R para Ciência de Dados 2

Purrr

Agosto de 2021

Motivação

Além de lidar com listas, o `{purrr}` lida com iterações, um dos padrões mais comuns em qualquer tarefa de programação (independentemente da linguagem).

Apesar de não ser tão evidente, todos os data frames do R não passam de listas com algumas propriedades específicas, então saber lidar com elas pode ser útil em diversos lugares.

Ainda pode ser difícil entender como isso funciona, mas, além de números, strings e datas, é possível colocar listas nas colunas de um data frame.

A sintaxe do `{purrr}` é capaz de mudar para sempre o modo como você funciona, ensinando padrões robustos de programação (sem efeitos colaterais e sem repetição de código).

Introdução

- Listas são como vetores, com a diferença de que elas não precisam ser homogêneas (seus elementos podem ter qualquer tipo)

```
l <- list(
  um_numero = 123,
  um_vetor = c(TRUE, FALSE, TRUE),
  uma_string = "abc",
  uma_lista = list(1, 2, 3)
)
str(l)
```

```
#> List of 4
#>  $ um_numero : num 123
#>  $ um_vetor  : logi [1:3] TRUE FALSE TRUE
#>  $ uma_string: chr "abc"
#>  $ uma_lista :List of 3
#>   ..$ : num 1
#>   ..$ : num 2
#>   ..$ : num 3
```

Estrutura

```
l
```

```
#> $um_numero
#> [1] 123
#>
#> $um_vetor
#> [1]  TRUE FALSE  TRUE
#>
#> $uma_string
#> [1] "abc"
#>
#> $uma_lista
#> $uma_lista[[1]]
#> [1] 1
#>
#> $uma_lista[[2]]
#> [1] 2
#>
#> $uma_lista[[3]]
#> [1] 3
```

Indexação

- Para acessar os elementos de uma lista precisamos tomar cuidado com a diferença entre `[]` e `[[]]` (ou `purrr::pluck()`): o primeiro acesa uma posição, enquanto o segundo acessa um elemento

```
l[3]
```

```
#> $uma_string
#> [1] "abc"
```

```
l[[3]]
```

```
#> [1] "abc"
```

```
library(purrr)
pluck(l, 4, 2) # Indexação profunda
```

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

Iterações

- Iteração não é nada mais do que a repetição de um trecho de código várias vezes, normalmente associada a um *loop* (laço)

```
vec <- 1:5
for (i in seq_along(vec)) {
  vec[i] <- vec[i] + 10
}
vec
```

```
#> [1] 11 12 13 14 15
```

- Note como a única coisa que fazemos é aplicar uma operação em cada elemento do vetor (`vec[i] + 10`)
- Identificamos algumas estruturas: **entrada** (vetor de 1 a 5) e **função** (somar 10 a cada elemento)

Simplificando

- O pacote `{purrr}` nos permite simplificar iterações e integrá-las a pipelines do `{tidyverse}`: a função `map()` realiza uma iteração recebendo apenas uma entrada e uma função

```
vec <- 1:5
soma_dez <- function(x) {
  x + 10
}

l <- map(vec, soma_dez)
str(l)
```

```
#> List of 5
#>  $ : num 11
#>  $ : num 12
#>  $ : num 13
#>  $ : num 14
#>  $ : num 15
```

Achatamento

- `map()` sempre retorna uma lista independente do objeto recebido porque ela não pode assumir nada sobre o resultado
- Se quisermos achatar os resultados só precisamos chamar uma função da família `map_***()` (onde `***` é a abreviação do tipo do objeto que deve ser retornado)

```
map_dbl(vec, soma_dez)
```

```
#> [1] 11 12 13 14 15
```

- Os tipos possíveis são: `dbl` (números), `chr` (strings), `dfc` (*data frame columns*), `dfr` (*data frame rows*), `int` (inteiros) e `lgl` (lógicos)

```
map_chr(vec, soma_dez)
```

```
#> [1] "11.000000" "12.000000" "13.000000" "14.000000" "15.000000"
```

Funções

- Para passar outros argumentos **fixos** a uma função, basta adicioná-los ao final da chamada de `map()`

```
soma_n <- function(x, n) {
  x + n
}
map_dbl(vec, soma_n, n = 3)
```

```
#> [1] 4 5 6 7 8
```

- Para simplificar funções curtas, podemos usar uma notação **lambda** na qual `.x` representa onde deve ser inserido o elemento atual da iteração (o valor "iterante") e ~ indica a declaração da função

```
map_dbl(vec, ~3+.x)
```

```
#> [1] 4 5 6 7 8
```

