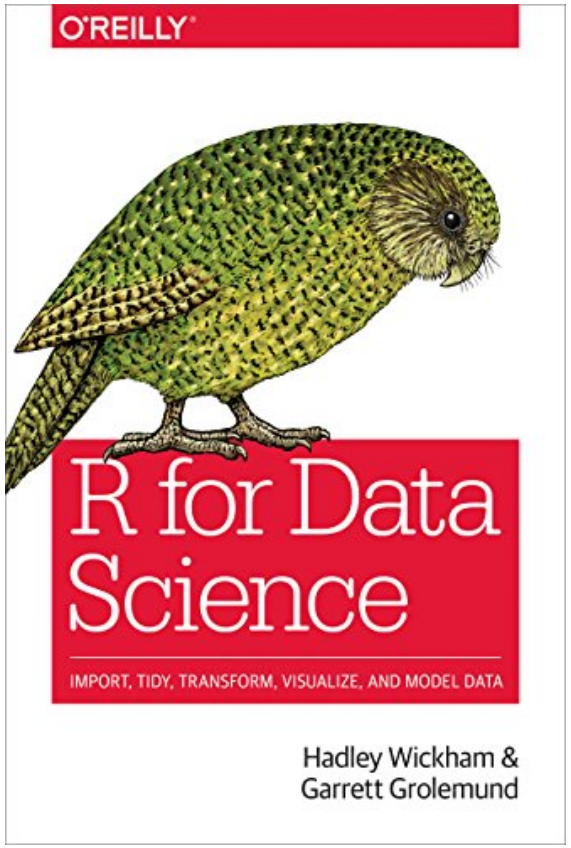
Conhecendo o Tidyverse

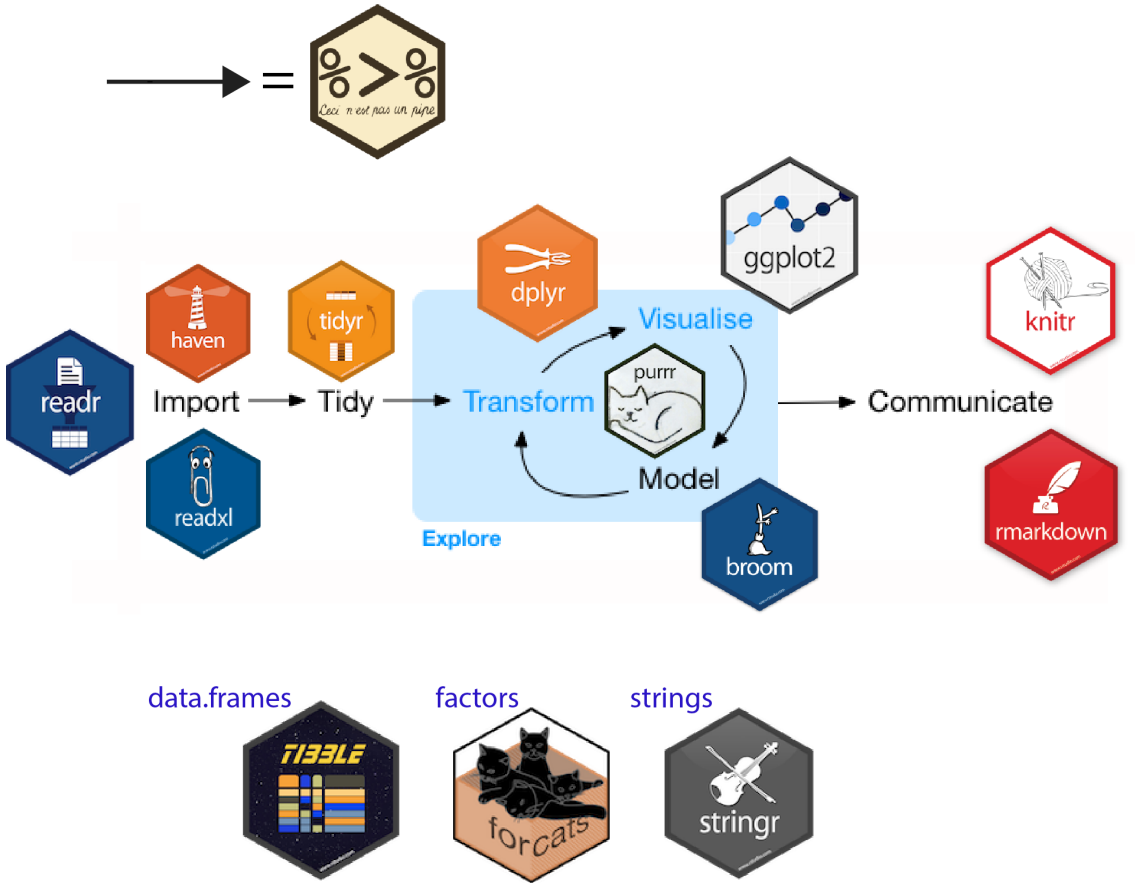
purrr

Fernando de Pol Mayer (LEG/DEST/UFPR)
2022-04-05

R for Data Science



R for Data Science, a principal referência sobre o emprego da linguagem R em ciência de dados.



Workflow de ciência de dados com o {tidyverse}. Fonte: https://oliviergimenez.github.io/intro_tidyverse/#7

{purrr}

Um overview do {purrr}

Utilidade da programação funcional

- **Programação funcional** é quando uma função chama uma outra função para ser aplicada repetidamente percorrendo elementos de um objeto.
- O recurso é útil para fazer tarefas em **série** ou **batelada**.
- Exemplos:
  - Importar vários arquivos em um diretório.
  - Tratar as imagens de um diretório.
  - Realizar a mesma análise para todas as UFs.
  - Fazer o ajuste de regressão polinomial de grau 1 até 5.

Programação funcional no R

- No R básico, a programação funcional é com a família `apply`.
- No `tidyverse` a programação funcional está no `purrr`.
- A principal é a função `map()` e suas variações.
- Além disso, tem
  - Funções para tratamento de excessões.
  - Acumular e reduzir.
  - Aninhar e aplanar objetos.

- Cheatsheet: https://github.com/rstudio/cheatsheets/blob/main/purrr.pdf

Funções `map*()`

Aplicando uma função em série

```
library(tidyverse)
x <- list(1:5,
          c(4, 5, 7),
          c(98, 34, -10),
