

Functional programming in R

01-16-2020

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

```
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 it simply, it is a programming style without side effects.

```
f <- function(x) {  
  x$a <- 2  
  x  
}  
  
x <- list(a = 1)  
isTRUE(all.equal(f(x), x))
```

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

```
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.

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

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

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

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

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

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

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

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

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

```
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
```

```
`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] -0.4432872
```

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

```
## [1] -0.09424343
```

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

```
## [1] 0.2235735
```

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

```
## [1] 0.06092786
```

```
# 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] -0.44328719 -0.09424343 0.22357347 0.06092786
```

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] 0.5170176
##
## $b
## [1] 0.5628878
##
## $c
## [1] 0.742106
##
## $d
## [1] 0.6425789

```

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

```

## $a
## [1] 0.5170176
##
## $b
## [1] 0.5628878
##
## $c
## [1] 0.742106
##
## $d
## [1] 0.6425789

```

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

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

```
fab_norm <- function(p) {  
  x <- c(1, 1, 2, 3, 5, 8)  
  (sum(abs(x)^p))^(1/p)  
}
```

```
fab_norm(1)
```

```
## [1] 20
```

```
fab_norm(1.5)
```

```
## [1] 12.43168
```

```
fab_norm(2)
```

```
## [1] 10.19804
```

```
# but fab_norm(c(1, 1.5, 2)) does not work  
c(1, 1.5, 2) %>% map(fab_norm)
```

```
## [[1]]  
## [1] 20  
##  
## [[2]]  
## [1] 12.43168  
##  
## [[3]]  
## [1] 10.19804
```

```
mean(df$a, trim = 0.5)
```

```
## [1] 0.5170176
```

```
mean(df$b, trim = 0.5)
```

```
## [1] 0.5628878
```

```
mean(df$c, trim = 0.5)
```

```
## [1] 0.742106
```

```
mean(df$d, trim = 0.5)
```

```
## [1] 0.6425789
```

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

```
## $a  
## [1] 0.5170176  
##  
## $b  
## [1] 0.5628878  
##  
## $c  
## [1] 0.742106  
##  
## $d  
## [1] 0.6425789
```

```
# equivalently  
df %>% map(mean, trim = 0.5)
```

```
## $a  
## [1] 0.5170176  
##  
## $b  
## [1] 0.5628878  
##  
## $c  
## [1] 0.742106  
##  
## $d  
## [1] 0.6425789
```

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

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

```
# equivalently
z %>% map(length)
```

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

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

```
(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
```

```
# 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.

```
list(LETTERS, letters) %>% map_chr(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 argument

Two arguments

```
a <- c(7, 8, 9)
b <- c(2, 3, 4)

map2(a, b, choose)
```

```
## [[1]]
## [1] 21
##
## [[2]]
## [1] 56
##
## [[3]]
## [1] 126
```

```
a %>% map2(b, choose)
```

```
## [[1]]
## [1] 21
##
## [[2]]
## [1] 56
##
## [[3]]
## [1] 126
```

```
b %>% map2(a, ., choose)
```

```
## [[1]]
## [1] 21
##
## [[2]]
## [1] 56
##
## [[3]]
## [1] 126
```

```
# or equivalently  
map2(a, b, ~ choose(.x, .y))
```

```
## [[1]]  
## [1] 21  
##  
## [[2]]  
## [1] 56  
##  
## [[3]]  
## [1] 126
```

```
map2(a, b, function(x, y) choose(x, y))
```

```
## [[1]]  
## [1] 21  
##  
## [[2]]  
## [1] 56  
##  
## [[3]]  
## [1] 126
```

More than two arguments

```
l <- list(  
  a = c(1, 2, 3),  
  b = c(5, 4, 8),  
  c = c(4, 7, 9))  
  
l %>% pmap(function(a, b, c) a / (b + c))
```

```
## [[1]]  
## [1] 0.1111111  
##  
## [[2]]  
## [1] 0.1818182  
##  
## [[3]]  
## [1] 0.1764706
```

```
# or equivalently  
l %>% pmap(~ ..1 / (..2 + ..3))
```

```
## [[1]]  
## [1] 0.1111111  
##  
## [[2]]  
## [1] 0.1818182  
##  
## [[3]]  
## [1] 0.1764706
```

Enumerating over the input

```
a <- c(11, 12, 13)
b <- c(4, 5, 6)
a %>% imap_dbl(~ .x + b[.y])
```

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

```
# equivalently
a %>% imap_dbl(function(v, i) v + b[i])
```

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

Filtering

