

R para Ciência de Dados 2

dplyr ++



Agosto de 2020

0 pacote dplyr

Verbos principais

Já vimos que com os principais verbos do `{dplyr}` já conseguimos fazer diversas operações de manipulação de bases de dados.

- Selecionar colunas: `select()`
- Ordenar linhas: `arrange()`
- Filtrar linhas: `filter()`
- Criar ou modificar colunas: `mutate()`
- Agrupar e sumarizar: `group_by()` + `summarise()`



Mas o dplyr tem muito mais para oferecer. MUITO MAIS!

Miscelânea de funções úteis

Para aquecer, vamos listar uma miscelânea de funções muito úteis, mas menos conhecidas do `{dplyr}`.

- `bind_rows()`: para empilhar duas bases.
- `case_when()`: generalização da `ifelse()` para várias condições.
- `first()`, `last()`: para pegar o primeiro ou último valor de um vetor/coluna.
- `na_if()`: para transformar um determinado valor de um vetor/coluna em `NA`.
- `coalesce()`: para substituir os `NA`s de uma coluna pelos valores equivalentes de uma segunda coluna.
- `lag()`, `lead()`: para gerar colunas defasadas.
- `pull()`: para transformar uma coluna da base em um vetor.
- `slice_sample`: para gerar amostras da base.

bind_rows()

Vamos usar a função `bind_rows()` para juntarmos duas bases com as mesmas colunas.

```
# Juntando duas bases
imdb_2015 <- readr::read_rds("../data/imdb_por_ano/imdb_2015.rds")
imdb_2016 <- readr::read_rds("../data/imdb_por_ano/imdb_2016.rds")

bind_rows(imdb_2015, imdb_2016) %>% head()
```

```
## # A tibble: 6 x 15
##   titulo    ano diretor duracao cor   generos pais classificacao orcamento
##   <chr>    <int> <chr>      <int> <chr> <chr>   <chr> <chr>          <int>
## 1 Aveng...  2015 Joss W...    141 Color Action... USA   A partir de ... 2500000000
## 2 Juras...  2015 Colin ...    124 Color Action... USA   A partir de ... 1500000000
## 3 Furio...  2015 James ...    140 Color Action... USA   A partir de ... 1900000000
## 4 The G...  2015 Peter ...     93 Color Advent... USA   Livre              NA
## 5 Jupit...  2015 Lana W...    127 Color Action... USA   A partir de ... 1760000000
## 6 Insid...  2015 Pete D...     95 Color Advent... USA   Livre             1750000000
## # ... with 6 more variables: receita <int>, nota_imdb <dbl>,
## #   likes_facebook <int>, ator_1 <chr>, ator_2 <chr>, ator_3 <chr>
```

Neste caso, a função `bind_rows()` é equivalente à função `rbind()`.

```
# Juntando duas bases
imdb_2015 <- readr::read_rds("../data/imdb_por_ano/imdb_2015.rds")
imdb_2016 <- readr::read_rds("../data/imdb_por_ano/imdb_2016.rds")

rbind(imdb_2015, imdb_2016) %>% head()

## # A tibble: 6 x 15
##   titulo    ano diretor duracao cor   generos pais classificacao orcamento
##   <chr>    <int> <chr>      <int> <chr> <chr>   <chr> <chr>          <int>
## 1 Aveng...  2015 Joss W...   141 Color Action... USA   A partir de ... 2500000000
## 2 Juras...  2015 Colin ...   124 Color Action... USA   A partir de ... 1500000000
## 3 Furio...  2015 James ...   140 Color Action... USA   A partir de ... 1900000000
## 4 The G...  2015 Peter ...    93 Color Advent... USA   Livre              NA
## 5 Jupit...  2015 Lana W...   127 Color Action... USA   A partir de ... 1760000000
## 6 Insid...  2015 Pete D...    95 Color Advent... USA   Livre             1750000000
## # ... with 6 more variables: receita <int>, nota_imdb <dbl>,
## #   likes_facebook <int>, ator_1 <chr>, ator_2 <chr>, ator_3 <chr>
```

Observação: tanto para a `bind_rows()` quanto para a `rbind()`, a ordem das colunas em ambas as bases pode ser diferente. As colunas são empilhadas pelo nome.

