Programação funcional com purrr

Prof. Walmes Zeviani

walmes@ufpr.br

Laboratório de Estatística e Geoinformação
Departamento de Estatística
Universidade Federal do Paraná

Um overview do purrr

Motivação

- Programação funcional é quando uma função chama uma outra função para ser aplicada repetidamente percorrendo elementos de um objeto.
- O recurso é útil para serializar tarefas.
- Exemplos:
  - Análise de experimento em vários locais.
  - Análise de todas as respostas de um experimento.
  - Fazer o ajuste de regressão polinomial de grau 1 até 5.
  - Análise dos dados com diferentes transformações de variáveis.
  - Simulação computacional com diferentes delineamentos.
  - Importação de todos os datasets de um diretório.
  - Tratamento de todas as imagens de uma diretório.

Programação funcional com purrr

- 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.

A ficha técnica

purrr: Functional Programming Tools

A complete and consistent functional programming toolkit for R.

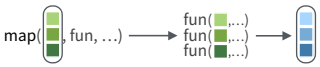
Version:	0.3.2
Depends:	R (≥ 3.1)
Imports:	magrittr (≥ 1.5), rlang (≥ 0.3.1)
Suggests:	covr, crayon, dplyr (≥ 0.7.8), knitr, rmarkdown, testthat, tibble, tidyselect
Published:	2019-03-15
Author:	Lionel Henry [aut, cre], Hadley Wickham [aut], RStudio [cph, fnd]
Maintainer:	Lionel Henry <lionel at rstudio.com>
BugReports:	https://github.com/tidyverse/purrr/issues
License:	GPL-3 | file LICENSE
URL:	http://purrr.tidyverse.org, https://github.com/tidyverse/purrr
NeedsCompilation:	yes
Materials:	README NEWS
CRAN checks:	purrr results

Figura 1. Ficha técnica do `purrr`.

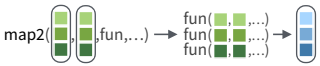
Apply functions with purrr : : **CHEAT SHEET**

Apply Functions

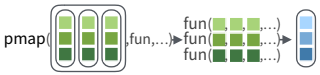
Map functions apply a function iteratively to each element of a list or vector.



map(.x, .f, …) Apply a function to each element of a list or vector. *map(x, is.logical)*



map2(.x, .y, .f, …) Apply a function to pairs of elements from two lists, vectors. *map2(x, y, sum)*



pmap(.l, .f, …) Apply a function to groups of elements from list of lists, vectors. *pmap(list(x, y, z), sum, na.rm = TRUE)*

invoke_map(.f, .x = list(NULL), …, .env=NULL) Run each function in a list. Also **invoke**. *l <- list(var, sd); invoke_map(l, x = 1:9)*

lmap(.x, .f, ...) Apply function to each list-element of a list or vector.
imap(.x, .f, ...) Apply .f to each element of a list or vector and its index.

OUTPUT

map(), map2(), pmap(), imap and **invoke_map** each return a list. Use a suffixed version to return the results as a specific type of flat vector, e.g. **map2_chr**, **pmap_lgl**, etc.

Use **walk**, **walk2**, and **pwalk** to trigger side effects. Each return its input invisibly.

function	returns
map	list
map_chr	character vector
map_dbl	double (numeric) vector
map_dfc	data frame (column bind)
map_dfr	data frame (row bind)
map_int	integer vector
map_lgl	logical vector
walk	triggers side effects, returns the input invisibly

SHORTCUTS - within a purrr function:

"name" becomes **function(x) x[["name"]]**, e.g. *map(l, "a")* extracts *a* from each element of *l*

~ .x becomes **function(x) x**, e.g. *map(l, ~ 2 +.x)* becomes *map(l, function(x) 2 + x)*

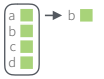
~ .x .y becomes **function(.x, .y) .x .y**, e.g. *map2(l, p, ~ .x +.y)* becomes *map2(l, p, function(l, p) l + p)*

