Introdução à Linguagem R: seus fundamentos e sua prática

Respostas dos exercícios de cada capítulo

Capítulo 1 - Noções Básicas do R

Exercício 1

1.1.A) Quando temos um conjunto pequeno de valores a serem somados, podemos utilizar o operador +:

```
32397 + 55405
```

```
## [1] 87802
```

```
### Ou de forma análoga:
n1 <- 32397
n2 <- 55405

n1 + n2
```

```
## [1] 87802
```

1.1.B) Lembre-se que a função `sum()` lhe permite calcular a soma total de um conjunto de valores, de maneira rápida e eficiente. Lembre-se também de criar o objeto `conj` em seu console, antes de calcular a soma. Pois se não você estaria pedindo à função `sum()`, que calculasse a soma de um objeto que **não existe em seu computador**.

```
### Lembre-se de criar o objeto conj
conj <- c(290, 34, 512, 54, 89)

resposta <- sum(conj)
print(resposta)
```

```
## [1] 979
```

```
### Ou de uma maneira bem menos eficiente:

conj[1] + conj[2] + conj[3] + conj[4] + conj[5]
```

```
## [1] 979
```

1.1.C) Considerando $y = 3x^3 - 12x^2 + \frac{1}{15}x + 25$, e $x = 5$, temos que:

```
x <- 5
y_resposta <- 3 * (x^3) - 12 * (x^2) + (1/15) * x + 25
print(y_resposta)
```

```
## [1] 100.3333
```

Exercício 2

1.2.A) Qualquer erro do tipo "objeto 'x' não encontrado", significa que o objeto pelo qual você requisitou não existe atualmente em sua seção do R. Portanto, o erro na questão que diz respeito a um objeto chamado `logica`, está lhe dizendo que você não criou ainda nenhum objeto chamado `logica` em sua sessão. A partir do momento em que você define um valor para o nome `logica`, esse erro passa a não ocorrer mais. Lembre-se que esse erro pode surgir **em qualquer lugar**

(dentro de qualquer função, ou de qualquer operação), pois sem esse objeto (que não existe em seu computador) o R não é capaz de completar a operação pela qual você requisitou.

```
### Suponha que o erro tenha surgido a partir do comando abaixo
### com a função sum()

sum(logica)

### A partir do momento em que defino um valor para logica
### o comando volta a funcionar

logica <- 1:3
sum(logica)
```

```
## [1] 6
```

1.2.B) A função `bind_rows()` (assim como a função `mutate()`) pertence ao pacote `dplyr`, que está fora dos pacotes básicos do R. Ou seja, sempre que você inicia uma nova sessão no R, a função `bind_rows()` não é automaticamente carregada para essa seção, pois ela pertence a um pacote (`dplyr`) que está fora do conjunto de pacotes básicos do R.

Logo, o erro disposto na questão surge quando tentamos acessar a função `bind_rows()`, quando ela ainda não foi carregada para a nossa seção do R. Como comentamos ao longo da seção Pacotes, para acessarmos as funções disponíveis dentro de um pacote, é necessário carregarmos esse pacote para a nossa seção (através da função `library()`). E para carregarmos um pacote para a nossa seção, é necessário que esse pacote esteja instalado em nosso computador.

```
dt1 <- data.frame(1:3)
dt2 <- data.frame(1:5)
### Não consigo acessar a função bind_rows()
bind_rows(dt1, dt2)

library(dplyr)
### Agora eu consigo acessar a função bind_rows()
bind_rows(dt1, dt2)
```

```
##   X1.3 X1.5
## 1    1   NA
## 2    2   NA
## 3    3   NA
## 4   NA    1
## 5   NA    2
## 6   NA    3
## 7   NA    4
## 8   NA    5
```

1.2.C) Como comentamos ao longo da seção Pacotes, para utilizarmos as funções disponíveis em um pacote, precisamos carregar esse pacote para a nossa seção do R. E para carregarmos esse pacote para a nossa seção, ele precisa estar instalá-do em nosso computador. Logo, o erro da questão (que se refere ao pacote `dplyr`) está nos dizendo que o R não foi capaz de encontrar um pacote instalado em sua máquina, que tenha o nome de `dplyr`. Por esse motivo, para utilizar

o pacote `dplyr` com o comando `library()`, você precisa primeiro instalar esse pacote em sua máquina, com o comando `install.packages()`

```
install.packages("dplyr")
```

Exercício 3

1.3.A) Lembre-se que a fórmula do índice Z de uma distribuição normal, usualmente assume a forma:

$$Z = \frac{X - \bar{X}}{\sigma}$$

Sendo que, as variáveis nessa equação são:

- X: a variável em questão.
- $\bar{X}$: média da variável X.
- σ: desvio padrão da variável X.

Logo, os comandos necessários para o cálculo são:

```
### Lembre-se que você precisa criar
### o objeto vec (com o comando abaixo)
### antes que você possa utilizá-lo em operações
### no R
vec <- c(0.5, 1.2, 2.5, 1.3, 2.2, 3.7)

desvio_padrao <- sd(vec)
media <- mean(vec)

resposta <- (media - vec) / desvio_padrao
print(resposta)
```

```
## [1]  1.2278812  0.6139406 -0.5262348  0.5262348 -0.2631174 -1.5787044
```

1.3.B) Lembre-se que o desvio médio de uma variável é simplesmente uma média dos desvios de seus valores em relação a sua média. Em outras palavras, a fórmula de cálculo do desvio médio (DM) de uma variável chamada x, seria:

$$DM = \frac{1}{n} \sum_{i=1}^{n} |x - \bar{x}|$$

Sendo que, as variáveis nessa equação são:

- n: número de observações (ou de valores) que a variável contém.
- $\bar{x}$: média da variável x.
- x: o valor da variável x.

```
### Lembre-se que você precisa criar
### o objeto vec (com o comando abaixo)
```

```r
### antes que você possa utilizá-lo em operações
### no R
vec <- c(0.5, 1.2, 2.5, 1.3, 2.2, 3.7)

desvios <- vec - mean(vec)
total_desvio <- sum(abs(desvios))

resposta <- (1 / length(vec)) * total_desvio
print(resposta)
```

```
## [1] 0.9
```

Referências Bibliográficas

FRIEDL, J. E. F. *Mastering Regular Expressions*. 3. ed. Sebastopol, CA: O'Reilly Media, Inc., 2006. ISBN 0-596-52812-4.

R CORE TEAM. *R Language Definition*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.

Capítulo 2 - Fundamentos da Linguagem R

Exercício 1

2.1.A) Uma lista contendo um vetor em seu primeiro (e único) elemento.

```
list(1:15)
```

```
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

2.1.B) Um vetor atômico contendo a sequência de 1 a 10.

```
1:10
```

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

2.1.C) Uma matriz 5×5 (5 linhas e 5 colunas) contendo valores do tipo `character`.

```
set.seed(1)
a <- sample(
  c("MG", "SP", "DF", "MS"),
  size = 25, replace = TRUE
)
dim(a) <- c(5,5)
print(a)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,] "MG" "MG" "DF" "SP" "MG"
## [2,] "MS" "DF" "DF" "SP" "DF"
## [3,] "DF" "DF" "MG" "SP" "MG"
## [4,] "MG" "SP" "MG" "SP" "MG"
## [5,] "SP" "SP" "MG" "DF" "MG"
```

2.1.D) Um `data.frame` que possui 10 linhas e 2 colunas.

```
data.frame(
  id = 1:10,
  valor = round(rnorm(10), 2)
)
```

```
##    id valor
## 1   1 -0.29
## 2   2 -0.30
## 3   3 -0.41
## 4   4  0.25
## 5   5 -0.89
## 6   6  0.44
## 7   7 -1.24
## 8   8 -0.22
## 9   9  0.38
## 10 10  0.13
```

2.1.E) Uma lista contendo 4 itens (ou 4 elementos).

```
## $estado
## [1] "MG"
##
## $cidade
## [1] "Belo Horizonte"
##
## $n_municipios
## [1] 853
##
## $regiao
## [1] "Sudeste"
```

2.1.F) Um vetor atômico preenchido por 20 `NA`'s.

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

Exercício 2

2.2.A) Essa é uma espécie de pegadinha, não porque ela seja maldosa, mas porque demonstra certos cuidados necessários. Provavelmente, o seu primeiro instinto nessa questão foi utilizar a função `is.vector()`, o que é um bom começo, pois ela é capaz de identificar os objetos `v_rep` e `v_seq` como vetores atômicos. Ao mesmo tempo, essa função também consegue caracterizar os objetos `dt` e `mt` como estruturas diferentes de um vetor atômico.

Entretanto, a função `is.vector()` acaba considerando o objeto `lst` como um vetor também! Lembre-se que, **uma lista é no fundo, um vetor,** onde cada elemento desse vetor pode ser de tipo e estrutura diferentes de seus pares. Portanto, a função `is.vector()` é capaz de identificar se um objeto é um vetor, mas não necessariamente se ele é um vetor atômico. Por esse motivo, para nos certificarmos de que um dado objeto é um vetor atômico, temos que saber se ele, além de um vetor, também é um objeto diferente de uma lista, através da função `is.list()`.

Utilizando o operador `!` sobre `is.list()`, podemos identificar se o objeto **não é uma lista**. Com isso, precisamos apenas conectar esse teste à função `is.vector()` e, dessa maneira, temos um teste lógico que segue a estrutura `is.vector(x) & !is.list(x)`.

```
v_seq <- 10:25
v_rep <- rep("abc", times = 30)
lst <- list(1:10)
mt <- matrix(1:20, nrow = 4, ncol = 5)
dt <- data.frame(15, "A", 1:10)


is.vector(v_rep) & !is.list(v_rep)
```

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

```
is.vector(v_seq) & !is.list(v_seq)
```

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

```
is.vector(lst) & !is.list(lst)
```

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

2.2.B) Nesta questão, ao utilizar a função `is.list()`, você já tem metade do teste lógico necessário para identificar o objeto `lst` como uma lista. Porém, esta questão, também inclui uma pegadinha parecida com a questão anterior. Pois da mesma maneira que uma lista é, no fundo, um vetor; um `data.frame` é, no fundo, uma lista nomeada. Isso significa que, a função `is.list()` também vai nos retornar `TRUE` para qualquer `data.frame`, e se desejamos identificar **apenas** listas, precisamos incrementar o nosso teste lógico de modo que ele possa diferenciar `data.frame`'s de listas.

A função necessária para esse trabalho é `is.data.frame()`, e utilizando novamente o operador `!` sobre o resultado dessa função, podemos identificar qualquer objeto que **não é** um `data.frame`. Com isso, temos um teste lógico que segue a estrutura `is.list(x) & !is.data.frame(x)`.

```
lst <- list(
  estado = "MG",
  cidade = "Belo Horizonte",
  n_municipios = 853,
  regiao = "Sudeste"
)


is.list(lst) & !is.data.frame(lst)
```

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

2.2.C) Quando temos uma lista nomeada (isto é, cada elemento dessa lista possui um nome), podemos descobrir os nomes de cada elemento dessa lista por meio da função `names()`. Logo, para descobrirmos se essa lista inclui um item chamado "estado", precisamos apenas de um teste lógico que possa identificar se o valor `"estado"` está incluso no resultado da função `names()`.

```
lst <- list(
  estado = "MG",
  cidade = "Belo Horizonte",
  n_municipios = 853,
  regiao = "Sudeste"
)


"estado" %in% names(lst)
```

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

```
### Repare que ao aplicarmos o mesmo
### teste sobre o objeto lst_sem_estado
### o resultado é FALSE, indicando que
### essa lista não possui um item chamado
### "estado"

lst_sem_estado <- list(
  regiao = "Sudeste",
  n_municipios = 853
)

"estado" %in% names(lst_sem_estado)
```

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

2.2.D) Basta utilizarmos a função `is.double()` sobre a coluna `total` de `tab`, como está demonstrado abaixo:

```
tab <- data.frame(
  unidade = c("Centro", "Gameleira", "Santa Efigênia", "Centro",
              "Barro Preto", "Centro", "Gameleira", "Centro",
              "Barro Preto", "Santa Efigênia"),
  mes = c(1, 1, 1, 2, 2, 3, 3, 4, 4, 4),
  vendas = c(1502, 1430, 1100, 1200, 1443, 1621, 1854, 2200,
             1129, 1872),
  total = c(5362.14, 5105.1, 3927, 4284, 5151.51, 5786.97,
            6618.78, 7854, 4030.53, 6683.04)
)


### Use a função is.double() sobre a coluna
is.double(tab$total)
```

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

2.2.E) Lembre-se que **as três condições apresentadas no enunciado da questão são dependentes**. Logo, o objeto que você está testando deve se encaixar nas três condições ao mesmo tempo. Em termos técnicos, isso significa que os testes lógicos referentes a cada uma das três condições, devem obrigatoriamente retornar `TRUE` para este objeto. Se ao aplicarmos pelo me-

nos um desses testes, e o resultado for FALSE, isso significa que o objeto em questão não se encaixa no teste lógico como um todo, ou, o resultado geral do teste lógico é FALSE. Para que o R entenda que essas condições são dependentes, e que devem ser satisfeitas ao mesmo tempo, você deve conectar as três condições pelo operador &.

Para satisfazer a primeira condição posta no enunciado, podemos conferir se o resultado da função nrow() (que nos retorna o número de linhas presente em um data.frame) é igual a 10, construindo assim, o seguinte componente do teste: nrow(x) == 10. Já para a segunda condição, podemos pesquisar se o valor "vendas" aparece dentro do resultado da função colnames() (uma outra alternativa seria a função names() que gera o mesmo resultado de colnames()). Como colnames() geralmente nos retorna um conjunto de valores (ao invés de 1 único valor), é importante que você utilize o operador %in% (ao invés do operador ==) para pesquisar pelo valor "vendas". Dessa forma, temos o segundo componente do teste "vendas" %in% colnames(x). Por último, para conferirmos a terceira condição do teste, podemos aplicar a função is.character() sobre a 3° coluna do objeto em questão, criando assim o último componente do teste is.character(x[[3]]).

Sendo assim, temos um teste com a seguinte estrutura: nrow(x) == 10 & "vendas" %in% colnames(x) & is.character(x[[3]]). Perceba abaixo que, o resultado do teste lógico foi FALSE quando aplicado sobre tab, indicando assim, que o objeto tab não se encaixa em pelo menos uma das condições do teste. Para identificar qual dessas condições que tab não se encaixa, você pode observar o resultado de cada um dos três componentes do teste de forma separada.

```
nrow(tab) == 10 & "vendas" %in% colnames(tab) & is.character(tab[[3]])
```

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

Repare abaixo, que os testes nrow(tab) == 10 e "vendas" %in% colnames(tab) nos retornam um valor TRUE, logo, tab satisfaz ambas as condições. Contudo, o último teste resulta em FALSE. Portanto, tab não satisfaz a última condição do teste, ou, dito de outra forma, a 3° coluna de tab não é do tipo character.

```
nrow(tab) == 10
```

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

```
"vendas" %in% colnames(tab)
```

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

```
is.character(tab[[3]])
```

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

2.2.F) Lembre-se que as condições a serem satisfeitas, para que o ano seja considerado bissexto são: 1) o ano deve ser múltiplo de 4; 2) o ano não deve ser múltiplo de 100 a não ser que ele seja múltiplo de 400; 3) se o ano é múltiplo de 400, ele é obrigatoriamente um ano bissexto.

Para identificarmos se um dado número é múltiplo de um outro número, podemos observar se o resto da divisão entre esses dois números é igual a zero. Em outras palavras, se desejamos saber que um dado valor x é múltiplo de um dado valor y, podemos realizar o cálculo $x \div y$ e,

observar se o resto dessa divisão é ou não igual a zero. Lembre-se que no R, temos o operador aritmético %%, que nos retorna justamente o resto da divisão entre dois números.

Tendo isso em mente, para satisfazermos as condições 1 e 3, podemos simplesmente conferir se os resultados das operações `ano %% 4` e `ano %% 400` são iguais a zero, construindo assim os componentes `ano %% 4 == 0` e `ano %% 400 == 0`. Entretanto, como a condição 2 estabelece que o respectivo ano não deve ser múltiplo de 100, podemos aplicar o operador `!` sobre o componente do teste referente a essa condição. Deste modo, temos o componente `!(ano %% 100 == 0)`. Uma outra alternativa para essa condição 2, seria utilizarmos o operador `!=`, que significa "não igual a", ou, "diferente de". Com esse operador teríamos `ano %% 100 != 0`.

Agora que definimos os componentes do teste, precisamos nos atentar à relação de hierarquia entre essas condições. Pois a condição 3 predomina sobre as condições 1 e 2, assim como as condições 3 e 2 prevalecem sobre a condição 1. Com isso, se um dado ano é múltiplo de 400, não nos interessa se ele é ou não múltiplo de 100 ou de 4, ele é um ano bissexto e ponto final. Da mesma forma que, se o ano não é múltiplo de 400, mas ele é múltiplo de 100, ele não é um ano bissexto, mesmo que ele seja múltiplo de 4.

Primeiro, para que o ano seja bissexto, temos duas possibilidades dentro da relação entre as condições 2 e 3. Ou o número do ano é múltiplo de 400, ou ele não é múltiplo de 100. Como essas duas possibilidades são independentes (ou seja, ou o ano é uma coisa, ou ele é outra), podemos conectar essas duas condições pelo operador `|`, construindo assim o componente `(ano %% 100 != 0) | (ano %% 400 == 0)` do teste. Veja alguns exemplos abaixo:

```
ano <- 1240
(ano %% 100 != 0) | (ano %% 400 == 0)
```

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

```
ano <- 3200
(ano %% 100 != 0) | (ano %% 400 == 0)
```

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

```
ano <- 100
(ano %% 100 != 0) | (ano %% 400 == 0)
```

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

Devido a independência entre essas condições (estabelecida pelo operador `|`), o R vai nos retornar `TRUE` caso o valor de ano se encaixe em pelo menos uma dessas duas condições. Isso significa que, se o valor de ano for múltiplo de 400, ele adquire um valor `TRUE` para o teste `ano %% 400 == 0` e, consequentemente, um valor `TRUE` para todo o componente `(ano %% 100 != 0) | (ano %% 400 == 0)`.

Com isso, precisamos apenas conectar esse componente ao outro componente `(ano %% 4 == 0)`, que representa a condição 1, formando assim a estrutura final do teste: `(ano %% 4 == 0) & ((ano %% 100 != 0) | (ano %% 400 == 0))`. Dessa vez, utilizamos o operador que indica dependência (`&`) entre esses dois componentes principais do teste lógico, que são `(ano %% 4 == 0)` e `(ano %% 100 != 0) | (ano %% 400 == 0)`, pois um número que é múltiplo de 4, ainda

pode ser um múltiplo de 100. Portanto, essa condição de dependência apenas assegura que a condição 2 seja respeitada, caso o número atenda a condição 1.

```
### Por exemplo, 2006 não é um ano bissexto
ano <- 2006
(ano %% 4 == 0) & ((ano %% 100 != 0) | (ano %% 400 == 0))
```

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

```
### Mas o ano de 2004 é um ano bissexto
ano <- 2004
(ano %% 4 == 0) & ((ano %% 100 != 0) | (ano %% 400 == 0))
```

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

Exercício 3

2.3.A) O vetor resultante será do tipo `character`.

```
vec <- c(1.2, 2.4, "3.1", 1.9)
typeof(vec)
```

```
## [1] "character"
```

2.3.B) O vetor resultante será do tipo `double`.

```
integers <- 1:3
doubles <- c(2.23, 9.87, 3.2)

vec <- c(integers, doubles)
typeof(vec)
```

```
## [1] "double"
```

2.3.C) O vetor resultante será do tipo `double`

```
vec <- c(1.56, 3L, 1L, 5L,  2.32, 9.87)
typeof(vec)
```

```
## [1] "double"
```

2.3.D) O vetor resultante será do tipo `integer`.