Podemos também usar a função `bind_rows()` para juntar várias tabelas. Aqui, todas as tabelas continuam tendo as mesmas colunas.

```
arquivos <- list.files("../data/imdb_por_ano/", full.names = TRUE)

arquivos %>%
  purrr::map(readr::read_rds) %>%
  bind_rows() %>%
  head()
```

```
## # A tibble: 6 x 15
##   titulo    ano diretor duracao cor   generos pais classificacao orcamento
##   <chr>    <int> <chr>      <int> <chr> <chr>    <chr> <chr>          <int>
## 1 Intol...  1916 D.W. G...   123 Blac... Drama|... USA    Outros        385907
## 2 Over ...  1920 Harry ...   110 Blac... Crime|... USA    Outros        100000
## 3 The B...  1925 King V...   151 Blac... Drama|... USA    Outros        245000
## 4 The B...  1929 Harry ...   100 Blac... Musica... USA    Outros        379000
## 5 Hell'...  1930 Howard...    96 Blac... Drama|... USA    Outros       3950000
## 6 A Far...  1932 Frank ...    79 Blac... Drama|... USA    Outros        800000
## # ... with 6 more variables: receita <int>, nota_imdb <dbl>,
## #   likes_facebook <int>, ator_1 <chr>, ator_2 <chr>, ator_3 <chr>
```

Observação: a função `purrr::map()` está aplicando a função `readr::read_rds()` a todos os elementos do vetor `arquivos` e devolvendo uma lista de tibbles, uma para cada ano da base IMDB. Aprenderemos mais sobre essa função na aula de `purrr`.

Como a função `map()` devolve uma lista, a função `rbind()` não funcionaria.

```
arquivos <- list.files("../data/imdb_por_ano/", full.names = TRUE)

arquivos %>%
  purrr::map(readr::read_rds) %>%
  rbind()
```

```
##      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]      [,8]      [,9]
## . List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15
##     [,10]     [,11]     [,12]     [,13]     [,14]     [,15]     [,16]     [,17]     [,18]
## . List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15
##     [,19]     [,20]     [,21]     [,22]     [,23]     [,24]     [,25]     [,26]     [,27]
## . List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15
##     [,28]     [,29]     [,30]     [,31]     [,32]     [,33]     [,34]     [,35]     [,36]
## . List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15
##     [,37]     [,38]     [,39]     [,40]     [,41]     [,42]     [,43]     [,44]     [,45]
## . List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15
##     [,46]     [,47]     [,48]     [,49]     [,50]     [,51]     [,52]     [,53]     [,54]
## . List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15
##     [,55]     [,56]     [,57]     [,58]     [,59]     [,60]     [,61]     [,62]     [,63]
## . List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15
##     [,64]     [,65]     [,66]     [,67]     [,68]     [,69]     [,70]     [,71]     [,72]
## . List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15
##     [,73]     [,74]     [,75]     [,76]     [,77]     [,78]     [,79]     [,80]     [,81]
## . List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15
##     [,82]     [,83]     [,84]     [,85]     [,86]     [,87]     [,88]     [,89]     [,90]
## . List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15 List,15
```


Além disso, a função `bind_rows()` pode ser utilizada para empilhar bases com colunas diferentes.

```
tab1 <- tibble::tibble(  
  var1 = c(1, 2, 3),  
  var2 = c("a", "b", "c"),  
  var3 = c(10, 20, 30)  
)  
  
tab2 <- tibble::tibble(  
  var2 = c("d", "e", "f"),  
  var1 = c(4, 5, 6)  
)  
  
bind_rows(tab1, tab2)
```

```
## # A tibble: 6 x 3  
##   var1 var2   var3  
##   <dbl> <chr> <dbl>  
## 1     1 a      10  
## 2     2 b      20  
## 3     3 c      30  
## 4     4 d      NA  
## 5     5 e      NA  
## 6     6 f      NA
```

