R do Zero (1): Sintaxe Básica

João Pedro Oliveira, Pedro Cavalcante & Marina Merlo

Primeiros conceitos

Se você já tem a linguagem R instalada, bem como o RStudio, seu console deve mostrar algo como:

```
R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)
```

Projetos no RStudio

Antes de mais nada. Olhe para o canto direito superior do RStudio. Lá deve estar algo como `no project`. Clique e na caixinha que abrir clique em `New Project`. Você vai ser guiado a escolher uma pasta no computador (ou criar uma nova) e dar um nome ao projeto. Toda atividade autocontida é um projeto. Trabalho de faculdade, tarefa no trabalho, pacote de R, qualquer grupo coeso de código, texto, arquivos e dados que servem a uma finalidade comum.

Projetos são muito importantes para não termos que nos preocupar em dar endereços completos para arquivos no computador - gerenciar o sistema de diretórios de trabalho feito nos anos 90 do R é sempre desagradável e não precisamos disso. Projetos farão esse trabalho sujo por nós e tornarão sua experiência muito mais cômoda. Sempre que quiser trabalhar em um projeto, basta abrir o arquivo `.Rproj` na pasta dele ou usar o seletor na caixa do canto direito superior.

Por fim, se você sabe o que é git: sim, você pode *e deve* usar projetos de R junto com git. O lembrete do amigo é tome cuidado com repositórios públicos, eles idealmente não deveriam conter o seu arquivo `.Rproj`.

Sintaxe Básica

A sintaxe de uma linguagem de programação define o que é considerado um comando *válido*. Isso é diferente do comando ser útil, ou cumprir o que o usuário espera que cumpra - esses problemas estão no campo da *Semântica*.

Seguir a sintaxe é apenas escrever seguindo convenções que o computador também segue.

Vamos agora nos familiarizar com a sintaxe básica. Podemos usar R como uma calculadora potente:

```
2 + 2
```

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

```
2 - 3
```

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

```r
3^2
```

```
## [1] 9
```

```r
4^2
```

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

```r
3^3
```

```
## [1] 27
```

```r
2*3
```

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

```r
2/3
```

```
## [1] 0.6666667
```

Ou como um computador, que recebe comandos chamados *funções* e os executa:

```r
Sys.time() # horário do sistema
```

```
## [1] "2020-10-06 16:06:53 -03"
```

```r
seq(1, 10) # sequência de 1 até 10
```

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

```r
seq(1, 10, by = 2) # sequência de 1 até 10 pulando 2 unidades
```

```
## [1] 1 3 5 7 9
```

```r
paste("por exemplo, 2 + 2 =", 2 + 2)
```

```
## [1] "por exemplo, 2 + 2 = 4"
```

Podemos usar funções, que são unidades reaproveitaveis de certas operações. Executar o comando `Sys.Date()` irá printar a data do sistema, assim como `Sys.time()` irá printar o horário. Essas funções não recebem parâmetros, não podemos alterar seu comportamento. No entanto, funções como `seq()`, que geram sequências, dependem de argumentos informados, como de onde para onde a sequência deve ir.

Alguns argumentos são tidos como padrão. Por exemplo, se não informarmos o argumento `by =`, a sequência é gerada com passos de 1 unidade. Como sabemos disso? Usando o comando `help(seq)`, que exibe a documentação da função apresentada.

```r
1:5 # os números de 1 a 5
```

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

```r
seq(1, 5) # equivalente ao anterior
```

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

```r
seq(from = 1, to = 5, by = 0.5) # espaçamento de .5 e argumentos nomeados
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```r
exp(3) # exponencial de 3
```

```
## [1] 20.08554
```

```r
sqrt(3) # raiz quadrada de 3
```

```
## [1] 1.732051
```

```r
log(3) # log natural de 3
```

```
## [1] 1.098612
```

```r
log(3, base = 10) # base 10
```

```
## [1] 0.4771213
```

Testes lógicos

Mais importante, podemos testar se certas proposições têm validade lógica. É o que torna uma linguagem de programação uma caixinha de areia em que proposições podem ser enunciadas, testadas e com base nisso executar comandos. O sinal que usamos para testar a noção de *igualdade* é `==`. Dois sinais iguais seguidos.

```r
2 == 2
```

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

Vamos entender o que se passou aqui. Executamos uma *expressão*, um enunciado sintaticamente válido. E recebemos um *retorno*, o valor `TRUE`, a maneira da linguagem de se referir à noção abstrata de que uma expressão carrega um enunciado verdadeiro. O inverso de `==` é `!=`, o sinal para testar falsidade.

```r
2 == 3 # enunciado: dois é igual a três
```

