Transformação de dados

Introdução à ciência de dados

Daniel Brito dos Santos

4.1 Introdução

- Visualização é uma ferramenta importante para gerar insight, mas raramente os dados estão na **forma** que você precisa
- Frequentemente precisamos:
  - **Criar** novas variáveis
  - Fazer **sumários** para achar padrões importantes
  - **Renomear** variáveis
  - **Reordenar** observações para facilitar a manipulação

- Vamos aprender a fazer tudo isso usando o pacote **dplyr** e um novo dataset com os **voos** que sairam da cidade de **Nova York** em **2013**
- Nosso objetivo é apresentar as principais ferramentas para **transformar** um data frame.
- Vamos começar com operações em **linhas**, depois em **colunas** e finalmente ver **grupos**

4.1.1 Prerequisitos

- O **dplyr** é outro pacote fundamental do tidyverse
- Nosso dataset vai ser o **nycflights13**

```
library(nycflights13)
library(tidyverse)
#>   Attaching packages                     tidyverse 1.3.2
#>   ggplot2 3.4.0            purrr   1.0.1.9000
#>   tibble  3.1.8            dplyr   1.0.99.9000
#>   tidyr   1.2.1.9001       stringr 1.5.0
```

```
#>   readr   2.1.3            forcats 0.5.2
#>   Conflicts                       tidyverse_conflicts()
#>   dplyr::filter() masks stats::filter()
#>   dplyr::lag()    masks stats::lag()
```

4.1.2 nycflights13

- Vamos usar o dataset *nycflights13::flights*
- Ele contém 336,776 voos que sairam da cidade de Nova York em 2013
- Primeiramente vamos dar uma espiadinha

Quais são as formas de visualizar um data set em R?

. . .

```
flights
```

. . .

```
glimpse(flights)
```

. . .

```
view(flights)
```

- É interessante observamos as características gerais dos nossos dados como
  - Nomes de colunas
  - Formato de dados

```
#| echo: true
glimpse(flights)
```

4.1.3 dplyr basics

- Vamos ver os **verbos básicos** que vão nos ajudar a resolver a maior parte dos desafios de **manipulação de dados**
- Todos tem três coisas em comum:

  1. O **primeiro** argumento é sempre um **data frame** (df)
  2. O **segundo** argumento descreve **o que fazer** com o df
  3. O **resultado** é sempre um **novo df**

- Como o primeiro argumento e o resultado são dfs, é muito conveniente usarmos os pipes! (|>)

- O pipe pega o que está na esquerda e passa como primeiro argumento da função depois dele:

```
x |> f(y)
f(x, y)
```

. . .

Como seria nesse caso?

```
x |> f(y) |> g(z)
```

. . .

```
g(f(x, y), z)f(x, y)
```

- Quem quer explicar esse código? Só um geralzão!

```
flights |>
  filter(dest == "IAH") |>
  group_by(year, month, day) |>
  summarize(
    arr_delay = mean(arr_delay, na.rm = TRUE)
  )
```

. . .

- O código inicia com o dataset *fligths*, depois ele é **filtrado**, **agrupado** e **sumarizado**.

. . .

- E os detalhes?

. . .

- Pegamos os voos, **filtramos** os destinados a "IAH". **Agrupamos** por ano, mes e dia. Finalmente, **sumarizamos** o seu atraso médio.

- Os verbos do dplyr são organizados de acordo com o em que eles operam:
  - linhas, colunas, grupos ou tabelas
- Vamos ver os três primeiros casos, começando com

4.2 Rows

- Os verbos mais importantes que operam em linhas são:
  - *filter()* - que filtra as linhas desejadas
  - *arrange()* - que muda a ordem das linhas
  - NENHUM DOS DOIS MODIFICAM AS COLUNAS
- Também temos:
  - *distinct()* que encontra linhas com valores únicos, mas pode modificar colunas

filter()

- Essa função mantém as linhas baseado no valor de suas colunas
- O primeiro argumento é um data frame
- Do segundo argumento em diante são **condições** que devem ser **verdadeiras** para **manter a linha**

Por exemplo

- Queremos encontrar todos os voos que chegaram mais de 120 minutos (duas horas) atrasados
- Quem arrisca?