Ao contrário da função `rbind()`.

```
tab1 <- tibble::tibble(  
  var1 = c(1, 2, 3),  
  var2 = c("a", "b", "c"),  
  var3 = c(10, 20, 30)  
)  
  
tab2 <- tibble::tibble(  
  var2 = c("d", "e", "f"),  
  var1 = c(4, 5, 6)  
)  
  
rbind(tab1, tab2)
```

```
## Error in rbind(deparse.level, ...): numbers of columns of arguments do not match
```

Extra: o `{dplyr}` também possui a função `bind_cols()`, para juntar duas bases colocando suas colunas lado-a-lado.

case_when()

A função `case_when()` generaliza a função `ifelse()`, permitindo colocar quantas condições quisermos.

```
x <- sample(-10:10, 10)

case_when(
  x < 0 ~ "negativo",
  x == 0 ~ "zero",
  x > 0 ~ "positivo"
)
```

```
## [1] "negativo" "negativo" "positivo" "positivo" "negativo" "negativo"
## [7] "positivo" "negativo" "positivo" "negativo"
```

Com `ifelse()`, precisaríamos usar a função duas vezes:

```
ifelse(x < 0, "negativo", ifelse(x == 0, "zero", "positivo"))
```

```
## [1] "negativo" "negativo" "positivo" "positivo" "negativo" "negativo"
## [7] "positivo" "negativo" "positivo" "negativo"
```

A ordem das condições é importante no `case_when()`, pois os testes são realizados na ordem em que eles aparecem.

```
mtcars %>%
  mutate(
    mpg_cat = case_when(
      mpg < 15 ~ "economico",
      mpg < 22 ~ "regular",
      mpg >= 22 ~ "bebe bem"
    )
  ) %>%
  head(8)
```

##	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	mpg_cat
## 1	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4	regular
## 2	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4	regular
## 3	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1	bebe bem
## 4	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1	regular
## 5	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2	regular
## 6	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1	regular
## 7	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4	economico
## 8	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2	bebe bem

Repare que os carros com `mpg < 15` entram na primeira condição, a segunda condição pega os carros com `15 <= mpg < 25`.

Como a última condição é complementar, isto é, as observações que não entraram nas condições anteriores com certeza entrarão na última condição, podemos substituí-la por um simples `TRUE`.

```
mtcars %>%
  mutate(
    mpg_cat = case_when(
      mpg < 15 ~ "economico",
      mpg < 22 ~ "regular",
      TRUE ~ "bebe bem"
    )
  ) %>%
  head(8)
```

##	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	mpg_cat
## 1	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4	regular
## 2	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4	regular
## 3	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1	bebe bem
## 4	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1	regular
## 5	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2	regular
## 6	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1	regular
## 7	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4	economico
## 8	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2	bebe bem

O `case_when()` é muito útil dentro do `mutate()`.

```
mtcars %>%  
  mutate(  
    mpg_cat = case_when(  
      mpg < 15 ~ "econômico",  
      mpg < 22 ~ "regular",  
      TRUE ~ "bebe bem"  
    )  
  ) %>%  
  head(14)
```

##	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb	mpg_cat
## 1	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4	regular
## 2	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4	regular
## 3	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1	bebe bem
## 4	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1	regular
## 5	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2	regular
## 6	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1	regular
## 7	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4	econômico
## 8	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2	bebe bem
## 9	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2	bebe bem
## 10	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4	regular
## 11	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4	regular
## 12	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3	regular
## 13	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3	regular
## 14	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3	regular

first(), last()

Essas funções retornam o primeiro e último valor de um vetor.

```
x <- c(1, 12, 30, 41, 15)
```

```
first(x)
```

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

```
last(x)
```

```
## [1] 15
```

São funções úteis quando temos algum tipo de ordem:

```
tab <- tibble::tibble(  
  tempo = c(1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4),  
  var = c(1, 4, 10, 33, 1, 3, 0, 21, 12, 7, 9, 17),  
  grupo = c(rep("a", 4), rep("b", 4), rep("c", 4))  
)  
  
tab %>%  
  group_by(grupo) %>%  
  arrange(tempo, .by_group = TRUE) %>%  
  mutate(inicio = first(var), fim = last(var))
```

```
## # A tibble: 12 x 5  
## # Groups:   grupo [3]  
##   tempo  var grupo inicio  fim  
##   <dbl> <dbl> <chr>   <dbl> <dbl>  
## 1     1     1  a         1     33  
## 2     2     4  a         1     33  
## 3     3    10  a         1     33  
## 4     4    33  a         1     33  
## 5     1     1  b         1     21  
## 6     2     3  b         1     21  
## 7     3     0  b         1     21  
## 8     4    21  b         1     21  
## 9     1    12  c        12     17  
## 10    2     7  c        12     17  
## 11    3     9  c        12     17  
## 12    4    17  c        12     17
```


na_if()