~ ..1 ..2 etc becomes **function(..1, ..2,** etc**) ..1 ..2** etc, e.g. *pmap(list(a, b, c), ~ ..3 + ..1 - ..2)* becomes *pmap(list(a, b, c), function(a, b, c) c + a - b)*

RStudio

Work with Lists

FILTER LISTS

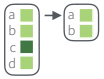


pluck(.x, ..., .default=NULL) Select an element by name or index, *pluck(x,"b")* ,or its attribute with **attr_getter**. *pluck(x,"b",attr_getter("n"))*

keep(.x, .p, …) Select elements that pass a logical test. *keep(x, is.na)*

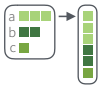
discard(.x, .p, …) Select elements that do not pass a logical test. *discard(x, is.na)*

compact(.x, .p = identity) Drop empty elements. *compact(x)*

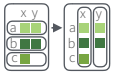


head_while(.x, .p, …) Return head elements until one does not pass. Also **tail_while**. *head_while(x, is.character)*

RESHAPE LISTS



flatten(.x) Remove a level of indexes from a list. Also **flatten_chr**, **flatten_dbl**, **flatten_dfc**, **flatten_dfr**, **flatten_int**, **flatten_lgl**. *flatten(x)*



transpose(.l, .names = NULL) Transposes the index order in a multi-level list. *transpose(x)*

SUMMARISE LISTS

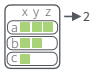
every(.x, .p, …) Do all elements pass a test? *every(x, is.character)*

some(.x, .p, …) Do some elements pass a test? *some(x, is.character)*

has_element(.x, .y) Does a list contain an element? *has_element(x, "foo")*

detect(.x, .f, ..., .right=FALSE, .p) Find first element to pass. *detect(x, is.character)*

detect_index(.x, .f, ..., .right = FALSE, .p) Find index of first element to pass. *detect_index(x, is.character)*



vec_depth(x) Return depth (number of levels of indexes). *vec_depth(x)*

JOIN (TO) LISTS

append(x, values, after = length(x)) Add to end of list. *append(x, list(d = 1))*

prepend(x, values, before = 1) Add to start of list. *prepend(x, list(d = 1))*

splice(…) Combine objects into a list, storing S3 objects as sub-lists. *splice(x, y, "foo")*

TRANSFORM LISTS

modify(.x, .f, ...) Apply function to each element. Also **map**, **map_chr**, **map_dbl**, **map_dfc**, **map_dfr**, **map_int**, **map_lgl**. *modify(x, ~.+2)*

modify_at(.x, .at, .f, ...) Apply function to elements by name or index. Also **map_at**. *modify_at(x, "b", ~.+2)*

modify_if(.x, .p, .f, ...) Apply function to elements that pass a test. Also **map_if**. *modify_if(x, is.numeric,~.+2)*

modify_depth(.x,.depth,.f,...) Apply function to each element at a given level of a list. *modify_depth(x, 1, ~.+2)*

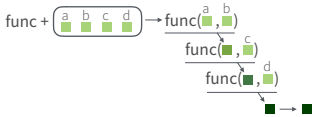
WORK WITH LISTS

array_tree(array, margin = NULL) Turn array into list. Also **array_branch**. *array_tree(x, margin = 3)*

cross2(.x, .y, .filter = NULL) All combinations of .x and .y. Also **cross**, **cross3**, **cross_df**. *cross2(1:3, 4:6)*

set_names(x, nm = x) Set the names of a vector/list directly or with a function. *set_names(x, c("p", "q", "r"))* *set_names(x, tolower)*

Reduce Lists



reduce(.x, .f, ..., .init) Apply function recursively to each element of a list or vector. Also **reduce_right**, **reduce2**, **reduce2_right**. *reduce(x, sum)*

accumulate(.x, .f, ..., .init) Reduce, but also return intermediate results. Also **accumulate_right**. *accumulate(x, sum)*

Modify function behavior

compose() Compose multiple functions.

lift() Change the type of input a function takes. Also **lift_dl**, **lift_dv**, **lift_ld**, **lift_lv**, **lift_vd**, **lift_vl**.

rerun() Rerun expression n times.

negate() Negate a predicate function (a pipe friendly !)

partial() Create a version of a function that has some args preset to values.

safely() Modify func to return list of results and errors.

