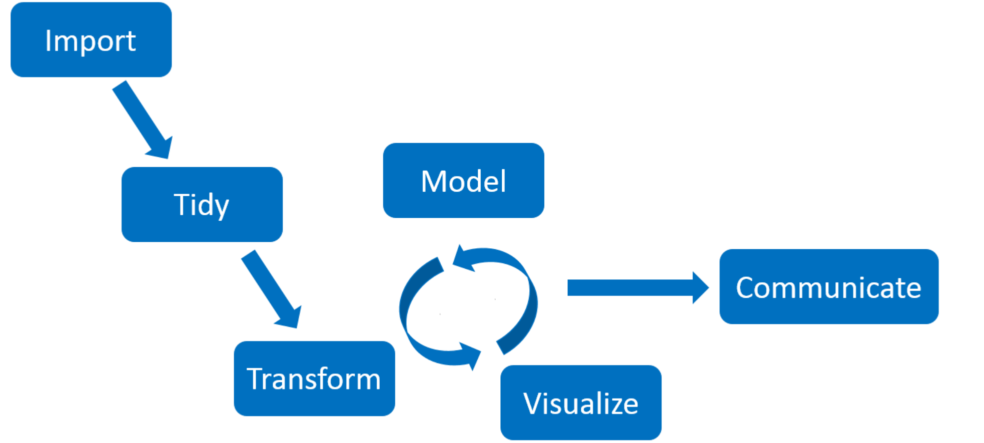
1. Reactive programming
2. Caching
3. targets
4. Reproducible environments

targets: reproducible computation at scale

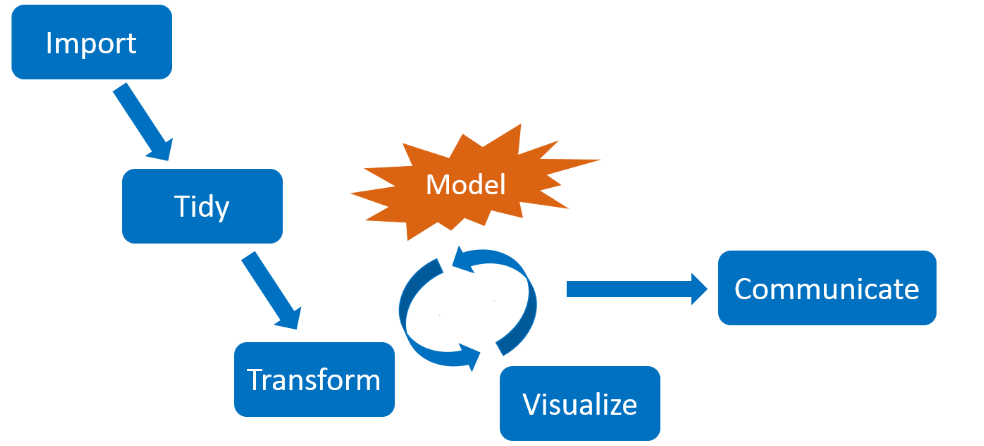
- Supports a clean, modular, function-oriented programming style.
- Learns how your pipeline fits together.
- Runs only the necessary computation.
- Abstracts files as R objects.
- Similar to Makefiles, but with R functions.

