R para Ciência de Dados I

Introdução ao R e RStudio

RStudio

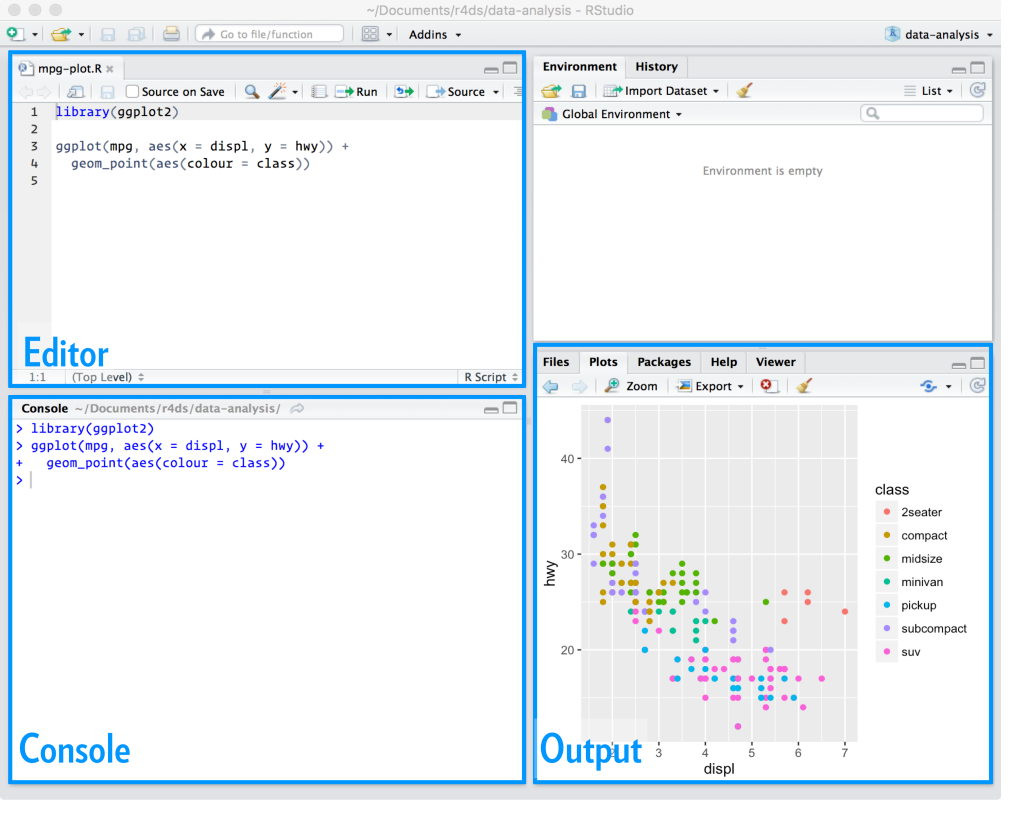
Objetivo desta seção

Você estará pronta(o) para seguir para a seção de **Introdução ao R** quando tiver cumprido os objetivos abaixo.

- Ficar confortável com a utilização do RStudio para programação em R, isto é:
  - souber a diferença entre o R e o RStudio;
  - entender para que serve o Console;
  - entender para que serve o Script.
- Entender a importância de se utilizar os projetos do RStudio para organizar sua análise.

Painéis

Ao abrir o RStudio, você verá 4 quadrantes:



Esses quadrantes representam o **editor**, o **console**, o **environment** e o **output**. Eles vêm nesta ordem, mas você pode organizá-los da forma que preferir acessando a seção *Pane Layout* da opção `Global options...` no menu `Tools`.

O editor e o console são os dois principais painéis do RStudio. Passaremos a maior parte do tempo neles:

- **Editor/Scripts**: é onde escrevemos nossos códigos. Repare que o RStudio colore algumas palavras e símbolos para facilitar a leitura do código.
- **Console**: é onde rodamos o código e recebemos as saídas. **O R vive aqui!**

Os demais painéis são auxiliares. O objetivo deles é facilitar pequenas tarefas que fazem parte tanto da programação quanto da análise de dados:

- **Environment**: painel com todos os objetos criados na sessão.
- **History**: painel com um histórico dos comandos rodados.
- **Files**: mostra os arquivos no diretório de trabalho. É possível navegar entre diretórios.
- **Plots**: painel onde os gráficos serão apresentados.
- **Packages**: apresenta todos os pacotes instalados e carregados.
- **Help**: janela onde a documentação das funções serão apresentadas.
- **Viewer**: painel onde relatórios e dashboards serão apresentados.

Atalhos

Conhecer os atalhos do teclado ajuda bastante quando estamos programando no RStudio. Veja os principais:

- **CTRL+ENTER**: avalia/roda a linha selecionada no script. O atalho mais utilizado.
- **ALT+-**: cria no script um sinal de atribuição (<−). Você o usará o tempo todo.
- **CTRL+SHIFT+M**: (%>% ou |>) operador *pipe*. Guarde esse atalho, será bastante utilizado.
- **CTRL+1**: altera cursor para o script.
- **CTRL+2**: altera cursor para o console.
- **ALT+SHIFT+K**: janela com todos os atalhos disponíveis.

No MacOS, substitua **CTRL** por **command** e **ALT** por **option**.

Projetos

Uma funcionalidade muito importante do RStudio é a possibilidade de criar **projetos**.

Um projeto é uma pasta no seu computador. Nessa pasta, estarão todos os arquivos que você usurá ou criará na sua análise.

A principal razão de utilizarmos projetos é **organização**. Com eles, fica muito mais fácil importar bases de dados para dentro do R, criar análises reprodutíveis e compartilhar o nosso trabalho.

