Transformação e Manipulação de Dados com a linguagem R

Eric Scopinho

Índice

Bem-Vindo

Este livro contém uma série de informações sobre transformação e manipulação de dados utilizando a **linguagem R**, mais especificamente o pacote **tidyverse**.

Estas são etapas muito importantes para quem trabalha com dados, como por exemplo uma área de negócio de uma organização, buscando tomar decisões com base em seus dados, ou até mesmo no ciclo de vida de um projeto de ciência de dados.

A comunidade R tem produzido ao longo dos anos uma série Folhas de Referências (*cheatsheets*) que fazem parte também da ferramenta de desenvolvimento RStudio.

Estas **Folhas de Referências**, são anotações de 2 páginas que visam resumir as principais informações sobre determinado tema ou pacote do R. São uma espécie de "cola" para nos lembrarmos dos comandos e informações mais relevantes.

Este livro se baseia nestas folhas de referências, que apesar de extremamente úteis, podem ser de difícil interpretação para usuários iniciantes.

Alem disso, outro motivador para este livro, foi a escassez de documentação sobre o tema em lingua Portuguesa, que apesar dos grandes esforços da comunidade brasileira e voluntários, ainda sofre com falta de acesso para quem não domina o idioma inglês.

> **Aviso**
>
> Para melhor utilizar o conteúdo deste livro, é importante que você já possua uma introdução à linguagem R e saiba carregar pacotes (packages) no R. Não iremos abortar comandos básicos da linguagem e nem mesmo instalação da ferramenta de desenvolvimento.
> Para mais informações acesse:
> https://education.rstudio.com/learn/beginner/.

Os temas abordados neste livro são:

- Importação de Dados: Trata sobre **importação de dados** tabulados (.csv, .tsv, .txt), planilhas do Excel e Google Sheets, através do pacote **tidyverse (readr, readxl and googlesheets4).** Ver folha de referência: data-import cheatsheet.

- Organização de Dados: Sobre **organização de dados** com o pacote **tidyr**. Mover colunas e linhas de forma a estruturar seus dados em tabelas organizadas. Ver folha de referência: tidyr cheatsheet.
- Transformação de dados : Apresenta a **transformação de dados** com o pacote **dplyr**. Aplicação de filtros, sumarização, criação de colunas calculadas e muitas outras funções de transformação. Ver folha de referência: data transformation cheatsheet
- Manipulação de strings: Fala sobre **manipulação de strings** (textos) com o pacote **stringr**. Apresenta também um bom conteúdo sobre **expressões regulares** (**regex**). Ver folha de referência: stringr cheatsheet
- Datas e horas: Sobre a formatação e cálculos de **variáveis** com **datas** e **horas** com o pacote **lubridate**. Ver folha da referência: lubridate cheatsheet.
- Visualização de Dados: Traz uma introdução à criação de gráficos através do pacote **ggplot2**. Ver folha da referência: data visualization cheatsheet.

Licença

1 Introdução

Neste livro teremos vários exemplos de transformação e manipulação de dados utilizando os pacotes da linguagem R. O principal pacote que iremos utilizar chamado **tidyverse**. Ele é uma espécie de "super pacote" para ciência de dados e contém outros pacotes que auxiliam nas atividades relativas à esta prática, como importação, transformação, manipulação, modelagem e visualização de dados.

Para saber mais sobre este pacote, acesse:

https://cran.r-project.org/package=tidyverse

Para os exemplos, iremos carregar inicialmente os seguintes pacotes:

- **tidyverse e gt**

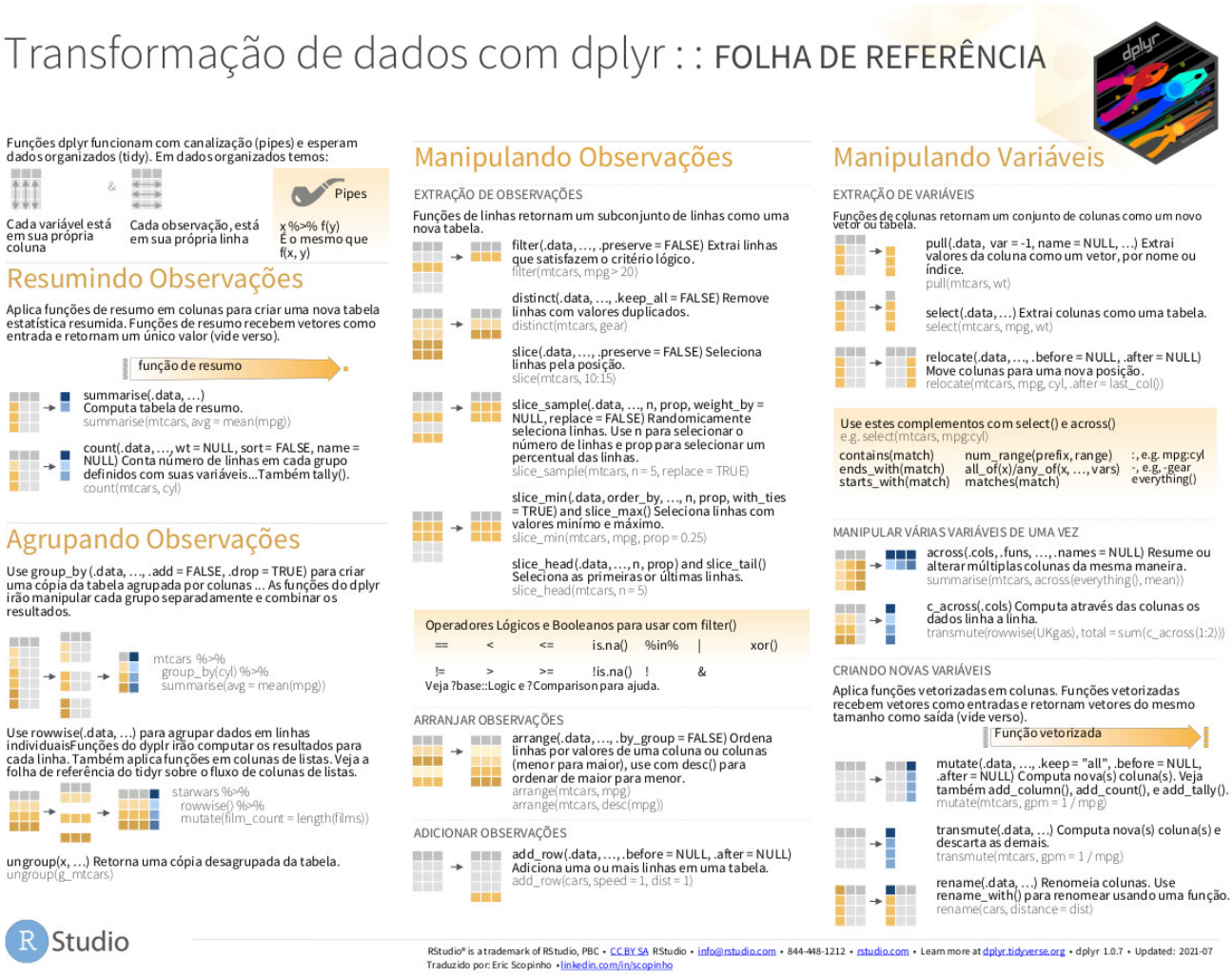
O pacote **gt** será utilizado apenas para eventualmente a formatação de tabelas, de modo a deixar a saída de alguns comandos mais clara.

```
library (tidyverse)
library (gt)
```

1.0.1 Folhas de Referências

A maioria dos exemplos, visam ajudar na interpretação dos exemplos e funções encontradas na **Folha de Referência** dos pacotes (cheatsheets) disponível no site do RStudio.

A seguir, um exemplo de uma "**Folha de Referência**" do pacote **DPLYR**:



> **Nota**
>
> *Ao final de cada seção de código, você poderá encontrar a chamada à função* ***gt()****. Isto é apenas para a formatação da tabela de saída e não é necessário para que você entenda os comandos precedentes. Em alguns casos, onde o volume de dados de saída pode ser extenso, usamos também a função* ***head()*** *para mostrar apenas as linhas iniciais. Quando o exemplo possui muitas colunas de saída, eventualmente utilizamos a função* ***select()***

para selecionar apenas algumas colunas. A função ***print()*** *também pode estar presente afim de apresentar o resultado de alguma linha de código relevante para o entendimento.*

2 Importação de Dados com TIDYVERSE

2.1 Introdução

A seguir temos vários exemplos de importação de dados utilizando o pacote TIDYVERSE do R. O pacote tidyverse possui vários pacotes de importação de dados, aqui iremos cobrir três deles (readr, readxl e googlesheets4). Para saber mais sobre estes pacotes, acesse:

https://cran.r-project.org/package=tidyverse.

https://cran.r-project.org/package=readr.

https://cran.r-project.org/package=readxl.

https://cran.r-project.org/package=googlesheets4.

> **Aviso**
>
> Para melhor utilizar este material, é importante que você tenha uma introdução à linguagem R e saiba carregar pacotes (packages) no R. Para mais informações acesse: https://education.rstudio.com/learn/beginner/.

Caso você precise trabalhar com outras formatos de arquivos que não sejam os vistos neste documento, pode buscar maiores informações sobre os pacotes a seguir:

Pacote	Formato
haven	Arquivos SPSS, Stata e SAS
DBI	Bancos de Dados
jsonlite	JSON
xml2	XML
httr	Web APIs
rvest	HTML (Web scraping)
readr::read_lines()	dados texto
pdftools	PDF

Para os exemplos, iremos carregar os seguintes pacotes:

- **tidyverse**
- **readxl**
- **googlesheets4**
- **gt**
- **openxlsx**

```
library (tidyverse)
library (readxl)
library (googlesheets4)
library (gt)
library (openxlsx)
```

2.1.1 Exemplos da Folha de Referência

A maioria dos exemplos, visam ajudar na interpretação dos exemplos e funções encontradas na **Folha de Referência** de importação de dados com tidyverse disponível no site do RStudio.

2.1.2 Arquivos

Para a maioria dos exemplos utilizaremos os seguintes arquivos de dados:

Alguns desses arquivos são baseados nas tabelas **mtcars, storms** e **starwars** provenientes do pacote **datasets e dplyr e** também algumas tabelas (**Table1**, **2, 3, 4a, 4b e 5**) que vem com o pacote **tidyr**.

ARQUIVOS TABULADOS: (TXT, CSV, TSV e FWF):

Iremos criar os arquivos tabulados para que possamos usá-los posteriormente. Para isso, execute o código abaixo:

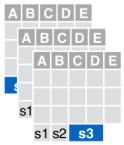
```
write_file("A|B|C\n1|2|3\n4|5|NA", file = "file.txt")
write_file("A,B,C\n1,2,3\n4,5,NA", file = "file.csv")
write_file("A;B;C\n1,5;2;3\n4,5;5;NA", file = "file2.csv")
write_file("A\tB\tC\n1\t2\t3\n4\t5\tNA\n", file = "file.tsv")
```

EXCEL_FILE.XLSX:

A seguir, você tem um link para o arquivo Excel utilizado nos exemplos.

Arquivo Exemplo - MS Excel

É um arquivo com três planilhas (S1, S2 e S3) e em cada uma delas um pequeno conjunto de dados.



E a primeira planilha (S1) possui algo como:

GOOGLE_SHEET:

A seguir, você tem o link para a planilha do google que será utilizado mais adiante.

Planilha exemplo - Google Sheets

2.2 READR

O pacote readr possui diversas funções para ler dados tabulados (ex: .csv, .tsv, .txt, etc). Estas funções começam com read_*().

read_* *(file, col_names = TRUE, col_types = NULL, col_select = NULL, id = NULL, locale, n_max = Inf, skip = 0, na = c("","NA"), guess_max = min(1000, n_max), show_col_types = TRUE)*

Os parametros acima, são comuns à estas funções. Veja a seguir algumas delas. Digite **?read_delim** para obter maiores detalhes de como utilzá-las.

2.2.1 Ler dados tabulados com readr

2.2.1.1 read_delim

Use para ler um arquivo tabulado com qualquer delimitador. Se nenhum delimitador é especificado, a função tentará advinhar automaticamente.

Por exemplo, para ler um arquivo .TXT tabulado com o caractere "|" como delimitador, fazemos:

```
read_delim("file.txt", delim = "|")
```

```
# A tibble: 2 x 3
      A     B     C
  <dbl> <dbl> <dbl>
1     1     2     3
2     4     5    NA
```

> Dica
>
> Para armazenar a leitura do arquivo em um objeto no R, podemos usar o operador <-.

```
meu_arquivo_csv <-read_delim("file.txt", delim = "|")
meu_arquivo_csv
```

```
# A tibble: 2 x 3
      A     B     C
  <dbl> <dbl> <dbl>
1     1     2     3
2     4     5    NA
```

2.2.1.2 read_cvs

Use para ler um arquivo tabulado **separado por vírgula**. Esta função entende que casas decimais que usam o **ponto** (ex 1.00) como separador de **casas decimais**.

```
read_csv("file.csv")
```

```
# A tibble: 2 x 3
      A     B     C
  <dbl> <dbl> <dbl>
1     1     2     3
2     4     5    NA
```

2.2.1.3 read_cvs2

Use para ler um arquivo tabulado **separado por ponto-e-vírgula**. Esta função entende que casas decimais que usam a **vírgula** (ex: 1,00) como separador de **casas decimais**.

```
read_csv2("file2.csv")
```

```
# A tibble: 2 x 3
      A     B     C
  <dbl> <dbl> <dbl>
1   1.5     2     3
2   4.5     5    NA
```

2.2.1.4 read_tsv

Use para ler um arquivo tabulado **separado por tab**.

```
read_tsv("file.tsv")
```

```
# A tibble: 2 x 3
      A     B     C
  <dbl> <dbl> <dbl>
1     1     2     3
2     4     5    NA
```

2.2.1.5 read_fwf

Use para ler um arquivo tabulado **com tamanhos fixos de colunas**.

> **Nota**
>
> Veja que a largura das colunas deve ser passada como um vetor para a parametro col_positions = usando a função fwf_width().

```r
read_fwf("file.tsv", fwf_widths(c(2,2,NA)))
```

```
# A tibble: 3 x 3
  X1    X2    X3
  <chr> <chr> <chr>
1 A     B     C
2 1     2     3
3 4     5     <NA>
```

2.2.2 Parâmetros Úteis

Alguns parametros das funções read_*() são muito úteis durante o processo de leitura pois permitem controlar melhor o que iremos obter como resultado da leitura.

2.2.2.1 Sem cabeçalho

Use o parâmetro **COL_NAMES** para não trazer a primeira linha como nome das colunas.

```r
read_csv2("file2.csv", col_names = FALSE)
```

```
# A tibble: 3 x 3
  X1    X2    X3
  <chr> <chr> <chr>
1 A     B     C
2 1,5   2     3
3 4,5   5     <NA>
```

2.2.2.2 Definir cabeçalho

Use o parâmetro **COL_NAMES** para definir manualmente os nomes das colunas.

```r
read_csv("file.csv", col_names = c("X", "Y", "Z"))
```

```
# A tibble: 3 x 3
  X     Y     Z
  <chr> <chr> <chr>
1 A     B     C
```

```
2 1     2     3
3 4     5     <NA>
```

2.2.2.3 Ler vários arquivos

Use o parametro **ID** para ler multiplos arquivos e armazená-los em uma mesma tabela.

```
write_file("A,B,C\n1,2,3\n4,5,NA", file = "f1.csv")
write_file("A,B,C\n6,7,8\n9,10,11", file = "f2.csv")
read_csv(c("f1.csv", "f2.csv"), id = "arq_origem")
```

```
# A tibble: 4 x 4
  arq_origem     A     B     C
  <chr>      <dbl> <dbl> <dbl>
1 f1.csv         1     2     3
2 f1.csv         4     5    NA
3 f2.csv         6     7     8
4 f2.csv         9    10    11
```

> **!** Importante
>
> Observe que as colunas dos diversos arquivos devem corresponder, ou seja, ter o mesmo nome de colunas.

2.2.2.4 Pular linhas

Use o prâmetro SKIP para pular as primeiras n linhas.

```
read_csv("file.csv", skip = 1)
```

```
# A tibble: 1 x 3
    `1`   `2` `3`
  <dbl> <dbl> <lgl>
1     4     5 NA
```

2.2.2.5 Ler um número máximo de linhas

Use o prâmetro **N_MAX** para ler um número máximo de linhas.

```r
read_csv("file.csv", n_max = 1)
```

```
# A tibble: 1 x 3
      A     B     C
  <dbl> <dbl> <dbl>
1     1     2     3
```

2.2.2.6 Ler valores como NA

Use o prâmetro **NA** para definir um ou mais valores como NA.

```r
read_csv("file.csv", na = c("1"))
```

```
# A tibble: 2 x 3
      A     B C
  <dbl> <dbl> <chr>
1    NA     2 3
2     4     5 NA
```

2.2.2.7 Especificar caractere decimal

Use o prâmetro **LOCALE** para definir o caractere de casa decimais.

```r
read_delim("file2.csv", locale = locale(decimal_mark = ","))
```

```
# A tibble: 2 x 3
      A     B     C
  <dbl> <dbl> <dbl>
1   1.5     2     3
2   4.5     5    NA
```

2.2.3 Salvar dados com readr

Similar às funções descritas na seção "Ler dados tabulados com readr" usadas para ler os aqruivos de texto tabulados, temos o conjunto de funções **write_*()** para gravar os arquivos correspondentes. Estas funções seguem o seguinte padrão:

write_*(x, file, na = "NA", append, col_names, quote, escape, eol, num_threads, progress)

As principais funções são:

2.2.3.1 write_delim

Use para gravar um arquivo delimitado por algum caractere específico. O parametro delim= permite definir este caractere. O caracteres padrão é o espaço (" ").

Por exemplo, se quisermos gravar uma tabela (tibble) em um arquivo .txt delimitado por ponto-e-vírgula";", podemos usar:

```
conteudo <- tribble(~col_A, ~col_B,
                    1, "A",
                    2, "B",
                    3, "C")
write_delim(conteudo, file = "arquivo_exemplo1.txt", delim=";")
```

2.2.3.2 write_csv

Use para gravar uma tabela em uma arquivo delimitado por "vírgula".

> Dica
>
> Podemos usar o arqumento **na =** para definirmos qual valor será usando para os valore ausentes, por padrão é utilizado "NA". No exemplo a seguir, iremos trocar por "NULL".

```
conteudo <- tribble(~col_A, ~col_B,
                    1, "A",
                    2, "B",
                    3, NA,
                    4, "D")
write_csv(conteudo, file = "arquivo_exemplo2.csv", na = "NULL")
```

2.2.3.3 write_csv2

Use para gravar uma tabela em um arquivo delimitado por "ponto-e-vírgula".

> Dica
>
> Pode usar o parametro "**col_names =**" para incluir ou não os nomes das colunas no arquivo de saída. No exemplo a seguir, não iremos incluir os nomes das colunas:

```
conteudo <- tribble(~col_A, ~col_B,
                    1, "A",
                    2, "B",
                    3, "C")
write_csv2(conteudo, file = "arquivo_exemplo3.csv", col_names = FALSE)
```

2.2.3.4 write_tsv

Use para gravar uma tabela em um arquivo delimitado por "TAB":

```
conteudo <- tribble(~col_A, ~col_B,
                    1, "A",
                    2, "B",
                    3, "C")
write_tsv(conteudo, file = "arquivo_exemplo4.tsv")
```

2.2.4 Especificação de colunas com readr

Ao importar um arquivo com readr, podemos definir qual o tipo de coluna que determinado dado será importado. Por padrão, o readr irá gerar a especificação de cada coluna quando o arquivo form lido e gerará um resumo na saída.

Podemos usar o argumento **spec()** para extrair as especificações das colunas de um arquivo importato para um data frame.

Por exemplo:

```
arq <- read_csv2("file2.csv")
spec(arq)
```

```
cols(
  A = col_double(),
  B = col_double(),
  C = col_double()
)
```

Observe que as colunas “A”, “B” e “C” são do formato **double**.

Há também uma mensagem de resumo ao importar um arquivo. Observe que ele informa o delimitador utilzado, mas também a especificação das colunas, neste caso, tipo double (**dbl**) para as colunas **A**, **B** e **C** conforme confirmamos com a função **spec()**.

Using “‘,’” as decimal and “‘.’” as grouping mark. Use `read_delim()` for more control. Rows: 2 Columns: 3 Column specification Delimiter: “;” dbl (3): A, B, C Use `spec()` to retrieve the full column specification for this data. Specify the column types or set `show_col_types = FALSE` to quiet this message. # READXL

> **Dica**
>
> Se quisermos omitir as especificações das colunas da mensagem de saída, usamos o parametro **show_col_types** = FALSE

2.2.4.1 col_types

Se utilizarmos o parametro **col_types** = podemos definir, por exemplo, a coluna “B” como inteiro (integer). Veja:

```
arq <- read_csv2("file2.csv", col_types = "did")
spec(arq)
```

```
cols(
  A = col_double(),
  B = col_integer(),
  C = col_double()
)
```

Há uma letra definida para cada tipo de coluna que quisermos especificar, veja a lista abaixo:

- *col_logical() - "l"*
- *col_integer() - "i"*
- *col_double() - "d"*
- *col_number() - "n"*
- *col_character() - "c"*
- *col_factor(levels, ordered = FALSE) - "f"*
- *col_datetime(format = "") -"T"*
- *col_date(format = "") -"D"*
- *col_time(format = "") -"t"*
- *col_skip() - "-", "_"*
- *col_guess() - "?"*

Por isso, usamos string **"did"** para definir um **double**, um **inteiro** e outro **double** para as colunas que importamos.

Podemos também passar a especificação das colunas como uma lista mesclando as funções e os caracteres correspondentes na lista acima.

Por exemplo:

```
arq <- read_csv2("file2.csv",
          col_types = list(A = col_double(), B = "i", C= "d")
          )
spec(arq)
```

```
cols(
  A = col_double(),
  B = col_integer(),
  C = col_double()
)
```

> Dica
>
> Use "**.default =**" na lista de especificações para definir o tipo padrão para as colunas, caso as mesmas não sejam explicitamente definidas.

2.2.4.2 col_select

Para selecionarmos apenas algumas colunas para importar do arquivo, utilzamos o parametro **col_select** = passanto um vetor com o nomes das colunas.

Por exemplo, para importar apenas as colunas "A" e "C", podemos fazer:

```
read_csv("file.csv", col_select = c("A", "C"))
```

```
# A tibble: 2 x 2
      A     C
  <dbl> <dbl>
1     1     3
2     4    NA
```

2.2.4.3 guess_max

Para definirmos o número máximo de linhas do arquivo para advinhar o tipo da coluna (guess), utilizamos o parametro **guess_max** =. O padrão são as primeiras 1000 linhas.

```
read_csv("file.csv",guess_max = 2)
```

```
# A tibble: 2 x 3
      A     B     C
  <dbl> <dbl> <dbl>
1     1     2     3
2     4     5    NA
```

2.3 READXL

Para lermos arquivos do Microsoft Excel, podemos usar o pacote **readxl**.

2.3.1 Ler arquivos do Excel

Apesar do pacote readxl ser instalado quando instalamos o pacote tidyverse, ele não é carregado quando carregamos o tidyverse. É por isso, que tivemos o código "library (readxl) na seção Introdução

2.3.1.1 read_excel

Use para ler um arquivo do Excel (.xls ou .xlsx) baseado na extensão do arquivo.

Se preferir, pode utilizar as funções read_xls() e read_xlsx() para ler um arquivo com .xls ou .xlsx independente da extensão do arquivo.

```
read_excel("excel_file.xlsx")
```

```
# A tibble: 2 x 5
  x1       x2 x3       x4    x5
  <chr> <dbl> <chr> <dbl> <dbl>
1 x        NA z         8    NA
2 y         7 <NA>      9    10
```

2.3.2 Ler planilhas

Sabemos que um arquivo Excel (*workbook*), pode conter uma ou mais planilhas (*worksheets*). Para definirmos as planilhas que precisamos importar, podemos utilizar o parametros **sheet** = da função read_excel(). Podemos passar uma string com o nome a planilha (ex: "S1") ou um índice númerico pela ordem de criação da planilha (ex: 1). Se nada for especificado, padrão é trazer a primeira planilha.

```
read_excel("excel_file.xlsx", sheet = "S1")
```

```
# A tibble: 2 x 5
  x1       x2 x3       x4    x5
  <chr> <dbl> <chr> <dbl> <dbl>
1 x        NA z         8    NA
2 y         7 <NA>      9    10
```

Para obter os nomes das planilhas presentes no arquivo, utilizamos a função **excel_sheets**()

```
excel_sheets("excel_file.xlsx")
```

```
[1] "S1" "S2" "S3"
```

> **Dica**
>
> Para lermos **múltiplas planilhas** podemos obter os nomes das planilhas usando a função excel_sheets(), pois definimos os nomes do vetor iguais aos nomes das planilhas e finalmente utilizamos a função purrr::map_dfr() para importar os arquivos no data frame.

```
arq <- "excel_file.xlsx"
arq |>
  excel_sheets() |>
  set_names() |>
  map_dfr(read_excel, path = arq)
```

```
# A tibble: 6 x 15
  x1       x2 x3       x4    x5 y1       y2 y3       y4    y5 z1       z2 z3
  <chr> <dbl> <chr> <dbl> <dbl> <chr> <dbl> <chr> <dbl> <dbl> <chr> <dbl> <chr>
1 x        NA z         8    NA <NA>     NA <NA>     NA    NA <NA>     NA <NA>
2 y         7 <NA>      9    10 <NA>     NA <NA>     NA    NA <NA>     NA <NA>
3 <NA>     NA <NA>     NA    NA x        NA z         8    NA <NA>     NA <NA>
4 <NA>     NA <NA>     NA    NA y         7 <NA>      9    10 <NA>     NA <NA>
5 <NA>     NA <NA>     NA    NA <NA>     NA <NA>     NA    NA x        NA z
6 <NA>     NA <NA>     NA    NA <NA>     NA <NA>     NA    NA y         7 <NA>
# ... with 2 more variables: z4 <dbl>, z5 <dbl>
# i Use `colnames()` to see all variable names
```

2.3.3 Especificação de colunas

Para especificar os tipos das colunas no data frame após a importação do arquivo, usamos o parametro col_types =, similar ao que fizemos para arquivos tabulados na seção Especificação de colunas com readr.

Os tipos de colunas podemos ser:

“skip”, “guess”, “logical”, “numeric”, “date”, “text” ou “list”.

> **Dica**
>
> Use uma coluna de lista (list-column) descrita no pacote **tidyr** para trabalhar com colunas com vários tipos.

2.3.4 Outros pacotes

Além do pacote readxl, há outros pacotes muito úteis para criar arquivos do MS Excel, tais como:

- **openxlsx**
- **writexl**

Para trabalhar com dados do Excel de forma não tabular, veja o pacote:

- **tidyxl**

2.3.5 Especificação de celulas

Use os argumentos **range =** para a função read_excel() ou googlesheets4::read_sheet() no caso de planilhas do Google para ler um subconjunto de células de uma planilha.

Por exemplo, se quiser ler **apenas o range** de células de "**A1**" até "**B3**" da planilha "S2" do arquivo excel de exemplo, por fazer:

```
read_excel("excel_file.xlsx", range = "S2!A1:B3")
```

```
# A tibble: 2 x 2
  y1       y2
  <chr> <dbl>
1 x        NA
2 y         7
```

O parametro range = , possui alguns argumentos que ajudam a melhor definir o range a ser importado. Veja ?'cell-specification' para maiores detalhes de como **cell_cols**(), **cell_rows**(), **cell_limits**() e **anchored**(). Por exemplo, usando cell_cols, podemos definir que iremos importar apenas as celulas que das colunas "B" até "D":

```
read_excel("excel_file.xlsx", sheet = "S1",
           range = cell_cols("B:D"))
```

```
# A tibble: 2 x 3
     x2 x3       x4
  <dbl> <chr> <dbl>
1    NA z         8
2     7 <NA>      9
```

2.4 GOOGLESHEETS4

2.4.1 Ler planilhas

2.4.1.1 read_sheet

Use para ler **planilhas do Google** a partir de uma **URL**, um IDde planilha ou um objeto do tipo "**dribble**" que é retornado pelo pacote googledrive. Esta função é um "apelido" para a função **range_read**() que é mais utilizada no contexto do pacote googlesheets4.

Diversos argumtos vistos para as funções read_* são aplicadas aqui também, como col_types = , sheet =, range = , guess_max = . Veja mais detalhes na seção do **readr** descrita anteriormente.

No exemplo a seguir iremos ler uma planilha do Google de exemplo. Para isso, recebemos o seguinte URL. Veja que a partes em negrito corresponde ao I**D do arquivo** e o **ID da planilha** respectivamente:

*https://docs.google.com/spreadsheets/d/**1__aRR__9UcMytZqjID0BkJ7PW29M1kt1__x2HxhBZOlFN8**/e*

Usamos então a função read_sheet():

```
googlesheets4::read_sheet("1_aRR_9UcMytZqjID0BkJ7PW29M1kt1_x2HxhBZOlFN8", sheet = "Sheet1"
```

```
# A tibble: 5 x 3
      A B     C
  <dbl> <chr> <chr>
1     1 A     XX
2     2 B     YY
3     3 C     ZZ
4     4 D     WW
5     5 E     AA
```

> **Cuidado**
>
> A primeira vez que executar este comendo, haverá um processo de autenticação da sua conta do Google e seeão do R. Reponda "**Yes**" para a pergunta "**Is it OK to cache OAuth access credentials in the folder ~/.cache/gargle between R sessions?**"
> 1: Yes
> 2: No
> Depois o navegador será aberto solicitando o acesso aos arquivo do Google. Selecoine o checkbox e click em "Continue".

2.4.2 Metadados das planilhas

2.4.2.1 gs4_gets

Use para obter os metadados do arquivo:

```
gs4_get("1_aRR_9UcMytZqjID0BkJ7PW29M1kt1_x2HxhBZOlFN8")
```

```
Spreadsheet name: tidyverse_exemplo
              ID: 1_aRR_9UcMytZqjID0BkJ7PW29M1kt1_x2HxhBZOlFN8
          Locale: en_US
       Time zone: America/Sao_Paulo
     # of sheets: 39

(Sheet name): (Nominal extent in rows x columns)
      Sheet1: 1000 x 26
          df: 4 x 2
      Sheet2: 4 x 2
      Sheet3: 4 x 2
      Sheet4: 4 x 2
      Sheet5: 4 x 2
      Sheet6: 4 x 2
      Sheet7: 4 x 2
      Sheet8: 4 x 2
      Sheet9: 4 x 2
     Sheet10: 4 x 2
     Sheet11: 4 x 2
     Sheet12: 4 x 2
     Sheet13: 4 x 2
     Sheet14: 4 x 2
     Sheet15: 4 x 2
     Sheet16: 4 x 2
     Sheet17: 4 x 2
     Sheet18: 4 x 2
     Sheet19: 4 x 2
     Sheet20: 4 x 2
     Sheet21: 4 x 2
     Sheet22: 4 x 2
     Sheet23: 4 x 2
     Sheet24: 4 x 2
     Sheet25: 4 x 2
     Sheet26: 4 x 2
```

```
  Sheet27: 4 x 2
  Sheet28: 4 x 2
  Sheet29: 4 x 2
  Sheet30: 4 x 2
  Sheet31: 4 x 2
  Sheet32: 4 x 2
  Sheet33: 4 x 2
  Sheet34: 4 x 2
  Sheet35: 4 x 2
  Sheet36: 4 x 2
  Sheet37: 4 x 2
  Sheet38: 4 x 2
```

2.4.2.2 gs4_find

Use para localizar suas planilhas do Google no drive. Ela retorna um objeto dibble, que é um "tibble" com uma linha por arquivo. E informa o ID dos arquivos.

```
my_dribble <- gs4_find(pattern = "tidyverse_exemplo")
my_dribble
```

```
# A dribble: 1 x 3
  name              id                                          drive_resource
  <chr>             <drv_id>                                    <list>
1 tidyverse_exemplo 1_aRR_9UcMytZqjID0BkJ7PW29M1kt1_x2HxhBZOlFN8 <named list>
```

sheet_properties

Use para obter uma tabela (tibble) com as propriedades de cada planilha.

```
sheet_properties("1_aRR_9UcMytZqjID0BkJ7PW29M1kt1_x2HxhBZOlFN8")
```

```
# A tibble: 39 x 8
   name   index         id type  visible grid_rows grid_columns data
   <chr>  <int>      <int> <chr> <lgl>       <int>        <int> <list>
 1 Sheet1     0          0 GRID  TRUE         1000           26 <NULL>
 2 df         1 1125179472 GRID  TRUE            4            2 <NULL>
 3 Sheet2     2 1251368119 GRID  TRUE            4            2 <NULL>
 4 Sheet3     3 1983355079 GRID  TRUE            4            2 <NULL>
 5 Sheet4     4  801422320 GRID  TRUE            4            2 <NULL>
```

```
 6 Sheet5     5 1405654832 GRID  TRUE           4           2 <NULL>
 7 Sheet6     6  772870347 GRID  TRUE           4           2 <NULL>
 8 Sheet7     7 1763203995 GRID  TRUE           4           2 <NULL>
 9 Sheet8     8  761411920 GRID  TRUE           4           2 <NULL>
10 Sheet9     9  234486202 GRID  TRUE           4           2 <NULL>
# ... with 29 more rows
# i Use `print(n = ...)` to see more rows
```

> Dica
>
> Você pode usar a função **sheet_names**() para obter os nomes da planilha dentro do arquivo.

2.4.3 Gravar planilhar

O pacote googlesheets4 tem várias maneiras de gravar dados em uma planilha.

2.4.3.1 write_sheet

Use esta função para salvar um data frame em uma planilha no arquivo do Google Sheets. Se a planilha não existir, ele cria uma planilha co mum nome aleatório através da função gs4_create().

```
df <- tribble(~x, ~y,
              1, "A",
              2, "B",
              3, "C")
write_sheet(df, "1_aRR_9UcMytZqjIDOBkJ7PW29M1kt1_x2HxhBZOlFN8")
read_sheet("1_aRR_9UcMytZqjIDOBkJ7PW29M1kt1_x2HxhBZOlFN8", sheet = "df")
```

```
# A tibble: 3 x 2
      x y
  <dbl> <chr>
1     1 A
2     2 B
3     3 C
```

2.4.3.2 gs4_create

Use para criar uma nova planilha do Google. Você pode fornecer o nome, mas caso não o faça o Google irá atribuir um nome aleatorio ao seu arquivo.

```
minha_planilha <- gs4_create(name = "meu_novo_arquivo_google_sheet", sheets = "Sheet1")
sheet_properties(minha_planilha)
```

```
# A tibble: 1 x 8
  name   index        id type  visible grid_rows grid_columns data
  <chr>  <int>     <int> <chr> <lgl>       <int>        <int> <list>
1 Sheet1     0 196384678 GRID  TRUE         1000           26 <NULL>
```

2.4.4 Especificação de colunas

Para especificar os tipos das colunas no data frame após a importação da planilha do Google, usamos o parametro **col_types =** como argumento da função **read_sheet/range_read()**, similar ao que fizemos para arquivos tabulados na seção Especificação de colunas com readr.

Os tipos de colunas aceitos são:

- skip - "_" ou "-"
- guess - "?"
- logical - "l"
- integer - "i"
- double - "d"
- numeric - "n"
- date - "D"
- datetime - "T"
- character - "c"
- list-column - "L"
- cell - "C" Retorna uma lista bruta dos dados das células.

2.4.5 Especificação de celulas - Google Sheets

Ver seção Especificação de celulas

2.4.6 Operadores de arquivos

O pacote googlesheets4 oferece várias forma de manipular os aspectos da planilha como congelar linhas, definir largura das colunas, etc. Acesse googlesheets4.tidyverse.org para mais informações.

Para operções de arquivos (ex: renomear, compartilhar, mover para outra pasta, etc), veja o pacote googledrive no link: googledrive.tidyverse.org.

3 Organização de Dados com TIDYR

3.1 Introdução

A seguir temos vários exemplos de organização de dados utilizando o pacote TIDYR do R. Para saber mais sobre este pacote, acesse:

https://cran.r-project.org/package=tidyr.

> **Aviso**
>
> Para melhor utilizar este material, é importante que você tenha uma introdução à linguagem R e saiba carregar pacotes (packages) no R. Para mais informações acesse: https://education.rstudio.com/learn/beginner/.

Para os exemplos, iremos carregar os seguintes pacotes:

- **tidyverse**
- **gt**

```
library (tidyverse)
library (gt)
```

3.1.1 Exemplos da Folha de Referência

A maioria dos exemplos, visam ajudar na interpretação dos exemplos e funções encontradas na **Folha de Referência** do tidyr disponível no site do RStudio.

3.1.2 Conjunto de Dados

Para a maioria dos exemplos utilizaremos as bases de dados **mtcars, storms** e **starwars** provenientes do pacote **datasets e dplyr e** também algumas tabelas (**Table1**, **2, 3, 4a, 4b e 5**) que vem com o pacote **tidyr**.

MTCARS: Dados de consumo de combustível, performance e design de 32 automóveis (*1974 Motor Trend US magazine*)

```
mtcars |>
  head ()
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

STORMS: Dados de furacões entre 1975-2020 medidos a cada 6 horas durante cada tempestade (*NOAA Atlantic hurricane database*),

```
storms |>
  head ()
```

```
# A tibble: 6 x 13
  name   year month   day  hour   lat  long status categ~1  wind press~2 tropi~3
  <chr> <dbl> <dbl> <int> <dbl> <dbl> <dbl> <chr>  <ord>   <int>   <int>   <int>
1 Amy    1975     6    27     0  27.5 -79   tropi~ -1         25    1013      NA
2 Amy    1975     6    27     6  28.5 -79   tropi~ -1         25    1013      NA
3 Amy    1975     6    27    12  29.5 -79   tropi~ -1         25    1013      NA
4 Amy    1975     6    27    18  30.5 -79   tropi~ -1         25    1013      NA
5 Amy    1975     6    28     0  31.5 -78.8 tropi~ -1         25    1012      NA
6 Amy    1975     6    28     6  32.4 -78.7 tropi~ -1         25    1012      NA
```

```
# ... with 1 more variable: hurricane_force_diameter <int>, and abbreviated
#   variable names 1: category, 2: pressure, 3: tropicalstorm_force_diameter
# i Use `colnames()` to see all variable names
```

STARWARS: Dados dos personagens de STAR WARS

```
starwars |>
  select(1:8) |>
  head()
```

```
# A tibble: 6 x 8
  name           height  mass hair_color  skin_color  eye_color birth_year sex
  <chr>           <int> <dbl> <chr>       <chr>       <chr>          <dbl> <chr>
1 Luke Skywalker    172    77 blond       fair        blue            19   male
2 C-3PO             167    75 <NA>        gold        yellow         112   none
3 R2-D2              96    32 <NA>        white, blue red             33   none
4 Darth Vader       202   136 none        white       yellow          41.9 male
5 Leia Organa       150    49 brown       light       brown           19   fema~
6 Owen Lars         178   120 brown, grey light       blue            52   male
```

TABELAS EXEMPLOS - Table1, 2, 3, 4a, 4b e 5

3.1.2.1 Table1

```
table1
```

```
# A tibble: 6 x 4
  country      year  cases population
  <chr>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

3.1.2.2 Table2

```
table2
```

```
# A tibble: 12 x 4
   country      year type            count
   <chr>       <int> <chr>           <int>
 1 Afghanistan  1999 cases             745
 2 Afghanistan  1999 population   19987071
 3 Afghanistan  2000 cases            2666
 4 Afghanistan  2000 population   20595360
 5 Brazil       1999 cases           37737
 6 Brazil       1999 population  172006362
 7 Brazil       2000 cases           80488
 8 Brazil       2000 population  174504898
 9 China        1999 cases          212258
10 China        1999 population 1272915272
11 China        2000 cases          213766
12 China        2000 population 1280428583
```

3.1.2.3 Table3

```
table3
```

```
# A tibble: 6 x 3
  country      year rate
* <chr>       <int> <chr>
1 Afghanistan  1999 745/19987071
2 Afghanistan  2000 2666/20595360
3 Brazil       1999 37737/172006362
4 Brazil       2000 80488/174504898
5 China        1999 212258/1272915272
6 China        2000 213766/1280428583
```

3.1.2.4 Table4a

```
table4a
```

```
# A tibble: 3 x 3
  country     `1999` `2000`
* <chr>        <int>  <int>
1 Afghanistan    745   2666
2 Brazil       37737  80488
3 China       212258 213766
```

3.1.2.5 Table4b

```
table4b
```

```
# A tibble: 3 x 3
  country         `1999`     `2000`
* <chr>            <int>      <int>
1 Afghanistan   19987071   20595360
2 Brazil       172006362  174504898
3 China       1272915272 1280428583
```

3.1.2.6 Table5

table5

> **Nota**
>
> *O termo data-frame descrito ao longo deste texto, é utilizado de forma livre para objetos do tipo data.frame, tibble, entre outros. Pense como se fosse uma tabela de um banco de dados e/ou uma planilha do MS Excel, contendo linhas e colunas. Apesar de não ser rigorosamente igual à uma tabela, muitas vezes usaremos estes termos de forma intercambiável para facilitar o entendimento de iniciantes.*

3.1.3 Dados Organizados e Canalização

Dados organizados (*tidy*) são estruturados onde:

Cada **variável** está em sua própria **coluna** e cada **observação** está em sua própria **linha**.

As **variáveis** (ou colunas) são acessadas como vetores.

Os **observações** (ou linhas) são preservadas em operações vetorizadas, ou seja, quando utilizamos funções que recebem vetores na entrada e retornam vetores na sua saída.

3.2 Tibbles

Podemos considerar que "**Tibbles**" são objetos similares aos "**Data Frames**", porém com algumas melhorias/vantagens. Estes objetos são fornecidos pelo packago **tibble**. Eles herdam a classe data frame, mas possuem alguns comportamentos melhorados, como:

- Extrai parte de um tibble usando colchetes] e um vetor com duplo colchetes]] ou $
- Sem encontros parciais quando extraindo partes das colunas
- Mostram uma resumo mais amigável na tela quando pedimos suas informações. Use options(tibble.print_max = n, tibble.print_min = m, tibble.width = Inf) para controlar a saída padrão.
- As funções View() ou glimpse() permitem visualizar todo o conjunto de dados.

Por exemplo, se tivermos um tibble (ex starwars) e quisermos acessar a coluna "name", e obter um **vetor**, podemos usar:

```
starwars[["name"]] #Por nome com [[  ]]
starwars$name      #Por nome com $
starwars[[1]]      #Por numero da coluna com [[  ]]
```

Já se tivermos um tibble (ex starwars) e quisermos acessar a coluna "name", e obter um **tibble**, podemos usar:

```
starwars[ , 1]            #Por numero com [   ]
select (starwars, 3)      #Por numero usando select()
select (starwars, name)   #Por nome usando select()
```

3.2.1 Criando tibble

Podemos utilziar as funções tibble() ou tribble() para criar a mesma tabela. A diferença é apenas na forma em que os parêmetros são utilizados.