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

```r
2 != 3 # dois é diferente de três
```

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

```r
2 != 2 # dois é diferente de dois
```

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

```r
2 < 3 # 2 é menor que 3
```

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

```r
2 > 3 # 2 é maior que 3
```

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

```r
2 >= 3 # 2 é maior ou igual a 3 (nem maior nem igual, portanto falso)
```

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

```r
2 <= 2 # 2 é menor ou igual a 2 (de fato, pois apesar de não ser menor é igual)
```

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

```r
2 %% 1 # resto da divisão
```

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

```r
3 %% 2
```

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

```r
468 %% 17
```

```
## [1] 9
```

Texto no R sempre está entre aspas, simples ou duplas.

```r
"a" == "b" # a letra "a" é igual à letra "b"
```

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

```r
"a" == 'a'
```

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

```
"palavra maior" != "apenas testando igualdades"
```

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

Podemos também testar proposições conjuntamente. O operador `|` liga duas expressões e irá retornar verdadeiro se pelo menos uma das duas retornas verdadeiro. É o operador do *OU*. '`A | B` é lido como "A ou B".

```
TRUE | FALSE # uma ou outra é verdadeira?
```

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

```
!TRUE | (2 == 3) # negação de verdadeiro ou 2 é igual a 3 são verdadeiros?
```

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

```
!TRUE | (2 == 2)
```

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

```
FALSE | FALSE
```

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

```
!FALSE | FALSE
```

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

```
2 == 2 | 3 == 3
```

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

Já o operador `&` é mais forte. É o operador do *E*. Ele requer que as duas expressões sejam verdadeiras.

```
TRUE & FALSE # uma ou outra é verdadeira?
```

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

```
!TRUE & (2 == 3) # negação de verdadeiro ou 2 é igual a 3 são verdadeiros?
```

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

```
!TRUE & (2 == 2)
```

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

```r
FALSE & FALSE
```

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

```r
!FALSE & FALSE
```

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

```r
2 == 2 & 3 == 3
```

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

Tipos e vetores

Observe que encontramos quatro variedades de dado até agora:

- Números reais com precisão dupla: 8.42, 8.0001, 0.0123128790182341
- Números inteiros: 1, 8, 15
- Valores lógicos: `TRUE`, `FALSE`
- Texto: `"um pedaço de texto"`

Chamamos isso de *tipo* do dado. Dado em texto é normalmente chamado de *string* (por ser um "fio" de letras) e em números é comum se refeir como *double* (do inglês, *double precision*, já que, ao contrário de números inteiros, aceitam casas decimais). Podemos armazenar dados do mesmo tipo em estruturas chamadas vetores, são listas de valores do mesmo tipo. A função `c()` - do inglês, *combine* - recebe valores e devolve um vetor.

```r
c(2, 2, 2.5)
```

```
## [1] 2.0 2.0 2.5
```

```r
c(2, 2, 2.5) == 2
```

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

```r
c("a", "b", "c") == "b"
```

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

Vetores são uma *estrutura de dados* muito utilizada em R. Sempre que precisamos representar um conjunto de dados do mesmo tipo usaremos vetores. É importante que sejam do *mesmo* tipo. Se não forem e conversão for possível, então o R irá converte-los.

```r
c(2, 2, 2.5, "e", TRUE) # números e valores lógicos convertidos em texto
```

```
## [1] "2"    "2"    "2.5"  "e"    "TRUE"
```

```r
c(2, TRUE, FALSE) # valor lógico convertido em número
```

```
## [1] 2 1 0
```

Veremos que nem sempre essas conversões são indesejadas - de fato elas funcionarão como ajudantes em vários momentos. O problema é que esse tipo de comportamento pode gerar

Designação

Podemos usar funções como `rnorm`, do inglês *random normal*, para simular dados tirados de uma distribuição normal, compara-los com algum critério e tabular os resultados.

```r
rnorm(n = 10)
```

```
##  [1] -0.0579228 -1.9721136  0.0143246  1.0258166 -0.5810752  0.3850357
##  [7] -0.6005037 -0.5407427 -0.6485530  1.4336425
```

```r
rnorm(n = 10) < 0
```

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

```r
table(rnorm(n = 10) < 0)
```

```
##
## FALSE  TRUE
##     5     5
```

Você percebeu que seria mais cômodo não ter que repetir em todo lugar o termo `rnorm(n = 10)`. Podemos nos referir a um conjunto de dados por nomes. Para isso usamos um operador, `<-`. A "seta de designação", como é chamada, atribui a um símbolo um valor.

```r
a <- 1

a # apenas nome do objeto printa ele no console
```

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

```r
(b <- 1:5 + 2) # uma expressão entre parênteses é avaliada e seu resultado é printado
```

```
## [1] 3 4 5 6 7
```

```r
(c <- letters) # alguns objetos com constantes já vêm carregados na memória, como as letras
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

```r
(d <- c(2, 4, 7) / 2)
```

```
## [1] 1.0 2.0 3.5
```

Refazendo nosso exemplo:

```r
normais <- rnorm(n = 100)
table(normais < 0)
```

```
##
## FALSE  TRUE
##    52    48
```

Agora também é muito mais prático e legível realizar operações sobre os mesmos dados.

