3 - Manipulação de Dados no R

Marcelo Prudente

17 de setembro de 2018

Contents

Seção 1

Manipulação de dados

A manipulação de dados demanda um bom tempo de qualquer analista de dados. Remover colunas, criar colunas, fundir tabelas, renomear variáveis, sumarizar variáveis, entre outros, são tarefas comuns bastante facilitadas pelo programa.

No **R** há diversas formas de manipular dados por meio de diversos pacotes. Podemos elencar, por exemplo, o pacote **base**, já instalado no sistema, o pacote **dplyr** do **tidyverse** ou o pacote **data.table**. Na prática, todos tem o mesmo objetivo, mas a sintaxe a performance são distintas.

Para esse curso, daremos grande ênfase à abordagem do **dplyr**, cuja linguagem é próxima a do SQL, por sua facilidade, disseminação e performance. Se o nosso objetivo fosse trabalhar intensivamente com bases grandes, daríamos ênfase ao **data.table**, mas caso você queira entender melhor essa forma de manipular dados recomendo acessar **este link**.

No entanto, o objetivo deste tópico também é apresentar algumas estratégias importantes para a manipulação dos dados com o pacote **base**. Afinal, é muito comum observar nos fóruns o uso da sintaxe desse pacote e, assim, é necessário conhecê-la. Ainda, essa pequena introdução à manipulação dos dados tem por objetivo explicitar a lógica por trás da linguagem do **R**. Assim, identificaremos as formas alternativas de trabalho.

Portanto, os comandos aprendidos nesta aula trazem "massa crítica" para as aulas seguintes. Para isso, a partir do banco **dados_sociais.csv**, vamos levantar algumas questões recorrentes na análise de dados.

Qual o primeiro passo? Ler os dados!

```r
# limpar ambiente global
rm(list = ls())
# carregar pacotes
library(data.table); library(readr)
# com fread
dados_sociais <- fread("C:/curso_r/dados/dados_sociais.csv", dec = ",")

# com read_csv2 - sep = ";" e dec = ","
dados_sociais <- read_csv2("C:/curso_r/dados/dados_sociais.csv",
                           locale = locale(encoding = "latin1"))
```

1.1 Como estão os meus dados

Com os dados carregados, devemos gastar nossa energia em conhecer os dados para promover as mudanças necessárias (desejadas).

O primeiro passo é olhar a nossa base de dados sociais e verificar se as estruturas de dados foram corretamente importadas.

```r
library(dplyr)
glimpse(dados_sociais)
```

```
## Observations: 16,695
## Variables: 8
## $ ano          <int> 1991, 1991, 1991, 1991, 1991, 1991, 1991, 1991, 1...
## $ uf           <int> 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 1...
## $ cod_ibge     <int> 1100015, 1100023, 1100031, 1100049, 1100056, 1100...
## $ municipio    <chr> "ALTA FLORESTA D'OESTE", "ARIQUEMES", "CABIXI", "...
## $ esp_vida     <dbl> 62.01, 66.02, 63.16, 65.03, 62.73, 64.46, 59.32, ...
## $ tx_analf_15m <dbl> 23.55, 17.18, 24.57, 21.41, 20.26, 25.44, 30.49, ...
## $ pop          <int> 23417, 56061, 7601, 69173, 19451, 25441, 11968, 7...
## $ rdpc         <dbl> 198.46, 319.47, 116.38, 320.24, 240.10, 224.82, 8...
```

Lembre-se que a manipulação permite melhorar a precisão dos dados analisados. Com o **R** é possível fazer isso de forma rápida e transparente, sobretudo em comparação ao uso de planilhas do Excel, por exemplo.

1.2 Manipulação de dados com o R

Entre os mais de 12.000 pacotes do **R**, alguns foram especificamente desenhados para manipular dados e explorar os dados.

Para as aulas, utilizaremos os pacotes do tidyverse:

- *dplyr* - manipulação dos dados
- *lubridate* - manipulação de datas
- *reshape2* - pivotar dados
- *tidyr* - pivotar dados
- *ggplot2* - gráficos
- Menção honrosa: *data.table()*

Seção 2

Manipulação de dados com dplyr

O pacote *dplyr* é hoje um dos mais utilizados para a manipulação de dados por algumas razões:

- sintaxe amigável (próxima à do SQL)
- diversos tutoriais disponíveis (livros, artigos no Rpubs, discussões no stackoverflow)

2.1 Comandos básicos do dplyr

Os seis mais importantes comandos (ou verbos) de manipulação de dados do *dplyr* são:

1. filter() - seleciona linhas
2. select() - seleciona colunas
3. arrange() - ordena os dados
4. mutate() - cria novas variáveis e renomeia
5. group_by() - agrupa os dados
6. summarise()- sumariza os dados

Lembre desses comandos! Eles serão seus grandes amigos!!

2.1.1 filter (): filtrando as linhas

Um dos problemas mais básicos da manipulação de dados é a necessidade de extrair subconjuntos dessas informações. Por exemplo, em dados segregados por Estado, você pode se interessar por analisar apenas essa unidade específica ou mesmo uma região (Nordeste ou

Sul). Mas como implementar isso? Com o *dplyr*, essa é uma tarefa intuitiva, conforme o comando abaixo. Observe o uso do operador lógico da igualdade.

```
# filtrar apenas informações do ano de 2010
dados_2010 <-  filter(dados_sociais, ano == 2000)
```

A aborgagem do dplyr é simples, mas a sua contraparte da base do sistema também:

```
dados_2010 <- subset(dados_sociais, ano == 2010)
```

Fácil, não?

2.1.1.1 Exercício

Tente filtrar as linhas dos dados de tal forma que: - o ano seja 2010 - a taxa de analfabetismo seja maior que a média - o Estado seja do Nordeste

Dica: utilize os comandos lógicos e de medida aprendidos no capítulo 1.