Transforma um valor especificado em NA.

```
tab <- tibble::tibble(  
  var = c(1, 10, 2, -99, 10, -99)  
)  
  
tab %>% mutate(var = na_if(var, -99))
```

```
## # A tibble: 6 x 1  
##       var  
##   <dbl>  
## 1     1  
## 2    10  
## 3     2  
## 4    NA  
## 5    10  
## 6    NA
```

coalesce()

A função `coalesce()` substitui os `NA`s de uma coluna pelos valores equivalentes de uma segunda coluna. No exemplo abaixo, substituímos os `NA`s da coluna `var1` pelos valores equivalentes da coluna `var2` (criamos uma nova coluna `var3` com o resultado para visualizarmos melhor). No caso em que as duas colunas apresentavam `NA`, a coluna `var3` continuou com o `NA`.

```
tab <- tibble::tibble(  
  var1 = c(1, 2, NA, 10, NA, NA),  
  var2 = c(NA, 2, 2, 3, 0, NA)  
)  
  
tab %>% mutate(var3 = coalesce(var1, var2))
```

```
## # A tibble: 6 x 3  
##   var1  var2  var3  
##   <dbl> <dbl> <dbl>  
## 1     1    NA     1  
## 2     2     2     2  
## 3    NA     2     2  
## 4    10     3    10  
## 5    NA     0     0  
## 6    NA    NA    NA
```

Você também pode usar para substituir `NA`s de uma variável por um único valor.

```
tab %>% mutate(var3 = coalesce(var1, 33))
```

```
## # A tibble: 6 x 3
##   var1  var2  var3
##   <dbl> <dbl> <dbl>
## 1     1     NA     1
## 2     2     2     2
## 3    NA     2    33
## 4    10     3    10
## 5    NA     0    33
## 6    NA    NA    33
```

Ou simplesmente usar a função `tidyr::replace_na()`.

```
tab %>% tidyr::replace_na(replace = list(var1 = 33, var2 = 66))
```

```
## # A tibble: 6 x 2
##   var1  var2
##   <dbl> <dbl>
## 1     1    66
## 2     2     2
## 3    33     2
## 4    10     3
## 5    33     0
## 6    33    66
```

lag(), lead()

Essas funções devolvem o valor defasado e valor futuro.

```
tab <- tibble::tibble(  
  tempo = c(1, 2, 3, 4, 5),  
  var = c(1, 4, 10, 33, 20)  
)  
  
tab %>%  
  dplyr::mutate(  
    var_lag = lag(var),  
    var_lead = lead(var)  
  )
```

```
## # A tibble: 5 x 4  
##   tempo    var var_lag var_lead  
##   <dbl> <dbl>   <dbl>   <dbl>  
## 1     1     1     NA       4  
## 2     2     4      1     10  
## 3     3    10      4     33  
## 4     4    33     10     20  
## 5     5    20     33     NA
```

pull()

Devolve uma coluna da base como vetor.

```
mtcars %>% pull(mpg)
```

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4  
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7  
## [31] 15.0 21.4
```

slice_sample()

Essa função pode ser utilizada para pegarmos uma amostra de linhas da nossa base.

No exemplo abaixo, pegamos uma amostra de tamanho 10 da base mtcars.

```
slice_sample(mtcars, n = 10)
```

##	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
## Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
## Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
## Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
## Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
## Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
## Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
## Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
## Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
## Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
## Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2

Agora, pegamos 50% da base mtcars.