```
vec <- c(TRUE, 1L, FALSE)
typeof(vec)
```

```
## [1] "integer"
```

2.3.E) O vetor resultante será do tipo `character`.

```
vec <- c("p", "b", "c", TRUE, 2L, 4.318)
typeof(vec)
```

```
## [1] "character"
```

Exercício 4

2.4.A) Perceba que as duas condições descritas no enunciado são dependentes, logo, elas precisam ser atendidas ao mesmo tempo. Por isso, os dois componentes do teste lógico são conectados pelo operador &.

```
library(nycflights13)

teste <- flights$month == 5 & flights$carrier == "B6"

flights[teste, ]
```

```
## # A tibble: 4,576 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <dbl>          <dbl>     <dbl>    <dbl>          <dbl>
##  1  2013     5     1      548            600       -12      831            854
##  2  2013     5     1      556            600        -4      818            835
##  3  2013     5     1      557            600        -3      934            942
##  4  2013     5     1      602            602         0      657            710
##  5  2013     5     1      603            610        -7      844            906
##  6  2013     5     1      621            627        -6      834            900
##  7  2013     5     1      624            630        -6      736            747
##  8  2013     5     1      624            630        -6      854            906
##  9  2013     5     1      627            630        -3      900            916
## 10  2013     5     1      639            645        -6      838            853
## # ... with 4,566 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <dbl>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

2.4.B) Primeiro, temos que calcular o tempo de atraso total de cada voô, ao somar os tempos de atraso no momento de partida (`dep_delay`) e no momento de chegada (`arr_delay`). Com isso, podemos apenas utilizar o operador > ("maior que") para comparar o tempo de atraso total de cada voô com o resultado da função `mean()`. Como podemos ver abaixo, 95.685 voôs obtiveram um atraso acima da média.

```
library(nycflights13)
### Primeiro, vamos calcular o atraso total
### de cada voô
atraso_total <- flights$dep_delay + flights$arr_delay

teste <- atraso_total > mean(atraso_total, na.rm = TRUE)

flights[teste, ]
```

```
## # A tibble: 95,685 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <dbl>          <dbl>     <dbl>    <dbl>          <dbl>
##  1  2013     1     1      533            529         4      850            830
##  2  2013     1     1      542            540         2      923            850
##  3  2013     1     1      559            600        -1      941            910
##  4  2013     1     1      608            600         8      807            735
##  5  2013     1     1      611            600        11      945            931
##  6  2013     1     1      624            630        -6      909            840
```

```
##  7  2013     1     1      628            630        -2     1016            947
##  8  2013     1     1      632            608        24      740            728
##  9  2013     1     1      635            635         0     1028            940
## 10  2013     1     1      702            700         2     1058           1014
## # ... with 95,675 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <dbl>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

2.4.C) Perceba que ao todo, foram descritas 4 condições no enunciado da questão. 1) `arr_delay` menor que 2; 2) `dest` igual a `"BOS"`; 3) `month` igual a 1; 4) `sched_dep_time` igual a 600.

```
library(nycflights13)

teste <- (flights$arr_delay < 2 & flights$dest == "BOS") |
  (flights$month == 1 & flights$sched_dep_time == 600)

flights[teste, ]
```

```
## # A tibble: 11,500 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <dbl>          <dbl>     <dbl>    <dbl>          <dbl>
##  1  2013     1     1      554            600        -6      812            837
##  2  2013     1     1      555            600        -5      913            854
##  3  2013     1     1      557            600        -3      709            723
##  4  2013     1     1      557            600        -3      838            846
##  5  2013     1     1      558            600        -2      753            745
##  6  2013     1     1      558            600        -2      849            851
##  7  2013     1     1      558            600        -2      853            856
##  8  2013     1     1      558            600        -2      924            917
##  9  2013     1     1      558            600        -2      923            937
## 10  2013     1     1      559            600        -1      941            910
## # ... with 11,490 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <dbl>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

```
### Ou de forma um pouco mais organizada:
###
condicao1 <- flights$arr_delay < 2 & flights$dest == "BOS"
condicao2 <- flights$month == 1 & flights$sched_dep_time == 600

teste <- condicao1 | condicao2

flights[teste, ]
```

Referências Bibliográficas

FRIEDL, J. E. F. *Mastering Regular Expressions*. 3. ed. Sebastopol, CA: O' Reilly Media, Inc., 2006. ISBN 0-596-52812-4.

R CORE TEAM. *R Language Definition*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.

Capítulo 3 - Importando e exportando dados com o R

Exercício 1

3.1.A) Primeiro, sempre comece identificando o caractere especial que está separando cada coluna nesse arquivo. No caso do objeto `t` abaixo, esse caractere especial é o til (~). Com isso, podemos utilizar o argumento `delim`, da função `read_delim()` para termos uma primeira leitura do arquivo, como demonstrado abaixo:

```
t <- "
ID~Valor/Grupo~Unidade
1~2,5488/Marketing~Kg
2~4,0101/Análise~Kg
3~1097/Vendas~g
4~12,76/Logísitica~Kg"

t <- iconv(t, from = "Latin1", to = "UTF-8")

readr::read_delim(t, delim = "~")

## Rows: 4 Columns: 3

## -- Column specification --------------------------------------------------------
## Delimiter: "~"
## chr (2): Valor/Grupo, Unidade
## dbl (1): ID

##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 4 x 3
##      ID `Valor/Grupo`    Unidade
##   <dbl> <chr>            <chr>
## 1     1 2,5488/Marketing Kg
## 2     2 4,0101/Análise   Kg
## 3     3 1097/Vendas      g
## 4     4 12,76/Logísitica Kg
```

3.1.B) Ao observarmos com cuidado os resultados apresentados pela questão, podemos identificar que a importação de ambos os arquivos (pac1 e pac2) apresentam erros. Focando primeiramente em pac1, perceba que os valores presentes em todas as colunas numéricas (Produção, Receita e Gasto em P&D) estão muito altos. Esse erro ocorre, devido ao padrão empregado pela função read_delim(). Lembre-se que grande parte das funções do pacote readr seguem o padrão americano, que utiliza o ponto como o separador decimal, e a vírgula, como separador de milhares.

```
pac1 <- "Setor;Produção;Receita;Gasto em P&D
Produtos alimentícios;10828,37;199907,55;3358,36
Bebidas;759,53;28093,21;
Produtos do fumo;69,99;8863,5;121,35
Produtos têxteis;4153,97;25804,16;746,83
Produtos de madeira;5088,78;15320,69;279,54
Celulose e outras pastas;26,95;4245,19;216,7
Refino de petróleo;75,48;114316,31;1550,73
Produtos químicos;3179,52;133582,8;2914,09
Produtos farmacêuticos;621,82;24972,07;1038,73"

pac1 <- iconv(pac1, from = "Latin1", to = "UTF-8")

readr::read_delim(pac1, delim = ";")
```

```
## Rows: 9 Columns: 4
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ";"
## chr (1): Setor
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 9 x 4
##   Setor                    Produção  Receita `Gasto em P&D`
##   <chr>                       <dbl>    <dbl>          <dbl>
## 1 Produtos alimentícios     1082837 19990755         335836
## 2 Bebidas                     75953  2809321             NA
## 3 Produtos do fumo             6999    88635          12135
## 4 Produtos têxteis           415397  2580416          74683
## 5 Produtos de madeira        508878  1532069          27954
## 6 Celulose e outras pastas     2695   424519           2167
## 7 Refino de petróleo           7548 11431631         155073
```

```
## 8 Produtos químicos          317952  1335828         291409
## 9 Produtos farmacêuticos      62182  2497207         103873
```

Portanto, ao ler o número `10828,37`, a função `read_delim()` entende que esse valor corresponde ao número 1.082.837. Por esse motivo, precisamos sobrepor esse padrão, ao descrevermos explicitamente à função `read_delim()`, o padrão utilizado pelos valores numéricos presentes no arquivo `pac1`. Lembre-se que uma descrição desse tipo é fornecida dentro da função `locale()`, mais especificamente, no argumento `locale` da função que você está utilizando para a importação.

```
readr::read_delim(
  pac1, delim = ";",
  locale = locale(grouping_mark = ".", decimal_mark = ",")
)
```

```
## Rows: 9 Columns: 4
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ";"
## chr (1): Setor
## dbl (3): Produção, Receita, Gasto em P&D
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 9 x 4
##   Setor                   Produção Receita `Gasto em P&D`
##   <chr>                      <dbl>   <dbl>          <dbl>
## 1 Produtos alimentícios     10828.  199908.          3358.
## 2 Bebidas                     760.   28093.            NA
## 3 Produtos do fumo             70.0   8864.           121.
## 4 Produtos têxteis           4154.   25804.           747.
## 5 Produtos de madeira        5089.   15321.           280.
## 6 Celulose e outras pastas     27.0   4245.           217.
## 7 Refino de petróleo           75.5 114316.          1551.
## 8 Produtos químicos          3180.  133583.          2914.
## 9 Produtos farmacêuticos      622.   24972.          1039.
```

Por outro lado, podemos perceber que o arquivo `pac2` enfrenta o mesmo problema de `pac1`. Pois valores como `18.828,37` e `69,99`, foram interpretados pela função `read_delim()` como os números 18,828 e 6.999, respectivamente.

```
pac2 <- "Setor;Produção;Receita;Gasto em P&D
Produtos alimentícios;10.828,37;199907,55;3358,36
Bebidas;759,53;28093,21;x
Produtos do fumo;69,99;8863,5;121,35
Produtos têxteis;4.153,97;25804,16;746,83
Produtos de madeira;5.088,78;15320,69;279,54
Celulose e outras pastas;26,95;4245,19;216,7
Refino de petróleo;75,48;114316,31;1550,73
Produtos químicos;3.179,52;133582,8;2914,09
```

```
Produtos farmacêuticos;621,82;24972,07;1038,73"

pac2 <- iconv(pac2, from = "Latin1", to = "UTF-8")

readr::read_delim(pac2, delim = ";")
```

```
## Rows: 9 Columns: 4
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ";"
## chr (2): Setor, Gasto em P&D
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 9 x 4
##   Setor                    Produção  Receita `Gasto em P&D`
##   <chr>                       <dbl>    <dbl> <chr>
## 1 Produtos alimentícios       10.8  19990755 3358,36
## 2 Bebidas                  75953     2809321 x
## 3 Produtos do fumo          6999       88635 121,35
## 4 Produtos têxteis             4.15  2580416 746,83
## 5 Produtos de madeira          5.09  1532069 279,54
## 6 Celulose e outras pastas  2695      424519 216,7
## 7 Refino de petróleo        7548    11431631 1550,73
## 8 Produtos químicos            3.18  1335828 2914,09
## 9 Produtos farmacêuticos   62182     2497207 1038,73
```

Para corrigir esse problema, utilizamos novamente a função `locale()`. Entretanto, um outro problema ainda persiste no arquivo `pac2`. Pois o tipo `character` foi aplicado sobre a coluna `Gasto em P&D`, a qual é claramente uma coluna numérica.

```
readr::read_delim(
  pac2, delim = ";",
  locale = locale(grouping_mark = ".", decimal_mark = ",")
)
```

```
## Rows: 9 Columns: 4
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ";"
## chr (2): Setor, Gasto em P&D
## dbl (1): Receita
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 9 x 4
##   Setor                    Produção Receita `Gasto em P&D`
##   <chr>                       <dbl>   <dbl> <chr>
## 1 Produtos alimentícios     10828.  199908. 3358,36
```

```
## 2 Bebidas                    760.   28093. x
## 3 Produtos do fumo            70.0   8864. 121,35
## 4 Produtos têxteis          4154.   25804. 746,83
## 5 Produtos de madeira       5089.   15321. 279,54
## 6 Celulose e outras pastas    27.0   4245. 216,7
## 7 Refino de petróleo          75.5 114316. 1550,73
## 8 Produtos químicos         3180.  133583. 2914,09
## 9 Produtos farmacêuticos     622.   24972. 1038,73
```

Tal erro, ocorre pelo simples fato de que a segunda linha dessa coluna é preenchida por um "x". Ao encontrar esse "x", a função `read_delim()` opta pelo tipo de dado mais flexível possível (o tipo `character`). Dessa maneira, precisamos apenas afirmar à função `read_delim()`, que essa coluna deve ser interpretada por um tipo numérico, como o tipo `double`.

```
readr::read_delim(
  pac2, delim = ";",
  locale = locale(grouping_mark = ".", decimal_mark = ","),
  col_types = cols(
    .default = col_number(), Setor = col_character()
  )
)
```

```
## Warning: One or more parsing issues, see `problems()` for details
```

```
## # A tibble: 9 x 4
##   Setor                   Produção Receita `Gasto em P&D`
##   <chr>                      <dbl>   <dbl>          <dbl>
## 1 Produtos alimentícios     10828.  199908.          3358.
## 2 Bebidas                     760.   28093.            NA
## 3 Produtos do fumo             70.0   8864.           121.
## 4 Produtos têxteis           4154.   25804.           747.
## 5 Produtos de madeira        5089.   15321.           280.
## 6 Celulose e outras pastas     27.0   4245.           217.
## 7 Refino de petróleo           75.5 114316.          1551.
## 8 Produtos químicos          3180.  133583.          2914.
## 9 Produtos farmacêuticos      622.   24972.          1039.
```

3.1.C) Quando erros desse tipo ocorrem, é interessante que você olhe para dentro do arquivo, ou seja, abra as primeiras milhares de linhas do arquivo, e tente identificar algum fator que possa estar causando este erro. Como o arquivo `challenge.csv` é relativamente pequeno, você pode abrí-lo em uma janela de seu RStudio por meio dos comandos abaixo:

```
file.edit(readr_example("challenge.csv"))
```

Ao navegar em direção ao final do arquivo, você vai perceber que os dados mudam drasticamente de formato a partir da linha 1001 do arquivo (veja abaixo, um retrato dessa porção do arquivo).

```
998   1843,NA
999   1687,NA
1000  4569,NA
```

```
1001  4548,NA
1002  0.23837975086644292,2015-01-16
1003  0.41167997173033655,2018-05-18
1004  0.7460716762579978,2015-09-05
1005  0.723450553836301,2012-11-28
1006  0.614524137461558,2020-01-13
```

Logo, podemos inferir que o problema gerado na importação, se trata novamente de um chute errado da função `read_csv()`. Pois a função interpretou que as duas colunas do arquivo, pertencem aos tipos `double` e `logical`, respectivamente, sendo que a segunda coluna é, de forma clara, do tipo `Date`, ao observarmos a sua porção à frente da linha 1001.

Lembre-se que, as funções do pacote `readr` vão, por definição, utilizar as 1000 primeiras linhas do arquivo para advinhar o tipo de dado contido em cada coluna do arquivo. Devido ao fato de que a mudança drástica nos dados armazenados em `challenge.csv`, ocorre após essas 1000 primeiras linhas, a função `read_csv()` acaba não percebendo o seu erro. Por esses motivos, precisamos sobrepor essa decisão, ao definirmos explicitamente os tipos desejados para cada coluna no argumento `col_types`.

```
read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_double(),
    y = col_date()
  )
)
```

```
## # A tibble: 2,000 x 2
##        x y
##    <dbl> <date>
##  1   404 NA
##  2  4172 NA
##  3  3004 NA
##  4   787 NA
##  5    37 NA
##  6  2332 NA
##  7  2489 NA
##  8  1449 NA
##  9  3665 NA
## 10  3863 NA
## # ... with 1,990 more rows
```

3.1.D) Novamente, sempre comece identificando o caractere especial que está separando cada coluna nesse arquivo. No caso do objeto `t`, esse caractere especial é o asterisco (`*`). Com isso, podemos utilizar o argumento `delim`, da função `read_delim()` para termos uma primeira leitura do arquivo, como demonstrado abaixo:

```
t <- "Data_execução*Unidades*Valor_compra
20/01/2020*21*R$ 3049,50
23/01/2020*502*R$ 1289,03
```

```
25/01/2020*90*R$ 678,00
02/02/2020*123*R$ 5401
05/02/2020*45*R$ 1450,10
07/02/2020*67*R$ 2320,97
09/02/2020*187*R$ 6231,76"

t <- iconv(t, from = "Latin1", to = "UTF-8")

readr::read_delim(t, delim = "*")

## Rows: 7 Columns: 3

## -- Column specification --------------------------------------------------------
## Delimiter: "*"
## chr (2): Data_execução, Valor_compra
## dbl (1): Unidades

##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

## # A tibble: 7 x 3
##   Data_execução Unidades Valor_compra
##   <chr>            <dbl> <chr>
## 1 20/01/2020          21 R$ 3049,50
## 2 23/01/2020         502 R$ 1289,03
## 3 25/01/2020          90 R$ 678,00
## 4 02/02/2020         123 R$ 5401
## 5 05/02/2020          45 R$ 1450,10
## 6 07/02/2020          67 R$ 2320,97
## 7 09/02/2020         187 R$ 6231,76
```

Ainda assim, há alguns pontos que precisamos melhorar. Primeiro, a coluna `Valor_compra` está sendo atualmente interpretada pelo tipo `character`, sendo que ela claramente guarda valores numéricos, isto é, valores do tipo `double`. O mesmo ocorre com a coluna `Data_execução`, que armazena datas específicas, as quais poderiam ser melhor interpretadas pelo tipo `Date`.

Por isso, precisamos definir explicitamente os tipos dessas colunas à função `read_delim()`, como demonstrado abaixo. Repare que utilizamos `col_number()` sobre a coluna `Valor_compra`, e não, `col_double()`. Pois `col_double()` não seria capaz de ler corretamente essa coluna, dado que os valores numéricos estão acompanhados de informações textuais (`R$`), as quais `col_double()` não é capaz de compreender. Já `col_number()`, busca extrair qualquer valor numérico presente em um *string* e, por isso, acaba ignorando por padrão todas as informações não-numéricas presentes neste mesmo *string*. Após extrair o valor numérico, `col_number()` ainda vai analisar esse valor, e decidir se ele deve ser convertido para o tipo `integer`, ou para o tipo `double`, o que nos dá bastante flexibilidade, e economiza certo trabalho de nossa parte.

```
read_delim(
  t, delim = "*",
```

```
  col_types = cols(
    col_date(format = "%d/%m/%Y"),
    col_integer(),
    col_number()
  ),
  locale = locale(
    decimal_mark = ",", grouping_mark = "."
  )
)
```

```
## # A tibble: 7 x 3
##   Data_execução Unidades Valor_compra
##   <date>           <int>        <dbl>
## 1 2020-01-20          21        3050.
## 2 2020-01-23         502        1289.
## 3 2020-01-25          90         678
## 4 2020-02-02         123        5401
## 5 2020-02-05          45        1450.
## 6 2020-02-07          67        2321.
## 7 2020-02-09         187        6232.
```

Exercício 2

3.2) Primeiro, falando especificamente de planilhas do Excel, a mescla de células é uma ferramenta que pode deixar a sua planilha esteticamente atraente. Porém, tal ferramenta gera sérias anomalias na estrutura de sua tabela. Pois duas células que foram mescladas, são apresentadas a você como uma única célula. Mas no fundo, o Excel armazena os valores presente nessa célula de uma maneira não uniforme (ou em uma estrutura não retangular) ao longo de sua tabela.