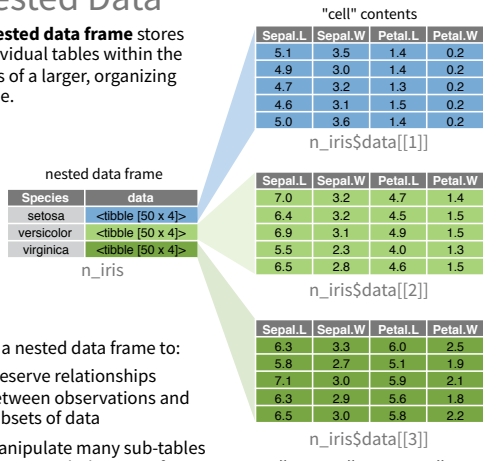
quietly() Modify function to return list of results, output, messages, warnings.

possibly() Modify function to return default value whenever an error occurs (instead of error).

Figura 2. Cartão de referência dos recursos do purrr.

Nested Data

A **nested data frame** stores individual tables within the cells of a larger, organizing table.

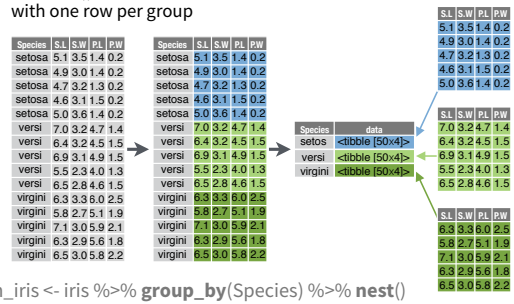


Use a nested data frame to:

- preserve relationships between observations and subsets of data
- manipulate many sub-tables at once with the **purrr** functions **map()**, **map2()**, or **pmap()**.

Use a two step process to create a nested data frame:

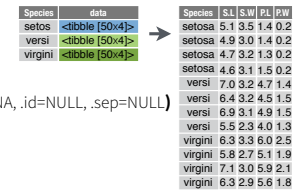
1. Group the data frame into groups with **dplyr::group_by()**
2. Use **nest()** to create a nested data frame with one row per group



n_iris <- iris %>% **group_by**(Species) %>% **nest**()

tidyr::**nest(**data, ..., .key = data**)**
For grouped data, moves groups into cells as data frames.

Unnest a nested data frame with **unnest()**:



n_iris %>% **unnest**()

tidyr::**unnest(**data, ..., .drop = NA, .id=NULL, .sep=NULL**)**
Unnests a nested data frame.

RStudio

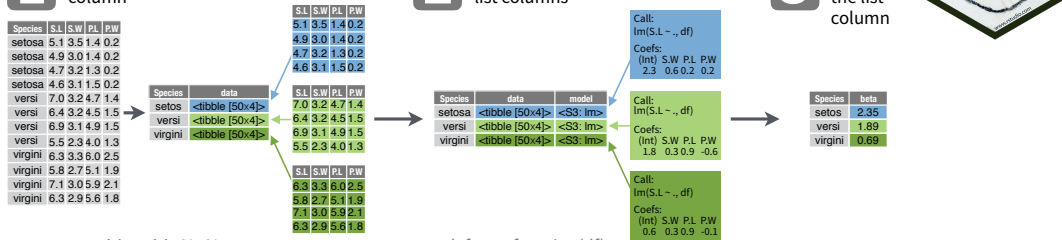
List Column Workflow

Nested data frames use a **list column**, a list that is stored as a column vector of a data frame. A typical **workflow** for list columns:

1 **Make** a list column

2 **Work with** list columns

3 **Simplify** the list column



```
n_iris <- iris %>%
  group_by(Species) %>%
  nest()
```

```
mod_fun <- function(df)
  lm(Sepal.Length ~ ., data = df)

m_iris <- n_iris %>%
  mutate(model = map(data, mod_fun))
```

```
b_fun <- function(mod)
  coefficients(mod)[[1]]

m_iris %>% transmute(Species,
  beta = map_dbl(model, b_fun))
```