2.2 filter(): string

É possível filtrar os dados que contenham textos. Nesse caso, utilizamos a função *grepl()*. Para maior informação, lembre-se sempre de olhar a documentação do comando ?grepl.

```
filter(dados_sociais, grepl("Araca", municipio, ignore.case = TRUE ))
```

```
## Warning: package 'bindrcpp' was built under R version 3.4.4
```

```
## # A tibble: 42 x 8
##      ano    uf cod_ibge municipio  esp_vida tx_analf_15m    pop  rdpc
##    <int> <int>    <int> <chr>         <dbl>        <dbl>  <int> <dbl>
##  1  1991    14  1400209 CARACARAÍ      63.7         32.8   6810 310.
##  2  1991    15  1504307 MARACANÃ       63.6         24.0  25485 117.
##  3  1991    21  2106326 MARACAÇUMÉ     57.0         44.3  11174 118.
##  4  1991    23  2301109 ARACATI        59.7         40.8  49704 159.
##  5  1991    23  2307650 MARACANAÚ      64.0         22.9 160065 174.
##  6  1991    25  2509305 MATARACA       54.1         51.3   5508 118.
##  7  1991    28  2800308 ARACAJU        63.3         14.4 392428 513.
##  8  1991    29  2902005 ARACATU        62.4         52.4  17549  72.8
```

```
##  9  1991    31  3147006 Paracatu       66.1         17.2  62096 322.
## 10  1991    33  3303609 PARACAMBI      64.0         19.0  34981 307.
## # ... with 32 more rows
```

2.3 select(): selecionando as colunas

Às vezes, não queremos filtrar as linhas, mas apenas selecionar algumas colunas, sobretudo quando estamos diante de bancos com muitas variáveis (+ 100). Mais uma vez, o *dplyr* oferece um comando intuitivo para esse tipo de esforço. Claro, para selecionar as colunas é necessário conhecer seus nomes.

```
# quais os nomes das colunas
colnames(dados_sociais)

# selecionar ano, uf e taxa de analfabetismo
dados_select <- select(dados_sociais, ano, uf, tx_analf_15m )

# olhar a nova base
head(dados_select)
```

Note que o comando **subset()**, utilizado para filtrar linhas, também conta com uma opção *select*, conforme o código abaixo:

```
dados_select <- subset(dados_sociais, select = c("ano", "uf", "tx_analf_15m"))
head(dados_select)
```

```
## # A tibble: 6 x 3
##     ano    uf tx_analf_15m
##   <int> <int>        <dbl>
## 1  1991    11         23.6
## 2  1991    11         17.2
## 3  1991    11         24.6
## 4  1991    11         21.4
## 5  1991    11         20.3
## 6  1991    11         25.4
```

Você deve estar se perguntando: qual a vantagem dos comandos do *dplyr* sobre o *subset*? Calma! Chegaremos lá. Antes, vamos a mais alguns exemplos.

Com o comando select é possívelselecionar um intervalo de variáveis de acordo com seus nomes ou mesmo sua posição no banco.

```
# Selecionar as variáveis de nome até municípios
dados_select <- select(dados_sociais, ano:municipio)

# Selecionar as 4 primeiras variáveis pela posição
dados_select <- select(dados_sociais, 1:4)
```

Da mesma forma, podemos excluir variáveis com muita facilidade

```
# Todas variáveis, exceto município
dados_select <- select(dados_sociais, -municipio)
```

Ainda, como em qualquer operação do **R**, é possível criar um vetor com as variáveis que se quer selecionar.

```
# Selecionar as variáveis uf e tx_analf_15m
minha_selecao <- c("uf", "tx_analf_15m")
dados_select <- dados_sociais %>% select(one_of(minha_selecao))
# mostrar o resultado
head(dados_select)
```

Ainda, quem pode selecionar tem o poder de reordenar as colunas.

```
# Você pode reordenar grupos
dados_select <- select(dados_sociais,  cod_ibge:rdpc, ano:uf)

# Você pode reordenar apenas 1 variável - note o everything
dados_select <- select(dados_sociais, cod_ibge, everything())
```

Atente: o **everything()** retorna todas outras variáveis do banco.

2.4 select(): select helpers

Séries de funções que permitem selecionar os nomes das variáveis de acordo com os seus nomes.

```
# ler dados datasus
library(read.dbc)
```

```r
sihsus <- read.dbc("C:/curso_r/dados/RDSE1701.dbc")

# nome das colunas
colnames(sihsus)
```

```
##   [1] "UF_ZI"      "ANO_CMPT"   "MES_CMPT"   "ESPEC"      "CGC_HOSP"
##   [6] "N_AIH"      "IDENT"      "CEP"        "MUNIC_RES"  "NASC"
##  [11] "SEXO"       "UTI_MES_IN" "UTI_MES_AN" "UTI_MES_AL" "UTI_MES_TO"
##  [16] "MARCA_UTI"  "UTI_INT_IN" "UTI_INT_AN" "UTI_INT_AL" "UTI_INT_TO"
##  [21] "DIAR_ACOM"  "QT_DIARIAS" "PROC_SOLIC" "PROC_REA"   "VAL_SH"
##  [26] "VAL_SP"     "VAL_SADT"   "VAL_RN"     "VAL_ACOMP"  "VAL_ORTP"
##  [31] "VAL_SANGUE" "VAL_SADTSR" "VAL_TRANSP" "VAL_OBSANG" "VAL_PED1AC"
##  [36] "VAL_TOT"    "VAL_UTI"    "US_TOT"     "DT_INTER"   "DT_SAIDA"
##  [41] "DIAG_PRINC" "DIAG_SECUN" "COBRANCA"   "NATUREZA"   "NAT_JUR"
##  [46] "GESTAO"     "RUBRICA"    "IND_VDRL"   "MUNIC_MOV"  "COD_IDADE"
##  [51] "IDADE"      "DIAS_PERM"  "MORTE"      "NACIONAL"   "NUM_PROC"
##  [56] "CAR_INT"    "TOT_PT_SP"  "CPF_AUT"    "HOMONIMO"   "NUM_FILHOS"
##  [61] "INSTRU"     "CID_NOTIF"  "CONTRACEP1" "CONTRACEP2" "GESTRISCO"
##  [66] "INSC_PN"    "SEQ_AIH5"   "CBOR"       "CNAER"      "VINCPREV"
##  [71] "GESTOR_COD" "GESTOR_TP"  "GESTOR_CPF" "GESTOR_DT"  "CNES"
##  [76] "CNPJ_MANT"  "INFEHOSP"   "CID_ASSO"   "CID_MORTE"  "COMPLEX"
##  [81] "FINANC"     "FAEC_TP"    "REGCT"      "RACA_COR"   "ETNIA"
##  [86] "SEQUENCIA"  "REMESSA"    "AUD_JUST"   "SIS_JUST"   "VAL_SH_FED"
##  [91] "VAL_SP_FED" "VAL_SH_GES" "VAL_SP_GES" "VAL_UCI"    "MARCA_UCI"
##  [96] "DIAGSEC1"   "DIAGSEC2"   "DIAGSEC3"   "DIAGSEC4"   "DIAGSEC5"
## [101] "DIAGSEC6"   "DIAGSEC7"   "DIAGSEC8"   "DIAGSEC9"   "TPDISEC1"
## [106] "TPDISEC2"   "TPDISEC3"   "TPDISEC4"   "TPDISEC5"   "TPDISEC6"
## [111] "TPDISEC7"   "TPDISEC8"   "TPDISEC9"
```

```r
# selecionar colunas que começam com VAL
sihsus1 <- select(sihsus, starts_with("val"))
head(sihsus1)
```

```
##    VAL_SH  VAL_SP VAL_SADT VAL_RN VAL_ACOMP VAL_ORTP VAL_SANGUE VAL_SADTSR
## 1 6990.61 1476.66        0      0         0        0          0          0
## 2  745.80  209.98        0      0         0        0          0          0
## 3 7077.69 1699.08        0      0         0        0          0          0
```

```
## 4 2142.77  692.39        0      0         0        0          0          0
## 5  416.38  241.54        0      0         0        0          0          0
## 6  997.50  341.61        0      0         0        0          0          0
##   VAL_TRANSP VAL_OBSANG VAL_PED1AC VAL_TOT VAL_UTI VAL_SH_FED VAL_SP_FED
## 1          0          0          0 8467.27 7659.52          0          0
## 2          0          0          0  955.78    0.00          0          0
## 3          0          0          0 8776.77 7180.80          0          0
## 4          0          0          0 2835.16 1436.16          0          0
## 5          0          0          0  657.92    0.00          0          0
## 6          0          0          0 1339.11    0.00          0          0
##   VAL_SH_GES VAL_SP_GES VAL_UCI
## 1          0          0       0
## 2          0          0       0
## 3          0          0       0
## 4          0          0       0
## 5          0          0       0
## 6          0          0       0
```

```r
# selecionar colunas que contém UTI
sihsus2<- select(sihsus, contains("UTI"))
head(sihsus2)
```

```
##   UTI_MES_IN UTI_MES_AN UTI_MES_AL UTI_MES_TO MARCA_UTI UTI_INT_IN
## 1          0          0          0         16        75          0
## 2          0          0          0          0        00          0
## 3          0          0          0         15        75          0
## 4          0          0          0          3        75          0
## 5          0          0          0          0        00          0
## 6          0          0          0          0        00          0
##   UTI_INT_AN UTI_INT_AL UTI_INT_TO VAL_UTI
## 1          0          0          0 7659.52
## 2          0          0          0    0.00
## 3          0          0          0 7180.80
## 4          0          0          0 1436.16
## 5          0          0          0    0.00
## 6          0          0          0    0.00
```