. . .

```
#| echo: true
flights |>
  filter(arr_delay > 120)
```

- E se quisermos ver os voos que sairam dia primeiro de janeiro?

. . .

```
#| echo: true
flights |>
  filter(month == 1 & day == 1)
```

- Voos que sairam em janeiro ou fevereiro?

. . .

```
#| echo: true
flights |>
  filter(month == 1 | month == 2)
```

- Uma outra forma de fazer “em janeiro ou em fevereiro”:

```
#| echo: true
flights |>
  filter(month %in% c(1, 2))
```

Salvando o resultado

- Quando executamos `filter()` o dplyr cria e retorna um novo data frame
- Ele **não** modifica o input
- Por que funções *dplyr* **nunca** modificam os inputs
- Para salvar o resultado precisamos usar o operador de atribuição `<-`:

. . .

```
jan1 <- flights |>
  filter(month == 1 & day == 1)
```

4.2.2 Erros comuns

- No início é fácil confundir `=` com `==` quando queremos testar igualdade.

. . .

- O que acham que vai acontecer?

```
flights |>
  filter(month = 1)
```

. . .

```
#> Error in `filter()`:
#> ! We detected a named input.
#>   This usually means that you've used `=` instead of `==`.
#>   Did you mean `month == 1`?
```

- Outro erro comum é usar "ou" como usamos no dia a dia

```
flights |>
  filter(month == 1 | 2)
```

. . .

- O que acham que vai acontecer?

. . .

- Por que será que aconteceu isso?

`arrange()`

- altera a **ordem** das linhas baseado no **valor** de suas **colunas**
- recebe um **data frame** e um **conjunto** de **nomes de colunas** (ou expressões) como **critérios** de ordenação
- Cada **novo argumento** é usado para resolver **empates** na ordenação do anterior

---

Exemplo

- O que esse código faz?

```
flights |>
  arrange(year, month, day, dep_time)
```

. . .

- **Ordena** por ordem de decolagem, que está separada nas quatro variáveis: ano, mês, dia e hora da decolagem

```
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_…[1] dep_d…[2] arr_t…[3] sched…  arr_d…  carrier
#>   <int> <int> <int>    <int>     <int>    <dbl>    <int>   <int>   <dbl> <chr>
#> 1  2013     1     1      517       515        2      830     819      11 UA
#> 2  2013     1     1      533       529        4      850     830      20 UA
#> 3  2013     1     1      542       540        2      923     850      33 AA
#> 4  2013     1     1      544       545       -1     1004    1022     -18 B6
#> 5  2013     1     1      554       600       -6      812     837     -25 DL
#> 6  2013     1     1      554       558       -4      740     728      12 UA
#> # … with 336,770 more rows, 9 more variables: flight <int>, tailnum <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
#> #   [1]sched_dep_time, [2]dep_delay, [3]arr_time, sched_arr_time, arr_delay
```

- O que acham que esse código faz?

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

. . .

- Podemos usar a função `desc()` para reordenar a coluna em ordem decrescente

---

Podemos combinar `arrange()` e `filter()` para resolver problemas mais complexos

. . .

- Exemplo:
  - como poderíamos encontrar os voos que **chegaram mais atrasados**, mas que **decolaram** mais ou menos na **hora certa**? (esse é mais difícil)

. . .

- Vamos deixar mais exato o "hora certa":
  - voos que sairam com **até** 10 minutos de **atraso** ou **adiantamento** . . .
- (sim, eu também não sabia que voos poderiam sair adiantados)

```
flights |>
  filter(dep_delay <= 10 & dep_delay >= -10) |>
  arrange(desc(arr_delay))
#> # A tibble: 239,109 × 19
#>    year month   day dep_time sched_…¹ dep_d…² arr_t…³ sched… arr_d… carrier
#>   <int> <int> <int>    <int>    <int>   <dbl>   <int>   <int>   <dbl> <chr>
#> 1  2013    11     1      658      700      -2    1329    1015     194 VX
#> 2  2013     4    18      558      600      -2    1149     850     179 AA
#> 3  2013     7     7     1659     1700      -1    2050    1823     147 US
#> 4  2013     7    22     1606     1615      -9    2056    1831     145 DL
#> 5  2013     9    19      648      641       7    1035     810     145 UA
#> 6  2013     4    18      655      700      -5    1213     950     143 AA
#> # … with 239,103 more rows, 9 more variables: flight <int>, tailnum <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
```

```
#> #   [1]sched_dep_time, [2]dep_delay, [3]arr_time, sched_arr_time, arr_delay
```