1. MAKE A LIST COLUMN - You can create list columns with functions in the **tibble** and **dplyr** packages, as well as **tidyr**'s nest()

tibble::**tribble(**...**)**
Makes list column when needed

```
tribble( ~max, ~seq,
  3,   1:3,
  4,   1:4,
  5,   1:5)
```

max	seq
3	<int [3]>
4	<int [4]>
5	<int [5]>

tibble::**tibble(**...**)**
Saves list input as list columns

```
tibble(max = c(3, 4, 5), seq = list(1:3, 1:4, 1:5))
```

tibble::**enframe(**x, name="name", value="value"**)**
Converts multi-level list to tibble with list cols

```
enframe(list('3'=1:3, '4'=1:4, '5'=1:5), 'max', 'seq')
```

dplyr::**mutate(**.data, ...**)** Also **transmute()**
Returns list col when result returns list.

```
mtcars %>% mutate(seq = map(cyl, seq))
```

dplyr::**summarise(**.data, ...**)**
Returns list col when result is wrapped with **list()**

```
mtcars %>%  group_by(cyl) %>%
  summarise(q = list(quantile(mpg)))
```

2. WORK WITH LIST COLUMNS - Use the purrr functions **map()**, **map2()**, and **pmap()** to apply a function that returns a result element-wise to the cells of a list column. **walk()**, **walk2()**, and **pwalk()** work the same way, but return a side effect.

purrr::**map(**.x, .f, ...**)**
Apply .f element-wise to .x as .f(.x)

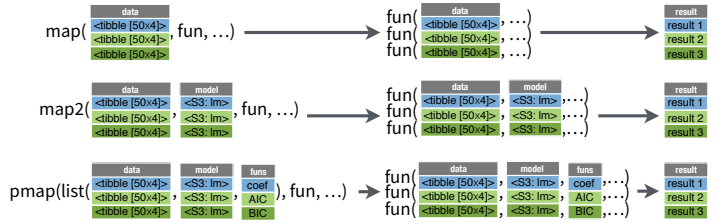
```
n_iris %>% mutate(n = map(data, dim))
```

purrr::**map2(**.x, .y, .f, ...**)**
Apply .f element-wise to .x and .y as .f(.x, .y)

```
m_iris %>% mutate(n = map2(data, model, list))
```

purrr::**pmap(**.l, .f, ...**)**
Apply .f element-wise to vectors saved in .l

```
m_iris %>%
  mutate(n = pmap(list(data, model, data), list))
```



3. SIMPLIFY THE LIST COLUMN (into a regular column)

Use the purrr functions **map_lgl()**, **map_int()**, **map_dbl()**, **map_chr()**, as well as tidyr's **unnest()** to reduce a list column into a regular column.

purrr::**map_lgl(**.x, .f, ...**)**
Apply .f element-wise to .x, return a logical vector

```
n_iris %>% transmute(n = map_lgl(data, is.matrix))
```

purrr::**map_int(**.x, .f, ...**)**
Apply .f element-wise to .x, return an integer vector

```
n_iris %>% transmute(n = map_int(data, nrow))
```

