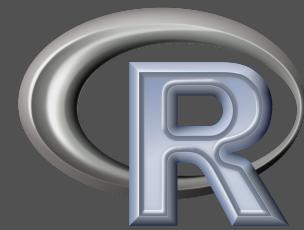




# ME115 - Linguagem R

Parte 08

1º semestre de 2023



# Manipulação de Dados no R

# Manipulação de Dados com `dplyr`

- O pacote `dplyr` é que temos hoje de mais moderno para limpeza, preparação e manipulação de dados no R.
- Gramática de manipulação de dados.
- Escrito por Hadley Wickham, autor também dos pacotes `readr`, `tidyverse`, `ggplot2`, entre outros.
- Funções mais rápidas que as da base do R, pois foram escritas em C e C++.
- Fáceis de entender, usar e lembrar.

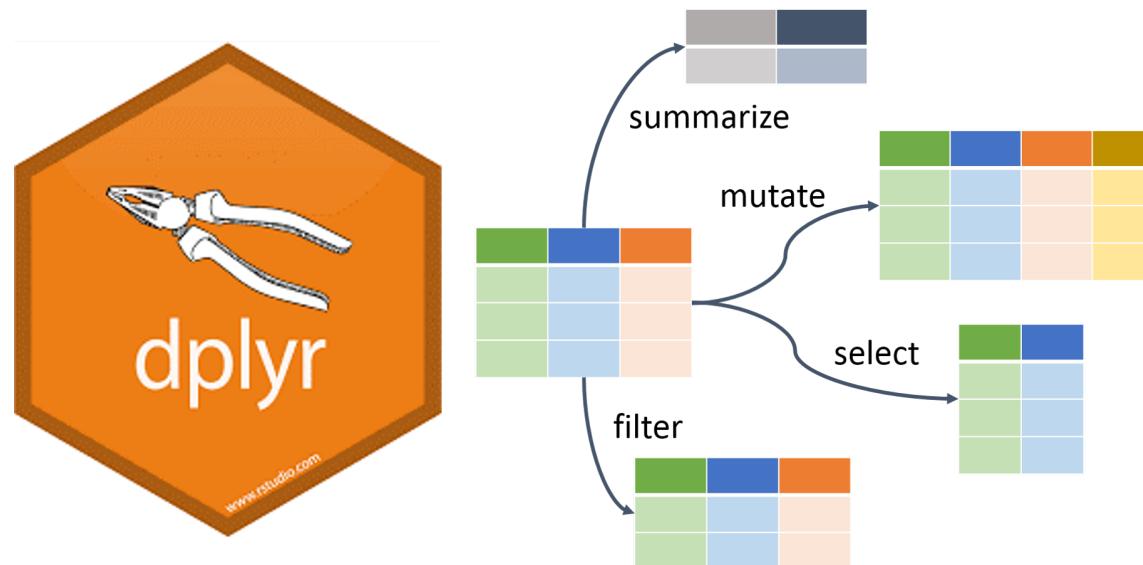


[ListenData.com](https://ListenData.com)



# Manipulação de Dados com **dplyr**

**dplyr**: conjunto de funções para realizar as tarefas mais comuns em preparação e manipulação de dados: filtros, seleção de variáveis, ordenar, agrregar dados, adicionar ou deletar colunas.



Fonte da Figura: [Towards Data Science - Data Manipulation in R with dplyr](#)

# Pipe

O operador pipe, representado pelo símbolo `%>%`, foi uma das grandes revoluções recentes do R, tornando a escrita e leitura de códigos mais intuitiva e compreensível.

Esse operador faz parte do pacote `magrittr`, que contém outros operadores também. Para usá-lo basta carregar o pacote `magritrr` ou o pacote `dplyr`.

**Exemplo:** vamos calcular a raíz quadrada da soma de um vetor.

```
sqrt(sum(1:10)) # da maneira convencional  
1:10 %>% sum() %>% sqrt() # usando pipe
```



Fonte da Figura: [Curso-R \(Figura 6.1\) - Reprodução do quadro La Trahison des images do pintor René Magritte](#)

# Pipe

De maneira bem simples, o pipe usa o resultado da expressão do lado esquerdo como primeiro argumento da função do lado direito e assim sucessivamente.



```
sqrt(sum(1:10)) # da maneira convencional  
1:10 %>% sum() %>% sqrt() # usando pipe
```

Usando o pipe, podemos executar uma série de operações em sequência, enviando os resultados de uma função para outra:

```
objeto %>%  
  funcao_1() %>%  
  funcao_2() %>%  
  ...  
  funcao_n()
```

# Tibble

Como já vimos na aula anterior, *tibbles* são tipos especiais de data frames e são os objetos resultantes das funções do pacote **dplyr**.