3.2.1.1 tibble

Use para criar um tibble por colunas.

```
tibble(x = 1:3, y = c("a", "b", "c"))
```

```
# A tibble: 3 x 2
      x y
  <int> <chr>
1     1 a
2     2 b
3     3 c
```

3.2.1.2 tribble

Use para criar um tibble por linhas.

```
tribble(~x, ~y,
        1, "a",
        2, "b",
        3, "c",
        4, "d")
```

```
# A tibble: 4 x 2
      x y
  <dbl> <chr>
1     1 a
2     2 b
3     3 c
4     4 d
```

3.2.1.3 as_tibble

Use para converter um data frame para um tibble.

Por exemplo, para converter o data frame MTCARS para um tibble, fazemos:

```
as_tibble(mtcars)
```

```
# A tibble: 32 x 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
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# ... with 22 more rows
# i Use `print(n = ...)` to see more rows
```

3.2.1.4 enframe

Use para converter um vetor para um tibble. Use deframe() parta fazer o inverso.

3.2.1.5 is_tibble

Use para saber se um objeto é um tibble ou não.

```
is_tibble(mtcars)
```

```
[1] FALSE
```

3.3 Reformatando Dados

Muitas vezes, os dados que recebemos não estão organizados (tidy) da maneira como vimos na seção Dados Organizados e Canalização. Para casos onde temos, por exemplo, as variáveis em linhas e/ou observações em colunas, etc, precisamos fazer uma ação conhecida como "pivotagem". Pivotar dados, no contexto do tidyr, significa ajustar linhas em colunas e/ou colunas em linhas, de forma a obtermos nossos dados da maneira organizada (tidy).

Veja, por exemplo, nossa tabela 1 (table1). Ela está em um formato organizado, pois possui cada variável em uma coluna e cada observações em sua linha, com os valores nas células.

```
table1
```

```
# A tibble: 6 x 4
  country      year  cases population
  <chr>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

3.3.0.1 pivot_longer

Use para "pivotar" os dados das colunas para as linhas, **alongando** a tabela, juntando várias colunas em duas, sendo uma a colunas que receberá os nomes das colunas e outra que recebera os valores das colunas.

Por exemplo, vejamos nossa tabela 4 (table4).

```
table4a
```

```
# A tibble: 3 x 3
  country     `1999` `2000`
* <chr>        <int>  <int>
1 Afghanistan    745   2666
2 Brazil       37737  80488
3 China       212258 213766
```

Observe que temos uma variável (potencialmente "Ano") que está na colunas 2 e 3 da tabela. Temos também outra variável (potencialmente "Numero_de_Casos" que está nas células da tabela.

Para organizar esta tabela em nosso formato "tidy", devemos pegar estas duas colunas e usar a função pivot_longer definindo os nomes para as respectivas variáveis (ex: ano e num_caso).

```
pivot_longer(table4a, cols = 2:3, names_to ="ano",
values_to = "num_casos")
```

```
# A tibble: 6 x 3
  country     ano   num_casos
  <chr>       <chr>     <int>
1 Afghanistan 1999        745
2 Afghanistan 2000       2666
3 Brazil      1999      37737
4 Brazil      2000      80488
5 China       1999     212258
6 China       2000     213766
```

3.3.0.2 pivot_wider

Use para "pivotar" os dados das linhas para as colunas, **expandindo** a tabela gerando novas colunas.

Vejamos o caso da tabela 2 (table2).

```
table2
```

```
# A tibble: 12 x 4
   country      year type            count
   <chr>       <int> <chr>           <int>
 1 Afghanistan  1999 cases             745
 2 Afghanistan  1999 population   19987071
 3 Afghanistan  2000 cases            2666
 4 Afghanistan  2000 population   20595360
 5 Brazil       1999 cases           37737
 6 Brazil       1999 population  172006362
 7 Brazil       2000 cases           80488
 8 Brazil       2000 population  174504898
 9 China        1999 cases          212258
```

```
10 China        1999 population 1272915272
11 China        2000 cases          213766
12 China        2000 population 1280428583
```

Neste exemplo, vemos que as variáveis casos (cases) e população (population) estão nas linhas e não nas colunas. Para deixarmos os dados organizados (tidy), devemos "expandir" a tabela, fazendo com que os dados de duas colunas sejam expandidos em vaŕias colunas. Os nomes das novas colunas virão de uma colunas e os valores da outra coluna. Veja:

```
table2 |>
  pivot_wider(names_from = type, values_from = count)
```

```
# A tibble: 6 x 4
  country      year  cases population
  <chr>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

3.4 Expandindo Tabelas

Em algumas situações, precisamos criar novas combinações das variáveis ou identificar valores ausentes implícitos, ou seja, combinações de variáveis não presentes nos dados.

Para isto, temos as funções **expand**() e **complete**().

3.4.0.1 expand

Use para criar um novo tibble com todas as possibilidades de combinações dos valores das variáveis passadas para a função expand(), **ignorando** as demais variáveis.

Por exemplo, se quisermos obter todas as combinações possíveis entre o número de cilindros (cyl), marchas (gear) e numero de carburadores (carb) da tabela mtcars, e ignorar todas as demais variáveis da tabela, podemos usar:

```
mtcars |>
  expand(cyl, gear, carb)
```

```
# A tibble: 54 x 3
     cyl  gear  carb
   <dbl> <dbl> <dbl>
 1     4     3     1
 2     4     3     2
 3     4     3     3
 4     4     3     4
 5     4     3     6
 6     4     3     8
 7     4     4     1
 8     4     4     2
 9     4     4     3
10     4     4     4
# ... with 44 more rows
# i Use `print(n = ...)` to see more rows
```

Observe que não há na tabela original (mtcars) um veículo de 4 cilindros e 8 carburadores, porém esta combinação foi possível usando a função expand().

3.4.0.2 complete

Use para criar um novo tibble com todas as possibilidades de combinações dos valores das variáveis passadas para a função expand(), **colocando NA** nas demais variáveis.

Por exemplo, se quisermos obter todas as combinações possíveis entre o número de cilindros (cyl), marchas (gear) e numero de carburadores (carb) da tabela mtcars, e colocar NA para as demais variáveis em que a combinação não exista, podemos usar:

```
mtcars |>
  complete(cyl, gear, carb)
```

```
# A tibble: 74 x 11
     cyl  gear  carb   mpg  disp    hp  drat    wt  qsec    vs    am
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1     4     3     1  21.5 120.     97  3.7   2.46  20.0     1     0
 2     4     3     2  NA    NA      NA NA    NA     NA      NA    NA
 3     4     3     3  NA    NA      NA NA    NA     NA      NA    NA
 4     4     3     4  NA    NA      NA NA    NA     NA      NA    NA
 5     4     3     6  NA    NA      NA NA    NA     NA      NA    NA
 6     4     3     8  NA    NA      NA NA    NA     NA      NA    NA
 7     4     4     1  22.8 108      93  3.85  2.32  18.6     1     1
```

```
 8     4     4     1  32.4  78.7    66  4.08  2.2   19.5     1     1
 9     4     4     1  33.9  71.1    65  4.22  1.84  19.9     1     1
10     4     4     1  27.3  79      66  4.08  1.94  18.9     1     1
# ... with 64 more rows
# i Use `print(n = ...)` to see more rows
```

3.5 Combinando e Dividindo Celulas

Use as funções a seguir para dividir ou combinar células da tabela em valores individuais isolados.

3.5.0.1 unite

Use para combinar celulas de diversas colunas em uma única coluna.

Vejamos com é a tabela 5 (table5) em seu formato original:

```
table5
```

```
# A tibble: 6 x 4
  country     century year  rate
* <chr>       <chr>   <chr> <chr>
1 Afghanistan 19      99    745/19987071
2 Afghanistan 20      00    2666/20595360
3 Brazil      19      99    37737/172006362
4 Brazil      20      00    80488/174504898
5 China       19      99    212258/1272915272
6 China       20      00    213766/1280428583
```

Agora queremos unir as colunas “century” e “year” em uma única coluna:

```
table5 |>
  unite(century, year, col = "ano_completo", sep = "")
```

```
# A tibble: 6 x 3
  country     ano_completo rate
  <chr>       <chr>        <chr>
1 Afghanistan 1999         745/19987071
2 Afghanistan 2000         2666/20595360
```

```
3 Brazil      1999          37737/172006362
4 Brazil      2000          80488/174504898
5 China       1999          212258/1272915272
6 China       2000          213766/1280428583
```

> **Nota**
>
> Veja que as colunas que deram origem à coluna combinada não são retornadas na saída da função.

3.5.0.2 separate

Use para dividir cada célula de **uma coluna** em **várias colunas**.

Por exemplo, na tabela 3 (table3), temos uma coluna "rate" que possui daos dos casos e população separados por uma barr ("/"). Neste caso, podemos utilzar a função separate para dividí-la e criar duas novas colunas com seus dados separados.

```
table3 |>
  separate(rate, sep = "/",
           into = c("casos", "pop"))
```

```
# A tibble: 6 x 4
  country      year casos  pop
  <chr>       <int> <chr>  <chr>
1 Afghanistan  1999 745    19987071
2 Afghanistan  2000 2666   20595360
3 Brazil       1999 37737  172006362
4 Brazil       2000 80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

3.5.0.3 separate_rows

Use para dividir cada célula de **uma coluna** em **várias linhas**.

É similar a função separate, porém o conteúdo de cada célula irá para uma linha ao invés de uma colunas.

```r
table3 |>
  separate_rows(rate, sep = "/")
```

```
# A tibble: 12 x 3
   country      year rate
   <chr>       <int> <chr>
 1 Afghanistan  1999 745
 2 Afghanistan  1999 19987071
 3 Afghanistan  2000 2666
 4 Afghanistan  2000 20595360
 5 Brazil       1999 37737
 6 Brazil       1999 172006362
 7 Brazil       2000 80488
 8 Brazil       2000 174504898
 9 China        1999 212258
10 China        1999 1272915272
11 China        2000 213766
12 China        2000 1280428583
```

3.6 Lidando com Valores Ausentes

Muitas vezes precisamos **ignorar** ou **substituir** valores ausentes (**NA**). Para isso, podemos usar as funções drop_na(), fill() ou replace_na()

3.6.0.1 drop_na

Ignora linhas que possuem valores ausentes (NA) nas colunas.

Por exemplo, na tabela starwars, temos 5 personagens que não possuim cor de cabelo (hair color):

```r
starwars |>
  select(name, hair_color) |>
  filter(is.na(hair_color))
```

```
# A tibble: 5 x 2
  name                hair_color
  <chr>               <chr>
1 C-3PO               <NA>
```

```
2 R2-D2                <NA>
3 R5-D4                <NA>
4 Greedo               <NA>
5 Jabba Desilijic Tiure <NA>
```

Se pedirmos para listar todos os personagens e utilizarmos a função drop_na(), estes 5 personagens não serão listados:

```
starwars |>
  select(name, hair_color) |>
  drop_na()
```

```
# A tibble: 82 x 2
   name               hair_color
   <chr>              <chr>
 1 Luke Skywalker     blond
 2 Darth Vader        none
 3 Leia Organa        brown
 4 Owen Lars          brown, grey
 5 Beru Whitesun lars brown
 6 Biggs Darklighter  black
 7 Obi-Wan Kenobi     auburn, white
 8 Anakin Skywalker   blond
 9 Wilhuff Tarkin     auburn, grey
10 Chewbacca          brown
# ... with 72 more rows
# i Use `print(n = ...)` to see more rows
```

3.6.0.2 fill

Use para substituir o valores ausente (NA) da coluna pelo último valor disponível em linhas anteriores ou posteriores.

Por exemplo:

Como vimos no exemplo da função drop_na, temos 5 personagens de starwars que não possuem cor de cabelo preenchido.

Digamos que decidimos substituir estes NAs pelo cor de cabelo do personagem anterior disponível. Para isso, faremos:

```
starwars |>
  select (name, hair_color) |>
  fill(hair_color)
```

```
# A tibble: 87 x 2
   name               hair_color
   <chr>              <chr>
 1 Luke Skywalker     blond
 2 C-3PO              blond
 3 R2-D2              blond
 4 Darth Vader        none
 5 Leia Organa        brown
 6 Owen Lars          brown, grey
 7 Beru Whitesun lars brown
 8 R5-D4              brown
 9 Biggs Darklighter  black
10 Obi-Wan Kenobi     auburn, white
# ... with 77 more rows
# i Use `print(n = ...)` to see more rows
```

Veja que o personagem C-3PO que tinha a cor de cabelo não preenchida, agora está como loiro (blond), pois o personagem anteriormente preenchido, era o Luke Skywalker, que tinha a cor de cabelo loiro (blond). Já o personagem R5-D4, teve sua cor de cabelo preenchida de marron (brown), pois o personagem anterior Beru Whitesun lars, tinha o cabelo marron (brown).

3.6.0.3 replace_na

Use para substituir os valores de NA por um valor específico.

Por exemplo, vamos substituir a cor de cabelo dos personagens que tem NA na coluna "hair_color" pela cor azul (blue).

```
starwars |>
  select(name, hair_color) |>
  replace_na(list(hair_color = "blue"))
```

```
# A tibble: 87 x 2
   name               hair_color
   <chr>              <chr>
 1 Luke Skywalker     blond
```

```
 2 C-3PO              blue
 3 R2-D2              blue
 4 Darth Vader        none
 5 Leia Organa        brown
 6 Owen Lars          brown, grey
 7 Beru Whitesun lars brown
 8 R5-D4              blue
 9 Biggs Darklighter  black
10 Obi-Wan Kenobi     auburn, white
# ... with 77 more rows
# i Use `print(n = ...)` to see more rows
```

> **Aviso**
>
> Veja que a função replace_na, recebe uma lista de valores (list()) se os dados passados no primeiro parâmetro é um data frame.

3.7 Dados Aninhados

3.7.1 Introdução

Um data frame aninhado (nested) é aquela que possui tabelas completas em colunas do tipo lista (colunas de lista) dentro de outro data frame maior e organizado. Uma coluna de lista, podem ser também listas de vetores ou listas de vários tipos de dados.

Alguns uso de um data frame de dados aninhados são:

- Preservar o relacionamento entre observações e sub-grupos de dados.
- Preservar o tipo da variável aninhada (factors e datetime não viram caracteres por exemplo).
- Manipular várias sub-tabelas de uma vez com funcções do pacote purrr como map(), map2() ou pmap() ou com a rowwise() do pacote dplyr.

3.7.2 Criando dados Aninhados

3.7.2.1 nest

Use para mover grupos de células para uma coluna de lista de um data frame. Pode ser usada sozinha ou em conjunto com a grupo_by().

Exemplo 1: nest() com groupg_by()

Digamos que gostaríamos ter uma linha para cada furacão de nosso data frame (storms) e em uma coluna de lista, uma tabela dos dados do respectivo furação. Para isso, podemos utilizar a função nest() em conjunto com a função group_by(). Veja abaixo:

```
n_storms <- storms  |>
  group_by(name) |>
  nest()
head(n_storms, 15)
```

```
# A tibble: 15 x 2
# Groups:   name [15]
   name      data
   <chr>     <list>
 1 Amy       <tibble [30 x 12]>
 2 Caroline  <tibble [33 x 12]>
 3 Doris     <tibble [23 x 12]>
 4 Belle     <tibble [18 x 12]>
 5 Gloria    <tibble [125 x 12]>
 6 Anita     <tibble [20 x 12]>
 7 Clara     <tibble [24 x 12]>
 8 Evelyn    <tibble [9 x 12]>
 9 Amelia    <tibble [6 x 12]>
10 Bess      <tibble [13 x 12]>
11 Cora      <tibble [19 x 12]>
12 Juliet    <tibble [16 x 12]>
13 Ana       <tibble [100 x 12]>
14 Bob       <tibble [71 x 12]>
15 Claudette <tibble [180 x 12]>
```

Observe que na coluna “data”, temos um objeto <tibble> para grupo criado pela função group_by. A função nest() aninhou todas as demais variáveis nestas pequenas tabelas para cada um deles e armazenou na coluna do tipo lista.

Para acessar a tibble gerada para o primeiro grupo (furacão “Amy” na primeira linha acima), podemos fazer:

```
n_storms$data[[1]]
```

```
# A tibble: 30 x 12
    year month   day  hour   lat  long status      categ~1  wind press~2 tropi~3
   <dbl> <dbl> <int> <dbl> <dbl> <dbl> <chr>       <ord>   <int>   <int>   <int>
 1  1975     6    27     0  27.5 -79   tropical d~ -1         25    1013      NA
 2  1975     6    27     6  28.5 -79   tropical d~ -1         25    1013      NA
 3  1975     6    27    12  29.5 -79   tropical d~ -1         25    1013      NA
 4  1975     6    27    18  30.5 -79   tropical d~ -1         25    1013      NA
 5  1975     6    28     0  31.5 -78.8 tropical d~ -1         25    1012      NA
 6  1975     6    28     6  32.4 -78.7 tropical d~ -1         25    1012      NA
 7  1975     6    28    12  33.3 -78   tropical d~ -1         25    1011      NA
 8  1975     6    28    18  34   -77   tropical d~ -1         30    1006      NA
 9  1975     6    29     0  34.4 -75.8 tropical s~ 0          35    1004      NA
10  1975     6    29     6  34   -74.8 tropical s~ 0          40    1002      NA
# ... with 20 more rows, 1 more variable: hurricane_force_diameter <int>, and
#   abbreviated variable names 1: category, 2: pressure,
#   3: tropicalstorm_force_diameter
# i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

Para acessar todas as observações as variáveis “month” e “status” especificamente, podemos usar:

```
n_storms$data[[1]][, c("month", "status")] |>
  head()
```

```
# A tibble: 6 x 2
  month status
  <dbl> <chr>
1     6 tropical depression
2     6 tropical depression
3     6 tropical depression
4     6 tropical depression
5     6 tropical depression
6     6 tropical depression
```

Exemplo 2: nest() especificando colunas

Digamos que precisamos especfificar as colunas que gostaríamos de aninhar em cada linha da coluna de lista. Apenas para simplificar o exemplo, iremos selecionar com a função selct() apenas as colunas “name”, “year”, “lat” e “long”. Depois iremos aninhar as colunas “year” até a coluna “long” em um coluna de lista chamada “data”. Veja a seguir:

```r
n_storms <- storms  |>
  select(name, year, lat, long) |>
  nest(data = c(year:long))
head(n_storms)
```

```
# A tibble: 6 x 2
  name     data
  <chr>    <list>
1 Amy      <tibble [30 x 3]>
2 Caroline <tibble [33 x 3]>
3 Doris    <tibble [23 x 3]>
4 Belle    <tibble [18 x 3]>
5 Gloria   <tibble [125 x 3]>
6 Anita    <tibble [20 x 3]>
```

Agora temos as colunas “year”, “lat” e “long” aninhadas na coluna de lista chamada “data” para cada observação da coluna “name”.

Assim como vimos anteriormente, podemos acessar as tibbles da coluna dat usando [[ ]]. Por exemplo:

```r
n_storms$data[[1]]
```

```
# A tibble: 30 x 3
    year   lat  long
   <dbl> <dbl> <dbl>
 1  1975  27.5 -79
 2  1975  28.5 -79
 3  1975  29.5 -79
 4  1975  30.5 -79
 5  1975  31.5 -78.8
 6  1975  32.4 -78.7
 7  1975  33.3 -78
 8  1975  34   -77
 9  1975  34.4 -75.8
10  1975  34   -74.8
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

3.7.3 Criando Tibbles com Colunas de Listas

Para criar um objeto tibble com colunas de lista (list-column), você pode utilizar as mesmas funções **tibble**(), **trible**() e **enframe**(), passando uma objeto lista para a coluna.

3.7.3.1 tibble

Use para criar uma tibble com uma coluna de lista salvando uma lista na coluna.

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

```
# A tibble: 3 x 2
    max seq
  <dbl> <list>
1     3 <int [3]>
2     4 <int [4]>
3     5 <int [5]>
```

3.7.3.2 tribble

Use para criar uma tibble com uma coluna de lista por linhas.

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

```
# A tibble: 3 x 2
    max seq
  <dbl> <list>
1     3 <int [3]>
2     4 <int [4]>
3     5 <int [5]>
```

3.7.3.3 enframe

Use para converter listas em um tibble dentro de uma coluna de lista.

```r
lista <- list('3'=1:3, '4'=1:4, '5'=1:5)
enframe(lista, 'max', 'seq')
```

```
# A tibble: 3 x 2
  max   seq
  <chr> <list>
1 3     <int [3]>
2 4     <int [4]>
3 5     <int [5]>
```

> **Nota**
>
> Observe que nossa lista possui "nomes" nos vetores (3, 4 e 5), se isso não for o caso, ele irá nomear as colunas com a sequencia lógica dos vetores (1, 2 e 3).

3.7.3.4 Outras Funções Retornam Coluna de Lista

Algumas funções, como por exemplo, mutate(), transmute() e summarise() do pacote dplyr tem como saída uma colunas de lista caso retornem uma lista.

Por exemplo, se criarmos uma lista com os quartis da variável consumo (mpg) da tatela mtcars agrupada por cilindros (cyl) e utilzarmos a função summarise(), teremos uma coluna de lista contendo os quartis para cada grupo de cilindro.

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

```
# A tibble: 3 x 2
    cyl q
  <dbl> <list>
1     4 <dbl [5]>
2     6 <dbl [5]>
3     8 <dbl [5]>
```

3.7.4 Reformatando dados Aninhados

3.7.4.1 unest

Use para desaninhar os dados. Esta função, faz o inverso da função nest.

Por exemplo, para desaninhar os dados da coluna data criada na tabela n_storms, fazemos:

```
n_storms |>
  unnest(data)
```

```
# A tibble: 11,859 x 4
   name   year   lat  long
   <chr> <dbl> <dbl> <dbl>
 1 Amy    1975  27.5 -79
 2 Amy    1975  28.5 -79
 3 Amy    1975  29.5 -79
 4 Amy    1975  30.5 -79
 5 Amy    1975  31.5 -78.8
 6 Amy    1975  32.4 -78.7
 7 Amy    1975  33.3 -78
 8 Amy    1975  34   -77
 9 Amy    1975  34.4 -75.8
10 Amy    1975  34   -74.8
# ... with 11,849 more rows
# i Use `print(n = ...)` to see more rows
```

3.7.4.2 unest_longer

Use para desaninhar um coluna de lista, tornando cada **elemento** da lista em uma **linha**.

Por exemplo, na tabela starwars, temos uma coluna de lista chamada "films", nesta coluna temos uma lista de filmes que cada personagem participou. Se quisermos desaninhar esta coluna e colocar cada filme em uma linha, faremos:

```
starwars |>
select(name, films) |>
unnest_longer(films)
```

```
# A tibble: 173 x 2
   name           films
   <chr>          <chr>
 1 Luke Skywalker The Empire Strikes Back
 2 Luke Skywalker Revenge of the Sith
 3 Luke Skywalker Return of the Jedi
 4 Luke Skywalker A New Hope
 5 Luke Skywalker The Force Awakens
 6 C-3PO          The Empire Strikes Back
 7 C-3PO          Attack of the Clones
 8 C-3PO          The Phantom Menace
 9 C-3PO          Revenge of the Sith
10 C-3PO          Return of the Jedi
# ... with 163 more rows
# i Use `print(n = ...)` to see more rows
```

3.7.4.3 unest_wider

Use para desaninhar um coluna de lista, tornando cada **elemento** da lista em uma **coluna**.

Por exemplo, na tabela starwars, temos uma coluna de lista chamada "films", nesta coluna temos uma lista de filmes que cada personagem participou. Se quisermos desaninhar esta coluna e colocar cada filme em uma coluna, faremos:

```
starwars |>
  select(name, films) |>
  unnest_wider(films, names_sep = '_')
```

```
# A tibble: 87 x 8
   name               films_1    films_2 films_3 films_4 films_5 films_6 films_7
   <chr>              <chr>      <chr>   <chr>   <chr>   <chr>   <chr>   <chr>
 1 Luke Skywalker     The Empir~ Reveng~ Return~ A New ~ The Fo~ <NA>    <NA>
 2 C-3PO              The Empir~ Attack~ The Ph~ Reveng~ Return~ A New ~ <NA>
 3 R2-D2              The Empir~ Attack~ The Ph~ Reveng~ Return~ A New ~ The Fo~
 4 Darth Vader        The Empir~ Reveng~ Return~ A New ~ <NA>    <NA>    <NA>
 5 Leia Organa        The Empir~ Reveng~ Return~ A New ~ The Fo~ <NA>    <NA>
 6 Owen Lars          Attack of~ Reveng~ A New ~ <NA>    <NA>    <NA>    <NA>
 7 Beru Whitesun lars Attack of~ Reveng~ A New ~ <NA>    <NA>    <NA>    <NA>
 8 R5-D4              A New Hope <NA>    <NA>    <NA>    <NA>    <NA>    <NA>
 9 Biggs Darklighter  A New Hope <NA>    <NA>    <NA>    <NA>    <NA>    <NA>
10 Obi-Wan Kenobi     The Empir~ Attack~ The Ph~ Reveng~ Return~ A New ~ <NA>
```

```
# ... with 77 more rows
# i Use `print(n = ...)` to see more rows
```

3.7.4.4 hoist

Use para selecionar componentes específicos de uma lista e desaninhá-lo em uma nova coluna. É similar ao unnest_wider(), mas desaninha colunas específicas usando a sintaxe do purrr:pluck().

Por exemplo, vamos desaninhar apenas o primeiro e segundo filmes em que o psernagem de starwars particiou e manter os demais anihados:

```
starwars %>%
select(name, films) %>%
hoist(films, "1o_filme" = 1, "2o_filme" = 2)
```

```
# A tibble: 87 x 4
   name               `1o_filme`              `2o_filme`           films
   <chr>              <chr>                   <chr>                <list>
 1 Luke Skywalker     The Empire Strikes Back Revenge of the Sith  <chr [3]>
 2 C-3PO              The Empire Strikes Back Attack of the Clones <chr [4]>
 3 R2-D2              The Empire Strikes Back Attack of the Clones <chr [5]>
 4 Darth Vader        The Empire Strikes Back Revenge of the Sith  <chr [2]>
 5 Leia Organa        The Empire Strikes Back Revenge of the Sith  <chr [3]>
 6 Owen Lars          Attack of the Clones    Revenge of the Sith  <chr [1]>
 7 Beru Whitesun lars Attack of the Clones    Revenge of the Sith  <chr [1]>
 8 R5-D4              A New Hope              <NA>                 <chr [0]>
 9 Biggs Darklighter  A New Hope              <NA>                 <chr [0]>
10 Obi-Wan Kenobi     The Empire Strikes Back Attack of the Clones <chr [4]>
# ... with 77 more rows
# i Use `print(n = ...)` to see more rows
```

3.7.5 Transfromando dados Aninhados

Uma função vetorizada recebe um vetor, transforma cada elemento em paralelo e retorna um vetor de mesmo tamanho que o vetor de entrada. Estas funções sozinhas não trabalham com listas, e consequentemente, não trabalham com colunas de listas.

A função dplyr::rownames() agrupa cada linha da tabela em um grupo diferente e dentro de cada grupo os elementos da coluna de lista aparecem diretamente (acessados por colchetes duplo) e não mais como uma lista e tamanho 1.

Portanto, quando usamos a rownames(), as funções vetorizadas do pacote dplyr poderão ser aplicadas em uma coluna de lista de uma forma vetorizada.

3.7.5.1 Exemplo 1:

Vamos aplicar a função mutate() para criar uma nova coluna de lista contendo as dimensões do tibble presente na coluna "data":

```
n_storms  |>
  rowwise() |>
  mutate ("dim" = list(dim(data)))
```

```
# A tibble: 214 x 3
# Rowwise:
   name     data               dim
   <chr>    <list>             <list>
 1 Amy      <tibble [30 x 3]>  <int [2]>
 2 Caroline <tibble [33 x 3]>  <int [2]>
 3 Doris    <tibble [23 x 3]>  <int [2]>
 4 Belle    <tibble [18 x 3]>  <int [2]>
 5 Gloria   <tibble [125 x 3]> <int [2]>
 6 Anita    <tibble [20 x 3]>  <int [2]>
 7 Clara    <tibble [24 x 3]>  <int [2]>
 8 Evelyn   <tibble [9 x 3]>   <int [2]>
 9 Amelia   <tibble [6 x 3]>   <int [2]>
10 Bess     <tibble [13 x 3]>  <int [2]>
# ... with 204 more rows
# i Use `print(n = ...)` to see more rows
```

3.7.5.2

Neste exemplo, utilzamos a função dim() que retorna as dimensões de um objeto. Como na coluna "data", temos um objeto tibble para cada linha, a função dim irá retornar dois valores (qtd de linha e qtd de colunas).

Isto só funcionou porque agrupamento através da função rowwise() a tabela anterior.

3.7.5.3 Exemplo 2:

Vamos aplicar a função mutate() para criar um coluna "normal" contendo o número de linhas da tibble presente na coluna "data":

```
n_storms  |>
  rowwise() |>
  mutate ("num_linhas" = nrow(data))
```

```
# A tibble: 214 x 3
# Rowwise:
   name     data               num_linhas
   <chr>    <list>                  <int>
 1 Amy      <tibble [30 x 3]>          30
 2 Caroline <tibble [33 x 3]>          33
 3 Doris    <tibble [23 x 3]>          23
 4 Belle    <tibble [18 x 3]>          18
 5 Gloria   <tibble [125 x 3]>        125
 6 Anita    <tibble [20 x 3]>          20
 7 Clara    <tibble [24 x 3]>          24
 8 Evelyn   <tibble [9 x 3]>            9
 9 Amelia   <tibble [6 x 3]>            6
10 Bess     <tibble [13 x 3]>          13
# ... with 204 more rows
# i Use `print(n = ...)` to see more rows
```

3.7.5.4 Exemplo 3:

Vamos aplicar a função mutate() para criar um coluna de lista contendo uma outra lista com a união entre as colunas de listas "vehicles" e "starships" presentes na tabela starwars:

```
starwars |>
  rowwise() |>
  mutate (transporte = list(append(vehicles, starships))) |>
  select(name, transporte)
```

```
# A tibble: 87 x 2
# Rowwise:
   name               transporte
   <chr>              <list>
```

```
 1 Luke Skywalker     <chr [4]>
 2 C-3PO              <chr [0]>
 3 R2-D2              <chr [0]>
 4 Darth Vader        <chr [1]>
 5 Leia Organa        <chr [1]>
 6 Owen Lars          <chr [0]>
 7 Beru Whitesun lars <chr [0]>
 8 R5-D4              <chr [0]>
 9 Biggs Darklighter  <chr [1]>
10 Obi-Wan Kenobi     <chr [6]>
# ... with 77 more rows
# i Use `print(n = ...)` to see more rows
```

> **Dica**
>
> Se quisermos pegar os transportes criados e colocá-los em linhas, podemos usar a função unest.

```
starwars |>
  rowwise() |>
  mutate (transporte = list(append(vehicles, starships))) |>
  select(name, transporte) |>
  unnest(transporte)
```

```
# A tibble: 44 x 2
   name               transporte
   <chr>              <chr>
 1 Luke Skywalker     Snowspeeder
 2 Luke Skywalker     Imperial Speeder Bike
 3 Luke Skywalker     X-wing
 4 Luke Skywalker     Imperial shuttle
 5 Darth Vader        TIE Advanced x1
 6 Leia Organa        Imperial Speeder Bike
 7 Biggs Darklighter X-wing
 8 Obi-Wan Kenobi     Tribubble bongo
 9 Obi-Wan Kenobi     Jedi starfighter
10 Obi-Wan Kenobi     Trade Federation cruiser
# ... with 34 more rows
# i Use `print(n = ...)` to see more rows
```

3.7.5.5 Exemplo 4:

Vamos aplicar a função mutate() para criar um coluna de lista contendo uma o tamanho das listas "vehicles" e "starships" presentes na tabela starwars, e depois iremos desaninhar esta lista, obtendo assim quantos transportes cada personagem possui:

```
starwars |>
  rowwise() |>
  mutate (transporte = list(length(c(vehicles, starships)))) |>
  select(name, transporte) |>
  unnest(transporte)
```

```
# A tibble: 87 x 2
   name               transporte
   <chr>                   <int>
 1 Luke Skywalker              4
 2 C-3PO                       0
 3 R2-D2                       0
 4 Darth Vader                 1
 5 Leia Organa                 1
 6 Owen Lars                   0
 7 Beru Whitesun lars          0
 8 R5-D4                       0
 9 Biggs Darklighter           1
10 Obi-Wan Kenobi              6
# ... with 77 more rows
# i Use `print(n = ...)` to see more rows
```

> Dica
>
> Veja o pacote **purrr** para outras funções que manipulam listas.

Observe pelos exemplos anteriores que quando temos uma função que retorna uma lista, devemos usar a função list() para criar uma coluna de lista. Se a função retorna um valor (ex um inteiro), a coluna criada será um coluna "normal", neste caso um coluna de inteiros.

4 Transformação de Dados com DPLYR

4.1 Introdução

A seguir temos vários exemplos de transformação de dados utilizando o pacote DYPLR do R. Para saber mais sobre este pacote, acesse:

https://cran.r-project.org/package=dplyr.

> **Aviso**
>
> Para melhor utilizar este material, é importante que você tenha uma introdução à linguagem R e saiba carregar pacotes (packages) no R. Para mais informações acesse: https://education.rstudio.com/learn/beginner/.

Para os exemplos à seguir, iremos usar os seguintes pacotes:

- **tidyverse**
- **gt**

```
library(tidyverse)
library (gt)
```

4.1.1 Exemplos da Folha de Referência

A maioria dos exemplos, visam ajudar na interpretação dos exemplos e funções encontradas na **Folha de Referência** do dyplr disponível no site do RStudio.

Para a maioria dos exemplos utilizaremos as bases de dados **mtcars** e **starwars** provenientes do pacote **datasets e dplyr**.

> **Nota**
>
> *Em geral, ao final de cada comando, você poderá ver uma chamada à função* ***gt()****. Isto é apenas para a formatação da tabela de saída e não é necessário para que você entenda os comandos precedentes. Em alguns casos, onde o volume de dados de saída pode ser*

*extenso, usamos também a função **head()** para mostrar apenas as linhas iniciais. Quando o exemplo possui muitas colunas de saída, eventualmente utilizamos a função **select()** para selecionar apenas algumas colunas.*

MTCARS: Dados de consumo de combustível, performance e design de 32 automóveis (*1974 Motor Trend US magazine*)

```
mtcars |>
  head ()
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

STARWARS: Dados dos personagens de STAR WARS

```
starwars |>
  select(1:8) |>
  head()
```

```
# A tibble: 6 x 8
  name           height  mass hair_color  skin_color  eye_color birth_year sex
  <chr>           <int> <dbl> <chr>       <chr>       <chr>          <dbl> <chr>
1 Luke Skywalker    172    77 blond       fair        blue            19   male
2 C-3PO             167    75 <NA>        gold        yellow         112   none
3 R2-D2              96    32 <NA>        white, blue red             33   none
4 Darth Vader       202   136 none        white       yellow          41.9 male
5 Leia Organa       150    49 brown       light       brown           19   fema~
6 Owen Lars         178   120 brown, grey light       blue            52   male
```

> **Nota**
>
> *O termo data-frame descrito ao longo deste texto, é utilizado de forma livre para objetos do tipo data.frame, tibble, entre outros. Pense como se fosse uma tabela de um banco de dados e/ou uma planilha do MS Excel, contendo linhas e colunas. Apesar de não ser rigorosamente igual à uma tabela, muitas vezes usaremos estes termos de forma intercambiável para facilitar o entendimento de iniciantes.*

4.1.2 Dados Organizados e Canalização

Dados organizados (*tidy*) são estruturados onde:

Cada **variável** está em sua própria **coluna** e cada **observação** está em sua própria **linha**.

Canalização (*Piping*) é uma forma de sequenciar as funções, facilitando a leitura de várias funções em conjunto. O símbolo |> ou %>% podem ser utilizados para esta finalidade.

Exemplo: x |> f(y), é o mesmo que f(x,y)

4.2 Resumindo Observações

Funções de resumo recebem vetores como entrada e retornam um único valor.

4.2.0.1 summarise

```
summarise (mtcars, media=mean(mpg))
```

```
    media
1 20.09062
```

Cria um novo data-frame. Ele gera uma linha para cada combinação de grupos de variáveis. Se não houver grupos, a saída será uma única linha resumindo todas as observações da entrada.

No exemplo acima, não especificamos nenhum grupo (mais na seção Agrupando Observações), por isso, ele retorna apenas uma linha resumindo todas as observações de mtcars. Como também especificamos a função de **mean()**, o valor retornado será a média da coluna **mpg** para todas as observações do data frame em uma única variável (coluna) chamada **média**.

Outra format de se escrever o comando acima, utilizando a canalização ("pinping") descrita acima, seria:

```
mtcars |>
  summarise(média = mean(mpg))
```

Ver a seção de Funções de Resumo para mais detalhes.

4.2.0.2 count

Conta valores únicos de uma ou mais variáveis.

```
mtcars |>
  count (cyl)
```

```
  cyl  n
1   4 11
2   6  7
3   8 14
```

No exemplo acima, contamos os valores únicos da variável **cyl** e mostramos na variável **n**. Um equivalente seria utilizar o summarise com o grupo criado na variável cyl: *mtcars |> group_by(cyl) |> summarise(n = n()).*

Mais sobre a função group_by na seção Agrupando Observações.

4.3 Agrupando Observações

4.3.0.1 group_by

Use **group_by** (.data, ..., .add = FALSE, .drop = TRUE) para criar uma cópia da tabela agrupada por colunas. As funções do dplyr irão manipular cada grupo separadamente e combinar os resultados.

> **!** Importante
>
> Se compararmos o resultado de uma tabela antes e depois do agrupamento (group_by), apenas uma informação sobre o grupo será visível.
> Isto significa que o agrupamento só afetará o resultado de saída se utilizado em conjunto com funções que entendam esta mudança de contexto, como as funções do dyplr.

Por exemplo, **ANTES** de agruparmos pela variável **cyl** (group_by(cyl)) temos:

```
mtcars |>
  as_tibble()
```

```
# A tibble: 32 x 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
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# ... with 22 more rows
# i Use `print(n = ...)` to see more rows
```

DEPOIS do group_by:

```
mtcars |>
  group_by(cyl)
```

```
# A tibble: 32 x 11
# Groups:   cyl [3]
     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
```

```
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# ... with 22 more rows
# i Use `print(n = ...)` to see more rows
```

Veja que, exceto pela informação :

'# Groups: cyl [3]

Todo o restante da saída é o mesmo. Por isso, é importante que utilizemos o contexto do grupo, junto com outra função.

Exemplo: Se quiser criar uma tabela com a função **summarise()** com apenas o grupo de cilindradas, podemos usar:

```
mtcars |>
  group_by(cyl) |>
  summarise()
```

```
# A tibble: 3 x 1
    cyl
  <dbl>
1     4
2     6
3     8
```

Supondo que quiséssemos agora, saber o número de carros, agrupados pela variável cyl, podemos utilizar a função summarise, após o agrupamento, criando uma coluna através da função n().

```
mtcars |>
  group_by(cyl) |>
  summarise(num_automoveis = n())
```

```
# A tibble: 3 x 2
    cyl num_automoveis
```

```
  <dbl>          <int>
1     4             11
2     6              7
3     8             14
```

Em outro exemplo, podemos extrair o consumo médio (miles/gallon) dos veículos agrupado pelo número de cilindros destes veículos.

```
mtcars |>
  group_by(cyl) |>
  summarise(consumo_medio = mean(mpg))
```

```
# A tibble: 3 x 2
    cyl consumo_medio
  <dbl>         <dbl>
1     4          26.7
2     6          19.7
3     8          15.1
```

O agrupamento pode ser feito por mais de uma variável também. Por exemplo, se quisermos obter o consumo médio (miles/gallon) dos veículos agrupado pelo número de cilindros destes veículos e também por número de marchas, podemos ter:

> **Nota**
>
> A saída da função summarise, dependendo do caso, irá ser agrupada também automaticamente, se quiser que isto não aconteça, utiliza a opção **.groups="drop"**.

```
mtcars |>
  group_by(cyl, gear) |>
  summarise(consumo_médio = mean(mpg), .groups = "drop")
```

```
# A tibble: 8 x 3
    cyl  gear consumo_médio
  <dbl> <dbl>         <dbl>
1     4     3          21.5
2     4     4          26.9
3     4     5          28.2
4     6     3          19.8
5     6     4          19.8
```

```
6     6     5          19.7
7     8     3          15.0
8     8     5          15.4
```

A função **un_group()** remove os grupos definidos em uma tabela. É uma boa prática, remover os grupos após efetuar uma sumarização, por exemplo, afim de evitar resultados indesejáveis em sumarizações futuras. Por exemplo:

```
mtcars |>
  group_by(cyl, gear) |>
  summarise(numero_marchas = n()) |>
  ungroup()
```

```
# A tibble: 8 x 3
    cyl  gear numero_marchas
  <dbl> <dbl>          <int>
1     4     3              1
2     4     4              8
3     4     5              2
4     6     3              2
5     6     4              4
6     6     5              1
7     8     3             12
8     8     5              2
```

No caso acima, se não tivessemos utilizado o desagrupamento (**un_group**), o resultado ainda teria os grupos demarcados.

4.3.0.2 rowwise

Use rowwise(.data, …) para agrupar dados em linhas individuais. Funções do dyplr irão computar os resultados para cada linha.

No exemplo abaixo, a tabela de dados startwars, possui uma variável (films) que é o tipo lista, ou seja, a coluna contém uma lista de filmes para cada personagem (observação).

Supondo que quiséssemos saber em quantos filmes cada personagem aparece:

```
starwars |>
  select(name, films) |>
  rowwise() |>
```

```
    mutate(quantos_filmes = length(films)) |>
    head ()
```

```
# A tibble: 6 x 3
# Rowwise:
  name           films     quantos_filmes
  <chr>          <list>             <int>
1 Luke Skywalker <chr [5]>              5
2 C-3PO          <chr [6]>              6
3 R2-D2          <chr [7]>              7
4 Darth Vader    <chr [4]>              4
5 Leia Organa    <chr [5]>              5
6 Owen Lars      <chr [3]>              3
```

Em geral, utilizamos a função **rowwise** quando não temos uma função vetorizada, isto é, que não retorna um vetor e precisamos aplicá-la em cada linha da tabela.

> **Nota**
>
> *Aqui utilizamos também a função **head ()** que retorna apenas as primeiras linhas de uma tablea e não todo seu conteúdo.*

Veja a seção de Funções Vetorizadaspara maiores informações.