2.5 Agora, devemos falar do pipe

Uma das vantagens da aborgagem de manipulação de dados do *dplyr* é o pipe. Introduzido pelo pacote *magrittr*, O *pipe* (expresso pelo comando %>%) é uma ferramenta para expressar uma sequência de múltiplas operações com clareza. Em outras palavras, podemos encadear operações (e essa é uma vantagem relevante).

Mas como isso funciona? Basicamente, o pipe transforma **f(x)** em **x %>% y**.

```
library(magrittr)

x = c(1.555, 2.555, 3.555, 4.555)
# tirar o log de x
log(x)

# mesma coisa com o pipe
x %>% log()

# vc pode ir além!
x %>% log() %>% round(2) # UAU!
```

Como o **R** é uma linguagem funcional, o uso dos *pipes* ajuda a reduzir o número de parênteses nas funções e a deixar o código organizado. Além disso. auxilia a leitura dos códigos da direita para a esquerda. Por fim, para a manipulação de dados a ser feita com o *dplyr*, o uso do *pipe* permite um acesso mais fácil às variáveis

Se aplicarmos filtros para as linhas e selecionarmos colunas com o *dplyr*, o código ficará assim:

```
# esperança de vida nos municipios do Estado de Sergipe em 2010
esp_vida_se_2010 <- dados_sociais %>%
  filter(ano == 2010 & uf == 28) %>%
  select(municipio, esp_vida)
head(esp_vida_se_2010, 4)
```

```
## # A tibble: 4 x 2
##   municipio               esp_vida
##   <chr>                      <dbl>
## 1 AMPARO DE SÃO FRANCISCO     68.7
## 2 AQUIDABÃ                    69.8
```

```
## 3 ARACAJU                    74.4
## 4 ARAUÁ                      72
```

Assim, no código acima, o pipe indica a seleção da base dados_sociais, a qual se aplicará um filtro de linhas e serão selecionadas algumas colunas. Muito simples, não? A partir de agora, vamos utilizar sempre o pipe para a manipulação.

2.6 mutate(): criar novas variáveis

A criação de novas variáveis no *R* é bastante facilitada com o *dplyr*. Mostraremos a maneira dessa aborgagem em comparação à forma usual do sistema.

Com o banco de dados sociais, vamos calcular a renda total do município e o logarítimo da população.

```
# encontrar a renda total e logaritmo da populacao
dados_sociais <- dados_sociais %>%
  mutate(renda_total = rdpc * pop,
         log_pop = log(pop))
# veja as novas colunas criadas
head(dados_sociais, 2)
```

```
## # A tibble: 2 x 10
##     ano    uf cod_ibge municipio         esp_vida tx_analf_15m   pop  rdpc
##   <int> <int>    <int> <chr>                <dbl>        <dbl> <int> <dbl>
## 1  1991    11  1100015 ALTA FLORESTA D'~     62.0         23.6 23417  198.
## 2  1991    11  1100023 ARIQUEMES             66.0         17.2 56061  319.
## # ... with 2 more variables: renda_total <dbl>, log_pop <dbl>
```

```
colnames(dados_sociais)
```

```
##  [1] "ano"          "uf"           "cod_ibge"     "municipio"
##  [5] "esp_vida"     "tx_analf_15m" "pop"          "rdpc"
##  [9] "renda_total"  "log_pop"
```

Essa é uma abordagem muito mais concisa em relação à base do sistema, observe:

```
# sintaxe verborrágica
dados_sociais$renda_total <- dados_sociais$rdpc *dados_sociais$pop
```

```
dados_sociais$log_pop <- log(dados_sociais$pop)
```

O mutate é um comando simples e é por meio dele que faremos a maior parte das nossas construções de variáveis.

2.7 rename()

Renomear variáveis pode ser um pequeno pesadela no R. O *dplyr* apresenta uma forma bastente simplificada para renomear variáveis do banco.

```
# Renomear ano e uf
dados_sociais %>%
  rename(ANO = ano,
         UF = uf)
```

2.8 summarise() e group_by(): agregar os dados

Agora, suponha o seu interesse em identificar a média e mediana da esperança de vida dos municípios brasileiros para cada ano. Como fazer isso?

