Estatística Computacional

Universidade Federal da Bahia

Gilberto Pereira Sassi

Tópico 3

Pacotes que iremos usar na semana 4

```
library(readxl)
library(readODS)
library(openxlsx)
library(ggthemes)
library(lvplot)
library(glue)
library(codetools)
library(tidyverse)
```

Linguagem **R** e **tidyverse**

um pouco de programação

`pivot_longer` e `pivot_wider`: pacote `tidyr`

- **`tidyr`**: transformar e modificar uma `tibble` para transformar em um `tidy data`.

`tidy data`

1. Cada coluna é uma variável
2. Cada linha é uma observação
3. Cada célula é um único valor

Para mais detalhes sobre `tidy data`, vide `tidy data`

`pivot_longer` e `pivot_wider`

- Transforma as colunas `names:values` em várias colunas
- Transforma várias colunas em colunas `names:values`

Para mais detalhes, vide a documentação oficial do pacote `tidyr`

`pivot_longer` e `pivot_wider`: pacote `tidyr`

- **`religiao_renda.csv`**: conjunto de dados sobre religião e renda. Disponível em dados sobre religião e renda.

`pivot_longer`

```
dados <- read_csv2('data/raw/religiao_renda.csv')

tab <- dados |>
  pivot_longer(-religion, names_to = 'Renda', values_to = "Frequência")
head(tab, n = 4)
```

```
## # A tibble: 4 × 3
##   religion Renda   Frequência
##   <chr>    <chr>        <dbl>
## 1 Agnostic <$10k           27
## 2 Agnostic $10-20k         34
## 3 Agnostic $20-30k         60
## 4 Agnostic $30-40k         81
```

```
write_xlsx(tab, 'data/processed/exemplo_pivot_longer.xlsx')
```

`pivot_longer` e `pivot_wider`: pacote `tidyr`

- **`peixe_descoberto`**: conjunto de dados sobre a visualização de um peixe em uma estação de monitoramento. Disponível em dados sobre peixes descobertos.

`pivot_longer`

```
dados <- read_xlsx('data/raw/peixe_descoberto.xlsx')

tab <- dados |>
  pivot_wider(names_from = 'fish', values_from = 'seen', names_glue = "peixe_id_{fish}")
head(tab, n = 2)
```

```
## # A tibble: 2 × 20
##   station peixe_id_4842 peixe_id_4843 peixe_id_4844 peixe_id_4845 peixe_id_4847 peixe_id_4848
##   <chr>           <dbl>         <dbl>         <dbl>         <dbl>         <dbl>         <dbl>
## 1 Release             1             1             1             1             1             1
## 2 I80_1               1             1             1             1             1             1
## # … with 13 more variables: peixe_id_4849 <dbl>, peixe_id_4850 <dbl>, peixe_id_4851 <dbl>,
## #   peixe_id_4854 <dbl>, peixe_id_4855 <dbl>, peixe_id_4857 <dbl>, peixe_id_4858 <dbl>,
## #   peixe_id_4859 <dbl>, peixe_id_4861 <dbl>, peixe_id_4862 <dbl>, peixe_id_4863 <dbl>,
## #   peixe_id_4864 <dbl>, peixe_id_4865 <dbl>
```

```
write_xlsx(tab, 'data/processed/exemplo_pivot_wider.xlsx')
```

Função

- Útil para evitar repetição de código
- Usamos para otimização de funções em estatística (por exemplo, achar ponto de máximo da função de verossimilhança)

Sintaxe básica

```
<nome da função> <- function(<parâmetros>){
  corpo da função
  último valor da última expressão avaliada é retornada
}
```

Funções úteis

- `body()`: retorna o corpo da função
- `formals()`: retorna uma lista com os parâmetros e os valores padronizados
- `environment()`: retorna a estrutura de dados associada com a criação da função

Função

Regra de escopo

- **Regra de escopo:** regras para de como avaliar um objeto em R
  - primeiro o R busca o objeto dentro do *escopo da função* (ou seja, verifica se o objeto foi criado dentro do corpo da função)
  - se o objeto não existe dentro do *escopo da função*
  - **importante:** evite o uso de funções que são definidas globalmente dentro de funções!