          c(2, 2, 2, 2, 2))
map(x, sum)
```

```
# [[1]]
# [1] 15
#
# [[2]]
# [1] 16
#
# [[3]]
# [1] 122
#
# [[4]]
# [1] 10
```

Variações da função `map()`

- Sufixo para o tipo de coerção do resultado: `_chr`, `_int`, `_dbl`, `_lgl`, `_df`, `_dfc` e `_dfr`.
- Sufixo para o tipo de atuação: `_if` e `_at`.

```
ls("package:purrr") %>%
    str_subset("^map_")
```

```
#  [1] "map_at"    "map_call"  "map_chr"   "map_dbl"
#  [5] "map_depth" "map_df"    "map_dfc"   "map_dfr"
#  [9] "map_if"    "map_int"   "map_lgl"   "map_raw"
```

```
map_dbl(x, sum)
```

```
# [1]  15  16 122  10
```

```
map_chr(x, paste, collapse = " ")
```

```
# [1] "1 2 3 4 5" "4 5 7"     "98 34 -10" "2 2 2 2 2"
```

Funções `map*()`

Aplicando uma função em série

```
x <- list(1:5,
          c(4, 5, 7),
          c(98, 34, -10),
          c(2, 2, 2, 2, 2))
map(x, sum)
```

```
# [[1]]
# [1] 15
#
# [[2]]
# [1] 16
#
# [[3]]
# [1] 122
#
# [[4]]
# [1] 10
```

Note que a função `map()` nada mais é do que um "atalho" para essa programação usando `for()`

```
simple_map <- function(x, f, ...) {
    out <- vector("list", length(x))
    for (i in seq_along(x)) {
        out[[i]] <- f(x[[i]], ...)
    }
    out
}

simple_map(x, sum)
```

```
# [[1]]
# [1] 15
#
# [[2]]
# [1] 16
#
# [[3]]
# [1] 122
#
# [[4]]
# [1] 10
```

A grande diferença é que as funções `map_*()` são escritas de maneira muito mais eficiente na linguagem C.

