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.

```
library(tidyverse)
```

Is R a functional language?

> R, at its heart, is a functional programming language. This means that it provides many tools for the creation and manipulation of functions.

Everything in R is function

```
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
```

```
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] -0.2340629
```

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

```
## [1] -0.140444
```

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

```
## [1] 0.2975208
```

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

```
## [1] -0.3248857
```

```
# 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] -0.2340629 -0.1404440  0.2975208 -0.3248857
```

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 family of functions to do it for you.

- `map()` makes a list.
- `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(df, median)
```

```
##         a         b         c         d
## 0.6037963 0.5202592 0.5759951 0.4250358
```

```
df %>% map_dbl(median)
```

```
##         a         b         c         d
## 0.6037963 0.5202592 0.5759951 0.4250358
```

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

```
##         a         b         c         d
## 0.6037963 0.5202592 0.5759951 0.4250358
```

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

```
## x y
## 3 2
```

```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
```

References

- R for Data Science https://r4ds.had.co.nz/iteration.html