```r
mean(normais) # a média
```

```
## [1] 0.01009291
```

```r
sd(normais) # desvio-padrão
```

```
## [1] 0.9884132
```

```r
head(normais) # as primeiras entradas do vetor
```

```
## [1]  0.3482275 -0.2219472 -0.2335395  0.1268364  0.6734493 -1.5652803
```

```r
head(round(normais, digits = 1)) # arredonde o vetor para 1 dígito
```

```
## [1]  0.3 -0.2 -0.2  0.1  0.7 -1.6
```

```r
paste("A média da simulação foi:", round(mean(normais), digits = 3))
```

```
## [1] "A média da simulação foi: 0.01"
```

Exercício rápido A função `runif` gera dados uniformemente distribuídos. Como a `rnorm`, seu primeiro argumento é `n=`, o número de dados a serem simulados. Os dois seguintes são `min=`, que por padrão é 0 e `max=` que por padrão é 1.

- Simule 100 observações tirados de uma uniforme entre 15 e 40 e armazene o resultado em um obeto chamado `vetor_simulado`
- Descubra quantas entradas ficaram abaixo de 30
- Printe uma mensagem no console dizendo qual foi o desvio-padrão, arredondado para 2 dígitos

Outras estruturas de dados

Listas

Vamos nos lembrar que vetores são homogêneos no tipo e irão converter dados se possível para garantir isso.

```r
(vetor_feao <- c(1.243, "feao", "vetores seguram dados apenas de um tipo")) # o número 1.243 é converti
```

```
## [1] "1.243"
## [2] "feao"
## [3] "vetores seguram dados apenas de um tipo"
```

```r
(vetor_erro <- c(TRUE, TRUE, "texto")) # de novo, convertido para texto
```

```
## [1] "TRUE"  "TRUE"  "texto"
```

Para quando precisamos de algo *como* um vetor que carregue dados de varios tipos, usamos listas.

```r
(lista1 <- list(A = 1, B = "já listas seguram qualquer tipo", C = FALSE))
```

```
## $A
## [1] 1
##
## $B
## [1] "já listas seguram qualquer tipo"
##
## $C
## [1] FALSE
```

```r
str(lista1) # a estrutura da lista
```

```
## List of 3
##  $ A: num 1
##  $ B: chr "já listas seguram qualquer tipo"
##  $ C: logi FALSE
```

```r
lista1$B # acessamos elementos nomeados de listas com o operador $
```

```
## [1] "já listas seguram qualquer tipo"
```

```r
(lista2 <- list(D = "e até mesmo:", E = list(Fa = "outras", G = "listas")))
```

```
## $D
## [1] "e até mesmo:"
##
## $E
## $E$Fa
## [1] "outras"
##
## $E$G
## [1] "listas"
```

```r
str(lista2)
```

```
## List of 2
##  $ D: chr "e até mesmo:"
##  $ E:List of 2
##   ..$ Fa: chr "outras"
##   ..$ G : chr "listas"
```

```
lista2$E
```

```
## $Fa
## [1] "outras"
##
## $G
## [1] "listas"
```

Dataframes/tibbles

Vetores armazenam apenas um tipo de dado, mas a maioria dos conjuntos de dados do mundo é multi-tipo. Para isso usamos "Data Frames", ou *tibbles* como eles são chamados em R moderno. Se já não tiver o pacote `tibble` instalado, resolva isso com o comando `install.packages("tibble", dependencies = TRUE)`.