Habitue-se desde a cedo a criar um projeto para cada nova análise que for fazer.

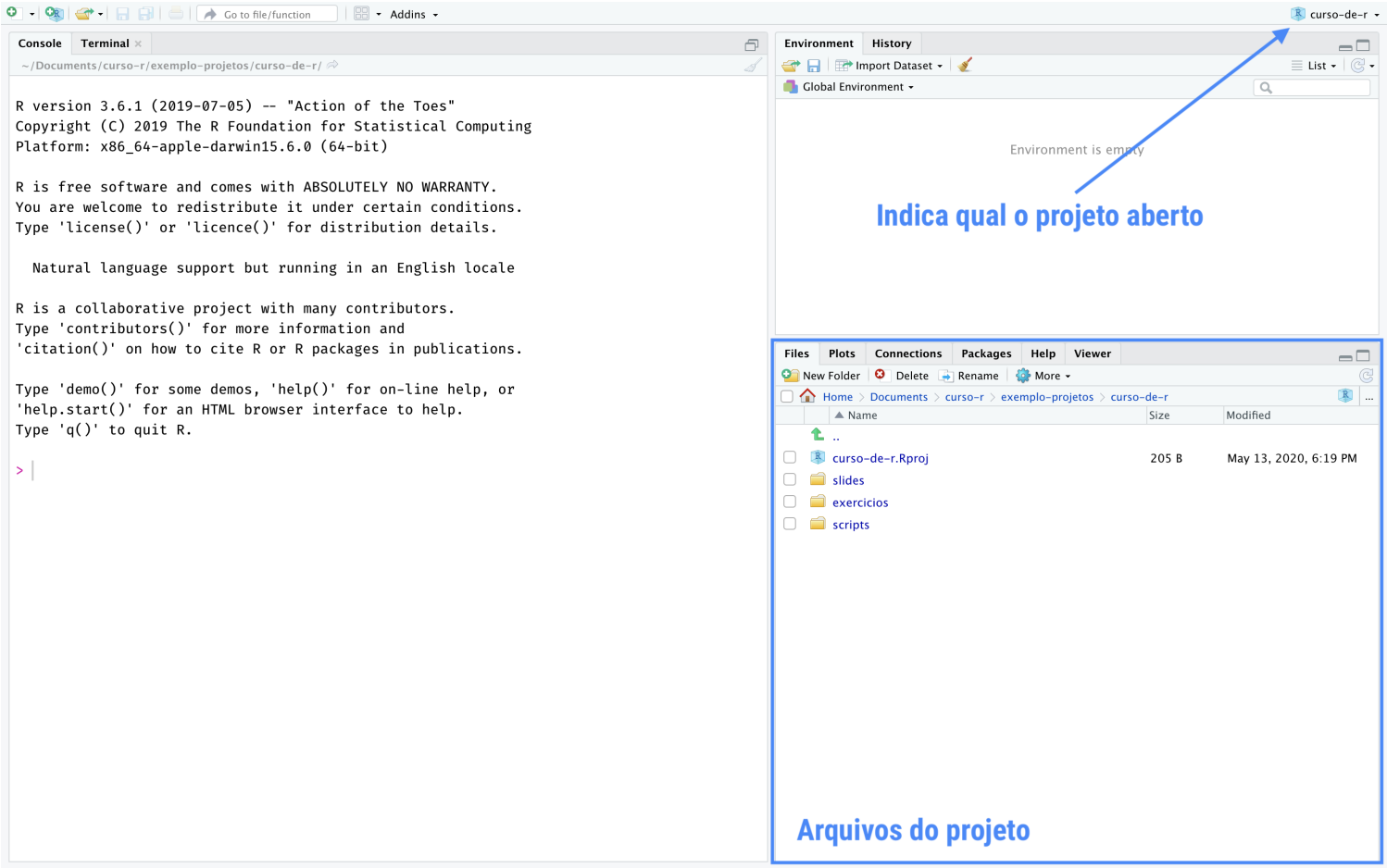
Para criar um projeto, clique em `New Project...` no Menu `File`. Na caixa de diálogo que aparecerá, clique em `New Directory` para criar o projeto em uma nova pasta ou `Existing Directory` para criar em uma pasta existente.

Se você tiver o `Git` instalado, você também pode usar projetos para conectar com repositórios do Github e outras plataformas de desenvolvimento. Para isso, basta clicar em `Version Control`.