Uma função muito utilizada para ver a estrutura de um tibble é `glimpse()`:

```
iris_tbl <- as_tibble(iris)  
glimpse(iris_tbl)
```

```
## Rows: 150
## Columns: 5
## $ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, 5.4, 4...
## $ Sepal.Width   <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7, 3...
## $ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5, 1...
## $ Petal.Width   <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2, 0...
## $ Species       <fct> setosa, setosa, setosa, setosa, setosa, setosa, setosa, setosa, s...
```

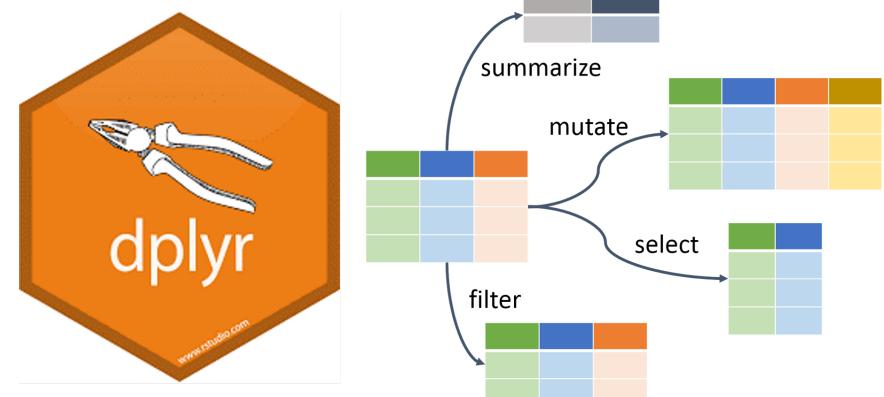
Veja que a função `glimpse()` do `dplyr` é a versão melhorada de `str()`.

# Principais Verbos do `dplyr`

As funções do `dplyr` são chamadas de verbos e os principais verbos são:

- `select()`: seleciona um subconjunto de colunas
- `filter()`: seleciona um subconjunto de linhas pelos seus valores
- `mutate()`: cria novas colunas baseada em colunas existentes
- `summarize()`: resume vários valores em um só
- `arrange()`: modifica a ordem das linhas

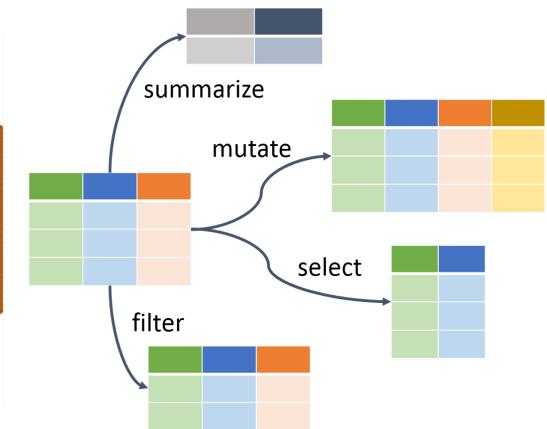
Esses verbos podem ser usados com `group_by()`, que muda o escopo de cada função para operar em subgrupos dos dados de acordo com os valores de uma ou mais variáveis.



# Verbos do `dplyr`

Todos os verbos funcionam de maneira semelhante:

- O primeiro argumento é um data frame/*tibble*.
- Os argumentos subsequentes descrevem o que fazer com o data frame.
- O resultado é um novo *tibble*.



Como o primeiro argumento é um *tibble*, o uso do operador `%>%` facilita a escrita de uma sequência de operações para realizar a preparação e manipulação dos dados (*data wrangling*).

Juntas, essas propriedades facilitam a interligação de várias etapas simples para alcançar um resultado complexo!

# Exemplo `nycflights13`

Como exemplo, iremos usar os dados `flights` do pacote `nycflights13`.