Funções anônimas

```
map_dbl(x, function(x) length(unique(x)))
```

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

```
map_dbl(x, ~ length(unique(.x)))
```

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

```
set.seed(1)
map(1:3, function(x) runif(n = x))
```

```
# [[1]]
# [1] 0.2655087
#
# [[2]]
# [1] 0.3721239 0.5728534
#
# [[3]]
# [1] 0.9082078 0.2016819 0.8983897
```

```
set.seed(1)
map(1:3, ~ runif(n = .x))
```

```
# [[1]]
# [1] 0.2655087
#
# [[2]]
# [1] 0.3721239 0.5728534
#
# [[3]]
# [1] 0.9082078 0.2016819 0.8983897
```

Filtrar listas

Mantém conforme condição

```
## Mantém apenas elementos da lista onde todos
## os valores sejam maiores do que zero
keep(x, .p = ~all(. > 0))
```

```
# [[1]]
# [1] 1 2 3 4 5
#
# [[2]]
# [1] 4 5 7
#
# [[3]]
# [1] 2 2 2 2 2
```

Descarta conforme condição

```
## Discarta todos os elementos da lista, menos
## aqueles que não possuem todos maiores do que zero
discard(x, .p = ~all(. > 0))
```

```
# [[1]]
# [1]  98  34 -10
```

De `?keep`:

```
.p: For ‘keep()’ and ‘discard()’, a predicate function. Only
those elements where ‘.p’ evaluates to ‘TRUE’ will be kept or
discarded.
```

Execução condicional

Aplicação condicional

```
# Função predicativa: retorna TRUE ou FALSE.
is_ok <- function(x)  length(x) > 3
map_if(x, .p = is_ok, .f = sum)
```

```
# [[1]]
# [1] 15
#
# [[2]]
# [1] 4 5 7
#
# [[3]]
# [1]  98  34 -10
#
# [[4]]
# [1] 10
```

Aplica em posição/elementos indicados

```
map_at(x, .at = c(2, 4), .f = sum)
```

```
# [[1]]
# [1] 1 2 3 4 5
#
# [[2]]
# [1] 16
#
# [[3]]
# [1]  98  34 -10
#
# [[4]]
# [1] 10
```

Aplanação de uma lista

```
map_dbl(x, sum)
```

```
# [1]  15  16 122  10
```

```
map(x, sum) %>%   # retorna uma lista
    flatten_dbl() # aplana
```

```
# [1]  15  16 122  10
```

Listas em paralelo

Duas listas em paralelo

- Está assumindo que as listas tem mesmo comprimento.
- Que a operação nos pares de elementos é válida.

```
y <- list(3, 5, 0, 1)
map2(x, y, function(x, y) x * y)
```

```
# [[1]]
# [1]  3  6  9 12 15
#
# [[2]]
# [1] 20 25 35
#
# [[3]]
# [1] 0 0 0
#
# [[4]]
# [1] 2 2 2 2 2
```

Várias listas aninhadas

```
z <- list(-1, 1, -1, 1)
pmap(list(x, y, z),
     .f = function(x, y, z) x * y * z)
```

```
# [[1]]
# [1]  -3  -6  -9 -12 -15
#
# [[2]]
# [1] 20 25 35
#
# [[3]]
# [1] 0 0 0
#
# [[4]]
# [1] 2 2 2 2 2
```