```
slice_sample(mtcars, prop = 0.5)
```

##	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
## Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
## Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
## Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
## 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
## Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
## Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
## Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
## Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
## Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
## Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
## Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
## Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
## Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
## 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

dplyr 1.0

O novo dplyr

A versão 1.0 do pacote `dplyr` foi oficialmente lançada em junho de 2020 e contou com diversas novidades. Vamos falar das principais mudanças:

- A nova função `across()`, que facilita aplicar uma mesma operação em várias colunas.
- A repaginada função `rowwise()`, para fazer operações por linha.
- Novas funcionalidades das funções `select()` e `rename()` e a nova função `relocate()`.

Motivação

Base de dados de venda de casas na cidade de Ames, nos Estados Unidos.

- 2930 linhas e 77 colunas.
- Cada linha corresponde a uma casa vendida e cada coluna a uma característica da casa ou da venda.
- Versão traduzida: faça o download [clikando aqui](#).
- Base original:

```
install.packages("AmesHousing")  
data(ames_raw, package = "AmesHousing")
```

across()

A função `across()` substitui a família de verbos `verbo_all()`, `verbo_if` e `verbo_at()`. A ideia é facilitar a aplicação de uma operação a diversas colunas da base. Para sumarizar a base para mais de uma variável, antigamente fazíamos

```
ames %>%
  group_by(geral_qualidade) %>%
  summarise(
    lote_area_media = mean(lote_area, na.rm = TRUE),
    venda_valor_medio = mean(venda_valor, na.rm = TRUE)
  )
```

```
## # A tibble: 10 x 3
##   geral_qualidade lote_area_media venda_valor_medio
##           <int>           <dbl>           <dbl>
## 1             1      15214.         48725
## 2             2       9326.         52325.
## 3             3       9439.         83186.
## 4             4       8464.        106485.
## 5             5       9995.        134753.
## 6             6       9788.        162130.
## 7             7      10309.        205026.
## 8             8      10618.        270914.
## 9             9      12777.        368337.
## 10            10      18071.        450217.
```

Ou então

```
ames %>%  
  group_by(geral_qualidade) %>%  
  summarise_at(  
    .vars = vars(lote_area, venda_valor),  
    ~mean(.x, na.rm = TRUE)  
  )
```

```
## # A tibble: 10 x 3  
##   geral_qualidade lote_area venda_valor  
##           <int>      <dbl>      <dbl>  
## 1             1    15214.    48725  
## 2             2     9326.    52325.  
## 3             3     9439.    83186.  
## 4             4     8464.   106485.  
## 5             5     9995.   134753.  
## 6             6     9788.   162130.  
## 7             7    10309.   205026.  
## 8             8    10618.   270914.  
## 9             9    12777.   368337.  
## 10            10    18071.   450217.
```

Com a nova função `across()`, fazemos

```
ames %>%  
  group_by(geral_qualidade) %>%  
  summarise(across(  
    .cols = c(lote_area, venda_valor),  
    .fns = mean, na.rm = TRUE  
  ))
```

```
## # A tibble: 10 x 3  
##   geral_qualidade lote_area venda_valor  
##           <int>      <dbl>      <dbl>  
## 1             1    15214.    48725  
## 2             2     9326.    52325.  
## 3             3     9439.    83186.  
## 4             4     8464.   106485.  
## 5             5     9995.   134753.  
## 6             6     9788.   162130.  
## 7             7    10309.   205026.  
## 8             8    10618.   270914.  
## 9             9    12777.   368337.  
## 10            10    18071.   450217.
```

A sintaxe é parecida com a função `summarise_at()`, mas agora não precisamos mais usar a função `vars()` e nem usar `list(nome_da_funcao)` ou `~nome_da_funcao(.x)` para definir a função aplicada nas colunas.

Usando `across()`, podemos facilmente aplicar uma função em todas as colunas da nossa base. Abaixo, calculamos o número de valores distintos para todas as variáveis da base `ames`.

```
# Pegando apenas 5 colunas por questão de espaço
ames %>%
  summarise(across(.fns = n_distinct)) %>%
  select(1:5)
```

```
## # A tibble: 1 x 5
##   lote_fachada lote_area lote_formato lote_config terreno_contorno
##         <int>     <int>         <int>         <int>         <int>
## 1           129      1960             4             5             4
```

O padrão do parâmetro `.cols` é `everything()`, que representa "todas as colunas".

Anteriormente, utilizaríamos a função `summarise_all()`.