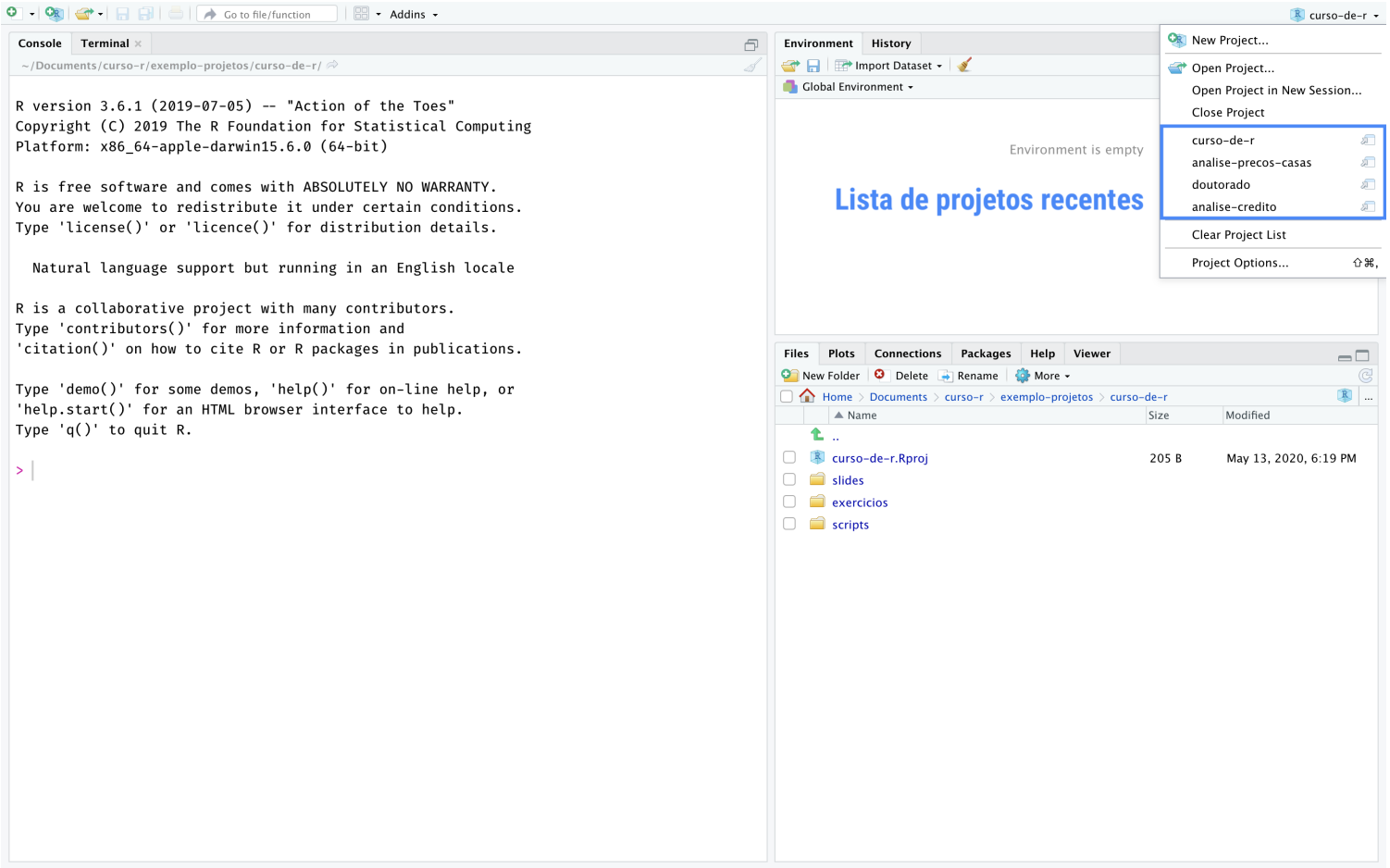
Se você não sabe o que é Git, Github ou versionamento, veja este material da Beatriz Milz ou este post no nosso blog.

Criando um projeto, o RStudio criará na pasta escolhida um arquivo `nome-do-projeto.Rproj`. Você pode usar esse arquivo para iniciar o RStudio já com o respectivo projeto aberto.

RStudio com um projeto aberto

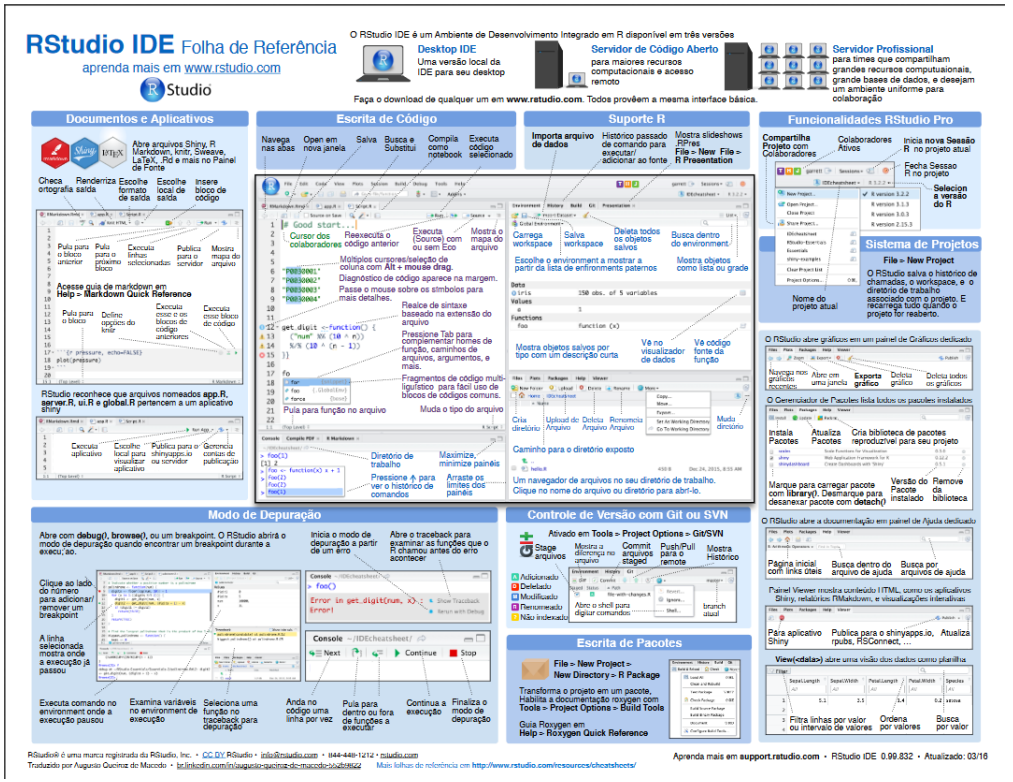


Navegando entre projetos



Cheatsheets

O RStudio tem à disposição algumas *folhas de cola*, as **cheatsheets**. Elas trazem um resumão de como utilizar diversos pacotes e até o próprio RStudio. Para acessá-las, basta clicar no menu `Help` e então em `Cheatsheets`, ou no site da RStudio.