Duas entradas

- Se for necessário iterar em duas listas ou vetores, basta usar `map2()`

```
strings <- c("oiii", "como vai", "tchau")
padroes <- c("i+",   "(.o){2}",  "[au]+$")
map2_chr(strings, padroes, stringr::str_extract)
```

```
#> [1] "iii"  "como" "au"
```

- A notação lambda funciona exatamente do mesmo modo, com `.x` e `.y` representando o primeiro e o segundo elementos da iteração

```
map2_chr(strings, padroes, ~stringr::str_c(.y, " | ", .x))
```

```
#> [1] "i+ | oiii"          "(.o){2} | como vai" "[au]+$ | tchau"
```

List-columns

- *List-columns* são colunas nas quais cada elemento é uma lista (ou até mesmo uma tabela completa)
- A função `str_split()`, por exemplo, retorna uma lista contendo os pedaços da string original quebrada com um regex

```
imdb %>%
  mutate(split_generos = str_split(generos, "\\|")) %>%
  select(titulo, generos, split_generos)
```

```
#> # A tibble: 3,713 × 3
#>   titulo                                     generos                  split_generos
#>   <chr>                                      <chr>                    <list>
#> 1 Avatar                                     Action|Adventure|Fanta… <chr [4]>
#> 2 Pirates of the Caribbean: At World's End Action|Adventure|Fanta… <chr [3]>
#> 3 The Dark Knight Rises                      Action|Thriller          <chr [2]>
#> 4 John Carter                                Action|Adventure|Sci-Fi <chr [3]>
#> 5 Spider-Man 3                               Action|Adventure|Roman… <chr [3]>
#> # … with 3,708 more rows
```

Unnest

- Usando `tidyr::unnest()` é possível "abrir" a list-column de modo que cada linha fique com um de seus elementos (neste caso, o título do filme vai ser repetido uma vez para cada gênero ao qual o filme pertence)

```
library(tidyr)
imdb %>%
  mutate(split_generos = str_split(generos, "\\|")) %>%
  select(titulo, split_generos) %>%
  unnest(split_generos)
```

```
#> # A tibble: 10,612 × 2
#>   titulo                                    split_generos
#>   <chr>                                     <chr>
#> 1 Avatar                                    Action
#> 2 Avatar                                    Adventure
#> 3 Avatar                                    Fantasy
#> 4 Avatar                                    Sci-Fi
#> 5 Pirates of the Caribbean: At World's End  Action
#> # … with 10,607 more rows
```

Nest

- A operação inversa do `unnest()` é o `nest()`, que transforma um grupo de linhas em uma list-column

```
imdb %>%
  mutate(split_generos = str_split(generos, "\\|")) %>%
  select(titulo, split_generos) %>%
  unnest(split_generos) %>%
  group_by(titulo) %>%
  nest(generos = c(split_generos))
```

```
#> # A tibble: 3,711 × 2
#> # Groups:   titulo [3,711]
#>   titulo                                    generos
#>   <chr>                                     <list>
#> 1 Avatar                                    <tibble [4 × 1]>
#> 2 Pirates of the Caribbean: At World's End  <tibble [3 × 1]>
#> 3 The Dark Knight Rises                     <tibble [2 × 1]>
#> 4 John Carter                               <tibble [3 × 1]>
#> 5 Spider-Man 3                              <tibble [3 × 1]>
#> # … with 3,706 more rows
```

Voltando ao {purrr}

- Trazendo o assunto de volta para o `{purrr}`, o pacote nos permite operar com facilidade em list-columns justamente pela sua capacidade de tratar listas

```
imdb %>%
  mutate(split_generos = str_split(generos, "\\|")) %>%
  select(titulo, split_generos) %>%
  unnest(split_generos) %>%
  group_by(titulo) %>%
  nest(generos = c(split_generos)) %>%
  ungroup() %>%
  mutate(n_generos = map_dbl(generos, nrow))
```

```
#> # A tibble: 3,711 × 3
#>   titulo                                    generos           n_generos
#>   <chr>                                     <list>                <dbl>
#> 1 Avatar                                    <tibble [4 × 1]>          4
#> 2 Pirates of the Caribbean: At World's End  <tibble [3 × 1]>          3
#> 3 The Dark Knight Rises                     <tibble [2 × 1]>          2
#> 4 John Carter                               <tibble [3 × 1]>          3
#> 5 Spider-Man 3                              <tibble [3 × 1]>          3
#> # … with 3,706 more rows
```

Para saber mais

O `{purrr}` simplifica loops, impede efeitos colaterais e ainda deixa seu código mais bonito! Para entender melhor como esse pacote incrível funciona, veja mais nos links abaixo:

- A Magia de Purrr
- Webinar de purrr avançado
- Exemplos com purrr
- Purrr cheat sheet
- Purrr tutorial
- R for Data Science: Iteration