```
# Pegando apenas 5 colunas por questão de espaço
ames %>%
  summarise_all(.funs = ~n_distinct(.x)) %>%
  select(1:5)
```

```
## # A tibble: 1 x 5
##   lote_fachada lote_area lote_formato lote_config terreno_contorno
##   <int>      <int>      <int>      <int>      <int>
## 1         129      1960          4          5          4
```

Se quisermos selecionar as colunas a serem modificadas a partir de um teste lógico, utilizamos o ajudante `where()`.

No exemplo abaixo, calculamos o número de valores distintos das colunas de texto.

```
# Pegando apenas 5 colunas por questão de espaço
ames %>%
  summarise(across(where(is.character), n_distinct)) %>%
  select(1:5)
```

```
## # A tibble: 1 x 5
##   lote_formato lote_config terreno_contorno terreno_declive rua_tipo
##         <int>         <int>         <int>         <int>         <int>
## 1             4             5             4             3             2
```

Todas as colunas da base resultante eram colunas com classe `character` na base `ames`.

Antes, utilizávamos a função `summarise_if()`.

```
# Pegando apenas 5 colunas por questão de espaço
ames %>%
  summarise_if(is.character, n_distinct) %>%
  select(1:5)
```

```
## # A tibble: 1 x 5
##   lote_formato lote_config terreno_contorno terreno_declive rua_tipo
##   <int>         <int>         <int>         <int>         <int>
## 1           4           5           4           3           2
```

Você também pode combinar as ações do `summarise_if()` e `summarise_at()` em um único `across()`. Calculamos as áreas médias, garantindo que pegamos apenas variáveis numéricas.

```
Pegando apenas 5 colunas por questão de espaço
mes %>%
  summarise(across(where(is.numeric) & contains("area"), mean, na.rm = TRUE)) %>%
  select(1:5)
```

```
## # A tibble: 1 x 5
##   lote_area alvenaria_area porao_area_com_ac... porao_area_com_a... porao_area_sem...
##   <dbl>         <dbl>         <dbl>         <dbl>         <dbl>
## 1   10148.         102.         443.         49.7         559.
```

Com a função `across()`, podemos fazer sumarizações complexas.

```
# Pegando apenas 5 colunas por questão de espaço
ames %>%
  group_by(fundacao_tipo) %>%
  summarise(
    across(contains("area"), mean, na.rm = TRUE),
    across(where(is.character), ~sum(is.na(.x))),
    n_obs = n(),
  ) %>%
  select(1:4, n_obs)
```

```
## # A tibble: 6 x 5
##   fundacao_tipo lote_area alvenaria_area porao_area_com_acabamento_1 n_obs
##   <chr>          <dbl>         <dbl>          <dbl> <int>
## 1 BrkTil          8712.           10.2           151.    311
## 2 CBlock         10616.           85.0           468.   1244
## 3 PConc          10054.          144.           506.   1310
## 4 Slab           10250.           35.2            0     49
## 5 Stone           8659.            0            43.9    11
## 6 Wood           9838.            16            812.     5
```

Isso não era possível utilizando apenas as funções `summarise()`, `summarise_if()` e `summarise_at()`.

Across outros verbos

Embora a nova sintaxe, usando `across()`, não seja muito diferente do que fazíamos antes, realizar sumarizações complexas não é a única vantagem desse novo *framework*.

O `across()` pode ser utilizado em todos os verbos do `{dplyr}` (com exceção do `select()` e `rename()`, já que ele não trás vantagens com relação ao que já existe) e isso unifica o modo de fazermos essas operações no R. Em vez de termos uma família de funções para cada verbo, temos agora apenas o próprio verbo e o `across()`.

Vamos ver um exemplo para o `mutate()` e para o `filter()`.

O código abaixo transforma todas as variáveis que possuem "area" no nome, passando os valores de pés quadrados para metros quadrados.

```
ames %>%  
  mutate(across(  
    contains("area"),  
    ~ .x / 10.764  
  ))
```

Já o código a seguir filtra apenas as casas que possuem varanda aberta, cerca e lareira.