Esses dados contêm todos os 336.776 voos que decolaram da cidade de New York em 2013 e 19 variáveis. Aqui mostramos apenas as primeiras 7 colunas:

```
library(nycflights13)
```

# Selecionar colunas com `select()`

Dados podem ter muitas colunas, mas nem todas são de interesse.

```
flights %>% names()
```

```
## [1] "year"          "month"         "day"           "dep_time"  
## [5] "sched_dep_time" "dep_delay"      "arr_time"       "sched_arr_time"  
## [9] "arr_delay"      "carrier"        "flight"        "tailnum"  
## [13] "origin"         "dest"          "air_time"      "distance"  
## [17] "hour"          "minute"        "time_hour"
```

Como selecionar uma ou mais colunas por nome?

```
select(flights, year, month, day) # modo tradicional  
flights %>% select(year, month, day) # usando pipe
```

Daqui em diante, usaremos sempre o operador `%>%`.

# Selecionar/Excluir colunas com `select()`

Como selecionar uma ou mais colunas por nome?

```
# Selecionar todas as colunas entre year e day (inclusive)
flights %>% select(year:day)
```

Como excluir uma ou mais colunas?

Para isso, usamos o sinal de “-” antes dos nomes das colunas que queremos excluir.

```
# Excluir colunas por nome
flights %>% select(-year)
```

```
# Excluir todas as colunas year e day (inclusive)
flights %>% select(-(year:day))
```

# Selecionar/Excluir colunas - Base do R

Na base do R, faríamos:

```
# Selecionar as 3 primeiras colunas pelos nomes
flights[, c("year", "month", "day")]
flights %>% subset(select = c("year", "month", "day"))

# Selecionar todas as colunas entre year e day (inclusive)
flights %>% subset(select = year:day)
flights[, 1:3]

# Excluir colunas
flights[, -"year"]
#> Error in -"year" : invalid argument to unary operator

flights[, -1]
flights$year <- NULL
```

# Selecionar Colunas - Funções Auxiliares

O `dplyr` possui um conjunto de funções auxiliares muito úteis para seleção de colunas baseadas em seus nomes.

As principais funções estão na tabela ao lado.

Falaremos sobre expressões regulares na próxima aula.

Por exemplo, vamos selecionar todas as variáveis que contenham “time” em seus nomes:

Helpers	Description
<code>starts_with()</code>	Starts with a prefix
<code>ends_with()</code>	Ends with a prefix
<code>contains()</code>	Contains a literal string
<code>matches()</code>	Matches a regular expression
<code>num_range()</code>	Numerical range like x01, x02, x03.
<code>one_of()</code>	Variables in character vector.
<code>everything()</code>	All variables.

```
flights %>% select(contains("time")) %>% names()
```

```
## [1] "dep_time"          "sched_dep_time"   "arr_time"        "sched_arr_time"  
## [5] "air_time"          "time_hour"
```

# Selecionar linhas com `filter()`

Permite selecionar um subconjunto de observações baseada nos seus valores.

Por exemplo, podemos selecionar todos os voos de 01/Janeiro/2013.

```
flights %>% filter(month == 1, day == 1)
```

```
## # A tibble: 5 × 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>     <int>           <int>     <dbl>    <int>           <int>
## 1 2013     1     1      517            515        2       830           819
## 2 2013     1     1      533            529        4       850           830
## 3 2013     1     1      542            540        2       923           850
## 4 2013     1     1      544            545       -1      1004          1022
## 5 2013     1     1      554            600       -6      812           837
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

# Operadores de Comparações

Para usar o verbo `filter()` de maneira mais efetiva, você deve saber como usar os operadores de comparação.

Comparações: `>`, `>=`, `<`, `<=`, `!=` (not equal), e `==` (equal).

Um erro comum:

```
filter(flights, month = 1)
#> Error: Problem with `filter()` input `..1`.
#> ✘ Input `..1` is named.
#> i This usually means that you've used `=` instead of `==`.
#> i Did you mean `month == 1`?
```

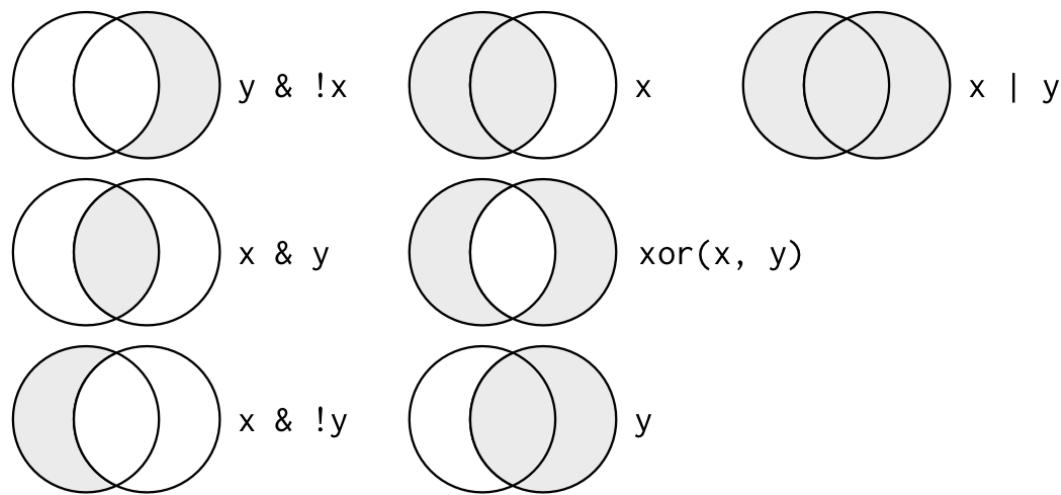
Use `==` em vez de `=` para testar igualdade.

```
filter(flights, month == 1)
```

# Operadores Lógicos

Múltiplos argumentos no verbo `filter()`, separados por vírgula, são combinados com o operador lógico `&` (*and* ou “e”).

Para outros tipos de combinações, você irá utilizar outros operadores lógicos, como ilustrado abaixo.



Fonte: [Figura 5.1 - R for Data Science](#): conjunto completo de operações lógicas.

# Comparações e Operadores Lógicos

Por exemplo, encontrar todos os voos em novembro ou dezembro.

```
flights %>% filter(month == 11 | month == 12) # CORRETO!!!
flights %>% filter(month == 11 | 12) # ERRO!!!
```

Veja que a ordem dos operadores lógicos na linguagem de programação não funciona como na nossa linguagem!

Um operador útil para os casos como o acima é usar `%in%`:

```
nov_dec <- flights %>% filter(month %in% c(11, 12))
```

Você pode armazenar o objeto resultante em um novo objeto, como fizemos com `nov_dec`.

Mas lembre-se, só armazene o que for realmente necessário!

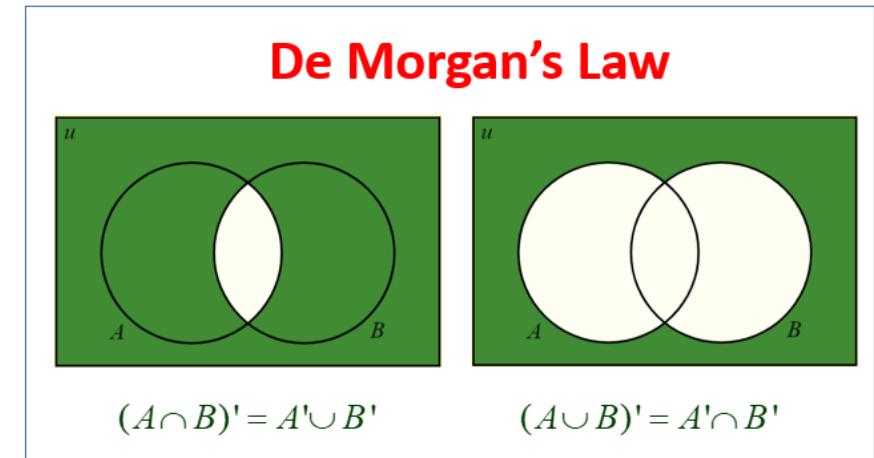
# Simplifique as Regras

Tome cuidado para não complicar demais as regras!!!

Você pode simplificar regras complicadas lembrando das leis de De Morgan.

- $!(x \ \& \ y)$  é o mesmo que  $!x \ | \ !y$
- $!(x \ | \ y)$  é o mesmo que  $!x \ \& \ !y$

**Exemplo:** encontrar voos que não atrasaram (na chegada ou na partida) em mais de duas horas:



```
flights %>% filter(!(arr_delay > 120 | dep_delay > 120))
flights %>% filter(arr_delay <= 120, dep_delay <= 120) # menos complicado!
```

# Subconjuntos - Base do R

O equivalente do verbo `filter()` na base do R é a função `subset()`.