Some images from https://wlandau.github.io/targets-tutorial

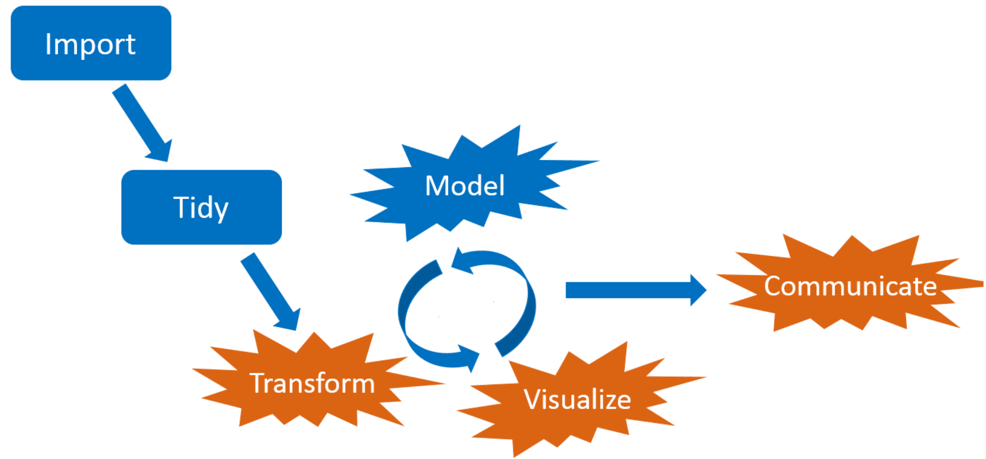
Interconnected tasks



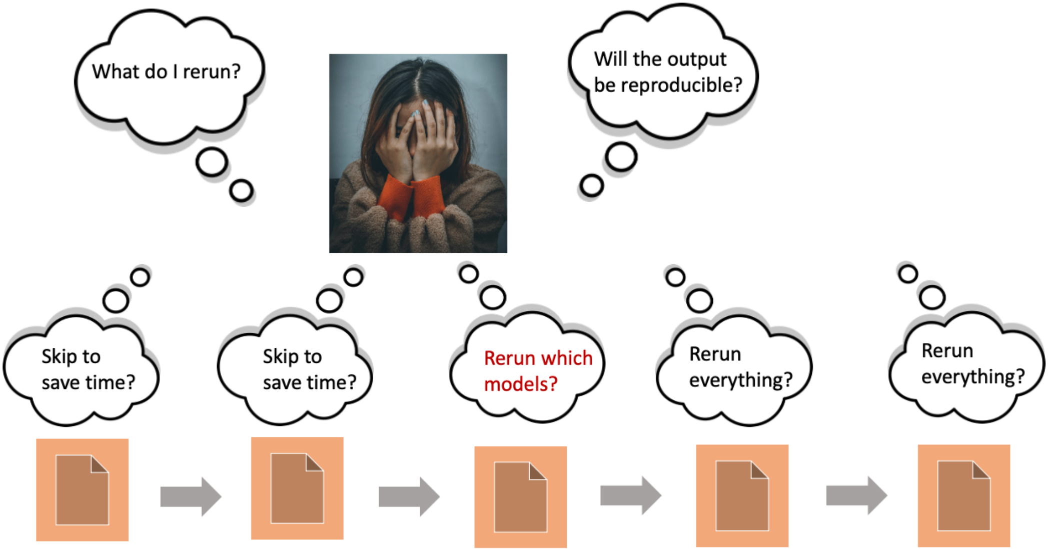
Interconnected tasks



Interconnected tasks



Dilemma: short runtimes or reproducible results?



What do I rerun?

Will the output be reproducible?

Skip to save time?

Skip to save time?

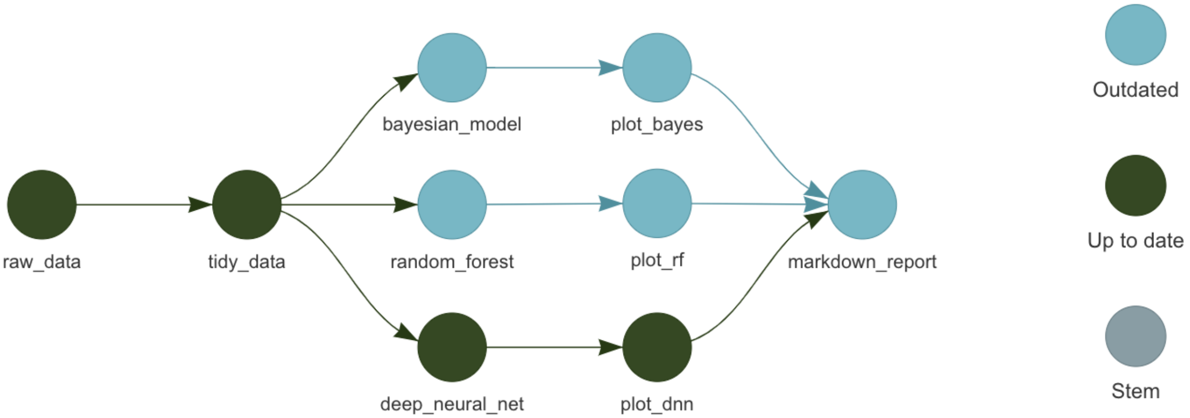
Rerun which models?