```
a <- 11:20
m <- list(lower = letters, upper = LETTERS)
a %>% pluck(1) # equivalent to a[1]
```

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

```
m %>% pluck("lower") # 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"
```

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

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

Keep and drop elements

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

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

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

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

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

```
## $supers
## [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 %>% keep(~!is_even(.))
```

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

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

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

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

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

Summerizing

```
# is every element in `a` even?
a %>% every(is_even) # maybe equivalent to all(is_even(a))
```

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

```
# are some elements in `a` even?
a %>% some(is_even) # maybe equivalent to any(is_even(a))
```

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

```
a %>% has_element(12) # 12 is a double
```

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

```
a %>% has_element(12L)
```

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

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

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

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

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

```
# 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), ~ . + 2L)
```

```
## [1] 13 12 15 14 17 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 %>% pluck("lowers", 2)

## [1] "b"

m1 %>% transpose() %>% map(~ stringr::str_c(.$lowers, .$uppers))

## [[1]]
## [1] "aA"
##
## [[2]]
## [1] "bB"
##
## [[3]]
## [1] "cC"
##
## [[4]]
## [1] "dD"
##
## [[5]]
## [1] "eE"

```

Add elements to lists

```

li <- list(a = 1, b = 2)
li %>%
  append(list(c = 3, d = 3)) %>%
  prepend(list(x = 0, y = 0)) %>%
  splice(u = 3, v = 4)

## $x
## [1] 0
##
## $y
## [1] 0
##
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
##
## $d
## [1] 3
##
## $u
## [1] 3
##
## $v
## [1] 4

```

Cross

The `map` function applies to the inputs elementwisely. Sometimes, we wish to consider all combinations of the inputs. The `cross*` functions are designed to handle those situations.

`cross2` usually is used together with `lift`

```
a <- c(7, 8, 9)
b <- c(2, 3, 4)
choose(7, 3)
```

```
## [1] 35
```

```
lift(choose)(list(7, 3))
```

```
## [1] 35
```

```
cross2(a, b) %>% map_dbl(lift(choose))
```

```
## [1] 21 28 36 35 56 84 35 70 126
```

```
li <- list(a = c(7, 8, 9),
           b = c(2, 3, 4))
li %>%
  set_names(c("n", "k")) %>%
  cross() %>%
  map(lift(choose))
```

```
## [[1]]
## [1] 21
##
## [[2]]
## [1] 28
##
## [[3]]
## [1] 36
##
## [[4]]
## [1] 35
##
## [[5]]
## [1] 56
##
## [[6]]
## [1] 84
##
## [[7]]
## [1] 35
##
## [[8]]
## [1] 70
##
## [[9]]
## [1] 126
```

```

dat <- tibble(
  x = seq(-1, 1, length = 50),
  y = seq(-1, 1, length = 50))