```
flights %>% filter(month == 1)  
flights %>% subset(month == 1)
```

Combinando várias condições:

```
flights %>% filter(arr_delay <= 120, dep_delay <= 120)  
flights %>% subset(arr_delay <= 120 & dep_delay <= 120)
```

A função `subset()` tem dois principais argumentos:

- `subset`: expressão lógica indicando quais **linhas** iremos selecionar
- `select`: expressão indicando quais **colunas** queremos selecionar de um data frame.

# Ordenar os dados usando `arrange()`

O verbo `arrange()` ordena os dados em ordem crescente (*default*) de acordo com uma ou mais variáveis.

```
flights %>% arrange(dep_delay)
```

```
## # A tibble: 6 × 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>     <int>          <int>     <dbl>     <int>          <int>
## 1 2013     12     7      2040            2123      -43       40          2352
## 2 2013      2     3      2022            2055      -33      2240          2338
## 3 2013     11    10      1408            1440      -32      1549          1559
## 4 2013      1    11      1900            1930      -30      2233          2243
## 5 2013      1    29      1703            1730      -27      1947          1957
## 6 2013      8     9       729             755      -26     1002          955
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

# Ordenar os dados usando `arrange()`

Use `desc()` para ordenar em ordem decrescente:

```
flights %>% arrange(desc(dep_delay))
```

```
## # A tibble: 6 × 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>     <int>           <int>     <dbl>     <int>           <int>
## 1 2013     1     9       641            900    1301    1242           1530
## 2 2013     6    15      1432           1935    1137    1607           2120
## 3 2013     1    10      1121           1635    1126    1239           1810
## 4 2013     9    20      1139           1845    1014    1457           2210
## 5 2013     7    22       845           1600    1005    1044           1815
## 6 2013     4    10      1100           1900     960    1342           2211
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

# Criar novas variáveis com `mutate()`

`mutate()`: usa funções que recebem um vetor de valores e retornam um outro vetor de valores. Por isso, é o verbo utilizado para criar novas variáveis.

Vamos criar três variáveis novas (`gain`, `hours` e `gain_per_hour`):

```
flights_sml <- flights %>%
  select(year:day, ends_with("delay"), distance, air_time) %>%
  mutate(gain = dep_delay - arr_delay,
         hours = air_time / 60,
         gain_per_hour = gain / hours)

flights_sml %>% names()
```

```
## [1] "year"          "month"        "day"           "dep_delay"
## [5] "arr_delay"      "distance"      "air_time"       "gain"
## [9] "hours"          "gain_per_hour"
```

# Criar novas variáveis com `transmute()`

Se você quiser manter apenas as novas variáveis, use `transmute()`:

```
flights %>%
  transmute(gain = dep_delay - arr_delay,
            hours = air_time / 60,
            gain_per_hour = gain / hours) %>%
  head()
```

```
## # A tibble: 6 × 3
##      gain  hours gain_per_hour
##   <dbl> <dbl>      <dbl>
## 1    -9   3.78     -2.38
## 2   -16   3.78     -4.23
## 3   -31   2.67    -11.6
## 4    17   3.05      5.57
## 5    19   1.93      9.83
## 6   -16   2.5       -6.4
```

# Resumir variáveis com `summarise()`

Esse verbo cria um *tibble* com uma única linha contendo as medidas resumidas de uma ou mais variáveis:

```
flights %>% summarise(delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 1 × 1
##   delay
##   <dbl>
## 1 12.6
```

`summarise()` não é muito útil isoladamente, mas é uma ferramenta bem poderosa quando combinado com `group_by()`.

`group_by()`: altera a unidade de análise dos dados completos para subgrupos de acordo com valores de uma ou mais variáveis.

# Exemplo usando `group_by()` e `summarise()`

Por exemplo, se queremos o atraso médio na decolagem para os dados agrupados por data (ano, mês e dia), obtemos:

```
by_day <- flights %>%
  group_by(year, month, day) %>%
  summarise(delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 6 × 4
## # Groups:   year, month [1]
##   year month   day delay
##   <int> <int> <int> <dbl>
## 1 2013     1     1  11.5
## 2 2013     1     2  13.9
## 3 2013     1     3  11.0
## 4 2013     1     4   8.95
## 5 2013     1     5   5.73
## 6 2013     1     6   7.15
```

# summarize()

`summarize()`: pode usar qualquer função sumária, ou seja, funções que recebem um vetor como argumento e retornam um único número.

Algumas funções sumárias do R que podem ser usadas:

- `min()` e `max()` - valor mínimo e máximo de um vetor
- `mean()` - média de um vetor
- `median()` - mediana de um vetor
- `quantile(x, p)` - p-ésimo quantil de um vetor x
- `sd()` - desvio padrão de um vetor
- `var()` - variância de um vetor
- `IQR()` - Distância Inter Quartílica (IQR) de um vetor
- `diff(range())` - amplitude de um vetor

# summarize()

dplyr tem algumas funções sumárias próprias que são úteis:

- `first()` e `last()` - primeiro e último elemento de um vetor
- `nth(x, n)` - n-ésimo elemento de um vetor x
- `n()` - número de elementos em um vetor
- `n_distinct()` - número de valores distintos em um vetor



# Exemplo usando `group_by()` e `summarise()`

Por exemplo, se queremos o número de voos, a distância média e o atraso médio por destino, e retornar os três destinos com maior atraso médio.

```
flights %>%
  group_by(dest) %>%
  summarise(count = n(),
            dist = mean(distance, na.rm = TRUE),
            delay = mean(arr_delay, na.rm = TRUE)) %>%
  slice_max(delay, n=3)
```