No caso do arquivo `emater_icms_solidario.xlsx`, as células mescladas se encontram no cabeçalho da tabela. Com isso, essas células mescladas nos impedem de importar diretamente da planilha, os nomes de cada coluna da tabela. Por isso, é mais fácil simplesmente ignorarmos o fato de que cada coluna possui um nome, e tentarmos selecionar apenas a parte da planilha que contém os dados em si, de forma crua. Por essa estratégia, podemos fornecer corretamente os nomes de cada coluna de forma separada, através do argumento `col_names`.

```
### Nomes de cada coluna
nomes <- c(
  "Semestre", "Ano", "Municipio", "Cod_IBGE",
  "Area_2017", "Area_2018", "Area_Media",
  "Pastagens_2006", "Area_Total", "N_pequeno_prod",
  "Extensao_Rural", "PM_Fundo_Rotativo", "PM_Mecanizacao_Agr",
  "PM_Sementes_Mudas", "PM_Calcario_Fertilizante",
  "PM_Apoio_Comercializacao"
)

### Lembre-se que o caminho até o arquivo
### será diferente em sua máquina.
### Pois muito provavelmente você não possui um
### um usuário chamado Pedro.
```

```
readxl::read_excel(
  "C:/Users/Pedro/Downloads/emater_icms_solidario.xlsx",
  range = "A6:P858",
  col_names = nomes
)
```

```
## # A tibble: 853 x 16
##    Semestre      Ano Municipio         Cod_IBGE Area_2017 Area_2018 Area_Media
##    <chr>       <dbl> <chr>             <chr>        <dbl>     <dbl>      <dbl>
##  1 1° semestre  2020 Abadia dos Doura~ 3100104      5192.     3612.      4402.
##  2 1° semestre  2020 Abaeté            3100203      4800      2042       3421
##  3 1° semestre  2020 Abre Campo        3100302      7116      4744       5930
##  4 1° semestre  2020 Acaiaca           3100401       104.      109.       107.
##  5 1° semestre  2020 Açucena           3100500     12525     12028.     12277.
##  6 1° semestre  2020 Água Boa          3100609      5185      4860.      5022.
##  7 1° semestre  2020 Água Comprida     3100708     36750     47020      41885
##  8 1° semestre  2020 Aguanil           3100807      4492.     4728.      4610.
##  9 1° semestre  2020 Águas Formosas    3100906       893       904        898.
## 10 1° semestre  2020 Águas Vermelhas   3101003      3242      3487       3364.
## # ... with 843 more rows, and 8 more variables: Pastagens_2006 <dbl>,
## #   Area_Total <dbl>, N_pequeno_prod <dbl>, Extensao_Rural <dbl>,
## #   PM_Fundo_Rotativo <dbl>, PM_Mecanizacao_Agr <dbl>, PM_Sementes_Mudas <dbl>,
## #   PM_Calcario_Fertilizante <dbl>, PM_Apoio_Comercializacao <dbl>
```

Referências Bibliográficas

FRIEDL, J. E. F. *Mastering Regular Expressions*. 3. ed. Sebastopol, CA: O' Reilly Media, Inc., 2006. ISBN 0-596-52812-4.

R CORE TEAM. *R Language Definition*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.

Capítulo 4 - Transformando dados com `dplyr`

Exercício 1

4.1.A) Em resumo, os comandos abaixo calculam o número de linhas que descrevem um personagem masculino que possui olhos vermelhos. Primeiro, `count()` calcula o número de linhas por sexo e por cada coloração do olho. Em seguida, a função `filter()` seleciona apenas as linhas que dizem respeito a personagens masculinos e que possuem olhos vermelhos.

```
starwars %>%
  count(sex, eye_color) %>%
  filter(sex == "male", eye_color == "red")
```

```
## # A tibble: 1 x 3
##   sex   eye_color     n
##   <chr> <chr>     <int>
## 1 male  red           2
```

4.1.B) Em resumo, os comandos abaixo calculam o peso médio de cada sexo descrito na tabela `starwars`. Primeiro, a função `select()` vai selecionar todas as colunas da tabela `starwars`, **exceto as colunas contidas no vetor vec**. Segundo, `group_by()` vai agrupar a base de acordo com os valores dispostos na coluna `sex`. Terceiro, `summarise()` vai tratar de calcular o peso médio dentro de cada grupo da coluna `sex`. Em outras palavras, `summarise()` vai separar as linhas da tabela de acordo com os grupos da coluna `sex` e, em seguida, vai aplicar a função `mean()` sobre a coluna `mass` de cada um desses grupos.

```
vec <- c("species", "homeworld", "films", "vehicles", "starships")

starwars %>%
  select(-all_of(vec)) %>%
```

```
  group_by(sex) %>%
  summarise(peso_medio = mean(mass, na.rm = TRUE))
```

```
## # A tibble: 5 x 2
##   sex            peso_medio
##   <chr>               <dbl>
## 1 female               54.7
## 2 hermaphroditic     1358
## 3 male                 81.0
## 4 none                 69.8
## 5 <NA>                 48
```

4.1.C) O código abaixo aplica os seguintes passos sobre a tabela `mpg`: primeiro, `mutate()` adiciona uma coluna chamada `pais_origem`, onde cada modelo de carro descrito na tabela é categorizado de acoro com o país de origem do fabricante deste modelo; em seguida, `count()` contabiliza a quantidade de modelos que pertencem a cada país; por último, um novo `mutate()` é aplicado com o objetivo de calcular a proporção de cada país em relação ao total de modelos descritos na tabela.

```
mpg %>%
  mutate(
    pais_origem = case_when(
      manufacturer %in% c("audi", "volkswagen") ~ "Alemanha",
      manufacturer %in% c("nissan", "honda",
                          "subaru", "toyota") ~ "Japão",
      manufacturer == "hyundai" ~ "Coréia do Sul",
      manufacturer == "land rover" ~ "Inglaterra",
      manufacturer %in% c("dodge", "jeep",
                          "chevrolet", "ford",
                          "lincoln", "pontiac",
                          "mercury") ~ "EUA"
    )
  ) %>%
  count(pais_origem) %>%
  mutate(
    prop = ( n * 100 ) / sum(n)
  )
```

```
## # A tibble: 5 x 3
##   pais_origem       n  prop
##   <chr>         <int> <dbl>
## 1 Alemanha         45 19.2
## 2 Coréia do Sul    14  5.98
## 3 EUA             101 43.2
## 4 Inglaterra        4  1.71
## 5 Japão            70 29.9
```

Exercício 2

4.2.A) Primeiro, identifique as principais colunas que são de seu interesse para responder a pergunta estipulada na questão. Para responder a pergunta, precisamos medir em quanto os

preços cobrados por cada universidade aumentou e, para isso, não precisaremos das colunas `net_cost`, `income_lvl` e `campus`. Por isso, podemos rapidamente eliminar essas colunas com um `select()`.

```
custos <- dados %>%
  select(-net_cost, -income_lvl, -campus)

custos
```

```
## # A tibble: 209,012 x 4
##    name                              state total_price  year
##    <chr>                             <chr>       <dbl> <dbl>
##  1 Piedmont International University NC          20174  2016
##  2 Piedmont International University NC          20174  2016
##  3 Piedmont International University NC          20174  2016
##  4 Piedmont International University NC          20174  2016
##  5 Piedmont International University NC          20514  2017
##  6 Piedmont International University NC          20514  2017
##  7 Piedmont International University NC          20514  2017
##  8 Piedmont International University NC          20514  2017
##  9 Piedmont International University NC          20514  2017
## 10 Piedmont International University NC          20829  2018
## # ... with 209,002 more rows
```

Agora, temos um problema importante a ser analisado: é possível que haja observações repetidas? Ou, será que há várias linhas descrevendo uma mesma universidade em um mesmo ano? As primeiras linhas da tabela acima já nos mostram que sim, há repetição de observações ao longo da base. Para corrigir essa repetição podemos aplicar a função `distinct()` sobre a base.

```
custos <- custos %>%
  distinct()

custos
```

```
## # A tibble: 40,991 x 4
##    name                              state total_price  year
##    <chr>                             <chr>       <dbl> <dbl>
##  1 Piedmont International University NC          20174  2016
##  2 Piedmont International University NC          20514  2017
##  3 Piedmont International University NC          20829  2018
##  4 Piedmont International University NC          23000  2016
##  5 Piedmont International University NC          26430  2017
##  6 Piedmont International University NC          26870  2018
##  7 Kaplan University-Milwaukee       WI          22413  2017
##  8 Kaplan University-Milwaukee       WI          22492  2018
##  9 Kaplan University-Indianapolis    IN          22413  2017
## 10 Kaplan University-Indianapolis    IN          22492  2018
## # ... with 40,981 more rows
```

Lembre-se que, mesmo após aplicarmos `distinct()` sobre a base, pode haver dois `total_price`'s para uma mesma universidade em um mesmo ano. Ou seja, `distinct()` tratou

de eliminar observações repetidas da tabela, isto é, observações que possuem exatamente os mesmos valores em todas as colunas. Sendo assim, podem existir na tabela, dois (ou mais) valores que se referem a uma mesma universidade e a um mesmo ano, mas que possuem valores diferentes na coluna `total_price`. O resultado dos comandos abaixo confirmam essa suspeita:

```
custos %>%
  group_by(name, year) %>%
  count(total_price) %>%
  filter(n > 1)
```

```
## # A tibble: 14 x 4
## # Groups:   name, year [14]
##    name                                  year total_price     n
##    <chr>                                <dbl>       <dbl> <int>
##  1 Academy of Interactive Entertainment  2015       30876     2
##  2 Bryan University                      2010       32572     2
##  3 Bryan University                      2011       40821     2
##  4 Bryan University                      2013       24449     2
##  5 Bryan University                      2014       25446     2
##  6 Bryan University                      2016       25595     2
##  7 Bryan University                      2017       25595     2
##  8 Stevens-Henager College               2011       28414     2
##  9 Stevens-Henager College               2012       29539     2
## 10 Stevens-Henager College               2014       32312     2
## 11 Stevens-Henager College               2015       32440     2
## 12 Stevens-Henager College               2016       33960     2
## 13 Stevens-Henager College               2017       33960     2
## 14 Stevens-Henager College               2018       33489     2
```

São poucas as universidades que possuem mais de um preço para um mesmo ano. Contudo, precisamos que cada ano de cada universidade possua um único preço. Logo, temos que encontrar um método que combine esses dois valores em um só. Calcular a média desses dois valores é uma solução razoável. Repare abaixo, que aplicamos um `group_by()` sobre `custos`, antes do `summarise()`, pois desejamos aplicar a média sobre `total_price`, dentro de cada ano (`year`) de cada universidade (`name`):

```
custos <- custos %>%
  group_by(name, year) %>%
  summarise(mean_price = mean(total_price))
```

```
## `summarise()` has grouped output by 'name'. You can override using the `.groups` argument.
```

```
custos
```

```
## # A tibble: 30,066 x 3
## # Groups:   name [3,664]
##    name                          year mean_price
##    <chr>                        <dbl>      <dbl>
##  1 Aaniiih Nakoda College        2010      17030
##  2 Aaniiih Nakoda College        2011      17030
```

```
##  3 Aaniiih Nakoda College        2012      17030
##  4 Aaniiih Nakoda College        2013      17030
##  5 Aaniiih Nakoda College        2014      17030
##  6 Aaniiih Nakoda College        2015      17030
##  7 Aaniiih Nakoda College        2016      17030
##  8 Aaniiih Nakoda College        2017      17030
##  9 Aaniiih Nakoda College        2018      17030
## 10 Abilene Christian University  2011      38250
## # ... with 30,056 more rows
```

Resolvido esse problema, podemos nos preocupar em calcular a variação anual do preço de cada universidade. As funções `lead()` e `lag()` são muito úteis para compararmos o valor de um determinado ano ao seu par do ano anterior. Porém, para que `lag()` capture corretamente o valor do ano anterior, é fundamental que esses anos estejam organizados dentro de cada universidade, em uma ordem crescente, ao longo de toda a base. Por esse motivo, um `arrange()` é aplicado sobre a base antes do `mutate()`.

```
custos <- custos %>%
  arrange(name, year) %>%
  mutate(
    var_price = mean_price - lag(mean_price)
  ) %>%
  ungroup()

custos
```

```
## # A tibble: 30,066 x 4
##    name                          year mean_price var_price
##    <chr>                        <dbl>      <dbl>     <dbl>
##  1 Aaniiih Nakoda College        2010      17030        NA
##  2 Aaniiih Nakoda College        2011      17030         0
##  3 Aaniiih Nakoda College        2012      17030         0
##  4 Aaniiih Nakoda College        2013      17030         0
##  5 Aaniiih Nakoda College        2014      17030         0
##  6 Aaniiih Nakoda College        2015      17030         0
##  7 Aaniiih Nakoda College        2016      17030         0
##  8 Aaniiih Nakoda College        2017      17030         0
##  9 Aaniiih Nakoda College        2018      17030         0
## 10 Abilene Christian University  2011      38250        NA
## # ... with 30,056 more rows
```

Com esses valores em mãos, podemos enfim responder à pergunta da questão. Basta reordenarmos a base de acordo com as maiores variações de preço (`var_price`) com `arrange()` e, em seguida, extraírmos as 10 primeiras linhas com `head()`. Com isso, temos que o custo anual do Los Medanos College subiu 95.944 dólares em 2012 (comparado ao valor do ano anterior).

```
custos %>%
  arrange(desc(var_price)) %>%
  head(n = 10)
```

```
## # A tibble: 10 x 4
```

```
##    name                                          year mean_price var_price
##    <chr>                                        <dbl>      <dbl>     <dbl>
##  1 Los Medanos College                           2012     114083     95944
##  2 Webb Institute                                2013      61820     43300
##  3 Jewish Theological Seminary of America        2016      75590     34140
##  4 Santa Barbara Business College-Ventura        2018      57535     31207
##  5 Rosedale Technical College                    2011      52240     21883
##  6 Michigan Career and Technical Institute       2011      28462.    19388.
##  7 Hawaii Medical College                        2013      35918     18951
##  8 St Paul's School of Nursing-Queens            2016      56189     18065
##  9 Phillips School of Nursing at Mount Sinai Be~ 2017      62850     17920
## 10 Trinity International University-Florida      2017      30468     17718
```

4.2.B) Ao filtrarmos especificamente as observações do Los Medanos College, podemos identificar que a variação de mais de 94 mil dólares ocorre entre os valores $18.139 e $114.083. Perceba que os demais preços referentes a essa universidade se encontram entre 18 e 21 mil dólares. Logo, tal variação de mais 94 mil parece muito distoante para o padrão da universidade.

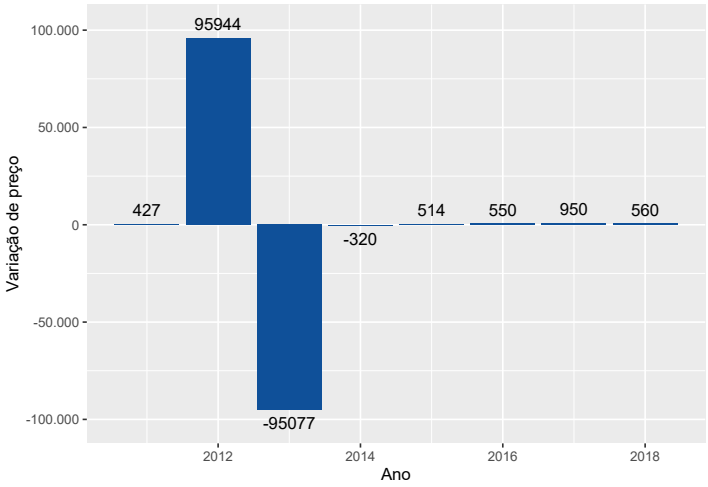
```
los_medanos <- custos %>%
  filter(name == "Los Medanos College", !is.na(var_price))

los_medanos
```

```
## # A tibble: 8 x 4
##   name                year mean_price var_price
##   <chr>              <dbl>      <dbl>     <dbl>
## 1 Los Medanos College  2011      18139       427
## 2 Los Medanos College  2012     114083     95944
## 3 Los Medanos College  2013      19006    -95077
## 4 Los Medanos College  2014      18686      -320
## 5 Los Medanos College  2015      19200       514
## 6 Los Medanos College  2016      19750       550
## 7 Los Medanos College  2017      20700       950
## 8 Los Medanos College  2018      21260       560
```

Expondo essa variação de maneira visual, temos:

```
nudge <- if_else(los_medanos$var_price > 0, 7000, -7000)

los_medanos %>%
  ggplot(
    aes(x = year, y = var_price)
  ) +
  geom_col(
    fill = "#0f5099"
  ) +
  geom_text(
    aes(label = round(var_price, 0)),
    nudge_y = nudge
  ) +
  scale_y_continuous(
```

```
    labels = function(x){
      format(x, digits = 0, big.mark = ".")
    }
  ) +
  labs(
    x = "Ano",
    y = "Variação de preço"
  )
```



Como definimos no enunciado, não temos uma resposta certa ou errada para a questão. O objetivo era apenas que você encontrasse esses dados e questionasse a sua validade. Qual o motivo para uma variação dessa magnitude é a principal questão aqui, e ela levanta altas suspeitas de que esse dado está incorreto, ou que foi alterado de alguma maneira durante o seu processo de coleta. Não sabemos exatamente o que ocorreu com esse dado, mas muito provavelmente há algo de errado com ele.

Exercício 3

4.3.A) Primeiro, como vamos utilizar apenas da coluna 1 até a coluna 11, podemos selecionar essas colunas da tabela com um `select()`:

```
dados <- starwars %>% select(1:11)
```

Em seguida, temos que descobrir o número de valores únicos presentes em cada coluna do tipo `character` de `dados`. Essas colunas são: `name`, `hair_color`, `skin_color`, `eye_color`, `sex`, `gender`, `homeworld` e `species`. Para esse cálculo, poderíamos aplicar as funções `length()` e `unique()` separadamente em cada uma dessas colunas, como mostrado abaixo:

```
name_n <- length(unique(dados$name))
hair_color_n <- length(unique(dados$hair_color))
skin_color_n <- length(unique(dados$skin_color))
    .
    .
```

```
    .
species_n <- length(unique(dados$species))
```

Porém, uma forma muito mais eficiente de realizarmos esse mesmo cálculo, é com o uso da função `across()`, que lhe permite aplicar uma mesma função sobre várias colunas de seu `data.frame`. Detalhe que a função `n_distinct()` pertence ao pacote `dplyr`, sendo apenas uma função equivalente e mais rápida do que a operação `length(unique(x))`. Em outras palavras, as operações `n_distinct(x)` e `length(unique(x))` trazem o mesmo resultado.

```
colunas <- c("name", "hair_color", "skin_color", "eye_color",
             "sex", "gender", "homeworld", "species")

contagens <- dados %>%
  summarise(
    across(all_of(colunas), n_distinct)
  )

contagens
```

```
## # A tibble: 1 x 8
##    name hair_color skin_color eye_color   sex gender homeworld species
##   <int>      <int>      <int>     <int> <int>  <int>     <int>   <int>
## 1    87         13         31        15     5      3        49      38
```

Como resultado, temos um `data.frame` de uma única linha e várias colunas em `contagens`. Temos a capacidade de transformar esse `data.frame` em um vetor, através da função `unlist()`. Dessa maneira, para descobrirmos o maior número de valores únicos em cada coluna, podemos simplesmente aplicar a função `sort()` sobre o vetor resultante de `unlist()`. Repare abaixo, que as colunas `name` e `homeworld` são as colunas que contém mais valores únicos da coluna 1 até a coluna 11 da base, contendo 87 e 49 valores únicos respectivamente.

```
contagens %>%
  unlist() %>%
  sort(decreasing = TRUE)
```

```
##       name  homeworld    species skin_color  eye_color hair_color
##         87         49         38         31         15         13
##        sex     gender
##          5          3
```

