Advanced R for Econometricians (Summer 2022)

Functional R Programming — Notes and Examples

Martin Arnold, Jens Klenke

We need `tidyverse`.

```
library(tidyverse)
```

Slide 10: `purrr::map()`

- Note that `map()` always returns a list.
- There is a ton of variations of `map()`, see `?map`. We can't cover everything. But how cool is this one:

  ```
  # another example of the map_*() family
  map_if(ggplot2::economics, is.numeric, mean)
  ```

Slide 11: `purrr::map_dbl()` and `purrr::map_int()`

Of course, the '`*`' in `map_*()` must match the return type of the functions used for mapping!

Slide 14: `purrr::map_dbl()` — Producing Atomic Vectors

Solution to the task:

Please do not use a `for` loop!

As discussed in *Advanced R concepts*, everything we do in `R` involves function calls. `[[` is a primitive function—a fast C implementation for `list` subsetting. We can thus use it to iterate over `x` and subset by position/name.

```
# 1.
sapply(x, "[[", "x")
# 2.
sapply(x, "[[", 1)
```

Slide 15: `purrr::map_*()` — Producing Atomic Vectors

Note that `.default = NA` requires your subsequent code to be compatible with `NA` values.

Slide 19: `purrr::map_*()` — Exercises

1. Note that
   - `map(1:3, ~ runif(2))` evaluates `runif()` with argument `n = 2` in *every* iteration since `~` converts the formula to an anonymous function (`function(x) runif(2)`).
   - `map(1:3, runif(2))` evaluates `runif(2)` *only once* and cannot do the mapping as `runif(2)` cannot be transformed to a function, see `?purrr::as_mapper()`. Thus `NULL` is returned in every iteration.

2. ```
   library(ggplot2)

   trials_df <- tibble(p_value = map_dbl(trials, "p.value"))

   trials_df %>%
     ggplot(aes(x = p_value, fill = p_value < 0.05)) +
     geom_histogram(binwidth = .025) +
     ggtitle("Distribution of p-values for random Poisson data.")
   ```

3. ```
   models <- map(formulas, lm, data = mtcars)
   ```

Slide 20: Case Study — Model Fitting with purrr

Read in the data and split by `Drive`.

```
# (make sure to specify the correct path below)
cars2018 <- readr::read_csv("../data/cars2018.csv")
by_drive <- split(cars2018, cars2018$Drive)
```

- **purrr-style approach:**

  ```
  by_drive %>%
    map(~ lm(MPG ~ Cylinders, data = .x)) %>%
    map(coef) %>%
    map_dbl(2)
  ```

- **`apply()`-style:**

  ```
  models <- lapply(by_drive, function(data) lm(MPG ~ Cylinders, data = data))
  vapply(models, function(x) coef(x)[[2]], double(1))
  ```

- **`for()` loop:**

  ```
  slopes <- double(length(by_drive))
  for (i in seq_along(by_drive)) {
    model <- lm(MPG ~ Cylinders, data = by_drive[[i]])
    slopes[[i]] <- coef(model)[[2]]
  }
  slopes
  ```

Additional notes:

- `purrr` code is most accessible since each line encapsulates a single step and `map_*` conveys what is done in each step.
- Moving from `purrr` to base R we see that the number functions which iterate decreases while each iteration becomes increasingly complicated:
  - Using `purrr` we iterate 3 times (`map()`, `map()` and `map_dbl()`)
  - The `apply()` approach iterates twice (`lapply()` and `vapply()`)
  - Everything may be done in a single (but messy!) `for()` loop

Slide 26: `purrr::walk()`

Assignment to an environment is a commonly used side-effect:

```r
# ABC(1) => A <- 1, ABC(2) => B <- 2, ...
ABC <- function(x) {
  assign(LETTERS[x], x, envir = globalenv())
}

# both return invisibly:
invisible(lapply(1:3, ABC))
walk(1:3, ABC)

# `walk()` in functional-style 'workflow' (un-silenced :) )
1:26 %>% walk(., ABC) %>% cat(.)
```

Slide 27: `purrr::walk2()`

Writing to disc is another side-effect. We need to map over two arguments: an object and a path.

```r
cars2018 <- readr::read_csv("../data/cars2018.csv")

t <- tempfile()     # temporary path
dir.create(t)       # create folder at t

# list of splits
tm <- split(cars2018, cars2018$Transmission)

# generate paths
paths <- file.path(t, paste0(names(tm), ".csv"))

# walk over two arguments.
walk2(tm, paths, write.csv)

# inspect temporary folder
dir(t)
```

Slide 28: `purrr::imap()`

```r
cars2018 %>%
  select_if(is.numeric) %>%
  imap_chr(~ paste0("The Mean of ", .y, " is ", mean(.x)))
```

Slide 33: `purrr::pmap()` — Exercises

1. `modify()` is a shortcut for `x[[i]] <- f(x[[i]]); return(x)`. So every row is filled with its *first* value.

2. This a a good example of a quite complex operation which is relatively easy to comprehend by only looking at the code.

   ```r
   nm <- names(trans)
   cars2018[nm] <- map2(trans, cars2018[nm], function(f, var) f(var))
   ```

   - The functions in `trans` are intended to modify certain columns in `cars2018` (column names are provided as entry names in `trans`)
   - `map2()` runs over a named `list` of functions, `trans`, and a set of columns in `cars2018` which is obtained by subsetting using the function `names`

- An anonymous function is used to call transform columns using the corresponding functions in `trans`
- The results are used to replaced the original columns.

3. - Note that both approaches yield the same result
   - `map()` iterates over the variable names and calls the corresponding functions. Usage of `[[` and `.x` in the formula interface is pretty compact and conveys that columns of `cars2018` are modified.
   - Using the iteration over functions and variables in `map2()` allows us to use expressive variable names (`f`, `var`) which is not possible for the `map()` approach which iterates over names.
   - We could've also used the formula interface in `map2()` (which is even more compact) but the result looks rather cryptic:

   ```
   cars2018[nm] <- map2(nm, cars2018[nm], ~ .x(.y))
   ```

   - You should decide what you consider the most comprehensible!