```
## # A tibble: 3 × 4
##   dest  count  dist delay
##   <chr> <int> <dbl> <dbl>
## 1 CAE     116  604.  41.8
## 2 TUL     315 1215   33.7
## 3 OKC     346 1325   30.6
```

# Outras funções do `dplyr`

Não cobrimos todas as funções do `dplyr` nessa aula, mas vejam o Cheat Sheet do `dplyr`.

Algumas funções a serem exploradas:

- `mutate_each()`: aplica uma função para cada coluna especificada
- `summarise_each()`: aplica uma função sumária para cada coluna especificada
- `across()`: facilita aplicar a mesma transformação em várias colunas ao mesmo tempo.
- `slice()` e suas variações `slice_*`: seleciona linhas pelos seus índices
- `sample_n()` e `sample_frac()`: retira uma amostra aleatória dos dados



# Data transformation with dplyr - Cheat Sheet

## Data transformation with dplyr :: CHEAT SHEET



dplyr functions work with pipes and expect **tidy data**. In tidy data:



Each variable is in its own column

&



Each **observation**, or **case**, is in its own row



$x \rightarrow f(y)$  becomes  $f(x, y)$

### Summarize Cases

Apply **summary** functions to columns to create a new table of summary statistics. Summary functions take vectors as input and return one value (see back).

summary function ➔



`summarize(data, ...)`

Compute table of summaries.

`mtcars %> summarize(avg = mean(mpg))`

`count(data, ..., wt = NULL, sort = FALSE, name = NULL)` Count number of rows in each group defined by the variables in ... Also `tally()`, `add_count()`,

`add_tally()`.

`mtcars %> count(cyl)`

### Group Cases

Use `group_by(data, ...)`, `add = FALSE`, `drop = TRUE` to create a "grouped" copy of a table grouped by columns in ... dplyr functions will manipulate each "group" separately and combine the results.



`mtcars %> group_by(cyl) %> summarize(avg = mean(mpg))`

Use `rowwise_(data, ...)` to group data into individual rows. dplyr functions will compute results for each row. Also apply functions to list-columns. See tidyR cheat sheet for list-column workflow.



`starwars %> rowwise_() %> mutate(film_count = length(films))`

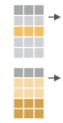
`ungroup(x, ...)` Returns ungrouped copy of table.  
`g_mtcars <- mtcars %> group_by(cyl)`  
`ungroup(g_mtcars)`



### Manipulate Cases

#### EXTRACT CASES

Row functions return a subset of rows as a new table.



`filter(data, ..., .preserve = FALSE)` Extract rows that meet logical criteria.  
`mtcars %> filter(mpg > 20)`



`distinct(data, ..., .keep_all = FALSE)` Remove rows with duplicate values.  
`mtcars %> distinct(cyl)`



`slice(data, ..., .preserve = FALSE)` Select rows by position.  
`mtcars %> slice(10:15)`



`slice_sample(data, ..., n, prop, weight, by = NULL, replace = FALSE)` Randomly select rows. Use n to select a number of rows and prop to select a fraction of rows.  
`mtcars %> slice_sample(n = 5, replace = TRUE)`



`slice_min(data, order_by, ..., n, prop, with_ties = TRUE) and slice_max()` Select rows with the lowest and highest values.  
`mtcars %> slice_min(mpg, prop = 0.25)`



`slice_head(data, ..., n, prop) and slice_tail()` Select the first or last rows.  
`mtcars %> slice_head(n = 5)`

#### Logical and boolean operators to use with filter()

`==`   `<`   `<=`   `is.na()`   `%in%`   `|`   `xor()`

`!=`   `>`   `>=`   `is.na()`   `!`   `&`

See `base:::Logic` and `?Comparison` for help.

#### ARRANGE CASES



`arrange(data, ..., .by_group = FALSE)` Order rows by values of a column or columns (low to high), use with `desc()` to order from high to low.  
`mtcars %> arrange(mpg)`  
`mtcars %> arrange(desc(mpg))`

#### ADD CASES



`add_row(data, ..., .before = NULL, .after = NULL)` Add one or more rows to a table.  
`cars %> add_row(speed = 1, dist = 1)`

### Manipulate Variables

#### EXTRACT VARIABLES

Column functions return a set of columns as a new vector or table.



`pull(.data, var = -1, name = NULL, ...)` Extract column values as a vector, by name or index.  
`mtcars %> pull(wt)`



`select(.data, ...)` Extract columns as a table.  
`mtcars %> select(mpg, wt)`



`relocate(.data, ..., .before = NULL, .after = NULL)` Move columns to new position.  
`mtcars %> relocate(mpg, cyl, after = last_col())`

#### Use these helpers with `select()` and `across()`

e.g. `mtcars %> select(mpg:cyl)`

`contains(match)`   `num_range(prefix, range)` ; e.g., `mpg:cyl`  
`ends_with(match)`   `all_of(x)`/`any_of(x, ..., vars)` ; e.g., `!gear`  
`starts_with(match)`   `matches(match)`   `everything()`

#### MANIPULATE MULTIPLE VARIABLES AT ONCE

`df <- tibble(x_1 = c(1, 2), x_2 = c(3, 4), y = c(4, 5))`



`across(.cols, .funs, ..., .names = NULL)` Summarize or mutate multiple columns in the same way.  
`df %> summarize(across(everything(), mean))`



`c_across(.cols)` Compute across columns in row-wise data.  
`df %> rowwise() %> mutate(x_total = sum(c_across(1:2)))`

#### MAKE NEW VARIABLES

Apply **vectorized functions** to columns. Vectorized functions take vectors as input and return vectors of the same length as output (see back).



`vectorized function`  
`mutate(.data, ..., .keep = "all", .before = NULL, .after = NULL)` Compute new column(s). Also `add_column()`.  
`mtcars %> mutate(gpm = 1 / mpg)`  
`mtcars %> mutate(gpm = 1 / mpg, keep = "none")`



`rename(.data, ...)` Rename columns. Use `rename_with()` to rename with a function.  
`mtcars %> rename(miles_per_gallon = mpg)`

CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at [dplyr.tidyverse.org](https://dplyr.tidyverse.org) • dplyr 1.1.2 • Updated: 2023-05

Fonte: [Data transformation with dplyr - Cheat Sheet](#)

# Agora é a sua vez...

Digamos que você queira realizar as seguintes operações com os dados `flights`:

- Passo 1: Filtrar pelos voos com origem no aeroporto JFK;
- Passo 2: Contar o total de voos e voos atrasados para cada companhia aérea;
- Passo 3: Calcular a porcentagem de voos atrasados (escala de 0 a 100) e chamá-la de `percDelay`;
- Passo 4: Ordenar o resultado por `percDelay` em ordem decrescente.



# Referências

Algumas referências utilizadas para a construção desse material:

- [R for Data Science - Chapter 5](#)
- [Introdução à Ciência de Dados - Capítulo 4](#)
- [Curso R - Capítulos 6 e 7](#)
- [Data Wrangling with dplyr and tidyr - Cheat Sheet](#)
- [Data transformation with dplyr - Cheat Sheet](#)



Slides produzidos pela profa. Tatiana Benaglia.