Esse tipo de operação, em que agrupamos uma estatística por grupo, é muito comum e de fácil solução aqui. Devemos sempre perguntar: quero observar esse fenômeno sob qual nível de agregação? Pensando assim, fica fácil entender o comando abaixo.

```
# agrupar por ano
dados_sociais %>% # banco
  group_by(ano) %>% # agrupa por ano
  summarise(media = mean(esp_vida),
            mediana = median(esp_vida))
```

```
## # A tibble: 3 x 3
##     ano media mediana
##   <int> <dbl>   <dbl>
## 1  1991  63.7    64.5
## 2  2000  68.4    69.0
## 3  2010  73.1    73.5
```

```r
# agrupar por ano e uf
dados_sociais %>% # banco
  group_by(ano, uf) %>% # agrupa por ano
  summarise(media = mean(esp_vida),
            mediana = median(esp_vida)) %>%
  head(10)
```

```
## # A tibble: 10 x 4
## # Groups:   ano [1]
##      ano    uf media mediana
##    <int> <int> <dbl>   <dbl>
##  1  1991    11  62.1    62.7
##  2  1991    12  63.0    63.6
##  3  1991    13  61.7    61.9
##  4  1991    14  61.4    60.4
##  5  1991    15  62.5    62.9
##  6  1991    16  63.2    63.9
##  7  1991    17  59.9    60.2
##  8  1991    21  57.0    57.0
##  9  1991    22  58.6    58.8
## 10  1991    23  59.9    59.8
```

- Note que executamos duas operações com o *pipe*. Primeiro, agrupamos pela variável de interesse. Depois, pedimos a média para cada um desses anos.

2.9 Exercício

Filtre os dados Extraia a média da esperança de vida por Estado a cada ano. Armazene os dados em um objeto chamado: "med_esp_ano_uf"

Seção 3

Mesclar dados no R

3.1 JOIN (merge): melhor que o PROCV

Em geral, as bases de dados disponíveis não estão completas: precisam ser cruzadas para obter maiores informações. Por exemplo, no caso da base **dados_sociais**, não foi especificada a região a qual os municípios e estados pertencem. Como proceder?

No excel, certamente você iria se atrapalhar um pouco até aprender a utilizar o PROCV. Felizmente, o **R** apresenta soluções rápidas e intuitivas para essa atividade.

Examinando os dados, nota-se que as regiões representam o primeiro número da variável *uf*.

Código	Região
1	Norte
2	Nordeste
3	Sudeste
4	Sul
5	Centro Oeste

Portanto, essa é uma chave para mesclar as tabelas.

3.2 JOIN (merge): junção natural

- Para ilustrar as formas de mesclar dados no **R**, vamos criar dois pequenos dataframes.

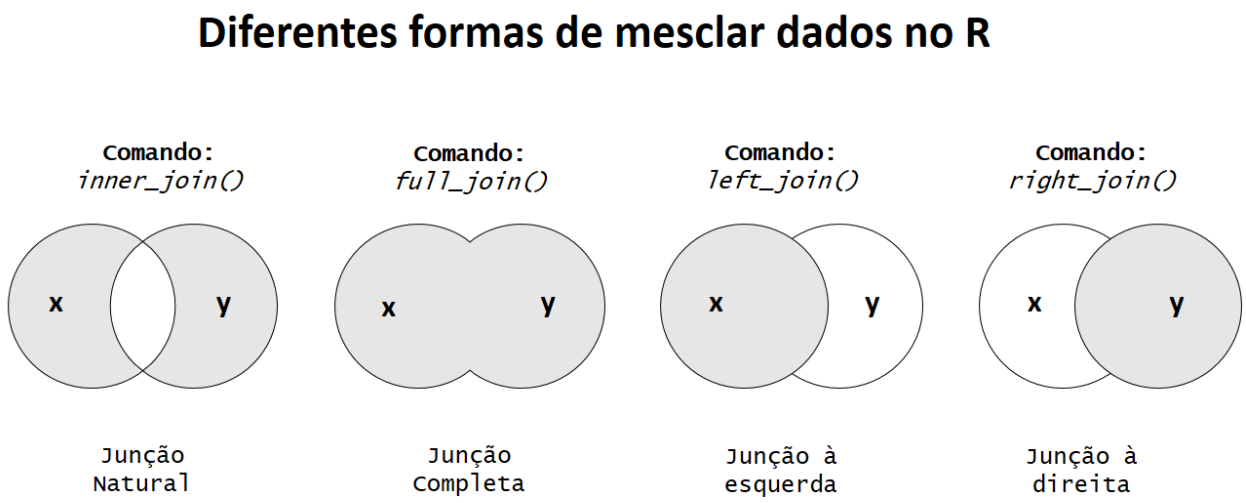

Fig. 3.1: Ilustração das formas de merge no R

Se os nomes das variáveis chaves são iguais, o *dplyr* as identifica

```
inner_join( x , y )     é igual a   inner_join( x , y , by = "chave" )
```

Se os nomes das variáveis chaves são distintos, devem ser identificados

```
inner_join( x , y , by = c( "chave_de_x" = "chave_de_y"  ) )


inner_join( x , y , by = c( "chave1_de_x" = "chave1_de_y" ,
                            "chave2_de_x" = "chave2_de_y" ) )
```

Fig. 3.2: Ilustração das formas de merge no R

```r
df1 <- tibble(letras = letters[1:8], X = 1:8)
df2 <- tibble(letras = letters[5:12], Y = 1:8)
```

- Assim, o comando geral é:

```r
# Apenas os dados em comum
inner_join(df1, df2)
# Idêntico, mas preferível!
inner_join(df1, df2, by = "letras")
```

3.3 JOIN (merge): outros casos

```r
# Junção total
full_join(df1, df2, by = "letras")

# Junção à esquerda
left_join(df1, df2, by = "letras")

# Junção à direita
right_join(df1, df2, by = "letras")
```

- Ainda, pode-se mesclar apenas os dados não coincidentes.

```r
anti_join(df1, df2, by ="letras")
anti_join(df2, df1, by ="letras")
```

Seção 4

Exercícios

4.1 Fixação do comandos básicos

- Aplicaremos os comandos aprendidos para efetura análises dos dados sobre o programa Financiamento Estudantil (FIES) e do banco *dados_sociais*.
- Essa base administrativa retrata a população dos alunos matriculados no programa FIES.
- Para esse execício, será utilizada uma amostra de 10% das observações.
- Acesse o arquivo: **exercicio_fixacao_dplyr.pdf**

4.2 Primeiro passo: quem são as variáveis?

- Vamos seguir os seguintes passos:

1. No diretório atual encontram-se os arquivos? Tente utilizar **list.files()**.
2. Baixe o arquivo **fies_sample.csv**.

3. Cheque a estrutura do banco. **str()**
4. Identifique o nome das variáveis **colnames()**. Se quiser, salve em um objeto.

4.3 select(): selecionando variáveis relevantes

- O banco tem 50 variáveis. Nem todas são relevantes.

- Selecione algumas variáveis relevantes e salve no objeto fies_sub:
  - UF, código do contrato, raça, sexo, valor da mensalidade, nome da mantenedora, a data de nascimento, quantidade de semestres financiados, descrição e código do curso e situação de ensino médio escola pública .
- **Pergunta:** é possível selecionar as variáveis apenas por alguns atributos dos nomes (ex: DS, CO, NO, ST ou QT)?