4.2.4 `distinct()`

- **Encontra** todas as linhas **únicas** em um dataset
- Como os outros, seu primeiro argumento é o dataset
- Seus próximos argumentos são os **nomes das colunas** que queremos as linhas distintas

---

- O que esse código faz?

```
flights |>
  distinct(origin, dest)
```

. . .

```
#> # A tibble: 224 × 2
#>   origin dest
#>   <chr>  <chr>
#> 1 EWR    IAH
#> 2 LGA    IAH
#> 3 JFK    MIA
#> 4 JFK    BQN
#> 5 LGA    ATL
#> 6 EWR    ORD
#> # … with 218 more rows
```

- Encontra todos os pares únicos de `origin` e `dest`

- E esse código?

```
flights |>
  distinct()
```

. . .

- Se não colocarmos nomes, ele **remove** as linhas **duplicadas**
- Pergunta: ele altera o dataset que recebeu de input?
- **não!** Nenhum verbo dplyr altera o dataset original!

4.3 Columns

- Existem quatro verbos importantes que alteram as colunas sem alterar as linhas
  - `mutate()` - Cria novas colunas partindo de funções das colunas existentes
  - `select()` - Seleciona quais colunas manter no novo dataset
  - `rename()` - Renomea colunas
  - `relocate()` - Altera a posição das colunas
- Também vamos discutir o `pull()` que permite obter uma coluna do dataset

`mutate()`

- **Adiciona** novas colunas a partir da **manipulação** das colunas existentes
- Nos capítulos de transformação tem uma série de funções úteis para manipular diferentes tipos de variáveis
- Aqui vamos focar na álgebra básica

Computar o `gain` e `speed`

- **gain**: quanto tempo um voo atrasado compensou no ar
- **speed**: velocidade em milhas por hora

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60
  )
```

. . .

- Viram as novas colunas?

- Por padrão, o `mutate()` cria **novas colunas** na **extremidade direita** do dataset, fica difícil de ver no resumo

. . .

- podemos usar o argumento `.before`:

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60,
    .before = 1
  )
```

- o `.` é um sinal de que `.before` é um argumento e não uma nova variável

. . .

- Também podemos usar `.after` e `.before` com variáveis:

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60,
    .after = day
  )
```

- O que faz esse código?

. . .

- Cria as variáveis apos a variável `day`

Argumento `.keep`

- permite definirmos as **variáveis** que serão **mantidas**
- É muito útil usar com o `"used"` para **manter** as variáveis **utilizadas** e **resultantes** do "mutação"

. . .

```
flights |>
  mutate(,
    gain = dep_delay - arr_delay,
    hours = air_time / 60,
```

```
    gain_per_hour = gain / hours,
    .keep = "used"
  )
```

4.3.2 `select()`

- **Seleciona** um subgrupo de variáveis do dataset a partir dos seus nomes
- Muito útil para datasets com centenas ou até milhares de colunas
- Vamos aos exemplos do seu funcionamento

Selecionar colunas pelos nomes

- Quem chuta a sintaxe para selecionarmos: `year`, `month` e `day`?

. . .

```
flights |>
  select(year, month, day)
#> # A tibble: 336,776 × 3
#>    year month   day
#>   <int> <int> <int>
#> 1  2013     1     1
#> 2  2013     1     1
#> 3  2013     1     1
#> 4  2013     1     1
#> 5  2013     1     1
#> 6  2013     1     1
#> # … with 336,770 more rows
```

Selecionar todas as colunas entre `year` e `day`

- e aí?

. . .