Rerun everything?

Rerun everything?

01-import.Rmd → 02-tidy.Rmd → 03-models.Rmd → 04-visualize.Rmd → 05-communicate.Rmd

Let a pipeline tool do the work



- Save time while ensuring computational reproducibility.
- Automatically skip tasks that are already up to date.

Typical project structure

no_targets.R

```r
library(tidyverse)
library(fable)
source("R/functions.R")
my_data <- read_csv("data/my_data.csv")
my_model <- model_function(my_data)
```

Typical project structure

no_targets.R

```r
library(tidyverse)
library(fable)
source("R/functions.R")
my_data <- read_csv("data/my_data.csv")
my_model <- model_function(my_data)
```

_targets.R

```r
library(targets)
tar_option_set(packages = c("tidyverse", "fable"))
tar_source() # source all files in R folder
list(
  tar_target(my_file, "data/my_data.csv", format = "file"),
  tar_target(my_data, read_csv(my_file)),
  tar_target(my_model, model_function(my_data))
)
```

Generate `_targets.R` in working directory

```r
library(targets)
tar_script()
```

Activity

- Set up a project using targets: `tar_script()`
- Add targets to generate a plot from the mtcars dataset, and fit a linear regression model.
- Make the project using `tar_make()`
- Visualize the pipeline using `tar_visnetwork()`

Useful targets commands

- `tar_make()` to run the pipeline.
- `tar_make(starts_with("fig"))` to run only targets starting with “fig”.
- `tar_read(object)` to read a target.
- `tar_load(object)` to load a target.
- `tar_load_everything()` to load all targets.
- `tar_manifest()` to list all targets
- `tar_visnetwork()` to visualize the pipeline.
- `tar_destroy()` to remove all targets.
- `tar_outdated()` to list outdated targets.

Debugging

Errored targets to return `NULL` so pipeline continues.

```
tar_option_set(error = "null")
```

Debugging

Errored targets to return `NULL` so pipeline continues.

```r
tar_option_set(error = "null")
```

See error messages for all targets.

```r
tar_meta(fields = error, complete_only = TRUE)
```

Debugging

Errored targets to return `NULL` so pipeline continues.

```
tar_option_set(error = "null")
```

See error messages for all targets.

```
tar_meta(fields = error, complete_only = TRUE)
```

See warning messages for all targets.

```
tar_meta(fields = warnings, complete_only = TRUE)
```

Debugging

- Try loading all available targets: `tar_load_everything()`. Then run the command of the errored target in the console.
- Pause the pipeline with `browser()`
- Use the debug option: `tar_option_set(debug = "target_name")`
- Save the workspaces:
  - `tar_option_set(workspace_on_error = TRUE)`
  - `tar_workspaces()`
  - `tar_workspace(target_name)`

Random numbers

- Each target runs with its own seed based on its name and the global seed from `tar_option_set(seed = ???)`
- So running only some targets, or running them in a different order, will not change the results.

Folder structure

```
├── .git/
├── .Rprofile
├── .Renviron
├── renv/
├── index.Rmd
├── _targets/
├── _targets.R
├── _targets.yaml
├── R/
├──── functions_data.R
├──── functions_analysis.R
├──── functions_visualization.R
├── data/
└──── input_data.csv
```

`_targets.R` with quarto

```
library(targets)
library(tarchetypes)                                                        ①
tar_source() # source all files in R folder
tar_option_set(packages = c("tidyverse", "fable"))
list(
  tar_target(my_file, "data/my_data.csv", format = "file"),
  tar_target(my_data, read_csv(my_file)),
  tar_target(my_model, model_function(my_data)),
  tar_quarto(report, "file.qmd", extra_files = "references.bib")           ②
  )
```

① Load `tarchetypes` package for quarto support.