4.4 Manipulando Observações

4.4.1 Extração de Observações

O dplyr possui uma série de funções que nos ajudam a extrair observações (linhas) de uma tabela, dentre estas, temos:

4.4.1.1 filter

Extrai linhas de uma tablea que satisfazem o critério lógico.

filter(.data, ..., .preserve = FALSE)

Exemplo: Para extrair apenas os veículos que possuem consumo (miles/galon) acima de 20, podemos usar:

```r
filter(mtcars, mpg > 20)
```

```
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Podemos utilizar operadores lógicos para ajustar os critérios do filtro, como por exemplo, os operadores abaixo:

==, >, >=, &, |, !, xor(), is.na(), between(), near(), entre outros

Por exemplo, para filtrar os veículos com consumo acima de 20 **E** com apenas 3 marchas, temos:

```r
mtcars |>
  filter(mpg > 20 & gear == 3)
```

```
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
```

4.4.1.2 distinct

Remove linhas com valores duplicados, retornando apenas os valores únicos da variável (coluna).

distinct(.data, ..., .keep_all = FALSE) .

Por exemplo, se precisarmos saber quais os valores da variável gear (marcha), podemos utilizar:

```r
mtcars |>
  distinct(gear)
```

```
              gear
Mazda RX4        4
Hornet 4 Drive   3
Porsche 914-2    5
```

> **Nota**
>
> Se utilizar o código acima, verá que o R possui um "nome" para as linhas. É importante ressaltar que este nome NÃO é uma variável da tabela, ou seja, não temos uma coluna com o "nome" do veículo, é por isso que você vê nomes de veículos, mesmo pedindo os valores únicos das marchas. Para maiores informações sobre isso, veja a seção Nome de Linhas.

4.4.1.3 slice

Seleciona linhas pelas suas respectivas posições nda tabela.

slice(.data, ..., .preserve = FALSE)

Por exemplo, para selecionarmos apenas da linha 10 até a linha 15 da tabela usamos:

```r
mtcars |>
  slice(10:15)
```

```
                  mpg cyl  disp  hp drat   wt  qsec vs am gear carb
Merc 280         19.2   6 167.6 123 3.92 3.44 18.30  1  0    4    4
Merc 280C        17.8   6 167.6 123 3.92 3.44 18.90  1  0    4    4
Merc 450SE       16.4   8 275.8 180 3.07 4.07 17.40  0  0    3    3
Merc 450SL       17.3   8 275.8 180 3.07 3.73 17.60  0  0    3    3
Merc 450SLC      15.2   8 275.8 180 3.07 3.78 18.00  0  0    3    3
Cadillac Fleetwood 10.4 8 472.0 205 2.93 5.25 17.98  0  0    3    4
```

Veja que em alguns casos, podemos utilizar um equivalente filtro para obter o mesmo resultado:

```
mtcars |>
  filter(between(row_number(), 10, 15))
```

```
                  mpg cyl  disp  hp drat   wt  qsec vs am gear carb
Merc 280         19.2   6 167.6 123 3.92 3.44 18.30  1  0    4    4
Merc 280C        17.8   6 167.6 123 3.92 3.44 18.90  1  0    4    4
Merc 450SE       16.4   8 275.8 180 3.07 4.07 17.40  0  0    3    3
Merc 450SL       17.3   8 275.8 180 3.07 3.73 17.60  0  0    3    3
Merc 450SLC      15.2   8 275.8 180 3.07 3.78 18.00  0  0    3    3
Cadillac Fleetwood 10.4   8 472.0 205 2.93 5.25 17.98  0  0    3    4
```

4.4.1.4 slice_sample

Para randomicamente selecionar linhas da tabela. Use **n** para selecionar o número de linhas ou **prop** para selecionar um percentual das linhas.

slice_sample(.data, ..., n, prop, weight_by = NULL, replace = FALSE) .

Para selecionar **5** linhas randomicas da tabela usamos:

```
mtcars |>
  slice_sample(n = 5, replace = TRUE)
```

```
              mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Ferrari Dino  19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Fiat X1-9     27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Toyota Corona 21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Fiat 128...4  32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Fiat 128...5  32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
```

Para selecionar **25%** do total de linhas da tabela de forma dandomica usamos:

```
mtcars |>
  slice_sample(prop = 0.25, replace = TRUE)
```

```
                  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Merc 450SE       16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Pontiac Firebird 19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Merc 450SL       17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
```

```
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

4.4.1.5 slice_min

Seleciona linhas com valores minímo (slice_min) ou máximo (slice_max) de uma variável.

slice_min(.data, order_by, ..., n, prop, with_ties = TRUE) and slice_max() .

Por exemplo, com base no menor valor da variável de consumo do veículo (mpg), retorne 25% da tabela.

```
mtcars |>
  slice_min(mpg, prop = 0.25)
```

```
                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
```

Outro exemplo, poderia ser que você precise retornar os 5 veículos de maior consumo:

```
mtcars |>
  slice_max(mpg, n = 5)
```

```
                mpg cyl disp  hp drat    wt  qsec vs am gear carb
Toyota Corolla 33.9   4 71.1  65 4.22 1.835 19.90  1  1    4    1
Fiat 128       32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
Lotus Europa   30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2
Fiat X1-9      27.3   4 79.0  66 4.08 1.935 18.90  1  1    4    1
```

4.4.1.6 slice_head

Seleciona as primeiras (slice_head) or últimas (slide_tail) linhas de uma tabela.

slice_head(.data, ..., n, prop) and slice_tail() .

Por exemplo, vamos selecionar as 5 primeiras linhas de mtcars:

> **Nota**
>
> *Apenas para exemplificar, no código abaixo, deixamos a função gt "mostrar" os nomes das linhas em sua saída. Para maiores informações sobre isso, veja a seção Nome de Linhas.*

```
mtcars |>
  slice_head(n = 5) |>
  gt(rownames_to_stub = TRUE)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2

4.4.2 Arranjar Observações

4.4.2.1 arrange

A função arrange ordena linhas por valores de uma coluna(s) (menor para maior), use com a função **desc()** para ordenar de maior para menor.

arrange(.data, ..., .by_group = FALSE) arrange(mtcars, mpg) ou arrange(mtcars, desc(mpg))

No exemplo abaixo, vamos ordenar a variável mpg, de forma a mostrar primeiro os veículos com menor consumo de combustível até o de maior consumo:

```
mtcars |>
  arrange(mpg) |>
  gt(rownames_to_stub = TRUE)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1

4.4.3 Adicionar Observações

Algumas vezes precisamos adicionar observações (linhas) em uma tabela já existente, neste caso podemos utilizar a função add_row().

4.4.3.1 add_row

add_row(cars, speed = 1, dist = 1).

Neste caso, iremos adicionar uma nova linha na tables cars (não mtcars), que possui apenas duas variáveis (speed e dist).

```
cars |>
  add_row(speed = 1, dist = 1) |>
  tail()
```

```
   speed dist
46    24   70
47    24   92
48    24   93
49    24  120
50    25   85
51     1    1
```

4.5 Manipulando Variáveis

4.5.1 Extração de Variáveis

O dplyr possui uma série de funções que nos ajudam a obter um conjunto de variáveis (colunas) como um novo vetor ou nova tabela:

4.5.1.1 pull

Extrai valores da coluna como um vetor, por nome ou índice.

pull(.data, var = -1, name = NULL, ...)

```
mtcars |>
  pull (wt)
```

```
 [1] 2.620 2.875 2.320 3.215 3.440 3.460 3.570 3.190 3.150 3.440 3.440 4.070
[13] 3.730 3.780 5.250 5.424 5.345 2.200 1.615 1.835 2.465 3.520 3.435 3.840
[25] 3.845 1.935 2.140 1.513 3.170 2.770 3.570 2.780
```

No exemplo acima, a coluna peso (wt), é extraída da tabela e um obeto do tipo vetor é retornado na saída.

Podemos extrair um vator de uma colunas também utilizando o número da coluna. Se utilizarmos valores negativos, podemos extrair um vetor das colunas contando a partir do lado direto.

Por exemplo, se quisermos extratir um vetor da penúltima coluna de uma tabela, podemos usar:

```
mtcars |>
  pull (-2)
```

```
 [1] 4 4 4 3 3 3 3 4 4 4 4 3 3 3 3 3 3 4 4 4 3 3 3 3 3 4 5 5 5 5 5 4
```

4.5.1.2 select

Extrai valores de uma ou mais variáveis (colunas) e retorna uma nova tabela, por nome ou índice.

select(.data, …)

Por exemplo, podemos obter uma nova tabela contendo apenas as variáveis mpg e wt usando:

```
mtcars |>
  select (mpg, wt)
```

```
                   mpg    wt
Mazda RX4         21.0 2.620
Mazda RX4 Wag     21.0 2.875
Datsun 710        22.8 2.320
Hornet 4 Drive    21.4 3.215
Hornet Sportabout 18.7 3.440
Valiant           18.1 3.460
Duster 360        14.3 3.570
Merc 240D         24.4 3.190
Merc 230          22.8 3.150
Merc 280          19.2 3.440
Merc 280C         17.8 3.440
Merc 450SE        16.4 4.070
Merc 450SL        17.3 3.730
Merc 450SLC       15.2 3.780
```

```
Cadillac Fleetwood  10.4 5.250
Lincoln Continental 10.4 5.424
Chrysler Imperial   14.7 5.345
Fiat 128            32.4 2.200
Honda Civic         30.4 1.615
Toyota Corolla      33.9 1.835
Toyota Corona       21.5 2.465
Dodge Challenger    15.5 3.520
AMC Javelin         15.2 3.435
Camaro Z28          13.3 3.840
Pontiac Firebird    19.2 3.845
Fiat X1-9           27.3 1.935
Porsche 914-2       26.0 2.140
Lotus Europa        30.4 1.513
Ford Pantera L      15.8 3.170
Ferrari Dino        19.7 2.770
Maserati Bora       15.0 3.570
Volvo 142E          21.4 2.780
```

A função select possui um série de opções que tornam a seleção das counas mais fáceis. A seguir temos algumas delas. Consulte a ajuda do select usando "?select" para obter a lista completa.

- : para selecionar um range consecutivo de colunas
- ! para pegar o complemento de uma conjunto de colunas
- & e | para selecionar a intersecção ou união (E OU) de um conjunto de colunas
- c() para combinar seleções

Além disso, possui também algumas funções que ajudam na seleção como:

- eveything(), last_col(), starts_with(), ends_with(), contains(), matches(), num_range()

Por exemplo, se quisermos selecionar toda a base mtcars, exceto as colunas wt e mpg, podemos usar:

```
mtcars |>
  select(!c(mpg, wt))
```

```
                  cyl  disp  hp drat  qsec vs am gear carb
Mazda RX4           6 160.0 110 3.90 16.46  0  1    4    4
Mazda RX4 Wag       6 160.0 110 3.90 17.02  0  1    4    4
```

```
Datsun 710            4 108.0  93 3.85 18.61  1  1    4    1
Hornet 4 Drive        6 258.0 110 3.08 19.44  1  0    3    1
Hornet Sportabout     8 360.0 175 3.15 17.02  0  0    3    2
Valiant               6 225.0 105 2.76 20.22  1  0    3    1
Duster 360            8 360.0 245 3.21 15.84  0  0    3    4
Merc 240D             4 146.7  62 3.69 20.00  1  0    4    2
Merc 230              4 140.8  95 3.92 22.90  1  0    4    2
Merc 280              6 167.6 123 3.92 18.30  1  0    4    4
Merc 280C             6 167.6 123 3.92 18.90  1  0    4    4
Merc 450SE            8 275.8 180 3.07 17.40  0  0    3    3
Merc 450SL            8 275.8 180 3.07 17.60  0  0    3    3
Merc 450SLC           8 275.8 180 3.07 18.00  0  0    3    3
Cadillac Fleetwood    8 472.0 205 2.93 17.98  0  0    3    4
Lincoln Continental   8 460.0 215 3.00 17.82  0  0    3    4
Chrysler Imperial     8 440.0 230 3.23 17.42  0  0    3    4
Fiat 128              4  78.7  66 4.08 19.47  1  1    4    1
Honda Civic           4  75.7  52 4.93 18.52  1  1    4    2
Toyota Corolla        4  71.1  65 4.22 19.90  1  1    4    1
Toyota Corona         4 120.1  97 3.70 20.01  1  0    3    1
Dodge Challenger      8 318.0 150 2.76 16.87  0  0    3    2
AMC Javelin           8 304.0 150 3.15 17.30  0  0    3    2
Camaro Z28            8 350.0 245 3.73 15.41  0  0    3    4
Pontiac Firebird      8 400.0 175 3.08 17.05  0  0    3    2
Fiat X1-9             4  79.0  66 4.08 18.90  1  1    4    1
Porsche 914-2         4 120.3  91 4.43 16.70  0  1    5    2
Lotus Europa          4  95.1 113 3.77 16.90  1  1    5    2
Ford Pantera L        8 351.0 264 4.22 14.50  0  1    5    4
Ferrari Dino          6 145.0 175 3.62 15.50  0  1    5    6
Maserati Bora         8 301.0 335 3.54 14.60  0  1    5    8
Volvo 142E            4 121.0 109 4.11 18.60  1  1    4    2
```

Se quisermos selecionar apenas as 4 primeiras colunas e também a coluna wt, podemos usar:

```
mtcars |>
  select ((1:4) | wt)
```

```
                   mpg cyl  disp  hp    wt
Mazda RX4         21.0   6 160.0 110 2.620
Mazda RX4 Wag     21.0   6 160.0 110 2.875
Datsun 710        22.8   4 108.0  93 2.320
Hornet 4 Drive    21.4   6 258.0 110 3.215
```

```
Hornet Sportabout   18.7   8 360.0 175 3.440
Valiant             18.1   6 225.0 105 3.460
Duster 360          14.3   8 360.0 245 3.570
Merc 240D           24.4   4 146.7  62 3.190
Merc 230            22.8   4 140.8  95 3.150
Merc 280            19.2   6 167.6 123 3.440
Merc 280C           17.8   6 167.6 123 3.440
Merc 450SE          16.4   8 275.8 180 4.070
Merc 450SL          17.3   8 275.8 180 3.730
Merc 450SLC         15.2   8 275.8 180 3.780
Cadillac Fleetwood  10.4   8 472.0 205 5.250
Lincoln Continental 10.4   8 460.0 215 5.424
Chrysler Imperial   14.7   8 440.0 230 5.345
Fiat 128            32.4   4  78.7  66 2.200
Honda Civic         30.4   4  75.7  52 1.615
Toyota Corolla      33.9   4  71.1  65 1.835
Toyota Corona       21.5   4 120.1  97 2.465
Dodge Challenger    15.5   8 318.0 150 3.520
AMC Javelin         15.2   8 304.0 150 3.435
Camaro Z28          13.3   8 350.0 245 3.840
Pontiac Firebird    19.2   8 400.0 175 3.845
Fiat X1-9           27.3   4  79.0  66 1.935
Porsche 914-2       26.0   4 120.3  91 2.140
Lotus Europa        30.4   4  95.1 113 1.513
Ford Pantera L      15.8   8 351.0 264 3.170
Ferrari Dino        19.7   6 145.0 175 2.770
Maserati Bora       15.0   8 301.0 335 3.570
Volvo 142E          21.4   4 121.0 109 2.780
```

4.5.1.3 relocate

Move colunas para uma nova posição e retonar uma tabela com esta nova order de colunas.

relocate(.data, ..., .before = NULL, .after = NULL)

```
mtcars |>
  relocate (mpg, cyl, .after = last_col())
```

```
                  disp  hp drat    wt  qsec vs am gear carb  mpg cyl
Mazda RX4          160.0 110 3.90 2.620 16.46  0  1    4    4 21.0   6
Mazda RX4 Wag      160.0 110 3.90 2.875 17.02  0  1    4    4 21.0   6
```

```
Datsun 710          108.0  93 3.85 2.320 18.61  1  1    4    1 22.8   4
Hornet 4 Drive      258.0 110 3.08 3.215 19.44  1  0    3    1 21.4   6
Hornet Sportabout   360.0 175 3.15 3.440 17.02  0  0    3    2 18.7   8
Valiant             225.0 105 2.76 3.460 20.22  1  0    3    1 18.1   6
Duster 360          360.0 245 3.21 3.570 15.84  0  0    3    4 14.3   8
Merc 240D           146.7  62 3.69 3.190 20.00  1  0    4    2 24.4   4
Merc 230            140.8  95 3.92 3.150 22.90  1  0    4    2 22.8   4
Merc 280            167.6 123 3.92 3.440 18.30  1  0    4    4 19.2   6
Merc 280C           167.6 123 3.92 3.440 18.90  1  0    4    4 17.8   6
Merc 450SE          275.8 180 3.07 4.070 17.40  0  0    3    3 16.4   8
Merc 450SL          275.8 180 3.07 3.730 17.60  0  0    3    3 17.3   8
Merc 450SLC         275.8 180 3.07 3.780 18.00  0  0    3    3 15.2   8
Cadillac Fleetwood  472.0 205 2.93 5.250 17.98  0  0    3    4 10.4   8
Lincoln Continental 460.0 215 3.00 5.424 17.82  0  0    3    4 10.4   8
Chrysler Imperial   440.0 230 3.23 5.345 17.42  0  0    3    4 14.7   8
Fiat 128             78.7  66 4.08 2.200 19.47  1  1    4    1 32.4   4
Honda Civic          75.7  52 4.93 1.615 18.52  1  1    4    2 30.4   4
Toyota Corolla       71.1  65 4.22 1.835 19.90  1  1    4    1 33.9   4
Toyota Corona       120.1  97 3.70 2.465 20.01  1  0    3    1 21.5   4
Dodge Challenger    318.0 150 2.76 3.520 16.87  0  0    3    2 15.5   8
AMC Javelin         304.0 150 3.15 3.435 17.30  0  0    3    2 15.2   8
Camaro Z28          350.0 245 3.73 3.840 15.41  0  0    3    4 13.3   8
Pontiac Firebird    400.0 175 3.08 3.845 17.05  0  0    3    2 19.2   8
Fiat X1-9            79.0  66 4.08 1.935 18.90  1  1    4    1 27.3   4
Porsche 914-2       120.3  91 4.43 2.140 16.70  0  1    5    2 26.0   4
Lotus Europa         95.1 113 3.77 1.513 16.90  1  1    5    2 30.4   4
Ford Pantera L      351.0 264 4.22 3.170 14.50  0  1    5    4 15.8   8
Ferrari Dino        145.0 175 3.62 2.770 15.50  0  1    5    6 19.7   6
Maserati Bora       301.0 335 3.54 3.570 14.60  0  1    5    8 15.0   8
Volvo 142E          121.0 109 4.11 2.780 18.60  1  1    4    2 21.4   4
```

No exemplo acima, escolhermos mover as colunas mpg e cyl para depois (à direita) da última coluna (last_col()).

4.5.2 Manipular Várias Variáveis de Uma Vez

Em algumas situações, desejamos manipular várias variáveis (colunas) de uma só vez ou invés de cada coluna de cada vez. Para estes casos ,podemos usar as funções across e c_across.

4.5.2.1 across

Resume ou alterar múltiplas colunas da mesma maneira. across(.cols, .funs, ..., .names = NULL)

```
mtcars |>
  summarise(across(everything(), mean))
```

```
       mpg    cyl     disp       hp     drat      wt     qsec     vs      am
1 20.09062 6.1875 230.7219 146.6875 3.596563 3.21725 17.84875 0.4375 0.40625
    gear   carb
1 3.6875 2.8125
```

No exemplo acima, varremos todas (everything) as colunas da tabela e resumimos aplicando a função de média (mean) nestas colunas.

No exemplo a seguir, iremos "varrer" as colunas 5 até 7 e arredondar seus valores com apenas um digito usando a função round().

```
mtcars |>
  mutate(across(5:7, ~ round(.x, digits = 1) ))
```

```
                   mpg cyl  disp  hp drat  wt qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110  3.9 2.6 16.5  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110  3.9 2.9 17.0  0  1    4    4
Datsun 710        22.8   4 108.0  93  3.9 2.3 18.6  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110  3.1 3.2 19.4  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175  3.1 3.4 17.0  0  0    3    2
Valiant           18.1   6 225.0 105  2.8 3.5 20.2  1  0    3    1
Duster 360        14.3   8 360.0 245  3.2 3.6 15.8  0  0    3    4
Merc 240D         24.4   4 146.7  62  3.7 3.2 20.0  1  0    4    2
Merc 230          22.8   4 140.8  95  3.9 3.1 22.9  1  0    4    2
Merc 280          19.2   6 167.6 123  3.9 3.4 18.3  1  0    4    4
Merc 280C         17.8   6 167.6 123  3.9 3.4 18.9  1  0    4    4
Merc 450SE        16.4   8 275.8 180  3.1 4.1 17.4  0  0    3    3
Merc 450SL        17.3   8 275.8 180  3.1 3.7 17.6  0  0    3    3
Merc 450SLC       15.2   8 275.8 180  3.1 3.8 18.0  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205  2.9 5.2 18.0  0  0    3    4
Lincoln Continental 10.4   8 460.0 215  3.0 5.4 17.8  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230  3.2 5.3 17.4  0  0    3    4
Fiat 128          32.4   4  78.7  66  4.1 2.2 19.5  1  1    4    1
```

```
Honda Civic         30.4   4  75.7  52  4.9 1.6 18.5  1  1    4    2
Toyota Corolla      33.9   4  71.1  65  4.2 1.8 19.9  1  1    4    1
Toyota Corona       21.5   4 120.1  97  3.7 2.5 20.0  1  0    3    1
Dodge Challenger    15.5   8 318.0 150  2.8 3.5 16.9  0  0    3    2
AMC Javelin         15.2   8 304.0 150  3.1 3.4 17.3  0  0    3    2
Camaro Z28          13.3   8 350.0 245  3.7 3.8 15.4  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175  3.1 3.8 17.0  0  0    3    2
Fiat X1-9           27.3   4  79.0  66  4.1 1.9 18.9  1  1    4    1
Porsche 914-2       26.0   4 120.3  91  4.4 2.1 16.7  0  1    5    2
Lotus Europa        30.4   4  95.1 113  3.8 1.5 16.9  1  1    5    2
Ford Pantera L      15.8   8 351.0 264  4.2 3.2 14.5  0  1    5    4
Ferrari Dino        19.7   6 145.0 175  3.6 2.8 15.5  0  1    5    6
Maserati Bora       15.0   8 301.0 335  3.5 3.6 14.6  0  1    5    8
Volvo 142E          21.4   4 121.0 109  4.1 2.8 18.6  1  1    4    2
```

4.5.2.2 c_across

Computa através das colunas os dados linha a linha. Em geral, esta função é utilizado em conjunto com a função rowwise().

c_across(.cols)

```
mtcars |>
  rowwise() |>
  transmute (total = sum(c_across(4:6)))
```

```
# A tibble: 32 x 1
# Rowwise:
   total
   <dbl>
 1 117.
 2 117.
 3  99.2
 4 116.
 5 182.
 6 111.
 7 252.
 8  68.9
 9 102.
10 130.
# ... with 22 more rows
# i Use `print(n = ...)` to see more rows
```

No exemplo acima, “varremos” linha a linha da tabela e depois fazemos a soma da coluna 4 até a coluna 6.

4.5.3 Criando novas variáveis

O dyplr possui a habilidade de criar novas variáveis (colunas) ou alterar variáveis já existentes. Estes comandos, aplicam funções que são de um tipo especial chamadas “funções vetorizadas. Elas recebem vetores como entradas e retornam vetores do mesmo tamanho como. Para maiores detalhes veja a seção Funções Vetorizadas.

4.5.3.1 mutate

Altera ou cria uma nova variável.

mutate(.data, ..., .keep = “all”, .before = NULL, .after = NULL).

Por exemplo, se quisermos utilizar a variável **mpg** (milhas por galão) e criar uma nova variável chamada **gpm** (galões por milha), usamos:

```
mtcars |>
  mutate (gpm = 1 / mpg)
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128          32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
```

```
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
                           gpm
Mazda RX4           0.04761905
Mazda RX4 Wag       0.04761905
Datsun 710          0.04385965
Hornet 4 Drive      0.04672897
Hornet Sportabout   0.05347594
Valiant             0.05524862
Duster 360          0.06993007
Merc 240D           0.04098361
Merc 230            0.04385965
Merc 280            0.05208333
Merc 280C           0.05617978
Merc 450SE          0.06097561
Merc 450SL          0.05780347
Merc 450SLC         0.06578947
Cadillac Fleetwood  0.09615385
Lincoln Continental 0.09615385
Chrysler Imperial   0.06802721
Fiat 128            0.03086420
Honda Civic         0.03289474
Toyota Corolla      0.02949853
Toyota Corona       0.04651163
Dodge Challenger    0.06451613
AMC Javelin         0.06578947
Camaro Z28          0.07518797
Pontiac Firebird    0.05208333
Fiat X1-9           0.03663004
Porsche 914-2       0.03846154
Lotus Europa        0.03289474
```

```
Ford Pantera L      0.06329114
Ferrari Dino        0.05076142
Maserati Bora       0.06666667
Volvo 142E          0.04672897
```

> **Cuidado**
>
> É importante observar que a função mutate() considera o agrupamento da tabela, caso houver. Em casos de funções de agregação e ranqueamento (ex. média, ranque, etc), os valores de mutate serão considerados a partir do agrupamento.

Por exemplo, para criar uma coluna que mostra a diferença entre o consumo do veículo e o consumo médio de todos os veículos, podemos usar:

```
mtcars |>
  mutate (diferenca = mpg - mean(mpg))
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128          32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic       30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla    33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona     21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger  15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin       15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
```

```
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
                    diferenca
Mazda RX4            0.909375
Mazda RX4 Wag        0.909375
Datsun 710           2.709375
Hornet 4 Drive       1.309375
Hornet Sportabout   -1.390625
Valiant             -1.990625
Duster 360          -5.790625
Merc 240D            4.309375
Merc 230             2.709375
Merc 280            -0.890625
Merc 280C           -2.290625
Merc 450SE          -3.690625
Merc 450SL          -2.790625
Merc 450SLC         -4.890625
Cadillac Fleetwood  -9.690625
Lincoln Continental -9.690625
Chrysler Imperial   -5.390625
Fiat 128            12.309375
Honda Civic         10.309375
Toyota Corolla      13.809375
Toyota Corona        1.409375
Dodge Challenger    -4.590625
AMC Javelin         -4.890625
Camaro Z28          -6.790625
Pontiac Firebird    -0.890625
Fiat X1-9            7.209375
Porsche 914-2        5.909375
Lotus Europa        10.309375
Ford Pantera L      -4.290625
Ferrari Dino        -0.390625
Maserati Bora       -5.090625
Volvo 142E           1.309375
```

Mas se quisermos saber a diferença de consumo com a média dos veículos que possuem o mesmo número de cilindros, podemos fazer:

```
mtcars |>
  group_by(cyl) |>
  mutate (diferenca = mpg - mean(mpg))
```

```
# A tibble: 32 x 12
# Groups:   cyl [3]
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb diferenca
   <dbl> <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     1.26
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4     1.26
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1    -3.86
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1     1.66
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2     3.6
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1    -1.64
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4    -0.800
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2    -2.26
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2    -3.86
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4    -0.543
# ... with 22 more rows
# i Use `print(n = ...)` to see more rows
```

*Veja também add_column(), add_count(), e add_tally().

4.5.3.2 transmutate

A função **transmutate** funciona de forma similar a função **mutate**, porém ela cria/altera uma ou mais colunas e ignora todas as demais em suas saída.

4.5.3.3 rename

Renomeia variáveis (colunas).

Há também a função rename_with() para chamar uma função para renomear a coluna.

A função rename(.data, …)

```
cars |>
  rename (distancia = dist) |>
```

```
  head ()
```

```
  speed distancia
1     4         2
2     4        10
3     7         4
4     7        22
5     8        16
6     9        10
```

4.6 Funções Vetorizadas

As funções **mutate()** e **transmute()** aplicam **funções vetorizadas** em colunas para criar novas colunas.

> **Nota**
>
> Funções vetorizadas, são chamadas também de funções de janela (**window functions**) e **recebem vetores** como argumento de entrada e **retornar vetores** de mesmo tamanho como saída.

A seguir temos algumas funções vetorizadas que auxiliam na manipulação de dados e, em geral, são utilizadas com mutate() ou filter().

4.6.1 Deslocamento

O dplyr possui funções para ajuste de deslocamento (offset). Estas funções são muito úteis para "encontrar" valores antes ou depois em relação aos valores atuais.

4.6.1.1 lag

Desloca elementos em n posições positivas. Usado para "encontrar o valor **anterior** em n posições".

Supondo que precisarmos criar uma coluna contendo o consumo do veiculo que aparece na linha anterior da tabela, podemos fazer:

```
mtcars |>
  mutate (mpg_anterior = lag(mpg))
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

	mpg_anterior
Mazda RX4	NA
Mazda RX4 Wag	21.0
Datsun 710	21.0
Hornet 4 Drive	22.8
Hornet Sportabout	21.4
Valiant	18.7
Duster 360	18.1
Merc 240D	14.3
Merc 230	24.4

```
Merc 280                  22.8
Merc 280C                 19.2
Merc 450SE                17.8
Merc 450SL                16.4
Merc 450SLC               17.3
Cadillac Fleetwood        15.2
Lincoln Continental       10.4
Chrysler Imperial         10.4
Fiat 128                  14.7
Honda Civic               32.4
Toyota Corolla            30.4
Toyota Corona             33.9
Dodge Challenger          21.5
AMC Javelin               15.5
Camaro Z28                15.2
Pontiac Firebird          13.3
Fiat X1-9                 19.2
Porsche 914-2             27.3
Lotus Europa              26.0
Ford Pantera L            30.4
Ferrari Dino              15.8
Maserati Bora             19.7
Volvo 142E                15.0
```

Caso os dados não estejam ordenados,

4.6.1.2 lead

Desloca elementos em n posições negativas. Usado para "encontrar o valor **posterior** em n posições".

Supondo que precisarmos criar uma coluna contendo o consumo do veiculo que aparece duas linhas posteriores da tabela, podemos fazer:

```
mtcars |>
  mutate (duas_linhas_posteriores = lead(mpg, n = 2))
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
```

```
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
                    duas_linhas_posteriores
Mazda RX4                              22.8
Mazda RX4 Wag                          21.4
Datsun 710                             18.7
Hornet 4 Drive                         18.1
Hornet Sportabout                      14.3
Valiant                                24.4
Duster 360                             22.8
Merc 240D                              19.2
Merc 230                               17.8
Merc 280                               16.4
Merc 280C                              17.3
Merc 450SE                             15.2
Merc 450SL                             10.4
```

```
Merc 450SLC                10.4
Cadillac Fleetwood         14.7
Lincoln Continental        32.4
Chrysler Imperial          30.4
Fiat 128                   33.9
Honda Civic                21.5
Toyota Corolla             15.5
Toyota Corona              15.2
Dodge Challenger           13.3
AMC Javelin                19.2
Camaro Z28                 27.3
Pontiac Firebird           26.0
Fiat X1-9                  30.4
Porsche 914-2              15.8
Lotus Europa               19.7
Ford Pantera L             15.0
Ferrari Dino               21.4
Maserati Bora                NA
Volvo 142E                   NA
```

4.6.2 Agregação Acumulada

O dplyr possui funções de agregações acumuladas. São versões vetorizadas de all, any e mean, enquanto outras são de soma e produtos acumulados e extremos (min/max).

> Dica
>
> As funções cumall() e cumany() são muito úteis quando usadas com **filter()**, pois avaliam uma expressão retornando um **vetor lógico** a partir do valor avaliado pela expressão:

4.6.2.1 cumall

Retorna todas as observações (linhas) até que o primeiro caso da expressão a ser avaliada seja falso.

No exemplo a seguir, iremos retornar todos os veículos até que encontre um veículo onde sua potência (**hp**) seja maior ou igual à **110**. Quando encontrar esta linha, todas as demais à partir dela na tabela serão ignoradas, mesmo que atendam a expressão. Ou seja, mesmo que houver um veículo com potencia 110 na última linha, nexte exemplo ela será ignorada.

```
mtcars |>
  filter(cumall(hp <= 110))
```

```
              mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710     22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
```

Se quisermos 'varrer" a tabela até encontrarmos um veículo que tenha a potência (hp) menor que 90 e ignorar todas as linhas depois dela, podemos usar:

```
mtcars |>
  filter(cumall(!hp <= 90))
```

```
                  mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4        21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag    21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710       22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive   21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7  8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant          18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
Duster 360       14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
```

4.6.2.2 cumany

Retorna todas as observações (linhas) após o primeiro caso da expressão a ser avaliada seja verdadeiro.

Por exemplo: Se quisermos "varrer" a tabela em busca do veículo com potência menor que 70 e obter esta e todas as demais linhas após, usamos:

```
mtcars |>
  filter (cumany(hp <70))
```

```
                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Merc 240D       24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230        22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
```

```
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

4.6.2.3 cummean

A função cummean() é similar a função cumall ou cumany, porém retorna um **vetor numérico** contento as médias do vetor de entrada.

Por exemplo, se quisermos criar uma coluna com o valor da média de potência conforme "varremos" a tabela, ou seja, conforme a potência de cada veículo é listada, usamos:

```
mtcars |>
  mutate (media_acumulada = cummean(hp))
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
```

```
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
                    media_acumulada
Mazda RX4                  110.0000
Mazda RX4 Wag              110.0000
Datsun 710                 104.3333
Hornet 4 Drive             105.7500
Hornet Sportabout          119.6000
Valiant                    117.1667
Duster 360                 135.4286
Merc 240D                  126.2500
Merc 230                   122.7778
Merc 280                   122.8000
Merc 280C                  122.8182
Merc 450SE                 127.5833
Merc 450SL                 131.6154
Merc 450SLC                135.0714
Cadillac Fleetwood         139.7333
```

```
Lincoln Continental        144.4375
Chrysler Imperial          149.4706
Fiat 128                   144.8333
Honda Civic                139.9474
Toyota Corolla             136.2000
Toyota Corona              134.3333
Dodge Challenger           135.0455
AMC Javelin                135.6957
Camaro Z28                 140.2500
Pontiac Firebird           141.6400
Fiat X1-9                  138.7308
Porsche 914-2              136.9630
Lotus Europa               136.1071
Ford Pantera L             140.5172
Ferrari Dino               141.6667
Maserati Bora              147.9032
Volvo 142E                 146.6875
```

4.6.2.4 cummax

Use para saber o valor máximo acumulado até aquela linha.

Por exemplo, se agruparmos os veículos de mtcars de acordo com a coluna de número de cilindros (cyl), podemos saber qual o maior valor acumulado da coluna potência (hp) por grupo de veículos conforme "varremos" a tabela:

```
mtcars |>
  group_by(cyl) |>
  mutate (cummax = cummax(hp))
```

```
# A tibble: 32 x 12
# Groups:   cyl [3]
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb cummax
   <dbl> <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    110
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4    110
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1     93
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1    110
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2    175
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1    110
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4    245
```

```
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2     93
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2     95
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4    123
# ... with 22 more rows
# i Use `print(n = ...)` to see more rows
```

4.6.2.5 cummin

Use para saber o valor mínimo acumulado até aquela linha.

Por exemplo, se agruparmos os veículos de mtcars de acordo com a coluna de número de cilindros (cyl), podemos saber qual o menor valor acumulado da coluna potência (hp) por grupo de veículos conforme "varremos" a tabela:

```
mtcars |>
  group_by(cyl) |>
  mutate (cummin = cummin (hp))
```

```
# A tibble: 32 x 12
# Groups:   cyl [3]
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb cummin
   <dbl> <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    110
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4    110
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1     93
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1    110
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2    175
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1    105
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4    175
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2     62
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2     62
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4    105
# ... with 22 more rows
# i Use `print(n = ...)` to see more rows
```

4.6.2.6 cumprod

Similar a cummin() ou cummax(). Use para saber o valor multiplicado acumulado até aquela linha.

Por exemplo, para acumularmos a multiplicação dos pesos (wt) de cada veículo até a linha, usamos:

```
mtcars |>
  mutate (cumprod = cumprod (wt))
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood 10.4  8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4 8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128          32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic       30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla    33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona     21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger  15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin       15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28        13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird  19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9         27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2     26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa      30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L    15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino      19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora     15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E        21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
                      cumprod
Mazda RX4         2.620000e+00
Mazda RX4 Wag     7.532500e+00
```

```
Datsun 710          1.747540e+01
Hornet 4 Drive      5.618341e+01
Hornet Sportabout   1.932709e+02
Valiant             6.687174e+02
Duster 360          2.387321e+03
Merc 240D           7.615555e+03
Merc 230            2.398900e+04
Merc 280            8.252215e+04
Merc 280C           2.838762e+05
Merc 450SE          1.155376e+06
Merc 450SL          4.309553e+06
Merc 450SLC         1.629011e+07
Cadillac Fleetwood  8.552308e+07
Lincoln Continental 4.638772e+08
Chrysler Imperial   2.479424e+09
Fiat 128            5.454732e+09
Honda Civic         8.809392e+09
Toyota Corolla      1.616523e+10
Toyota Corona       3.984730e+10
Dodge Challenger    1.402625e+11
AMC Javelin         4.818017e+11
Camaro Z28          1.850118e+12
Pontiac Firebird    7.113706e+12
Fiat X1-9           1.376502e+13
Porsche 914-2       2.945714e+13
Lotus Europa        4.456866e+13
Ford Pantera L      1.412826e+14
Ferrari Dino        3.913529e+14
Maserati Bora       1.397130e+15
Volvo 142E          3.884021e+15
```

4.6.2.7 cumsum

Similar a cummin() ou cummax(). Use para saber o valor da soma acumulada até aquela linha.

Por exemplo, para acumularmos a soma dos pesos (wt) de cada veículo até a linha, usamos:

```
mtcars |>
  mutate (cumsum = cumsum(wt))
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb  cumsum
```

Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4	2.620
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4	5.495
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1	7.815
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1	11.030
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2	14.470
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1	17.930
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4	21.500
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2	24.690
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2	27.840
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4	31.280
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4	34.720
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3	38.790
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3	42.520
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3	46.300
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4	51.550
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4	56.974
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4	62.319
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1	64.519
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2	66.134
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1	67.969
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1	70.434
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2	73.954
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2	77.389
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4	81.229
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2	85.074
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1	87.009
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2	89.149
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2	90.662
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4	93.832
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6	96.602
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8	100.172
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2	102.952

4.6.3 Ranqueamento

4.6.3.1 row_number

Use para adicionar o número da linha. Como as demais funções do dplyr, ela respeita os grupos da tabela quando houver.

Por exemplo, para enumerar as linhas dos veículos agrupados por número de cilindros, podemos usar:

```
mtcars |>
  group_by(cyl) |>
  mutate (num_linha = row_number())
```

```
# A tibble: 32 x 12
# Groups:   cyl [3]
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb num_linha
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>     <int>
 1  21       6  160    110  3.9   2.62  16.5     0     1     4     4         1
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4         2
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1         1
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1         3
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2         1
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1         4
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4         2
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2         2
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2         3
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4         5
# ... with 22 more rows
# i Use `print(n = ...)` to see more rows
```

4.6.3.2 rank

Use para criar um ranqueamento de uma variável.

Por exemplo, para fazer um ranqueamento dos veículos mais leves entre grupos de mesma cilindragem, podemos usar:

```
mtcars |>
  group_by (cyl) |>
  mutate (rank = rank(wt))
```

```
# A tibble: 32 x 12
# Groups:   cyl [3]
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb  rank
   <dbl> <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   1
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4   3
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1   7
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1   4
```

```
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2   3
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1   7
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4   5.5
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2  11
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2  10
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4   5.5
# ... with 22 more rows
# i Use `print(n = ...)` to see more rows
```

Observe que para veículos de 8 cilindros, a segunda e última linha possuem o mesmo peso e portanto tiveram o mesmo ranqueamento (5.5).

4.6.3.3 dense_rank

Use para ajustar o ranqueamento sem pular lacunas em casos de empate.

Vejo abaixo como ele se compara ao ranqueamento da função rank().

```
mtcars |>
  group_by (cyl) |>
  mutate (rank = rank(wt), dense_rank = dense_rank(wt))
```

```
# A tibble: 32 x 13
# Groups:   cyl [3]
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb  rank
   <dbl> <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   1
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4   3
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1   7
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1   4
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2   3
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1   7
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4   5.5
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2  11
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2  10
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4   5.5
# ... with 22 more rows, and 1 more variable: dense_rank <int>
# i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

Observe que na segunda e última linha de 8 cilindros, empatam em 5.5 na função rank() e a posição 6 é pulada, sendo o próximo do ranqueamento o número 7. Já na função dense_rank(), o empate fica na posição 5 e o próximo na posição 6.

4.6.3.4 percent_rank

Use para retornar um número entre 0 e 1 (percentual) do ranqueamento mínimo.

Veja abaixo como ela se compara com as funções rank e dense_rank:

```
mtcars |>
  group_by (cyl) |>
  mutate (rank = rank(wt), dense_rank = dense_rank(wt), percent_rank = percent_rank(wt))
```

```
# A tibble: 32 x 14
# Groups:   cyl [3]
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb  rank
   <dbl> <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   1
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4   3
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1   7
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1   4
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2   3
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1   7
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4   5.5
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2  11
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2  10
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4   5.5
# ... with 22 more rows, and 2 more variables: dense_rank <int>,
#   percent_rank <dbl>
# i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

4.6.3.5 cume_dist

Use para saber a distribuição acumulada de acordo com o ranqueamento atribuído.