```
flights |>
  select(year:day)
#> # A tibble: 336,776 × 3
#>    year month   day
#>   <int> <int> <int>
#> 1  2013     1     1
#> 2  2013     1     1
#> 3  2013     1     1
#> 4  2013     1     1
#> 5  2013     1     1
#> 6  2013     1     1
#> # … with 336,770 more rows
```

- E o contrário? Como selecionar todas EXCETO entre `year` e `day`?

. . .

```
flights |>
  select(!year:day)
#> # A tibble: 336,776 × 16
#>   dep_time sched_dep…¹ dep_d…² arr_t…³ sched… arr_d… carrier flight tailnum
#>      <int>       <int>   <dbl>   <int>   <int>   <dbl> <chr>    <int> <chr>
#> 1      517         515       2     830     819      11 UA        1545 N14228
#> 2      533         529       4     850     830      20 UA        1714 N24211
#> 3      542         540       2     923     850      33 AA        1141 N619AA
#> 4      544         545      -1    1004    1022     -18 B6         725 N804JB
#> 5      554         600      -6     812     837     -25 DL         461 N668DN
#> 6      554         558      -4     740     728      12 UA        1696 N39463
#> # … with 336,770 more rows, 7 more variables: origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#> #   time_hour <dttm>, and abbreviated variable names ¹sched_dep_time,
#> #   ²dep_delay, ³arr_time, sched_arr_time, arr_delay
```

Selecionar todas as colunas que são caracteres

. . .

```
flights |>
  select(where(is.character))
#> # A tibble: 336,776 × 4
#>   carrier tailnum origin dest
#>   <chr>   <chr>   <chr>  <chr>
#> 1 UA      N14228  EWR    IAH
#> 2 UA      N24211  LGA    IAH
#> 3 AA      N619AA  JFK    MIA
#> 4 B6      N804JB  JFK    BQN
#> 5 DL      N668DN  LGA    ATL
#> 6 UA      N39463  EWR    ORD
#> # … with 336,770 more rows
```

. . .

- `where()` é uma "helper function" do `select()`

---

outras helper functions do `select()`

- `starts_with("abc")`: corresponde aos nomes que começam com "abc"
- `ends_with("xyz")`: corresponde aos nomes que terminam com "xyz"
- `contains("ijk")`: corresponde aos nomes que contém "ijk"
- `num_range("x", 1:3)`: corresponde com x1, x2 and x3.
- Para mais detalhes vejam consultem a documentação:
  - `?select`

Podemos renomear as variáveis selecionadas

- Alguém arrisca? Como mudar "tailnum" para "tail_num"?

. . .

```
flights |>
  select(tail_num = tailnum)
#> # A tibble: 336,776 × 1
#>   tail_num
#>   <chr>
```

```
#> 1 N14228
#> 2 N24211
#> 3 N619AA
#> 4 N804JB
#> 5 N668DN
#> 6 N39463
#> # … with 336,770 more rows
```

- novo_nome = nome_antigo

4.3.3 rename()

- se quiser manter as variáveis e apenas **renomar** algumas use `rename()` ao invés de `select()`
- Quem vai?

. . .

```
flights |>
  rename(tail_num = tailnum)
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_…¹ dep_d…² arr_t…³ sched… arr_d… carrier
#>   <int> <int> <int>    <int>    <int>   <dbl>   <int>   <int>   <dbl> <chr>
#> 1  2013     1     1      517      515       2     830     819      11 UA
[...]
```

- Funciona exatamente como `select()`, mas mantém **todas as variáveis**, não apenas as explicitamente selecionadas

4.3.4 `relocate()`

- **Move** variáveis
- Podemos querer deixar variáveis relacionadas próximas, ou variáveis importantes mais pra esquerda
- Por padrão ele **trás pra esquerda** as variáveis citadas

. . .