4.4 Dicas

- Em caso de dúvida, use os mecanismos de ajuda:
  - help(comando): help(mutate)
  - ?comando: ?mutate
  - também use os *cheetsheets* do *dplyr* **aqui** ou na pasta *cheet_sheets*
  - ou acesse a página oficial do *dplyr* **aqui**

Seção 5

Mais funções para manipulação de dados

5.1 funções auxiliares

- Na sumarização dos dados, algumas funções são muito úteis para

Função	Decrição
n()	Número de observações no grupo
n_distinct()	Valores únicos de um vetor
cumsum()	Soma cumulativa
rank()	Ranqueia Variáveis
any()	Alguns valores são verdadeiros?
all()	Todos valores são verdadeiros?
quantile()	Quantis
ifelse()	Criar booleano

Acesse a pasta cheat_sheet para mais dicas

5.2 arrange(): classificando os dados

- No *Excel* é comum ordenar os dados. O comando **arrange()** permite fazer isso com muita facilidade.

```
# Exibir os dados de acordo com a menor população
dados_sociais %>% arrange(pop)
```

- **Exercício**: de acordo com a base, quais os três municípios com a menor população no ano de 2010?

5.3 n(): contando informações

Quando observamos uma base de dados, muitas vezes queremos contar quantos valores uma determinada observação carrega. Essa informação é facilmente adquirida com a combinação do comando group_by() e summarise.

```
# Utilizar o exemplo do datasus
library(read.dbc)
sihsus <- read.dbc("C:/curso_r/dados/RDSE1701.dbc")

# contar quantos casos por municipio
sihsus %>%
  group_by(MUNIC_RES) %>%
  summarise(n = n()) %>%
  arrange(-n)
```

```
## # A tibble: 136 x 2
##    MUNIC_RES     n
##    <fct>     <int>
##  1 280030     1732
##  2 280480      671
##  3 280350      393
##  4 280670      326
##  5 280210      296
##  6 280290      214
##  7 280130      213
##  8 280740      170
##  9 280020      141
## 10 280300      126
## # ... with 126 more rows
```

5.4 distinct(): extirpando linhas repetidas

Um problema comum em dados administrativos é a repetição de registros - por exemplo, uma mesma observação é repetida em diversos meses seguidos. As estimativas que fizemos dos contratos do FIES não são tão precisas pois há grande números de contratos repetidos ao longo dos meses. No entanto, buscamos apenas informações únicas sobre os contratos - por exemplo, quantas pessoas inscritas pelo FIES são do sexo feminino e cursam direito. Para isso, devemos identificar a existência de casos repetidos e a de casos únicos.

```
# há casos duplicados?
fies_sample %>%
  select(CO_CONTRATO_FIES) %>%
  duplicated() %>%
  sum()
```

```
## [1] 200351
```

```
# quantos casos únicos?
fies_sample %>%
  summarise(unicos = n_distinct(CO_CONTRATO_FIES))
```

```
##   unicos
## 1 606856
```

- Você percebeu que a soma do resultado do comando duplicated() é uma soma de um vetor lógico?*

Como a base administrativa do FIES é semestral, cada contrato tem diversas observações para cada mês. Então, é necessário encontrar as observações únicas:

```
fies_sub_dist <- fies_sample %>%
  distinct(CO_CONTRATO_FIES, .keep_all = TRUE)
```

O argumento *.keep_all = TRUE* mantém todas as variáveis no banco.

5.5 Exemplo

- Veja um exemplo de como encontrar o valor médio da mensalidade

```
mens_uf <- fies_sample %>%
  group_by(SG_UF) %>%
  select(SG_UF, VL_MENSALIDADE)%>%
  summarise(media_mens = mean(VL_MENSALIDADE, na.rm = TRUE))
mens_uf
```

```
## # A tibble: 27 x 2
##    SG_UF media_mens
##    <chr>      <dbl>
##  1 AC         1161.
##  2 AL          841.
##  3 AM          831.
##  4 AP         1300.
##  5 BA         1060.
##  6 CE         1121.
##  7 DF         1136.
##  8 ES         1155.
##  9 GO         1060.
## 10 MA          959.
## # ... with 17 more rows
```

Veja como o pipe %>% permite encadear uma grande quantidade de comandos.

5.6 ifelse(): criar booleno

Outra função comum para organizar base de dados é a criação de booleanos. Imagine que você quer categorizar as mensalidades médias entre os estados de acordo com um determinado valor. Se a mensalidade for superior a R$1.000,00, você classificará como **1**, nos outros casos como **0**. Essa operação é muito simples. Como você está criando uma nova coluna, há necessidade de utilizar o *mutate()* em conjunto com um operador lógico.

Primeiro, verifique quais os argumentos do *ifelse()*. ?ifelse

```
mens_uf %>%
  mutate(acima_mil = ifelse(media_mens >= 1000, 1, 0),
         acima_mil2 = ifelse(media_mens >= 1000, "Maior que R$1.000",
                             "Menor que R$1.000"))
```

```
## # A tibble: 27 x 4
##    SG_UF media_mens acima_mil acima_mil2
##    <chr>      <dbl>     <dbl> <chr>
##  1 AC         1161.         1 Maior que R$1.000
##  2 AL          841.         0 Menor que R$1.000
##  3 AM          831.         0 Menor que R$1.000
##  4 AP         1300.         1 Maior que R$1.000
##  5 BA         1060.         1 Maior que R$1.000
##  6 CE         1121.         1 Maior que R$1.000
##  7 DF         1136.         1 Maior que R$1.000
##  8 ES         1155.         1 Maior que R$1.000
##  9 GO         1060.         1 Maior que R$1.000
## 10 MA          959.         0 Menor que R$1.000
## # ... with 17 more rows
```

O comando não limita a criação de booleanos numéricos, pois permite criar o mesmo um classificador de texto. Ainda, é possível fazer múltiplas condições encadeadas

```
mens_uf %>%
  mutate(tres_classe = ifelse(media_mens <900, 0,
                              ifelse(media_mens >= 900 & media_mens <= 1100, 1, 2)))
```

```
## # A tibble: 27 x 3
##    SG_UF media_mens tres_classe
##    <chr>      <dbl>       <dbl>
##  1 AC         1161.           2
##  2 AL          841.           0
##  3 AM          831.           0
##  4 AP         1300.           2
##  5 BA         1060.           1
##  6 CE         1121.           2
##  7 DF         1136.           2
##  8 ES         1155.           2
##  9 GO         1060.           1
## 10 MA          959.           1
## # ... with 17 more rows
```