Por exemplo, se agruparmos por número de cilindros (cyl), criarmos uma coluna de ranqueamento (**rank**) e outra coluna com a distribuição, podemos saber como o ranqueamento acumulado está distribuído.

```
mtcars |>
  group_by(cyl) |>
  mutate (rank = rank(wt), cume_dist = cume_dist(wt)) |>
  arrange (cume_dist)
```

```
# A tibble: 32 x 13
# Groups:   cyl [3]
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb  rank
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  15.8     8 351     264  4.22  3.17  14.5     0     1     5     4     1
 2  30.4     4  95.1   113  3.77  1.51  16.9     1     1     5     2     1
 3  21       6 160     110  3.9   2.62  16.5     0     1     4     4     1
 4  15.2     8 304     150  3.15  3.44  17.3     0     0     3     2     2
 5  30.4     4  75.7    52  4.93  1.62  18.5     1     1     4     2     2
 6  18.7     8 360     175  3.15  3.44  17.0     0     0     3     2     3
 7  33.9     4  71.1    65  4.22  1.84  19.9     1     1     4     1     3
 8  15.5     8 318     150  2.76  3.52  16.9     0     0     3     2     4
 9  19.7     6 145     175  3.62  2.77  15.5     0     1     5     6     2
10  27.3     4  79      66  4.08  1.94  18.9     1     1     4     1     4
# ... with 22 more rows, and 1 more variable: cume_dist <dbl>
# i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

4.6.3.6 ntile

Use para retornar um quantil (percentil, quartil, etc) de um vetor.

Por exemplo, para segregarmos os veículos em quartis (quatro partes) (0-0.25-.50-.75-1), podemos usar:

```
mtcars |>
  mutate (quartil = ntile(wt, 4))
```

```
                  mpg cyl  disp  hp drat    wt  qsec vs am gear carb quartil
Mazda RX4        21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4       2
Mazda RX4 Wag    21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4       2
Datsun 710       22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1       1
Hornet 4 Drive   21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1       2
Hornet Sportabout 18.7  8 360.0 175 3.15 3.440 17.02  0  0    3    2       3
Valiant          18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1       3
Duster 360       14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4       3
Merc 240D        24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2       2
Merc 230         22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2       2
Merc 280         19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4       3
Merc 280C        17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4       3
Merc 450SE       16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3       4
Merc 450SL       17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3       4
Merc 450SLC      15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3       4
```

```
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4       4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4       4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4       4
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1       1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2       1
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1       1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1       1
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2       3
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2       3
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4       4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2       4
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1       1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2       1
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2       1
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4       2
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6       2
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8       3
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2       2
```

4.6.4 Matemática

4.6.4.1 operações e logs:

São símbolos para funções matemáticas mais comuns como:

+, - , *, /, ^, %/%, %% - Usados para as oper. aritiméticas

log(), **log2()**, **log10()** - Usados para logaritmos

4.6.4.2 between

Use para retornar os valores do vetor entre dois valores. É o equivalente a escrever: x >= valor_esquerda & x <= valor_direita.

Por exemplo, se quisermos obter os veículos com o peso entre 3 a 4 mil libras, podemos usar a função **between()** juntamente com a função **filter()**.

```
mtcars |>
  filter (between(wt, 3, 4))
```

```
                  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Dodge Challenger  15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin       15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28        13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird  19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Ford Pantera L    15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Maserati Bora     15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
```

4.6.4.3 near

Use para verificar se dois vetores de ponto fultuante são iguais:

O exemplo abaixo, duplica a coluna de peso (wt –> wt2) dos veículos e altera o primeiro valor de **2.620** para **2.600**, salvando em um novo data frame (*mtcars_novo*) que será usado em seguida;

```
mtcars_novo <- mtcars |>
  mutate(wt2 = wt) |>
  mutate (wt2 = ifelse(row_number() == 1, 2.600, wt2))
```

Se usarmos o igual (==) para validar ambas colunas, devido às diferenças nas casa decimais, teríamos **FALSO**:

```
mtcars_novo$wt == mtcars_novo$wt2
```

```
 [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[25]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Se usarmos a função **near**, por esta ter um parâmetro de **tolerância** que iremos definiar como acima de 0.020 (0.021 por exemplo), teríamos **VERDADEIRO** para todos os items do vetor:

```
near (mtcars_novo$wt, mtcars_novo$wt2, tol = 0.021)
```

```
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[16] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[31] TRUE TRUE
```

4.6.5 Miscelânea

4.6.5.1 if_else

Use para fazer um "SE" (IF) vetorizado, ou seja, elemento por elemento.

Por exemplo, se quisermos saber quais veículos percorrem 1/4 de milha em menos de 20 segundos, podemos usar a coluna **qsec** com a função **if_else** para criar uma nova coluna marcando como "**MAIX VELOZ**" os veículos que percorrem com menos de 20 segundos e "**MENOS VELOZ**" os que não atendem a este critério:

```
mtcars |>
  mutate (categoria_arranque =
            ifelse(qsec < 20, "MAIS VELOZ", "MENOS VELOZ"))
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3

```
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
                    categoria_arranque
Mazda RX4                    MAIS VELOZ
Mazda RX4 Wag                MAIS VELOZ
Datsun 710                   MAIS VELOZ
Hornet 4 Drive               MAIS VELOZ
Hornet Sportabout            MAIS VELOZ
Valiant                     MENOS VELOZ
Duster 360                   MAIS VELOZ
Merc 240D                   MENOS VELOZ
Merc 230                    MENOS VELOZ
Merc 280                     MAIS VELOZ
Merc 280C                    MAIS VELOZ
Merc 450SE                   MAIS VELOZ
Merc 450SL                   MAIS VELOZ
Merc 450SLC                  MAIS VELOZ
Cadillac Fleetwood           MAIS VELOZ
Lincoln Continental          MAIS VELOZ
Chrysler Imperial            MAIS VELOZ
Fiat 128                     MAIS VELOZ
Honda Civic                  MAIS VELOZ
Toyota Corolla               MAIS VELOZ
Toyota Corona               MENOS VELOZ
Dodge Challenger             MAIS VELOZ
AMC Javelin                  MAIS VELOZ
Camaro Z28                   MAIS VELOZ
```

```
Pontiac Firebird            MAIS VELOZ
Fiat X1-9                   MAIS VELOZ
Porsche 914-2               MAIS VELOZ
Lotus Europa                MAIS VELOZ
Ford Pantera L              MAIS VELOZ
Ferrari Dino                MAIS VELOZ
Maserati Bora               MAIS VELOZ
Volvo 142E                  MAIS VELOZ
```

4.6.5.2 case_when

Use quando quiser fazer um if_else com mais de dois casos, ou seja, ao invés de aninhá-los(nested if).

Usando o mesmo exemplo da função if_else , porém gostaríamos de criar um terceira classificação chamada de "SUPER VELOZ" para os veículos com tempo abaixo de 16 segundos, teríamos:

```
mtcars |>
  mutate (categoria_arranque = case_when(
    qsec < 17 ~ "SUPER VELOZ",
    qsec < 20 ~ "MAIS VELOZ",
    TRUE ~ "MENOS VELOZ"))
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
```

```
Fiat 128           32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic        30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla     33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona      21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger   15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin        15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28         13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird   19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9          27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2      26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa       30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L     15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino       19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora      15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E         21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
                   categoria_arranque
Mazda RX4                 SUPER VELOZ
Mazda RX4 Wag              MAIS VELOZ
Datsun 710                 MAIS VELOZ
Hornet 4 Drive             MAIS VELOZ
Hornet Sportabout          MAIS VELOZ
Valiant                   MENOS VELOZ
Duster 360                SUPER VELOZ
Merc 240D                 MENOS VELOZ
Merc 230                  MENOS VELOZ
Merc 280                   MAIS VELOZ
Merc 280C                  MAIS VELOZ
Merc 450SE                 MAIS VELOZ
Merc 450SL                 MAIS VELOZ
Merc 450SLC                MAIS VELOZ
Cadillac Fleetwood         MAIS VELOZ
Lincoln Continental        MAIS VELOZ
Chrysler Imperial          MAIS VELOZ
Fiat 128                   MAIS VELOZ
Honda Civic                MAIS VELOZ
Toyota Corolla             MAIS VELOZ
Toyota Corona             MENOS VELOZ
Dodge Challenger          SUPER VELOZ
AMC Javelin                MAIS VELOZ
Camaro Z28                SUPER VELOZ
Pontiac Firebird           MAIS VELOZ
Fiat X1-9                  MAIS VELOZ
Porsche 914-2             SUPER VELOZ
```

```
Lotus Europa                SUPER VELOZ
Ford Pantera L              SUPER VELOZ
Ferrari Dino                SUPER VELOZ
Maserati Bora               SUPER VELOZ
Volvo 142E                   MAIS VELOZ
```

A função **case_when**, também permite que você utilize diferentes colunas, e operações lógicas também.

Por exemplo, se quisermos colocar a categoria "**INVÁLIDO**" para os veículos com 8 cilindros (cyl) **OU** consumo acima de 30 galões/milha (mpg), usamos:

```
mtcars |>
  mutate (categoria_arranque =
            case_when(
              cyl == 8 | mpg > 30 ~ "INVALIDO",
              qsec < 17 ~ "SUPER VELOZ",
              qsec < 20 ~ "MAIS VELOZ",
              TRUE ~ "MENOS VELOZ"))
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood 10.4  8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4 8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128          32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic       30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla    33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona     21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
```

```
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
                    categoria_arranque
Mazda RX4                  SUPER VELOZ
Mazda RX4 Wag               MAIS VELOZ
Datsun 710                  MAIS VELOZ
Hornet 4 Drive              MAIS VELOZ
Hornet Sportabout             INVALIDO
Valiant                    MENOS VELOZ
Duster 360                    INVALIDO
Merc 240D                  MENOS VELOZ
Merc 230                   MENOS VELOZ
Merc 280                    MAIS VELOZ
Merc 280C                   MAIS VELOZ
Merc 450SE                    INVALIDO
Merc 450SL                    INVALIDO
Merc 450SLC                   INVALIDO
Cadillac Fleetwood            INVALIDO
Lincoln Continental           INVALIDO
Chrysler Imperial             INVALIDO
Fiat 128                      INVALIDO
Honda Civic                   INVALIDO
Toyota Corolla                INVALIDO
Toyota Corona              MENOS VELOZ
Dodge Challenger              INVALIDO
AMC Javelin                   INVALIDO
Camaro Z28                    INVALIDO
Pontiac Firebird              INVALIDO
Fiat X1-9                   MAIS VELOZ
Porsche 914-2              SUPER VELOZ
Lotus Europa                  INVALIDO
Ford Pantera L                INVALIDO
Ferrari Dino               SUPER VELOZ
Maserati Bora                 INVALIDO
```

```
Volvo 142E                  MAIS VELOZ
```

4.6.5.3 coalesce

Use para sobrescrever os valores de NA em um vetor.

O exemplo abaixo, sobrescreve com zeros os valores faltantes (missing) do vetor x:

```
x <- sample(c(1:5, NA, NA, NA))
coalesce(x, 0L)
```

```
[1] 0 5 0 0 2 3 4 1
```

> Dica
>
> Use **na_if()** para sobrescrever um valor específico com NA e **tidyr::replace_na()** para sobrescrever os NAs com um valor.

4.6.5.4 na_if

Use para sobrescrever um valor específico com NA.

Por exemplo, se quisermos colocar NAs na coluna de consumo (mpg) para os veículos com 8 cilindros (cyl), podemos usar a na_if() juntamente com a mutate():

```
mtcars |>
  mutate(mpg = na_if(cyl, 8))
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4            6   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag        6   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710           4   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive       6   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout   NA   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant              6   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360          NA   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D            4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230             4   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280             6   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

```
Merc 280C             6   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE           NA   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL           NA   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC          NA   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood   NA   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental  NA   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial    NA   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128              4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic           4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla        4   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona         4   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger     NA   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin          NA   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28           NA   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird     NA   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9             4   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2         4   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa          4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L       NA   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino          6   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora        NA   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E            4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

4.6.5.5 pmax - max

Use max() para retornar o **máximo** valor de um vetor ou pmax() para retornar o valor máximo entre elementos de vetores em paralelo.

Por exemplo, se tiver um vetor dois vetores (x e y) com 5 elementos cada, a função **max()** irá retornar o maior vetor contido no vetor escolhido:

```r
x <- c(1, 2, 3, 4, 5)
y <- c(1, 2, 10, 1, 20)
max(x)
```

```
[1] 5
```

```r
max(y)
```

```
[1] 20
```

Já se usarmos o pmax(), ele irá comparar cada elemento de x com seu respectivo elemento de y e retornar o máximo valor.

```
x <- c(1, 2, 3, 4, 5)
y <- c(1, 1, 10, 1, 20)
pmax(x, y)
```

```
[1]  1  2 10  4 20
```

4.6.5.6 pmin - min

É idêntico às funções pmax - max (), porém ao invés de retornar o valor máximo ou o máximo do elemento em paralelo, estas funções retornam seus respectivos valores **mínimos**.

4.7 Funções de Resumo

summarise() aplica funções de resumo em colunas para criar uma nova tabela. Funções de resumo **recebem** vetores como entrada e **retornam um valor único** na saída.

> Cuidado
>
> Estas funções retornam informações sobre o grupo ou variável corrente, portanto só funcionam dentro de um **contexto** específico como **summarise()** ou **mutate()**.

4.7.1 Contagem

4.7.1.1 n

Use para retornar o tamanhio do grupo corrente.

Por exemplo, para contar quantos veículos temos em cada grupo de cilindors, podemos usar a summarise() com o n() depois de agrupar pela coluna de cilindors (cyl).

```
mtcars |>
  group_by(cyl) |>
  summarise(numero_veículos = n())
```

```
# A tibble: 3 x 2
    cyl numero_veículos
  <dbl>           <int>
1     4              11
2     6               7
3     8              14
```

4.7.1.2

4.7.1.3 n_distinct

Use para contar os valores unicos em um vetor.

Por exemplo, para saber **QUANTAS** categorias de cilindros temos na tabela mtcars, podemos usar:

```
  mtcars |>
    summarise (cat_cilindros = n_distinct(cyl))
```

```
  cat_cilindros
1             3
```

Se quisessemos saber **QUAIS** categoris de cilindros temos, podemos fazer usando a distinct:

mtcars |> unique (cyl) ### Posição

```
  mtcars |>
    distinct(cyl)
```

```
                  cyl
Mazda RX4           6
Datsun 710          4
Hornet Sportabout   8
```

4.7.1.4 sum

Use para retornar a soma de todos os valores presentes em seu argumento.

Por exemplo, para sabermos a soma dos pesos (wt) dos veículos agrupados pelo número de cilindros (cyl), podemos usar a função sum, juntamente com o summarise() e group_by().

```r
mtcars |>
  group_by(cyl) |>
  summarise(soma_pesos = sum(wt))
```

```
# A tibble: 3 x 2
    cyl soma_pesos
  <dbl>      <dbl>
1     4       25.1
2     6       21.8
3     8       56.0
```

Veja que se houver NA na variável (coluna), a função sum irá retornar NA. Para ignorar os NAs e fazer a soma, use o argumento na.rm = TRUE.

4.7.2 Posição

4.7.2.1 mean

Use para obter a média dos elementos do vetor. Por exemplo, se quisermos saber a média de peso de todos os veículos, podemos usar:

```r
mtcars |>
  summarise (media_peso =mean(wt))
```

```
  media_peso
1    3.21725
```

4.7.2.2 median

Use para obter a **mediana** dos elementos de um vetor. è similar à função mean, porém retorna a mediana ou invés da média.

4.7.3 Ordem

4.7.3.1 first

Use para obter o **primeiro** elemento de um vetor.

Por exemplo, se quisermos obter o peso do primeiro veículo de cada grupo de cilindros, podemos usar:

```
mtcars |>
  group_by(cyl) |>
  summarise(primeiro_veiculo_do_grupo = first(wt))
```

```
# A tibble: 3 x 2
    cyl primeiro_veiculo_do_grupo
  <dbl>                     <dbl>
1     4                      2.32
2     6                      2.62
3     8                      3.44
```

> Dica
>
> Usando o parametro order_by = , podemos passar um vetor para determinar uma ordem.

Por exemplo, se criarmos um agrupamento pelo numero de cilindros e criarmos um ranqueamento pelo consumo do veículo, podemos obter o peso do veículo de menor consumo usando:

```
mtcars |>
  group_by(cyl) |>
  mutate (rank_peso = rank(mpg)) |>
  summarise(primeiro_veiculo_ordenado_pelo_cosumo =
      first(wt, order_by = rank_peso))
```

```
# A tibble: 3 x 2
    cyl primeiro_veiculo_ordenado_pelo_cosumo
  <dbl>                                 <dbl>
1     4                                  2.78
2     6                                  3.44
3     8                                  5.25
```

4.7.3.2 last

Use para obter o **último** elemento. É similar a função first, porém ao invés de retornar o primeiro elemento, irá retornar o último.

4.7.3.3 nth

Use para obter o **n-ésimo** elemento. É similar a função first, porém ao invés de retornar o primeiro elemento, irá retornar o n-ésimo elemento.

Por exemplo, se quisermos obter o peso do **segundo** veículo de cada grupo de cilindros, podemos usar:

```
mtcars |>
  group_by(cyl) |>
  summarise(segundo_veiculo_do_grupo = nth(wt,2))
```

```
# A tibble: 3 x 2
    cyl segundo_veiculo_do_grupo
  <dbl>                    <dbl>
1     4                     3.19
2     6                     2.88
3     8                     3.57
```

Use valor negativo para contar a partir do último elemento.

Por exemplo, se quisermos obter o peso do **penúltimo** veículo de cada grupo de cilindros, podemos usar:

```
mtcars |>
  group_by(cyl) |>
  summarise(penultimo_veiculo_do_grupo = nth(wt,-2))
```

```
# A tibble: 3 x 2
    cyl penultimo_veiculo_do_grupo
  <dbl>                      <dbl>
1     4                       1.51
2     6                       3.44
3     8                       3.17
```

4.7.4 Ranqueamento

4.7.4.1 quantile

Use para obter os quantils de um vetor. Por padrão retorna os quartis (0, 0.25, .5, .75 e 1) de um vetor.

Por exemplo, se quisermos saber os quartis dos pesos dos grupos de cilindors dos veículos, podemos usar:

```
mtcars |>
  group_by(cyl) |>
  summarise(q = quantile(mpg)) |>
  ungroup()
```

```
# A tibble: 15 x 2
     cyl     q
   <dbl> <dbl>
 1     4  21.4
 2     4  22.8
 3     4  26
 4     4  30.4
 5     4  33.9
 6     6  17.8
 7     6  18.6
 8     6  19.7
 9     6  21
10     6  21.4
11     8  10.4
12     8  14.4
13     8  15.2
14     8  16.2
15     8  19.2
```

Usando o parâmeto probs = , podemos definir quanquer quantil.

Por exemplo, se quisermos obter apenas o segundo (média) quartil e o último (valor máximo) do cosumo de grupo de cilindors dos veículos, podemos usar:

```
mtcars |>
  group_by(cyl) |>
  summarise(q = quantile(mpg, probs = c(.5, 1))) |>
```

```
    ungroup()
```

```
# A tibble: 6 x 2
    cyl     q
  <dbl> <dbl>
1     4  26
2     4  33.9
3     6  19.7
4     6  21.4
5     8  15.2
6     8  19.2
```

4.7.4.2 min

Use para obter o valor **mínimo**. Similar a quantile.

4.7.4.3 max

Use para obter o valor **máximo**. Similar a quantile.

4.7.5 Dispersão

O dyplr possuem algumas funções para avaliar o espalhamento dos dados (dispersão) em torno da média central. Estas funções vem do campo da estatística.

4.7.5.1 var

Use para calcular a variância dos dados.

Por exemplo, par calcular a **variância** total dos pesos (wt) dos veículos de tabela mtcars agrupada pelo número de cilindros (cyl), podemos usa-la junto com a função summarise:

```
mtcars |>
  group_by(cyl) |>
  summarise(varianca = var(wt))
```

```
# A tibble: 3 x 2
    cyl varianca
  <dbl>    <dbl>
1     4    0.324
2     6    0.127
3     8    0.577
```

No exemplo acima, observamos que os veículos com 6cilindros possuem seus pesos mais próximos da média que os veículos com 4 ou 8 cilindros.

4.7.5.2 sd

Use para calcular o **desvio padrão**. É similar a função var, porém retorna o desvio padrão ao invés da variância.

Usando o mesmo exemplo da função var, o código ficaria:

```
mtcars |>
  group_by(cyl) |>
  summarise(desvio_padrao = sd(wt))
```

```
# A tibble: 3 x 2
    cyl desvio_padrao
  <dbl>         <dbl>
1     4         0.570
2     6         0.356
3     8         0.759
```

4.7.5.3 IQR

Use para calcular a **distância inter-quartil**. É similar a função var, porém retorna o range entre os quartis ao invés da variância.

Usando o mesmo exemplo da função var, o código ficaria:

```
mtcars |>
  group_by(cyl) |>
  summarise(distancia_interquartil = IQR(wt))
```

```
# A tibble: 3 x 2
    cyl distancia_interquartil
  <dbl>                  <dbl>
1     4                  0.737
2     6                  0.618
3     8                  0.481
```

4.7.5.4 mad

Use para calcular a **desvio absoluto da mediana**. É similar a função var, porém o desvio absoluto da mediana ao invés da variância.

Usando o mesmo exemplo da função var, o código ficaria:

```
mtcars |>
  group_by(cyl) |>
  summarise(Desv_absolto_mediana = mad(wt))
```

```
# A tibble: 3 x 2
    cyl Desv_absolto_mediana
  <dbl>                <dbl>
1     4                0.541
2     6                0.363
3     8                0.408
```

4.8 Combinando Tabelas

4.8.1 Juntando Variáveis

Quando você tem uma ou mais variáveis de um dataframe com o **mesmo número de observações** de outra(s) variável(eis), você pode uní-las diretamente atraveś da função **bind_cols**(), porém se o **número de observações forem diferentes**, é necessário utilizar funções de união transformadoras (mutating joins). Ver a seção Relacionando Dados para maiores detalhes sobre as uniões transformadoras.

4.8.1.1 bind_cols

Use para unir variáveis de mesmo número de observações.

Por exemplo, no pacotes datasets, temos um dataframe (tabela), chamado **state.names,** que possui o nomes de todos os estados dos EUA. Há também o **state.abb**, que possui uma lista das abreviações dos estados americanos. Se quiser unir ambas variáveis, podemos usar:

```
bind_cols(state.abb, state.name)
```

```
# A tibble: 50 x 2
   ...1  ...2
   <chr> <chr>
 1 AL    Alabama
 2 AK    Alaska
 3 AZ    Arizona
 4 AR    Arkansas
 5 CA    California
 6 CO    Colorado
 7 CT    Connecticut
 8 DE    Delaware
 9 FL    Florida
10 GA    Georgia
# ... with 40 more rows
# i Use `print(n = ...)` to see more rows
```

4.8.2 Relacionando Dados

Quando temos **dois dataframes** (tabelas) e queremos adicionar colunas de um à outro, usamos funções chamadas de **uniões transformadoras** ("mutating joins"). Estas funções transformam um tabela, adicionando coluna(s) de outra tabela de acordo com as linhas baseado em chaves ("keys") definidas ao usar a função.

Preprarando as tabelas de Exemplo:

Para os exemplos a seguir, utilizaremos o dataframe **state.x77**, que possui 8 variáveis dos estados americanos com expectativa de vida, renda per capita, população, etc. Chamaremos esta tabela de **X**.

Chamaremos de **Y,** uma tabela states.abb e state.name, que possui as abreviações e nomes dos estados americanos respectivamente. Iremos unir estas variáveis com a função bind_cols, já que elas possuem o mesmo número de observações.

Porém para exemplificar casos onde a tabela Y, não possui extamente o mesmo número de observações da tabela X, iremos excluir as abreviações que começam com as letras C, M, I e V. Com isto, nossa table **Y** final, irá contar apenas **37** observações, enquanto a tabela **X** irá conter todos os **50** estados americados.

Observe que ambas as tabelas (X e Y) contém ao menos uma coluna em comum (coluna "Estado"). Esta coluna será utilizada como chave (key) para fazermos as uniões.

Para criar as tabelas X e Y, usaremos:

```
X <- state.x77 |>
  as_tibble() |>
  bind_cols(state.name) |>
  rename("Estado" = "...9")
X
```

```
# A tibble: 50 x 9
   Population Income Illiteracy `Life Exp` Murder `HS Grad` Frost   Area Estado
        <dbl>  <dbl>      <dbl>      <dbl>  <dbl>     <dbl> <dbl>  <dbl> <chr>
 1       3615   3624        2.1       69.0   15.1      41.3    20  50708 Alabama
 2        365   6315        1.5       69.3   11.3      66.7   152 566432 Alaska
 3       2212   4530        1.8       70.6    7.8      58.1    15 113417 Arizona
 4       2110   3378        1.9       70.7   10.1      39.9    65  51945 Arkans~
 5      21198   5114        1.1       71.7   10.3      62.6    20 156361 Califo~
 6       2541   4884        0.7       72.1    6.8      63.9   166 103766 Colora~
 7       3100   5348        1.1       72.5    3.1      56     139   4862 Connec~
 8        579   4809        0.9       70.1    6.2      54.6   103   1982 Delawa~
 9       8277   4815        1.3       70.7   10.7      52.6    11  54090 Florida
10       4931   4091        2         68.5   13.9      40.6    60  58073 Georgia
# ... with 40 more rows
# i Use `print(n = ...)` to see more rows
```

```
Y <- bind_cols(state.abb, state.name) |>
  as_tibble() |>
  rename("Abreviacao" = "...1", "Estado" = "...2") |>
  filter(!str_detect(Abreviacao, "^[CMIV]"))
Y
```

```
# A tibble: 33 x 2
   Abreviacao Estado
   <chr>      <chr>
```

```
 1 AL         Alabama
 2 AK         Alaska
 3 AZ         Arizona
 4 AR         Arkansas
 5 DE         Delaware
 6 FL         Florida
 7 GA         Georgia
 8 HI         Hawaii
 9 KS         Kansas
10 KY         Kentucky
# ... with 23 more rows
# i Use `print(n = ...)` to see more rows
```

4.8.2.1 left_join

Use para unir valores iguais de X em Y.

Por exemplo, usando as tabelas **X** e **Y** preparadas no início da seção Relacionando Dados, podemos unir a variável população (population) que está na tabela X na tabela Y.

> **!** Importante
>
> A coluna que será usada como chave (key) na união e as variáveis que você quer unir devem ser selecionadas (função **select**()). Se não houver uma seleção explícita, todas as colunas da tabela X serão unídas.

```
left_join(Y, select(X, Estado, Population))
```

```
# A tibble: 33 x 3
   Abreviacao Estado   Population
   <chr>      <chr>         <dbl>
 1 AL         Alabama        3615
 2 AK         Alaska          365
 3 AZ         Arizona        2212
 4 AR         Arkansas       2110
 5 DE         Delaware        579
 6 FL         Florida        8277
 7 GA         Georgia        4931
 8 HI         Hawaii          868
 9 KS         Kansas         2280
10 KY         Kentucky       3387
```

```
# ... with 23 more rows
# i Use `print(n = ...)` to see more rows
```

Veja que no exemplo acima, foram retornadas apenas **37** observações. Isto porque definimos como tabela "base" a tabela Y. Se fizermos o inverso, ou seja, left_join (X, Y), a saída irá conter todos os **50** registros de X e os que ele encontrar na tabela Y. Os que não forem encontrados de acordo com a chave escolhida (neste caso a coluna Estado), será preenchido com N/A. Veja exemplo abaixo:

```
left_join(X, Y, by = "Estado")
```

```
# A tibble: 50 x 10
   Population Income Illite~1 Life ~2 Murder HS Gr~3 Frost   Area Estado Abrev~4
        <dbl>  <dbl>    <dbl>   <dbl>  <dbl>   <dbl> <dbl>  <dbl> <chr>  <chr>
 1       3615   3624      2.1    69.0   15.1    41.3    20  50708 Alaba~ AL
 2        365   6315      1.5    69.3   11.3    66.7   152 566432 Alaska AK
 3       2212   4530      1.8    70.6    7.8    58.1    15 113417 Arizo~ AZ
 4       2110   3378      1.9    70.7   10.1    39.9    65  51945 Arkan~ AR
 5      21198   5114      1.1    71.7   10.3    62.6    20 156361 Calif~ <NA>
 6       2541   4884      0.7    72.1    6.8    63.9   166 103766 Color~ <NA>
 7       3100   5348      1.1    72.5    3.1    56     139   4862 Conne~ <NA>
 8        579   4809      0.9    70.1    6.2    54.6   103   1982 Delaw~ DE
 9       8277   4815      1.3    70.7   10.7    52.6    11  54090 Flori~ FL
10       4931   4091      2      68.5   13.9    40.6    60  58073 Georg~ GA
# ... with 40 more rows, and abbreviated variable names 1: Illiteracy,
#   2: `Life Exp`, 3: `HS Grad`, 4: Abreviacao
# i Use `print(n = ...)` to see more rows
```

As colunas unidas da tabela Y, são adicionadas ao lado direito da tabela X.

> **Nota**
>
> Se não for definido o parâmetro "**by =**", a função irá automaticamente selecionar todas as colunas com o mesmo nome para definir uma chave. Veja mais detalhes na seção Combinando colunas para uniões.

4.8.2.2 right_join

Use para unir valores iguais de Y em X. Veja que right_join(Y, X) , é o mesmo que escrever left_join(X, Y). Veja a função left_join para mais informações.

4.8.2.3 inner_join

Use para unir todos os dados de X em Y e retornar somente as linhas em comum.

Por exemplo, se quisermos obter todas as linhas da tabela X (com 50 registros) unindo-as com as linhas correspondentes em Y (com 37 registros), fazemos:

```
inner_join(X, Y)
```

```
# A tibble: 33 x 10
   Population Income Illite~1 Life ~2 Murder HS Gr~3 Frost   Area Estado Abrev~4
        <dbl>  <dbl>    <dbl>   <dbl>  <dbl>   <dbl> <dbl>  <dbl> <chr>  <chr>
 1       3615   3624      2.1    69.0   15.1    41.3    20  50708 Alaba~ AL
 2        365   6315      1.5    69.3   11.3    66.7   152 566432 Alaska AK
 3       2212   4530      1.8    70.6    7.8    58.1    15 113417 Arizo~ AZ
 4       2110   3378      1.9    70.7   10.1    39.9    65  51945 Arkan~ AR
 5        579   4809      0.9    70.1    6.2    54.6   103   1982 Delaw~ DE
 6       8277   4815      1.3    70.7   10.7    52.6    11  54090 Flori~ FL
 7       4931   4091      2      68.5   13.9    40.6    60  58073 Georg~ GA
 8        868   4963      1.9    73.6    6.2    61.9     0   6425 Hawaii HI
 9       2280   4669      0.6    72.6    4.5    59.9   114  81787 Kansas KS
10       3387   3712      1.6    70.1   10.6    38.5    95  39650 Kentu~ KY
# ... with 23 more rows, and abbreviated variable names 1: Illiteracy,
#   2: `Life Exp`, 3: `HS Grad`, 4: Abreviacao
# i Use `print(n = ...)` to see more rows
```

4.8.2.4 full_join

Use para unir os dados de X e Y, mantendo todas as linhas e todas as variáveis.

Para este exemplo, iremos incluir uma linha (add_row) na tabela Y, que não possui um valores correspondente em X.

```
Y <- add_row(Y, Abreviacao = "XX", Estado = "Estado inexistente")
full_join(X, Y)
```

```
# A tibble: 51 x 10
   Population Income Illite~1 Life ~2 Murder HS Gr~3 Frost   Area Estado Abrev~4
        <dbl>  <dbl>    <dbl>   <dbl>  <dbl>   <dbl> <dbl>  <dbl> <chr>  <chr>
 1       3615   3624      2.1    69.0   15.1    41.3    20  50708 Alaba~ AL
 2        365   6315      1.5    69.3   11.3    66.7   152 566432 Alaska AK
```

```
 3       2212   4530      1.8    70.6    7.8    58.1    15 113417 Arizo~ AZ
 4       2110   3378      1.9    70.7   10.1    39.9    65  51945 Arkan~ AR
 5      21198   5114      1.1    71.7   10.3    62.6    20 156361 Calif~ <NA>
 6       2541   4884      0.7    72.1    6.8    63.9   166 103766 Color~ <NA>
 7       3100   5348      1.1    72.5    3.1    56     139   4862 Conne~ <NA>
 8        579   4809      0.9    70.1    6.2    54.6   103   1982 Delaw~ DE
 9       8277   4815      1.3    70.7   10.7    52.6    11  54090 Flori~ FL
10       4931   4091      2      68.5   13.9    40.6    60  58073 Georg~ GA
# ... with 41 more rows, and abbreviated variable names 1: Illiteracy,
#   2: `Life Exp`, 3: `HS Grad`, 4: Abreviacao
# i Use `print(n = ...)` to see more rows
```

Veja que no exemplo acima, temos agora 51 registros, já que uma linha adicional foi criada com o novo registro de Y que não existia em X.

4.8.2.5 Combinando colunas para uniões

Se não for definido o parâmetro "**by =**", a função irá automaticamente selecionar todas as colunas com o mesmo nome para definir uma chave.

Se usarmos by = vetor, por exemplo, by = c("Estado"), estamos definindo a chave de maneira explícita. Podemos definir uma chave contendo mais que uma colunas usando c("coluna1", "coluna2"), com isto, a união será feita encontrando os valores comuns em ambas as colunas das duas tabelas.

Se quisermos definir uma chave que não possui o mesmo nome nas duas tabelas, podemos ainda usar o parametro by = c("coluna de X" = "aa", "coluna de Y" = "bb").

Para exemplificar este caso, vamos renomear a coluna "Estado" da tabela Y para "State" e salvarmos numa tabela Z. Depois iremos usar o parametro by = para definir esta coluna como chave e então podermos efetuar uma união (ex, left_join()).

```
Z <- rename(Y, State = Estado)
left_join (Z, X, by = c("State" = "Estado"))
```

```
# A tibble: 34 x 10
   Abreviacao State   Popul~1 Income Illit~2 Life ~3 Murder HS Gr~4 Frost   Area
   <chr>      <chr>     <dbl>  <dbl>   <dbl>   <dbl>  <dbl>   <dbl> <dbl>  <dbl>
 1 AL         Alabama    3615   3624     2.1    69.0   15.1    41.3    20  50708
 2 AK         Alaska      365   6315     1.5    69.3   11.3    66.7   152 566432
 3 AZ         Arizona    2212   4530     1.8    70.6    7.8    58.1    15 113417
 4 AR         Arkans~    2110   3378     1.9    70.7   10.1    39.9    65  51945
```

```
 5 DE         Delawa~     579   4809     0.9    70.1    6.2    54.6   103   1982
 6 FL         Florida    8277   4815     1.3    70.7   10.7    52.6    11  54090
 7 GA         Georgia    4931   4091     2      68.5   13.9    40.6    60  58073
 8 HI         Hawaii      868   4963     1.9    73.6    6.2    61.9     0   6425
 9 KS         Kansas     2280   4669     0.6    72.6    4.5    59.9   114  81787
10 KY         Kentuc~    3387   3712     1.6    70.1   10.6    38.5    95  39650
# ... with 24 more rows, and abbreviated variable names 1: Population,
#   2: Illiteracy, 3: `Life Exp`, 4: `HS Grad`
# i Use `print(n = ...)` to see more rows
```

```
Z
```

```
# A tibble: 34 x 2
   Abreviacao State
   <chr>      <chr>
 1 AL         Alabama
 2 AK         Alaska
 3 AZ         Arizona
 4 AR         Arkansas
 5 DE         Delaware
 6 FL         Florida
 7 GA         Georgia
 8 HI         Hawaii
 9 KS         Kansas
10 KY         Kentucky
# ... with 24 more rows
# i Use `print(n = ...)` to see more rows
```

> **Dica**
>
> Caso tenhamos colunas com o mesmo nome que não seja a chave, as uniões irão colocar automaticamento um sufixo (.x e .y). Se quiser alterar este sufixo, use o parametro suffix = (ex: suffix c("Tabela_1", "Tabela_2")).

4.8.3 Juntando Observações

Quando tivermos linhas em tabelas diferentes para serem unidas, podemos fazê-lo usndo a função bind_rows(). Em alguns casos, desejamos filtrar linhas de uma tabela, baseada em

linhas em comum de outra tabela, para estes casos, iremos usar as uniões de filtro (filtering joins).

> **Nota**
>
> As tabelas usadas nos exemplos a seguir são as mesmas utilizadas na seção Relacionando Dados.

4.8.3.1 bind_rows

Use para unir observações (linhas) que possuem o mesmo número de colunas.

Por exemplo, usando as tabelas **X** e **Y** preparadas no início da seção Relacionando Dados, podemos criar uma tabela Z, com mais 3 observações e uní-la na tabela Y.

Tabela Z

```
Z <- tibble(
  Abreviacao = c("01", "02", "03"),
  Estado = c("Estado01", "Estado02", "Estado03"))
Z
```

```
# A tibble: 3 x 2
  Abreviacao Estado
  <chr>      <chr>
1 01         Estado01
2 02         Estado02
3 03         Estado03
```

Agora, podemos unir as tabela Y com a tabela Z usando bind_rows():

```
bind_rows(Y, Z)
```

```
# A tibble: 37 x 2
   Abreviacao Estado
   <chr>      <chr>
 1 AL         Alabama
 2 AK         Alaska
 3 AZ         Arizona
 4 AR         Arkansas
 5 DE         Delaware
```

```
 6 FL         Florida
 7 GA         Georgia
 8 HI         Hawaii
 9 KS         Kansas
10 KY         Kentucky
# ... with 27 more rows
# i Use `print(n = ...)` to see more rows
```

> **!** Importante
>
> Observe que, se tivermos alguma coluna que não é comum entre as tabelas, esta coluna será criada automaticamente. Por exemplo:

```
Z <- mutate (Z, Coluna_Extra = c("Valor01", "Valor02", "Valor03"))
bind_rows(Y, Z)
```

```
# A tibble: 37 x 3
   Abreviacao Estado   Coluna_Extra
   <chr>      <chr>    <chr>
 1 AL         Alabama  <NA>
 2 AK         Alaska   <NA>
 3 AZ         Arizona  <NA>
 4 AR         Arkansas <NA>
 5 DE         Delaware <NA>
 6 FL         Florida  <NA>
 7 GA         Georgia  <NA>
 8 HI         Hawaii   <NA>
 9 KS         Kansas   <NA>
10 KY         Kentucky <NA>
# ... with 27 more rows
# i Use `print(n = ...)` to see more rows
```

4.8.3.2 semi_join

Use para retornar todas as linhas da tabela X que são **comuns** na tabela Y.

Por exemplo, para filtrar a tabela Y com apenas os valores presentes na tabela X, usamos:

```
semi_join(Y, X)
```

```
# A tibble: 33 x 2
   Abreviacao Estado
   <chr>      <chr>
 1 AL         Alabama
 2 AK         Alaska
 3 AZ         Arizona
 4 AR         Arkansas
 5 DE         Delaware
 6 FL         Florida
 7 GA         Georgia
 8 HI         Hawaii
 9 KS         Kansas
10 KY         Kentucky
# ... with 23 more rows
# i Use `print(n = ...)` to see more rows
```

4.8.3.3 anti_join

Use para retornar todas as linhas da tabela X que são **NÃO comuns** na tabela Y.

Por exemplo, para filtrar a tabela Y com apenas os valores NÃO presentes na tabela X, usamos:

```
anti_join(Y, X)
```

```
# A tibble: 1 x 2
  Abreviacao Estado
  <chr>      <chr>
1 XX         Estado inexistente
```

4.8.3.4 nest_join

Use para retornar todas as linhas e colunas de X em uma nova coluna contendo todos os valores encontrados em Y.

Por exemplo, para criar uma colunas na tabela Y, contendo os registros encontrados na tabela X que são comuns entre elas, ou seja, que possuem a coluna Estado com valores comuns entre elas:

```
nest_join(Y, X, by = "Estado", name = "Dados de X")
```

```
# A tibble: 34 x 3
   Abreviacao Estado   `Dados de X`
   <chr>      <chr>    <list>
 1 AL         Alabama  <tibble [1 x 8]>
 2 AK         Alaska   <tibble [1 x 8]>
 3 AZ         Arizona  <tibble [1 x 8]>
 4 AR         Arkansas <tibble [1 x 8]>
 5 DE         Delaware <tibble [1 x 8]>
 6 FL         Florida  <tibble [1 x 8]>
 7 GA         Georgia  <tibble [1 x 8]>
 8 HI         Hawaii   <tibble [1 x 8]>
 9 KS         Kansas   <tibble [1 x 8]>
10 KY         Kentucky <tibble [1 x 8]>
# ... with 24 more rows
# i Use `print(n = ...)` to see more rows
```

> Dica
>
> Podemos usar a função tidyr::**unnest**() para desaninhar a coluna.

4.8.3.5 Operações de Definição

O pacote generics, possui também algumas funções que ajudam a identificar uniões (union), intersecções (intersect) e diferenças entre dois vetores. Esta funções podem ser úteis para obter estão operações em dois data-frames.

4.8.3.6 intersect

Use para obter a intersecção de dois conjuntos de dados.

Por exemplo, se quisermos saber quais linhas estão presentes no vetor v (1,2,3,4,5) e w(2,4) , podemos fazer:

```
v <- c(1:5)
w <- c(2,4)

intersect(v,w)
```

```
[1] 2 4
```

4.8.3.7 setdiff

Use para saber quais elementos de v NÃO estão presentes em w.

```
v <- c(1:5)
w <- c(2,4)

setdiff(v,w)
```

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

4.8.3.8 union

Use para fazer a união de dois conjuntos de dados.

```
v <- c(1:5)
w <- c(2,4,"a","b")
union (v,w)
```

```
[1] "1" "2" "3" "4" "5" "a" "b"
```

4.9 Nome de Linhas

Dados organizados (tidy) não usam **nomes** de linhas (que contém uma variável fora das colunas). Para trabalhar com este nomes, mova para uma coluna.

Por exemplo, quando usamos a função **head**() para obter as primeiras linhas da tabela mtcars, obtemos a seguinte resposta:

```
mtcars |>
  head()
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Observe que na lista acima, o nome de cada veículo aparece na saída.

Porém, se pedirmos para mostrar os nomes as variáveis da table usando a função **names**(), não há nenhuma coluna que comporta o nome dos veículos.

Isto acontece porque o nomes dos veículos está associado ao nomes das linhas e não a coluna.

4.9.0.1 rownames_to_column

Se quisermos mover os nomes das linha para uma coluna, usamos a função rownames_to_column do pacote tibble:

```
tibble::rownames_to_column(mtcars, var = "Nomes_Veiculos")
```

	Nomes_Veiculos	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
1	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
2	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
3	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
4	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
5	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
6	Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
7	Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
8	Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
9	Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
10	Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
11	Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
12	Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
13	Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
14	Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
15	Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
16	Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
17	Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
18	Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
19	Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
20	Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
21	Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
22	Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
23	AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
24	Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
25	Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
26	Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
27	Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2

```
28        Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
29      Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
30        Ferrari Dino 19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
31       Maserati Bora 15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
32          Volvo 142E 21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

4.9.0.2 column_to_rownames

Use para nomear as linhas de acordo com uma variável já existente. Digamos que esta função faz o oposto da rownames_to_column, pois ao invés de pegar os nomes das linhas e colocar em uma variável, ela pega uma variável e a transforma em nomes para as linha.

5 Manipulacao de Strings com STRINGR

5.1 Introdução

A seguir temos vários exemplos de manipulação de strings (cadeia de caracteres) utilizando o pacote STRINGR do R. Para saber mais sobre este pacote, acesse:

https://cran.r-project.org/package=stringr.

> **Aviso**
>
> Para melhor utilizar este material, é importante que você tenha uma introdução à linguagem R e saiba carregar pacotes (packages) no R. Para mais informações acesse: https://education.rstudio.com/learn/beginner/.

Para os exemplos, iremos carregar os seguintes pacotes:

- **tidyverse**
- **gt**
- **htmlwidgets**

```
library (tidyverse)
library (gt)
library (htmlwidgets)
```

> **Nota**
>
> **String:** O termo string, ou cadeia de caracteres, é uma **sequência de caracteres** interpretadas como uma constante literal ou até mesmo uma variável.

Por exemplo:

```
minha_string <- "Isto é uma string"
minha_string
```

```
[1] "Isto é uma string"
```

5.1.1 Exemplos da Folha de Referência

A maioria dos exemplos, visam ajudar na interpretação dos exemplos e funções encontradas na **Folha de Referência** do stringr disponível no site do RStudio.

Para a maioria dos exemplos utilizaremos as bases de dados **fruit** (frutas) que será criado a seguir.

FRUIT: Tabela com nome de frutas (em inglês).