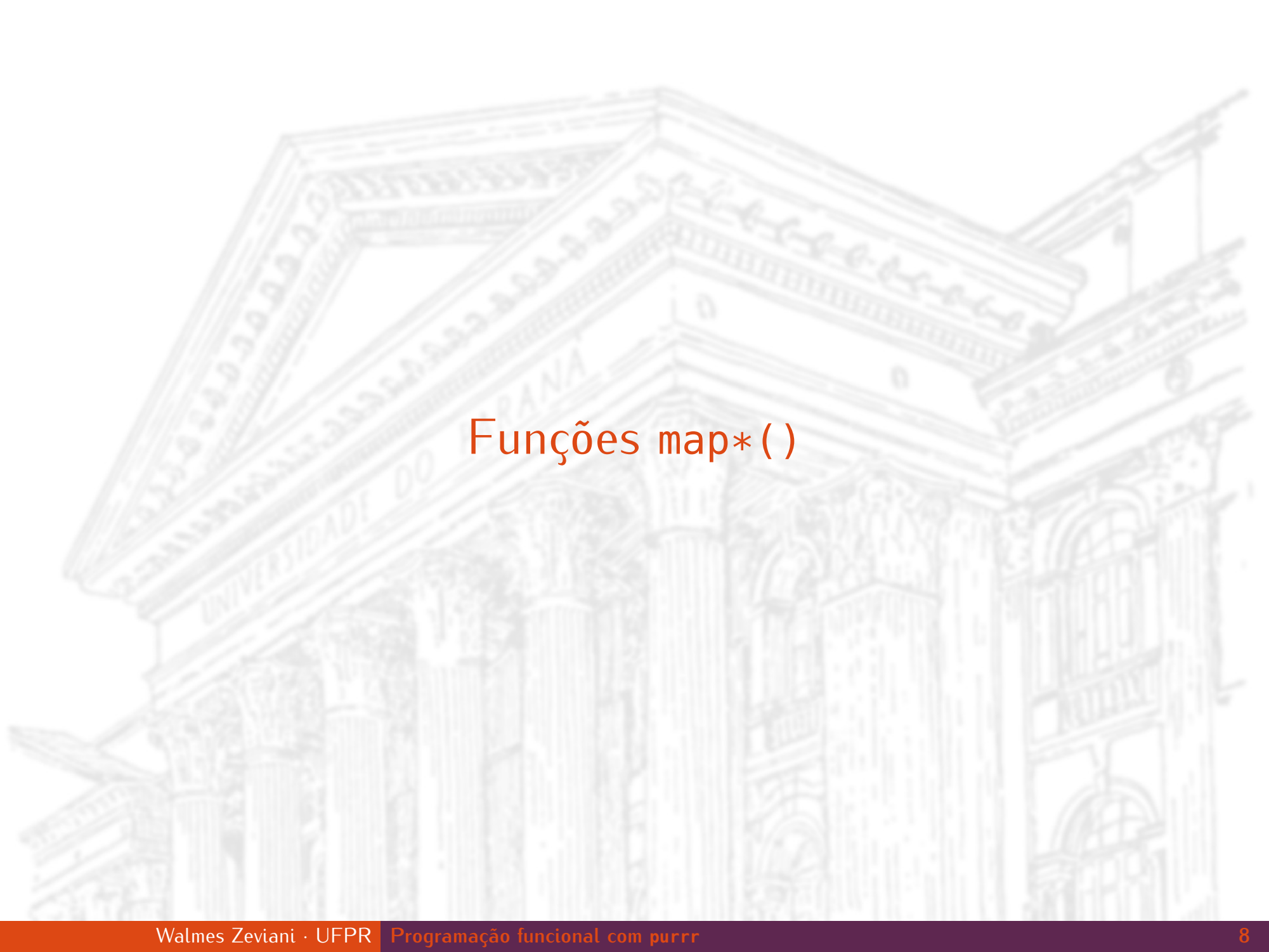
purrr::**map_dbl(**.x, .f, ...**)**
Apply .f element-wise to .x, return a double vector

```
n_iris %>% transmute(n = map_dbl(data, nrow))
```

purrr::**map_chr(**.x, .f, ...**)**
Apply .f element-wise to .x, return a character vector

```
n_iris %>% transmute(n = map_chr(data, nrow))
```


Figura 3. Cartão de referência dos recursos do purrr.