```
library(tibble)

citation("tibble") # citação do pacote
```

```
##
## To cite package 'tibble' in publications use:
##
##   Kirill Müller and Hadley Wickham (2020). tibble: Simple Data Frames.
##   R package version 3.0.3. https://CRAN.R-project.org/package=tibble
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {tibble: Simple Data Frames},
##     author = {Kirill Müller and Hadley Wickham},
##     year = {2020},
##     note = {R package version 3.0.3},
##     url = {https://CRAN.R-project.org/package=tibble},
##   }
```

```
(tib <- tibble(
  a = 1:10,
  B = list(10:13), # tibbles podem carregar listas
  c = 100*a/a^2, # e receber vetores sem nomes
  D = sample(letters, size = 10), # chamando uma função
  e = 2.56*a/3*a + pi))
```

```
## # A tibble: 10 x 5
##        a B             c D         e
##    <int> <list>    <dbl> <chr> <dbl>
##  1     1 <int [4]> 100   f      3.99
```

```
##  2     2 <int [4]>  50   b      6.55
##  3     3 <int [4]>  33.3 i     10.8
##  4     4 <int [4]>  25   e     16.8
##  5     5 <int [4]>  20   h     24.5
##  6     6 <int [4]>  16.7 u     33.9
##  7     7 <int [4]>  14.3 y     45.0
##  8     8 <int [4]>  12.5 c     57.8
##  9     9 <int [4]>  11.1 n     72.3
## 10    10 <int [4]>  10   a     88.5
```

```
nrow(tib) # número de linhas
```

```
## [1] 10
```

```
ncol(tib) # número de colunas
```

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

```
glimpse(tib) # descrição da estrutura
```

```
## Rows: 10
## Columns: 5
## $ a <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
## $ B <list> [<10, 11, 12, 13>, <10, 11, 12, 13>, <10, 11, 12, 13>, <10, 11, ...
## $ c <dbl> 100.00000, 50.00000, 33.33333, 25.00000, 20.00000, 16.66667, 14.2...
## $ D <chr> "f", "b", "i", "e", "h", "u", "y", "c", "n", "a"
## $ e <dbl> 3.994926, 6.554926, 10.821593, 16.794926, 24.474926, 33.861593, 4...
```

Dataframes são a estrutura de dado que você mais irá utilizar no seu dia a dia programando em R. Isso porque eles são feitos para representar dados estruturados - aqueles que você acessaria com programas como SPSS, SAS, Stata e Excel.

Podemos importar dados de variados formatos com a função `import` do pacote `rio`. Basta darmos o link do arquivo *ou* um endereço na pasta com o arquivo.

```
library(rio)
# tirado do link original https://www.learningcontainer.com/bfd_download/sample-xls-file-for-testing/
link <- "https://www.learningcontainer.com/bfd_download/excel-spreadsheet-examples-for-students/"

(dados <- import(link, setclass = "tibble"))
```

```
## Unrecognized file format. Try specifying with the format argument.
```

```
## # A tibble: 500 x 9
##    `First Name` `Last Name` `Company Name` Address City  State `Phone No` Email
##    <chr>        <chr>       <chr>          <chr>   <chr> <chr> <chr>      <chr>
##  1 Rebbecca     Didio       Brandt, Jonat~ 171 E ~ Leith TAS   03-8174-9~ rebb~
##  2 Stevie       Hallo       Landrum Tempo~ 22222 ~ Pros~ QLD   07-9997-3~ stev~
##  3 Mariko       Stayer      Inabinet, Mac~ 534 Sc~ Hamel WA    08-5558-9~ mari~
##  4 Gerardo      Woodka      Morris Downin~ 69206 ~ Talm~ NSW   02-6044-4~ gera~
##  5 Mayra        Bena        Buelt, David ~ 808 Gl~ Lane~ NSW   02-1455-6~ mayr~
```

```
##  6 Idella       Scotland    Artesian Ice ~ 373 La~ Cart~ WA    08-7868-1~ idel~
##  7 Sherill      Klar        Midway Hotel   87 Syl~ Nyam~ WA    08-6522-8~ skla~
##  8 Ena          Desjardiws  Selsor, Rober~ 60562 ~ Bend~ NSW   02-5226-9~ ena_~
##  9 Vince        Siena       Vincent J Pet~ 70 S 1~ Purr~ QLD   07-3184-9~ vinc~
## 10 Theron       Jarding     Prentiss, Pau~ 8839 V~ Blan~ SA    08-6890-4~ tjar~
## # ... with 490 more rows, and 1 more variable: Web <chr>
```

Funções e Noções de Programação Funcional

Você já deve ter percebido que funções são algo de importante em R. São maneiras de reproduzir operações complexas. Como guia, se você escreve mais de duas vezes a mesma operação, vale por numa função.

Funções têm três componentes, vamos nos preocupar com dois deles no momento:

- O corpo

Tudo que estará entre `{}`, é lá que a lógica da função é escrita.

- Os argumentos, ou parâmetros

Valores que passamos para a função quando queremos executar a sua lógica. Ao usar `mean` para calcular médias damos um vetor de números ou podemos falar se dados faltantes deverão ser ignorados. Esses parâemtros são importantes porque alteram o comportamento da função.

Por padrão a função retorna o valor da última expressão dentro do corpo

```
library(glue) # biblioteca que ajuda a "colar" valores de código em texto

termometro <- function(temperatura) {

    glue("Ontem a temperatura foi de {temperatura} graus Celsius")

  }

termometro()
```

```
## Error in eval(parse(text = text, keep.source = FALSE), envir): argument "temperatura" is missing, wi
```

```
termometro(25.34534123)
```

```
## Ontem a temperatura foi de 25.34534123 graus Celsius
```

Podemos melhorar a nossa função dando um valor padronizado para ela. E arredondando os números.