```
fruit <- tibble(name = c("Apple", "Apricot", "Avocado", "Banana", "Blackberry", "Blueberry
fruit
```

```
# A tibble: 30 x 1
   name
   <chr>
 1 Apple
 2 Apricot
 3 Avocado
 4 Banana
 5 Blackberry
 6 Blueberry
 7 Cherry
 8 Coconut
 9 Custard-Apple
10 Dragonfruit
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

> **Nota**
>
> *Ao final de algum comando, você verá a chamada à função **gt()**. Isto é apenas para a formatação da tabela de saída e não é necessário para que você entenda os comandos precedentes. Em alguns casos, onde o volume de dados de saída pode ser extenso, usamos*

*também a função **head()** para mostrar apenas as linhas iniciais. Quando o exemplo possui muitas colunas de saída, eventualmente utilizamos a função **select()** para selecionar apenas algumas colunas.*
Em alguns casos usaremos funções de manipulação de dados do pacote **dplyr,** como **mutate** () ou **count**().

i Nota

O termo data-frame descrito ao longo deste texto, é utilizado de forma livre para objetos do tipo data.frame, tibble, entre outros. Pense como se fosse uma tabela de um banco de dados e/ou uma planilha do MS Excel, contendo linhas e colunas. Apesar de não ser rigorosamente igual à uma tabela, muitas vezes usaremos estes termos de forma intercambiável para facilitar o entendimento de iniciantes.

5.2 Detectando Combinações

O pacote stringr possui uma série de funções para identificar a ocorrência ou não de padrões de caracteres (patterns).

Na maioria das vezes o mecanismo de interpretação padrão é o de "Expressão Regular" (regex). Isto significa que podemos construir um padrão de caracteres não somente com letras ou números, mas criando expressões que significam uma combinação mais flexivel no padrão de busca. Para maiores informações veja: Expressões Regulares.

5.2.0.1 str_detect

Use para **detectar** a presença de um **padrão** em uma **string**.

Por exemplo, para detectar quais frutas tem a letra "a", podemos usar:

```
str_detect(fruit$name, "a")
```

```
 [1] FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
[13]  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE
[25]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

No exemplo acima, temos TRUE para todas as linhas que contém a letra "a" e FALSE para aquelas que não tem a letra "a".

> ❗ Importante
>
> Observe que o R é sensível à letras maiúsculas e minúsculas. Como definimos a letra "a" (minúscula) como nosso **padrão** de busca, ele não retorna TRUE para palavras como "Apple" que possui a letra "A" maiúscula.

Se quisermos criar uma coluna ao lado para facilitar a visualização, podemos usar a função **mutate**():

```
fruit |>
  mutate (Padrao_Encontrado = str_detect(fruit$name, "a"))
```

```
# A tibble: 30 x 2
   name          Padrao_Encontrado
   <chr>         <lgl>
 1 Apple         FALSE
 2 Apricot       FALSE
 3 Avocado       TRUE
 4 Banana        TRUE
 5 Blackberry    TRUE
 6 Blueberry     FALSE
 7 Cherry        FALSE
 8 Coconut       FALSE
 9 Custard-Apple TRUE
10 Dragonfruit   TRUE
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

5.2.0.2 str_starts

Use para determinar **se há o padrão definido no início** da string.

Por exemplo, se quisermos identificar quais frutas que começam com o padrão "Bl", usamos:

```
fruit |>
  mutate (Padrao_Encontrado_Inicio = str_starts(fruit$name, "Bl"))
```

```
# A tibble: 30 x 2
   name          Padrao_Encontrado_Inicio
   <chr>         <lgl>
```

```
 1 Apple         FALSE
 2 Apricot       FALSE
 3 Avocado       FALSE
 4 Banana        FALSE
 5 Blackberry    TRUE
 6 Blueberry     TRUE
 7 Cherry        FALSE
 8 Coconut       FALSE
 9 Custard-Apple FALSE
10 Dragonfruit   FALSE
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

Veja que apenas as frutas “Blackberry” e “Bluberry” retornaram verdadeiro (TRUE).

É comum utilizar a função filter() para filtrar apenas as linhas que retornam verdadeiro (TRUE) nas funções de detecção de padrão com o str_detect, str_starts, etc. Veja o exemplo abaixo:

```
fruit |>
  filter (str_starts(name, "Bl"))
```

```
# A tibble: 2 x 1
  name
  <chr>
1 Blackberry
2 Blueberry
```

5.2.0.3 str_which

Use para retornar em **qual linha** o pdrão foi encontrado.

Por exemplo, supondo que o padrão sejam as letras “Bl” (B maiúscula e l minúscula), usamos:

```
str_which(fruit$name, "Bl")
```

```
[1] 5 6
```

Neste exemplo, identificamos que os únicos registros que atendem ao padrão “Bl” estão nas linhas 5 e 6 da tabela.

5.2.0.4 str_locate

Use para **localizar a posição** do padrão na string.

Por exemplo, se criarmos uma coluna contendo onde, em cada nome de fruta, o padrão de busca "er" é encontrado, usamos:

```
fruit |>
  mutate (Localização_na_string = str_locate(name, "er"))
```

```
# A tibble: 30 x 2
   name          Localização_na_string[,"start"] [,"end"]
   <chr>                                   <int>    <int>
 1 Apple                                      NA       NA
 2 Apricot                                    NA       NA
 3 Avocado                                    NA       NA
 4 Banana                                     NA       NA
 5 Blackberry                                  7        8
 6 Blueberry                                   6        7
 7 Cherry                                      3        4
 8 Coconut                                    NA       NA
 9 Custard-Apple                              NA       NA
10 Dragonfruit                                NA       NA
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

A função str_locale retorna **NA** caso o padrão não seja encontrado na string.

> Dica
>
> Ao encontrar o padrão, a str_locale para imediatamente a busca na string. Caso precise encontrar todas as posições que o padrão existir na mesma string, utiliza a **str_locale_all()**.

5.2.0.5 str_count

Use para identificar o número de vez que o padrão foi encontrado na string.

Por exemplo, se buscarmos pelo padrão "na", identificamos que a fruta *banana*, possui o padrão três vezes, enquanto a fruta *pomegranade* apenas uma vez.

```r
fruit |>
  mutate (Vezes_padrao_encontrado = str_count(name, "na"))
```

```
# A tibble: 30 x 2
   name          Vezes_padrao_encontrado
   <chr>                           <int>
 1 Apple                               0
 2 Apricot                             0
 3 Avocado                             0
 4 Banana                              2
 5 Blackberry                          0
 6 Blueberry                           0
 7 Cherry                              0
 8 Coconut                             0
 9 Custard-Apple                       0
10 Dragonfruit                         0
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

5.3 Partes da String

O pacote stringr possui uma série de funções que permitem obter partes de uma string baseado em um padrão de busca. Assim como nas funções de detecção, o interpretador padrão é o regex. Para maiores informações veja: Expressões Regulares.

5.3.0.1 str_sub

Use para **extrair** ou **substituir** partes de uma string a partir de um vetor de caracteres.

Por exemplo, para **extrair** do segundo até o quarto caractere dos nomes das frutas, usamos:

```r
fruit |>
  mutate (Segundo_ao_Quarto_Caractere = str_sub(name, 2, 4))
```

```
# A tibble: 30 x 2
   name          Segundo_ao_Quarto_Caractere
   <chr>         <chr>
 1 Apple         ppl
 2 Apricot       pri
```

```
 3 Avocado       voc
 4 Banana        ana
 5 Blackberry    lac
 6 Blueberry     lue
 7 Cherry        her
 8 Coconut       oco
 9 Custard-Apple ust
10 Dragonfruit   rag
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

Se precisarmos variar o início ou fim da extração de parte da string, podemos passar valores negativos para os parametros start = e/ou end =, fazendo com que a contagem acontece de trás para frente.

5.3.0.2 str_subset

Use para retornar as strings que contém o padrão. É equivalente a fazer str_detect(x, pattern), porém ao invés de retornar verdadeiro ou falso, retorna a string.

```
str_subset(fruit$name, "Bl")
```

```
[1] "Blackberry" "Blueberry"
```

5.3.0.3 str_extract

Use para obter o padrão encontrado na string.

Por exemplo, queremos obter parte da string que atenda ao padrão "erry". Neste caso, a função irá retornar NA para as strings que não contém o padrão e o padrão para aquelas que o contém.

```
fruit |>
  mutate (Str_Extract = str_extract(name, "erry"))
```

```
# A tibble: 30 x 2
   name          Str_Extract
   <chr>         <chr>
 1 Apple         <NA>
 2 Apricot       <NA>
```

```
 3 Avocado       <NA>
 4 Banana        <NA>
 5 Blackberry    erry
 6 Blueberry     erry
 7 Cherry        erry
 8 Coconut       <NA>
 9 Custard-Apple <NA>
10 Dragonfruit   <NA>
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

5.3.0.4 str_match

Use para obter os grupos identificados pelo padrão de busca. Ela retorna uma matriz, onde a primeira coluna retorna a combinação (match) toda e as demais colunas será uma para cada grupo identificado.

Por exemplo, se buscarmos por dois grupos, sendo o primeiro (Ba) e o segundo grupo (na), teremos o seguinte resultado:

```
str_match(fruit$name, "(Ba)(na)") |>
  as_tibble(.name_repair = "unique")
```

```
# A tibble: 30 x 3
   ...1  ...2  ...3
   <chr> <chr> <chr>
 1 <NA>  <NA>  <NA>
 2 <NA>  <NA>  <NA>
 3 <NA>  <NA>  <NA>
 4 Bana  Ba    na
 5 <NA>  <NA>  <NA>
 6 <NA>  <NA>  <NA>
 7 <NA>  <NA>  <NA>
 8 <NA>  <NA>  <NA>
 9 <NA>  <NA>  <NA>
10 <NA>  <NA>  <NA>
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

> **Nota**
>
> Os detalhes sobre grupos nos padrões de busca faz parte das expressões regulares (regex) e estão mais detalhadas na seção: Expressões Regulares

5.4 Gerenciando Tamanho

5.4.0.1 str_length

Use para obter o tamanho da string.

Por exemplo, para obtermos o tamanho das strings correspondentes aos nomes das frutas e adicioná-las em uma coluna chamada "Tamanho", podemos fazer:

```
fruit |>
  mutate(Tamanho = str_length(name))
```

```
# A tibble: 30 x 2
   name          Tamanho
   <chr>           <int>
 1 Apple               5
 2 Apricot             7
 3 Avocado             7
 4 Banana              6
 5 Blackberry         10
 6 Blueberry           9
 7 Cherry              6
 8 Coconut             7
 9 Custard-Apple      13
10 Dragonfruit        11
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

5.4.0.2 str_pad

Use para adicionar espaços em branco ao lado (esquerdo, direito, ambos) da string.

Por exemplo, para adicionar espaços em branco para ajustar em 20 caracteres os nomes das frutas, usamos:

```r
fruit |> mutate(Tamanho = str_pad(name, 20, "left")) |> as.matrix()
```

```
      name            Tamanho
 [1,] "Apple"         "               Apple"
 [2,] "Apricot"       "             Apricot"
 [3,] "Avocado"       "             Avocado"
 [4,] "Banana"        "              Banana"
 [5,] "Blackberry"    "          Blackberry"
 [6,] "Blueberry"     "           Blueberry"
 [7,] "Cherry"        "              Cherry"
 [8,] "Coconut"       "             Coconut"
 [9,] "Custard-Apple" "       Custard-Apple"
[10,] "Dragonfruit"   "         Dragonfruit"
[11,] "Fig"           "                 Fig"
[12,] "Gooseberry"    "          Gooseberry"
[13,] "Grapes"        "              Grapes"
[14,] "Guava"         "               Guava"
[15,] "Jackfruit"     "           Jackfruit"
[16,] "Java Plum"     "           Java Plum"
[17,] "Kiwifruit"     "           Kiwifruit"
[18,] "Lime"          "                Lime"
[19,] "Mango"         "               Mango"
[20,] "MuskMelon"     "           MuskMelon"
[21,] "Olives"        "              Olives"
[22,] "Orange"        "              Orange"
[23,] "Papaya"        "              Papaya"
[24,] "Peach"         "               Peach"
[25,] "Pear"          "                Pear"
[26,] "Pineapple"     "           Pineapple"
[27,] "Pomegranate"   "         Pomegranate"
[28,] "Strawberry"    "          Strawberry"
[29,] "Tamarind"      "            Tamarind"
[30,] "Watermelon"    "          Watermelon"
```

5.4.0.3 str_trunc

Use para truncar a string em um número fixo de caracteres.

Por exemplo, para truncar os nomes das frutas em até 8 caracteres, usamos:

```r
fruit |> mutate(Tamanho = str_trunc(name, 8, "right")) |> as.matrix()
```

```
      name             Tamanho
 [1,] "Apple"          "Apple"
 [2,] "Apricot"        "Apricot"
 [3,] "Avocado"        "Avocado"
 [4,] "Banana"         "Banana"
 [5,] "Blackberry"     "Black..."
 [6,] "Blueberry"      "Blueb..."
 [7,] "Cherry"         "Cherry"
 [8,] "Coconut"        "Coconut"
 [9,] "Custard-Apple"  "Custa..."
[10,] "Dragonfruit"    "Drago..."
[11,] "Fig"            "Fig"
[12,] "Gooseberry"     "Goose..."
[13,] "Grapes"         "Grapes"
[14,] "Guava"          "Guava"
[15,] "Jackfruit"      "Jackf..."
[16,] "Java Plum"      "Java ..."
[17,] "Kiwifruit"      "Kiwif..."
[18,] "Lime"           "Lime"
[19,] "Mango"          "Mango"
[20,] "MuskMelon"      "MuskM..."
[21,] "Olives"         "Olives"
[22,] "Orange"         "Orange"
[23,] "Papaya"         "Papaya"
[24,] "Peach"          "Peach"
[25,] "Pear"           "Pear"
[26,] "Pineapple"      "Pinea..."
[27,] "Pomegranate"    "Pomeg..."
[28,] "Strawberry"     "Straw..."
[29,] "Tamarind"       "Tamarind"
[30,] "Watermelon"     "Water..."
```

> **Dica**
>
> Observe que a função adiciona "..." para identificar as strings que tinham mais que o limite definido". Utilize o parametro ellipsis = "..." para alterar para outros caracteres.

5.4.0.4 str_trim

Use para remover os espaços em brancos do início em final da string.

```
string <- "  Aqui temos espaços em branco no início e no final   "
str_trim(string)
```

```
[1] "Aqui temos espaços em branco no início e no final"
```

5.4.0.5 str_squish

Use para remover espaços em branco no início e final da string e também espaços em brancos repetidos no meio da string.

```
string <- "  Aqui temos espaços em       branco no início, no final e repetidos no meio
str_squish(string)
```

```
[1] "Aqui temos espaços em branco no início, no final e repetidos no meio"
```

5.5 Modificando String

5.5.0.1 str_sub

Use para **extrair** ou **substituir** partes de uma string a partir de um vetor de caracteres.

Por exemplo, para **substituir** do segundo até o quarto caractere dos nomes das frutas, usamos:

```
minha_string <- "Esta é minha string"; minha_string
```

```
[1] "Esta é minha string"
```

```
str_sub(minha_string, 2, 4) <- "XXX"; minha_string
```

```
[1] "EXXX é minha string"
```

> **Dica**
>
> Observe que a função **str_sub não é vetorizada**, ou seja, não recebe ou retorna um vetor como parêmetro.
> Desta forma, se quisermos aplicá-la em conjunto com a função **mutate**(). Uma alternativa para este tipo de situação, é utilziar a função **map**() do **pacote purrr**.

Por exemplo, vamos criar uma função chamada "substitui_string". Esta função irá utilizar a função str_sub() de acordo com os parametros recebidos de str_troca, inicio e fim. Utilizando a função purrr::map() iremos iterar através dos nomes das frutas e utilizar a função substitui_string() para trocar por "XX" os caracteres -2 a -3 de todos os nomes em uma coluna ao lado.

```
substitui_string <- function(str_origin, str_troca, inicio, fim){
  str_sub(str_origin, inicio, fim) <- str_troca
  return (str_origin)
}

fruit |>
  mutate (Segundo_ao_Quarto_Caracteres = purrr::map_chr (name, substitui_string, str_troca
```

```
# A tibble: 30 x 2
   name          Segundo_ao_Quarto_Caracteres
   <chr>         <chr>
 1 Apple         ApXXe
 2 Apricot       ApriXXt
 3 Avocado       AvocXXo
 4 Banana        BanXXa
 5 Blackberry    BlackbeXXy
 6 Blueberry     BluebeXXy
 7 Cherry        CheXXy
 8 Coconut       CocoXXt
 9 Custard-Apple Custard-ApXXe
10 Dragonfruit   DragonfrXXt
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

> **Nota**
>
> Observe que valores negativos para os parametros start = e/ou end =, fazem com que a contagem aconteça de trás para frente.

5.5.0.2 str_replace

Use para substituir partes de uma string por outra string de acordo com o padrão de busca (ex: regex) definido.

> **Dica**
>
> Para saber mais sobre o método de expressão regular (regex) veja: Expressões Regulares e para os outros métodos de interpretação, veja Outras Interpretações.

Por exemplo, vamos definir inicialmente que nosso padrão de busca são as letras "er". Agora vamos substituir este padrão pela string " XX " colocando em uma coluna ao lado usando a função mutate().

```
fruit |>
  mutate (nomes_substituidos = str_replace(name, "er", " XX "))
```

```
# A tibble: 30 x 2
   name          nomes_substituidos
   <chr>         <chr>
 1 Apple         Apple
 2 Apricot       Apricot
 3 Avocado       Avocado
 4 Banana        Banana
 5 Blackberry    Blackb XX ry
 6 Blueberry     Blueb XX ry
 7 Cherry        Ch XX ry
 8 Coconut       Coconut
 9 Custard-Apple Custard-Apple
10 Dragonfruit   Dragonfruit
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

> **Nota**
>
> Note que diferente da função str_sub(), a função str_replace() é vetorizada, com isto não precisamos utilizar o purrr:map para retornar um vetor.

5.5.0.3 str_replace_all

Use para substituir partes de uma string por outra string de acordo com o padrão de busca (ex: regex) definido em TODAS as vezes que o padrão for encontrado.

Por exemplo, vamos definir inicialmente que nosso padrão de busca são as letras "na". Agora vamos substituir este padrão pela string " XX " colocando em uma coluna ao lado usando a função mutate().

```
fruit |>
  mutate (nomes_substituidos = str_replace_all(name, "an", " XX "))
```

```
# A tibble: 30 x 2
   name          nomes_substituidos
   <chr>         <chr>
 1 Apple         Apple
 2 Apricot       Apricot
 3 Avocado       Avocado
 4 Banana        B XX  XX a
 5 Blackberry    Blackberry
 6 Blueberry     Blueberry
 7 Cherry        Cherry
 8 Coconut       Coconut
 9 Custard-Apple Custard-Apple
10 Dragonfruit   Dragonfruit
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

Note que se tivessemo utilizado a função str_replace ao invés da str_replace_all, a palavra "**Banana**" retornaria "B**a XX na**", pois ela substituiria apenas a primeira vez que o padrão fosse encontrado.

5.5.0.4 str_to_lower

Use para colocar a string em letras minúsculas.

```
fruit |>
  mutate (tolowe = str_to_lower(name))
```

```
# A tibble: 30 x 2
   name          tolowe
   <chr>         <chr>
 1 Apple         apple
 2 Apricot       apricot
 3 Avocado       avocado
 4 Banana        banana
 5 Blackberry    blackberry
 6 Blueberry     blueberry
 7 Cherry        cherry
 8 Coconut       coconut
 9 Custard-Apple custard-apple
10 Dragonfruit   dragonfruit
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

5.5.0.5 str_to_upper

Use para colocar a string em letras maiúsculas.

```
fruit |>
  mutate (toupper = str_to_upper(name))
```

```
# A tibble: 30 x 2
   name          toupper
   <chr>         <chr>
 1 Apple         APPLE
 2 Apricot       APRICOT
 3 Avocado       AVOCADO
 4 Banana        BANANA
 5 Blackberry    BLACKBERRY
 6 Blueberry     BLUEBERRY
 7 Cherry        CHERRY
 8 Coconut       COCONUT
 9 Custard-Apple CUSTARD-APPLE
10 Dragonfruit   DRAGONFRUIT
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

5.5.0.6 str_to_title

Use para colocar a string com a primeira letra maiúscula e as demais em letras minúsculas de cada palavra.

```
  fruit |>
    mutate (totitle= str_to_title(name))
```

```
# A tibble: 30 x 2
   name          totitle
   <chr>         <chr>
 1 Apple         Apple
 2 Apricot       Apricot
 3 Avocado       Avocado
 4 Banana        Banana
 5 Blackberry    Blackberry
 6 Blueberry     Blueberry
 7 Cherry        Cherry
 8 Coconut       Coconut
 9 Custard-Apple Custard-Apple
10 Dragonfruit   Dragonfruit
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

5.6 Juntando e Dividindo

5.6.0.1 str_c

Use para juntar várias strings em uma única string.

Para exemplificar, vamos criar uma segunda coluna em nossa tabela de frutas.

```
  # Nova coluna com uma string qualquer
  col_nova <- bind_cols(c(letters, LETTERS), seq(1:52), seq(1:52), c(letters, LETTERS))
  col_nova <- col_nova |>
    unite(nova_string, names(col_nova)) |>
    slice (n = 1:30)
  frutas <- bind_cols (fruit, col_nova)

  #Concatenando ambas colunas
    frutas |>
```

```
    mutate ( str_c = str_c(name, nova_string))
```

```
# A tibble: 30 x 3
   name          nova_string str_c
   <chr>         <chr>       <chr>
 1 Apple         a_1_1_a     Applea_1_1_a
 2 Apricot       b_2_2_b     Apricotb_2_2_b
 3 Avocado       c_3_3_c     Avocadoc_3_3_c
 4 Banana        d_4_4_d     Bananad_4_4_d
 5 Blackberry    e_5_5_e     Blackberrye_5_5_e
 6 Blueberry     f_6_6_f     Blueberryf_6_6_f
 7 Cherry        g_7_7_g     Cherryg_7_7_g
 8 Coconut       h_8_8_h     Coconuth_8_8_h
 9 Custard-Apple i_9_9_i     Custard-Applei_9_9_i
10 Dragonfruit   j_10_10_j   Dragonfruitj_10_10_j
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

> **Dica**
>
> Use o parametro **sep** = para definir um caractere de separação quando juntar as strings se desejar.

5.6.0.2 str_flatten

Use para "achatar" o vetor de string. O parametro collapse = " " pode ser alterado para incluir um caractere específico enquanto ocorre o processo.

Por exemplo, temos uma string "Bom dia". Neste caso, temos um vetor de caracteres de tamanho 7 ("B" "o" "m" " " "d" "i" "a"). A função flatten irá achatar este vetor e retornar apenas uma string com um único vetor ("Bom dia").

```
minha_string <- c("B","o","m"," ","d","i","a")
length (minha_string); minha_string
```

```
[1] 7
```

```
[1] "B" "o" "m" " " "d" "i" "a"
```

```r
minha_string <- str_flatten(minha_string)
length (minha_string); minha_string
```

```
[1] 1
```

```
[1] "Bom dia"
```

5.6.0.3 str_dup

Use para duplicar uma string determinado número de vezes.

```r
fruit |>
  mutate (str_dup = str_dup(name, 3))
```

```
# A tibble: 30 x 2
   name          str_dup
   <chr>         <chr>
 1 Apple         AppleAppleApple
 2 Apricot       ApricotApricotApricot
 3 Avocado       AvocadoAvocadoAvocado
 4 Banana        BananaBananaBanana
 5 Blackberry    BlackberryBlackberryBlackberry
 6 Blueberry     BlueberryBlueberryBlueberry
 7 Cherry        CherryCherryCherry
 8 Coconut       CoconutCoconutCoconut
 9 Custard-Apple Custard-AppleCustard-AppleCustard-Apple
10 Dragonfruit   DragonfruitDragonfruitDragonfruit
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

5.6.0.4 str_split_fixed

Use para "quebrar" um string em partes. A funçao **str_split**() retorna uma string dividida enquanto a funçao **str_split_fixed**() retorna uma matriz de caracteres com número fixo de colunas.

Por exemplo, vamos usar a mesma tabela usada no exemplo da funçao str_c chamada frutas (na fruit). Iremos "quebrar as strings da coluna"nova_string" usando oseparator "_". Como temos exatamente o mesmo número de sperador em todas as strings, a funções irá nos retornar um vetor de caracteres de tamanho 4.

```r
frutas |>
  mutate (str_split = str_split(nova_string, "_")) |>
  pull(str_split)
```

```
[[1]]
[1] "a" "1" "1" "a"

[[2]]
[1] "b" "2" "2" "b"

[[3]]
[1] "c" "3" "3" "c"

[[4]]
[1] "d" "4" "4" "d"

[[5]]
[1] "e" "5" "5" "e"

[[6]]
[1] "f" "6" "6" "f"

[[7]]
[1] "g" "7" "7" "g"

[[8]]
[1] "h" "8" "8" "h"

[[9]]
[1] "i" "9" "9" "i"

[[10]]
[1] "j"  "10" "10" "j"

[[11]]
[1] "k"  "11" "11" "k"

[[12]]
[1] "l"  "12" "12" "l"

[[13]]
```

```
[1] "m"  "13" "13" "m"

[[14]]
[1] "n"  "14" "14" "n"

[[15]]
[1] "o"  "15" "15" "o"

[[16]]
[1] "p"  "16" "16" "p"

[[17]]
[1] "q"  "17" "17" "q"

[[18]]
[1] "r"  "18" "18" "r"

[[19]]
[1] "s"  "19" "19" "s"

[[20]]
[1] "t"  "20" "20" "t"

[[21]]
[1] "u"  "21" "21" "u"

[[22]]
[1] "v"  "22" "22" "v"

[[23]]
[1] "w"  "23" "23" "w"

[[24]]
[1] "x"  "24" "24" "x"

[[25]]
[1] "y"  "25" "25" "y"

[[26]]
[1] "z"  "26" "26" "z"

[[27]]
[1] "A"  "27" "27" "A"
```

```
[[28]]
[1] "B"  "28" "28" "B"

[[29]]
[1] "C"  "29" "29" "C"

[[30]]
[1] "D"  "30" "30" "D"
```

Se quisermos "quebrar" uma string usando um separador e já gerarmos as respectivas colunas em uma tabela, podemos usar a função str_split_fixed(), extrairmos as respectivas matrizes e adicionarmos como colunas na tabela, podemos fazer: .

5.6.0.5 str_glue

Use para interpolar/formatar uma string.

Por exemplo, se tivermos duas strings: s1 = "Fulano" e s2 = "da Silva". Podemos "colar" estas strings usando a função **str_glue**().

```
s1 <- "Fulano"
s2 <- "da Silva"
str_glue("{s1}"," ", "{s2}")
```

```
Fulano da Silva
```

Podemos também juntar strings fixas e variáveis como no exemplo abaixo.

```
s1 <- "Fulano"
s2 <- "da Silva"
str_glue("Meu nome é {s1}"," ", "{s2}")
```

```
Meu nome é Fulano da Silva
```

5.6.0.6 str_glue_data

É similar a função str_glue, mas adequada a objetos de dados.

Por exemplo, vamos juntar o nome das frutas, o nome da linha da tabela que ela está e mais uma string fixa:

```
fruit |> str_glue_data("A {name} é uma fruta.")
```

```
A Apple é uma fruta.
A Apricot é uma fruta.
A Avocado é uma fruta.
A Banana é uma fruta.
A Blackberry é uma fruta.
A Blueberry é uma fruta.
A Cherry é uma fruta.
A Coconut é uma fruta.
A Custard-Apple é uma fruta.
A Dragonfruit é uma fruta.
A Fig é uma fruta.
A Gooseberry é uma fruta.
A Grapes é uma fruta.
A Guava é uma fruta.
A Jackfruit é uma fruta.
A Java Plum é uma fruta.
A Kiwifruit é uma fruta.
A Lime é uma fruta.
A Mango é uma fruta.
A MuskMelon é uma fruta.
A Olives é uma fruta.
A Orange é uma fruta.
A Papaya é uma fruta.
A Peach é uma fruta.
A Pear é uma fruta.
A Pineapple é uma fruta.
A Pomegranate é uma fruta.
A Strawberry é uma fruta.
A Tamarind é uma fruta.
A Watermelon é uma fruta.
```

5.7 Ordenando String

5.7.0.1 str_order

Use para sequenciar um vetor de caracteres.

Por exemplo, para colocar em sequência de forma decrescente os nomes da frutas, podemos usar:

```
    str_order(fruit$name, decreasing = TRUE)
```

```
 [1] 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6
[26]  5  4  3  2  1
```

O resultado será uma sequência (descrescente) em que cada item do vetor está.

5.7.0.2 str_sort

Use para ordenar um vetor de caracteres.

Por exemplo, para ordenar de forma decrescente os nomes da frutas, podemos usar:

```
  str_sort(fruit$name, decreasing = TRUE) |>
    as_tibble(.name_repair = "unique")
```

```
# A tibble: 30 x 1
   value
   <chr>
 1 Watermelon
 2 Tamarind
 3 Strawberry
 4 Pomegranate
 5 Pineapple
 6 Pear
 7 Peach
 8 Papaya
 9 Orange
10 Olives
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

5.8 Auxiliares

5.8.0.1 str_conv

Use para converter o "encode" de uma string.

```
x <- rawToChar(as.raw(177))
x
```

```
[1] "\xb1"
```

```
str_conv(x, "ISO-8859-2") # Polones a com cedilha"
```

```
[1] "ą"
```

```
str_conv(x, "ISO-8859-1") # Mais-Menos
```

```
[1] "±"
```

5.8.0.2 str_view_all

Use para ver os valores encontrados na string de acordo com um padrão de busca.

Por exemplo, se tivermos o padrão de busca como "er", podemos ver onde na strings ele é encontrado.

```
str_view(c("Banana", "Blueberry", "Blackberry"), "er")
```

> **Nota**
>
> A função **str_view_all**(), mostrará **todos** os encontros na string, se quiser para a busca do padrão no **primeiro** encontro, usa **str_view**().

5.8.0.3 str_wrap

Use para formatar uma string em parágrafos.

```
thanks_path <- file.path(R.home("doc"), "THANKS")
thanks <- str_c(readLines(thanks_path), collapse = "\n")
thanks <- word(thanks, 1, 3, fixed("\n\n"))
cat(str_wrap(thanks, width = 60, indent = 2), "\n")
```

```
  R would not be what it is today without the invaluable
help of these people outside of the (former and current)
R Core team, who contributed by donating code, bug fixes
and documentation: Valerio Aimale, Suharto Anggono, Thomas
Baier, Gabe Becker, Henrik Bengtsson, Roger Bivand, Ben
Bolker, David Brahm, G"oran Brostr"om, Patrick Burns,
Vince Carey, Saikat DebRoy, Matt Dowle, Brian D'Urso,
Lyndon Drake, Dirk Eddelbuettel, Claus Ekstrom, Sebastian
Fischmeister, John Fox, Paul Gilbert, Yu Gong, Gabor
Grothendieck, Frank E Harrell Jr, Peter M. Haverty,
Torsten Hothorn, Robert King, Kjetil Kjernsmo, Roger
Koenker, Philippe Lambert, Jan de Leeuw, Jim Lindsey,
Patrick Lindsey, Catherine Loader, Gordon Maclean, Arni
Magnusson, John Maindonald, David Meyer, Ei-ji Nakama,
Jens Oehlschl"agel, Steve Oncley, Richard O'Keefe, Hubert
Palme, Roger D. Peng, Jose' C. Pinheiro, Tony Plate, Anthony
Rossini, Jonathan Rougier, Petr Savicky, Guenther Sawitzki,
Marc Schwartz, Arun Srinivasan, Detlef Steuer, Bill Simpson,
Gordon Smyth, Adrian Trapletti, Terry Therneau, Rolf Turner,
Bill Venables, Gregory R. Warnes, Andreas Weingessel, Morten
Welinder, James Wettenhall, Simon Wood, and Achim Zeileis.
Others have written code that has been adopted by R and is
acknowledged in the code files, including
```

5.9 Expressões Regulares

Padrões de buscas são interpretados na funções do pacote stringr como Expressões Regulares (regex). Ou seja, quando uma função possui o parametro pattern =, significa que o interpretador irá entendem como uma expressão regular. Você pode alterar o interpretadores para outros tipos se necessário. Para saber mais sobre isso, acesse Outras Interpretações.

Expressão Regular é uma sequencia de caracteres que especificam um padrão de busca em uma string.

No R, você escreve uma expressão regular como um string, ou seja, uma sequencia de caracteres entre asps simples ’ ou duplas “.

Alguns caracteres de uma expressão regular não podem ser representados diretamente como uma string no R. Estes são conhecidos como caracteres especiais e são uma sequencia de caracteres que tem um significado específico.

Por exemplo:

Caracteres Especiais	Representa
\\	\
\"	"
\?	?

> **Dica**
>
> Para obter a lista completa, digite ? “”’.

Devido a isto, sempre que aparecer uma barra invertida (\ ) em uma expressão regular, você deve digitar duas barras (\\ ) na strings da expressão.

Isto é uma particularidade do R e outras linguagens isto pode não ser necessário.

Use a função **writeLines**() para ver como o R vê sua string depois dos caracteres especiais forem lido.

```
writeLines("\\.")
```

\.

```
writeLines("\\")
```

\

Como exemplo inicial, vamos utilizar a função **str_extract**() que recebe um string como parametro e também aceita o padrão de busca como outro parametro.

Iremos definir nosso **padrão** de busca como a letra “a”. Desta forma, se passarmos para a função str_detect a string “Banana” e o padrão “a”, ele deve retornar a letra “a”, pois a string Banana possui a letra “a”.

```
str_extract ("Banana", "a")
```

```
[1] "a"
```

Por outro lado, se passarmos a string “Fig” com o mesmo padrão de busca, teremos NA como retorno, pois a string “Fig” não possui a letra “a”.

```
str_extract ("Fig", "a")
```

```
[1] NA
```

5.9.1 Combinando Caracteres

No exemplo anterior utilizamos apenas um caractere como padrão de busca, no caso a letra “a”.

Quando desejamos combinar diversos caracteres (letras, numeros, simbolos, espaços, etc) utilizamos a expressões na tabela abaixo:

Para facilitar o entendimento, utilizaremos uma string com letras maiúsculas, minúsculas, símbolos e números:

```
Str_Teste <- "abc ABC 123\t.!?\\(){}\n"
Str_Teste
```

```
[1] "abc ABC 123\t.!?\\(){}\n"
```

String Regex no R	Busca por
a	a (etc.)
\\.	.
\\!	\!
\\?	\?
\\\\	\\
\\(	\(
\\)	\)
\\{	\{
\\}	\}
\\n	nova linha (ENTER)
\\t	TAB
\\s	qualquer caractere em branco
\\d	qualquer digito

String Regex no R	Busca por
\\w	qualquer letra
\\b	barra de espaço
[:digit:]	digitos
[:alpha:]	letras
[:lower:]	letras minúsculas
[:upper:]	letras maiúsculas
[:alnum:]	letras e números
[:punct:]	pontuação
[:graph:]	letras, números e pontuação
[:space:]	qualquer espaço em branco
[:blank:]	espaço em branco e barra de espaço (mas não nova linha)
.	qualquer caractere exceto nova linha (ENTER)

Vamos mostrar como usar a tabela acima com alguns exemplos. Para isso, iremos usar a string criada anteriormente chamada "Str_Teste".

Exemplo 1:

Vamos buscar em nossa string de teste (Str_Teste) a letra minúscula "a".

Na coluna da tabela acima chamada **String**, encontramos oque devemos digitar para construir o padrão de busca. Neste caso, seria "a".

Se usarmos a função **str_view_all**() pssando nossa "Str_Teste" e o padrão de busca "a", observamos que teremos marcado apenas a letra "a" na string. Isto significa que o padrão de busca foi encontrado na string.

```
str_view_all (Str_Teste, "a")
```

Exemplo 2:

Vamos buscar agora pelo padrão do símbolo de ponto de interrogação "?". Similar ao exemplo anterior, vemos que apenas o ponto de interrogação foi encontrado.

```
str_view_all (Str_Teste, "\\?")
```

Exemplo 3:

Vamos criar agora um padrão que busque por todos os digitos em nossa string.

```
str_view_all (Str_Teste, "\\d")
```

> **Dica**
>
> Para buscarmos pelo inverso do caso anterior, ou seja, todos os caracteres que NÃO são digitos, usamos a letra "D" maiúscula. Isto é válido também para os casos de "\\S" e "\\W" que seriam o inverso de "\\s" e "\\w" respectivamente.

```
str_view_all (Str_Teste, "\\D")
```

Exemplo 4:

Vamos criar agora um padrão que busque por todos os digitos e letras em nossa string.

```
str_view_all (Str_Teste, "[:alnum:]")
```

5.9.2 Quantificadores

Agora que já saber como criar padrões de busca para identificar diversos tipos de caracteres, veremos como difinir a quantidade desses caracteres em nosso padrão. Veja a tabela abaixo:

Regex	Busca
?	Zero ou um
*****	Zero ou mais
+	Um ou mais
{n}	Exatamente **n**
{n,}	**n** ou mais
{n,m}	Entre **n** e **m**

Vamos ver como utilizamos estes quantificadores juntamente com os caracteres especiais vistos anteriormente (ver Combinando Caracteres).

Para os exemplos a seguir utilizaremos a seguinte string de teste: Str_Teste_2 = ".a.aa.aaa"

```
Str_Teste_2 <- ".a.aa.aaa"
```

Exemplo 1:

Digamos que queremos buscar em nossa string de teste "Str_Teste" a letra "a" zero ou uma vez, para isso faremos:

```
str_view_all(Str_Teste_2, "a?")
```

Neste caso, todas as vezes que a funções encontrar a letra “a” zero ou uma vez, elá irá marcar.

> **Aviso**
>
> Se usarmos a função **str_view**() ela irá utilizar o padrão apenas até o primeiro encontro e depois irá para a busca, veja:

```
str_view(Str_Teste_2, "a?")
```

Observe que a busca para logo no primeiro caractere, pois estamos buscando pela letra “a” **ZERO** ou mais vezes.

Exemplo 2:

Agora vamos iremos buscar pela letra “a” **UMA** ou mais vezes, porém iremos utilizar a função str_view() ou invés da str_view_all(), parando a busca assim que o primeiro encontro ocorra:

```
str_view(Str_Teste_2, "a+")
```

Exemplo 3:

Neste exemplo, queremos criar um padrão de busca pela letra “a”, mas que ela ocorra **DUAS** a **TRÊS** vezes.

```
str_view(Str_Teste_2, "a{2,3}")
```

Veja que ele localizou apenas as duas letras “aa” e não marcou as letras “aaa”. Isto é porque utilizamos a função str_view(), que parou a busca assim que a primeiro encontro ocorreu. Se quisermos continuar a busca, devemos utilizar a função str_view_all().

```
str_view_all(Str_Teste_2, "a{2,3}")
```

Exemplo 4:

Neste exemplo, usaremos a tabela frutas, criada quando descrevemos a função str_c. Veja com ela era para se recordar:

```
frutas |>
  head()
```

```
# A tibble: 6 x 2
  name       nova_string
  <chr>      <chr>
1 Apple      a_1_1_a
2 Apricot    b_2_2_b
3 Avocado    c_3_3_c
4 Banana     d_4_4_d
5 Blackberry e_5_5_e
6 Blueberry  f_6_6_f
```

Digamos que precisamos extrair apenas os numeros da coluna "nova_string". E colocá-los em uma nova coluna chamda "numeros".

Neste caso, podemos usar a função str_extract() com um padrão que encontre um número de **0 até 9**, seguido por **um ou mais** "qualquer caractere" e depois outro número de 0 até 9.

Este padrão irá encontrar padrões como "1_1" ou "2_2".

Em seguida, usamos um outro padrão **[:punct:]** na função str_remove para remover a pontuação.

```
frutas |>
  mutate (numeros = str_extract(nova_string, "[0-9].+[0-9]")) |>
  mutate (numeros = str_remove(numeros, "[:punct:]"))
```

```
# A tibble: 30 x 3
   name          nova_string numeros
   <chr>         <chr>       <chr>
 1 Apple         a_1_1_a     11
 2 Apricot       b_2_2_b     22
 3 Avocado       c_3_3_c     33
 4 Banana        d_4_4_d     44
 5 Blackberry    e_5_5_e     55
 6 Blueberry     f_6_6_f     66
 7 Cherry        g_7_7_g     77
 8 Coconut       h_8_8_h     88
 9 Custard-Apple i_9_9_i     99
10 Dragonfruit   j_10_10_j   1010
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

5.9.3 Alternadores

Até aqui, utilizamos os caracteres especiais (Combinando Caracteres) e sabemos como localizá-los em diversas quantidades (Quantificadores). Mas em muitos casos precisamos organizá-los de forma lógica, possibilitando utilizá-los em combinações mais flexíveis. Para isto, utilizamos os símbolos de alternadores, veja:

Para os exemplos a seguir utilizaremos a seguinte string de teste: Str_Teste_3 = "abcde"

Regex	Busca
|	OU
[ ]	Um dos
[^]	Tudo exceto
[-]	Range

```
Str_Teste_3 <- "abcde"
```

Exemplo 1:

Digamos que desejamos criar um padrão que busque pela letras "ab" **OU** a letra "d", para isto podemos usar:

```
str_view_all(Str_Teste_3, "ab|d")
```

Exemplo 2:

Digamos que desejamos criar um padrão que busque qualquer **um dos** caracteres "abe", para isto podemos usar:

```
str_view_all(Str_Teste_3, "[abe]")
```

Exemplo 3:

Digamos que desejamos criar um padrão que busque qualquer um **range** de letras entre as letras "a" até a "c", para isto podemos usar:

```
str_view_all(Str_Teste_3, "[a-c]")
```

Exemplo 4:

Neste exemplo, usaremos novamente a tabela **frutas**, criada quando descrevemos a função str_c.

Digamos que precisamos filtrar nesta tabela, apenas as frutas que possuem nomes compostos, ou seja, separados por espaço ou uma pontuação (ex "-").

Podemos usar a função filter() passando o resultado da função str_detect() junto com um padrão. Há diversas maneiras de construir este padrão. Aqui optar por buscar por caracteres alfa-numéricos (letras e números) e usamos o alternador [^] para negar tais caracteres, portanto, iremos identificar se a string NÃO possui letras ou números.