Exercício 4

4.4.A) Temos dois caminhos possíveis aqui, os quais se diferenciam apenas pelo posicionamento da função `filter()`. Em resumo, a questão pede por uma média que diz respeito apenas ao atendente Eduardo, logo, vamos precisar, em algum momento, de aplicar um `filter()` com o objetivo de pegar apenas as observações que dizem respeito ao Eduardo. Podemos: 1) filtrar a base inteira para pegar apenas as observações do Eduardo e, em seguida, calcular a média; ou, 2) agrupar a base por cada atendente, calcular a média de cada um e, em seguida, filtrar apenas a média de Eduardo. Ambas as opções chegam ao mesmo resultado de $3462 de receita média por parte de Eduardo.

```
## Opção 1:
transf %>%
  filter(Usuario == "Eduardo") %>%
  summarise(media = mean(Valor))
```

```
## # A tibble: 1 x 1
##   media
##   <dbl>
## 1 3462.
```

```
## Opção 2:
transf %>%
  group_by(Usuario) %>%
  summarise(media = mean(Valor)) %>%
  filter(Usuario == "Eduardo")
```

```
## # A tibble: 1 x 2
##   Usuario media
##   <chr>   <dbl>
## 1 Eduardo 3462.
```

4.4.B) Como podemos observar abaixo, Ana tem maior costume de enviar transferências para o Equador, com um total de 302 transferências destinadas para esse país ao longo da base. Tal cálculo consiste em: 1) filtrar da base todas as linhas que dizem respeito à Ana; 2) contar o número de linhas que dizem respeito a cada país; 3) ordenar a tabela resultante de acordo com a contagem de cada país em ordem crescente; 4) com o resultado ordenado em ordem crescente, os países mais populares ficam nas últimas linhas do resultado, logo, basta extraírmos a última do linha do resultado que teremos o país de destino mais popular de todos.

```
transf %>%
  filter(Usuario == "Ana") %>%
  count(Pais) %>%
  arrange(n) %>%
  tail(n = 1)
```

```
## # A tibble: 1 x 2
##   Pais        n
##   <chr>   <int>
## 1 Equador   302
```

4.4.C) O pacote `dplyr` nos oferece a função `last()`, que é capaz de extrair o último valor de um vetor específico. Perceba que eu ainda forneço a coluna `Data` no argumento `order_by`. Dessa forma, `last()` vai pegar o último valor de um vetor com base na ordem dos valores da coluna `Data`. Porém, como a função `last()` é capaz de extrair o último valor de um vetor, eu preciso utilizar a função `across()` para aplicar `last()` sobre cada uma das colunas da tabela.

```
transf %>%
  group_by(Usuario) %>%
  summarise(across(.fns = last, order_by = Data))
```

```
## # A tibble: 8 x 6
##   Usuario     Data                Valor TransferID Pais     Descricao
```

```
##   <chr>       <dttm>              <dbl>      <dbl> <chr>    <lgl>
## 1 Ana         2018-12-23 22:06:50 16169.  115756250 Alemanha NA
## 2 Armando     2018-12-23 18:54:36 17630.  114268959 Alemanha NA
## 3 Eduardo     2018-12-23 23:49:44 16983.  115188827 Alemanha NA
## 4 Júlio       2018-12-23 13:29:04 15614.  114836120 Alemanha NA
## 5 Júlio Cesar 2018-12-23 20:17:38 16601.  115054244 Alemanha NA
## 6 nathalia    2018-12-23 17:48:23 15256.  115476749 Alemanha NA
## 7 Nathália    2018-12-23 21:12:50 17621.  114970801 Alemanha NA
## 8 Sandra      2018-12-23 17:59:44 16081.  114979909 Alemanha NA
```

Uma outra alternativa é utilizar a função `slice_max()`, que precisa apenas de uma coluna no argumento `order_by`, que corresponde à coluna de referência, ou, a coluna pela qual a função vai determinar o último valor de cada atendente.

```
transf %>%
  group_by(Usuario) %>%
  slice_max(order_by = Data)
```

```
## # A tibble: 8 x 6
## # Groups:   Usuario [8]
##   Data                Usuario      Valor TransferID Pais     Descricao
##   <dttm>              <chr>        <dbl>      <dbl> <chr>    <lgl>
## 1 2018-12-23 22:06:50 Ana         16169.  115756250 Alemanha NA
## 2 2018-12-23 18:54:36 Armando     17630.  114268959 Alemanha NA
## 3 2018-12-23 23:49:44 Eduardo     16983.  115188827 Alemanha NA
## 4 2018-12-23 13:29:04 Júlio       15614.  114836120 Alemanha NA
## 5 2018-12-23 20:17:38 Júlio Cesar 16601.  115054244 Alemanha NA
## 6 2018-12-23 17:48:23 nathalia    15256.  115476749 Alemanha NA
## 7 2018-12-23 21:12:50 Nathália    17621.  114970801 Alemanha NA
## 8 2018-12-23 17:59:44 Sandra      16081.  114979909 Alemanha NA
```

Referências Bibliográficas

FRIEDL, J. E. F. *Mastering Regular Expressions*. 3. ed. Sebastopol, CA: O'Reilly Media, Inc., 2006. ISBN 0-596-52812-4.

R CORE TEAM. *R Language Definition*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.

Capítulo 6 - Introdução a base de dados relacionais no R

Exercício 1

6.1) Primeiro de tudo, precisamos identificar quais são as informações que nós precisamos para calcular o indicador requisitado na questão. Queremos estimar o tempo de trabalho necessário (após a graduação) para cobrir os custos totais do curso de graduação em cada universidade. Portanto, precisamos saber qual o custo total do curso em cada universidade, além do salário estimado dos profissionais que formam nessa respectiva universidade.

Por morarmos no Brasil, universidades americanas nos cobrariam o preço de um estudante não residente do estado de sua sede (*out of state*). O custo total para esse tipo de aluno está na coluna `out_of_state_total` da tabela `tuition_cost`. Já o salário potencial de um aluno que acaba de se formar, é descrito na coluna `early_career_pay` da tabela `salary_potential`.

```
library(tidyverse)

github <- "https://raw.githubusercontent.com/rfordatascience/"
pasta <- "tidytuesday/master/data/2020/2020-03-10/"
cost <- "tuition_cost.csv"
salary <- "salary_potential.csv"

tuition_cost <- read_csv(paste0(github, pasta, cost))
```

```
## Rows: 2973 Columns: 10
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (5): name, state, state_code, type, degree_length
## dbl (5): room_and_board, in_state_tuition, in_state_total, out_of_state...
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
salary_potential <- read_csv(paste0(github, pasta, salary))
```

```
## Rows: 935 Columns: 7
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): name, state_name
## dbl (5): rank, early_career_pay, mid_career_pay, make_world_better_perc...
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Porém, como os custos descritos na tabela `tuition_cost` são anuais, precisamos multiplicar esses custos pelo número de anos presentes na coluna `degree_length`. Para extrairmos o número de cada *string* da coluna `degree_length`, podemos utilizar a função `parse_number()` do pacote `readr`. Em seguida, utilizo a função `colnames()` sobre as duas primeiras colunas da tabela, com o objetivo de traduzir os nomes dessas colunas em uma linguagem mais acessível.

```
custo <- tuition_cost %>%
  select(name, degree_length, out_of_state_total) %>%
  mutate(
    anos = readr::parse_number(degree_length),
    custo_total = anos * out_of_state_total
  ) %>%
  select(-degree_length)
```

```
## Warning: 1 parsing failure.
##  row col expected actual
## 2632  -- a number  Other
```

```
colnames(custo)[1:2] <- c(
  "nome_universidade", "custo_anual"
)
```

```
custo
```

```
## # A tibble: 2,973 x 4
##    nome_universidade                  custo_anual  anos custo_total
##    <chr>                                    <dbl> <dbl>       <dbl>
##  1 Aaniiih Nakoda College                    2380     2        4760
##  2 Abilene Christian University             45200     4      180800
##  3 Abraham Baldwin Agricultural College     21024     2       42048
##  4 Academy College                          17661     2       35322
##  5 Academy of Art University                44458     4      177832
##  6 Adams State University                   29238     4      116952
##  7 Adelphi University                       54690     4      218760
##  8 Adirondack Community College             21595     2       43190
##  9 Adrian College                           48405     4      193620
```

```
## 10 Advanced Technology Institute              13680     2       27360
## # ... with 2,963 more rows
```

Como precisamos apenas das colunas `early_career_pay` e `name` da tabela `salary_potential`, eu aplico um `select()` sobre a tabela. Além disso, eu também renomeio as colunas (para nomes mais fáceis de se interpretar) dessa tabela.

```
salario <- salary_potential %>%
  select(name, early_career_pay)

colnames(salario) <- c(
  "nome_universidade", "salario_inicio_carreira"
)
```

Com essas informações em mãos, podemos utilizar a função `inner_join()` para unir as duas tabelas criadas (`salario` e `custo`) em uma só. Em seguida, precisamos apenas dividir o custo total do curso pela renda esperada para adquirir uma estimativa dos anos de trabalho necessários para repor o investimento aplicado no curso. Como você pode ver abaixo, um aluno formado na Adams State University levaria em torno de 2,63 anos (isto é, $2,63 \times 365 \approx 960$ dias) de trabalho para recompor os valores dispendidos em sua graduação.

```
custo %>%
  inner_join(
    salario,
    by = "nome_universidade"
  ) %>%
  mutate(
    anos_necessarios = custo_total / salario_inicio_carreira
  )
```

```
## # A tibble: 728 x 6
##    nome_universidade        custo_anual  anos custo_total salario_inicio_c~
##    <chr>                          <dbl> <dbl>       <dbl>             <dbl>
##  1 Adams State University         29238     4      116952             44400
##  2 Adventist University of~       19350     4       77400             51600
##  3 Agnes Scott College            53490     4      213960             46000
##  4 Alabama State University       24818     4       99272             39800
##  5 Alaska Pacific Universi~       28130     4      112520             50300
##  6 Albany College of Pharm~       46025     4      184100             81000
##  7 Albertus Magnus College        45260     4      181040             49700
##  8 Albion College                 58155     4      232620             52100
##  9 Alcorn State University        16752     4       67008             40900
## 10 Allen College                  27252     4      109008             51600
## # ... with 718 more rows, and 1 more variable: anos_necessarios <dbl>
```

Exercício 2

6.2.A) Na tabela `consumidores`, temos as colunas `Id_consumidor` e `Id_vendedor` que representam as *keys* nessa tabela. Já na tabela `vendedores`, temos apenas a coluna `Id_vendedor` como *key*.

6.2.B) Lembre-se que, uma *primary key* é uma variável capaz de identificar unicamente cada observação presente em sua tabela. Logo, podemos perceber que a coluna `Id_consumidor` é a *primary key* da tabela `consumidores`. Pois cada observação da tabela, possui um valor diferente na coluna `Id_consumidor`. Já uma *foreign key* é uma coluna que não é capaz de identificar unicamente cada uma das observações de uma tabela. Com isso, podemos chegar à conclusão de que a coluna `Id_vendedor` é a *foreign key* da tabela `consumidores`.

Entenda que as colunas que representam as *keys* de uma tabela, podem mudar de acordo com o contexto. A princípio, as colunas `Id_consumidor` e `Id_vendedor` são as *keys*, pelo simples fato de que elas identificam o objeto foco que está sendo descrito nas tabelas `consumidores` e `vendedores`. Em outras palavras, a tabela `consumidores` apresenta um conjunto de dados sobre **consumidores** e, a coluna `Id_consumidor` identifica unicamente esses consumidores.

A partir do momento em que meu foco de atenção muda, eu posso estar preocupado em identificar unidades, pessoas, grupos, empresas e características diferentes. Por exemplo, se eu estou mais interessado nas **cidades onde o atendimento foi realizado**, é provável que a coluna `Cidade_atendimento` seja uma *key* mais importante que as colunas `Id_vendedor` e `Id_consumidor`.

6.2.C) Após importarmos as tabelas para o R, precisamos aplicar um *join* entre elas, para que possamos relacionar as cidades de atendimento (`Cidade_atendimento`) aos respectivos vendedores (`Nome_vendedor`). Em seguida, podemos aplicar dois `count()`'s seguidos para chegarmos ao resultado desejado.

```
library(tidyverse)

github <- "https://raw.githubusercontent.com/pedropark99/"
pasta <- "Curso-R/master/Dados/"
arquivo1 <- "consumidor.csv"
arquivo2 <- "vendedores.csv"

consumidores <- read_csv2(paste0(github, pasta, arquivo1))
vendedores <- read_csv2(paste0(github, pasta, arquivo2))
```

```
### Resposta:
consumidores %>%
  inner_join(vendedores) %>%
  count(Nome_vendedor, Cidade_atendimento) %>%
  count(Nome_vendedor)
```

```
## Joining, by = "Id_vendedor"
```

```
## # A tibble: 6 x 2
##   Nome_vendedor           n
##   <chr>               <int>
## 1 Jaiminho da Cerveja     1
## 2 Laura Lima              2
## 3 Miguel Anabiguel        1
## 4 Natália Vista           2
```

```
## 5 Pablo Osmar             2
## 6 Paulo Morato            2
```

Exercício 3

6.3) Esse comando de *join* não funciona, pelo fato de que as tabelas `filmes` e `filmes_receita` não possuem colunas de nomes congruentes. Ou seja, a função `left_join()` procura por colunas de mesmo nome entre as tabelas `filmes` e `filmes_receita`, para utilizar como *key* no processo de *join*. Porém, ao não uma coluna de nome correspondente, o *join* acaba falhando.

Para corrigirmos esse problema, podemos: 1) renomear uma das colunas que representa a *key* do *join*, de modo que os seus nomes fiquem iguais; ou 2) dizer explicitamente à `left_join()`, quais são as colunas equivalente entre essas tabelas, através do argumento by da função. Temos a capacidade de realizar a segunda opção de uma maneira bem direta, como demonstrado abaixo:

```
filmes %>%
  left_join(
    filmes_receita,
    by = c("FilmeId" = "Movie_id")
  )
```

```
## # A tibble: 14 x 8
##    FilmeId Titulo             Diretor   Ano DuracaoMinutos Nota_do_publico
##      <dbl> <chr>              <chr>   <dbl>          <dbl>           <dbl>
##  1       1 Toy Story          John L~  1995             81              83
##  2       2 A Bug's Life       John L~  1998             95              72
##  3       3 Toy Story 2        John L~  1999             93              79
##  4       4 Monsters, Inc.     Pete D~  2001             92              81
##  5       5 Finding Nemo       Andrew~  2003            107              82
##  6       6 The Incredibles    Brad B~  2004            116               8
##  7       7 Cars               John L~  2006            117              72
##  8       8 Ratatouille        Brad B~  2007            115               8
##  9       9 WALL-E             Andrew~  2008            104              85
## 10      10 Up                 Pete D~  2009            101              83
## 11      11 Toy Story 3        Lee Un~  2010            103              84
## 12      12 Cars 2             John L~  2011            120              64
## 13      13 Brave              Brenda~  2012            102              72
## 14      14 Monsters University Dan Sc~  2013            110              74
## # ... with 2 more variables: Receita_interna <dbl>,
## #   Receita_internacional <dbl>
```

Por outro lado, a primeira opção envolve o uso da função `colnames()` para renomear a coluna desejada. Após esse passo, a função `left_join()` volta a funcionar normalmente.

```
colnames(filmes_receita)[1] <- "FilmeId"
```

```
filmes %>%
  left_join(
    filmes_receita
  )
```

```
## Joining, by = "FilmeId"
```

```
## # A tibble: 14 x 8
##    FilmeId Titulo              Diretor   Ano DuracaoMinutos Nota_do_publico
##      <dbl> <chr>               <chr>   <dbl>          <dbl>           <dbl>
##  1       1 Toy Story           John L~  1995             81              83
##  2       2 A Bug's Life        John L~  1998             95              72
##  3       3 Toy Story 2         John L~  1999             93              79
##  4       4 Monsters, Inc.      Pete D~  2001             92              81
##  5       5 Finding Nemo        Andrew~  2003            107              82
##  6       6 The Incredibles     Brad B~  2004            116               8
##  7       7 Cars                John L~  2006            117              72
##  8       8 Ratatouille         Brad B~  2007            115               8
##  9       9 WALL-E              Andrew~  2008            104              85
## 10      10 Up                  Pete D~  2009            101              83
## 11      11 Toy Story 3         Lee Un~  2010            103              84
## 12      12 Cars 2              John L~  2011            120              64
## 13      13 Brave               Brenda~  2012            102              72
## 14      14 Monsters University Dan Sc~  2013            110              74
## # ... with 2 more variables: Receita_interna <dbl>,
## #   Receita_internacional <dbl>
```

Referências Bibliográficas

FRIEDL, J. E. F. *Mastering Regular Expressions*. 3. ed. Sebastopol, CA: O' Reilly Media, Inc., 2006. ISBN 0-596-52812-4.

R CORE TEAM. *R Language Definition*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.

Capítulo 7 - *Tidy Data*: uma abordagem para organizar os seus dados

Exercício 1

7.1.A) Primeiro, comece sempre identificando a unidade básica que está sendo tratada na tabela. Como foi definido no enunciado, a tabela `world_bank_pop` lhe apresenta a série histórica de diversos indicadores populacionais de diferentes países. Com isso, podemos chegar à conclusão de que a unidade básica dessa tabela são os países tratados nessa tabela.

Caso houvesse um único indicador nessa tabela, como por exemplo, crescimento anual da população, poderíamos dizer que a unidade básica da tabela seria o crescimento anual da população de cada país. Mas como há pelo menos 4 indicadores para cada país, e cada um deles apresenta uma característica diferente da população desse país, não podemos considerar todos eles como parte da unidade básica da tabela.

```
n_indicadores <- n_distinct(world_bank_pop$indicator)
n_indicadores
```

```
## [1] 4
```

Como você pode ver abaixo, temos 264 países diferentes ao longo da base, logo, se é de nosso desejo respeitar o pressuposto de que "cada observação deve possuir a sua própria linha", a tabela `world_bank_pop` deve acomodar 264 linhas. Pelo fato dessa tabela possuir 1056 linhas, sabemos que esse pressuposto é violado pela tabela.

```
n_paises <- n_distinct(world_bank_pop$country)
n_paises
```

```
## [1] 264
```

```
nrow(world_bank_pop)
```

```
## [1] 1056
```

Além disso, temos a capacidade de rapidamente identificar que o pressuposto de que "cada variável deve possuir a sua própria coluna" também é violado pelo formato da tabela. Pois os diferentes anos da série histórica de cada indicador estão espalhados ao longo de várias colunas, sendo que, esses anos deveriam estar concentrados em uma única coluna, assim como os valores de cada série histórica em si.

7.1.B) Com a resposta do item anterior, podemos chegar aos seguintes passos a serem realizados, com o objetivo de transportar a tabela `world_bank_pop` para o formato *tidy*: 1) trazer os diversos anos para uma única coluna e, o mesmo deve ser feito para os valores de cada série histórica; 2) separar uma coluna diferente para cada um dos indicadores da coluna `indicator`. Podemos realizar esses passos com as funções `pivot_longer()` e `pivot_wider()`.

```
tab_tidy <- world_bank_pop %>%
  pivot_longer(
    cols = matches("[0-9]{4}"),
    names_to = "ano",
    values_to = "valor"
  ) %>%
  pivot_wider(
    id_cols = c("country", "ano"),
    names_from = "indicator",
    values_from = "valor"
  )
```

```
tab_tidy
```

```
## # A tibble: 4,752 x 6
##    country ano   SP.URB.TOTL SP.URB.GROW SP.POP.TOTL SP.POP.GROW
##    <chr>   <chr>       <dbl>       <dbl>       <dbl>       <dbl>
##  1 ABW     2000        42444      1.18         90853      2.06
##  2 ABW     2001        43048      1.41         92898      2.23
##  3 ABW     2002        43670      1.43         94992      2.23
##  4 ABW     2003        44246      1.31         97017      2.11
##  5 ABW     2004        44669      0.951        98737      1.76
##  6 ABW     2005        44889      0.491       100031      1.30
##  7 ABW     2006        44881     -0.0178      100832      0.798
##  8 ABW     2007        44686     -0.435       101220      0.384
##  9 ABW     2008        44375     -0.698       101353      0.131
## 10 ABW     2009        44052     -0.731       101453      0.0986
## # ... with 4,742 more rows
```

7.1.C) Realizar o primeiro passo descrito no enunciado do item é muito simples com a função `filter()`. Porém, ainda assim, o formato no qual a base se encontra, torna o cálculo da variação muito trabalhoso. Porque nós teríamos que calcular a variação entre cada uma das colunas anuais. Como temos 18 anos diferentes em cada série histórica, precisaríamos calcular 17 variações diferentes. Isso significa que teríamos de construir 17 colunas diferentes com `mutate()`

para chegarmos a esses números.