5.7 round() - arredondar

No *Excel* arredondar os números exige pouco esforço. Assim também ocorre no **R**.

```
# Gerar uma distribuição normal aleatória
x <- rnorm(10, 5, 1)
# Arredondar
round(x)
# Arredondar com duas casas decimais
round(x, digits = 2)
# Ou ainda...
round(x, 2)
```

Claro, você pode aplicar isso à lógica de manipulação do *dplyr*. Nesse caso, vamos aplicar não só o arredondamento comum, mas o *ceiling()* e o *floor()*.

```
#
mens_uf %>%
  mutate(media_mens1 = round(media_mens, 1),
         media_mens2 = ceiling(media_mens),
         media_mens3 = floor(media_mens))
```

```
## # A tibble: 27 x 5
##    SG_UF media_mens media_mens1 media_mens2 media_mens3
##    <chr>      <dbl>       <dbl>       <dbl>       <dbl>
##  1 AC         1161.       1161.        1162        1161
##  2 AL          841.        841          841         840
##  3 AM          831.        831.         831         830
##  4 AP         1300.       1300.        1301        1300
##  5 BA         1060.       1060         1061        1060
##  6 CE         1121.       1121.        1122        1121
##  7 DF         1136.       1136         1137        1136
##  8 ES         1155.       1155.        1155        1154
##  9 GO         1060.       1060.        1060        1059
## 10 MA          959.        959          960         959
## # ... with 17 more rows
```

5.8 any() - algum valor é verdadeiro?

- Em um banco grande não é possível inspecionar visualmente elementos como os *NAs* ou outras informações.
- A função *any()* permite identificar facilmente se algum elemento possui determinada característica especificada.

```
# Algum elemento do banco "df_na" é NA?
any(is.na(df_na))
# A coluna letras do banco "df_na" é NA?
any(is.na(df_na$letras))
# A coluna letras do banco "df_na" contém a letra E?
any(df_na == "E")
any(df_na$idade >10)
```

Outra utilidade é a criação de variáveis de acordo com grupos. Para explicitar a situação, Para o exemplo, vamos utilizar dados de uma pesquisa de domicílios.

```
# baixar os dados
dom <- read_csv2("C:/curso_r/dados/dados_domiciliares.csv",
          locale = locale(encoding = "Latin1"))
```

```
## Using ',' as decimal and '.' as grouping mark. Use read_delim() for more control.
```

```
## Parsed with column specification:
## cols(
##   domicilio = col_double(),
##   condicao_domicilio = col_character(),
##   sexo = col_character(),
##   idade = col_integer(),
##   cor = col_character(),
##   sabe_ler = col_character(),
##   escolaridade = col_character()
## )
```

```
# olhar os dados
glimpse(dom)
```

```
## Observations: 4,000
## Variables: 7
```

```
## $ domicilio          <dbl> 1.1e+10, 1.1e+10, 1.1e+10, 1.1e+10, 1.1e+10...
## $ condicao_domicilio <chr> "Pessoa responsável pelo domicílio", "Filho...
## $ sexo               <chr> "Mulher", "Homem", "Mulher", "Homem", "Home...
## $ idade              <int> 51, 22, 43, 33, 6, 24, 22, 36, 50, 18, 17, ...
## $ cor                <chr> "Parda", "Parda", "Parda", "Branca", "Parda...
## $ sabe_ler           <chr> "Não", "Sim", "Não", "Não", "Sim", "Sim", "...
## $ escolaridade       <chr> "Regular do ensino fundamental ou do 1º gra...
```

```
# nome das colunas
colnames(dom)
```

```
## [1] "domicilio"          "condicao_domicilio" "sexo"
## [4] "idade"              "cor"                "sabe_ler"
## [7] "escolaridade"
```

Agora, como saber quantos domicílio tem idosos (pessoas acima de 60 anos)? Você pode agrupar os dados por domicílios (*group_by()*) e utilizar o comando *any()* em conjunto com o *ifelse()*.

```
dom <- dom %>%
  group_by(domicilio) %>%
  mutate(dom_idoso = ifelse(any(idade >= 60),
                            "Dom. tem idoso", "Não tem idoso"))


# quantos domicílios tem idosos
dom %>%
  distinct(dom_idoso , .keep_all = TRUE) %>%
  group_by(dom_idoso) %>%
  count()
```

```
## # A tibble: 2 x 2
## # Groups:   dom_idoso [2]
##   dom_idoso          n
##   <chr>          <int>
## 1 Dom. tem idoso   340
## 2 Não tem idoso    975
```

5.9 cut(): transformar dados numéricos em categóricos

Imagine que você tem um vetor com a idade de diversos indivíduo e há a necessidade de reclassifica-la faixas etárias. Por exemplo, a cada 5 anos.

```
# cortar as idades em intervalos de 5 anos
dom <- dom %>%
  mutate( idade_cut = cut(idade, seq(0,100, 5)))
```

É claro que essa solução, embora prática, não gera um resultado legível. Nesse caso, você pode utilizar uma função pronta, como a *age_cat*, que está na nossa pasta de funções.

```
# ler uma função criada
source("C:/curso_r/funcoes/age_cat.R")
# ler os argumentos da funcao
args(age.cat)
```

```
## function (x, lower = 0, upper, by = 10, sep = "-", above.char = "+")
## NULL
```

```
# criar categoria de idades
dom <- dom %>%
  mutate( idade_cat = age.cat(idade, lower = 0,
                              upper = 90))
```

5.10 paste(): concatenar strings

-Muitas vezes faz-se necessário editar elementos de texto no **R**. Por exemplo, os nomes de um banco. - A função paste permite fazer isso de forma direta.

```
# Irmãos Peixoto
irmaos <- c("Edgar", "Edclésia", "Edmar", "Edésia", "Edésio")

# Como colocar os sobrenomes?
paste(irmaos, "Peixoto")
```

No *dplyr*, você poder criar variáveis juntando colunas. Suponha que você quer uma categoria para especificar o sexo e cor das pessoas pesquisadas.

```r
dom <- dom %>%
  mutate(sexo_cor = paste(sexo, "-", cor))

dom %>%
  group_by(sexo_cor) %>%
  count()
```

```
## # A tibble: 10 x 2
## # Groups:   sexo_cor [10]
##    sexo_cor              n
##    <chr>             <int>
##  1 Homem - Amarela       8
##  2 Homem - Branca      592
##  3 Homem - Indígena      6
##  4 Homem - Parda      1240
##  5 Homem - Preta       161
##  6 Mulher - Amarela      4
##  7 Mulher - Branca     606
##  8 Mulher - Indígena     7
##  9 Mulher - Parda     1243
## 10 Mulher - Preta      133
```