```
frutas |>
  filter (str_detect(name, "[^[:alnum:]]"))
```

```
# A tibble: 2 x 2
  name          nova_string
  <chr>         <chr>
1 Custard-Apple i_9_9_i
2 Java Plum     p_16_16_p
```

5.9.4 Ancoragem

Para definir se a sequência do padrão de busca está no início ou fim da string, utilizamos as expressões de ancoragem:

Regex	Busca
^	Início da string
$	Fim da string

Para os exemplos a seguir utilizaremos a seguinte string de teste: Str_Teste_4 = "aaa"

```
Str_Teste_4 <- "aaa"
```

Exemplo 1:

Para criar um padrão que busque a letra "a" apenas no **fim** da string, usamos:

```
str_view_all(Str_Teste_4, "a$")
```

Exemplo 2:

Para criar um padrão que busque a letra "a" apenas no **início** da string, usamos:

```
str_view_all(Str_Teste_4, "^a")
```

Exemplo 3:

Neste exemplo, usaremos novamente a tabela **frutas**, criada quando descrevemos a função str_c.

Digamos que queremos filtrar apenas as frutas que **terminem** com a letra "**a**" **E** também que terminem com a letra "**o**". Podemos fazer:

```
frutas |>
  filter (str_detect(name, "[ao]$"))
```

```
# A tibble: 5 x 2
  name    nova_string
  <chr>   <chr>
1 Avocado c_3_3_c
2 Banana  d_4_4_d
3 Guava   n_14_14_n
4 Mango   s_19_19_s
5 Papaya  w_23_23_w
```

5.9.5 Grupos

Você pode utilizar **parênteses ( )** para definir expressões de **precedência** ou para serem **referenciados** posteriormente através da **ordem de criação**.

Para os exemplos a seguir utilizaremos a seguinte string de teste: Str_Teste_5 = "abbaab"

```
Str_Teste_5 <- "abbaab"
```

Exemplo 1:

Digamos que tenhamos a string "Blueberry" e você queira criar um padrão que busque pela letra "e" precedida das letras "lu" OU "b".

Neste caso, devemos criar um grupo de precedência para "lu" OU "b". Para isto iremos colocar esta parte da expressão entre parênteses (lu|b). agora podemos utilzar este grupo e concluir o padrão de busca conforme a seguir:

```
str_view_all("Blueberry", "(lu|b)e")
```

Veja que se nossa string fosse “Blueberry is special”, a letra “e” de “special” não seria encontrada:

```
str_view_all("Blueberry is special", "(lu|b)e")
```

Se quisermos criar um padrão que encontre a letra “e” precedida de qualquer letra, podemos fazer:

```
str_view_all("Blueberry is special", "([:alpha:])e")
```

Exemplo 2:

Ao criar um grupo, como vimos no exemplo anterior, podemos fazer referência à este grupo usando ***n*,** on ***n*** é a ordem de criação do grupo.

Por exemplo, digamos que criamos um grupo utilizando os parênteses ( ) que contenha apenas letra “a”. Seu código ficaria **(a)**, e ele poderia ser referenciado com **\\1**, pois foi o primeiro grupo a ser criado.

Digamos que agora, você crie um segundo grupo com a letra “b”, seu código ficaria **(b)** e poderia ser referenciado com **\\2**.

Sabendo como criar os grupos e como referênciá-los, podemos montar um padrão de busca utilizando tanto os grupos quanto suas referência. Veja este exemplo:

```
str_view_all(Str_Teste_5, "(a)(b)\\2\\1")
```

Neste exemplo, nosso padrão busca por “ba”, atraveś de \\2\\1, desde que tenham precedência de “ab”, através dos grupos (a)(b).

Exemplo 3:

Digamos que tenhamos a string “Tem uma banana na mesa”. Queremos criar uma padrão que busque as letras “nana”. Apesar de termos soluções mais simples, poderíamos criar um grupo contendo “na” e usar a ordem de referência para concluir a expressão:

```
str_view_all("Tem uma banana na mesa", "(na)\\1")
```

Exemplo 4:

Neste exemplo, usaremos novamente a tabela **frutas**, criada quando descrevemos a função str_c.

Aqui iremos obter o mesmo resultado para o Problema 4 descrito na seção Quantificadores.

Porém agora vamos usar a função **str_replace**() e o suporte à **grupos** que acabamos de ver para atingir o mesmo resultado, ou seja, extrair apenas os números da coluna nova_string.

```
frutas |>
  mutate (numeros = str_replace(nova_string, ".+([0-9]).?([0-9]).+", "\\1\\2"))
```

```
# A tibble: 30 x 3
   name          nova_string numeros
   <chr>         <chr>       <chr>
 1 Apple         a_1_1_a     11
 2 Apricot       b_2_2_b     22
 3 Avocado       c_3_3_c     33
 4 Banana        d_4_4_d     44
 5 Blackberry    e_5_5_e     55
 6 Blueberry     f_6_6_f     66
 7 Cherry        g_7_7_g     77
 8 Coconut       h_8_8_h     88
 9 Custard-Apple i_9_9_i     99
10 Dragonfruit   j_10_10_j   10
# ... with 20 more rows
# i Use `print(n = ...)` to see more rows
```

Detalhes do exemplo acima:

Observe que a função **str_replace**(), recebe dois padrões, sendo o primeiro de busca e o segundo daquilo que iremos substituir o primeiro.

Neste caso, nosso padrão de busca, encontra "qualquer caractere" "uma ou mais vezes", depois cria um "grupo com números de 0 a 9", seguido por "qualquer caractere zero ou uma vez" e depois cria o segundo "grupo com números de 0 à 9" e conclui com "qualquer caractere uma ou mais vezes".

Com nosso padrão de busca criado, iremos criar nosso padrão de substituição, ou seja, aquilo que for encontrado pelo padrão de busca, será substituído pelo padrão de substituição.

Nosso padrão de substituição ficou simples ("\\1\\2"). Veja que ele apenas pega o conteúdo do grupo 1 e grupo 2 criados no padrão de busca usando parênteses para substituir.

5.9.6 Pesquisa ao Redor

Em alguns casos, precisamos criar um padrão que **olhe ao redor** para encontrar o que buscamos.

Há símbolos para definirmos grupos que estão **precedendo** o que buscamos e há símbolos para definirmos grupos que estão **posteriores** ao que buscamos. Há também símbolos para negar os casos anteriores e posteriores.

Veja a tabela:

Para os exemplos a seguir utilizaremos a seguinte string de teste: Str_Teste_6 = "bacad"

Regex	Busca
(**?=**)	Seguido por
(**?!**)	Não seguido por
(**?<=**)	Precedido por
(**?<!**)	Não precedido por

```
Str_Teste_6 <- "bacad"
```

Exemplo 1:

Vamos criar um padrão de busca que localize a letra "a", mas queremos a(s) letra(s) "a" que são seguidas apenas pela letra "c".

Para isso iremos criar um grupo ("c"), mas como é um grupo que irá seguir aquilo que buscamos, ao invés dos parêntese apenas, iremos utilizar o símbolo da tabela anterior "?="a fazer (?=c). Depois adicionamos a busca pela letra "a".

```
str_view_all(Str_Teste_6, "a(?=c)")
```

Exemplo 2:

Vamos criar um padrão de busca que localize a letra "a", mas queremos a(s) letra(s) "a" que são precedidas pela letra "b". Usando a mesma tabela e raciocínio do exemplo anterior, podemos criar o grupo com a letra b, mas como é um grupo de precedência, temos que adicionar os símbolos "?<=" e fazer:

```
str_view_all(Str_Teste_6, "(?<=b)a")
```

Exemplo 4:

Neste exemplo, usaremos novamente a tabela **frutas**, criada quando descrevemos a função str_c.

Digamos que iremos filter as frutas que comecem com as letras "B" e "P" se forem seguidas das letras "e" e "l". Desta forma, não teremos na saída frutas como "Banana" ou "Pineapple"

```r
frutas |>
  filter(str_detect(name, "^[BP](?=[el])"))
```

```
# A tibble: 4 x 2
  name       nova_string
  <chr>      <chr>
1 Blackberry e_5_5_e
2 Blueberry  f_6_6_f
3 Peach      x_24_24_x
4 Pear       y_25_25_y
```

5.10 Outras Interpretações

5.10.0.1 regex

Conforme visto na seção Expressões Regulares, o interpretador padrão das funções do pacote stringr, é o regex, ou seja, sempre que tivermos o parâmetro pattern =, se não especificarmos nada, ele irá interpretar a string deste parêmetro como se fosse uma expressão regular (regex).

A seguir, veremos como mudar este padrão e introduzir outros interpretadores disponíveis.

5.10.0.2 fixed

Para buscar bytes nativos (raw), podemos usar o interpretador fixed(). Esta opção é bastante rápida, mas pode perder alguns caracteres que podem estar representados de maneiras diferentes (ex não ASCII).

Exemplo:

```r
str_detect("\u0130", fixed("i"))
```

```
[1] FALSE
```

5.10.0.3 coll

Para comparar strings respeitando seu agrupamento. Interessante para strings com localização e não sensíveis a maiúsculas ou minúsculas.

Exemplo:

```
str_detect("\u0130", coll("i", TRUE, locale = "tr"))
```

```
[1] TRUE
```

5.10.0.4 boundary

Para localizar fronterias entre caracteres. quebra de linhas, sentenças ou palavras.

> Dica
>
> Para maiores informações veja os exemplo digitando ?stringr::modifiers

6 Datas e horas com LUBRIDATE

6.1 Introdução

A seguir temos vários exemplos de manipulação de **variáveis data e hora** utilizando o pacote LUBRIDATE do R. Para saber mais sobre este pacote, acesse:

https://cran.r-project.org/package=lubridate.

> ⚠ Aviso
>
> Para melhor utilizar este material, é importante que você tenha uma introdução à linguagem R e saiba carregar pacotes (packages) no R. Para mais informações acesse: https://education.rstudio.com/learn/beginner/.

Para os exemplos, iremos carregar os seguintes pacotes:

- **tidyverse**
- **gt**
- **lubridate**

```
library (tidyverse)
library (gt)
library (lubridate)
```

6.2 Tipos de objetos de data e hora

6.2.0.1 Datetime

Uma variável do tipo "**datetime**" (*data e hora*) representa um ponto na linha do tempo armazenado em um número que representa o número de **segundos** desde **01-01-1970 00:00:00 (UTC)**.

> **Nota**
>
> Universal Time Coordinated (**UTC**), é uma escala coordenada de tempo, mantida pelo "Bureau International des Poids et Mesures (BIPM)". Até 1972, era chamado de (**GTM** ou *Greenwich Mean Time*). É também conhecida como "Z time" ou "Zulu Time".

6.2.0.2 Date

Quando nos referimos à uma variável "**date**"(*data*), significa que ela armazena um número inteiro que representa o número de **dias desde 01-01-1970.**

6.2.0.3 Time

Quando nos referimos à uma variável "**time**" (tempo em segundos), ela armazena um número inteiro que representa o número de **segundos desde às 00:00:00** (hms).

Para os vários exemplos a seguir, utilizaremos os seguintes objetos data e hora:

```
dt <-  as_datetime(1511870400)
d <- as_date(17498)
t <-hms::as_hms(85)
dt; d; t
```

```
[1] "2017-11-28 12:00:00 UTC"
```

```
[1] "2017-11-28"
```

```
00:01:25
```

> **Dica**
>
> Os objetos gerados pela maioria das funções do lubridate usam os padrões POSIXct, POSIXlt, Date, Period ou objetos que podem ser convertidos para o POSIXlt. Para maiores informações sobre estas classes, digite:
> ?DateTimeClasses
> POSIXct: armazena segundos desde 01-01-1970 00:00:00 (Unix epoch) POSIXlt: armazena uma lista de dia, mês, ano, hora, min, segundos, etc.

6.2.1 Exemplos da Folha de Referência

A maioria dos exemplos, visam ajudar na interpretação dos exemplos e funções encontradas na **Folha de Referência** do lubridate disponível no site do RStudio.

> **Nota**
>
> *Em geral, ao final de cada comando, as vezes você verá a chamada à função* ***gt()****. Isto é apenas para a formatação da tabela de saída e não é necessário para que você entenda os comandos precedentes. Em alguns casos, onde o volume de dados de saída pode ser extenso, usamos também a função* ***head()*** *para mostrar apenas as linhas iniciais. Quando o exemplo possui muitas colunas de saída, eventualmente utilizamos a função* ***select()*** *para selecionar apenas algumas colunas.*
> Em alguns casos usaremos funções de manipulação de dados do pacote **dplyr,** como **mutate** () ou **count**().

> **Nota**
>
> *O termo data-frame descrito ao longo deste texto, é utilizado de forma livre para objetos do tipo data.frame, tibble, entre outros. Pense como se fosse uma tabela de um banco de dados e/ou uma planilha do MS Excel, contendo linhas e colunas. Apesar de não ser rigorosamente igual à uma tabela, muitas vezes usaremos estes termos de forma intercambiável para facilitar o entendimento de iniciantes.*

6.3 Validando Data e Hora

O pacote lubridate possui uma série de funções para obter e definir os elementos de ano, mês, dia, hora, minuto e segundos de um objeto data e hora.

Use as funções a seguir servem para identificar estes elementos em seus dados a partir de uma string. Cada uma delas aceita o argumento "tz" para definir o fuso-horário (*timezone*), se este não for definido, UTC é utilizado.

Estas funções são nomeadas conforme a tabela abaixo e sua ordem obedece tal nomenclatura:

Elemento	Letra
y	ano (*year*)
m	mês (*month*)

Elemento	Letra
d	dia (*day*)
h	hora (*hour*)
m	minuto (*minute*)
s	segundos (*seconds*)

Por exemplo, para criar um objeto **datetime** passando a string "**2017-11-28T14:02:00**", utilzamos a função **ymd_hms**(). Isto porque ao montar a string de data e hora, colocamos como ordem o ano, mês, dia, hora, minuto e segundo.

```
ymd_hms("2017-11-28T14:02:00")
```

```
[1] "2017-11-28 14:02:00 UTC"
```

Se passarmos a string trocando o ano pela dia, devemos usar a função dmy_hms():

```
dmy_hms("28-11-2017T14:02:00")
```

```
[1] "2017-11-28 14:02:00 UTC"
```

> **Nota**
>
> Veja que tanto a função ymd_hms() quanto a dmy_hms(), geraram o mesmo objeto datetime. Se quisermos obter o valor inteiro que representa este objeto desde 01-01-1970 00:00:00, podemos usar a função **decimal_date**()

Veja o código abaixo com mais alguns exemplos das funções validando os elementos da string passada para a função:

```
# ymd_hms(), ymd_hm(), ymd_h().
ymd_hms("2017-11-28T14:02:00") |> print()
```

```
[1] "2017-11-28 14:02:00 UTC"
```

```
# ydm_hms(), ydm_hm(), ydm_h().
ydm_hms("2017-22-12 10:00:00") |>  print()
```

```
[1] "2017-12-22 10:00:00 UTC"
```

```
# mdy_hms(), mdy_hm(), mdy_h().
mdy_hms("11/28/2017 1:02:03") |>  print()
```

```
[1] "2017-11-28 01:02:03 UTC"
```

```
# dmy_hms(), dmy_hm(), dmy_h().
dmy_hms("1 Jan 2017 23:59:59")|>  print()
```

```
[1] "2017-01-01 23:59:59 UTC"
```

```
# ymd(), ydm().
ymd(20170131)|>  print()
```

```
[1] "2017-01-31"
```

```
# mdy(), myd().
mdy("July 4th, 2000")|>  print()
```

```
[1] "2000-07-04"
```

```
# dmy(), dym().
dmy("4th of July '99")|>  print()
```

```
[1] "1999-07-04"
```

```
# yq() Q para quartil.
yq("2001: Q3")|>  print()
```

```
[1] "2001-07-01"
```

```
# my(), ym().
my("07-2020")|>  print()
```

```
[1] "2020-07-01"
```

```
#hms::hms() ou lubridate::hms(), ms() ou hm() para períodos.
hms::hms(sec = 0, min = 1, hours = 2)
```

```
02:01:00
```

6.3.1 Outras funções úteis

6.3.1.1 date_decimal

Use para converter um número decimala para data e hora:

```
date_decimal(2017.5)
```

```
[1] "2017-07-02 12:00:00 UTC"
```

6.3.1.2 now

Use para obter um objeto data e hora do instante:

```
now(tzone = "America/Sao_Paulo")
```

```
[1] "2022-08-26 12:04:07 -03"
```

> **!** Importante
>
> Se o fuso-horário (tzone =) não for informado, a função utilizará aquele utilzado pelo sitema operacional em execução.

6.3.1.3 today

Use para obter a data atual.

```
today()
```

```
[1] "2022-08-26"
```

6.3.1.4 fast_strptime

Use para converter vetores de caracteres para objetos data e hora (POSIXlt) de forma rápida.

```
fast_strptime('9/1/01', '%y/%m/%d')
```

```
[1] "2009-01-01 UTC"
```

6.3.1.5 parse_date_time

Use para converter vetores de caracteres para objetos data e hora (POSIXct) de forma mais simplificada.

```
parse_date_time("19/1/1", "ymd")
```

```
[1] "2019-01-01 UTC"
```

6.4 Obtendo e Definindo Componentes de Data e Hora

Use as funções abaixo para obter um componente de um objeto data e hora.

```
# Obter o "DIA" de um objeto "datetime"
day(dt)
```

```
[1] 28
```

```
# Obter a "DATA"
date(dt)
```

```
[1] "2017-11-28"
```

```r
# Obter a "ANO".
# Para obter o "ANO ISO 8610 use isoyear()
# Para obter o "ANO Epidemiológico use epiyear()
year(dt)
```

```
[1] 2017
```

```r
# Obter o "MÊS".
# Use argumentos label= e addr= para obter o nome ou abreviação do mês.
month(dt)
```

```
[1] 11
```

```r
# Obter o "DIA DA SEMANA".
# Use argumentos label= e addr= para obter o nome ou abreviação do dia.
wday(dt, label = TRUE)
```

```
[1] ter
Levels: dom < seg < ter < qua < qui < sex < sáb
```

```r
# Obter o "DIA DO TRIMESTRE".
qday(dt)
```

```
[1] 59
```

```r
# Obter a HORA".
hour(dt)
```

```
[1] 12
```

```r
# Obter os "MINUTOS".
minute(dt)
```

```
[1] 0
```

```r
# Obter os "SEGUNDOS".
second(dt)
```

```
[1] 0
```

```r
# Obter o "FUSO-HORÁRIO.
tz(dt)
```

```
[1] "UTC"
```

```r
# Obter a "SEMANA DO ANO".
week(dt)
```

```
[1] 48
```

```r
# ara obter a "SEMANA DO ANO" ISO 8160 use isoyear()
# Para obter a "SEMANA DO ANO" Epidemiológico use epiyear()

# Obter o "TRIMESTRE".
quarter(dt)
```

```
[1] 4
```

```r
# Obter o "SEMESTRE".
semester(dt)
```

```
[1] 2
```

```r
# Saber se é "MANHÃ (am).
am(dt)
```

```
[1] FALSE
```

```r
# Saber se é "TARDE" (pm).
pm(dt)
```

```
[1] TRUE
```

```r
# Saber se é "HORÁRIO DE VERÃO"
dst(d)
```

```
[1] FALSE
```

```r
# Saber se é "ANO BISEXTO"
leap_year(d)
```

```
[1] FALSE
```

Para definir um componente de um objeto, podemos utilzar as funções acima, porém com o sinal de atribuição.

Por exemplo, para alterar o dia de “28” do objeto “d”, para dia “1”, podemos fazer:

```r
day(d) |>  print()
```

```
[1] 28
```

```r
day(d) <- 1
print(d)
```

```
[1] "2017-11-01"
```

Podemos também atualizar um componente do objeto data e hora:

```r
# Atualizar um componente do objeto
update(dt, mday = 2, hour = 1)
```

```
[1] "2017-11-02 01:00:00 UTC"
```

O exemplo acima, altera o dia do mês para 2 e a hora para 01.

6.5 Arredondando Data e Hora

Use as funções a seguir para "arredondar" ou aproximar um objeto data e hora para unidades de ajuste. As unidades válidas são:

- *second, minute, hour, day, week, month, bimonth, quarter, season, halfyear e year.*

6.5.0.1 floor_date

Use para "**arredondar para baixo**" a data e hora para a unidade mais próxima.

Por exemplo, digamos que temos um objeto data = "2017-11-28" e queremos arredondar para baixo, sendo que a unidade é mês, ou seja, arredondar para o início do mês:

```
floor_date(dt, unit="month")
```

```
[1] "2017-11-01 UTC"
```

round_date

Use para "**arredondar**" a data para a unidade mais próxima.

```
round_date(dt, unit="month")
```

```
[1] "2017-12-01 UTC"
```

Veja que no exemplo acima, como tínhamos dia 28/11 e pedimos para arredondar na unidade "month", ele arredondou para o mês 01/12.

Se o dia fosse 14/11, a função arredondaria para 01/11.

6.5.0.2 ceiling_date

Use para "**arredondar para cima**" a data e hora para a unidade mais próxima.

```
ceiling_date(dt, unit="month")
```

```
[1] "2017-12-01 UTC"
```

6.6 Imprimindo data e hora

Em alguns casos, desejamos imprimir um objeto data e hora de uma maneira específica e/ou mais amigável. O pacote lubridate tem a capacidade de utilizar "templates" e ainda permite modificá-los para customizar como a impressão do objeto será feita.

6.6.0.1 stamp

Use para criar um "template" mais amigável à partir de uma string de exemplo. Veja também as função stamp_date() e stamp_time() que são funções específicas para lidar com datas e horas respectivamente.

Em geral criamos uma função que utiliza a função stamp() e depois a utilizamos em nosso script passando o objeto data e hora. Veja este exemplo:

```
sf <- stamp("Criado terça-feira, 17 de janeiro de 2022 às 3:34")
sf(ymd("2020-04-05"))
```

```
[1] "Criado domingo-feira, 05 de abril de 2020 às 00:00"
```

> Dica
>
> Procure usar o **dia maior que 12** na hora de criar o template. Isto facilita para função distinguir que parte do template é o mês e qual parte é o dia.

6.7 Fuso-Horários

O R reconhece ~600 fuso-horários. Cada um deles, tem iformações sobre o fuso-horário, horário de verão e variações de calendário históricas de uma área. O R define apenas um fuso-horário por vetor.

Use o fuso-horário "UTC" para evitar horários-de-verão nos objetos.

Para obter uma lista dos fuso-horários disponíveis, use:

```
OlsonNames() |>
  as_tibble()
```

```
# A tibble: 595 x 1
   value
   <chr>
 1 Africa/Abidjan
 2 Africa/Accra
 3 Africa/Addis_Ababa
 4 Africa/Algiers
 5 Africa/Asmara
 6 Africa/Asmera
 7 Africa/Bamako
 8 Africa/Bangui
 9 Africa/Banjul
10 Africa/Bissau
# ... with 585 more rows
# i Use `print(n = ...)` to see more rows
```

6.7.0.1 Sys.timezone

Use para obter o fuso-horário atual, use:

```
Sys.timezone()
```

```
[1] "America/Sao_Paulo"
```

6.7.0.2 with_tz

Use para obter **o mesmo objeto** data e hora em um novo fuso-horário (**novo relógio**).

```
with_tz(dt, tzone = "US/Alaska")
```

```
[1] "2017-11-28 03:00:00 AKST"
```

6.7.0.3 local_time

Para saber a diferença entre fuso-horários, podemos usar a função local_time e definir a unidade. Por exemplo:

```
local_time(dt, tz = "US/Alaska", units = "hours")
```

```
Time difference of 3 hours
```

6.7.0.4 force_tz

Use para obter **o mesmo objeto** data e hora em um novo fuso-horário (**novo data e hora**).

```
force_tz(dt, "US/Pacific")
```

```
[1] "2017-11-28 12:00:00 PST"
```

6.8 Matemática com Data e Hora

6.8.1 Introdução

O pacote lubridate fornece três classes de intervalo de tempo para fazer cálculos com data e hora.

- **Períodos**: Acompanham mudanças no horário do relógio, isto ignora irregularidades na "linha do tempo".
- **Durações**: Acompanham a passagem do "tempo físico", o que diverge do horário do relógio quando irregularidades na "linha do tempo" acontecem.
- **Intervalos**: Representam um intervalo específico da "linha do tempo", limitado pelo início e fim da data e hora.

Este três formas de enchergam a "linha do tempo" é necessário pois cálculos de data e hora usando a "linha do tempo" são inconsistentes.

Sabemos que nem todos os anos têm 365 dias, com no caso do ano bi-sexto. Ou no caso de minutos de um retorno do horário de verão tem 60 segundos.

Pense nos seguintes cenários:

Se tivermos um dia normal, a "linha do tempo" ficaria algo como:


```
nor <- ymd_hms("2018-01-01 01:30:00",tz="US/Eastern")
print(nor)
```

```
[1] "2018-01-01 01:30:00 EST"
```

Já, quando o horário de verão se inicia, termos o seguinte cenário na linha do tempo:


```
gap <- ymd_hms("2018-03-11 01:30:00",tz="US/Eastern")
```

Quando o horário então se encerra, temos na linha do tempo este cenário:


```
lap <- ymd_hms("2018-11-04 00:30:00",tz="US/Eastern")
```

E ainda temos o "ano-bisexto", que também causa inconsistência na linha do tempo:


```
leap <- ymd("2019-03-01")
```

Para os casos acima, criamos quarto objetos data e hora: **nor**, **gap**, **lap** e **leap** para representar cada cenário de inconsistência na linha do tempo.

Agora veremos com as três classes do lubridate citadas anteriormente reagem em cada situação:

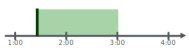
6.8.2 Períodos

Vimos que os períodos acompanham as mudanças no horário do relógio, isto ignora irregularidades na "linha do tempo".

Por exemplo, se quisermos adicionar 90 minutos ao objeto nor criado anteriormente, teremos:

```
nor + minutes(90)
```

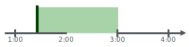
```
[1] "2018-01-01 03:00:00 EST"
```



Já, se quisermos adicionar 90 minutos no dia do início do horário de verão (objeto gap), teremos:

```
gap + minutes(90)
```

```
[1] "2018-03-11 03:00:00 EDT"
```

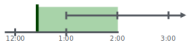


Veja que o período ignorou a inconsistência na linha do linha e trouxe o resultado como ela não existisse.

O mesmo aconteceria com a data e hora do objeto lap criado no fim do horário de verão:

```
lap + minutes(90)
```

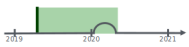
```
[1] "2018-11-04 02:00:00 EST"
```



Situação identica aconteceria para o objeto leap criado em ano bisexto. Por exemplo, digamos que queremos somar um período de 1 ano.

```
leap + years(1)
```

```
[1] "2020-03-01"
```



As funções de períodos para adicionar ou subtrair data e hora, tem o nome da unidade seguido de um "s". Nos exemplos anterior somamos minutos usando minutes() e anos usando years().

A lista abaixo traz as funções que criam objetos períodos, ou sejam, que modelam eventos que acontecem em horário específico do relógio.

Podemos utilzar estes objetos para somar ou subtrair de objetos data ae hora.

Função	Objeto Período
years(x = 1)	x anos
months(x)	x meses
weeks(x = 1)	x semanas

Função	Objeto Período
days(x = 1)	x dias
hours(x = 1)	x horas
minutes(x = 1)	x minutos
seconds(x = 1)	x segundos
milliseconds(x = 1)	x milisegundos
microseconds(x = 1)	x microsegundos
nanoseconds(x = 1)	x nanosegundos
picoseconds(x = 1)	x picosegundos

Por exemplo, se quisermos criar um objeto período com 3 meses e 12 dias, fazemos:

```
p <- months(3) + days(12)
p
```

```
[1] "3m 12d 0H 0M 0S"
```

Para subtrair este período de um objeto data e hora, fazemos:

```
dt - p
```

```
[1] "2017-08-16 12:00:00 UTC"
```

Podemos também usar as funções abaixo para criar objetos período:

6.8.2.1 period

Use para automatizar a criação de períodos.

Por exemplo, para criar um objeto com período de 5 anos, podemos usar years(5) ou:

```
period(5, unit = "years")
```

```
[1] "5y 0m 0d 0H 0M 0S"
```

6.8.2.2 as.period

Use para transformar objetos de duração, intervalos e números para obejtos do tipo período:

Por exemplos, temos um número 5 e queremos criar um período de 5 dias, podemos fazer:

```
as.period(5, unit="days")
```

```
[1] "5d 0H 0M 0S"
```

6.8.2.3 period_to_seconds

Use para transformar um objeto do tipo período no total de número de segundos do período:

```
period_to_seconds(p)
```

```
[1] 8926200
```

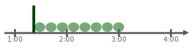
6.8.3 Duração

Diferentes dos objetos períodos, os objetos do tipo duração (*duration*), Acompanham a passagem do "tempo físico", o que diverge do horário do relógio quando irregularidades na "linha do tempo" acontecem.

Por exemplo, digamos que temos nosso "dia normal" na linha do tempo e adicionarmos 90 minutos de duração:

```
nor + dminutes(90)
```

```
[1] "2018-01-01 03:00:00 EST"
```



Até aqui, o resultado foi similar à adicionarmos um objeto do tipo período de 90 minutes.

Porém, veja o que acontece quando temos uma inconsistência na linha do tempo, como por exemplo nosso início de horário de verão em nosso objeto gap.

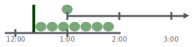
```
gap + dminutes(90)
```

```
[1] "2018-03-11 04:00:00 EDT"
```

O mesmo acontece com nosso término de horário de verão em nosso objeto lap:

```
lap + dminutes(90)
```

```
[1] "2018-11-04 01:00:00 EST"
```



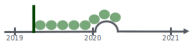
Ou mesmo com nosso ano bi-sexto:

```
leap
```

```
[1] "2019-03-01"
```

```
leap + dyears(1)
```

```
[1] "2020-02-29 06:00:00 UTC"
```



Podemos pensar em objetos de duração como um modelo físico, como uma vida útil de uma bateria. As durações são armazenados como segundos, que é a única unidade distância consistente.

Por exemplo, se criarmos um objeto duração equivalente à 14 dias, ele irá armazenar 1209600s.

```
dd <- ddays(14)
dd
```

```
[1] "1209600s (~2 weeks)"
```

> Dica
>
> Há também uma classe chamada "**difftime**", que se encontra no R base, ou seja, fora do pacote lubridate, usada para lidar com durações de tempo.

As funções para criar objetos de duração, são similares às dos objetos períodos, porém se iniciam com a letra "d", veja:

Podemos também usar as funções abaixo para criar objetos duração:

Função	Objeto Duração
dyears(x = 1)	31536000x anos
dmonths(x)	2629800x meses
dweeks(x = 1)	604800x semanas
ddays(x = 1)	x86400x dias
dhours(x = 1)	3600x horas
dminutes(x = 1)	60x minutos
dseconds(x = 1)	x segundos
dmilliseconds(x = 1)	x X 10^3 milisegundos
dmicroseconds(x = 1)	x X 10^6 microsegundos
dnanoseconds(x = 1)	x X 10^9 nanosegundos
dpicoseconds(x = 1)	x X 10^{12}picosegundos

6.8.3.1 duration

Use para automatizar a criação de durações.

Por exemplo, para criar um objeto com duração de 5 anos, podemos usar dyears(5) ou:

```
duration(5, unit = "years")
```

```
[1] "157788000s (~5 years)"
```

6.8.3.2 as.duration

Use para transformar objetos de períodos, intervalos e números para objetos do tipo duração:

Por exemplos, temos um número 10 e queremos criar um período de 10 segundos, podemos fazer:

```
as.duration(10)
```

```
[1] "10s"
```

6.8.3.3 make_difftime

Use para criar um objeto difftime (R base) com um número específico de unidades.

```
make_difftime(3600)
```

```
Time difference of 1 hours
```

6.8.4 Intervalo

Objeto do tipo intervalo, representam um intervalo específico da "linha do tempo", limitado pelo início e fim da data e hora. Se dividirmor o intervalo, pela pela duração teremos a distância física do tempo. Se dividirmos o intervalo pelo período, termeos a distância relativa ao relógio.

Podemos criar um objeto de intervalo, usando a função interval() ou o símbolo %--%.

```
i <- interval(ymd("2017-01-01"), d)
j <- d %--% ymd("2017-12-31")
i; j
```

```
[1] 2017-01-01 UTC--2017-11-01 UTC
```

```
[1] 2017-11-01 UTC--2017-12-31 UTC
```

Observe pelo resultado acima, temos duas data para cada objeto, a da esquerda representa o início do intervalo e a da direito o fim.

Por exemplo, vamos pegar um dia normal na linha do tempo, representado pelo objeto nor e definirmos como o início do intervalo, e para o fim do intervalo usaremos nor mais um período de 90 minutos.

```
interval(nor, nor + minutes(90))
```

```
[1] 2018-01-01 01:30:00 EST--2018-01-01 03:00:00 EST
```

Agora, em uma linha do tempo inconsistente, o intervalo se mantém alinhado com o relógio. Veja como fica quando adicionamos um intervalo de 90 minutos quando temos o início de um horário de verão:

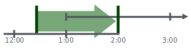
```
interval(gap, gap+minutes(90))
```

```
[1] 2018-03-11 01:30:00 EST--2018-03-11 03:00:00 EDT
```

De forma similar, ocorre quando temos um intervalo quando há o término de um horário de verão:

```
interval(lap, lap+minutes(90))
```

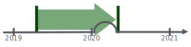
```
[1] 2018-11-04 00:30:00 EDT--2018-11-04 02:00:00 EST
```



Ou mesmo quando temos um intervalo em um ano bi-sexto:

```
interval(leap, leap + years(1))
```

```
[1] 2019-03-01 UTC--2020-03-01 UTC
```



O pacote lubridate possui diversas funções para lidar com intervalo.

6.8.4.1 %within%

Use para identificar se um objeto do tipo intervalo ou data e hora "a" cai dentro de um interválo "b"

Por exemplo, se quisermos se a data e hora atual está dentro do intervalo "i".

```
now () %within% i
```

```
[1] FALSE
```

6.8.4.2 int_start

Use para obter ou definir o **início** de um **intervalo**:

```
int_start(i)
```

```
[1] "2017-01-01 UTC"
```

```
int_start(i) <- now()
```

> **i** Nota
>
> A função int_end() faz o oposto, ou seja, obtem ou define o **fim** de um intervalo.

int_aligns

Use para identificar se dois objetos do tipo intervalo estão alinhados, ou seja, compartilham de uma mesma data e hora.

```
int_aligns(i,j)
```

```
[1] TRUE
```

No exemplo acima, temos "2017-11-28" como início de um objeto e fim de outro, por isso dizemos que eles estão alinhados.

> **i** Nota
>
> Se quisermos saber se estes objetos estão **sobrepostos**, ou seja, tem partes de uma intervá-lo que também fazem parte de outro, utilizamos a função **int_overlaps**().

6.8.4.3 int_diff

Use para transformar em intervalos, os valores que estão em um vetor de data e hora.

```
  v <- c(dt, dt+100, dt+1000); int_diff(v)
```

```
[1] 2017-11-28 12:00:00 UTC--2017-11-28 12:01:40 UTC
[2] 2017-11-28 12:01:40 UTC--2017-11-28 12:16:40 UTC
```

int_flip

Use para colocar em ordem reversa a direção de um intervalo, ou seja, a dat e hora do fim vai para o início do intervalo e a data e hora do início vai para o final.

```
  int_flip(i)
```

```
[1] 2017-11-01 UTC--2022-08-26 15:04:08 UTC
```

Para colocar em ordem padrão um intervalo de acordo com a linha do tempo, podemos usar a função **int_standardize**().

```
  int_standardize(i)
```

```
[1] 2017-11-01 UTC--2022-08-26 15:04:08 UTC
```

6.8.4.4 int_length

Use para obter, em segundos, o tempo total de um intervalo:

```
  int_length(i)
```

```
[1] -152031848
```

6.8.4.5 int_shift

Use para mover um intervalo para mais ou para menos na linha do tempo.

Por exemplo, se mover todo o intervalo (início e fim) em um dia antes da linha do tempo, podemos fazer:

```
  int_shift(i, days(-1))
```

```
[1] 2022-08-25 12:04:08 -03--2017-10-30 22:00:00 -02
```

6.8.4.6 as.interval

Use para criar um objeto intervalo com determinado periodo definindo uma dat e hora de início.

Por exemplo, para criarmos um intervalo de 1 dia, iniciando na data atual, podemos fazer:

```
as.interval(days(1), start = now())
```

```
[1] 2022-08-26 12:04:08 -03--2022-08-27 12:04:08 -03
```

> **Nota**
>
> Podemos usar a função **is.interval**() para saber se um objeto é um intervalo válido ou não.

6.9 Datas Imaginárias

É importante observar que nem todos os anos tem 365 dias (ex: ano bi-sexto) e nem todos os minutos tem 60 segundos (ex: fim de horário de verão).

Isso é importante de ser observado, pois em alguns casos tentamos criar data imaginárias, como por exemplo "Fev 31", adicionando um mês à "Jan 31". As funções do pacote lubridate são inteligentes o suficiente e neste caso retornaria um valor NA:

```
jan31 <- ymd(20180131)
jan31 + months(1)
```

```
[1] NA
```

6.9.1 Aritmética dos meses

Porém, as vezes, intuitivamente, é isto que desejamos fazer, ou seja, adicionar "um mês" a "Jan 31", mas que a função seja inteligente o suficiente para **rolar** para o **último dia do mês**.

Adicionar ou subtrair **meses** as vezes é uma tarefa difícil, pois temos meses de diferentes tamanhos (ex: 30, 31, 28 dias ou até 29). Por isso, em alguns casos é útil termos a possibildade de fazermos um ajustes automáticos.

Para isso usamos, ao invés do sinal de adição "+", utilizamos o símbolo %**m+**% para adicionar meses (ou %**m-**% para subtrair). Veja:

```
jan31 %m+% months(1)
```

```
[1] "2018-02-28"
```

A função **add_with_rollback**() nos permite rolar a data da soma para o primeiro dia do mês seguinte (e não o último dia do mês anterior) usando o argumento **roll_to_first.**

```
add_with_rollback(jan31, months(1), roll_to_first = TRUE)
```

```
[1] "2018-03-01"
```

7 Visualização de Dados com GGPLOT2

7.1 Introdução

A seguir temos vários exemplos de visualização de dados utilizando o pacote GGPLOT2 do R.

> **Nota**
>
> Apesar de visualização de dados não ser especificamente parte das etapas de transformação e manipulação de dados, acreditamos ser importante um conhecimento básico sobre o tema, pois muitas vezes, para explicarmos aquilo que estamos transformando ou manipulando, o fazemos de forma gráfica para melhor compreensão.
> O objetivo não é explicar a aplicabilidade e/ou o a função de cada gráfico, mas sim, como ele pode ser construído.

Para saber mais sobre este pacote, acesse:

https://cran.r-project.org/package=ggplot2.

> **Aviso**
>
> Para melhor utilizar este material, é importante que você tenha uma introdução à linguagem R e saiba carregar pacotes (packages) no R. Para mais informações acesse: https://education.rstudio.com/learn/beginner/.

Para os exemplos, iremos carregar os seguintes pacotes:

- **tidyverse**
- **gt**

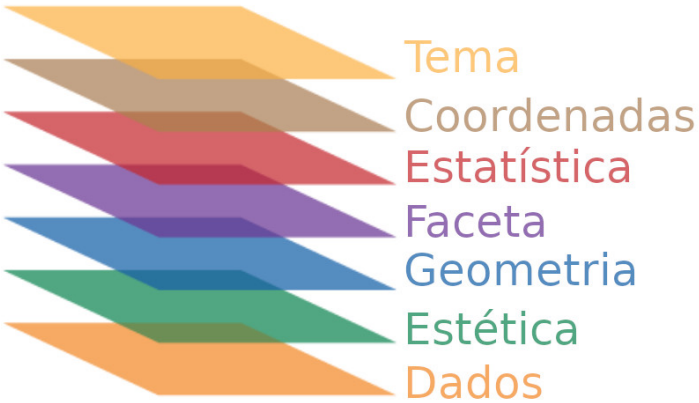
```
library (tidyverse)
library (gt)
```

7.1.1 Exemplos da Folha de Referência

A maioria dos exemplos, visam ajudar na interpretação dos exemplos e funções encontradas na **Folha de Referência** do stringr disponível no site do RStudio.

7.2 Gramática dos Gráficos

O pacote ggplot2 é uma implementação do livro "***Grammar of Graphics***", que apresenta um conceito de quais seriam os elementos de um gráfico e suas interconexões. O modelo teórico proposto no livro é que através de camadas definidas qualquer gráfico pode ser construído. Abaixo o modelo, que deve ser interpretado de baixo para cima.



De uma forma bem resumida, a idéia é que você possa construir qualquer gráfico com base nestes mesmos elementos: um **conjunto de dados**, um **sistema de coordenadas** e **geoms**, que seriam marcas visuais que representas os pontos dos dados.

No caso do ggplot2, para mostrar valores no gráfico, as varíaveis do conjunto de dados são **mapeadas** em propriedadas visuais da geometria (geom).

Este mapeamento de estética (aesthetic) é feito através da função aes() e podemos fazer mapeamentos estéticos como cor ou tamanho.

7.3 Modelo para Construção de um Gráfico

No caso do ggplot2, podemos definir uma espécie de modelo (template) para a sintaxe de construção de gráficos:

```
ggplot(data = <DADOS>) +
<FUNÇÃO_GEOM> (mapping = aes(<MAPEAMENTO>),
stat = <ESTATÍSTICA>, position = <POSIÇÃO>) +
<FUNÇÃO_COORDENADA> +
<FUNÇÃO_FACETA> +
<FUNÇÃO_ESCALA> +
<FUNÇÃO_TEMA>
```

Necessário

Não mandatórios, valores padrão fornecidos

7.3.0.1 ggplot

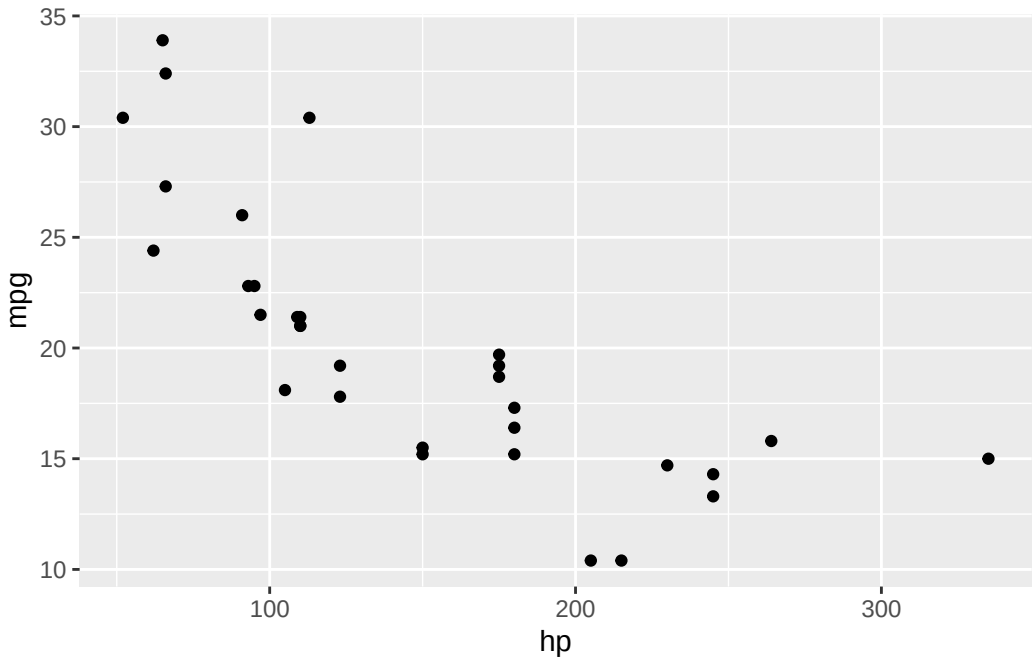
A função ggplot() inicia um gráfico que será completado com as camadas adicionadas na sequência. Adicione uma geometria (geom) para cada camada.

Usando o modelo anterior e o que vimos até aqui, vamos gerar nosso primeiro gráfico. Como base de dados usaremos a tabela **mtcars**. Para maiores informação sobre as colunas e dados do mtcars, digite ?mtcars.