```
pop_total <- world_bank_pop %>%
  filter(
    indicator == "SP.POP.TOTL"
  )

pop_total
```

```
## # A tibble: 264 x 20
##    country indicator      `2000`  `2001` `2002` `2003` `2004` `2005` `2006`
##    <chr>   <chr>           <dbl>   <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
##  1 ABW     SP.POP.TOTL     90853  9.29e4 9.50e4 9.70e4 9.87e4 1.00e5 1.01e5
##  2 AFG     SP.POP.TOTL  20093756  2.10e7 2.20e7 2.31e7 2.41e7 2.51e7 2.59e7
##  3 AGO     SP.POP.TOTL  16440924  1.70e7 1.76e7 1.82e7 1.89e7 1.96e7 2.03e7
##  4 ALB     SP.POP.TOTL   3089027  3.06e6 3.05e6 3.04e6 3.03e6 3.01e6 2.99e6
##  5 AND     SP.POP.TOTL     65390  6.73e4 7.00e4 7.32e4 7.62e4 7.89e4 8.10e4
##  6 ARB     SP.POP.TOTL 283832016  2.90e8 2.96e8 3.02e8 3.09e8 3.16e8 3.24e8
##  7 ARE     SP.POP.TOTL   3154925  3.33e6 3.51e6 3.74e6 4.09e6 4.58e6 5.24e6
##  8 ARG     SP.POP.TOTL  37057452  3.75e7 3.79e7 3.83e7 3.87e7 3.91e7 3.96e7
##  9 ARM     SP.POP.TOTL   3069588  3.05e6 3.03e6 3.02e6 3.00e6 2.98e6 2.96e6
## 10 ASM     SP.POP.TOTL     57521  5.82e4 5.87e4 5.91e4 5.93e4 5.91e4 5.86e4
## # ... with 254 more rows, and 11 more variables: 2007 <dbl>, 2008 <dbl>,
## #   2009 <dbl>, 2010 <dbl>, 2011 <dbl>, 2012 <dbl>, 2013 <dbl>,
## #   2014 <dbl>, 2015 <dbl>, 2016 <dbl>, 2017 <dbl>
```

Por isso, é muito mais fácil calcularmos essa variação ao transportarmos esses anos para uma única coluna e, em seguida, subtraírmos o valor de uma linha específica do valor da linha anterior, com a função `lag()` que apresentamos no Capítulo 4.

```
pop_total <- pop_total %>%
  pivot_longer(
    cols = matches("[0-9]{4}"),
    names_to = "ano",
    values_to = "valor"
  ) %>%
  group_by(country) %>%
  mutate(
    variacao = valor - lag(valor)
  )

pop_total
```

```
## # A tibble: 4,752 x 5
## # Groups:   country [264]
##    country indicator   ano    valor variacao
##    <chr>   <chr>       <chr>  <dbl>    <dbl>
##  1 ABW     SP.POP.TOTL 2000   90853       NA
##  2 ABW     SP.POP.TOTL 2001   92898     2045
##  3 ABW     SP.POP.TOTL 2002   94992     2094
##  4 ABW     SP.POP.TOTL 2003   97017     2025
```

```
##  5 ABW     SP.POP.TOTL 2004   98737     1720
##  6 ABW     SP.POP.TOTL 2005  100031     1294
##  7 ABW     SP.POP.TOTL 2006  100832      801
##  8 ABW     SP.POP.TOTL 2007  101220      388
##  9 ABW     SP.POP.TOTL 2008  101353      133
## 10 ABW     SP.POP.TOTL 2009  101453      100
## # ... with 4,742 more rows
```

Referências Bibliográficas

FRIEDL, J. E. F. *Mastering Regular Expressions*. 3. ed. Sebastopol, CA: O'Reilly Media, Inc., 2006. ISBN 0-596-52812-4.

R CORE TEAM. *R Language Definition*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.

Capítulo 8 - Visualização de dados com `ggplot2`

Exercício 1

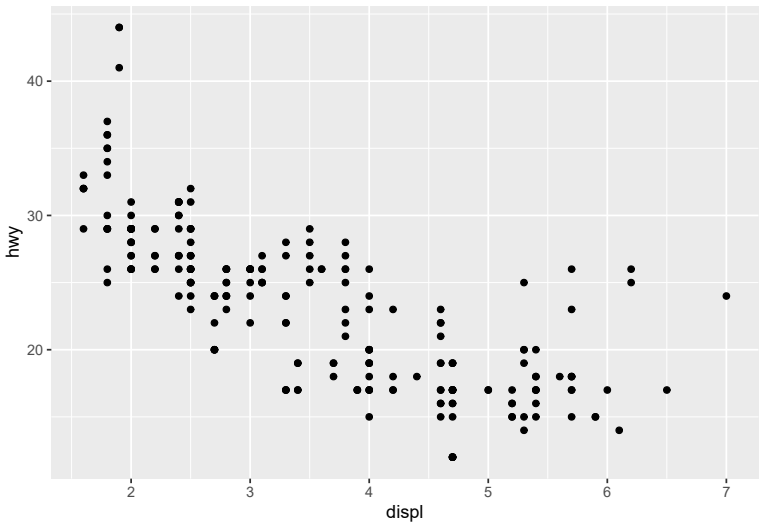
8.1.A) O erro nesse item está no uso do *pipe* para conectar os componentes do gráfico `ggplot`. Lembre-se que, as camadas de um gráfico `ggplot` são adicionadas umas as outras, por meio do operador `+`. Logo, basta substituírmos o *pipe* pelo operador `+` que os comandos voltam a funcionar normalmente.

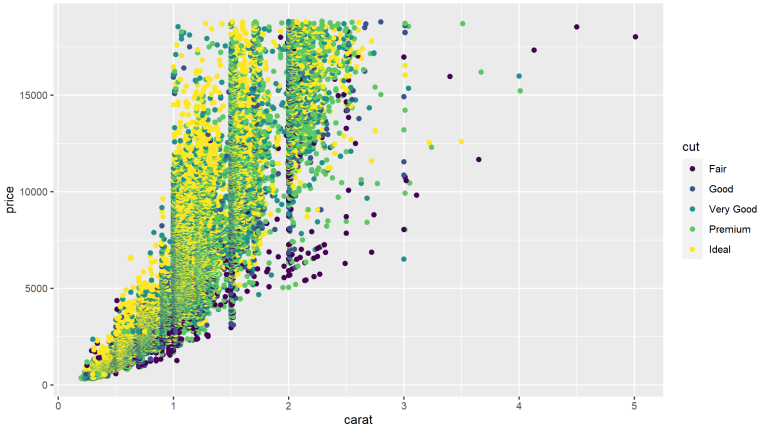
```
ggplot(data = mpg) +
  geom_point(
    aes(x = displ, y = hwy)
  )
```

8.1.B) O erro nesse item está no uso do argumento `fill` ao invés do argumento `color`. Lembre-se que, para colorir geometrias criadas por `geoms` individuais (como pontos - `geom_point()` e linhas - `geom_line()`), utiliza-se o argumento `color`. Já geomtrias que são criadas por `geoms` coletivos (como barras - `geom_bar()` ou `boxplots` - `geom_boxplot()`), utiliza-se o argumento `fill`. Logo, ao substituírmos o argumento `fill` pelo `color`, o resultado desejado é atingido.

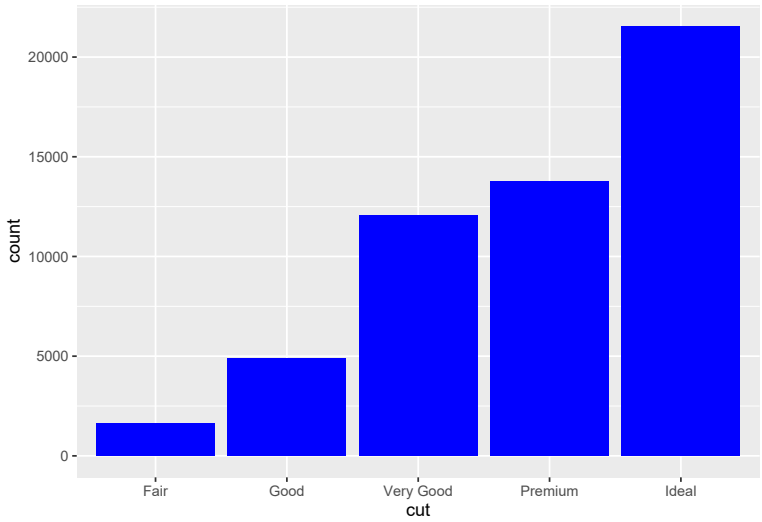
```
ggplot(data = diamonds) +
  geom_point(
    aes(x = carat, y = price, color = cut)
  )
```

8.1.C) Lembre-se que, se desejamos manter uma característica do gráfico (cores, formatos, posição, etc.) constante ao longo de todo o gráfico, nos definimos essa característica **fora da função `aes()`**. Portanto, para atingirmos o resultado desejado, precisamos apenas retirar o termo `color = "blue"` de dentro da função `aes()`.


hwy
40
30
20
2
3
4
5
6
7
displ


price
15000
10000
5000
0
0
1
2
3
4
5
carat
cut
Fair
Good
Very Good
Premium
Ideal

```
ggplot(diamonds) +
  geom_bar(
    aes(x = cut), fill = "blue"
  )
```



Exercício 2

8.2.A) Antes de tudo, é interessante armazenarmos os códigos HEX das cores que formam a bandeira LGBTQ+. Pode ser em um vetor qualquer, como o `vec_colors` abaixo.

```
vec_colors <- c(
  "#a319ff",
  "#1294ff",
  "#19bf45",
  "#ffdc14",
  "#ff6a00",
  "#ff1919"
)
```

Agora, o pacote `ggplot2` nos permite desenhar essa mesma bandeira de diversas formas, mas nessa seção vou mostrar apenas 2 formas intuitivas. Provavelmente, a primeira estratégia que passou pela sua cabeça é simplesmente desenhar 6 faixas empilhadas uma em cima da outra, sendo uma de cada cor presente no vetor `vec_colors` acima.

Como desejamos desejar faixas (ou linhas retas consideravelmente largas), uma opção seria utilizarmos a função `geom_bar()` para construírmos 6 barras de mesma altura. Perceba abaixo, que em `geom_bar()` essas barras são posicionadas (horizontalmente) uma do lado da outra. Porém, utilizo logo em seguida a função `coord_flip()` para inverter o plano cartesiano por completo, isto é, trocar o eixo x pelo eixo y, e, trocar o eixo y pelo eixo x. Dessa forma, as barras que estavam uma do lado da outra, passam a estar uma em cima da outra, ou, são empilhadas verticalmente. Por último, precisamos apenas colorir essas barras com as cores da bandeira. Para isso, basta conectarmos o argumento `fill` ao vetor `vec_colors` e, utilizar a função `scale_fill_identity()` para ler os códigos das cores presentes nesse vetor.

```
dados <- tibble(
    y = 10,
    x = 1:6
  )

dados %>%
  ggplot() +
  geom_bar(
    aes(x = x, y = y, fill = vec_colors),
    position = "dodge",
    stat = "identity",
    width = 1
  ) +
  coord_flip() +
  scale_fill_identity() +
  theme_void()
```

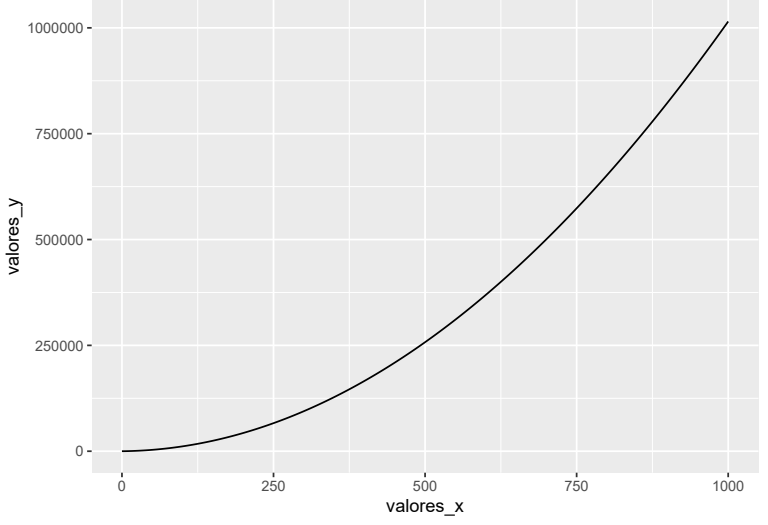
Esse é certamente um método simples e eficaz de desenhar essa bandeira. Entretanto, uma outra forma de desenharmos essa bandeira, seria desenhando 6 retângulos, um sobre o outro. Podemos realizar esse processo por meio de `geom_rect()`. Todos os retângulos possuem a

mesma largura (de 10 unidades), porém, as alturas se reduzem em 1 unidade (6, 5, 4, 3,…) em cada retângulo. Ou seja, o primeiro retângulo (de cor vermelha) é o maior de todos, ou, dito de outra forma, sua altura cobre todas as 6 faixas da bandeira. Já o último retângulo (de cor roxa), é o menor de todos, pois sua altura cobre apenas 1 única faixa.

```
dados <- tibble(
  colors = vec_colors[length(vec_colors):1],
  xmin = 0,
  xmax = 10,
  ymin = 0,
  ymax = 6:1
)


dados %>%
  ggplot() +
  geom_rect(
    aes(xmin = xmin, xmax = xmax,
        ymin = ymin, ymax = ymax,
        fill = colors)
  ) +
  scale_fill_identity() +
  theme_void()
```

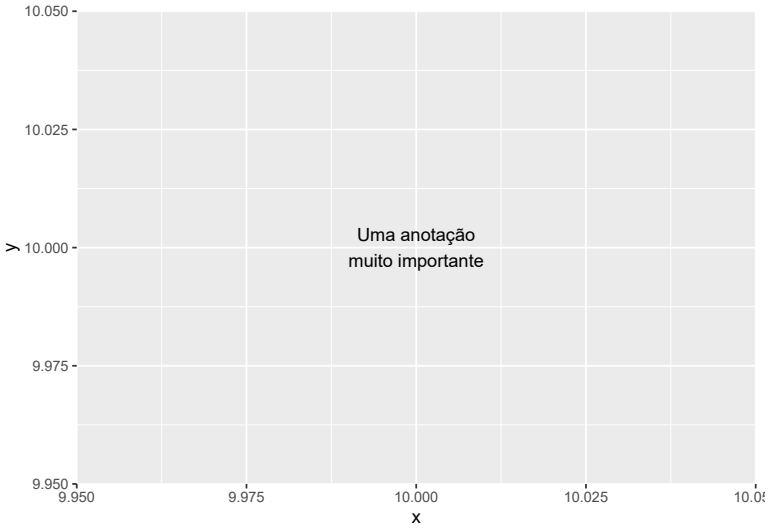
8.2.B) Primeiro, precisamos calcular os valores de y a medida em que x varia de 0 a 1000. Lembre-se que a função é $y = x^2 + 15x + 32$. Com os valores de x e os seus respectivos valores de y calculados, precisamos apenas fornecer esses vetores ao `aes()` de `geom_line()` para atingirmos o resultado esperado.

```
valores_x <- 0:1000
valores_y <- (valores_x ^ 2) + (15 * valores_x) + 32

ggplot() +
  geom_line(
    aes(x = valores_x, y = valores_y)
  )
```



8.2.C) Com os comandos abaixo, você poderia desenhar o texto "Uma anotação muito importante" em seu gráfico. Com isso, precisamos agora incrementar esses comandos com novas camadas do gráfico que vão desenhar as setas desejadas.

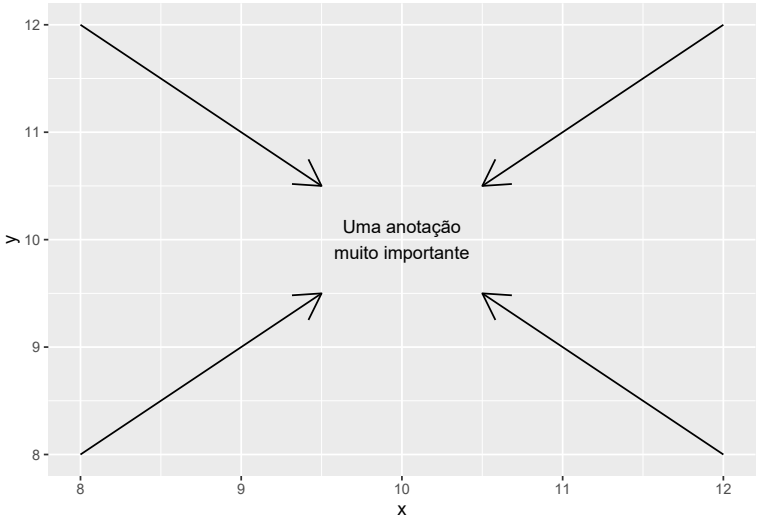
```
anotacao <- "Uma anotação\nmuito importante"

ggplot() +
  geom_text(
    aes(x = 10, y = 10, label = anotacao)
  )
```



A função `geom_segment()` serve para desenhar linhas retas (ou "segmentos") em seu gráfico. Para mais, essa função oferece um argumento `arrow` que permite desenhar a cabeça de uma seta de forma automática em cada segmento. Portanto, preciso apenas definir as coordenadas de cada seta (valores da tabela `setas`) e deixar que o argumento `arrow` desenhe as cabeças de cada seta.