Exemplo

Variável definida localmente dentro de uma função com o mesmo nome de um objeto global.

```
x <- 10
f <- function() {
  x <- 1
  x
}
f()
```

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

Função

Exemplo
Função criada dentro de outra função.

```
f <- function() {
  y <- 2
  function(x) c('x' = x, 'y' = y)
}
y <- 1
h <- f()
h(3)
```

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

Função

Exemplo
Podemos descobrir se uma função depende definidos globalmente com a função `findGlobals` definida em `codetools`.

```
x <- 1
f <- function(y) x + y

# checando se uma função depende de objetos definidos globalmente
codetools::findGlobals(f)
```

```
## [1] "+" "x"
```

Função

shorthand (a partir da versão 4.1)

```
z <- \(x) {
  <body function>
}
```

ou

```
z <- \(x) <expression>
```

Função

Valor padrão e organização dos parâmetros

```
z <- function(x, translado = 0) {
  y <- x + translado
  round((y - mean(y)) / sd(y), 2)
}
z(1:5)
```

```
## [1] -1.26 -0.63  0.00  0.63  1.26
```

```
z(1:5, 0)
```

```
## [1] -1.26 -0.63  0.00  0.63  1.26
```

```r
z(1:5, translado = 0)
```

```
## [1] -1.26 -0.63  0.00  0.63  1.26
```

```r
z(translado = 0, 1:5)
```

```
## [1] -1.26 -0.63  0.00  0.63  1.26
```

Função

Parâmetro ellipsis

- `...`: parâmetros adicionais (pode ser um ou mais parâmetros)
- Pontos transformar `...` in lista para acessar os valores

```
z <- function(x, translado = 0, ...) {
  y <- x + translado
  argumentos <- list(...)
  print(argumentos)
  m <- mean(x, ...)
  s <- sd(x, ...)
  round((x - m) / s, 2)
}
x <- c(NA, 1:5, NA)
z(x, translado = 0, na.rm = TRUE)
```

```
## $na.rm
## [1] TRUE
```

```
## [1]    NA -1.26 -0.63  0.00  0.63  1.26    NA
```

Função

Função anônima

- **Função anônima:** objeto função sem qualquer identificação (usado principalmente com as funções `*apply` e `map*`)

```
library(numDeriv)
grad(\(x) sum(x^2), 1:4)
```

```
## [1] 2 4 6 8
```

```
hessian(\(x) sum(x^2), 1:6)
```

```
##              [,1]          [,2]          [,3]          [,4]          [,5]          [,6]
## [1,]  2.000000e+00  3.285113e-11  4.380191e-11  4.103830e-16  1.912785e-17 -1.583754e-14
## [2,]  3.285113e-11  2.000000e+00  1.005240e-11 -8.886179e-12  3.864731e-17 -5.476024e-12
## [3,]  4.380191e-11  1.005240e-11  2.000000e+00 -4.488308e-13  1.221506e-16 -5.168776e-15
## [4,]  4.103830e-16 -8.886179e-12 -4.488308e-13  2.000000e+00  7.729463e-17 -2.737964e-12
## [5,]  1.912785e-17  3.864731e-17  1.221506e-16  7.729463e-17  2.000000e+00 -3.636127e-16
## [6,] -1.583754e-14 -5.476024e-12 -5.168776e-15 -2.737964e-12 -3.636127e-16  2.000000e+00
```

Operadores lógicos

- **Valores lógicos:** `TRUE(T)` ou `FALSE(F)`
- *Negação*: `!`

```
!F
```

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

- *E*: `&`

```
F & T
```

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

- *Igualdade*: `==`

```
1 == 2
```

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

- *Ou*: |

```
F | T
```

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

- *Ou exclusivo*: `xor`

```
xor(F, T)
```

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

```
xor(T, T)
```

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

- `any()` retorna `TRUE` se algum elemento de um vetor lógico

```
any(c(T, T, T))
```

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

```
any(c(F, T, T))
```

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

```
any(c(F, F, F))
```

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

- `all()` retorna `TRUE` se todos os elementos de um vetor lógico

```
all(c(T, T, T))
```