Introdução ao R

Objetos desta seção

Você estará pronta(o) para seguir para a seção de **Importação** quando tiver cumprido os objetivos abaixo.

- Entender o fluxo de escrever e avaliar (rodar) códigos em R.
- Entender o que são objetos, como criá-los e como retornar os valores que eles guardam.
- Endender o que são *data frames* no R.
- Endender o que são funções e como utilizá-las no R.
- Entender o que são pacotes, como instalá-los e como carregá-los.

R como calculadora

O papel do **Console** no R é executar os nossos comandos. Ele avalia o código que passamos para ele e devolve a saída correspondente (se tudo der certo) ou uma mensagem de erro (se o seu código tiver algum problema).

> Para rodar um código, escreva o código no script e, com o cursor em cima da linha que você quer rodar, use o atalho "CTRL+ENTER". Você não precisa selecionar o código, a não ser que queria rodar várias linhas de uma única vez.

Vamos começar com um exemplo de operação comum:

```
1 + 1
```

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

Nesse caso, o nosso comando foi o código `1 + 1` e a saída foi o valor 2.

Tente agora jogar no console a expressão: `2 * 2 - (4 + 4) / 2`.

Repare que as operações e suas precedências são mantidas como na matemática, ou seja, divisão e multiplicação são calculadas antes da adição e subtração. E parênteses nunca são demais!

Pronto! Você já é capaz de pedir ao R para fazer qualquer uma das quatro operações aritméticas básicas.

> **Quando compilamos?** Quem vem de linguagens como o C ou Java espera que seja necessário compilar o código em texto para o código das máquinas (geralmente um código binário). No R, isso não é necessário. O R é uma linguagem de programação dinâmica que interpreta o seu código enquanto você o executa.

Comandos incompletos

Se você digitar um comando incompleto, como `5  +`, e apertar `Enter`, o R mostrará um `+`, o que não tem nada a ver com a adição da matemática. Isso significa que o R está esperando que você enviar **mais** algum código para completar o seu comando. Termine o seu comando ou aperte `Esc` para recomeçar.

```
> 5 -
+
+ 5
[1] 0
```

Erros

Se você digitar um comando que o R não reconhece, ele retornará uma mensagem de erro.

NÃO ENTRE EM PÂNICO!

Ele só está avisando que não conseguiu interpretar o comando. Você pode digitar outro comando normalmente em seguida.

```
> 5 % 2
Error: unexpected input in "5 % 2"
> 5 ^ 2
[1] 25
```

Objetos

O R te permite salvar valores dentro de um **objeto**. Um objeto é um nome que guarda um valor. Para criar um objeto, utilizamos o operador <-.

No exemplo abaixo, salvamos o valor 1 em a. Sempre que avaliarmos o objeto a, o R vai devolver o valor 1.

```
# Salvando `1` em `a`
a <- 1

# Avaliando o objeto `a`
a
```

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

Nomeando objetos

Existem algumas regras para dar nomes aos objetos. A mais importante é: o nome deve começar com uma letra. O nome pode conter números, mas não pode começar com números. Você pode usar pontos `.` e underlines `_` para separar palavras.

```
# Permitido

x <- 1
x1 <- 2
objeto <- 3
meu_objeto <- 4
meu.objeto <- 5

# Não permitido

1x <- 1
_objeto <- 2
meu-objeto <- 3
```

O R **diferencia letras maiúsculas e minúsculas**, isto é, b é considerado um objeto diferente de B. Rode o exemplo abaixo e observe que dois objetos diferentes são criados no **Environment**.

```
b <- 2
B <- 3

b
```

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

```
B
```

```
## [1] 3
```

Data frames

O objeto mais importante para o cientista de dados é, claro, a base de dados. No R, uma base de dados é representa por objetos chamados de *data frames*. Eles são equivalentes a uma tabela do SQL ou uma planilha do Excel.

A principal característica de um *data frame* é possuir linhas e colunas:

```
mtcars
```

```
##                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

O `mtcars` é um *data frame* nativo do R que contém informações sobre diversos modelos de carros. Ele possui 32 linhas e 11 colunas (só estamos vendo as primeiras 10 linhas no slide anterior).

A primeira "coluna" representa apenas o *nome* das linhas (modelo do carro), não é uma coluna da base. Repare que ela não possui um nome, como as outras. Essa estrutura de nome de linha é própria de *data frames* no R. Se exportássemos essa base para o Excel, por exemplo, essa coluna não apareceria.

Se você quiser saber mais sobre o `mtcars`, veja a documentação dele rodando `?mtcars` no **Console**.

Para entender melhor sobre *data frames*, precisamos estudar um pouco sobre classes, vetores e testes lógicos.

Classes

A classe de um objeto é muito importante dentro do R. É a partir dela que as funções e operadores conseguem saber exatamente o que fazer com um objeto.

Por exemplo, podemos somar dois números, mas não conseguimos somar duas letras (texto):

```
1 + 1
```

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

```
"a" + "b"
```

```
## Error in "a" + "b": non-numeric argument to binary operator
```

O operador + verifica que "a" e "b" não são números (ou que a classe deles não é numérica) e devolve uma mensagem de erro informando isso.

Texto

Observe que para criar texto no R, colocamos os caracteres entre aspas. As aspas servem para diferenciar *nomes* (objetos, funções, pacotes) de *textos* (letras e palavras). Os textos são muito comuns em variáveis categóricas.