```
mtcars
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

```
ggplot(data = mtcars) +
  geom_point (mapping = aes(x = hp, y = mpg ))
```

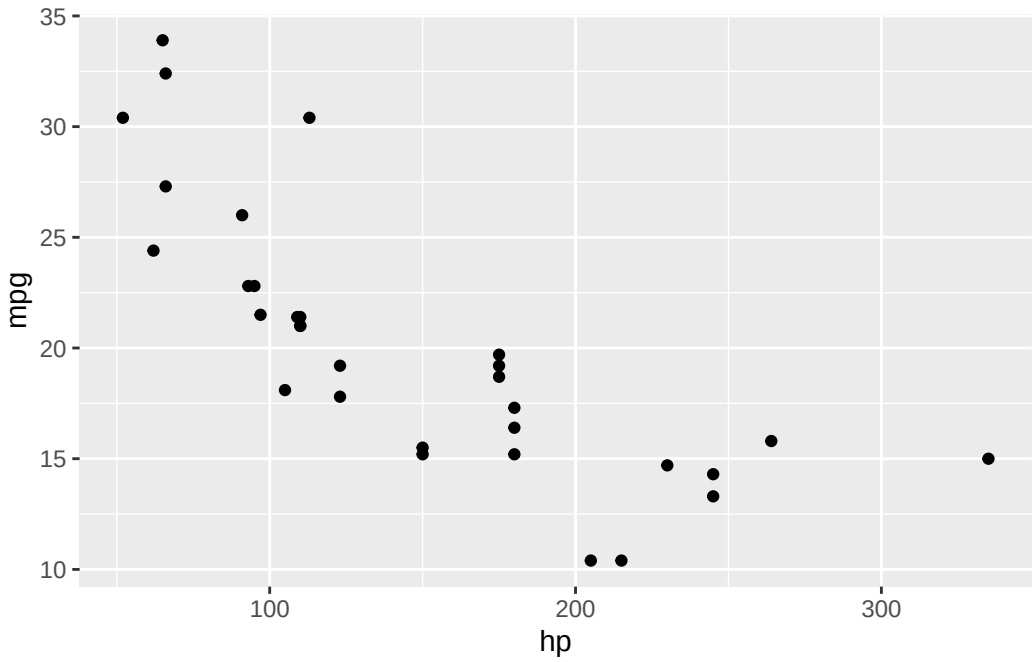


> **Nota**
>
> Observe que mesmo sem definirmos escalas, temas, estatísticas ou outras camadas, o gráfico ainda assim foi gerado. Isto é porque o ggplot tem valores padrões dinâmicos que tentam prover o melhor resultado possível com o mínimo de esforço.

> **Dica**
>
> A definição da estética com seu mapeamento, feito neste exemplo na função de geometria (geom_point) poderá ser feito também na função ggplot().

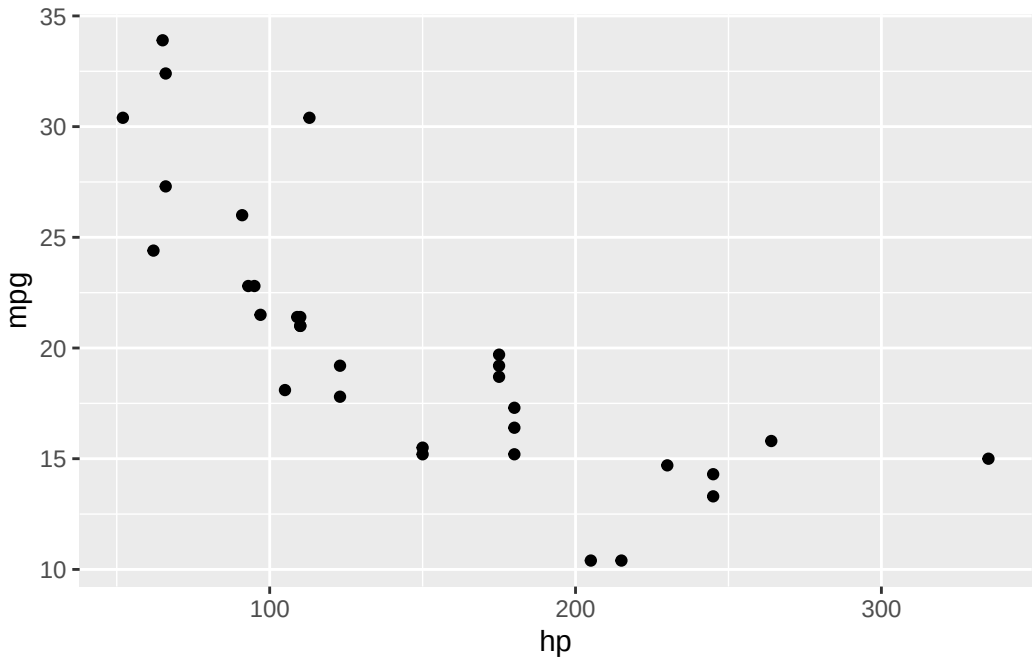
Apesar do gráfico final ser similar, a estética mapeamento poderá ser utilizada pelas funções geom_*() nas camadas seguintes, sem a necessidade de repetí-la, caso contrário, ela será restrito apenas àquela geom() na qual foi declarada. Veja:

```
ggplot(data = mtcars, mapping = aes(x = hp, y = mpg))+
  geom_point()
```



Por ser comum o uso do argumento data = e mapping =, podemos excluídas do código, deixando-o mais enxuto, porém mais difícil para quem não tem tanta familiaridade, veja:

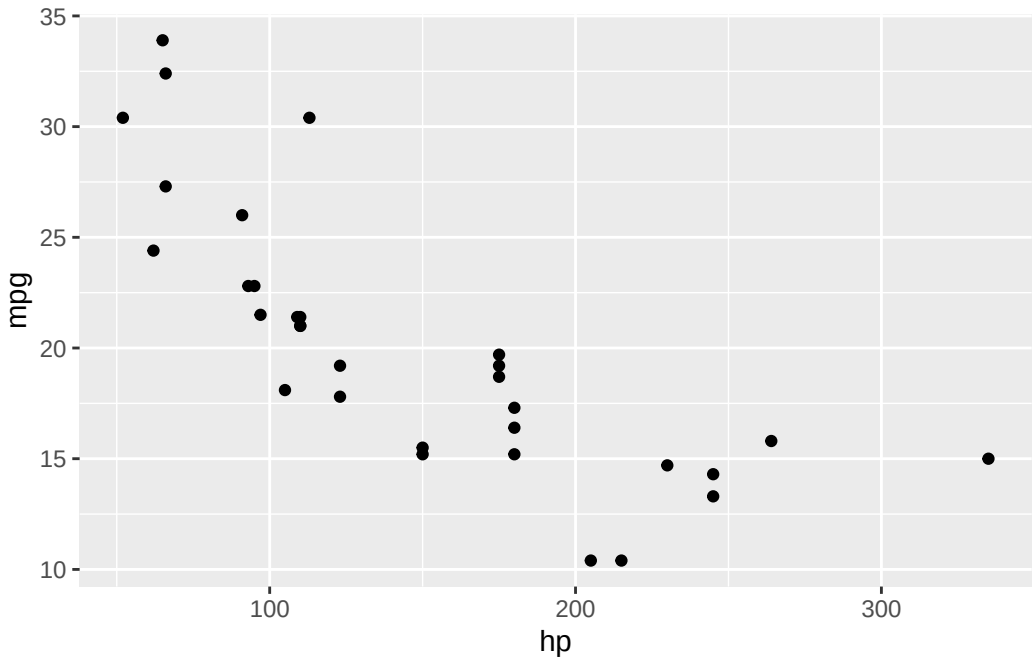
```
ggplot(mtcars,aes(x = hp, y = mpg)) +
  geom_point()
```



7.3.0.2 last_plot

Use para retornar o último gráfico gerado.

```
last_plot()
```



7.3.0.3 ggsave

Use para salvar o gráfico em uma imagem. O tipo de imagem é selecionado pela extensão do arquivo de saída.

Por exemplo, para salvar o diretório atual um arquivo com o gráfico chamado "plot-01.png" com o formato de imagem ".PNG" de tamanho 5x5 centímetros, fazemos:

```
ggsave("plot-01.png", width = 5, height = 5, units = "cm")
```

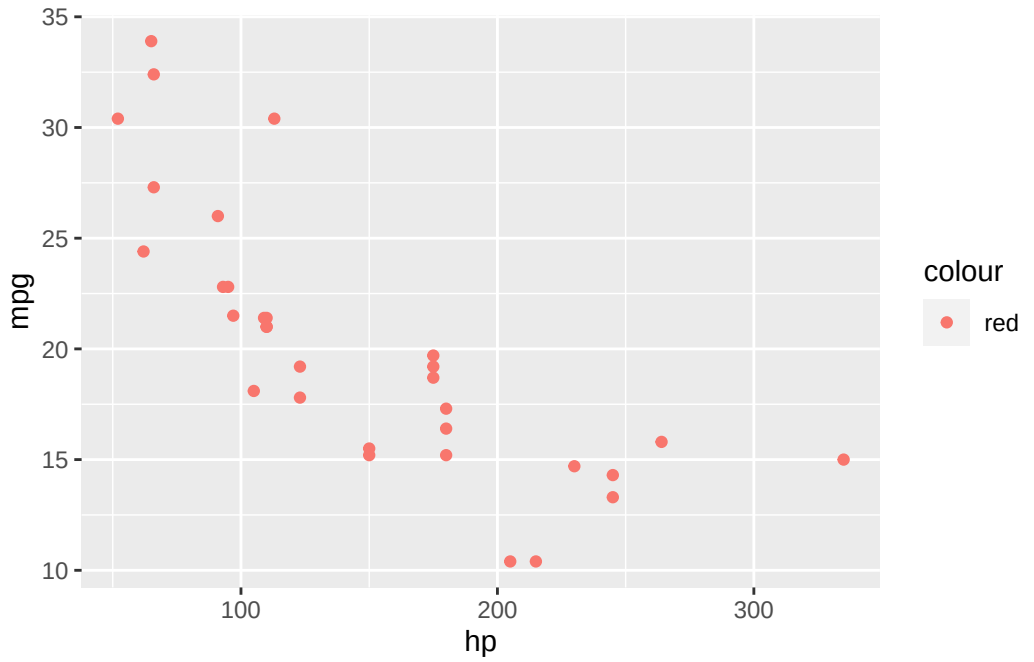
7.4 Estética (aes)

Cada estética possui valores padrões que serão usados por determinada função de geometria ou de estatística. Mas existem valores que são comuns a quase todas as geom/stat(), a seguir veremos alguns deles:

7.4.0.1 color e fill

Use para definir a cor (color) ou preenchimento (fill) para a estética.

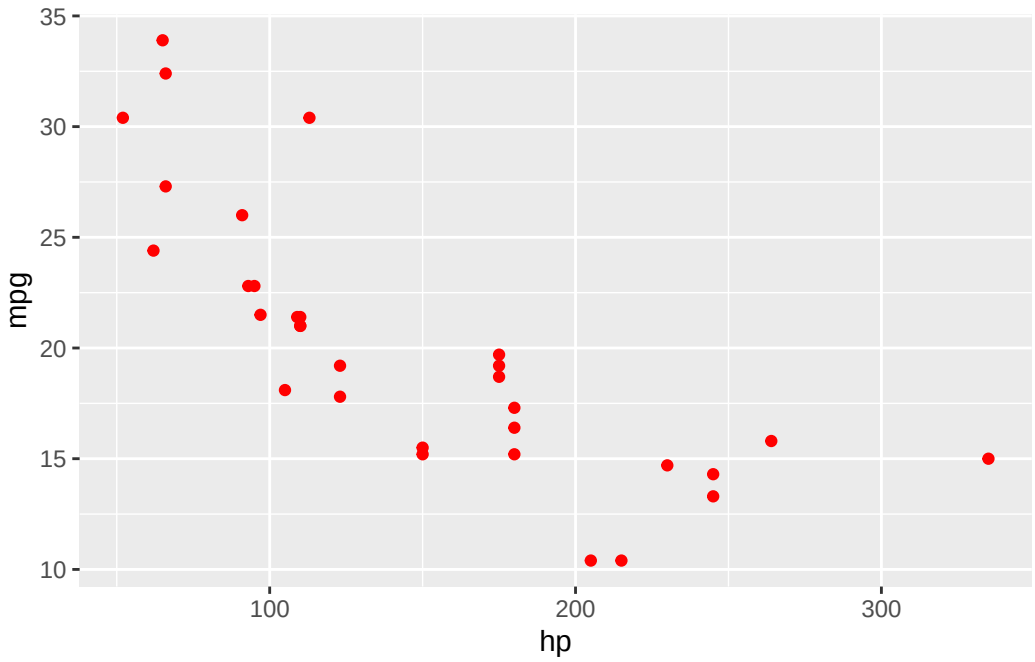
```
ggplot(mtcars)+
  geom_point(aes(x = hp, y = mpg, color = "red"))
```



Na string de color =, podemos usar valores de cores em inglês, como “red” ou “blue”, ou também códigos RGB, como ““#00ABFF” ou “FF00AA”.

> **!** Importante
>
> Observe que o valor definido da cor, dentro fora dos parênteses da função aes(), ou seja, ela está fazendo o mapeamento da cor com os dados. Se utilizarmos o argumento color = da função geom_point(),estaremos definindo um valor fora do mapeamento.

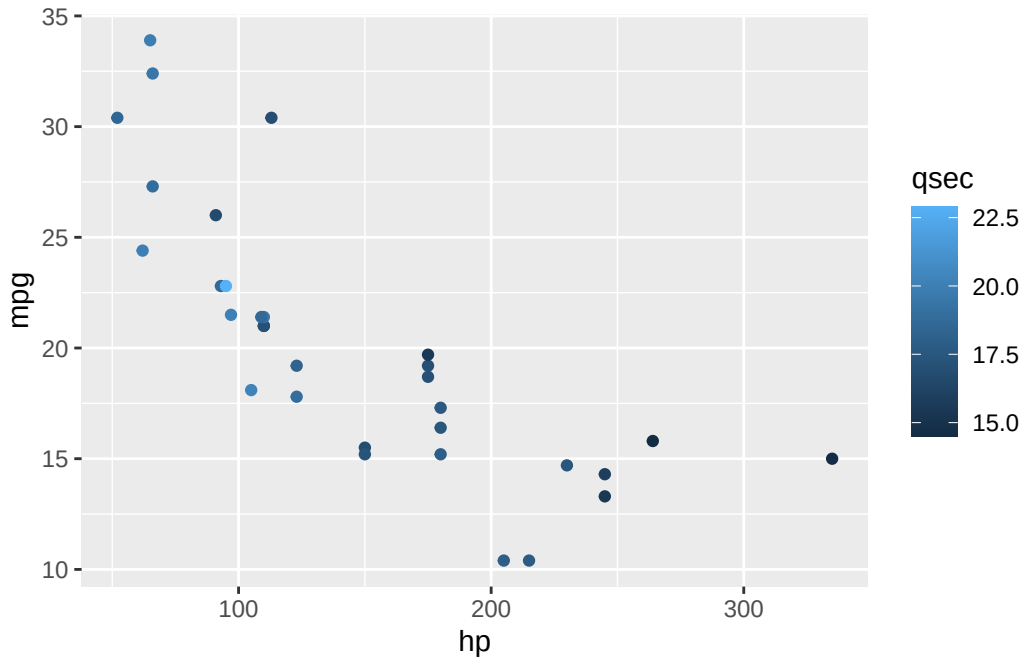
```
ggplot(mtcars)+
  geom_point(aes(x = hp, y = mpg) , color = "red")
```



Apesar do resultado ser similar, observe que no exemplo anterior tivemos color como um mapeamento da estética, como definimos o valor fixo com a cor "red", seria como tivessemos em nossos dados uma coluna onde todas as linhas tivessem o valor "red". Já no exemplo acima, estamos simplemente dizendo à função da geometria que queremos a cor vermelha para ela.

Vejamos um outro exemplo para que fique claro este ponto. Vamos gerar o mesmo gráfico onde colocamos "red" como parte da estética, porém agora, iremos mapeá-la com a coluna "qsec" de nossos dados mtcars:

```
ggplot(mtcars)+
  geom_point(aes(x = hp, y = mpg, color = qsec))
```



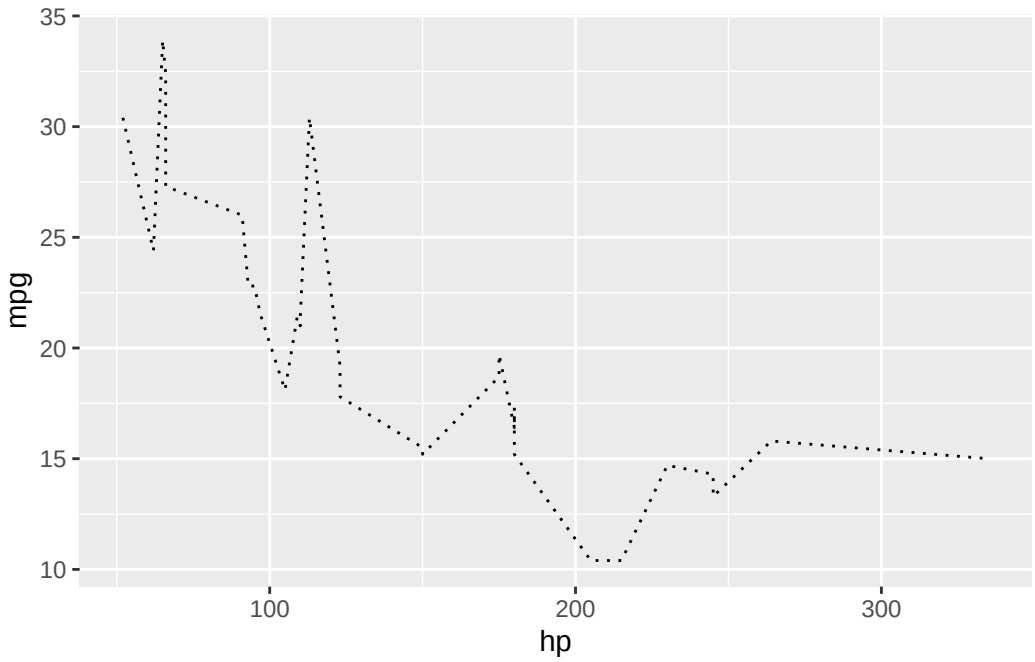
7.4.0.2 linetype

Use para definir a linha utilizada no gráfico. Podemos utilizar um número de 0 a 6 ou um nome:

(0 = "blank", 1 = "solid", 2 = "dashed", 3 = "dotted", 4 = "dotdash", 5 = "longdash",6 = "twodash")

Por exemplo, ao invés de utilizarmos a geometria de pontos (geom_point), vamos utilzar a geometria de linha (geom_line):

```
ggplot(mtcars)+
  geom_line(aes(x = hp, y = mpg) , linetype = "dotted")
```

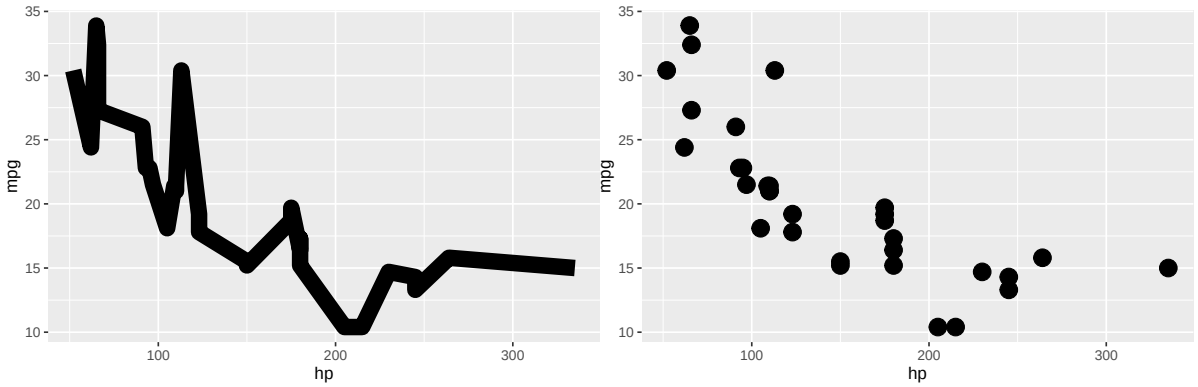


7.4.0.3 size

Use para definir o tamanho, no caso de uma linha em "mm".

Como dito anteriormente, estes valores são comuns para vários geometrias. Por exemplo, podemos utilizar o argumento size também na geometria de pontos (geom_point):

```
ggplot(mtcars)+
  geom_line(aes(x = hp, y = mpg), size = 5)
ggplot(mtcars)+
  geom_point(aes(x = hp, y = mpg), size = 5)
```

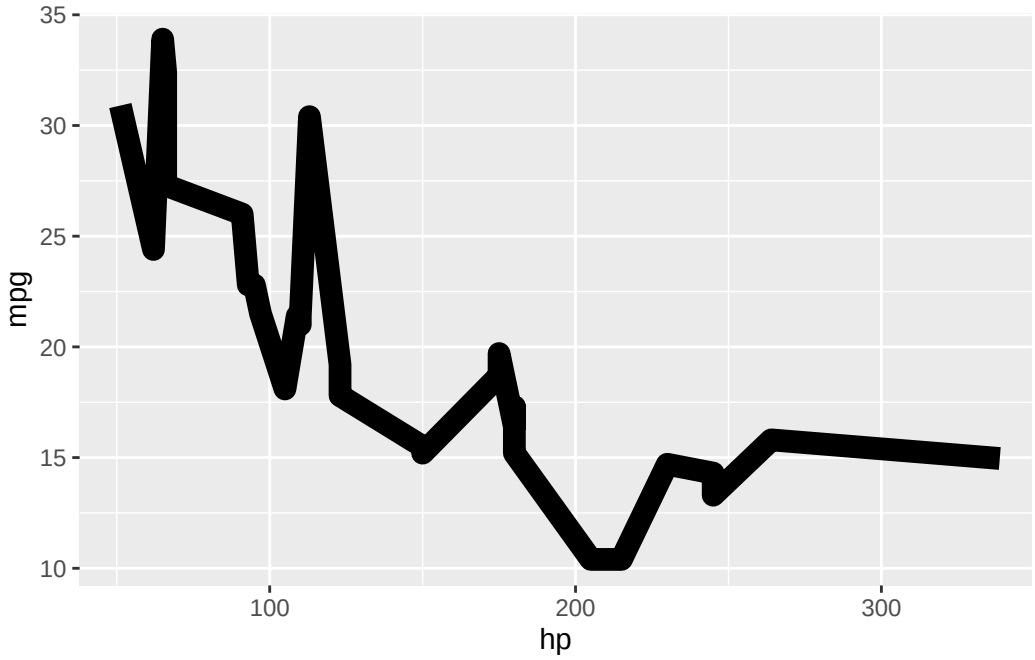


7.4.0.4 lineend

Use para definir o terminador da linha. Os valores podem ser: *round*, *butt* e *square*.

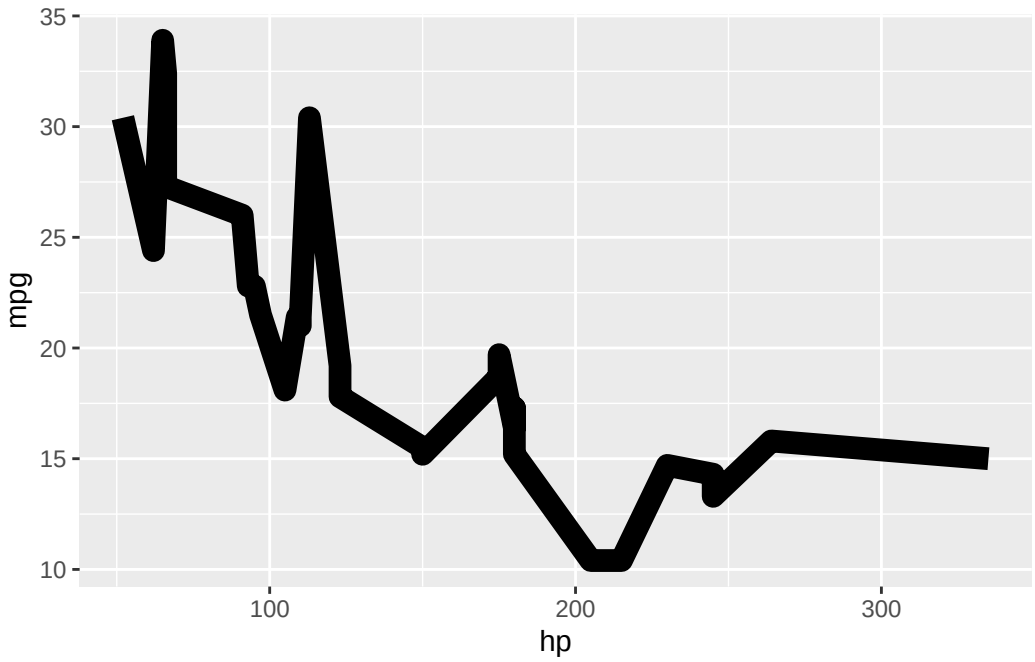
7.4.0.5 squared

```
ggplot(mtcars)+
  geom_line(aes(x = hp, y = mpg), lineend = "square", size = 4)
```



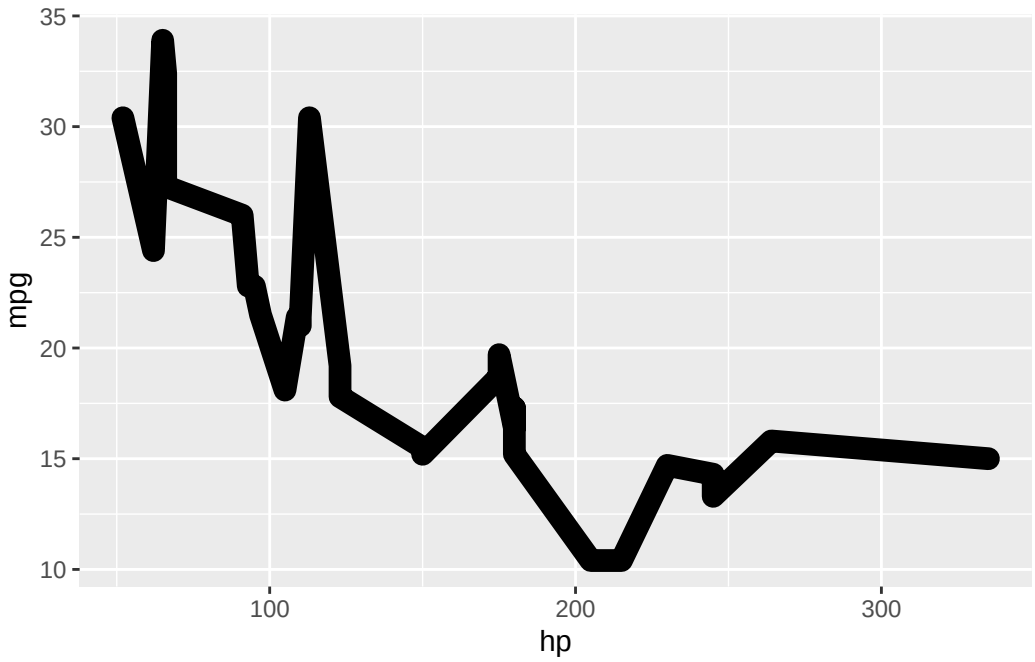
7.4.0.6 butt

```
ggplot(mtcars)+
  geom_line(aes(x = hp, y = mpg), lineend = "butt", size = 4)
```



7.4.0.7 round

```
ggplot(mtcars)+
  geom_line(aes(x = hp, y = mpg), lineend = "round", size = 4)
```

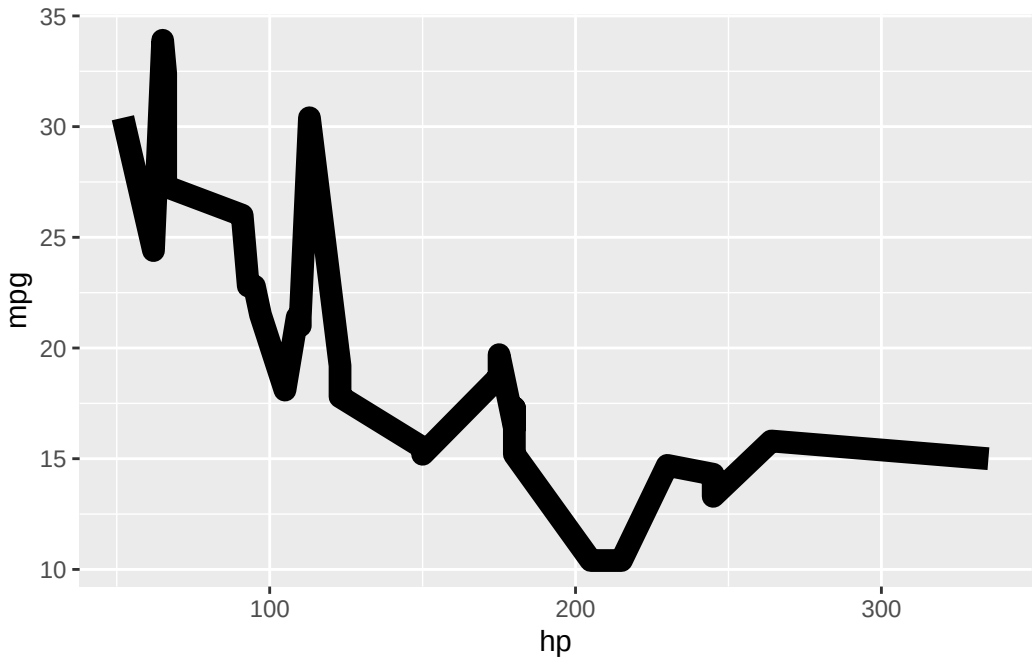


7.4.0.8 linejoin

Use para definir a junção da linha. Os valores podem ser: *round*, *mitre* e *bevel*.

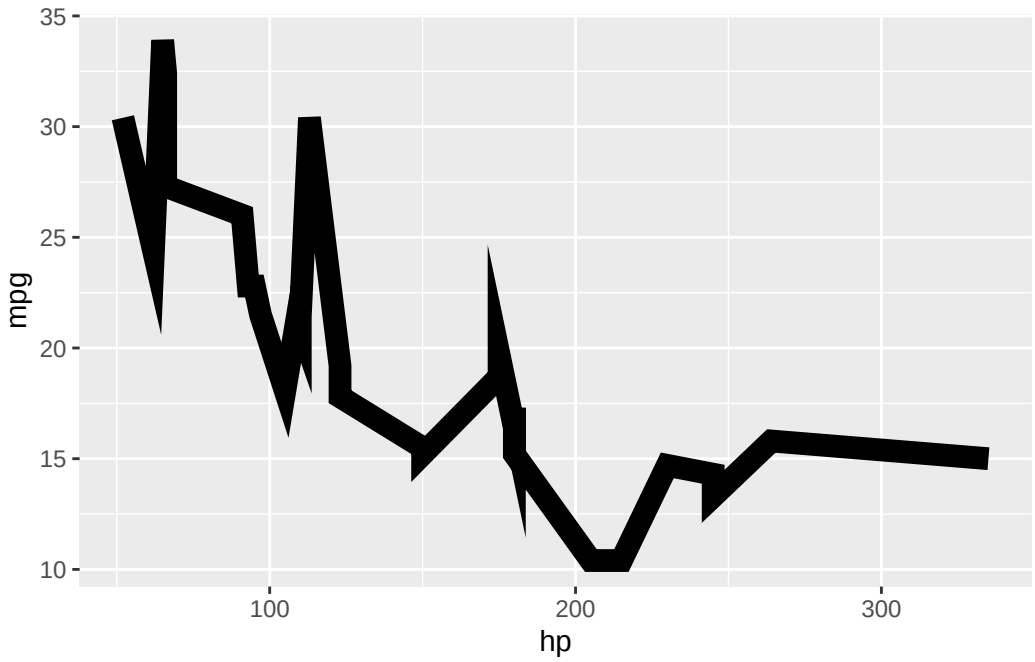
7.4.0.9 round

```
ggplot(mtcars)+
  geom_line(aes(x = hp, y = mpg), linejoin = "round", size = 4)
```



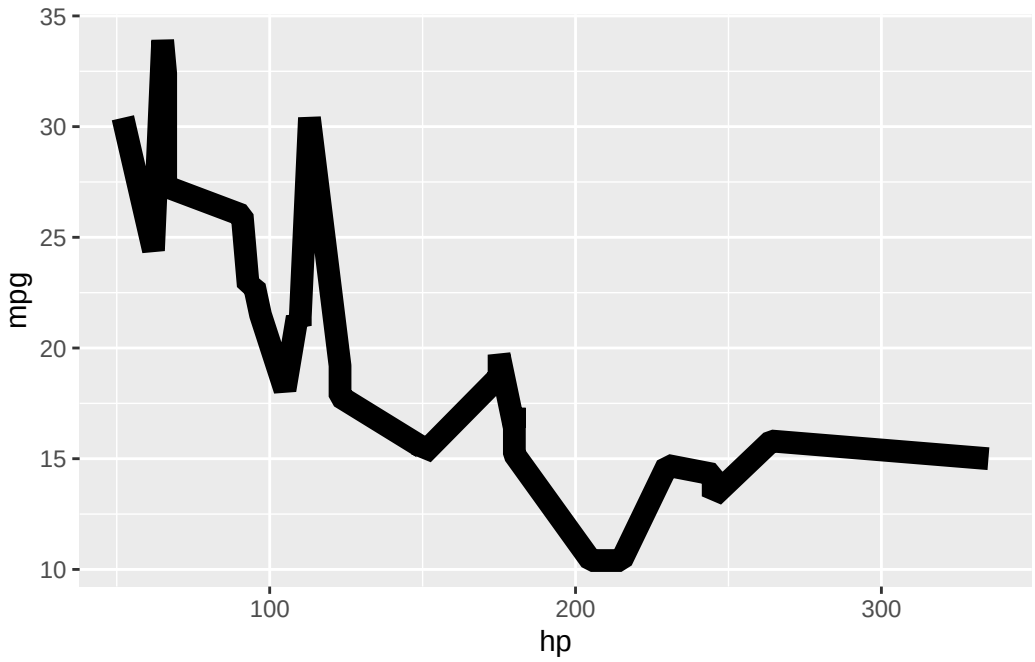
7.4.0.10 mitre

```
ggplot(mtcars)+
  geom_line(aes(x = hp, y = mpg), linejoin = "mitre", size = 4)
```



7.4.0.11 bevel

```
ggplot(mtcars)+
  geom_line(aes(x = hp, y = mpg), linejoin = "bevel", size = 4)
```



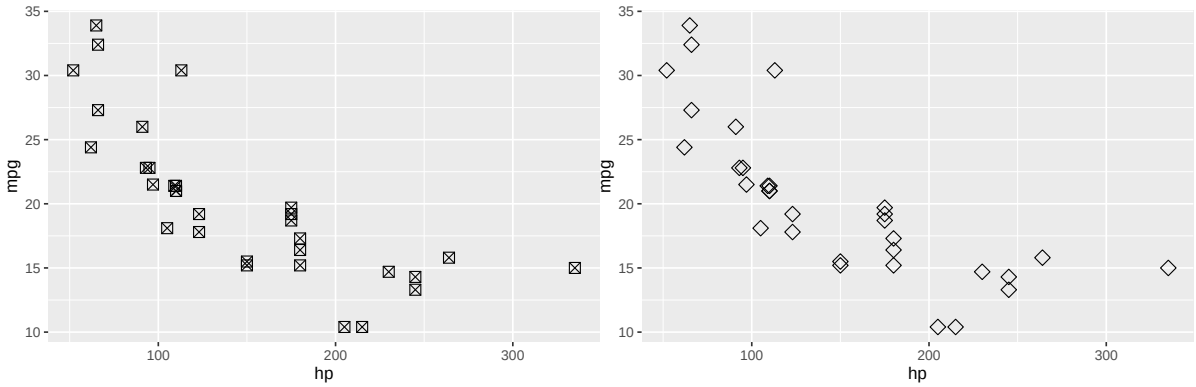
7.4.0.12 shape

Use para definir a forma. Podemos passar um número ou um nome em inglês da forma conforme abaixo:

```
ggplot(mtcars)+
  geom_point(aes(x = hp, y = mpg), shape = "square cross",  size = 3)
ggplot(mtcars)+
  geom_point(aes(x = hp, y = mpg), shape = 5, size = 3)
```

7.5 Geometrias (geoms)

Use as funções de geometria para representar os pontos de dados. Use as estéticas das geometrias para representar as variáveis.



> **Aviso**
>
> Lembre-se que cada função retornará uma camada, portanto, a ordem de criação das camadas afeta o resultado do gráfico, pois uma será criada sobre a anterior.

7.5.1 Gráficos Primitivos

Para os exemplos a seguir, utilizaremos duas bases de dados, uma chamada **economics** e outra **chamada** seals.

7.5.1.1 Economics

```
economics
```

```
# A tibble: 574 x 6
   date         pce    pop psavert uempmed unemploy
   <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
 1 1967-07-01  507. 198712    12.6     4.5     2944
 2 1967-08-01  510. 198911    12.6     4.7     2945
 3 1967-09-01  516. 199113    11.9     4.6     2958
 4 1967-10-01  512. 199311    12.9     4.9     3143
 5 1967-11-01  517. 199498    12.8     4.7     3066
 6 1967-12-01  525. 199657    11.8     4.8     3018
 7 1968-01-01  531. 199808    11.7     5.1     2878
 8 1968-02-01  534. 199920    12.3     4.5     3001
 9 1968-03-01  544. 200056    11.7     4.1     2877
10 1968-04-01  544  200208    12.3     4.6     2709
```

```
# ... with 564 more rows
# i Use `print(n = ...)` to see more rows
```

7.5.1.2 Seals

```
seals
```

```
# A tibble: 1,155 x 4
     lat  long delta_long delta_lat
   <dbl> <dbl>      <dbl>     <dbl>
 1  29.7 -173.     -0.915   0.143
 2  30.7 -173.     -0.867   0.128
 3  31.7 -173.     -0.819   0.113
 4  32.7 -173.     -0.771   0.0980
 5  33.7 -173.     -0.723   0.0828
 6  34.7 -173.     -0.674   0.0675
 7  35.7 -173.     -0.626   0.0522
 8  36.7 -173.     -0.577   0.0369
 9  37.7 -173.     -0.529   0.0216
10  38.7 -173.     -0.480   0.00635
# ... with 1,145 more rows
# i Use `print(n = ...)` to see more rows
```

Para simplificar o código dos próximos exemplos, iremos criar dois objetos gráfico do ggplot:

O primeiro chamado "**a**", irá conter um objeto ggplot com os dados de **economics**, mapeando a estética de x para a variável "date" e y para a variável "unemployed".

O segundo chamado "**b**", irá conter um objeto ggplot com os dados de **seals**, mapeando a estética de x para a variável "long" e y para a variável "lat".

Desta forma trabalharemos com os mesmos objetos base e adicionaremos apenas novas camadas afim de identificar os detalhes das diversas funções de geometria.

```
a <- ggplot(economics, aes(date, unemploy))
b <- ggplot(seals, aes(x = long, y = lat))
```

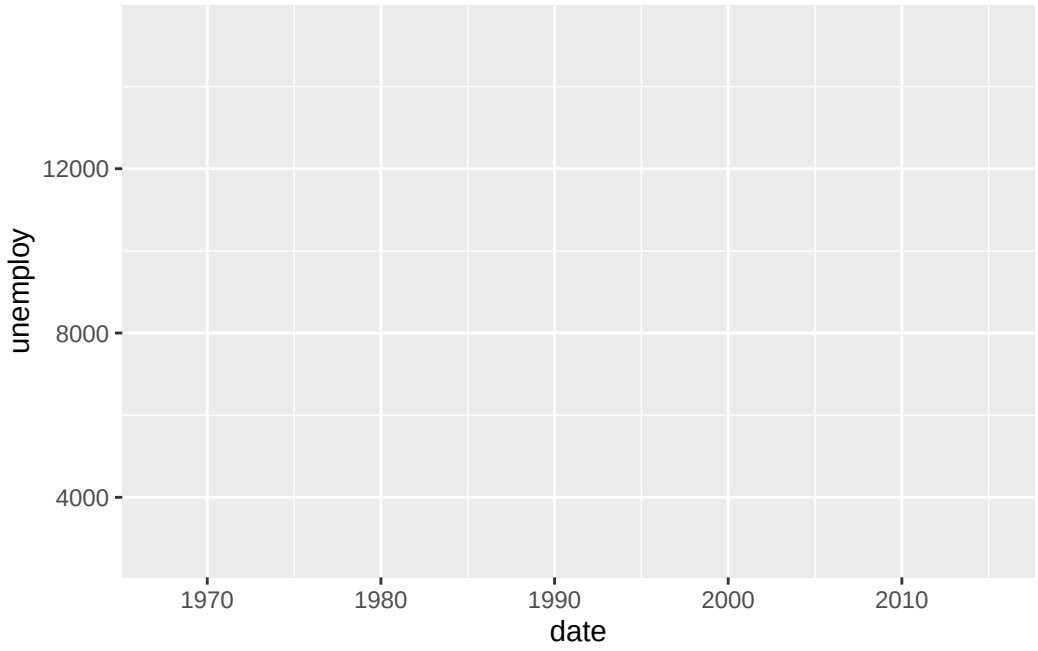
> **Nota**
>
> Para simplificar o código, em algum momentos podemos omitir o argumento, como data=, mapping = , x = , y = . Isto porque já estamos os valores na ordem da função.