```
setas <- tibble(
  id = 1:4,
  x = c(8, 12, 12, 8),
  y = c(8, 8, 12, 12),
  xend = c(9.5, 10.5, 10.5, 9.5),
  yend = c(9.5, 9.5, 10.5, 10.5)
)

ggplot() +
  geom_text(
    aes(x = 10, y = 10, label = anotacao)
  ) +
  geom_segment(
    aes(x = x, y = y, xend = xend, yend = yend,
        group = id),
```

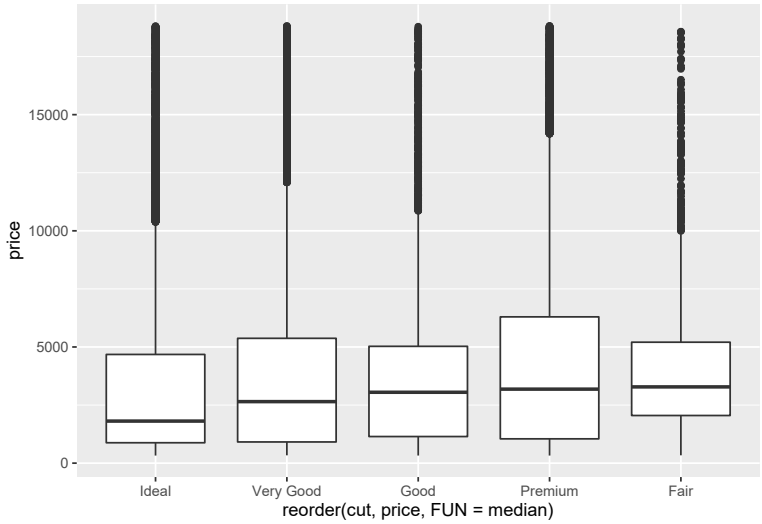
```
    data = setas,
    ### Ao definirmos o argumento arrow
    ### com a função arrow(),
    ### podemos adicionar a cabeça da
    ### seta de forma fácil e automática
    arrow = arrow()
)
```



Exercício 3

8.3) Poderíamos responder essa pergunta de diversas formas e, utilizando diferentes tipos de gráfico. Por exemplo, uma primeira aproximação da resposta, seria empregarmos um simples gráfico de *boxplot*, com a função `geom_boxplot()`. Ainda podemos empregar a função `reorder()` sobre o a variável `cut`, para reordenarmos o eixo x de maneira crescente segundo a mediana de `price`. Dessa maneira, podemos observar facilmente que os diamantes que possuem corte *"Ideal"* são os de menor preço na média. Enquanto isso, também identificamos que os diamentes de pior corte (isto é, os de corte *"Fair"* ou "justo") são, na média, os diamantes mais caros no mercado.

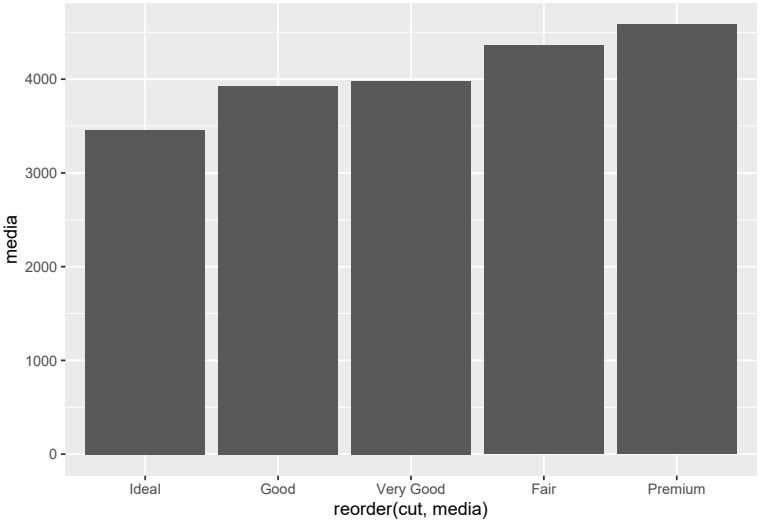
```
diamonds %>%
  ggplot() +
  geom_boxplot(
    aes(
      x = reorder(cut, price, FUN = median),
      y = price
    )
  )
```



Uma alternativa, seria primeiro calcular as estatísticas descritivas e, em seguida, requisitar às funções do `ggplot` que apenas identifiquem esses valores no gráfico. Tendo isso em mente, nós agrupamos a base de acordo com a variável `cut` com a função `group_by()`, calculamos o preço médio de cada grupo com `summarise()` e, por último, utilizamos `geom_col()` para desenhar um gráfico de barras simples, onde a altura de cada barra representa os preços médios calculados por `summarise()`.

```
estatisticas <- diamonds %>%
  group_by(cut) %>%
  summarise(
    media = mean(price, na.rm = TRUE)
```

```
  )

estatisticas %>%
  ggplot() +
  geom_col(
    aes(
      x = reorder(cut, media),
      y = media
    )
  )
```



Referências Bibliográficas

FRIEDL, J. E. F. *Mastering Regular Expressions*. 3. ed. Sebastopol, CA: O' Reilly Media, Inc., 2006. ISBN 0-596-52812-4.

R CORE TEAM. *R Language Definition*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.

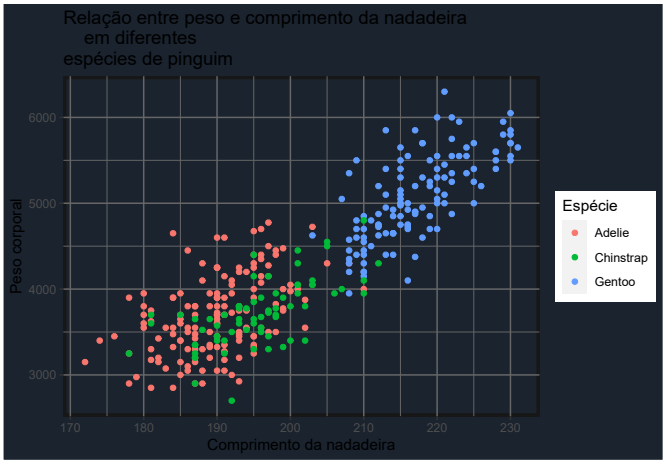
Capítulo 9 - Configurando componentes estéticos do gráfico no `ggplot2`

Exercício 1

9.1.) Primeiro, podemos colorir o plano de fundo do gráfico e do *grid* da seguinte maneira:

```
tema <- theme(
  plot.background = element_rect(fill = "#1a232e"),
  panel.background = element_rect(
    fill = "#1a232e", color = "#171717", size = 2
  ),
  panel.grid = element_line(color = "#666666")
)

plot_exemplo + tema
```

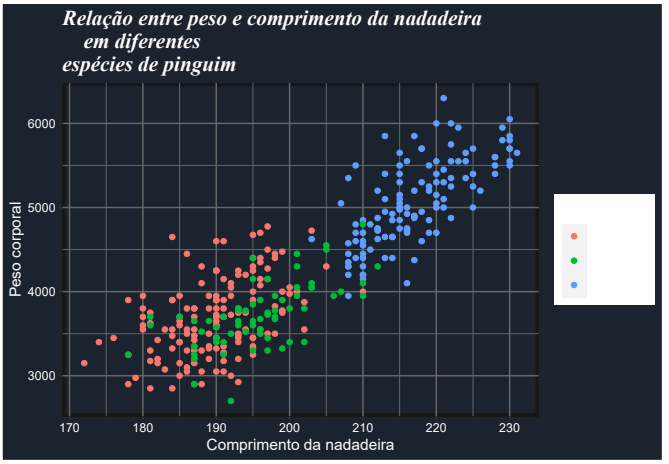
```
## Warning: Removed 2 rows containing missing values (geom_point).
```



Segundo, seria ideal colorirmos todos os textos do gráfico de branco, além de alterarmos a fonte e o estilo empregados sobre o título do gráfico, como demonstrado abaixo.

```
tema <- theme(
  plot.background = element_rect(fill = "#1a232e"),
  panel.background = element_rect(
    fill = "#1a232e", color = "#171717", size = 2
  ),
  panel.grid = element_line(color = "#666666"),
  text = element_text(color = "white"),
  axis.text = element_text(color = "white"),
  plot.title = element_text(
    family = "serif", size = 15,
    face = "bold.italic"
  )
)

plot_exemplo + tema
```

```
## Warning: Removed 2 rows containing missing values (geom_point).
```

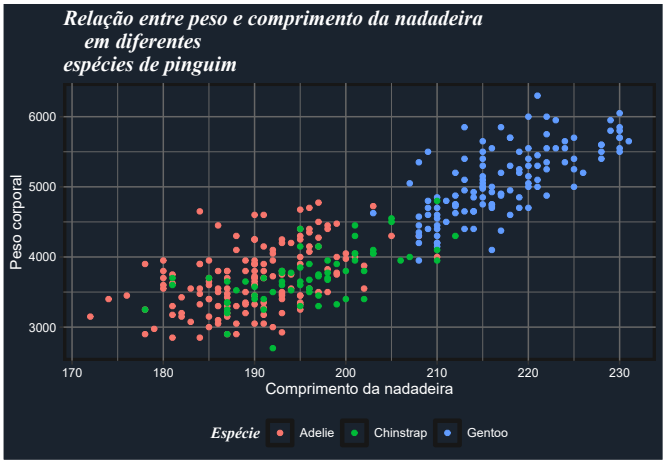


Por último, podemos adicionar as alterações necessárias sobre a legenda do gráfico, por meio dos seguintes comandos:

```
tema <- theme(
  plot.background = element_rect(fill = "#1a232e"),
  panel.background = element_rect(
    fill = "#1a232e", color = "#171717", size = 2
  ),
  panel.grid = element_line(color = "#666666"),
  text = element_text(color = "white"),
  axis.text = element_text(color = "white"),
  plot.title = element_text(
    family = "serif", size = 15,
    face = "bold.italic"
  ),
  legend.position = "bottom",
  legend.title = element_text(
    family = "serif", face = "bold.italic"
  ),
  legend.background = element_rect(fill = "#1a232e"),
  legend.key = element_rect(
    fill = "#1a232e", color = "#171717", size = 1.5
  )
)
```

```
plot_exemplo + tema
```

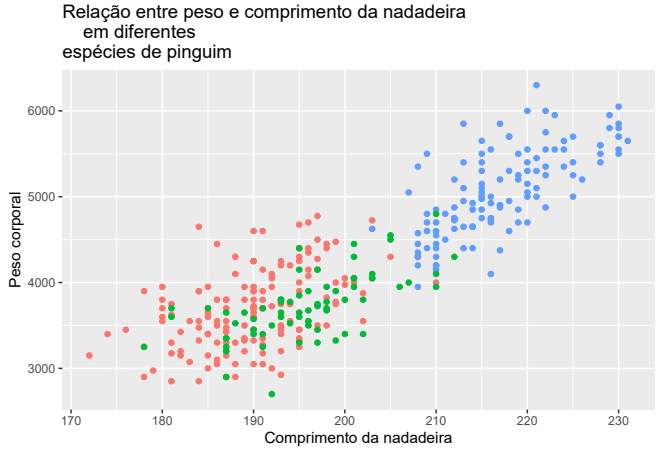
```
## Warning: Removed 2 rows containing missing values (geom_point).
```



9.2.A) Lembre-se que para eliminarmos a legenda de um gráfico, basta configurar o argumento `legend.position` para `"none"`.

```
plot_exemplo +
  theme(legend.position = "none")
```

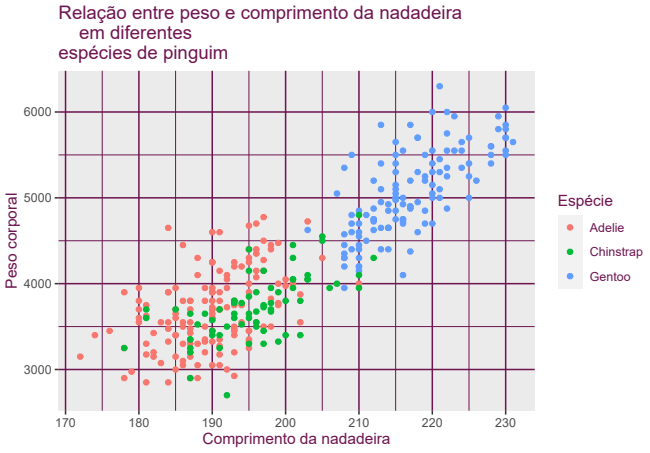
```
## Warning: Removed 2 rows containing missing values (geom_point).
```



9.2.B) O erro ocorre, devido ao fato de que estamos empregando a função `element_*()` errada no elemento `panel.grid`. Lembre-se que o elemento `panel.grid` diz respeito às linhas do *grid*, logo, para alterar esse elemento deve-se utilizar a função `element_line()`, ao invés de `element_rect()`.

```
plot_exemplo +
  theme(
    text = element_text(color = "#6E1450"),
    panel.grid = element_line(color = "#6E1450")
  )
```

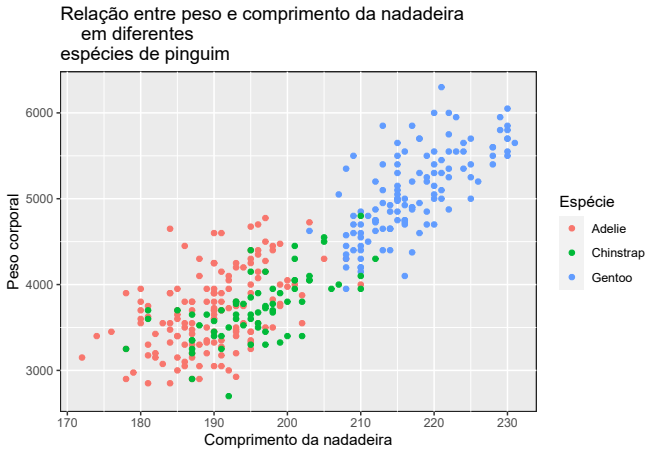
```
## Warning: Removed 2 rows containing missing values (geom_point).
```



9.2.C) Para isso, precisamos definir o argumento `color` da função `element_rect()` no argumento `panel.background` de `theme()`.

```
plot_exemplo +
  theme(panel.background = element_rect(color = "#222222"))
```

```
## Warning: Removed 2 rows containing missing values (geom_point).
```



Referências Bibliográficas

FRIEDL, J. E. F. *Mastering Regular Expressions*. 3. ed. Sebastopol, CA: O' Reilly Media, Inc., 2006. ISBN 0-596-52812-4.

R CORE TEAM. *R Language Definition*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.

Capítulo 10 - Manipulação e transformação de *strings* com `stringr`

Exercício 1

10.1.A) Tal sequência de caracteres pode ser representada pela expressão regular `"[bc]al"`, ou ainda, pela expressão `"bal|cal"`. Ambas as expressões atingem o mesmo resultado:

```
index <- str_which(
  words,
  "[bc]al"
)

words[index]
```

```
## [1] "balance" "ball"    "call"    "local"
```

10.1.B) Perceba que, pelo que foi requisitado no enunciado do item, o primeiro caractere da expressão regular pode ser qualquer um (**exceto a letra "a"**). Logo, podemos começar a expressão regular pela classe de caracteres negativa `"[^a]"`, que lista o caractere não permitido nessa primeira posição. Em seguida, precisamos apenas incluir a letra "c", além do *metacharacter* `$` que representa o final do *string*.

```
str_subset(
  words,
  "[^a]c$"
)
```

```
## [1] "electric" "music"    "politic"  "public"   "specific" "traffic"
```

10.1.C) A primeira sequência citada ("s-p-a-c-e") é bem específica e, por isso, talvez seja mais fácil utilizarmos um *metacharacter* de alternação `|` para separa essa sequência como um caso

especial. Do outro lado do *metacharacter* `|` podemos a expressão `"ess$"`, ou ainda, a expressão `"e(s{2})$"`, ambas são expressões equivalentes.

```
str_subset(
  words,
  "space|e(s{2})$"
)
```

```
##  [1] "address"  "business" "dress"    "express"  "guess"    "less"
##  [7] "press"    "process"  "space"    "unless"
```

```
str_subset(
  words,
  "space|ess$"
)
```

```
##  [1] "address"  "business" "dress"    "express"  "guess"    "less"
##  [7] "press"    "process"  "space"    "unless"
```

10.1.D) Muitas pessoas ao verem uma questão como essa, tendem rapidamente para a expressão `"..."`, ou de outra maneira, `".{3}"`, que representa a mesma coisa. Pois essa expressão significa "um caractere qualquer, imediatamente seguido por um outro caractere qualquer, que por sua vez, é seguido por um outro caractere qualquer". Bem, porque não testamos essa expressão. Como podemos ver pelo resultado abaixo, essa expressão acaba nos retornando praticamente todo o vetor `words` de volta.

```
resultado <- str_subset(
  words,
  "..."
)

print(resultado, max = 30)
```

```
##  [1] "able"      "about"     "absolute"  "accept"    "account"
##  [6] "achieve"   "across"    "act"       "active"    "actual"
## [11] "add"       "address"   "admit"     "advertise" "affect"
## [16] "afford"    "after"     "afternoon" "again"     "against"
## [21] "age"       "agent"     "ago"       "agree"     "air"
## [26] "all"       "allow"     "almost"    "along"     "already"
##  [ reached getOption("max.print") -- omitted 931 entries ]
```

O que está acontecendo? Em momentos como esse, é importante que você compreenda bem o que a sua expressão regular significa. Lembre-se sempre de ler ou interpretar a sua expressão como a "descrição de uma sequência específica de caracteres", e não como uma palavra ou frase específicas.

A expressão `"..."` significa "um caractere qualquer, imediatamente seguido por um outro caractere qualquer, que por sua vez, é seguido por um outro caractere qualquer". Logo, em resumo, essa expressão está procurando por um sequência de 3 caracteres quaisquer. Com isso em mente, podemos chegar à conclusão de que quase todo o vetor `words` é retornado por essa expressão, pelo fato de que quase todas as palavras desse vetor **contém pelo menos 3 caracteres**.

Em outras palavras, a expressão `"..."` não possui qualquer noção de limite, ou quantidade exata a ser procurada. Essa expressão vai procurar por uma sequência de três letras quaisquer, independente de onde ela ocorra, seja em uma palavra que possui exatos 3 caracteres, ou em uma palavra que possui 10 caracteres. Como a maior parte das palavras do vetor `words` possuem 3 ou mais letras, quase todas as palavras desse vetor contém uma sequência de 3 letras quaisquer em algum lugar dentro de si.

Portanto, precisamos adicionar "limites" à expressão `"..."` para que ela possa procurar pelas palavras que desejamos encontrar. Uma opção de limite apropriada, seria o uso de *metacharacters* do tipo âncora (`$` e `^`), formando assim, a expressão `"^...$"`. Dessa forma, estaríamos procurando pela seguinte sequência de caracteres: "o início do texto, imediatamente seguido por três caracteres quaisquer, que por sua vez, são imediatamente seguidos pelo fim desse mesmo texto".

```
str_subset(
  words,
  "^...$"
)
```

```
##   [1] "act" "add" "age" "ago" "air" "all" "and" "any" "arm" "art" "ask"
##  [12] "bad" "bag" "bar" "bed" "bet" "big" "bit" "box" "boy" "bus" "but"
##  [23] "buy" "can" "car" "cat" "cup" "cut" "dad" "day" "die" "dog" "dry"
##  [34] "due" "eat" "egg" "end" "eye" "far" "few" "fit" "fly" "for" "fun"
##  [45] "gas" "get" "god" "guy" "hit" "hot" "how" "job" "key" "kid" "lad"
##  [56] "law" "lay" "leg" "let" "lie" "lot" "low" "man" "may" "mrs" "new"
##  [67] "non" "not" "now" "odd" "off" "old" "one" "out" "own" "pay" "per"
##  [78] "put" "red" "rid" "run" "say" "see" "set" "sex" "she" "sir" "sit"
##  [89] "six" "son" "sun" "tax" "tea" "ten" "the" "tie" "too" "top" "try"
## [100] "two" "use" "war" "way" "wee" "who" "why" "win" "yes" "yet" "you"
```

Exercício 2

10.2.A) Há alguns métodos diferentes que você pode utilizar para resolver essa questão. Em algum ponto, essa solução vai envolver o uso de expressões regulares. Nessa resposta, vamos apresentar o método que envolve o uso de uma única expressão regular, capaz de descrever toda a sequência de caracteres presente em cada *string* do vetor `compras`.

A ideia básica desse método, é utilizar uma expressão regular que possa descrever cada uma das seções dos *strings*, e com a ajuda da função `str_replace()`, inserir um caractere especial entre cada uma dessas seções, criando assim uma espécie de arquivo CSV, onde cada variável está agora separada por um delimitador. Dessa forma, podemos fornecer esse resultado à função `read_csv()`, que fara todo o trabalho de separar essas seções em diferentes colunas de um `data.frame` por nós.