```
termometro <- function(temperatura = 25) {

    glue("Ontem a temperatura foi de {round(temperatura, 2)} graus Celsius")

  }

termometro()
```

```
## Ontem a temperatura foi de 25 graus Celsius
```

```
termometro(23.122314)
```

```
## Ontem a temperatura foi de 23.12 graus Celsius
```

Podemos passar também expressões a serem avaliadas. R é uma linguagem *preguiçosa*, no sentido de que expressões só são calculadas quando absolutamente necessárias. Por isso podemos fazer:

```
termometro <- function(
  temperatura = rnorm(n = 1, mean = 25, sd = 5) # um valor aleatório
  ) {

    glue("Ontem a temperatura foi de {round(temperatura, 2)} graus Celsius")

  }

termometro()
```

```
## Ontem a temperatura foi de 29.35 graus Celsius
```

```
termometro()
```

```
## Ontem a temperatura foi de 25.65 graus Celsius
```

```
termometro()
```

```
## Ontem a temperatura foi de 23.61 graus Celsius
```

Controle de Fluxo

Vamos melhorar o nosso termômetro. Imagine que precisamos agora poder receber um vetor com uma lista de leituras de temperatura e exibi-las. Funções como `if` permitem criar *controle de fluxo*, que é o que precisamos para isso.

Se o parâmetro temperatura tiver apenas uma leitura, o R interpretará como um vetor de uma entrada. Por exemplo:

```
length(2)
```

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

```
length(letters) # comprimento do vetor com o alfabeto latino
```

```
## [1] 26
```

```
length(rnorm(10))
```

```
## [1] 10
```

E podemos passar um teste lógico para `if` verificar e então decidir o que executar. Podemos colocar mais condições a serem verificadas com `else if` e podemos definir `else` para todos os outros casos não-testados.

O código a ser executado precisa ficar dentro de `{}`. É o que chamamos, em R, de um *escopo*.

```
if(runif(1) < 1/2) {

  print("Menor que 1/2")

  } else {

  print("Maior que 1/2")

  }
```

```
## [1] "Menor que 1/2"
```

Cuidado com a pegadinha Um erro muito comum no início é passar um vetor como condição:

```
if(runif(10) < 1/2) {

  print("Menor que 1/2")

  } else {

  print("Maior que 1/2")

  }
```

```
## Warning in if (runif(10) < 1/2) {: the condition has length > 1 and only the
## first element will be used
```

```
## [1] "Maior que 1/2"
```

A condição pode ter apenas comprimento 1. Se tiver mais a lógica de `if` irá usar apenas a primeira entrada e seu código não irá se comportar como você espera. Esse é um problema comum de semântica.

Vamos agora aplicar isso ao nosso exemplo do termometro.

```
termometro <- function(temperatura = rnorm(n = 1, mean = 25, sd = 5)) {

  if(length(temperatura) > 1) {

    return(
      glue("As temperaturas nos últimos dias foram: {
        paste(round(temperatura, 2), collapse = ', ')
        }")
    )

  } else {

    return(
```

```
      glue("Ontem a temperatura foi de {round(temperatura, 2)} graus Celsius")
    )

  }

}

termometro()
```

```
## Ontem a temperatura foi de 24.36 graus Celsius
```

```
termometro(25.234)
```

```
## Ontem a temperatura foi de 25.23 graus Celsius
```

```
termometro(runif(n = 5, min = 10, max = 40))
```

```
## As temperaturas nos últimos dias foram: 28.53, 11.1, 31.48, 14.34, 14.89
```

Funções anônimas e manipulação de vetores e listas

Imagine que você recebe um vetor contendo medidas de algum sistema. Se cair uma letra o sistema está bem, um número é um erro.

Para fazer algumas análises você precisa filtrar os erros do vetor. Como faríamos isso?

Algumas funções do pacote `purrr` ajudam. Elas precisam receber duas coisas para funcionar:

- Uma lista, ou um vetor
- Uma função

Em particular algumas dessas funções são as que recebem qualquer tipo de dado e retornam algo `TRUE/FALSE`. Chamamos elas de *predicados*. Podemos inveter qualquer precicado com a função `negate` do pacote `purrr`.

As funções, também do `purrr` para manipular listas e vetores são:

- `keep`, que preserva as entradas dado um predicado.
- `discard`, que descarta as entradas que atendem ao predicado.
- `modify`, que modifica a lista/vetor de acordo com a função dada.

Como sabemos por exemplo se um valor é uma letra? Basta estar contido no vetor `letters` que já está pré-carregado na memória.