```
flights |>
  relocate(time_hour, air_time)
```

```
#> # A tibble: 336,776 × 19
#>   time_hour           air_time  year month   day dep_time sched_dep…¹ dep_d…²
#>   <dttm>                 <dbl> <int> <int> <int>    <int>       <int>   <dbl>
#> 1 2013-01-01 05:00:00      227  2013     1     1      517         515       2
#> 2 2013-01-01 05:00:00      227  2013     1     1      533         529       4
#> 3 2013-01-01 05:00:00      160  2013     1     1      542         540       2
#> 4 2013-01-01 05:00:00      183  2013     1     1      544         545      -1
#> 5 2013-01-01 06:00:00      116  2013     1     1      554         600      -6
#> 6 2013-01-01 05:00:00      150  2013     1     1      554         558      -4
#> # … with 336,770 more rows, 11 more variables: arr_time <int>,
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, and abbreviated variable names ¹sched_dep_time, ²dep_delay
```

Também podemos usar os mesmos `.before` e `.after` que usamos em `mutate()`

- O que acham que esse código faz?

```
flights |>
  relocate(year:dep_time, .after = time_hour)
```

```
flights |>
  relocate(year:dep_time, .after = time_hour)
#> # A tibble: 336,776 × 19
#>   sched…¹ dep_d…² arr_t…³ sched… arr_d… carrier flight tailnum origin dest
#>     <int>   <dbl>   <int>   <int>   <dbl> <chr>    <int> <chr>   <chr>  <chr>
#> 1     515       2     830     819      11 UA        1545 N14228  EWR    IAH
#> 2     529       4     850     830      20 UA        1714 N24211  LGA    IAH
#> 3     540       2     923     850      33 AA        1141 N619AA  JFK    MIA
#> 4     545      -1    1004    1022     -18 B6         725 N804JB  JFK    BQN
#> 5     600      -6     812     837     -25 DL         461 N668DN  LGA    ATL
#> 6     558      -4     740     728      12 UA        1696 N39463  EWR    ORD
#> # … with 336,770 more rows, 9 more variables: air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>, year <int>,
#> #   month <int>, day <int>, dep_time <int>, and abbreviated variable names
```

```
#> #   [1]sched_dep_time, [2]dep_delay, [3]arr_time, sched_arr_time, arr_delay
```

- **realoca** todas as colunas de `year` até `dep_time` para depois de `time_hour`

- Como poderíamos colocar as variáveis sobre a chegada do voo antes de `dep_time`?

. . .

- Estamos interessados nas variáveis que inicial com "arr", então:

. . .

```
flights |>
  relocate(starts_with("arr"), .before = dep_time)
#> # A tibble: 336,776 × 19
#>    year month   day arr_time arr_de…[1] dep_t…[2] sched…[3] dep_d… sched… carrier
#>   <int> <int> <int>    <int>    <dbl>   <int>   <int>   <dbl>   <int> <chr>
#> 1  2013     1     1      830       11     517     515       2     819 UA
#> 2  2013     1     1      850       20     533     529       4     830 UA
#> 3  2013     1     1      923       33     542     540       2     850 AA
#> 4  2013     1     1     1004      -18     544     545      -1    1022 B6
#> 5  2013     1     1      812      -25     554     600      -6     837 DL
#> 6  2013     1     1      740       12     554     558      -4     728 UA
#> # … with 336,770 more rows, 9 more variables: flight <int>, tailnum <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
#> #   [1]arr_delay, [2]dep_time, [3]sched_dep_time, dep_delay, sched_arr_time
```

4.4 Groups

4.4 Groups

- Até aqui aprendemos sobre funções que trabalham com **linhas** e **colunas**.
- O verdadeiro poder de dplyr é te permitir trabalhar com **grupos**
- Vamos ver as duas funções mais importantes para isso:
  - `group_by()` e `summarize()`

4.4.1 `group_by()`

- **Divide** o datasets em grupos **significativos** para a análise
- Vamos explicar com um exemplo

. . .