```
library(tidyverse)

github <- "https://raw.githubusercontent.com/pedropark99/"
pasta <- "Curso-R/master/Dados/"
arquivo <- "compras_completo.txt"
```

```
compras <- read_lines(paste0(github, pasta, arquivo))
```

Vamos começar pelos nomes dos consumidores. Uma expressão capaz de descrever essa seção é `"[a-zA-ZÁÉÍÓÚáéíóú]+"`. Uma outra alternativa seria a expressão `"([:lower:]|[:upper:])+"`. Com essas duas expressões somos capazes de encontrar nomes sem acentos (e.g. “Luiz”), assim como os nomes que contém acento (e.g. “Bárbara”) em alguma letra (seja ela maiúscula ou minúscula).

A parte que apresenta o CPF do consumidor é uma das mais fáceis dessa *string*, pois todo brasileiro que já precisou assinar algum contrato, ou comprar um produto pela internet, ou adquirir um documento pessoal, como a sua CNH (carteira de habilitação), conhece muito bem a estrutura tradicional de um CPF brasileiro. Em resumo, CPF’s costumam seguir a estrutura 123.456.789-00. Com isso, podemos chegar à expressão `"[0-9]{3}[.][0-9]{3}[.][0-9]{3}[-][0-9]{2}"` como uma boa candidata para descrever essa seção.

Em seguida, temos a seção que guarda o código de identificação da venda. Essa é uma das partes que mais contribuem para a complexidade da estrutura desses *string* como um todo. Principalmente pelo fato de que esses códigos variam em comprimento, podendo conter de 16 a 24 caracteres diferentes. Porém, ao sabermos todos os caracteres possíveis de aparecer nesses códigos, fica muito mais fácil de descrevermos essa seção. Com a lista de caracteres apresentada na figura 10.8, podemos chegar à expressão `"[a-zA-Z0-9!#$*&]{16,24}"` como uma possível candidata.

Prosseguindo, temos a seção que guarda o código de identificação do produto. Essa seção é a mais fácil de todas, pois todos os códigos possuem 4 dígitos de comprimento. Tendo isso em mente, uma expressão simples como `"[0-9]{4}"` já resolve o nosso problema.

Após essa parte, temos o valor unitário do produto. O único detalhe que devemos estar atentos a essa seção, é que a parte decimal do valor sempre contém dois dígitos (preços não costumam ter 3 dígitos ou mais de centavos), mas a parte inteira do valor pode variar entre 1 e 3 dígitos (ou até mais, pois os preços dos produtos podem variar com o tempo e, como resultado, atingir o 4° dígito). Logo, podemos utilizar a expressão `"[0-9]+[.][0-9]{2}"` para descrever essa seção em específico.

A sexta parte dos *strings* contém datas e horários, os quais formam uma informação também muito familiar a várias pessoas. Lembre-se que no Brasil, datas e horários geralmente seguem a estrutura “Dia/Mês/Ano Hora:Minuto:Segundo”. No caso específico de `compras`, essas datas e horários vem acompanhados de uma descrição do fuso horário empregado. Essa descrição é igual a “-03” em todas as *strings* de `compras`. Com essas informações, você pode identificar a expressão `"[0-9]{2}/[0-9]{2}/[0-9]{4} [0-9]{2}:[0-9]{2}:[0-9]{2} -03"` como uma candidata possível.

Todavia, por algum motivo, os dia abaixo de 10 (ou seja, dias 1, 2, 3, 4, 5, etc.) estão desacompanhados do zero inicial. Ou seja, uma data que deveria estar escrita como “04/05/2020” está na verdade escrita na forma “4/05/2020”. Portanto, uma alternativa para corrigir esse problema é incluir o *metacharacter* da dúvida (?) no zero, formando assim a expressão `"[0-9]?[0-9]/[0-`

`9]{2}/[0-9]{4} [0-9]{2}:[0-9]{2}:[0-9]{2} -03"`. Dessa forma, estamos definindo indiretamente que o dia pode ter 1 ou 2 caracteres, pois o primeiro caractere nessa região é opcional.

Caso você queira ser mais estrito, ou, mais preciso a respeito das datas, você poderia utilizar a expressão `"(3[0-1]|[0-2][0-9])/(0[1-9]|1[1-2])/2[0-9]{3}"`. O mesmo poderia ser feito com o horário, o qual seria melhor representado por uma expressão como `"(0[0-9]|1[0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9])"`. Entranto, se tratando dos *strings* contidos no vetor `compras`, nós não precisamos ser tão específicos assim.

Por último, temos uma seção que armazena a quantidade do produto adquirida pelo consumidor. Essa parte também é bem simples de ser descrita, especialmente se levarmos em conta que a sequência ”unidades”é constante ao longo de todo o vetor `compras`. Para mais, a quantidade adquirida pode variar entre 1 e 2 dígitos. Desse modo, podemos chegar à expressão `"[0-9]{1,2} unidades"` para descrever tal seção.

Com todos os fragmentos de expressões regulares em mãos, podemos começar a unir essas peças, com o objetivo de formarmos uma única grande expressão regular capaz de descrever toda a sequência de caracteres presentes em cada *string* do vetor `compras`. Neste processo, **é fundamental contornar cada fragmento da expressão regular por um par de parênteses**. Pois assim, temos acesso ao mecanismo de *backreferencing* em cada seção do *string*.

```
nome <- "([a-zA-ZÁÉÍÓÚáéíóú]+)"
cpf <- "([0-9]{3}[.][0-9]{3}[.][0-9]{3}[-][0-9]{2})"
idcompra <- "([a-zA-Z0-9!#$*&]{16,24})"
idproduto <- "([0-9]{4})"
preço <- "([0-9]+[.][0-9]{2})"
horario <- "([0-9]?[0-9]/[0-9]{2}/[0-9]{4} [0-9]{2}:[0-9]{2}:[0-9]{2} -03)"
unidades <- "([0-9]{1,2} unidades)"

expressao_completa <- str_c(
  nome, cpf, idcompra,
  idproduto, preço,
  horario, unidades
)

str_trunc(expressao_completa, width = 50, ellipsis = "~")
```

```
## [1] "([a-zA-ZÁÉÍÓÚáéíóú]+)([0-9]{3}[.][0-9]{3}[.][0-9]~"
```

Com a expressão completa formada, podemos adicionar pontos e vírgulas entre cada seção, com o uso de *backreferencing* na função `str_replace()`, como demonstrado abaixo. Após essa modificação, basta fornecermos o texto resultante a qualquer função que seja capaz de ler um arquivo CSV que utiliza o caractere `;` como separador. Nesse caso, eu utilizo a função `read_csv2()`, a qual introduzimos no capítulo 3 deste livro.

```
resultado <- str_replace(
  compras,
  expressao_completa,
  "\\1;\\2;\\3;\\4;\\5;\\6;\\7"
)
```

```
tab <- resultado %>%
  str_c(collapse = "\n") %>%
  read_csv2(col_names = FALSE)
```

10.2.B) Para extrairmos a parte que contém a data e horário da compra, nós podemos utilizar a mesma expressão regular que empregamos na resposta do item anterior. Dessa maneira, podemos usar a função `str_extract()` para extrair de cada *string* do vetor `compras`, todo o pedaço de texto encontrado por essa expressão regular. Em seguida, podemos utilizar a função `str_sub()` para extrair os dois primeiros caracteres (que correspondem à parte do dia em cada data) de cada texto resultante de `str_extract()`.

```
horario <- "([0-9]?[0-9]/[0-9]{2}/[0-9]{4} [0-9]{2}:[0-9]{2}:[0-9]{2} -03)"

parte <- str_extract(
  compras, pattern = horario
)

dias <- as.integer(
  str_sub(parte, end = 2)
)
```

Agora, precisamos apenas fazer um cálculo de frequência sobre os valores presentes no vetor `dias`. Para realizar cálculos desse tipo, estivemos utilizando bastante a função `count()` ao longo desse livro. Porém, essa função trabalha com `data.frame`'s, ao invés de vetores. Por isso, neste caso específico, eu substituo a função `count()` pela função `table()` que é uma alternativa mais adequada para vetores. Perceba pelo resultado abaixo, que, aparentemente, o dia 26 é o dia do mês em que ocorre o maior número de vendas na loja.

```
contagens <- sort(table(dias), decreasing = TRUE)
### Para facilitar a leitura dos resultados
### eu utilizo str_c() para modificar o nome
### de cada elemento desse resultado.
names(contagens) <- str_c(
  "Dia ",
   names(contagens)
)

print(contagens)
```

```
## Dia 26 Dia 12 Dia 14 Dia 20  Dia 3 Dia 13 Dia 19 Dia 29  Dia 7 Dia 24
##     56     49     46     46     42     42     42     41     40     39
##  Dia 4  Dia 9 Dia 10 Dia 27  Dia 1 Dia 21 Dia 22  Dia 8 Dia 25  Dia 6
##     38     38     38     38     37     37     37     36     36     35
## Dia 16 Dia 30 Dia 11 Dia 18  Dia 5 Dia 15 Dia 17  Dia 2 Dia 23 Dia 28
##     35     35     34     33     32     32     32     29     29     28
## Dia 31
##     18
```

10.2.C) O problema principal nessa questão é o fato do CPF estar incluso após o nome do consumidor, o qual varia radicalmente em seu número de caracteres. Em outras palavras, seria muito simples extrairmos os 3 primeiros dígitos do CPF, caso ele fosse antecedido por um número fixo de caracteres, com o uso da função `str_sub()`.

Entretanto, devido ao número variável de caracteres que podem anteceder o CPF do consumidor, nós precisamos utilizar um outro método. Uma alternativa tão simples quanto a primeira, é utilizar a função `str_extract()` com a expressão regular `"[0-9]{3}"`.

```
resultado <- str_extract(compras, "[0-9]{3}")
print(resultado, max = 30)
```

```
##  [1] "390" "944" "395" "322" "475" "031" "528" "890" "571" "339" "753"
## [12] "110" "059" "543" "072" "327" "096" "138" "608" "079" "141" "841"
## [23] "078" "472" "988" "647" "493" "446" "236" "417"
##  [ reached getOption("max.print") -- omitted 1120 entries ]
```

Nesse momento você pode estar confuso, pois a expressão `"[0-9]{3}"` é muito geral, essa expressão representa uma sequência de três dígitos quaisquer e, há diversos locais ao longo de cada *string* que poderiam ser representados por tal expressão. Em outras palavras, porque a expressão `"[0-9]{3}"` extrai especificamente os 3 primeiros dígitos do CPF? Sendo que ela poderia extrair os 3 primeiros dígitos do código de identificação do produto? Ou parte do ano presente na data e horário da compra?

A resposta para essas perguntas se baseia no princípio de que pesquisas por expressões regulares ocorrem da esquerda para a direita em um *string* (**??**). Portanto, a expressão `"[0-9]{3}"` é capaz de extrair os 3 primeiros dígitos de cada CPF, pelo simples fato de que o CPF é o primeiro campo numérico a aparecer no *string*, antes do código de identificação do produto, e antes da data e horário da compra.

Referências Bibliográficas

FRIEDL, J. E. F. *Mastering Regular Expressions*. 3. ed. Sebastopol, CA: O'Reilly Media, Inc., 2006. ISBN 0-596-52812-4.

R CORE TEAM. *R Language Definition*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.

Capítulo 12 - Introdução à variáveis de tempo com `lubridate`

Exercício 1

12.1.A) Alguns valores do vetor `vec` são convertidos para `NA`, pelo simples fato de que eles não representam datas válidas segundo o calendário Gregoriano. Por exemplo, a data `"2020-02-30"` (30 de fevereiro de 2020) não existe em nosso calendário, assim como a data `"2020-09-87"` (87 de setembro de 2020).

```
vec <- c("2020-01-17", "2020-02-21", "2020-02-30",
         "2020-04-12", "2020-13-19", "2020-09-87")


as.Date(vec)
```

```
## [1] "2020-01-17" "2020-02-21" NA           "2020-04-12" NA
## [6] NA
```

Portanto, a menos que você queira utilizar um calendário diferente do Gregoriano, esse comportamento é correto e, provavelmente é exatamente o que você deseja que ocorra com datas inexistentes ou não válidas.

12.1.B) Uma primeira alternativa, seria empregarmos a função `dmy()` do pacote `lubridate`.

```
library(lubridate)
vec <- c("02, 02, 2020", "15, 03, 2020", "21, 04, 2020",
         "19, 09, 2020", "22, 06, 2020", "25, 12, 2020")

dmy(vec)
```

```
## [1] "2020-02-02" "2020-03-15" "2020-04-21" "2020-09-19" "2020-06-22"
## [6] "2020-12-25"
```

Uma segunda alternativa, seria utilizarmos o argumento `format` da função `as.Date()` para definir a estrutura dessa data.

```
as.Date(vec, format = "%d, %m, %Y")
```

```
## [1] "2020-02-02" "2020-03-15" "2020-04-21" "2020-09-19" "2020-06-22"
## [6] "2020-12-25"
```

12.1.C) Este ponto no tempo (meia noite de 01 de janeiro de 1970) foi armazenado como 10800 devido ao fuso horário aplicado pela função `as.POSIXct()`. Lembre-se que, quando não definimos algum fuso horário no argumento `tz`, a função `as.POSIXct()` vai automaticamente utilizar o fuso horário padrão de seu sistema operacional. No meu caso, esse fuso é o de São Paulo (o qual é equivalente ao fuso horário de Brasília).

O fuso horário de Brasília está a 3 desvios negativos do horário UTC, ou, dito de outra forma, está 3 horas atrasado em relação ao horário UTC. Tendo isso em mente, o instante "meia noite de 01 de janeiro de 1970" no horário de Brasília, está a 3 horas (ou a 10.800 segundos) de distância deste mesmo instante no horário UTC. Em outras palavras, o ano de 1970 começou oficialmente no Brasil, depois de 3 horas que ele já tinha começado no mundo como um todo.

Logo, lembre-se que valores do tipo `date-time` são armazenados como o número de segundos desde o instante "meia noite de 01 de janeiro de 1970" **no horário UTC**. Tanto que se eu recriar o objeto `ponto`, dessa vez utilizando o fuso horário UTC, ao retirarmos a sua classe podemos perceber abaixo que este ponto é armazenado pelo valor zero.

```
ponto <- as.POSIXct("1970-01-01 00:00:00", tz = "UTC")
unclass(ponto)
```

```
## [1] 0
## attr(,"tzone")
## [1] "UTC"
```

Exercício 2

12.2) A primeira coisa que chama atenção nos valores do vetor `numero_no_excel` é que todos eles estão muito próximos do número 43881. Com algum tempo de reflexão, você pode acabar chegando a conclusão de que esses valores representam **o número de dias desde a data de origem**. Ou seja, enquanto no R, valores do tipo `date-time` são armazenados em segundos, esses mesmos valores são armazenados em dias no Excel.

```
numero_no_excel <- c(
  43881.1527777778,
  43881.15625,
  43881.159722222226,
  43881.1632060185,
  43881.1666666667
)
```

Por isso, ao fornecermos esses valores para a função `as.POSIXct()` precisamos converter esses valores para segundos (ao multiplicá-los pelo fator $60 \times 60 \times 24$). Como foi destacado no enunciado, o Excel utiliza o dia 30 de dezembro de 1899 como o seu ponto de origem e, por

isso, precisamos fornecer esse dia ao argumento `origin`. Dessa maneira, `as.POSIXct()` vai utilizar a data 30 de dezembro de 1899 (ao invés de 01 de janeiro de 1970) como base para calcular as datas.

Para mais, podemos configurar o argumento `tz` para o fuso UTC, com o objetivo de evitarmos possíveis ajustes automáticos adicionados aos valores, os quais podem ocorrer devido as diferenças entre o fuso horário de seu sistema operacional e o fuso UTC.

```
ajuste <- as.POSIXct(
  numero_no_excel * (60 * 60 * 24),
  origin = "1899-12-30 00:00:00",
  tz = "UTC"
)

print(ajuste)
```

```
## [1] "2020-02-20 03:40:00 UTC" "2020-02-20 03:45:00 UTC"
## [3] "2020-02-20 03:50:00 UTC" "2020-02-20 03:55:00 UTC"
## [5] "2020-02-20 04:00:00 UTC"
```

Referências Bibliográficas

FRIEDL, J. E. F. *Mastering Regular Expressions*. 3. ed. Sebastopol, CA: O' Reilly Media, Inc., 2006. ISBN 0-596-52812-4.

R CORE TEAM. *R Language Definition*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.

Capítulo 14 - Funções

Exercício 1

14.1.A) Para aceitarmos um número variável de argumentos, precisamos utilizar o argumento especial *dot-dot-dot* (. . .). Em seguida, para incluir os argumentos dentro de uma lista, podemos apenas aplicar a função `list()` sobre o argumento especial. Com isso, temos a função `f()` abaixo:

```
f <- function(...){
  return(list(...))
}

### 1 argumento:
f(a = 1)
```

```
## $a
## [1] 1
```

```
### 3 argumentos:
f(a = 1, b = 2, c = 3)
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
```

14.1.B) A área de um círculo (A) é definido como $A = \pi \times r^2$, sendo π o número pi, e r^2, o raio elevado ao quadrado. Vale destacar que, o R já possui um objeto pré-definido chamado `pi`, que contém o número pi. Portanto, uma função que calcule essa equação seria:

```
area_circulo <- function(r){
  pi * (r ^ 2)
}

### A área de um círculo de raio 5
area_circulo(5)
```

```
## [1] 78.53982
```

Exercício 2

14.2.) Existem algumas alternativas para construir tal função. Estou mostrando abaixo, uma dessas alternativas. Primeiro, precisamos pensar no input dessa função que estou dando o nome de `scrabble_score()`.

Essa função vai receber uma palavra, porém, a pontuação do jogo é baseada em letras, logo, precisamos de alguma forma separar as diferentes letras da palavra, em diferentes elementos de um vetor. Para isso, podemos utilizar a função `str_split()` do pacote `stringr`, que mostramos no capítulo 10.

```
scrabble_score <- function(palavra){
  letras <- unlist(stringr::str_split(palavra, ""))
  return(letras)
}

scrabble_score("teste")
```

```
## [1] "t" "e" "s" "t" "e"
```

É possível que o usuário dessa função, acabe misturando letras maiúsculas e minúsculas dentro da palavra fornecida. Como é um pouco trabalhoso lidar com todas as diferentes possibilidades que surgem dessa mistura, seria uma boa ideia incluírmos uma correção sobre o input da função, com o objetivo de sempre transformar os caracteres da palavra para letras minúsculas. Para isso, temos a função `str_to_lower()` que também apresentamos no capítulo 10.

```
scrabble_score <- function(palavra){
  palavra <- stringr::str_to_lower(palavra)
  letras <- unlist(stringr::str_split(palavra, ""))
  return(letras)
}

scrabble_score("TeStE")
```

```
## [1] "t" "e" "s" "t" "e"
```

Agora, precisamos adicionar um sistema que relaciona cada letra ao seu respectivo ponto. Uma forma eficiente de se implementar tal sistema é através de um *lookup vector*. No exemplo abaixo, tal *lookup vector* é o vetor `pontos`.

```
scrabble_score <- function(palavra){
  pontos <- rep.int(c(1:5, 8, 10), c(10, 2, 4, 5, 1, 2, 2))
  names(pontos) <- c(
    "a","e","i","o","u","l","n","r","s","t",
    "d","g",
    "b","c","m","p",
    "f","h","v","w","y",
    "k",
    "j","x",
    "q","z"
  )


  palavra <- stringr::str_to_lower(palavra)
  letras <- unlist(stringr::str_split(palavra, ""))

  total <- unname(pontos[letras])

  return(total)
}

scrabble_score("yamandu")
```

```
## [1] 4 1 3 1 1 2 1
```

Por último, precisamos apenas somar esse vetor de pontos armazenados no objeto `total`, como demonstrado abaixo.

```
scrabble_score <- function(palavra){
  pontos <- rep.int(c(1:5, 8, 10), c(10, 2, 4, 5, 1, 2, 2))
  names(pontos) <- c(
    "a","e","i","o","u","l","n","r","s","t",
    "d","g",
    "b","c","m","p",
    "f","h","v","w","y",
    "k",
    "j","x",
    "q","z"
  )


  palavra <- stringr::str_to_lower(palavra)
  letras <- unlist(stringr::str_split(palavra, ""))

  total <- sum(unname(pontos[letras]), na.rm = TRUE)

  return(total)
}
```

```
scrabble_score("yamandu")
```

```
## [1] 13
```

Para confirmar que essa função está funcionando corretamente, podemos aplicá-la sobre cada elemento do vetor w, e conferir os resultados. Perceba abaixo, que os resultados são idênticos aos elementos do vetor p mostrado na questão.