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

```
all(c(F, T, T))
```

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

```
all(c(F, F, F))
```

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

- **near()** função do pacote `dplyr` para comparar tipo de dados `double`

```
sqrt(2)^2 == 2
```

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

```
near(sqrt(2)^2, 2)
```

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

- **all.equal()** retorna `TRUE` se todos os valores de dois vetores são iguais

```
all.equal(c(1,1), c(1,1) + 1e-4, tol = 1e-3)
```

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

```r
all.equal(c(1,1), c(1,1) + 1e-3, tol = 1e-3)
```

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

```r
all.equal(c(1,1), c(1,1) + 1e-2, tol = 1e-3)
```

```
## [1] "Mean relative difference: 0.01"
```

Função

Mensagens aos usuários

- `stop()`: para a execução e exibe uma mensagem de erro ao usuário
- `warning()`: exibe mensagem de atenção ao usuário
- `message()`: exibe mensagem úteis ao usuário

```
fracao <- function(x) {
  message('Olá, eu criei essa função!!!')

  # problema com divisão por zero
  if (near(x, 0)) {
    stop("Proibido divisão por zero!")
  }

  # problema com valores muito pequenos
  if (abs(x) < 1e-4){
    warning("Valor muito pequeno.")
  }

  1 / x
}
```

```
fracao(0.5)
```

```
## Olá, eu criei essa função!!!
```

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

```
fracao(0)
```

```
## Olá, eu criei essa função!!!
```

```
## Error in fracao(0): Proibido divisão por zero!
```

```
fracao(1e-5)
```

```
## Olá, eu criei essa função!!!
```

```
## Warning in fracao(1e-05): Valor muito pequeno.
```

```
## [1] 1e+05
```

Lidando com mensagens de erro

- `possibly()` se acontecer retorno uma valor padrão
- `quietly()` Se acontecer um erro, a execução para imediatamente. Quando funciona, retorna uma lista com os seguintes componentes `result`, `output`, `warnings` e `messages`.
- `safely()` retorna uma lista com os componentes `result` e `error`.

```
possibly_fracao <- possibly(fracao,
                            otherwise = NA,
                            quiet = TRUE)
possibly_fracao(0)
```

```
## [1] NA
```

```
possibly_fracao(1)
```

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

Lidando com mensagens de erro

```
quietly_fracao <- quietly(fracao)
quietly_fracao(0)
```

```
## Error in .f(...): Proibido divisão por zero!
```

```
quietly_fracao(1)
```

```
## $result
## [1] 1
##
## $output
## [1] ""
##
## $warnings
## character(0)
##
## $messages
## [1] "Olá, sou eu de novo!\n"
```

Lidando com mensagens de erro

```
safely_fracao <- safely(fracao, otherwise = NA, quiet = TRUE)
safely_fracao(0)
```

```
## Olá, sou eu de novo!
```

```
## $result
## [1] NA
##
## $error
## <simpleError in .f(...): Proibido divisão por zero!>
```

```
safely_fracao(1)
```

```
## Olá, sou eu de novo!
```

```
## $result
## [1] 1
##
## $error
## NULL
```

Controle de fluxo

- **Principais de fluxo:** `if-else`, `ifelse`, `for` e `while`
- `if (<expressão>) {}`: executa as expressões em `{}` se o valor em `<expressão>` é `TRUE`

```
if (TRUE) {
  print("Olá")
  print("Mundo")
}
```

```
## [1] "Olá"
## [1] "Mundo"
```

- `if-else`: Executa o corpo de `if` se a expressão em `if` é `TRUE`, caso contrário o corpo de `else`

```
x <- FALSE
if (x) {
  print("Aqui temos TRUE.")
} else {
  print("Aqui temos FALSE.")
}
```

```
## [1] "Aqui temos FALSE."
```

Controle de fluxo

`ifelse`

`ifelse(<expressão condicional>, <expressão 1>, <expressão 2>)`. `<expressão condicional>` retorna um vetor lógico. Versão vetorizada de `if-else`.

```
x <- c(T, T, F)
ifelse(x, 'Isto é Verdadeiro', 'Isto é Falso')
```

```
## [1] "Isto é Verdadeiro" "Isto é Verdadeiro" "Isto é Falso"
```