Aplicação de funções

Invocar funções

- Permite invocar uma função de forma não tradicional.
- Chamar várias funções sobre o mesmo junto de argumentos.

```
invoke(sample, x = 1:5, size = 2)
```

```
# [1] 3 2
```

```
invoke(runif, n = 3)
```

```
# [1] 0.06178627 0.20597457 0.17655675
```

Invocar com lista de parâmetros/funções

```
invoke_map(runif, list(n = 2, n = 4))
```

```
# [[1]]
# [1] 0.6870228 0.3841037
#
# [[2]]
# [1] 0.7698414 0.4976992 0.7176185 0.9919061
```

```
invoke_map(c("runif", "rnorm"), n = 3)
```

```
# [[1]]
# [1] 0.3800352 0.7774452 0.9347052
#
# [[2]]
# [1] -0.7990092 -1.1476570 -0.2894616
```

Tratamento de excessões

Cuidar de excessões

- Permite tratar excessões **sem interromper execução**.
- E também capturar mensagens de erro, alerta e notificação.

```
# Função que pode provocar `Error`.
my_fun <- function(x) {
    if (all(x > 0)) {
        sum(log(x))
    } else {
        stop("x must be > 0")
    }
}

## Sem tratar, o comando para no erro
map_dbl(x, my_fun)
```

```
# Error in .f(.x[[i]], ...): x must be > 0
```

```
## Com tratamento, retorna o que por possível
map_dbl(x,
        .f = possibly(my_fun,
                      otherwise = NA))
```

```
# [1] 4.787492 4.941642       NA 3.465736
```

Funções envelope

- São 3: `possibly()`, `safely()` e `quietly()`.

```
# Captura as mensagens de erro e resultados.
u <- map(x[c(3:4)], safely(my_fun))
str(u)
```

```
# List of 2
#  $ :List of 2
#   ..$ result: NULL
#   ..$ error :List of 2
#   .. ..$ message: chr "x must be > 0"
#   .. ..$ call   : language .f(...)
#   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "con
#  $ :List of 2
#   ..$ result: num 3.47
#   ..$ error : NULL
```

```
# Captura avisos, notificações e resultados.
u <- map(x[c(3:4)], quietly(sum))
## str(u)
```

Acumular e reduzir

Uma função aplicada recursivamente

```
# Para ser didático.
juros <- function(valor, taxa = 0.025) {
    valor * (1 + taxa)
}

# Rendimento por 4 meses.
juros(10) %>% juros() %>% juros() %>% juros()
```

```
# [1] 11.03813
```

```
# A conta de forma mais simples.
10 * (1 + 0.025)^(1:4)
```

```
# [1] 10.25000 10.50625 10.76891 11.03813
```

Usando as funções do `purrr`

```
# O outuput retorna como input.
reduce(rep(0.025, 4), juros, .init = 10)
```

```
# [1] 11.03813
```

```
# Essa mantém os resultados intermediários.
accumulate(rep(0.025, 4), juros, .init = 10)
```

```
# [1] 10.00000 10.25000 10.50625 10.76891 11.03813
```

Considerações finais

- Programação funcional é um recurso extremamente importante!
- Facilita aplicar funções de forma serial.
- Reduz a quantidade de código escrito e deixa a manutenção mais fácil.
- No R básico, a programação funcional é com a família `*apply` e amigos.

```
apropos("^\\w*apply") %>%
    append(c("replicate", "ave",
             "aggregate", "Reduce",
             "Filter"))
```

```
#  [1] "apply"      "dendrapply" "eapply"     "kernapply"
#  [5] "lapply"     "mapply"     "rapply"     "sapply"
#  [9] "tapply"     "vapply"     "replicate"  "ave"
# [13] "aggregate"  "Reduce"     "Filter"
```

- O pacote `purrr` fornece um framework consistente de funções para programação funcional.
- O domínio destes recursos vai permitir trabalhar em série com dados tabulares, listas, vetores, etc.
- É um recursos extremamente útil para modelagem e geração de gráficos.