# we wish to plot a heat map of sqrt(x^2 + y^2)

dat %>% mutate(z = sqrt(x^2 + y^2)) # wrong

```

```

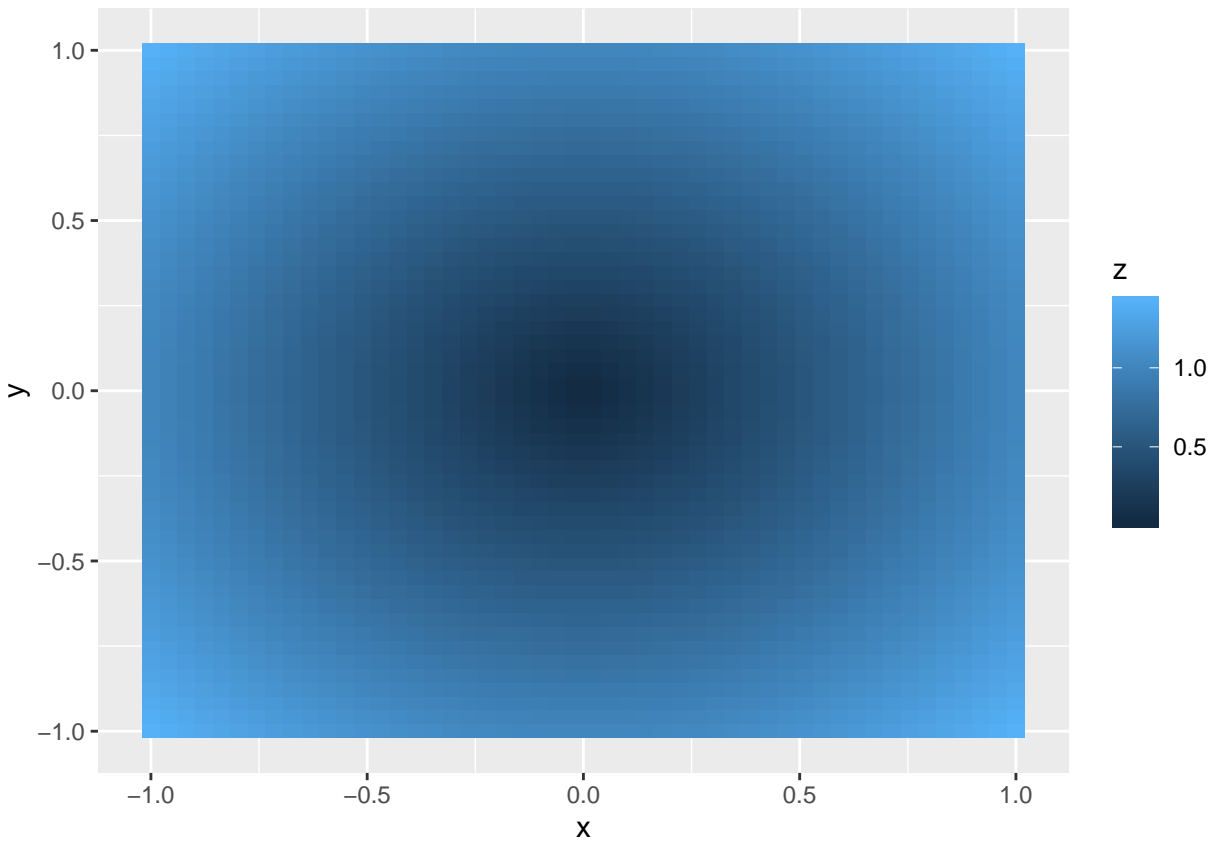
## # A tibble: 50 x 3
##       x       y       z
##   <dbl> <dbl> <dbl>
## 1 -1     -1     1.41
## 2 -0.959 -0.959 1.36
## 3 -0.918 -0.918 1.30
## 4 -0.878 -0.878 1.24
## 5 -0.837 -0.837 1.18
## 6 -0.796 -0.796 1.13
## 7 -0.755 -0.755 1.07
## 8 -0.714 -0.714 1.01
## 9 -0.673 -0.673 0.952
## 10 -0.633 -0.633 0.895
## # ... with 40 more rows

```

```

dat %>%
  cross_df() %>%
  mutate(z = sqrt(x^2 + y^2)) %>%
  ggplot(aes(x, y, fill = z)) + geom_tile()

```

Reduce iterable

The `reduce` function combines the values of a vector by iteratively applying a binary function.

```
reduce(1:4, `+`)
```

```
## [1] 10
```

```
# it means ((1 + 2) + 3) + 4  
reduce(1:4, `+`, .dir = "backward")
```

```
## [1] 10
```

```
# it means 1 + (2 + (3 + 4))  
reduce(4:1, choose)
```

```
## [1] 6
```

```
# it means choose(choose(choose(4, 3), 2), 1)
```

```
# shortcircuit a reduction
sum_until <- function(x, y) {
  out <- x + y
  if (out > 30) {
    done(out)
  } else {
    out
  }
}

reduce(1:10, sum_until)
```

```
## [1] 36
```

accumulate works similarly but keep all the immediate calculations.

```
accumulate(1:4, `+`)
```

```
## [1] 1 3 6 10
```

```
accumulate(1:10, sum_until)
```

```
## [1] 1 3 6 10 15 21 28 36
```

Nesting

```
mtcars %>%
  nest(data = -cyl)
```

```
## # A tibble: 3 x 2
##   cyl      data
##   <dbl> <list<df[,10]>>
## 1     6   [7 x 10]
## 2     4   [11 x 10]
## 3     8   [14 x 10]
```

```
# or more readable
mtcars %>%
  group_by(cyl) %>%
  nest() %>%
  mutate(model = data %>% map(~ lm(mpg ~ wt, data = .))) %>%
  transmute(r_sq = model %>% map(summary) %>% map_dbl("r.squared"))
```

```
## # A tibble: 3 x 2
## # Groups:   cyl [3]
##   cyl r_sq
##   <dbl> <dbl>
## 1     6 0.465
## 2     4 0.509
## 3     8 0.423
```

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>