```
2 %in% letters
```

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

```
"a" %in% letters
```

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

Então uma função para isso seria:

```r
is_letter <- function(x) {

  x %in% letters

}

is_letter(2)
```

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

```r
is_letter("a")
```

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

```r
is_letter(c("a","b", "c"))
```

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

Vamos a alguns exemplos. Primeiro, precisamos criar algumas medidas simuladas e armazenar no vetor `entradas`.

```r
(alfanumerico <- c(letters, 1:10))
```

```
##  [1] "a"  "b"  "c"  "d"  "e"  "f"  "g"  "h"  "i"  "j"  "k"  "l"  "m"  "n"  "o"
## [16] "p"  "q"  "r"  "s"  "t"  "u"  "v"  "w"  "x"  "y"  "z"  "1"  "2"  "3"  "4"
## [31] "5"  "6"  "7"  "8"  "9"  "10"
```

```r
(entradas <- sample(alfanumerico, 10))
```

```
##  [1] "n"  "e"  "z"  "l"  "6"  "5"  "10" "p"  "j"  "2"
```

Tendo `entradas` em mãos podemos filtrar usando `is_letter` apenas os valores que nos interessam:

```r
library(purrr)

keep(entradas, is_letter) # mantém as letras
```

```
## [1] "n" "e" "z" "l" "p" "j"
```

```r
keep(entradas, negate(is_letter)) # mantém as não-letras
```

```
## [1] "6"  "5"  "10" "2"
```

```r
discard(entradas, is_letter) # descarta as letras
```

```
## [1] "6"  "5"  "10" "2"
```

```
discard(entradas, negate(is_letter)) # descarta as não-letras
```

```
## [1] "n" "e" "z" "l" "p" "j"
```

Também podemos usar funções que não justificam serem salvas, podemos chamar elas de anônimas. Existem algumas maneiras de escrever funções anônimas em R, vamos ver duas por enquanto.

- Podemos usar o construtor de fórmulas `~` e nos referir ao valor como `.x`:

```
keep(entradas, ~ .x %in% letters)
```

```
## [1] "n" "e" "z" "l" "p" "j"
```

- Ou podemos usar a função que cria funções, `function`:

```
keep(entradas, function(.x) .x %in% letters ) # para expressões de uma linha os {} são opcionais
```

```
## [1] "n" "e" "z" "l" "p" "j"
```

Esses construtores funcionam com `modify` também - na verdade com várias outras funções importantes que veremos mais à frente - e comporta controle de fluxo.

```
modify(entradas,
    function(.x) {
      ifelse(
        test = .x %in% letters,
        yes = glue("A entrada {.x} é uma letra"),
        no = glue("A entrada {.x} é um número")
      )
    }
)
```

```
##  [1] "A entrada n é uma letra"  "A entrada e é uma letra"
##  [3] "A entrada z é uma letra"  "A entrada l é uma letra"
##  [5] "A entrada 6 é um número"  "A entrada 5 é um número"
##  [7] "A entrada 10 é um número" "A entrada p é uma letra"
##  [9] "A entrada j é uma letra"  "A entrada 2 é um número"
```

```
# ou com fórmulas

modify(entradas,
    ~ ifelse(
        test = .x %in% letters,
        yes = glue("A entrada {.x} é uma letra"),
        no = glue("A entrada {.x} é um número")
      )
)
```

```
##  [1] "A entrada n é uma letra"  "A entrada e é uma letra"
##  [3] "A entrada z é uma letra"  "A entrada l é uma letra"
##  [5] "A entrada 6 é um número"  "A entrada 5 é um número"
##  [7] "A entrada 10 é um número" "A entrada p é uma letra"
##  [9] "A entrada j é uma letra"  "A entrada 2 é um número"
```

Exercício rápido Escreva uma função que receba um vetor numérico e devolva apenas os números pares contidos nele. Lembre-se que o resto da divisão de números pares por 2 é zero.

Um estudo de caso para validar CPFs

Digamos que você esteja trabalhando com um banco de dados identificado por CPF e há suspeita de algumas entradas são de identidades falsas.

Você também sabe que os falsificadores não eram muito bons porque esqueceram que existe uma maneira de validar CPF. Podemos escrever uma função que faça isso.

Como validar um CPF

A maneira de calcular é simples. Os dois dígitos no final são os *dígitos verificadores*. Sabemos isso com as seguintes contas:

- Pegue os 9 primeiros dígitos do CPF, multiplique pelos números de 2 a 10 em ordem decrescente

Então no CPF `793.224.651-23` a conta seria:

```
7*10 + 9*9 + 3*8 + 2*7 + 2*6 + 4*5 + 6*4 + 5*3 + 1*2
```

Agora multiplique o resultados por 10 e divida por 11. Se o resto da divisão bater com o primeiro dígito, ele está validado.

```
(262*10) %% 11 == 2 # primeiro dígito é válido
```

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

- Pegue os 10 primeiros números e multiplique pelos números de 11 a 2 em ordem decrescente

```
7*11 + 9*10 + 3*9 + 2*8 + 2*7 + 4*6 + 6*5 + 5*4 + 1*3 + 2*2
```

Agora multiplique o resultados por 10 e divida por 11. Se o resto da divisão bater com o segundo dígito, ele está validado.

```
(305*10) %% 11 == 3
```

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

Como os dois são verdadeiros, está válidado o CPF. Vamos por isso numa função?

Escrevendo essa regra em uma função

A primeira coisa é garantir que o usuário inseriu 11 dígitos - com ou sem sinais. Então vamos primeiro quebrar uma entrada em um vetor de caracteres, manter apenas os números e verificar o comprimento.

```r
(numeros <- as.character(0:9))
```

```
##  [1] "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
```

```r
cpf <- "793.224.651-23"
```

A biblioteca `stringr` tem funções que ajudam a manipular texto. `str_split` quebra texto de acordo com algum padrão especificado, sempre em listas de caracteres, então se quisermos apenas o primeiro vetor precisamos puxar com `[[1]]`. Por exemplo:

```r
library(stringr) # funções para lidar com texto
str_split("maçãs, bananas, pêras, uvas", ",")[[1]] # quebre nas vírgulas
```

```
## [1] "maçãs"    " bananas" " pêras"   " uvas"
```

```r
str_split("maçãs, bananas, pêras, uvas", "a")[[1]] # quebre nos a
```

```
## [1] "m"      "çãs, b" "n"      "n"      "s, pêr" "s, uv"  "s"
```

```r
library(stringr)

(cpf_quebrado <- str_split(cpf, "")[[1]]) # quebra em uma lista com um vetor e pega apenas o vetor
```

```
##  [1] "7" "9" "3" "." "2" "2" "4" "." "6" "5" "1" "-" "2" "3"
```

```r
(cpf_apenas_numeros <- keep(cpf_quebrado, ~ .x %in% numeros)) # mantém apenas os números
```

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

```r
length(cpf_apenas_numeros) == 11 # o dado oferecido tem 11 entradas?
```

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

Sabemos que nossa função precisa primeira verificar isso. Também sabemos que é possível que o usuário ofereça apenas números então precisamos lidar com essa possibilidade convertendo o CPF para texto antes de tudo.

Por fim vamos oferecer ao usuário a possibilidade de receber uma mensagem formatada informando o CPF e a validade ou apenas um `TRUE/FLASE`.

```r
validar_cpf <- function(.cpf, formatar = TRUE) {

  cpf_apenas_numeros <- keep(str_split(as.character(.cpf), "")[[1]],
                             ~ .x %in% numeros)

  cpf_numerico <- as.numeric(cpf_apenas_numeros)

  if(length(cpf_apenas_numeros) != 11) {
```

```
    rlang::abort("O CPF informado não tem 11 dígitos")

  }

  primeiro_digito <- cpf_numerico[10]
  segundo_digito <- cpf_numerico[11]

  verificao_primeiro <- (sum(cpf_numerico[1:9]*10:2)*10) %% 11
  verificao_segundo <- (sum(cpf_numerico[1:10]*11:2)*10) %% 11

  digitos_conferem <- primeiro_digito == verificao_primeiro & segundo_digito == verificao_segundo

  if(digitos_conferem == TRUE & formatar == TRUE) {

    return(glue::glue("O CPF {.cpf} é válido"))

  } else if(digitos_conferem == TRUE & formatar == FALSE) {

    return(TRUE)

  } else if(digitos_conferem == FALSE & formatar == TRUE) {

    return(glue::glue("O CPF {.cpf} NÃO é válido"))

  } else if(digitos_conferem == FALSE & formatar == FALSE) {

    return(FALSE)

  }

}

## agora testamos

validar_cpf(cpf)
```

```
## O CPF 793.224.651-23 é válido
```

```
validar_cpf(cpf, formatar = FALSE)
```

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

```
validar_cpf(45330654300)
```

```
## O CPF 45330654300 é válido
```

```
validar_cpf(45330654300, formatar = FALSE)
```

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

```
validar_cpf(11111111122)
```

```
## O CPF 11111111122 NÃO é válido
```

Exercícios

Exercício 1 O R já vem com alguns dados prontos, como a base `mtcars` que carrega algumas medidas para variados modelos de carro.