```
ames %>%  
  filter(across(  
    c(varanda_aberta_area, cerca_qualidade, lareira_qualidade),  
    ~!is.na(.x)  
  ))
```

select()

Não precisamos do `across()` na hora de selecionar colunas. A função `select()` já usa naturalmente o mecanismo de seleção de colunas que o `across()` proporciona.

```
# Pegando apenas 5 colunas por questão de espaço
ames %>%
  select(where(is.numeric)) %>%
  select(1:5)
```

```
## # A tibble: 2,930 x 5
##   lote_fachada lote_area construcao_ano remodelacao_ano geral_qualidade
##   <int>      <int>      <int>      <int>      <int>
## 1       141    31770      1960      1960         6
## 2        80    11622      1961      1961         5
## 3        81   14267      1958      1958         6
## 4        93   11160      1968      1968         7
## 5        74   13830      1997      1998         5
## 6        78    9978      1998      1998         6
## 7        41    4920      2001      2001         8
## 8        43    5005      1992      1992         8
## 9        39    5389      1995      1996         8
## 10       60    7500      1999      1999         7
## # ... with 2,920 more rows
```

rename()

O mesmo vale para o `rename()`. Se quisermos renomear várias colunas, a partir de uma função, utilizamos o `rename_with()`.

```
# Pegando apenas 5 colunas por questão de espaço
ames %>%
  rename_with(toupper, contains("venda")) %>%
  select(73:77)
```

```
## # A tibble: 2,930 x 5
##   VENDA_ANO VENDA_MES VENDA_TIPO VENDA_CONDICAO VENDA_VALOR
##   <int>     <int> <chr>         <chr>          <int>
## 1      2010         5 "WD "         Normal         215000
## 2      2010         6 "WD "         Normal         105000
## 3      2010         6 "WD "         Normal         172000
## 4      2010         4 "WD "         Normal         244000
## 5      2010         3 "WD "         Normal         189900
## 6      2010         6 "WD "         Normal         195500
## 7      2010         4 "WD "         Normal         213500
## 8      2010         1 "WD "         Normal         191500
## 9      2010         3 "WD "         Normal         236500
## 10     2010         6 "WD "         Normal         189000
## # ... with 2,920 more rows
```

relocate()

O {dplyr} possui agora uma função própria para reorganizar colunas: `relocate()`. Por padrão, ela coloca uma ou mais colunas no começo da base.

```
# Pegando apenas 5 colunas por questão de espaço
ames %>%
  relocate(venda_valor, venda_tipo) %>%
  select(1:5)
```

```
## # A tibble: 2,930 x 5
##   venda_valor venda_tipo lote_fachada lote_area lote_formato
##   <int> <chr>          <int>    <int> <chr>
## 1    215000 "WD "             141    31770 IR1
## 2    105000 "WD "              80    11622 Reg
## 3    172000 "WD "              81    14267 IR1
## 4    244000 "WD "              93    11160 Reg
## 5    189900 "WD "              74    13830 IR1
## 6    195500 "WD "              78     9978 IR1
## 7    213500 "WD "              41     4920 Reg
## 8    191500 "WD "              43     5005 IR1
## 9    236500 "WD "              39     5389 IR1
## 10   189000 "WD "              60     7500 Reg
## # ... with 2,920 more rows
```

Podemos usar os argumentos `.after` e `.before` para fazer mudanças mais complexas.

O código baixo coloca a coluna `venda_ano` depois da coluna `construcao_ano`.

```
ames %>%  
  relocate(venda_ano, .after = construcao_ano)
```

O código baixo coloca a coluna `venda_ano` antes da coluna `construcao_ano`.

```
ames %>%  
  relocate(venda_ano, .before = construcao_ano)
```


rowwise()

Por fim, vamos discutir operações feitas por linha. Tome como exemplo a tabela abaixo. Ela apresenta as notas de alunos em quatro provas.