5.11 gsub(): padrões e substituição

Há uma série de comandos que facilitam a identificação de padrões e substituição desses valores. Por exemplo, suponha a necessidade de substituir um **-** por uma **@**.

```r
# funcionamento do gsub
gsub("-", "@", "curso-hotmail.com")
```

```
## [1] "curso@hotmail.com"
```

```r
# aplicando a um data.frame
dom %>%
  mutate(sexo_cor2 = gsub("-", "@", sexo_cor)) %>%
  select(sexo_cor2) %>%  #selecionar variável de interesse
  head(10) # mostrar 10 primeiros
```

```
## Adding missing grouping variables: `domicilio`
```

```
## # A tibble: 10 x 2
## # Groups:   domicilio [4]
##      domicilio sexo_cor2
##          <dbl> <chr>
##  1 11000001601 Mulher @ Parda
##  2 11000001601 Homem @ Parda
##  3 11000001603 Mulher @ Parda
##  4 11000001603 Homem @ Branca
##  5 11000001603 Homem @ Parda
##  6 11000001604 Homem @ Parda
##  7 11000001604 Mulher @ Branca
##  8 11000001605 Homem @ Parda
##  9 11000001605 Mulher @ Parda
## 10 11000001605 Homem @ Branca
```

5.12 grepl():padrões e substituição

Outra função para tratamento de strings é a *grepl()*.

```
dom %>% filter(grepl("Pret", cor, ignore.case = TRUE))
```

```
## # A tibble: 294 x 11
## # Groups:   domicilio [200]
##     domicilio condicao_domicil~ sexo  idade cor   sabe_ler escolaridade
##         <dbl> <chr>             <chr> <int> <chr> <chr>    <chr>
##  1    1.10e10 Pessoa responsáv~ Mulh~    60 Preta Não      Antigo primári~
##  2    1.10e10 Pessoa responsáv~ Mulh~    40 Preta Não      Superior - gra~
##  3    1.10e10 Pessoa responsáv~ Homem    83 Preta Não      <NA>
##  4    1.10e10 Pai, mãe, padras~ Homem   103 Preta Não      Classe de alfa~
##  5    1.10e10 Neto(a)           Mulh~    22 Preta Não      Regular do ens~
##  6    1.10e10 Cônjuge ou compa~ Mulh~    77 Preta Sim      <NA>
##  7    1.10e10 Pessoa responsáv~ Homem    34 Preta Não      Regular do ens~
##  8    1.10e10 Filho(a) do resp~ Mulh~     9 Preta Sim      <NA>
##  9    1.10e10 Pessoa responsáv~ Mulh~    24 Preta Não      Regular do ens~
## 10    1.10e10 Cônjuge ou compa~ Mulh~    35 Preta Não      Educação de jo~
```

```
## # ... with 284 more rows, and 4 more variables: dom_idoso <chr>,
## #   idade_cut <fct>, idade_cat <fct>, sexo_cor <chr>
```

5.13 lead() e lag(): achar valores anteriores e posteriores em um vetor

Outra tarefa comum em uma base de dados é tentar observar a variação de uma variável no tempo ou a necessidade de defasar uma série.

```
x = 1:10
# uma defasagem
lag(x)
```

```
##  [1] NA  1  2  3  4  5  6  7  8  9
```

```
# duas defasagens
lag(x, 2)
```

```
##  [1] NA NA  1  2  3  4  5  6  7  8
```

```
# adiantar uma vez
lead(x)
```

```
##  [1]  2  3  4  5  6  7  8  9 10 NA
```

```
# adiantar duas vezes
lead(x, 2)
```

```
##  [1]  3  4  5  6  7  8  9 10 NA NA
```

Claro isso, pode ser aplicado à fórmula básica do *dplyr*.

```
# criar uma série
df = tibble(a = rep(1:5, 3),
            b = rnorm(15, 15, 10))


# lead e lag
df %>%
  mutate(c = lead(b, 1),
         d = lag(b, 2))
```

```
## # A tibble: 15 x 4
##        a      b      c      d
##    <int>  <dbl>  <dbl>  <dbl>
##  1     1  10.3    1.20  NA
##  2     2   1.20   3.57  NA
##  3     3   3.57  35.0   10.3
##  4     4  35.0   27.9    1.20
##  5     5  27.9    7.33   3.57
##  6     1   7.33   7.90  35.0
##  7     2   7.90  15.6   27.9
##  8     3  15.6   10.6    7.33
##  9     4  10.6   22.7    7.90
## 10     5  22.7   23.2   15.6
## 11     1  23.2  -10.6   10.6
## 12     2 -10.6   18.9   22.7
## 13     3  18.9   12.6   23.2
## 14     4  12.6   22.8  -10.6
## 15     5  22.8   NA     18.9
```

No exemplo abaixo, temos um caso mais concreto. Suponha que você quer saber a variação da renda per capita nos municípios do Estado de Rondônia.

```
TO <- dados_sociais %>%
  filter(uf == "11") %>%
  select(ano:municipio, rdpc)

# exibir primeiras linhas
head(TO)
```

```
## # A tibble: 6 x 5
##     ano    uf cod_ibge municipio              rdpc
##   <int> <int>    <int> <chr>                 <dbl>
## 1  1991    11  1100015 ALTA FLORESTA D'OESTE  198.
## 2  1991    11  1100023 ARIQUEMES              319.
## 3  1991    11  1100031 CABIXI                 116.
## 4  1991    11  1100049 CACOAL                 320.
## 5  1991    11  1100056 CEREJEIRAS             240.
## 6  1991    11  1100064 COLORADO DO OESTE      225.
```

```r
# lead
TO <- TO %>%
  group_by(cod_ibge) %>%
  mutate(lag_rdpc = lag(rdpc),
         var_rdpc = (rdpc - lag_rdpc)/ lag_rdpc,
         var_rdpc_pec = paste(round(var_rdpc*100, 2), "%"))
```

5.14 Exercício

- Faça o uso do o lead e lag para todo o banco dados sociais.
- Observe quais municípios tiveram o maior aumento da renda per capita no período.
- Quais tiveram redução.
- Cria uma variável booleana para isso.