② Add a quarto target.

Replace quarto chunks with `tar_read()` or `tar_load()`.

Chunk options

Chunk with regular R code

~~~
```{r}
#| label: fig-chunklabel
#| fig-caption: My figure
mtcars |>
  ggplot(aes(x = mpg, y = wt)) +
  geom_point()
```
~~~

Chunk options

Chunk with regular R code

~~~
```{r}
#| label: fig-chunklabel
#| fig-caption: My figure
mtcars |>
  ggplot(aes(x = mpg, y = wt)) +
  geom_point()
```
~~~

Chunk with targets

~~~
```{r}
#| label: fig-chunklabel
#| fig-caption: My figure
tar_read(my_plot)
```
~~~

Exercise

Add a quarto document to your targets project that includes the plot and the output from the linear regression model.

Outline

1. Reactive programming
2. Caching
3. targets
4. Reproducible environments

Reproducible environments

- To ensure that your code runs the same way on different machines and at different times, you need the computing environment to be the same.
  1. Operating system
  2. System components
  3. R version
  4. R packages
- Solutions for 1–4: Docker, Singularity, `containerit`, `rang`
- Solutions for 4: `packrat`, `checkpoint`, `renv`

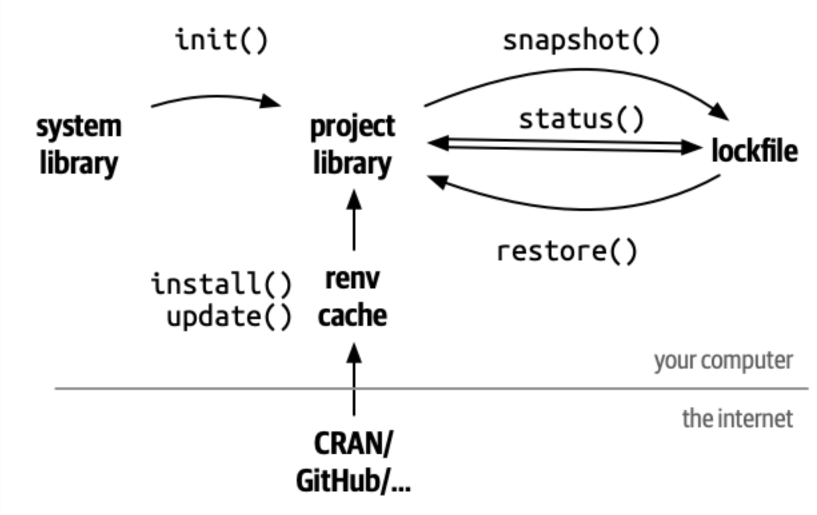
Reproducible environments

- Creates project-specific R environments.
- Uses a package cache so you are not repeatedly installing the same packages in multiple projects.
- Does not ensure R itself, system dependencies or the OS are the same.
- Not a replacement for Docker or Apptainer.

Reproducible environments

- Can use packages from CRAN, Bioconductor, GitHub, Gitlab, Bitbucket, etc.
- `renv::init()` to initialize a new project.
- `renv::snapshot()` to save state of project to `renv.lock`.
- `renv::restore()` to restore project as saved in `renv.lock`.

renv package



renv package

- `renv::install()` can install from CRAN, Bioconductor, GitHub, Gitlab, Bitbucket, etc.
- renv uses a package cache so you are not repeatedly installing the same packages in multiple projects.
- `renv::update()` gets latest versions of all dependencies from wherever they were installed from.
- `renv::deactivate(clean = TRUE)` will remove the renv environment.

Activity

Add renv to your targets project.

Example paper

ISSN: 0160-5682
JORS
JOURNAL OF THE OPERATIONAL RESEARCH SOCIETY
or
THE OPERATIONAL RESEARCH SOCIETY
Taylor & Francis
Taylor & Francis Group

Hyndman RJ, Rostami-Tabar B (2024) Forecasting interrupted time series, *Journal of the Operational Research Society*, in press.

bahmanrostamitabar/forecasting_interrupted_time_series