```
flights |>
  group_by(month)
#> # A tibble: 336,776 × 19
#> # Groups:   month [12]
#>    year month   day dep_time sched_…[1] dep_d…[2] arr_t…[3] sched… arr_d… carrier
#>   <int> <int> <int>    <int>      <int>    <dbl>    <int>   <int>   <dbl> <chr>
#> 1  2013     1     1      517        515        2      830     819      11 UA
#> 2  2013     1     1      533        529        4      850     830      20 UA
#> 3  2013     1     1      542        540        2      923     850      33 AA
#> 4  2013     1     1      544        545       -1     1004    1022     -18 B6
#> 5  2013     1     1      554        600       -6      812     837     -25 DL
#> 6  2013     1     1      554        558       -4      740     728      12 UA
#> # … with 336,770 more rows, 9 more variables: flight <int>, tailnum <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
#> #   [1]sched_dep_time, [2]dep_delay, [3]arr_time, sched_arr_time, arr_delay
```

- `group_by()` não altera os dados, mas note que aparece "Groups: month"
- Isso significa que as **próximas operações** vão funcionar "**por mês**"
- `group_by()` não faz nada sozinho:
  - ele **altera** o comportamento dos **verbos subsequentes**

4.4.2 `summarize()`

- é a operação agrupada mais importante
- **reduz** cada grupo para uma **única linha**
- Vamos ao exemplo!

- O que acham que esse código faz?

```r
flights |>
  group_by(month) |>
  summarize(
    delay = mean(dep_delay)
  )
```

. . .

- Ele vai **agrupar** nossos dados por mês, em seguida **resumir** cada mês pela **média** da variável `dep_delay` naquele mês
- Testem!

. . .

- Ué

O que deu errado?

```r
flights |>
  group_by(month) |>
  summarize(
    delay = mean(dep_delay)
  )
#> # A tibble: 12 × 2
#>   month delay
#>   <int> <dbl>
#> 1     1    NA
#> 2     2    NA
#> 3     3    NA
#> 4     4    NA
#> 5     5    NA
#> 6     6    NA
#> # … with 6 more rows
```

- NA é contagioso, quase todas as operações com NA resultam em NA. Se `dep_delay` tiver pelo menos um NA, sua média será NA
- Vamos corrigir removendo todas as NAs: `NA.rm = TRUE`

- Onde acham que devemos colocar o `NA.rm = TRUE`?

```
flights |>
  group_by(month) |>
  summarize(
    delay = mean(dep_delay)
  )
```

...

```
flights |>
  group_by(month) |>
  summarize(
    delay = mean(dep_delay, na.rm = TRUE)
  )
#> # A tibble: 12 × 2
#>   month delay
#>   <int> <dbl>
#> 1     1  10.0
#> 2     2  10.8
#> 3     3  13.2
#> 4     4  13.9
#> 5     5  13.0
#> 6     6  20.8
#> # … with 6 more rows
```

`n()`

- retorna o **número** de **linhas** em **cada** grupo
- podemos gerar a contagem e média no mesmo `summarize()`

```
flights |>
  group_by(month) |>
  summarize(
    delay = mean(dep_delay, na.rm = TRUE),
    n = n()
  )
#> # A tibble: 12 × 3
#>   month delay     n
```

```
#>   <int> <dbl> <int>
#> 1     1  10.0 27004
#> 2     2  10.8 24951
#> 3     3  13.2 28834
#> 4     4  13.9 28330
#> 5     5  13.0 28796
#> 6     6  20.8 28243
#> # … with 6 more rows
```

- Existem outros tipos de sumários, mas nesta parte do livro focamos nesses dois. Segundo o Whikham:

  Means and counts can get you a surprisingly long way in data science!

4.4.3 As funções `slice_`

- cinco funções úteis para especificar **quais linhas** pegar de **cada grupo**
  - `df |> slice_head(n = 1)` pega a **primeira** linha de cada grupo
  - `df |> slice_tail(n = 1)` pega a **última** linha de cada grupo

- `df |> slice_min(x, n = 1)` pega a linha com o **menor** valor de `x`
- `df |> slice_max(x, n = 1)` pega a linha com o **maior** valor de `x`
- `df |> slice_sample(n = 1)` pega uma linha **aleatória** de cada grupo
- Podemos variar o `n =` para pegar **mais de uma** linha
  - Podemos substituir pelo `prop = 0.1` para pegar 10% das linhas de cada grupo

---

Exemplo

- Como podemos encontrar os voos **mais** atrasados para cada destino?

. . .