```
a <- 10

# O objeto `a`, sem aspas
a
```

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

```
# A letra (texto) `a`, com aspas
"a"
```

```
## [1] "a"
```

A classe de um objeto

Para saber a classe de um objeto, basta rodarmos `class(nome-do-objeto)`.

```
x <- 1
class(x)
```

```
## [1] "numeric"
```

```
y <- "a"
class(y)
```

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

```
class(mtcars)
```

```
## [1] "data.frame"
```

Objetos atômicos

As classes mais básicas dentro do R são:

- *numeric*
- *character*
- *logical*

Um objeto de qualquer uma dessas classes é chamado de **objeto atômico**.

Esse nome se deve ao fato de essas classes não se misturarem, isto é, para um objeto ter a classe `numeric`, por exemplo, todos os seus valores precisam ser numéricos.

Mas como atribuir mais de um valor a um mesmo objeto? Para isso, precisamos criar **vetores**.

Vetores

Vetores são estruturas muito importantes dentro R. Em especial, pensando em análise de dados, precisamos estudá-los pois cada coluna de um *data frame* será representada como um vetor.

Vetores são apenas **conjuntos indexados de valores**. Para criá-los, basta colocar os valores separados por vírgulas dentro de um `c()`.

```
vetor1 <- c(1, 5, 3, -10)
vetor2 <- c("a", "b", "c")

vetor1
```

```
## [1]   1   5   3 -10
```

```
vetor2
```

```
## [1] "a" "b" "c"
```

Sequências

Uma maneira fácil de criar um vetor com uma sequência de números é utilizar o operador `:`.

```
# Vetor de 1 a 10
1:10
```

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

```
# Vetor de 10 a 1
10:1
```

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

```
# Vetor de -3 a 3
-3:3
```

```
## [1] -3 -2 -1  0  1  2  3
```

Subsetting

Quando dizemos que vetores são conjuntos *indexados*, isso quer dizer que cada valor dentro de um vetor tem uma **posição**. Essa posição é dada pela ordem em que os elementos foram colocados no momento em que o vetor foi criado. Isso nos permite acessar individualmente cada valor de um vetor.

Para isso, colocamos o índice do valor que queremos acessar dentro de colchetes `[]`.

```
vetor <- c("a", "b", "c", "d")

vetor[1]
```

```
## [1] "a"
```

```
vetor[4]
```

```
## [1] "d"
```

Você também pode colocar um conjunto de índices dentro dos colchetes, para pegar os valores contidos nessas posições:

```
vetor[c(2, 3)]
```

```
## [1] "b" "c"
```

```
vetor[c(1, 2, 4)]
```

```
## [1] "a" "b" "d"
```

Essas operações são conhecidas como *subsetting*, pois estamos pegando subconjuntos de valores de um vetor.

Classe de um vetor

Um vetor só pode guardar um tipo de objeto e ele terá sempre a mesma classe dos objetos que guarda. Para saber a classe de um vetor, rodamos `class(nome-do-vetor)`.

```
vetor1 <- c(1, 5, 3, -10)
vetor2 <- c("a", "b", "c")

class(vetor1)
```

```
## [1] "numeric"
```

```
class(vetor2)
```

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

Coerção

Se tentarmos misturar duas classes, o R vai apresentar o comportamento conhecido como **coerção**.

```
vetor <- c(1, 2, "a")

vetor
```

```
## [1] "1" "2" "a"
```

```
class(vetor)
```

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

Veja que todos os elementos do vetor se transformaram em texto.

Como um vetor só pode ter uma classe de objeto dentro dele, classes mais fracas serão sempre reprimidas pelas classes mais fortes. Como regra de bolso: caracteres serão sempre a classe mais forte. Então, sempre que você misturar números e texto em um vetor, os números virarão texto.

Operações com vetores

De forma bastante intuitiva, você pode fazer operações com vetores.

```
vetor <- c(0, 5, 20, -3)
vetor + 1
```

```
## [1]  1  6 21 -2
```

Ao rodarmos `vetor1 + 1`, o R subtrai `1` de cada um dos elementos do vetor. O mesmo acontece com qualquer outra operação aritmética.

Vetorização

Você também pode fazer operações que envolvem mais de um vetor:

```
vetor1 <- c(1, 2, 3)
vetor2 <- c(10, 20, 30)

vetor1  + vetor2
```

```
## [1] 11 22 33
```

Neste caso, o R irá alinhar os dois vetores e somar elemento a elemento. Esse tipo de comportamento é chamado de **vetorização**.

Recliclagem

Isso pode ficar um pouco confuso quando os dois vetores não possuem o mesmo tamanho.

```
vetor1 <- c(1, 2)
vetor2 <- c(10, 20, 30, 40)

vetor1 + vetor2
```

```
## [1] 11 22 31 42
```

Embora estejamos somando dois vetores de tamanho diferentes, o R não devolve um erro (o que parecia ser a resposta mais intuitiva). O R alinhou os dois vetores e, como eles não possuíam o mesmo tamanho, o primeiro foi repetido para ficar do mesmo tamanho do segundo. É como se o primeiro vetor fosse na verdade `c(1, 2, 1, 2)`.

Esse comportamento é chamado de **reciclagem**.

Embora contra-intuitiva, a reciclagem é muito útil no R graças a um caso particular muito importante.

Quando somamos `vetor + 1` no nosso primeiro exemplo, o que o R está fazendo por trás é transformando o `1` em `c(1, 1, 1, 1)` e realizando a soma vetorizada `c(0, 5, 20, -3) + c(1, 1, 1, 1)`. Isso porque o número `1` é um vetor de tamanho 1, isto é, `1` é igual a `c(1)`.

Usaremos esse comportamento no R o tempo todo. É muito importante a reciclagem para termos certeza de que o R está fazendo exatamente aquilo que gostaríamos que ele fizesse.

Um outro caso interessante de reciclagem é quando o comprimento dos vetores não são múltiplos um do outro.