`for`

`for(k in <vetor ou lista>) {}` repete os comandos do corpo `for` para cada elemento de `<vetor ou lista>`

```
for(i in c(1, 4, 11)) {
  glue("Elemento: {i}.") |> print()
}
```

```
## Elemento: 1.
## Elemento: 4.
## Elemento: 11.
```

Controle de fluxo

- `while(<expressão>) {}` repete os comandos do corpo `while` enquanto `<expressão>` for `TRUE`

```
x <- 'continue'
while(x == 'continue') {
  print("Estamos aqui com 'continue'")
  x <- sample(c('continue', 'pare'), 1)
}
```

```
## [1] "Estamos aqui com 'continue'"
## [1] "Estamos aqui com 'continue'"
## [1] "Estamos aqui com 'continue'"
## [1] "Estamos aqui com 'continue'"
## [1] "Estamos aqui com 'continue'"
## [1] "Estamos aqui com 'continue'"
```

Controle de fluxo

- `break` sai do loop imediatamente

```
while(TRUE) {
  print("Estamos continuando.")
  x <- sample(c('continue', 'pare'), 1)
  if (x == 'pare') {
    print("Eu vou parar.")
    break
  }
}
```

```
## [1] "Estamos continuando."
## [1] "Estamos continuando."
## [1] "Estamos continuando."
## [1] "Estamos continuando."
## [1] "Estamos continuando."
## [1] "Estamos continuando."
## [1] "Eu vou parar."
```

Controle de fluxo

- `next` volta ao início no corpo do laço

```
for(k in 1:5) {
  if(near(k, 3)) {
    next
  }
  glue("k == {k}") |> print()
}
```

```
## k == 1
## k == 2
## k == 4
## k == 5
```

Controle de fluxo

- `repeat {}`: repete os comandos do corpor `repeat`. Precisamos usar `break` para sair do laço

```
repeat{
  x <- sample.int(3, 1)
  if(near(x, 2)) break
  print(x)
}
```

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

Programação funcional com `map*` e `*apply`

- `map*` do pacote `purrr` e as funções `*apply`(abreviação de laço)
  - laços podem ser lentos, e `map*` e `*apply` são geralmente mais rápidos
  - facilita a documentação do código
- `apply(x, MARGIN, FUN,...)` retorna um vetor ou matrix (se `FUN` retornar um vetor) e retorna uma lista (se `FUN` retornar uma lista). `apply` computa `FUN` em cada linha de x se `MARGIN == 1`, ou computa `FUN` em cada coluna de x se `MARGIN == 2`.
  - `x`: uma matriz ou data frame;
  - `MARGIN`: `1` para calcular `FUN` em cada linhas; `2` para calcular `FUN` em cada coluna
  - `...`: argumentos adicionais de `FUN`

Programação funcional com `map*` e `*apply`

```
dados <- read_xlsx('data/raw/iris.xlsx')

matriz <- dados |>
  dplyr::select(-Species) |>
  apply(2, \(x, ...) {
    c(`Média` = mean(x, ...), `Desvio padrão` = sd(x, ...))
  }, na.rm = T)
class(matriz)
```

```
## [1] "matrix" "array"
```

```
matriz
```

```
##               Sepal.Length Sepal.Width Petal.Length Petal.Width
## Média            5.8433333   3.0573333     3.758000   1.1993333
## Desvio padrão    0.8280661   0.4358663     1.765298   0.7622377
```

Programação funcional com `map*` e `*apply`

- `sapply(x, FUN,...)` retorna um vetor ou matriz. `sapply` computa `FUN` para cada coluna (ou elemento) de `x`.
  - `x`: vetor, lista, matriz ou data frame;
  - `FUN`: uma função;
  - `...`: argumentos adicionais para `FUN`;

```
dados <- read_xlsx('data/raw/iris.xlsx')

matriz <- dados |>
  dplyr::select(-Species) |>
  sapply(\(x, ...){
    c(`Média` = mean(x, ...), `Desvio padrão` = sd(x, ...))
  }, na.rm = T)
matriz
```

```
##              Sepal.Length Sepal.Width Petal.Length Petal.Width
## Média           5.8433333   3.0573333     3.758000   1.1993333
## Desvio padrão   0.8280661   0.4358663     1.765298   0.7622377
```