```
w <- c("isabela", "caderno", "mouse", "elevador",
       "solar", "gaveta", "porta", "eduardo")


vec <- vector("double", length = length(w))
for(i in seq_along(w)){
  vec[i] <- scrabble_score(w[i])
}

print(vec)
```

```
## [1]  9 10  7 12  5 10  7  9
```

Referências Bibliográficas

FRIEDL, J. E. F. *Mastering Regular Expressions*. 3. ed. Sebastopol, CA: O' Reilly Media, Inc., 2006. ISBN 0-596-52812-4.

R CORE TEAM. *R Language Definition*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.

Capítulo 15 - *Loops*

Exercício 1

15.1.A) A resposta curta é: nada! Nada ocorre se executarmos o `for` *loop* abaixo. Tal fato ocorre, pois o objeto `x` é um vetor vazio. Ou seja, ele possui 0 elementos.

```
x <- vector("integer")

for(i in seq_along(x)){
  print(i)
}
```

Na seção "Descrevendo `for` *loops*" nós destacamos que a quantidade de repetições que são executadas por um `for` *loop*, depende diretamente no número de elementos que o objeto contido na definição deste *loop* possui. Como o objeto `x` possui 0 elementos, o `for` *loop* vai executar 0 repetições.

15.1.B) O *loop* abaixo é um *loop* infinito devido ao `if` *statement* que está presente em seu corpo. Primeiro, a condição lógica que rege o while loop abaixo é `x < 10`. Portanto, para que este while loop pare a sua execução em algum momento, o objeto `x` deve conter um valor maior ou igual a 10.

A expressão `x <- x + 1` garante que o valor armazenado nesse objeto `x` aumente a cada repetição do *loop*. Contudo, o `if` *statement* analisa a condição lógica `x %% 5 == 0`. Por causa dessa condição, assim que o objeto `x` atingir o primeiro número múltiplo de 5 (que neste caso é o próprio 5), o `if` *statement* vai resetar o objeto `x` para o seu estado inicial, ao executar a expressão `x <- 1`.

Portanto, todo esse processo se repete. O loop vai aumentando o valor de `x`, porém, assim que

esse objeto atinge o número 5, ele é resetado de volta para o valor 1, e, por isso, o loop continua executando os comandos de seu *body* indefinidamente.

```
x <- 1
while ( x < 10 ) {
  print(x)
  x <- x + 1
  if (x %% 5 == 0) {
    x <- 1
  }
}
```

15.1.C) Lembre-se que, o número de repetições executadas por um `for` *loop*, depende diretamente do número de elementos presentes no objeto que você fornece na definição deste `loop`.

```
df <- data.frame(id = 1:10)
for(name in letters[1:24]){
  df[[name]] <- NA
}
```

No exemplo desta questão, esse objeto é `letters[1:24]`. Portanto, para descobrirmos o número de repetições executadas por este *loop*, podemos executar o comando `length(letters[1:24])`. Perceba pelo resultado abaixo, que são 24 repetições executadas pelo *loop*.

```
length(letters[1:24])
```

```
## [1] 24
```

15.1.D) A expressão `1:length(y)` determina os valores que o iterador (`i`) vai assumir ao longo do *loop* abaixo.

```
y <- vector("integer")
x <- 1:10
for(i in 1:length(y)){
  print(x[i] + 1)
}
```

```
## [1] 2
## numeric(0)
```

Se executarmos essa expressão de maneira separada, podemos descobrir que valores são esses. Porém, perceba abaixo, que o resultado dessa expressão é um vetor contendo os números 1 e 0. O zero em questão, é resultado de `length(y)`. Isso significa que o vetor `y` é um vetor vazio (em outras palavras, ele possui 0 elementos).

```
1:length(y)
```

```
## [1] 1 0
```

Por causa disso, o `for` *loop* acima acaba executando a expressão `x[0] + 1` na segunda repetição do *loop*. Contudo, nunca existe um elemento 0 em qualquer objeto, pois índices no R começam a partir do número 1. Em outras palavras, uma expressão como `objeto[0]` vai sempre retornar um vetor vazio como resultado. Logo, um vetor vazio somado de 1 (isto é, a expressão `numeric(0)`

+ 1) é igual a um novo vetor vazio. Por esse motivo, que um vetor vazio foi retornado na segunda repetição do *loop*.

Exercício 2

15.2) Primeiro, precisamos de um for loop que visite cada um dos elementos do vetor vec.

```
vec <- c(
  5.2, 6.1, 2.3, 7.4, 1.1, 3.6,
  7.2, 8.1, 3.3, 4.5, 0.8, 5.4
)

for(i in seq_along(vec)){
  ### Acessando cada elemento de vec
  vec[i]
}
```

Para encontrarmos o valor máximo em vec, vamos basicamente, armazenar o primeiro elemento de vec em um objeto chamado x, e, ao longo do *loop*, vamos comparar cada elemento de vec a este objeto x. Caso o valor do elemento em que estamos no momento, for maior que o valor do objeto x, vamos redefinir o objeto x para este valor, e, prosseguir com o loop, comparando o próximo elemento com o valor do objeto x.

```
x <- vec[1]
for(i in seq_along(vec)){
  if(vec[i] > x){
    x <- vec[i]
  }
}
### Após a execução do loop, temos o valor máximo de vec:
print(x)
```

```
## [1] 8.1
```

```
### Perceba que a função max() retorna o mesmo resultado:
max(vec)
```

```
## [1] 8.1
```

Exercício 3

15.3) Recapitulando, temos a matriz mt abaixo, e desejamos preencher cada elemento desta matriz, com o resultado da multiplicação entre os índice que localizam o respectivo elemento. Repare que esse objeto mt é uma matriz de 30 linhas e 30 colunas.

```
mt <- matrix(ncol = 30, nrow = 30)
```

Primeiro, vamos criar um *loop* que visita cada linha dessa matriz, como demonstrado abaixo:

```
for (i in 1:nrow(mt)) {
  ### Corpo do loop:
  mt[i, ]
}
```

Agora, para cada linha da matriz `mt`, desejamos visitar cada uma das 30 colunas. Por isso, precisamos incluir um novo *loop* dentro do *loop* atual. Dessa forma, esse segundo *loop* será responsável por navegar ao longo das colunas desta matriz.

O iterador do primeiro *loop* é o objeto `i`, enquanto o iterador do segundo *loop* é o objeto `j`. Para cada repetição do primeiro *loop* (ou para cada valor assumido por `i`), o segundo *loop* é executado por completo (ou seja, todas as 30 repetições são executadas).

```
for (i in 1:nrow(mt)) {
  for (j in 1:ncol(mt)) {
    mt[i, j]
  }
}
```

Por último, precisamos apenas salvar em cada elemento visitado, o resultado da multiplicação dos iteradores desses dois *loops*, como demonstrado abaixo.

```
for (i in 1:nrow(mt)) {
  for (j in 1:ncol(mt)) {
    mt[i, j] <- i * j
  }
}
```

Podemos visualizar abaixo, as 5 primeiras linhas das 5 primeiras colunas dessa matriz `mt` após a execução do *loop* aninhado.

```
mt[1:5, 1:5]
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    2    4    6    8   10
## [3,]    3    6    9   12   15
## [4,]    4    8   12   16   20
## [5,]    5   10   15   20   25
```

Referências Bibliográficas

FRIEDL, J. E. F. *Mastering Regular Expressions*. 3. ed. Sebastopol, CA: O'Reilly Media, Inc., 2006. ISBN 0-596-52812-4.

R CORE TEAM. *R Language Definition*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.

Capítulo 16 - *Functional programming* com `purrr`

Exercício 1

16.1.A) Como desejamos calcular a média apenas das colunas númericas da tabela `diamonds`, podemos primeiro selecionar essas colunas específicas, antes de executar a função `map_dbl()`. Para isso, podemos utilizar a função `select()` que introduzimos no capítulo 5. Para selecionarmos todas as colunas numéricas, utilizamos a função `where()` dentro de `select()`, como demonstrado abaixo:

```
ggplot2::diamonds %>%
  select(where(is.numeric))
```

```
## # A tibble: 53,940 x 7
##    carat depth table price     x     y     z
##    <dbl> <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.23  61.5    55   326  3.95  3.98  2.43
##  2  0.21  59.8    61   326  3.89  3.84  2.31
##  3  0.23  56.9    65   327  4.05  4.07  2.31
##  4  0.29  62.4    58   334  4.2   4.23  2.63
##  5  0.31  63.3    58   335  4.34  4.35  2.75
##  6  0.24  62.8    57   336  3.94  3.96  2.48
##  7  0.24  62.3    57   336  3.95  3.98  2.47
##  8  0.26  61.9    55   337  4.07  4.11  2.53
##  9  0.22  65.1    61   337  3.87  3.78  2.49
## 10  0.23  59.4    61   338  4     4.05  2.39
## # ... with 53,930 more rows
```

Agora que temos todas as colunas numéricas da tabela, podemos repassar esse resultado para a função `map_dbl()`, e, pedir a ela que aplique a função `mean()` sobre cada coluna.

```
ggplot2::diamonds %>%
  select(where(is.numeric)) %>%
  map_dbl(mean)
```

```
##        carat        depth        table        price            x
##    0.7979397   61.7494049   57.4571839 3932.7997219    5.7311572
##            y            z
##    5.7345260    3.5387338
```

16.1.B) Podemos utilizar basicamente o mesmo processo do item anterior para responder esta questão. Basta selecionarmos todas as colunas não numéricas da tabela `diamonds` e, repassarmos o resultado para uma função `map`.

Para selecionarmos todas as colunas não numéricas, podemos apenas inverter o teste lógico (com o operador `!`) que utilizamos dentro da função `where()` para descobrirmos todas as colunas não numéricas.

```
ggplot2::diamonds %>%
  select(where(~!is.numeric(.)))
```

```
## # A tibble: 53,940 x 3
##    cut       color clarity
##    <ord>     <ord> <ord>
##  1 Ideal     E     SI2
##  2 Premium   E     SI1
##  3 Good      E     VS1
##  4 Premium   I     VS2
##  5 Good      J     SI2
##  6 Very Good J     VVS2
##  7 Very Good I     VVS1
##  8 Very Good H     SI1
##  9 Fair      E     VS2
## 10 Very Good H     VS1
## # ... with 53,930 more rows
```

Por último, repassamos o resultado para `map()` abaixo, e pedimos a essa função que aplique a função `dplyr::n_distinct()` sobre cada coluna.

```
ggplot2::diamonds %>%
  select(where(~!is.numeric(.))) %>%
  map(dplyr::n_distinct)
```

```
## $cut
## [1] 5
##
## $color
## [1] 7
##
## $clarity
## [1] 8
```

Exercício 2

16.2.A) Toda função `map` vai aplicar a função definida em seu segundo argumento, sobre cada elemento do objeto definido em seu primeiro argumento. Portanto, no comando `map(1:5, rnorm)`, a função `map()` vai aplicar a função `rnorm()` sobre cada elemento do vetor resultante de `1:5`.

Sendo assim, o primeiro resultado gerado por `map(1:5, rnorm)` é o resultado do comando `rnorm(1)`; já o segundo resultado é equivalente ao comando `rnorm(2)`; enquanto o terceiro resultado, equivale a `rnorm(3)`; e assim por diante.

Como o primeiro argumento (`n`) de `rnorm()` define a quantidade de valores que serão gerados pela função, a cada iteração de `map()`, uma quantidade diferente de valores são gerados pela função `rnorm()`.

16.2.B) Como descrevemos no item anterior, toda função `map` vai aplicar a função definida em seu segundo argumento, sobre cada elemento do objeto definido em seu primeiro argumento. Porém, todo argumento que é definido em argumentos posteriores ao segundo argumento da função, são iterpretados como argumentos estáticos.

Isso significa que, o terceiro argumento `n = 5` é sempre repassado para a função `rnorm()`. Portanto, se no item anterior, estávamos executando os comandos `rnorm(1)`, `rnorm(2)`, etc., neste item, estamos executando os comandos `rnorm(n = 5, 1)`, `rnorm(n = 5, 2)`, etc.

Por esse motivo, a funçao `rnorm()` sempre gera 5 valores diferentes a cada iteração da função `map()`. Por outro lado, cada elemento do vetor resultante de `1:5`, são interpretados como o segundo argumento de `rnorm()` (pois o primeiro argumento dessa função - `n`, já foi definido em `n = 5`), o qual define a média aproximada dos valores gerados pela função.

Ou seja, o número de valores gerados a cada iteração de `map()` permanece o mesmo. Contudo, a média estimada desses valores está mudando a cada iteração.

Exercício 3

16.3.A) Para descobrirmos o comprimento de um dado elemento de `l` podemos aplicar a função `length()` sobre este elemento. Consequentemente, para descobrirmos se esse elemento possui comprimento maior que 3, podemos simplesmente aplicar o teste `length(x) > 3` sobre cada elemento (`x`).

Você pode realizar esse trabalho de duas maneiras. Primeiro, você pode coletar o comprimento de todos os elementos da lista `l`, com o comando `map_int(l, length)`, e, em seguida, aplicar o teste lógico de “maior que 3” sobre o resultado desse comando, como demonstrado abaixo.

```
l <- list(
  c(1, 5),
  c(5, 6, 1),
  c(9, 8, 9, 0, 0, 1),
  c(7, 4, 4, 2),
  c(4, 5)
)
```

```
map_int(l, length) > 3
```

```
## [1] FALSE FALSE  TRUE  TRUE FALSE
```

Uma outra forma, seria incluir todo o teste lógico dentro da função map. Dessa forma, estamos aplicando o comando `length(x) > 3` sobre cada elemento da lista `l`.

```
map_lgl(l, ~length(.) > 3)
```

```
## [1] FALSE FALSE  TRUE  TRUE FALSE
```

Possuindo os resultados do teste lógico em mãos, basta repassá-lo para a função de *subsetting*.

```
l[ map_lgl(l, ~length(.) > 3) ]
```

```
## [[1]]
## [1] 9 8 9 0 0 1
##
## [[2]]
## [1] 7 4 4 2
```

16.3.B) Para descobrirmos quais elementos de `l` possuem valores `NA`, precisamos primeiro aplicar a função `is.na()` sobre cada elemento de `l`. Porém, perceba abaixo que essa função nos retorna um vetor para cada elemento de `l`.

```
l <- list(
  c(1, 1, 2),
  c(6, 7, NA, 9),
  c(NA, NA, 1, 3, 4),
  c(3, 3, 1, 8),
  c(6, 6, 6)
)

map(l, is.na)
```

```
## [[1]]
## [1] FALSE FALSE FALSE
##
## [[2]]
## [1] FALSE FALSE  TRUE FALSE
##
## [[3]]
## [1]  TRUE  TRUE FALSE FALSE FALSE
##
## [[4]]
## [1] FALSE FALSE FALSE FALSE
##
## [[5]]
## [1] FALSE FALSE FALSE
```

Precisamos reduzir os resultados dessa função para um único valor para cada elemento de `l`. Para isso, podemos aplicar a função `any()` sobre os resultados de `is.na()`. Com isso, temos

um vetor do tipo `logical`, que indica se cada elemento de `l` possui pelo menos 1 valor `NA`.

```
map_lgl(l, ~any(is.na(.)))
```

```
## [1] FALSE  TRUE  TRUE FALSE FALSE
```

Porém, o objetivo deste item é filtrar todos os elementos de `l` que **não possuem** algum valor `NA`. Portanto, precisamos inverter esse teste lógico, com o operador `!`, como demonstrado abaixo:

```
map_lgl(l, ~!any(is.na(.)))
```

```
## [1]  TRUE FALSE FALSE  TRUE  TRUE
```

Agora que temos os resultados do teste lógico que precisávamos, podemos repassar esses resultados para a função de *subsetting*. Sendo assim, temos todos os elementos de `l` que não possuem algum valor `NA`.

```
l[ map_lgl(l, ~any(is.na(.))) ]
```

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

Referências Bibliográficas

FRIEDL, J. E. F. *Mastering Regular Expressions*. 3. ed. Sebastopol, CA: O'Reilly Media, Inc., 2006. ISBN 0-596-52812-4.

R CORE TEAM. *R Language Definition*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.

Capítulo 18 - *Environments* ou ambientes no R

Exercício 1

18.1.A) Antes de tudo, vale reforçar que, em toda expressão que inclui o operador de *super-assignment* (`<<-`), existe uma diferença importante entre os objetos que estão à direita do operador, e os objetos que estão à esquerda do mesmo.

Por exemplo, em uma expressão como `x <<- y + z`, o R procura os objetos que estão à direita do operador (`y` e `z`) da mesma que forma que faria nos demais tipos de expressão (isto é, ele segue as regras de *scoping*). Logo, ele procura no *environment* atual, depois, sobe a árvore genealógica até atingir o *global environment*, em seguida, sobe o *search path* de sua sessão.

Em contrapartida, ao procurar pelo objeto que está à esquerda do operador (`x`), o R segue as regras que descrevemos na sessão 18.8 a respeito do operador de *super-assignment*. Portanto, o R procura por `x` no *parent environment*, caso ele não encontre esse objeto lá, ele então realiza o *assignment* dentro de seu *global environment* (**??**).

Tendo essas informações em mente, quando avaliamos a expressão `obj <<- obj[i] + 1` dentro do *environment* `env1`, o R primeiro tenta procurar pelos objetos `i` e `obj`, dentro do próprio `env1`. Consequentemente, o R encontra os valores 1 e `c(10, 20)`, e calcula o valor da sub-expressão `obj[i] + 1`, que é igual a `11`.

Depois, o R tenta descobrir onde o *assignment* vai ocorrer. Por isso, ele começa a procurar por `obj` no *parent environment* de `env1`, que no caso, é o *global environment*. Porém o R não encontra `obj` neste environment. Como resultado, o R vai avaliar o *assignment* dentro do *global environment*.

Por esse motivo que, ao avaliarmos a expressão `obj <<- obj[i] + 1` dentro de `env1`, um novo

objeto chamado `obj` é criado dentro do *global environment*.

```
get("obj", envir = globalenv())
```

```
## [1] 11
```

18.1.B) Dessa vez, como estamos executando a expressão `obj <<- obj[i] + 1` dentro de `env2`, os objetos `i` e `obj` vão ser inicialmente procurados dentro de `env2`, depois em seu parent environment (`env1`). Consequentemente, o valor 2 será encontrado para o objeto `i`, e `c(10, 20)` para o objeto `obj`. Como resultado, dessa vez, a sub-expressão `obj[i] + 1` gera o valor 21.

Em seguida, o R tenta descobrir o local onde o *assignment* será avaliado. Logo, ele tenta encontrar um objeto chamado `obj` no *parent environment* de `env2` (isto é, `env1`). Como `env1` possui um objeto chamado `obj`, tal objeto é redefinido para o valor 21.

```
env1$obj
```

```
## [1] 21
```

Exercício 2

18.2.A) O resultado será 16.

```
x <- 1
f <- function(x){
  x <- 15
  print(x + 1)
}

### Qual será o resultado do comando print()
### executado por f() ?
f()
```

```
## [1] 16
```

18.2.B) O resultado será o vetor `c(11, 21)`.

```
### Iniciei uma nova sessão no R
obj <- c(1, 2)

env1 <- rlang::env()
env2 <- rlang::env(env1)
env3 <- rlang::env(env2)

env1$obj <- c(10, 20)
env3$obj <- c(100, 200)

eval(quote(obj <<- obj + 1), envir = env2)

### Qual é o resultado do print() abaixo ?
print(get("obj", envir = env1))
```

```
## [1] 11 21
```

Referências Bibliográficas

FRIEDL, J. E. F. *Mastering Regular Expressions*. 3. ed. Sebastopol, CA: O'Reilly Media, Inc., 2006. ISBN 0-596-52812-4.

R CORE TEAM. *R Language Definition*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.