```
vetor1 <- c(1, 2, 3)
vetor2 <- c(10, 20, 30, 40, 50)

vetor1 + vetor2
```

```
## Warning in vetor1 + vetor2: longer object length is not a multiple of
## shorter object length
```

```
## [1] 11 22 33 41 52
```

Neste caso, duas coisas aconteceram:

1. O R realizou a conta, repetindo cada valor do primeiro vetor até que os dois tenham o mesmo tamanho. No fundo, a operação realizada foi `c(1, 2, 3, 1, 2) + c(10, 20, 30, 40, 50)`.

2. Como essa operação é ainda menos intuitiva e raramente desejada, o R devolveu um aviso dizendo que o comprimento do primeiro vetor maior não é um múltiplo do comprimento do vetor menor.

Operações lógicas

Uma operação lógica é um teste que retorna **verdadeiro** ou **falso**. No R (e em outras linguagens de programação), esses valores dois valores recebem uma classe especial: `logical`.

O verdadeiro no R vai ser representado pelo valor `TRUE` e o falso pelo valor `FALSE`. Esses nomes no R são **reservados**, isto é, você não pode chamar nenhum objeto de TRUE ou FALSE.

```
TRUE <- 1
## Error in TRUE <- 1 : invalid (do_set) left-hand side to assignment
```

Valores lógicos

Checando a classe desses valores, vemos que são lógicos (também conhecidos como valores binários ou booleanos). Eles são os únicos possíveis valores dessa classe.

```
class(TRUE)
```

```
## [1] "logical"
```

```
class(FALSE)
```

```
## [1] "logical"
```

Agora que conhecemos o `TRUE` e `FALSE`, podemos explorar os teste lógicos.

Igualdades

Começando pela igualdade: vamos testar se um valor é igual ao outro. Para isso, usamos o operador ==.

```
# Testes com resultado verdadeiro
1 == 1
```

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

```
"a" == "a"
```

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

```
# Testes com resultado falso
1 == 2
```

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

```
"a" == "b"
```

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

Diferenças

Também podemos testar se dois valores são diferentes. Para isso, usamos o operador `!=`.

```
# Testes com resultado falso
1 != 1
```

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

```
"a" != "a"
```

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

```
# Testes com resultado verdadeiro
1 != 2
```

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

```
"a" != "b"
```

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

Desigualdades

Para comparar se um valor é maior que outro, temos à disposição 4 operadores:

```
# Maior
3 > 3
```

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

```
3 > 2
```

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

```
# Maior ou igual
3 > 4
```

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

```
3 >= 3
```

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

```r
# Menor
3 < 3
```

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

```r
3 < 4
```

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

```r
# Menor ou igual
3 < 2
```

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

```r
3 <= 3
```

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

Pertence

Um outro operador muito útil é o `%in%`. Com ele, podemos verificar se um valor está dentro de um conjunto de valores (vetor).

```
3 %in% c(1, 2, 3)
```

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

```
"a" %in% c("b", "c")
```

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

Filtros

Os testes lógicos fazem parte de uma operação muito comum na manipulação de base de dados: os **filtros**. No Excel, por exemplo, quando você filtra uma planilha, o que está sendo feito por trás é um teste lógico.

Falamos anteriormente que cada coluna das nossas bases de dados será representada dentro do R como um vetor. O comportamento que explica a importância dos testes lógicos na hora de filtrar uma base está ilustrado abaixo:

```
minha_coluna <- c(1, 3, 0, 10, -1, 5, 20)

minha_coluna > 3
```

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

```
minha_coluna[minha_coluna > 3]
```

```
## [1] 10  5 20
```

Muitas coisas aconteceram no código anterior, vamos por partes.

Primeiro, na operação `minha_coluna > 3` o R fez um excelente uso do comportamento de reciclagem. No fundo, o que ele fez foi transformar (reciclar) o valor `3` no vetor `c(3, 3, 3, 3, 3, 3, 3)` e testar se `c(1, 3, 0, 10, -1, 5, 20) > c(3, 3, 3, 3, 3, 3, 3)`.

Como os operadores lógicos também são vetorizados (fazem operações elemento a elemento), os testes realizados foram `1 > 3`, `3 > 3`, `0 > 3`, `10 > 3`, `-1 > 3`, `5 > 3` e, finalmente, `20 > 3`. Cada um desses testes tem o seu próprio resultado. Por isso a saída de `minha_coluna > 3` é um vetor de verdadeiros e falsos, respectivos a cada um desses 7 testes.

A segunda operação traz a grande novidade aqui: podemos usar os valores `TRUE` e `FALSE` para selecionar elementos de um vetor!

A regra é a seguinte: **retornar** as posições que receberem `TRUE`, **não retornar** as posições que receberem `FALSE`.

Portanto, a segunda operação é equivalente a:

```
minha_coluna[c(FALSE, FALSE, FALSE, TRUE, FALSE, TRUE, TRUE)]
```

```
## [1] 10  5 20
```

O vetor lógico filtra o vetor `minha_coluna`, retornando apenas os valores maiores que 3, já que foi esse o teste lógico que fizemos.

Essa é a *mágica* que acontece por trás de filtros no R. Na prática, não precisaremos usar colchetes, não lembraremos da reciclagem e nem veremos a cara dos `TRUE` e `FALSE`. Mas conhecer esse processo é muito importante, principalmente para encontrar problemas de código ou de base.

Mais sobre data frames

Chegou a hora de usarmos tudo o que aprendemos na seção anterior para exploramos ao máximo o nosso objeto favorito: o *data frame*.

Para isso, continuaremos a usar o `mtcars`.