Programação funcional com `map*` e `*apply`

- `vapply(x, FUN, FUN.VALUE, ...)` similar a `sapply`, mas geralmente mais rápido. `vapply` computa `FUN` em coluna (ou elmento) de `x` e precisamos especificar o tipo de dados retornado.
  - `x`: vector, matriz, lista ou data frame;
  - `FUN`: uma função;
  - `FUN.VALUE`: vetor especificando os valores que são retornados por FUN;
  - `...`: argumentos adicionais para `FUN`;

```
dados <- read_xlsx('data/raw/iris.xlsx')

matriz <- dados |>
  dplyr::select(-Species) |>
  vapply(\(x, ...) {
    c(`Média` = mean(x, ...), `Desvio padrão` = sd(x, ...))
  }, FUN.VALUE = c(double(1), double(1)), na.rm = T)
matriz
```

```
##               Sepal.Length Sepal.Width Petal.Length Petal.Width
## Média            5.8433333   3.0573333     3.758000   1.1993333
## Desvio padrão    0.8280661   0.4358663     1.765298   0.7622377
```

Programação funcional com `map*` e `*apply`

- `lapply(x, FUN,...)` retorna uma lista. `lapply` computa `FUN` para cada coluna (ou elemento) de `x`.
  - `x`: vetor, lista, matriz ou data frame;
  - `FUN`: uma função;
  - `...`: argumentos adicionais para `FUN`;

```
dados <- read_xlsx('data/raw/iris.xlsx')

lista <- dados |>
  dplyr::select(Sepal.Length, Petal.Length) |>
  lapply(\(x, ...) {
    c(`Média` = mean(x, ...), `Desvio padrão` = sd(x, ...))
  }, na.rm = T)
lista
```

```
## $Sepal.Length
##         Média Desvio padrão
##     5.8433333     0.8280661
##
## $Petal.Length
##         Média Desvio padrão
##      3.758000      1.765298
```

Programação funcional com `map*` e `*apply`

- **Pacote `purrr`:** faz quase a mesma coisa que `*apply`, mas a ideia é simplificar, padronizar e documentar para programação funcional. Também permite o uso de funções.
- Na lista abaixo, `FUN` precisa retornar apenas um número e `x` é um vetor, matriz, lista ou data frame:
  - `map(x, FUN, ...)` retorna uma lista;
  - `map_df(x, FUN, ...)` retorna um data frame;
  - `map_dbl(x, FUN, ...)` retorna um vetor de `double`;
  - `map_chr(x, FUN, ...)` retorna um vetor de `character`;
  - `map_int(x, FUN, ...)` retorna um vetor de `integer`;
  - `map_lgl(x, FUN, ...)` retorna um vetor de `logical`;

```r
dados <- read_xlsx('data/raw/iris.xlsx')

dados |> dplyr::select(ends_with('Length')) |> map_dbl(~ sd(.x))
```

```
## Sepal.Length Petal.Length
##    0.8280661    1.7652982
```

```r
dados |> dplyr::select(ends_with('Length')) |> map(\(x) sd(x))
```

```
## $Sepal.Length
## [1] 0.8280661
##
## $Petal.Length
## [1] 1.765298
```

```r
dados <- read_xlsx('data/raw/iris.xlsx')

dados |> dplyr::select(ends_with('Length')) |> map_df(~ c(sd(.x), mean(.x), median(.x)))
```

```
## # A tibble: 3 × 2
##   Sepal.Length Petal.Length
##          <dbl>        <dbl>
## 1        0.828         1.77
## 2        5.84          3.76
## 3        5.8           4.35
```