Agora com os objetos inicialmente criados, iremos adicionar novas camadas afim de obtermos os gráficos e entendermos algumas das funções de geometria para gráficos primitivos.

7.5.1.3 geom_blank

Use para garantir limites para todos os gráficos de acordo com os dados.

```
a + geom_blank()
```

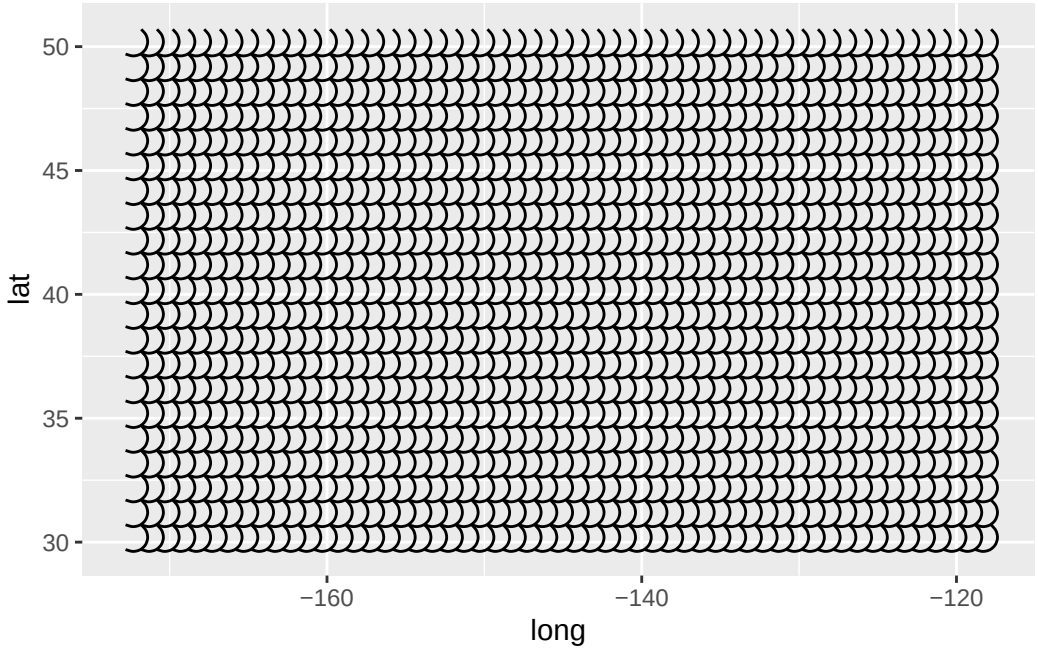


> Dica
>
> Use a função **expand_limits**() para expandir os limites do gráfico usando dados. Ver ?expand_limits para mais detalhes.

7.5.1.4 geom_curve

Use para criar curvas.

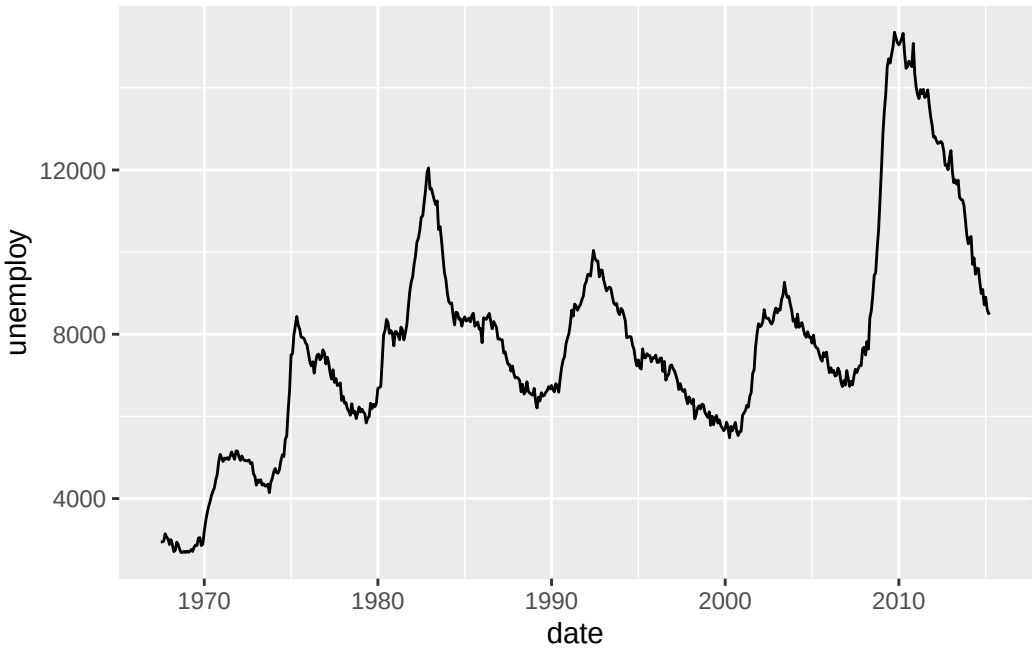
```
b + geom_curve(aes(yend = lat + 1, xend = long + 1), curvature = 1)
```



7.5.1.5 geom_path

Use para conectar observações.

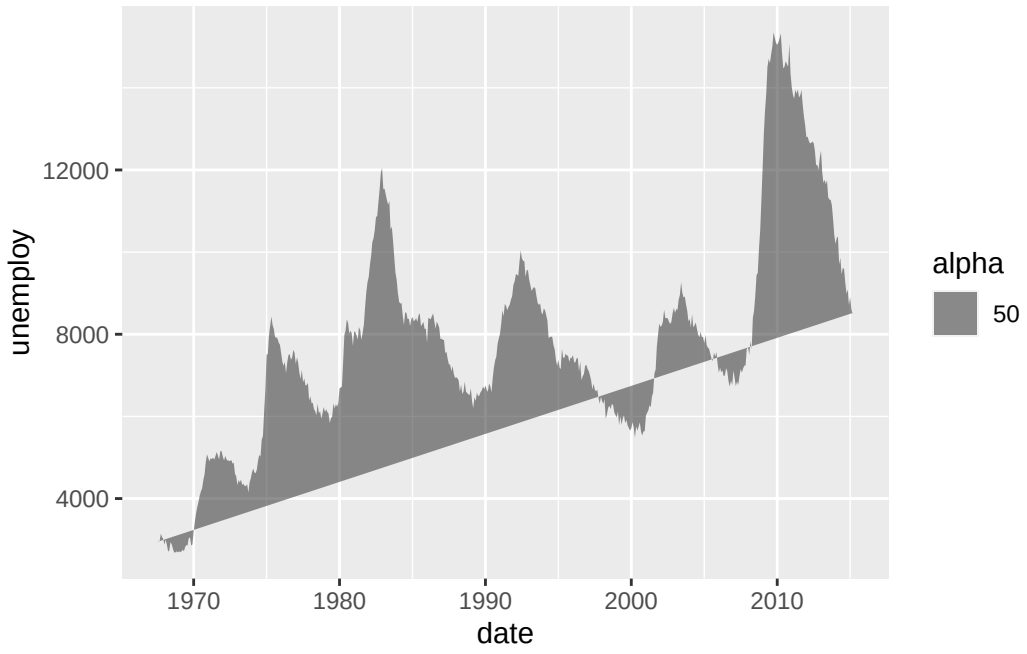
```
a + geom_path(lineend = "butt", linejoin = "round", linemitre = 1)
```



7.5.1.6 geom_polygon

Use para criar poligonos. É similar a geom_path(), porém a parte interna é preenchida com argumento fill =.

```
a + geom_polygon(aes(alpha = 50))
```

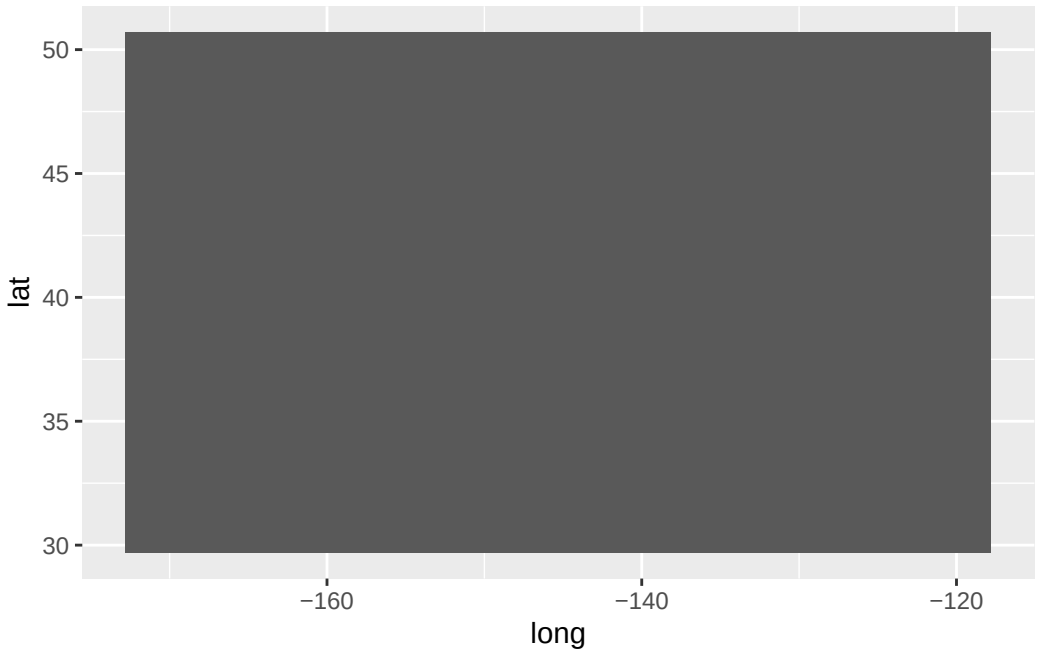


> **Nota**
>
> O argumento alpha = , é muito comum na definição de cores. De maneira simplificada, ele indica o nível de transparência de uma cor, sendo 0, nenhuma transparência e 1 transparência total.

7.5.1.7 geom_rect

Use para criar retângulos.

```
b + geom_rect(aes(xmin = long, ymin = lat,
xmax = long + 1, ymax = lat + 1))
```

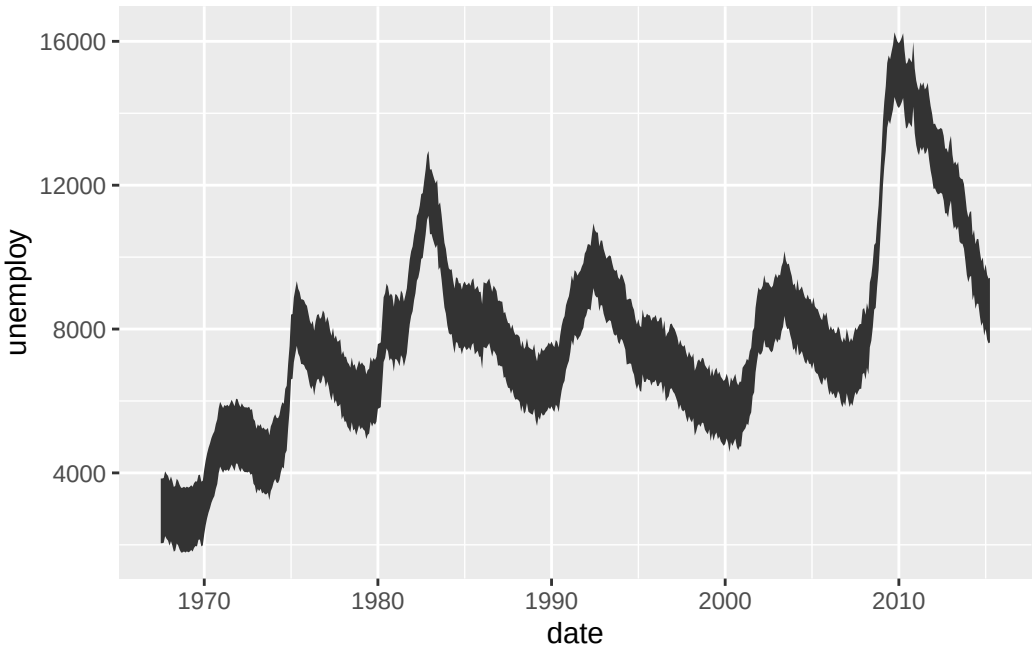


> **Nota**
>
> A função **geom_title()** fazem a mesma coisa, porém são parametrizadas de forma diferente. A geom_rect usa a localização dos 4 cantos do quadrado, enquanto a geom_tile usa o centro e seu tamanho (largura e altura) para dimensionar.

7.5.1.8 geom_ribbon

Use para criar uma "fita". Para cada valor de x, ela mostra um intervalo y, definido por ymin e ymax. Veja que geom_area() é um caso especial do geom_ribbon(), onde o ymin é fixo em 0 e y é usado no lugar de ymax.

```
a + geom_ribbon (aes(ymin = unemploy - 900,
ymax = unemploy + 900))
```



7.5.2 Segmentos de Linhas

Estas funções de geometrias, permitem criar diversos tipo de segmentos de linha e são úteis quando precisamos "marcar" ou definir algo no gráfico, com linhas verticais ou horizontais por exemplo.

7.5.2.1 geom_abline

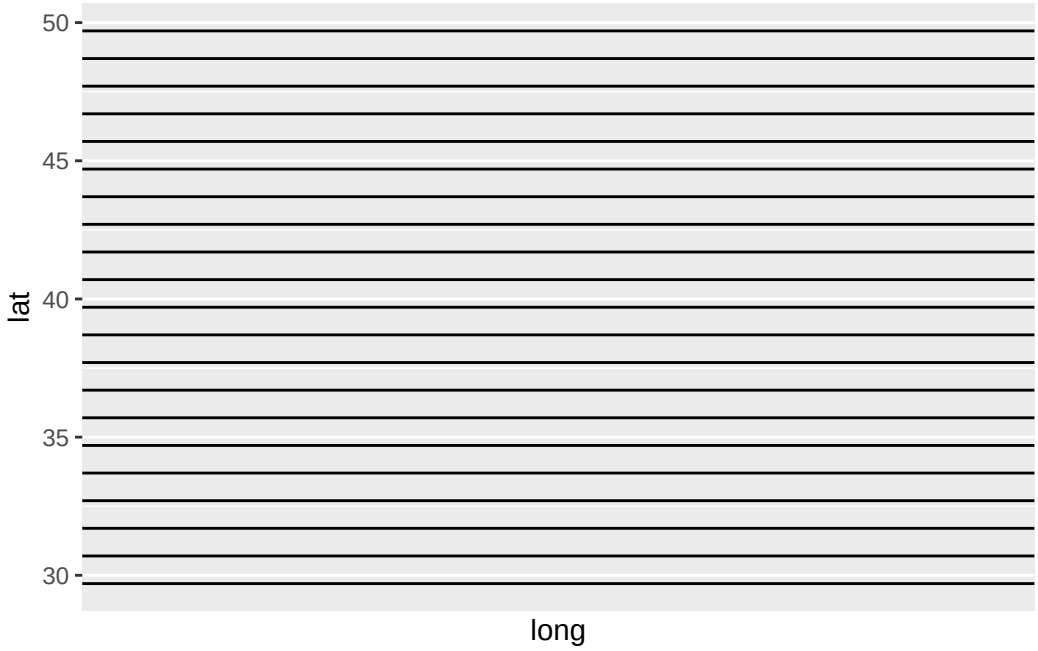
Use para criar uma linha de um ponto "a" até um ponto "b", passando o intercepto e a inclinação da reta.

```
b +  geom_abline(aes(intercept = 0, slope = 1))
```

7.5.2.2 geom_hline

Use para criar uma ou mais linhas horizontais.

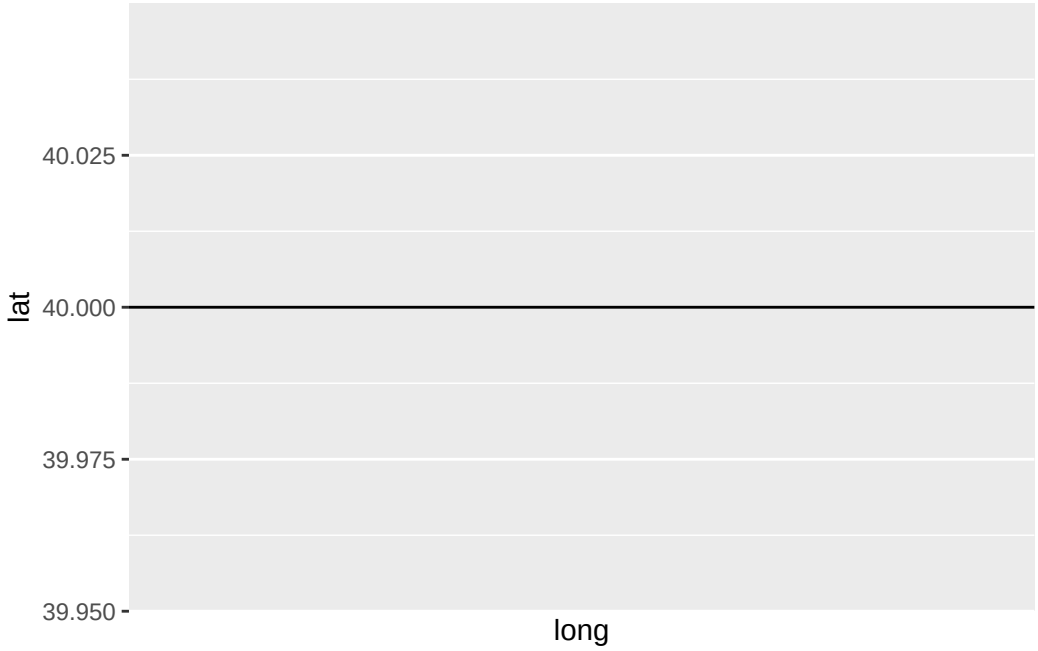
```
b + geom_hline(aes(yintercept = lat))
```



Veja que no exemplo anterior, fizemos o mapeamento do argumento yintercept como parte da estética (aes), mapeando este argumento com a variável "lat" de nossos dados.

Se quisermos simplesmente traçar uma linha com o intercepto do eixo y, correspondendo ao valor 40 (por exemplo, se fosse um tipo de meta), podemos definir este valor fora do mapeamento:

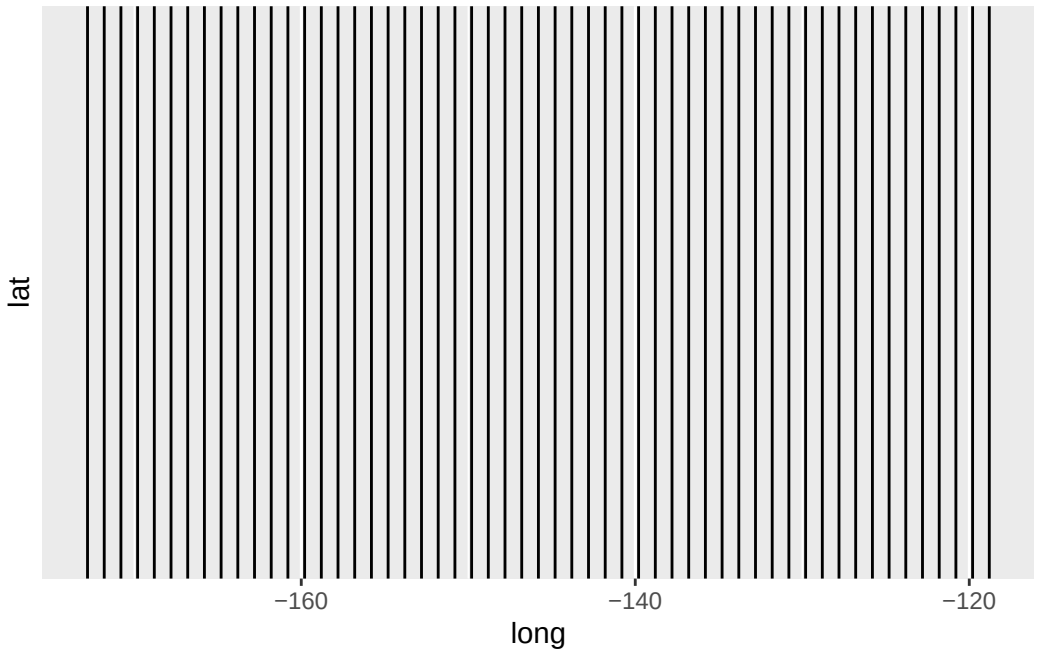
```
b + geom_hline(yintercept = 40)
```



7.5.2.3 geom_vline

Use para criar uma ou mais linhas horizontais.

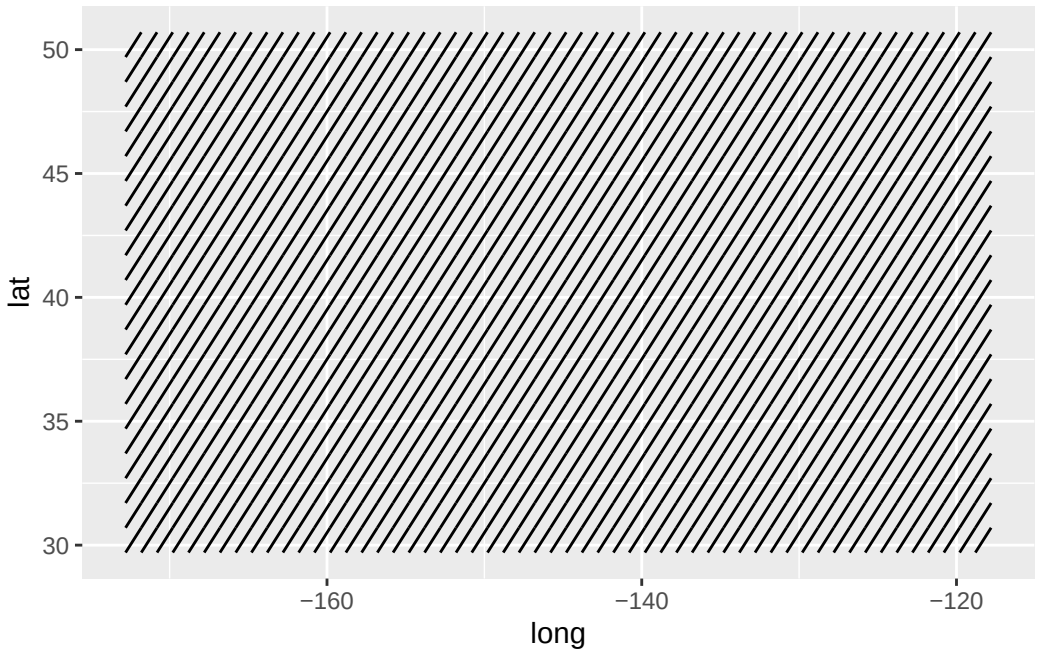
```
b + geom_vline(aes(xintercept = long))
```



7.5.2.4 geom_segment

É similar ao geom_curve, porém ao invés de criar uma curva, criar um segmento de reta.

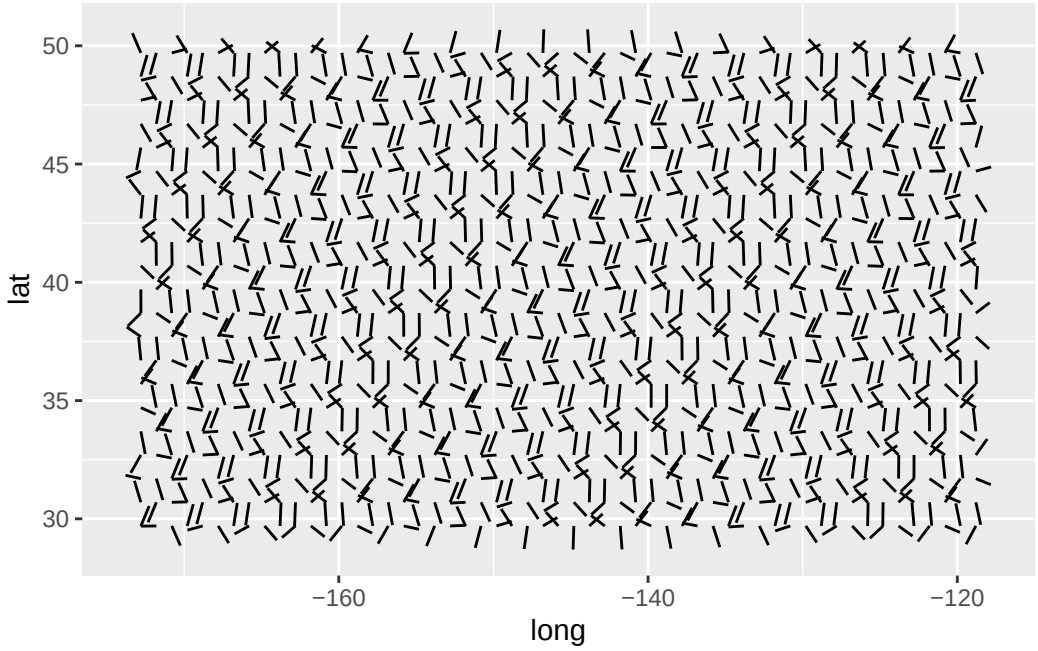
```
b + geom_segment(aes(yend = lat + 1, xend = long + 1))
```



7.5.2.5 geom_spoke

Use para criar um segmento, parametrizado pela localização, direção e distância.

```
b + geom_spoke(aes(angle = 1:1155, radius = 1))
```



7.5.3 Uma Variável Contínua

Novamente, iremos preparar nossos objetos básicos para evitar repetição de código. Agora teremos um objeto "**c**" e outro "**c2**", aos quais iremos adicionar novas camadas. Similar ao que fizemos anteriormente, porém agora utilizaremos a base de dados "**mpg**".

```
mpg
```

```
# A tibble: 234 x 11
   manufacturer model      displ  year   cyl trans drv     cty   hwy fl    class
   <chr>        <chr>      <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
 1 audi         a4           1.8  1999     4 auto~ f        18    29 p     comp~
 2 audi         a4           1.8  1999     4 manu~ f        21    29 p     comp~
 3 audi         a4           2    2008     4 manu~ f        20    31 p     comp~
 4 audi         a4           2    2008     4 auto~ f        21    30 p     comp~
 5 audi         a4           2.8  1999     6 auto~ f        16    26 p     comp~
 6 audi         a4           2.8  1999     6 manu~ f        18    26 p     comp~
 7 audi         a4           3.1  2008     6 auto~ f        18    27 p     comp~
 8 audi         a4 quattro   1.8  1999     4 manu~ 4        18    26 p     comp~
 9 audi         a4 quattro   1.8  1999     4 auto~ 4        16    25 p     comp~
10 audi         a4 quattro   2    2008     4 manu~ 4        20    28 p     comp~
# ... with 224 more rows
```

```
# i Use `print(n = ...)` to see more rows
```

Criando os objetos ggplot:

```
c <- ggplot(mpg, aes(hwy))
c2 <- ggplot(mpg)
```

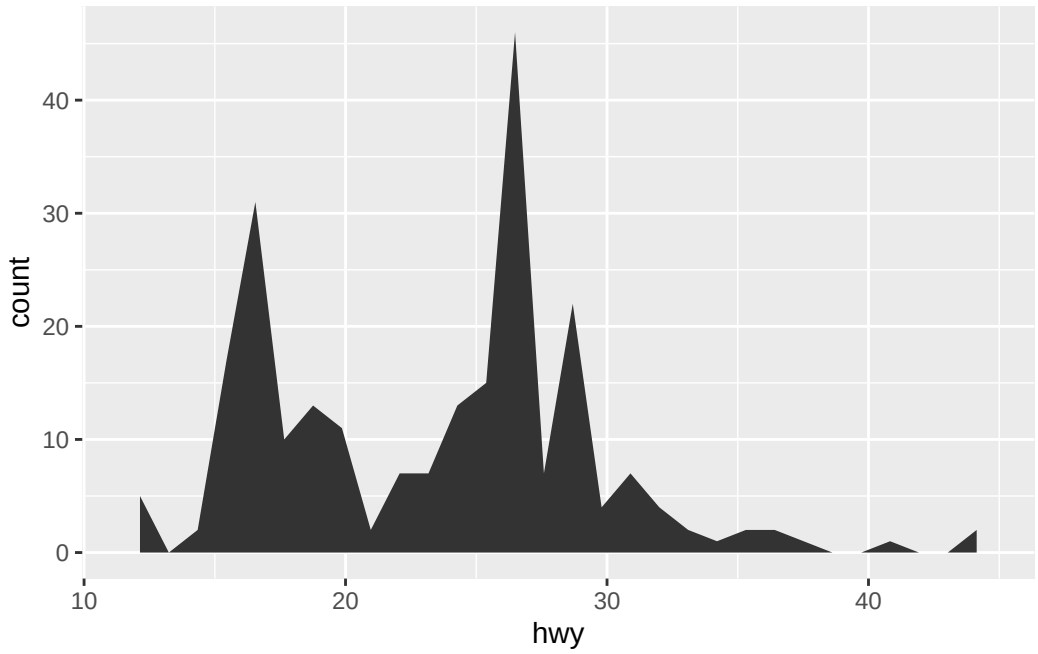
> **!** Importante
>
> Observe que o objeto "**c**", usa a base "mpg" **já atribuindo a variável "hwy" para x= da estética**. Esta configuração será aplicada para qualquer funções geom_*() a não ser que explicitamente tenhamos uma estética redefinindo isso dentro de outra camanda de geometria.
> Já para o objeto "**c2**", **não temos a estética** de x definida, portanto, teremos que definí-la em nas próximas camadas com a funções de geometria (geom_*()).

7.5.3.1 geom_area

Use para criar uma geometria de área.

> **i** Nota
>
> Você verá um argument (stat = "bin") abaixo. Veremos as **estatísticas** na seção ESTATÍSTICAS (STATS). Por hora, pense apenas que é uma forma de definirmos os valores que iremos mostrar para o eixo y. Por padrão o número de agrupamento (bin) é 30.

```
c + geom_area(stat = "bin")
```

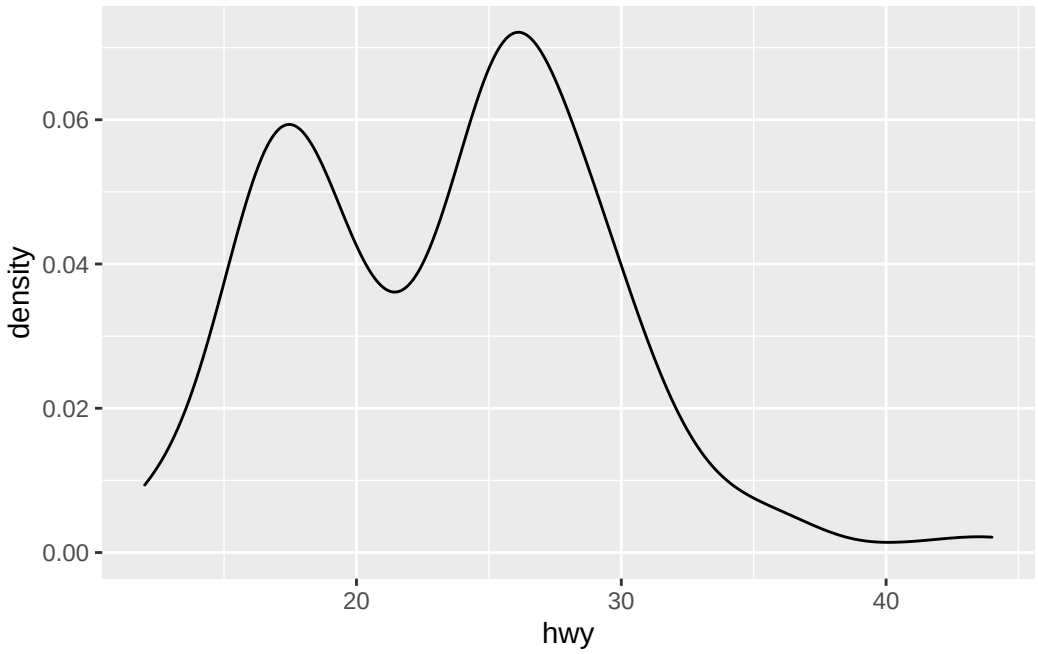


7.5.3.2 geom_density

Use para fazer um gráfico de **densidade** de um variável contínua.

Há a possibilidade de configurar vários kernels das funções de densidade, como “gaussian”, “rectangular”, “triangular”, “epanechnikov”, “biweight”, “cosine” ou “optcosine”.

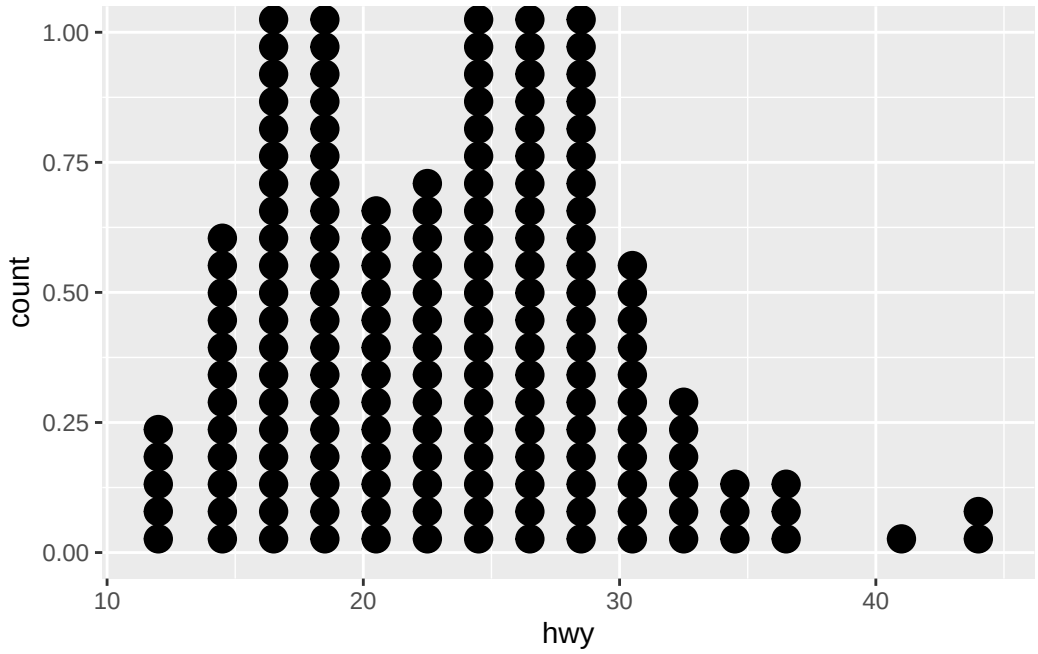
```
c + geom_density(kernel = "gaussian")
```



7.5.3.3 geom_dotplot

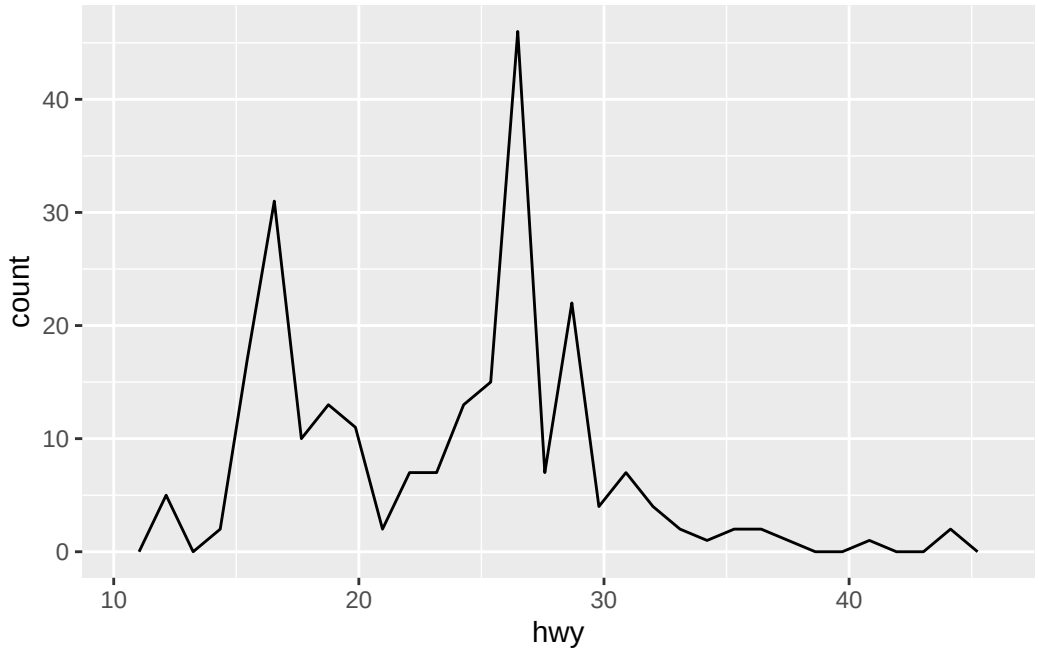
Use para criar uma geometria de pontos, com a largura dos pontos correspondem à largura do agrupamento e os pontos são empilhados. Cada ponto corresponde à uma observação dos dados.

```
c + geom_dotplot()
```



7.5.3.4 geom_freqpoly

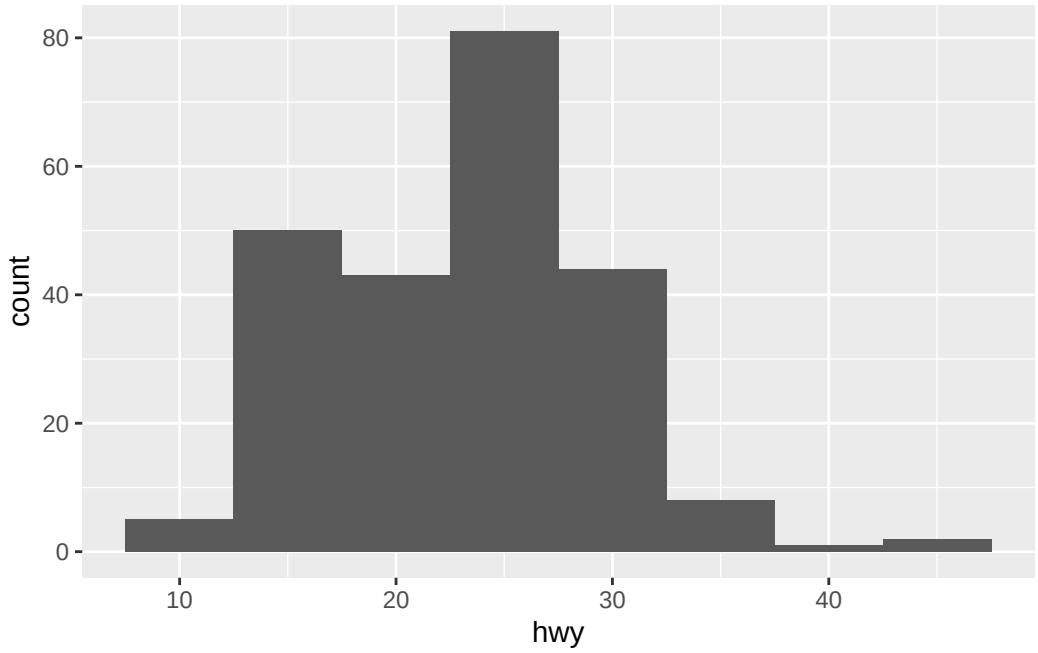
```
c + geom_freqpoly()
```



7.5.3.5 geom_histogram

Use para criar histogramas. O argumento binwidth = define o tamanho do agrupamento da variável.

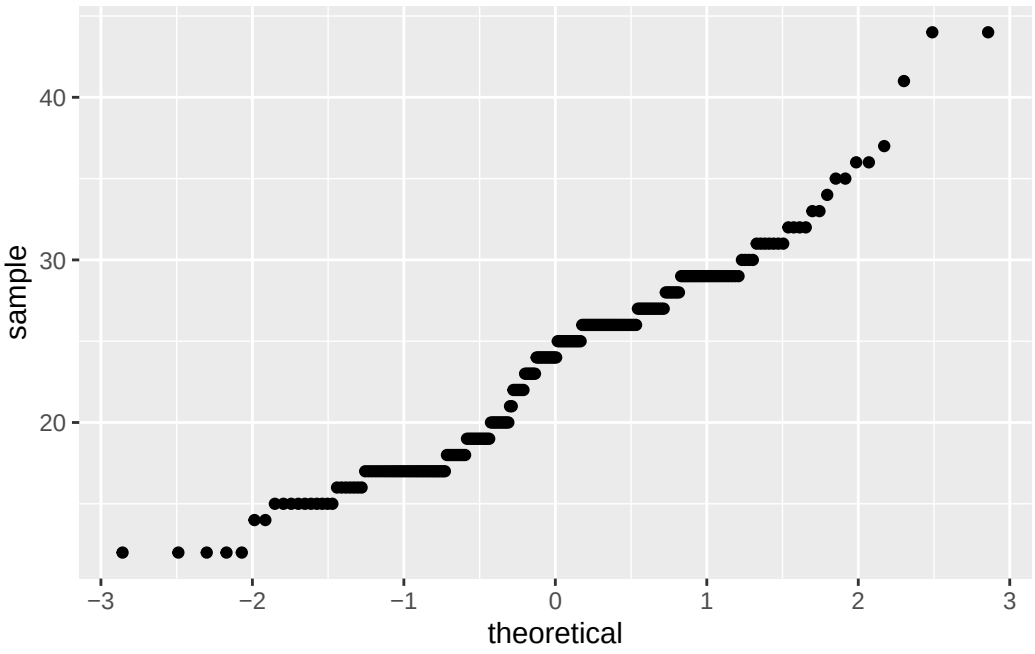
```
c + geom_histogram(binwidth = 5)
```



7.5.3.6 geom_qq

Use para criar um gráfico de quantil-quantil. Em geral utilizado para confirmar se uma variável tem uma distribuição gaussiana.

```
c2 + geom_qq(aes(sample = hwy))
```


40
30
20
sample
−3
−2
−1
0
1
2
3
theoretical