```
mtcars
```

```
##                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

Acessando as colunas

Lembrando que cada coluna de um *data frame* é um vetor, podemos usar o operador `$` para acessar cada uma de suas colunas.

```
mtcars$mpg
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

```
mtcars$cyl
```

```
##  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

```
mtcars$wt
```

```
##  [1] 2.620 2.875 2.320 3.215 3.440 3.460 3.570 3.190 3.150 3.440 3.440
## [12] 4.070 3.730 3.780 5.250 5.424 5.345 2.200 1.615 1.835 2.465 3.520
## [23] 3.435 3.840 3.845 1.935 2.140 1.513 3.170 2.770 3.570 2.780
```

Dimensões

A classe *data frame* possui uma característica especial: seus objetos possuem duas **dimensões**.

```
class(mtcars)
```

```
## [1] "data.frame"
```

```
dim(mtcars)
```

```
## [1] 32 11
```

O resultado do código `dim(mtcars)` nos diz que a primeira dimensão tem comprimento 32 e a segunda dimensão tem comprimento 11. Em outras palavras: a base `mtcars` tem 32 linhas e 11 colunas.

Subsetting

Ter duas dimensões significa que devemos usar dois índices para acessar os valores de um *data frame* (fazer *subsetting*). Para isso, ainda usamos o colchete, mas agora com dois argumentos: `[linha, coluna]`.

```
mtcars[2, 3]
```

```
## [1] 160
```

O código acima está nos devolvendo o valor presente na segunda linha da terceira coluna da base `mtcars`.

Também podemos pegar todos as linhas de uma coluna ou todas as colunas de uma linha deixando um dos argumentos vazio:

```
# Todas as linhas da coluna 1
mtcars[,1]
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

```
# Todas as colunas da linha 1
mtcars[1,]
```

```
##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Mazda RX4  21   6  160 110  3.9 2.62 16.46  0  1    4    4
```

Selecionando colunas

Podemos usar o *subsetting* para selecionar colunas:

```
mtcars[, c(1, 2)]
```

```
##                    mpg cyl
## Mazda RX4         21.0   6
## Mazda RX4 Wag     21.0   6
## Datsun 710        22.8   4
## Hornet 4 Drive    21.4   6
## Hornet Sportabout 18.7   8
## Valiant           18.1   6
## Duster 360        14.3   8
## Merc 240D         24.4   4
## Merc 230          22.8   4
## Merc 280          19.2   6
```

```
mtcars[, c("mpg", "am")]
```

```
##                    mpg am
## Mazda RX4         21.0  1
## Mazda RX4 Wag     21.0  1
## Datsun 710        22.8  1
## Hornet 4 Drive    21.4  0
## Hornet Sportabout 18.7  0
## Valiant           18.1  0
## Duster 360        14.3  0
## Merc 240D         24.4  0
## Merc 230          22.8  0
## Merc 280          19.2  0
```

Nos dois exemplos, exibimos apenas as 5 primeiras linhas do *data frame*.

Filtrando colunas

Também podemos usar o *subsetting* para filtrar colunas:

```
mtcars$cyl == 4
```

```
##  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
## [23] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE
```

```
mtcars[mtcars$cyl == 4, ]
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

O código `mtcars$cyl == 4` nos diz em quais linhas estão os carros com 4 cilindros. Quando usamos o vetor de `TRUE` e `FALSE` resultante dentro do *subsetting* das linhas em `mtcars[mtcars$cyl == 4, ]`, o R nos devolve todos as colunas dos carros com 4 cilindros. A regra é a seguinte: linha com `TRUE` é retornada, linha com `FALSE` não.

Outro exemplo:

```
mtcars[mtcars$mpg > 25, ]
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
```

Funções

Enquanto objetos são *nomes* que guardam *valores*, funções no R são *nomes* que guardam um **código de R**. A ideia é a seguinte: sempre que você rodar uma função, o código que ela guarda será executado e um resultado nos será devolvido.

Funções são tão comuns e intuitivas (provavelmente você já usou funções no Excel), que mesmo sem definir o que elas são, nós já utilizamos funções nas seções anteriores:

- a função `c()` foi utilizada para criar vetores;
- a função `class()` foi utilizada para descobrir a classe de um objeto.
- a função `dim()` foi utilizada para verificarmos a dimensão de um *data frame*.

Argumentos

Diferentemente dos objetos, as funções podem receber **argumentos**. Argumentos são os valores que colocamos dentro dos parênteses e que as funções precisam para funcionar (calcular algum resultado). Por exemplo, a função `c()` precisa saber quais são os valores que formarão o vetor que ela irá criar.

```
c(1, 3, 5)
```

```
## [1] 1 3 5
```

Nesse caso, os valores 1, 3 e 5 são os argumentos da função `c()`. **Os argumentos de uma função são sempre separados por vírgulas**.

