Functional programming in R

01-14-2020

You'll learn some of the powerful programming tools provided by `purrr`, one of the tidyverse core packages.

```r
library(tidyverse)
```

What is functional programming?

> In computer science, functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

To put is simple, it is a programming style without side effects.

```r
f <- function(x) {
  x$a <- 2
  x
}

x <- list(a = 1)
isTRUE(all.equal(f(x), x))
```

```
## [1] FALSE
```

```python
def f(x):
  x["a"] = 2
  return x

x = {"a": 1}
f(x) == x
```

```
## True
```

Is R a functional language?

R, at its heart, is a functional programming language. Most base R functions are pure, with a few notable exceptions:

- `library()` which loads a package, and hence modifies the search path.

- `setwd()`, `Sys.setenv()`, `Sys.setlocale()` which change the working directory, environment variables, and the locale, respectively.

- `plot()` and friends which produce graphical output.

- `write()`, `write.csv()`, `saveRDS()`, etc. which save output to disk.

- `options()` and `par()` which modify global settings.

- S4 related functions which modify global tables of classes and methods.

- Random number generators which produce different numbers each time you run them

Almost everything else in R are functions.

```r
x <- 1
y <- 2
x + y
```

```
## [1] 3
```

```r
`+`(x, y)
```

```
## [1] 3
```

```r
a <- 3:5
a[3]
```

```
## [1] 5
```

```r
`[`(a, 3)
```

```
## [1] 5
```

```r
z <- 1
`<-`(z, 1)
```

```r
for (i in 1:10) print(i)
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

```r
`for`(i, 1:10, print(i))
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

We could even change the body of previously defined functions

```
add <- function(x, y) x + y
add(3, 1)
```

```
## [1] 4
```

```
body(add) <- quote(x - y)
add(3, 1)
```

```
## [1] 2
```

Revisit for loops

Suppose we want to compute the sample medians for each column of the data frame

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

A naive way to do so

```
median(df$a)
```

```
## [1] -1.136651
```

```
median(df$b)
```

```
## [1] 0.04256461
```

```
median(df$c)
```

```
## [1] 0.1794314
```

```
median(df$d)
```

```
## [1] -0.225158
```

```
# a base R for loop
output <- vector("double", ncol(df))   # 1. output
for (i in seq_along(df)) {             # 2. sequence
  output[[i]] <- median(df[[i]])       # 3. body
}
output
```

```
## [1] -1.13665051  0.04256461  0.17943139 -0.22515799
```

Sometimes we would like to modify existing objects

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}

df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)

# or simply
for (i in seq_along(df)) {
  df[[i]] <- rescale01(df[[i]])
}
```

For loops vs functionals

The pattern of looping over a vector, doing something to each element and saving the results is so common that the `purrr` package provides a function `map` to do it.

The basic syntax for `map` is

```
map(ITERATABLE, A_FUNCTION)
```

`ITERATABLE`'s are objects like vectors, list and data frame.

```
map(df, median)
```

```
## $a
## [1] 0.1510703
##
## $b
## [1] 0.3801144
##
## $c
## [1] 0.3172644
##
## $d
## [1] 0.3966711
```

```
df %>% map(median)
```

```
## $a
## [1] 0.1510703
##
## $b
## [1] 0.3801144
##
## $c
## [1] 0.3172644
##
## $d
## [1] 0.3966711
```

```r
# the input is not necessary to be a data frame
map(c(9, 16, 25), sqrt)
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 5
```

```r
# pass additional arguments
map(df, mean, trim = 0.5)
```

```
## $a
## [1] 0.1510703
##
## $b
## [1] 0.3801144
##
## $c
## [1] 0.3172644
##
## $d
## [1] 0.3966711
```

```r
# preserve names
z <- list(x = 1:3, y = 4:5)
map(z, length)
```

```
## $x
## [1] 3
##
## $y
## [1] 2
```

The function `group_split` could be used to split a data frame into a list

```r
(models <- mtcars %>%
  group_split(cyl) %>%
  map(function(df) lm(mpg ~ wt, data = df)))
```

```
## [[1]]
##
## Call:
## lm(formula = mpg ~ wt, data = df)
##
## Coefficients:
## (Intercept)           wt
##      39.571       -5.647
```

```
## 
## 
## [[2]]
## 
## Call:
## lm(formula = mpg ~ wt, data = df)
## 
## Coefficients:
## (Intercept)            wt
##       28.41         -2.78
## 
## 
## [[3]]
## 
## Call:
## lm(formula = mpg ~ wt, data = df)
## 
## Coefficients:
## (Intercept)            wt
##      23.868        -2.192
```

```r
# or simply
(models <- mtcars %>%
  group_split(cyl) %>%
  map(~lm(mpg ~ wt, data = .)))
```

```
## [[1]]
## 
## Call:
## lm(formula = mpg ~ wt, data = .)
## 
## Coefficients:
## (Intercept)            wt
##      39.571        -5.647
## 
## 
## [[2]]
## 
## Call:
## lm(formula = mpg ~ wt, data = .)
## 
## Coefficients:
## (Intercept)            wt
##       28.41         -2.78
## 
## 
## [[3]]
## 
## Call:
## lm(formula = mpg ~ wt, data = .)
## 
## Coefficients:
## (Intercept)            wt
##      23.868        -2.192
```

Type-specific map

`map()` always returns a list, even if all the elements have the same flavor and are of length one. But in that case, you might prefer a simpler object.

- `map_lgl()` makes a logical vector.
- `map_int()` makes an integer vector.
- `map_dbl()` makes a double vector.
- `map_chr()` makes a character vector.

