MONASH University

MONASH BUSINESS SCHOOL

ETC4500/ETC5450
Advanced R programming

Week 7: Reactive programming with targets and renv

Outline

Outline

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.

```
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!

```
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

Caching: using rds

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

Caching: using rds

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

Equivalently...

```
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]  1.534 -0.633  1.046 -0.325  1.565 -2.025
```

```
compute()
```

```
[1]  1.534 -0.633  1.046 -0.325  1.565 -2.025
```

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

```
[1] -1.547 -0.211 -2.169  0.381 -1.565  0.683
```

```
compute()
```

```
[1] -1.547 -0.211 -2.169  0.381 -1.565  0.683
```

Caching downloads

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

```
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