Funções no R têm personalidade. Cada uma pode funcionar de um jeito diferente das demais, mesmo quando fazem tarefas parecidas. Por exemplo, vejamos a função `sum()`.

```
sum(1, 3)
```

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

Como você deve ter percebido, essa função retorna a soma de seus argumentos. Também podemos passar um vetor como argumento, e ela retornará a soma dos elementos do vetor.

```
sum(c(1, 3))
```

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

Já a função `mean()`, que calcula a média de um conjunto de valores, exige que você passe valores na forma de um vetor:

```
# Só vai considerar o primeiro número na média
mean(1, 3)
```

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

```
# Considera todos os valores dentro do vetor na média
mean(c(1, 3))
```

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

Os argumentos das funções também têm nomes, que podemos ou não usar na hora de usar uma função. Veja por exemplo a função `seq()`.

```
seq(from = 4, to = 10, by = 2)
```

```
## [1]  4  6  8 10
```

Entre outros argumentos, ela possui os argumentos `from=`, `to=` e `by=`. O que ela faz é criar uma sequência (vetor) de `by` em `by` que começa em `from` e termina em `to`. No exemplo, criamos uma função de 2 em 2 que começa em 4 e termina em 10.

Também poderíamos usar a mesma função sem colocar o nome dos argumentos:

```
seq(4, 10, 2)
```

```
## [1]  4  6  8 10
```

Para utilizar a função sem escrever o nome dos argumentos, você precisa colocar os valores na ordem em que os argumentos aparecem. E se você olhar a documentação da função `seq()`, fazendo `help(seq)`, verá que a ordem dos argumentos é justamente `from=`, `to=` e `by=`.

Escrevendo o nome dos argumentos, não há problema em alterar a ordem dos argumentos:

```
seq(by = 2, to = 10, from = 4)
```

```
## [1]  4  6  8 10
```

Mas se especificar os argumentos, a ordem importa. Veja que o resultado será diferente.

```
seq(2, 10, 4)
```

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

Vocabulário

A seguir, apresentamos algumas funções nativas do R úteis para trabalhar com *data frames* :

- `head()` - Mostra as primeiras 6 linhas.
- `tail()` - Mostra as últimas 6 linhas.
- `dim()` - Número de linhas e de colunas.
- `names()` - Os nomes das colunas (variáveis).
- `str()` - Estrutura do *data frame*. Mostra, entre outras coisas, as classes de cada coluna.
- `cbind()` - Acopla duas tabelas lado a lado.
- `rbind()` - Empilha duas tabelas.

Criando a sua própria função

Além de usar funções já prontas, você pode criar a sua própria função. A sintaxe é a seguinte:

```
nome_da_funcao <- function(argumento_1, argumento_2) {

  # Código que a função irá executar

}
```

Repare que `function` é um nome reservado no R, isto é, você não pode criar um objeto com esse nome.

Um exemplo: vamos criar uma função que soma dois números.

```
minha_soma <- function(x, y) {
  soma <- x + y

  soma  # resultado retornado
}
```

Essa função tem os seguintes componentes:

- `minha_soma`: nome da função
- `x` e `y`: argumentos da função
- `soma <- x + y`: operação que a função executa
- `soma`: valor retornado pela função

Após rodarmos o código de criar a função, podemos utilizá-la como qualquer outra função do R.

```
minha_soma(2, 2)
```

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

O objeto `soma` só existe *dentro da função*, isto é, além de ele não ser colocado no seu *environment*, ele só existirá na memória (RAM) enquanto o R estiver executando a função. Depois disso, ele será apagado. O mesmo vale para os argumentos x e y.

O valor retornado pela função representa o resultado que receberemos ao utilizá-la. Por padrão, **a função retornará sempre a última linha de código que existir dentro dela**. No nosso exemplo, a função retorna o valor contido no objeto `soma`, pois é isso que fazemos na última linha de código da função.

Repare que se atribuirmos o resultado a um objeto, ele não será mostrado no console:

```
resultado <- minha_soma(3, 3)

# Para ver o resultado, rodamos o objeto `resultado`
resultado
```

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

Agora, o que acontece se a última linha da função não devolver um objeto? Veja:

```
minha_nova_soma <- function(x, y) {
  soma <- x + y
}
```

A função `minha_nova_soma()` apenas cria o objeto `soma`, sem retorná-lo como na função `minha_soma()`. Se utilizarmos essa nova função, nenhum valor é devolvido no console:

```
minha_nova_soma(1, 1)
```

No entanto, a última linha da função agora é a atribuição `soma <- x + y` e esse será o "resultado retornado". Assim, podemos visualizar o resultado da função fazendo:

```
resultado <- minha_nova_soma(1, 1)

resultado
```

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

É como se, por trás das cortinas, o R estivesse fazendo `resultado <- soma <- x + y`, mas apenas o objeto `resultado` continua existindo, já que os objetos `soma`, `x` e `y` são descartados após a função ser executada.

Pacotes

Um pacote no R é um conjunto de funções que visam resolver um problema em específico. O R já vem com alguns pacotes instalados. Geralmente chamamos esses pacotes de *base R*.

Mas a força do R está na gigantesca variedade de pacotes desenvolvidos pela comunidade, em especial, pelos criadores do `tidyverse`.

Instalando e carregando pacotes

Para instalar um pacote, usamos a função `install.packages`.

```
# Instalando um pacote
install.packages("tidyverse")

# Instalando vários pacotes de uma vez
install.packages(c("tidyverse", "rmarkdown", "devtools"))
```

Para usar as funções de um pacote, precisamos carregá-lo. Fazemos isso usando a função `library()`.

```
library(tidyverse)
```

Instale uma vez, carregue várias vezes!

Tidyverse

O `tidyverse` é uma coleção de pacotes de R desenvolvidos para Ciência de Dados.

Os pacotes do `tidyverse` compartilham uma mesma filosofia, gramática e estrutura de dados.

Isso significa que:

- As funções, na medida do possível, funcionam da mesma forma. Aprender uma função ajuda a aprender usar outras.
- As funções do mesmo pacote e de pacotes diferente se comunicam umas com as outras.
- Existe uma versão retrabalhada do *data frame* e todos os pacotes giram em torno dessa nova classe.