```r
flights |>
  group_by(dest) |>
  slice_max(arr_delay, n = 1)
#> # A tibble: 108 × 19
#> # Groups:   dest [105]
#>    year month   day dep_time sched_…¹ dep_d…² arr_t…³ sched… arr_d… carrier
#>   <int> <int> <int>    <int>    <int>   <dbl>   <int>   <int>   <dbl> <chr>
#> 1  2013     7    22     2145     2007      98     132    2259     153 B6
#> 2  2013     7    23     1139      800     219    1250     909     221 B6
#> 3  2013     1    25      123     2000     323     229    2101     328 EV
#> 4  2013     8    17     1740     1625      75    2042    2003      39 UA
#> 5  2013     7    22     2257      759     898     121    1026     895 DL
#> 6  2013     7    10     2056     1505     351    2347    1758     349 UA
#> # … with 102 more rows, 9 more variables: flight <int>, tailnum <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
#> #   ¹sched_dep_time, ²dep_delay, ³arr_time, sched_arr_time, arr_delay
```

4.4.4 Agrupando por múltiplas variáveis

- Podemos criar grupos usando mais de uma variável
- Ex: criar um grupo pra cada dia. Quem chuta?

. . .

```r
daily <- flights |>
  group_by(year, month, day)
daily
#> # A tibble: 336,776 × 19
#> # Groups:   year, month, day [365]
#>    year month   day dep_time sched_…¹ dep_d…² arr_t…³ sched… arr_d… carrier
#>   <int> <int> <int>    <int>    <int>   <dbl>   <int>   <int>   <dbl> <chr>
#> 1  2013     1     1      517      515       2     830     819      11 UA
#> 2  2013     1     1      533      529       4     850     830      20 UA
#> 3  2013     1     1      542      540       2     923     850      33 AA
#> 4  2013     1     1      544      545      -1    1004    1022     -18 B6
#> 5  2013     1     1      554      600      -6     812     837     -25 DL
#> 6  2013     1     1      554      558      -4     740     728      12 UA
#> # … with 336,770 more rows, 9 more variables: flight <int>, tailnum <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
```

```
#> #   [1]sched_dep_time, [2]dep_delay, [3]arr_time, sched_arr_time, arr_delay
```

Um detalhe

- Quando sumarizamos um tibble agrupado por mais de uma variável, o sumário ignora o último agrupamento
- A própria função avisa desse comportamento contraintuitivo e diz como mudar

```
daily_flights <- daily |>
  summarize(
    n = n()
  )
#> `summarise()` has grouped output by 'year', 'month'. You can override using
#> the `.groups` argument.
```

- Testem o `.groups =` com seus principais argumentos (`drop_last`, `keep`. `drop`)
- O que cada um faz?

```
daily_flights <- daily |>
  summarize(
    n = n(),
    .groups = "drop_last"
  )
```

- Com o `.groups` podemos usar:
  - `drop_last` para ignorar o último agrupamento, como o padrão
  - `keep` para manter todos os grupos
  - `drop` para ignorar todos os grupos

4.4.5 Desagrupar

- `ungroup()` remove os grupos
- Qual vocês acham que é o resultado desse código?

```
daily |>
  ungroup() |>
  summarize(
    delay = mean(dep_delay, na.rm = TRUE),
    flights = n()
  )
```

. . .

- Rodem. O que aconteceu? É o que esperavam?

. . .

```
#> # A tibble: 1 × 2
#>   delay flights
#>   <dbl>   <int>
#> 1  12.6  336776
```

- Quando sumarizamos um data frame desagrupado o dplyr trata como se fosse um grupo só

4.6 Sumário

- Neste capítulo aprendemos ferramentas do dplyr para trabalhar com data frames. Vimos ferramentas para manipular:
- linhas (filter() e arrange())
- colunas (select() e mutate())
- e grupos (group_by() e summarize()).
- Nosso enfoque nessas ferramentas gerais é um excelente ponto de partida para seguir aprendendo!
- Quem tiver interesse pode ler o capítulo de transformações ou já sair no soco com um dataframe!

Dúvidas?