Advanced R for Econometrician Summer 2022

Functional Programming — Notes and Examples

Martin Arnold

Slide 10: purrr::map()

Note that `map()` always returns a list.

There is a ton of variations of `map()`, see `?map`.

```
# 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 Task:

Please do not use a `for` loop! :-)

```
# 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 because `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)) +
```

```r
  geom_histogram(binwidth = .025) +
  ggtitle("Distribution of p-values for random Poisson data.")
```

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

Slide 20: Case Study Model Fitting with `purrr`

Read in the dataset and split by `Drive`.

```r
cars2018 <- readr::read_csv("../data/cars2018.csv")
by_drive <- split(cars2018, cars2018$Drive)
```

- **`purrr` style approach:**

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

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

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

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

  ```r
  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_*` allow to concisely convey 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 common 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))
```
```

```r
walk(1:3, ABC)

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

Slide 27: `purrr::walk2()`

Writing to disc is a side effect. We need a mapping 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, even from only looking at the code.

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

   - The functions in `trans` are intended to modify certain columns in `cars2018`
   - `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 apply the desired modification to the corresponding column
   - 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 with `map2()` (which is even more compact) but the result looks rather cryptic:

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

- You should decide what you consider the most comprehensible.

Slide 40: Case Study: Maximum Likelihood Estimation

Poisson Log-likelihood function factory:

```
ll_poisson <- function(x) {
  # components that depend on x only
  n <- length(x)
  sum_x <- sum(x)
  c <- sum(lfactorial(x))

  # manufactured function
  function(lambda) {
    log(lambda) * sum_x - n * lambda - c
  }
}
```

- The advantage of using a function factory here is fairly small, but there are two niceties:
  - We can precompute some values (quantities that dependent on the data only) in the factory, saving computation time in each iteration when running a maximisation algorithms on the manufactured function.
  - The two-level design better reflects the mathematical structure of the underlying problem.
- Of course, these advantages get bigger in more complex MLE problems, where we have multiple parameters and multiple data vectors.

Let's find the MLE for a Poisson random vector.

```
# random poisson sample
x1 <- rpois(100, 30)
# produce log-likelihood function using factory
llp <- ll_poisson(x1)

#### compute MLE ####

optimise(lprob_poisson, x = x1, c(0, 40), maximum = T)
# see slides for def. of lprob_poisson()

# more efficient to use the manufactured function:
optimise(llp, c(0, 40), maximum = T)
```