5.15 bind_cols e bind_rows

- Os comandos **bind_** permitem ligar colunas e linhas de bancos com dimensões iguais.

```r
# extrair linhas específicas
um  <-  dados_sociais[1:4, ]
dois <- dados_sociais[7011:7014, ]

# ligar em um novo objeto
meu_bind <- bind_rows(um, dois)
```

- Os comandos **bind_** permitem ligar colunas e linhas de bancos com dimensões iguais.

```r
# extrair colunas
um   <- dados_sociais[ , 3 ]
dois <- dados_sociais[ , 8 ]

# ligar em um novo objeto
meu_bind2 <- bind_cols(um, dois)
```

5.16 NAs: valores não especificados ou perdidos

- Ao realizar o **full_join**, o resultado apresenta algumas observações como *NA* (ver exemplo acima).
- Os *NAs* podem representar tanto informações indeterminadas quanto valores propositadamente omitidos.
- De qualquer forma, lidar com os NAs é muito fácil:

```
# Data frame com NAs
df_na <- tibble( letras = LETTERS[1:10],
                 idade =  c( seq(25L, 40L, 5L), NA,
                             NA, 32L, rep(NA, 3)))

# é possível saber se um vetor é NA
is.na(df_na)
# Também é possível remover esses valores
na.omit(df_na)
```

5.17 replace_na() : substituir NAs

- Porém, não é recomendado retirar os NAs sem alguma reflexão. Afinal, eles podem dizer alguma coisa sobre os dados. Ou, ainda, serem apenas campos numúricos não preenchidos.
- Por isso, é melhor substituir os NAs.

```
# Substituindo NAs por números
replace_na(df_na, list(idade = 0))

# Substituindo NAs por textos
replace_na(df_na, list(idade = "Idade não informada"))
```

5.18 quantile()

Para calcular os intervalos de uma distribuição acumulada de uma variável (quantis), há funções bastante diretas.

```r
# calcular quintis
quantile(dom$idade)
```

```
##   0%  25%  50%  75% 100%
##    0   16   32   48  103
```

```r
# calcular decis
quantile(dom$idade, seq(0.1, 1, 0.1))
```

```
##  10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
##    7   14   19   26   32   38   44   51   61  103
```

```r
# calcular os percentis
quantile(dom$idade, seq(0.01, 1, 0.01))
```

```
##     1%     2%     3%     4%     5%     6%     7%     8%     9%    10%
##   0.00   1.00   2.00   2.00   3.00   4.00   4.00   5.00   6.00   7.00
##    11%    12%    13%    14%    15%    16%    17%    18%    19%    20%
##   7.00   8.00   9.00  10.00  10.00  11.00  12.00  12.00  13.00  14.00
##    21%    22%    23%    24%    25%    26%    27%    28%    29%    30%
##  14.00  15.00  15.00  16.00  16.00  17.00  17.73  18.00  19.00  19.00
##    31%    32%    33%    34%    35%    36%    37%    38%    39%    40%
##  20.00  20.00  21.00  22.00  22.00  23.00  24.00  24.00  25.00  26.00
##    41%    42%    43%    44%    45%    46%    47%    48%    49%    50%
##  26.00  27.00  27.00  28.00  29.00  29.00  30.00  31.00  31.00  32.00
##    51%    52%    53%    54%    55%    56%    57%    58%    59%    60%
##  33.00  33.00  34.00  34.00  35.00  35.00  36.00  36.00  37.00  38.00
##    61%    62%    63%    64%    65%    66%    67%    68%    69%    70%
##  39.00  39.00  40.00  40.36  41.00  42.00  42.00  43.00  44.00  44.00
##    71%    72%    73%    74%    75%    76%    77%    78%    79%    80%
##  45.00  45.28  46.00  47.00  48.00  49.00  49.00  50.00  51.00  51.00
##    81%    82%    83%    84%    85%    86%    87%    88%    89%    90%
##  52.00  53.00  54.00  55.00  56.00  57.00  57.00  58.12  60.00  61.00
##    91%    92%    93%    94%    95%    96%    97%    98%    99%   100%
##  62.00  63.00  64.00  66.00  68.00  70.00  73.00  76.00  81.01 103.00
```

No dplyr, a função *ntile()* divide o vetor em n grupos de mesmo tamanho.

```r
dom <- dom %>%
  ungroup() %>% # desagrupa os dados
  mutate(quintil = ntile(idade, 5),
         decil = ntile(idade, 10))

# todos os grupos tem mesmo tamanho
dom %>%
  group_by(decil) %>%
  count()
```

```
## # A tibble: 10 x 2
## # Groups:   decil [10]
##    decil     n
##    <int> <int>
##  1     1   400
##  2     2   400
##  3     3   400
##  4     4   400
##  5     5   400
##  6     6   400
##  7     7   400
##  8     8   400
##  9     9   400
## 10    10   400
```

Seção 6

Como lidar com datas

Lidar com datas pode ser um pequeno problema quando lidamos com dados. Vamos utilizar o exemplo do seguro defeso.

- Primeiro, instale e ative o pacote *lubridate*:
  - você lembra como instalar e ativar pacotes?
  - dica: *inst.*...

6.1 lubridate: transformando as datas em datas (!)

- O lubridate opera datas levando em conta que:
  - y = ano
  - m = mês
  - d = dia
- Assim, você pode transformar *characters* em datas caso tenham dia, mês e ano, da seguinte forma:

```
# ano, mês e dia sem separador
ymd("20180131")
```

```
## [1] "2018-01-31"
```

```
# mês, dia e ano com separador "-"
mdy("01-31-2018")
```

```
## [1] "2018-01-31"
```

```r
# dia, mês e ano com separador "/"
dmy("31/01/2018")
```

```
## [1] "2018-01-31"
```

Observe também que quando as datas são apresentadas no formato mês/ano, não há como transformar em datas. Por isso, você pode fazer uso do *paste()* para completar as suas datas.

```r
# criamos um vetor com datas mes (m) e ano (y)
datas <- c("01/2014", "03/2016")
# adicionamos o dia (d) com o paste0
dmy(paste0("01", datas))
```

```
## [1] "2014-01-01" "2016-03-01"
```

6.2 lubridate: acessando informações das datas

- Estruturar os dados para o formato de datas permite extrair informações básicas para realizar operações lógicas.

```r
# criar um vetor de datas
datas<- ymd(c("20180131", "20170225", "20160512"))
# quais os anos
year(datas)
```

```
## [1] 2018 2017 2016
```

```r
# quais os meses
months(datas) # formato nome
```

```
## [1] "janeiro"   "fevereiro" "maio"
```

```r
month(datas) # fomato numero
```

```
## [1] 1 2 5
```

```r
# dias
day(datas)
```

```
## [1] 31 25 12
```

```r
# dia da semana
wday(datas)
```

```
## [1] 4 7 5
```

6.3 Exercício

- Leia os dados do seguro defeso na pasta dados. Lembre-se de efetuar uma inspeção visual antes.
- Peça o **head()** do banco. Quantas datas você identifica?
- A partir do banco seguro defeso, verifique:
  - as datas presentes no banco tem a classe de data? Utilize **is.Date()**
  - a data de início do defeso ocorre no período abrangido pelo banco de dados?
  - a data de início do defeso ocorre no período abrangido pelo banco de dados?
  - *melhor*: o defeso ocorre no período do banco de dados?
  - como retirar do banco as linhas em que o defeso não corresponda ao período abrangido pelo banco de dados?
  - em que dias os saques das parcelas ocorreram? Qual dia concentra mais saques?