```
library(tibble)
(dados <- as_tibble(mtcars))
```

```
## # A tibble: 32 x 11
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
##  2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
##  3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
##  4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
##  5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
##  6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
##  7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
##  8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
##  9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
## 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
## # ... with 22 more rows
```

- Tabele a variável `gear` e descubra quantos carros têm quantas marchas
- A variável `mpg` traz o consumo de combustível em milhas por galão. Descubra sua média, desvio-padrão e tire o sumário estatístico com a função `summary`

Exercício 2 Com a função `install.packages` você pode instalar mais pacotes, use-a para instalar a biblioteca `palmerpenguins` e carregue-a com `library`. A base traz medidas de cerca de 300 penguins nas Ilhas Palmer.

```
library(palmerpenguins)
penguins
```

```
## # A tibble: 344 x 8
##    species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g
##    <fct>   <fct>           <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torge~           39.1          18.7               181        3750
##  2 Adelie  Torge~           39.5          17.4               186        3800
##  3 Adelie  Torge~           40.3          18                 195        3250
##  4 Adelie  Torge~           NA            NA                  NA          NA
##  5 Adelie  Torge~           36.7          19.3               193        3450
##  6 Adelie  Torge~           39.3          20.6               190        3650
##  7 Adelie  Torge~           38.9          17.8               181        3625
##  8 Adelie  Torge~           39.2          19.6               195        4675
```

```
##  9 Adelie  Torge~           34.1          18.1              193        3475
## 10 Adelie  Torge~           42            20.2              190        4250
## # ... with 334 more rows, and 2 more variables: sex <fct>, year <int>
```

```
table(penguins$species, penguins$sex)
```

```
##
##             female male
##   Adelie        73   73
##   Chinstrap     34   34
##   Gentoo        58   61
```

Faça as seguintes tabulações:

- Quantos penguins de cada espécie habitam cada ilha
- Quantos penguins em cada ilha foram medidos em cada ano de estudo

Exercício 3 Construa uma função que receba um vetor e devolva a quantidade de entradas vazias do vetor, use a função `is.na` para testar quais entradas são vazias.

Se possível, faça a função retornar uma mensagem de erro caso o usuário dê algum dado que não seja um vetor. A função `rlang::abort` irá te ajudar.

Exercício 4 Armazene o link `https://www.learningcontainer.com/bfd_download/sample-xls-file-for-testing/` em um objeto.

- Armazene em um `tibble`. Use o parâmetro `setclass=` da função `rio::import`.
- A média da variável `Sales`.
- O desvio-padrão da variável `Profit`.
- O sumário estatístico de `COGS`.
- Tabele a variável `Product`.
- Faça um tabelamento cruzado das variáveis `Segment` e `Country`.

Exercício 5

- [*Matemática*] A função `rnorm()` gera números aleatórios com distribuição normal. Já `matrix()` recebe um vetor, um número de linhas e/ou colunas e devolve uma matriz. Algumas funções como `det`, `eigen` e `t` trazem as construções típicas da Álgebra Linear. Por exemplo, se eu quiser uma matriz 2x3 com números tirados de uma normal com média 0 e desvio-padrão 3 e multiplica-la pela sua transposta:

```
(A <- matrix(rnorm(n = 6, sd = 3), # 6 números tirados de uma N(0,3)
             nrow = 2)) # em duas linhas
```

```
##           [,1]       [,2]       [,3]
## [1,] -4.182128 10.0752948  2.0961302
## [2,]  4.453069 -0.1333722 -0.6709311
```

```
A %*% t(A) # %*% é o operador para multiplicar matrizes
```

```
##           [,1]      [,2]
## [1,] 123.39552 -21.37342
## [2,] -21.37342  20.29776
```

Sabendo que `help(funcao)` abre a documentação da função informando os nomes de parametros e seus efeitos sobre o resultado da função:

- Crie uma matriz 8x8 cujas entradas são tiradas de uma normal com média 0.5 e desvio-padrão 2.
- Ache seu determinante
- Ache a soma de seus autovalores