Manipulação de conjunto de dados: `dplyr`

- **mutate** atualiza ou cria novas variáveis em um conjunto de dados.

```
dados <- read_xlsx('data/raw/iris.xlsx')

k <- ceiling(1 + log2(nrow(dados))) # regra de Sturge

novo_dados <- dados |>
  mutate(comprimento_sepala_faixa = cut(Sepal.Length, breaks = k, include.lowest = TRUE,
                                        right = FALSE, digits = 2))

glimpse(novo_dados)
```

```
## Rows: 150
## Columns: 6
## $ Sepal.Length             <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, 5.4, 4.8, 4.8, 4.3, 5…
## $ Sepal.Width              <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7, 3.4, 3.0, 3.0, 4…
## $ Petal.Length             <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5, 1.6, 1.4, 1.1, 1…
## $ Petal.Width              <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2, 0.2, 0.1, 0.1, 0…
## $ Species                  <chr> "setosa", "setosa", "setosa", "setosa", "setosa", "setosa", "setosa", "…
## $ comprimento_sepala_faixa <fct> "[5.1,5.5)", "[4.7,5.1)", "[4.7,5.1)", "[4.3,4.7)", "[4.7,5.1)", "[5.1,…
```

Manipulação de conjunto de dados: `dplyr`

- **arrange** ordenar o data frame por alguma coluna.

```
dados <- read_csv2('data/raw/mtcars.csv')

head(dados, n = 3)
```

```
## # A tibble: 3 × 11
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  21       6   160   110  3.9   2.62  16.5     0     1     4     4
## 2  21       6   160   110  3.9   2.88  17.0     0     1     4     4
## 3  22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
```

```
novo_dados <- dados |>
  arrange(desc(cyl)) # sem desc ordena em ordem crescente
head(novo_dados, n = 3)
```

```
## # A tibble: 3 × 11
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
## 2  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
## 3  16.4     8  276.   180  3.07  4.07  17.4     0     0     3     3
```

Manipulação de conjunto de dados: `dplyr`

- **filter** filtra as linhas de um data frame.

```
dados <- read_xlsx('data/raw/iris.xlsx')
nrow(dados)
```

```
## [1] 150
```

```
novo_dados <- dados |>
  filter(Species == 'vetosa')

glimpse(dados)
```

```
## Rows: 150
## Columns: 5
## $ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, 5.4, 4.8, 4.8, 4.3, 5.8, 5.7, 5.4…
## $ Sepal.Width  <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7, 3.4, 3.0, 3.0, 4.0, 4.4, 3.9…
## $ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5, 1.6, 1.4, 1.1, 1.2, 1.5, 1.3…
## $ Petal.Width  <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2, 0.2, 0.1, 0.1, 0.2, 0.4, 0.4…
## $ Species      <chr> "setosa", "setosa", "setosa", "setosa", "setosa", "setosa", "setosa", "setosa", "se…
```

Manipulação de conjunto de dados: `dplyr`

- **rename** atualiza os nomes das variáveis.

Atenção para sintaxe!!!

```
rename(<data-frame>, <novo nome> = <velho nome>)
# ou
<data-frame> |> rename(<novo nome> = <velho nome>)
```

```
dados <- read_xlsx('data/raw/iris.xlsx')
colnames(dados)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

```
novo_dados <- dados |>
  rename(comprimento_sepala = Sepal.Length, comprimento_petala = Petal.Length,
         largura_sepala = Sepal.Width, largura_petala = Petal.Width,
         especies = Species)
colnames(novo_dados)
```

```
## [1] "comprimento_sepala" "largura_sepala"     "comprimento_petala" "largura_petala"
## [5] "especies"
```

Manipulação de conjunto de dados: **dplyr**

- **rename** atualiza os nomes das variáveis.

Atenção para sintaxe!!!

```
mutate(<data-frame>, <nome da variavel> = recode(<nome da variavel>, <velho nome> = <novo nome>))
# ou
<data-frame> |> mutate(<nome da variavel> = recode(<nome da variavel>, <velho nome> = <novo nome>))
```

```
dados <- read_xlsx('data/raw/iris.xlsx')

novo_dados <- dados |>
  filter(Species %in% 'setosa') |>
  mutate(especies = recode(Species, "setosa" = "NovoNome"))

head(novo_dados, n = 3)
```

```
## # A tibble: 3 × 6
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species especies
##          <dbl>       <dbl>        <dbl>       <dbl> <chr>   <chr>
## 1          5.1         3.5          1.4         0.2 setosa  NovoNome
## 2          4.9         3            1.4         0.2 setosa  NovoNome
## 3          4.7         3.2          1.3         0.2 setosa  NovoNome
```