```
tab_notas <- tibble(  
  student_id = 1:5,  
  prova1 = sample(0:10, 5),  
  prova2 = sample(0:10, 5),  
  prova3 = sample(0:10, 5),  
  prova4 = sample(0:10, 5)  
)
```

```
tab_notas
```

```
## # A tibble: 5 x 5  
##   student_id prova1 prova2 prova3 prova4  
##       <int> <int> <int> <int> <int>  
## 1         1     10      8      4      3  
## 2         2      8      0     10     10  
## 3         3      0      3      2      2  
## 4         4      7      7      3      1  
## 5         5      3      1      6      7
```

Se quisermos gerar uma coluna com a nota média de cada aluno nas quatro provas, não poderíamos usar o `mutate()` diretamente.

```
tab_notas %>% mutate(media = mean(c(prova1, prova2, prova3, prova4)))
```

```
## # A tibble: 5 x 6
##   student_id prova1 prova2 prova3 prova4 media
##       <int>  <int>  <int>  <int>  <int> <dbl>
## 1         1     10      8      4      3  4.75
## 2         2      8      0     10     10  4.75
## 3         3      0      3      2      2  4.75
## 4         4      7      7      3      1  4.75
## 5         5      3      1      6      7  4.75
```

Neste caso, todas as colunas estão sendo empilhadas e gerando uma única média, passada a todas as linhas da coluna `media`.

Para fazermos a conta para cada aluno, podemos agrupar por aluno. Agora sim a média é calculada apenas nas notas de cada estudante.

```
tab_notas %>%
  group_by(student_id) %>%
  mutate(media = mean(c(prova1, prova2, prova3, prova4)))
```

```
## # A tibble: 5 x 6
## # Groups:   student_id [5]
##   student_id prova1 prova2 prova3 prova4 media
##       <int>   <int>   <int>   <int>   <int> <dbl>
## 1         1      10      8       4       3  6.25
## 2         2       8       0      10      10   7
## 3         3       0       3       2       2  1.75
## 4         4       7       7       3       1  4.5
## 5         5       3       1       6       7  4.25
```

c_across()

Também podemos nos aproveitar da sintaxe do `across()` neste caso. Para isso, precisamos substituir a função `c()` pela função `c_across()`.

```
tab_notas %>%  
  group_by(student_id) %>%  
  mutate(media = mean(c_across(starts_with("prova"))))
```

```
## # A tibble: 5 x 6  
## # Groups:   student_id [5]  
##   student_id prova1 prova2 prova3 prova4 media  
##   <int>    <int>    <int>    <int>    <int> <dbl>  
## 1         1      10         8         4         3  6.25  
## 2         2         8         0        10        10   7  
## 3         3         0         3         2         2  1.75  
## 4         4         7         7         3         1  4.5  
## 5         5         3         1         6         7  4.25
```

Equivalentemente ao `group_by()`, neste caso, podemos usar a função `rowwise()`.

```
tab_notas %>%  
  rowwise(student_id) %>%  
  mutate(media = mean(c_across(starts_with("prova"))))
```

```
## # A tibble: 5 x 6  
## # Rowwise:  student_id  
##   student_id prova1 prova2 prova3 prova4 media  
##       <int>  <int>  <int>  <int>  <int> <dbl>  
## 1         1     10     8     4     3  6.25  
## 2         2      8     0    10    10   7  
## 3         3      0     3     2     2  1.75  
## 4         4      7     7     3     1  4.5  
## 5         5      3     1     6     7  4.25
```

Ela é muito útil quando queremos fazer operação por linhas, mas não temos uma coluna de identificação. Por padrão, se não indicarmos nenhuma coluna, cada linha será um "grupo".

```
tab_notas %>%  
  rowwise() %>%  
  mutate(media = mean(c_across(starts_with("prova"))))
```

```
## # A tibble: 5 x 6  
## # Rowwise:  
##   student_id prova1 prova2 prova3 prova4 media  
##   <int>    <int>  <int>  <int>  <int> <dbl>  
## 1         1      10      8      4      3  6.25  
## 2         2       8      0     10     10   7  
## 3         3       0      3      2      2  1.75  
## 4         4       7      7      3      1  4.5  
## 5         5       3      1      6      7  4.25
```

Veja que `student_id` não é passada para a função `rowwise()`. Não precisaríamos dessa coluna na base para reproduzir a geração da coluna `media` neste caso.