```
map_dbl(c(9, 16, 25), sqrt)
```

```
## [1] 3 4 5
```

Name and position shortcuts

The `map_` functions provide some shortcuts to extract elements from lists or vectors.

```
map_chr(list(LETTERS, letters), 2)
```

```
## [1] "B" "b"
```

```
mtcars %>%
  group_split(cyl) %>%
  map(~lm(mpg ~ wt, data = .)) %>%
  map(summary) %>%
  map_dbl("r.squared")
```

```
## [1] 0.5086326 0.4645102 0.4229655
```

Map more than one arguments

```
a <- 1:5
b <- 1:5
map2_dbl(a, b, `^`)
```

```
## [1]    1    4   27  256 3125
```

```
# or equivalently
map2_dbl(a, b, ~ .x ^ .y)
```

```
## [1]    1    4   27  256 3125
```

Enumerating over the input

```r
a <- c(11, 12, 13)
b <- c(1, 2, 3)
a %>% imap_dbl(~ .x + b[.y])
```

```
## [1] 12 14 16
```

Filtering

```r
a <- 11:20
m <- list(lowers = letters, uppers = LETTERS)

a %>% pluck(1)  # equivalent to a[1]
```

```
## [1] 11
```

```r
m %>% pluck("lowers")   # equivalent to m["lower"]
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

```r
m %>% pluck("lowers", 2)  # equivalent to m["lower"][2]
```

```
## [1] "b"
```

Keep and drop elements

```r
is_even <- function(x) x %% 2 == 0
a %>% keep(is_even)
```

```
## [1] 12 14 16 18 20
```

```r
# or equivalently
a %>% keep(~ . %% 2 == 0)
```

```
## [1] 12 14 16 18 20
```

```r
m %>% keep(~ "A" %in% .)
```

```
## $uppers
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

```r
a %>% keep(~!is_even(.))
```

```
## [1] 11 13 15 17 19
```

```r
a %>% keep(negate(is_even))
```

```
## [1] 11 13 15 17 19
```

```r
a %>% discard(is_even)
```

```
## [1] 11 13 15 17 19
```

Summerizing

```r
# is every element in a even?
a %>% every(is_even)
```

```
## [1] FALSE
```

```r
# are some elements in a even?
a %>% some(is_even)
```

```
## [1] TRUE
```

```r
a %>% has_element(2)  # 2 is a double
```

```
## [1] FALSE
```

```r
a %>% has_element(2L)
```

```
## [1] FALSE
```

```r
# the first even number is
a %>% detect(is_even)
```

```
## [1] 12
```

```r
# the first even number divisible by 7 is
a %>% detect(~ . %% 7 == 0)
```

```
## [1] 14
```

```r
# get the index of the first element  divisible by 7, then extract it
a %>% detect_index(~ . %% 7 == 0) %>% pluck(a, .)
```

```
## [1] 14
```

Modify elements selectively

```
# a %>% modify(~ . + 1) results in an error because a is an integer vector
a %>% modify(~ . + 1L)
```

```
##  [1] 12 13 14 15 16 17 18 19 20 21
```

```
a %>% modify_if(is_even, ~ -.)
```

```
##  [1]  11 -12  13 -14  15 -16  17 -18  19 -20
```

```
a %>% modify_if(~ !is_even(.), ~ -.)
```

```
##  [1] -11  12 -13  14 -15  16 -17  18 -19  20
```

```
a %>% modify_at(c(1, 3, 5), ~ 0L)
```

```
##  [1]  0 12  0 14  0 16 17 18 19 20
```

Reshape lists

```
m %>% flatten_chr()
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z" "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L"
## [39] "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
(m1 <- m %>% map(~ .[1:5]))
```

```
## $lowers
## [1] "a" "b" "c" "d" "e"
##
## $uppers
## [1] "A" "B" "C" "D" "E"
```

```
m1 %>% transpose()
```

```
## [[1]]
## [[1]]$lowers
## [1] "a"
##
## [[1]]$uppers
## [1] "A"
##
##
## [[2]]
## [[2]]$lowers
## [1] "b"
```

```
## 
## [[2]]$uppers
## [1] "B"
## 
## 
## [[3]]
## [[3]]$lowers
## [1] "c"
## 
## [[3]]$uppers
## [1] "C"
## 
## 
## [[4]]
## [[4]]$lowers
## [1] "d"
## 
## [[4]]$uppers
## [1] "D"
## 
## 
## [[5]]
## [[5]]$lowers
## [1] "e"
## 
## [[5]]$uppers
## [1] "E"
```

References

- R for Data Science https://r4ds.had.co.nz/iteration.html
- purrr tutorial https://jennybc.github.io/purrr-tutorial/
- purrr cheatsheet https://github.com/rstudio/cheatsheets/blob/master/purrr.pdf