Functional Programming — Notes and Examples

Slide 10: `purrr::map()`

`map()` returns a list, obviously (always).

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

```
# another example
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 the subsequent code to be compatible with `NA` values.

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

1. `map(1:3, ~ runif(2))` evaluates `runif()` with `n = 2` in every iteration since `~` converts to an anonymous function. `map(1:3, runif(2))` evaluates `runif(2)` only once and cannot do mapping because `runif(2)` is not treated as a function. `NULL` is returned in every iteration.

2.
```
# from exercise
trials <- map(1:100, ~ t.test(rpois(10, 10), rpois(10, 7)))

# solution
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.
```
# from exercise
formulas <- list(
```

```r
  mpg ~ disp,
  mpg ~ disp + wt,
  mpg ~ I(1 / disp) + wt
)

# 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 as each line encapsulates a single step and the `purrr` helpers allow us to concisely describe what to do 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 can be done in one `for()` loop

Slide 26: `purrr::walk()`

Assignment (to an environment) is a side-effect:

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

```r
# Both return invisibly:
invisible(lapply(1:3, ABC))
walk(1:3, ABC)

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

Slide 27: `purrr::walk2()`

Writing to disc needs two arguments

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

t <- tempfile()
dir.create(t)

tm <- split(cars2018, cars2018$Transmission)

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

walk2(tm, paths, write.csv)

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. ```r
   # from exercise
   modify(cars2018, 1)
   ```

   Solution:

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

2. Good example of a quite complex operation which is relatively easy to comprehend, even from only looking at the code.

   ```r
   # from exercise
   trans <- list(
     Displacement = function(x) x * 0.0163871,
     Transmission = function(x) factor(x, labels = c("Automatic", "Manual", "CVT"))
   )

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

   - The functions in `trans` are to modify certain variables 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. ```r
   # from exercise
   mtcars[nm] <- map(nm, ~ trans[[.x]](cars2018[[.x]]))
   ```

   - Both lead to 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 convey 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()` 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:

     ```r
     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:

```r
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 in the factory, saving computation time in each iteration.
  - The two-level design better reflects the mathematical structure of the underlying problem.
- These advantages get bigger in more complex MLE problems, where you have multiple parameters and multiple data vectors.

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

```r
x1 <- rpois(100, 30)
llp <- ll_poisson(x1)

optimise(lprob_poisson, x = x1, c(0, 40), maximum = T)
# better:
optimise(llp, c(0, 40), maximum = T)
```
```