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`.

```
apropos("^map_")
```

```
##  [1] "map_at"   "map_call" "map_chr"  "map_data" "map_dbl"  "map_df"
##  [7] "map_dfc"  "map_dfr"  "map_if"   "map_int"  "map_lgl"
```

```
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"
```

Aplicação condicional

```r
is_ok <- function(x)  length(x) > 3
map_if(x, is_ok, sum)
```

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

```r
map_at(x, c(2, 4), sum)
```

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

Aplanação de uma lista

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

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

```
map(x, sum) %>%
    flatten_dbl()
```

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

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), 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
```

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

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

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

1

```
## [1] 0.5690078 0.1955735 0.9976936
```

Invocar várias funções

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

```
## [[1]]
## [1] 0.5429372 0.5122683
##
## [[2]]
## [1] 0.2400834 0.5127896 0.7234898 0.2562734
```

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

```
## [[1]]
## [1] 0.9021426 0.6439952 0.2799856
##
## [[2]]
## [1] -0.3652052  0.1814090  0.8867628
```

Cuidar de excessões

- Permite tratar excessões sem interromper execução.
- Capturar mensagens de erro, alerta e notificação.

```
my_fun <- function(x) if (all(x > 0)) sum(x) else stop("x must be > 0")

map(x, possibly(my_fun, otherwise = NA)) %>%
    flatten_dbl()
```

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

```
u <- map(x, safely(my_fun))
glimpse(u)
```

```
## List of 4
##  $ :List of 2
##   ..$ result: int 15
##   ..$ error : NULL
##  $ :List of 2
##   ..$ result: num 16
##   ..$ error : NULL
##  $ :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" "condition"
```

Cuidar de excessões

```
u <- map(x, quietly(sum))
glimpse(u)
```

```
## List of 4
##  $ :List of 4
##   ..$ result  : int 15
##   ..$ output  : chr ""
##   ..$ warnings: chr(0)
##   ..$ messages: chr(0)
##  $ :List of 4
##   ..$ result  : num 16
##   ..$ output  : chr ""
##   ..$ warnings: chr(0)
##   ..$ messages: chr(0)
##  $ :List of 4
##   ..$ result  : num 122
##   ..$ output  : chr ""
##   ..$ warnings: chr(0)
##   ..$ messages: chr(0)
##  $ :List of 4
##   ..$ result  : num 10
##   ..$ output  : chr ""
##   ..$ warnings: chr(0)
##   ..$ messages: chr(0)
```

Acumular e reduzir

```r
# Para tentar ser didático.
juros <- function(valor, taxa = 0.025) valor * (1 + taxa)
juros(10) %>% juros() %>% juros() %>% juros()
```

```
## [1] 11.03813
```

```r
reduce(rep(0.025, 4), juros, .init = 10)
```

```
## [1] 11.03813
```

```r
accumulate(rep(0.025, 4), juros, .init = 10)
```

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

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

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

Aninhar

```
# iris %>% group_by(Species) %>% nest()
u <- iris %>% as_tibble() %>% nest(-Species)

u %>%
    mutate(correlation = map(data, cor),
           model = map(data, ~lm(Sepal.Length ~ ., data = .))) %>%
    mutate(AIC = map(model, AIC)) %>%
    unnest(AIC)
```

```
## # A tibble: 3 x 5
##   Species    data              correlation   model      AIC
##   <fct>      <list>            <list>        <list>   <dbl>
## 1 setosa     <tibble [50 x 4]> <dbl [4 x 4]> <S3: lm>  3.81
## 2 versicolor <tibble [50 x 4]> <dbl [4 x 4]> <S3: lm> 38.3
## 3 virginica  <tibble [50 x 4]> <dbl [4 x 4]> <S3: lm> 33.2
```

Exercícios para usar o purrr

Análise de experimentos em vários locais

1. Importe o conjunto de dados http://leg.ufpr.br/~walmes/data/soja_grupoexperimentos.txt.
2. Faça uma análise exploratória para a variável redimento.
3. Crie um `tibble` aninhando a tabela pelo local.
4. Crie uma função que retorne o quadro de análise de variância do modelo apropriado para a análise do experimento em cada local.
5. Obtenha o quadro de análise de variância de cada local.
6. Crie uma função que extraia os graus de liberdade, o quadrado médio de tratamentos e a signficância do teste F.
7. Monte uma tabela que resuma os quadros de anova.
8. Agradeça por o R existir.