

Ciência de Dados em R

Curso-R

Última atualização: 14/07/2021

Contents

Sobre	5
1 Instalação	11
1.1 Instalação do R	11
1.2 Instalação do RStudio	13
1.3 Instalação de softwares adicionais	13
2 RStudio	25
2.1 Telas	25
2.2 Atalhos	27
2.3 Projetos	28
2.4 Git e versionamento	33
2.5 Cheatsheets	34
2.6 Addins	36
2.7 Snippets	36
3 R Básico	39
3.1 Pedindo Ajuda	39
3.2 R como calculadora	43
3.3 Objetos e funções	45
3.4 Data frames	48
3.5 Classes	49
3.6 Vetores	52
3.7 Testes lógicos	56

3.8	Valores especiais	61
3.9	Listas	64
3.10	Mais sobre data frames	67
3.11	Mais sobre funções	74
3.12	Controle de Fluxo	79
3.13	Outros tópicos	86
4	Pacotes	97
4.1	Instalação de pacotes	97
4.2	Tidyverse	99
5	Importação	105
5.1	Caminhos	105
5.2	O pacote readr	106
5.3	Os pacotes readxl e writexl	115
5.4	haven	118
6	Pipe	121
6.1	O operador pipe	122
6.2	Outros operadores	125
7	Manipulação	129
7.1	Tibbles	131
7.2	O pacote dplyr	135
7.3	O pacote tidyr	170
7.4	O pacote stringr	182
7.5	O pacote lubridate	193
7.6	O pacoteforcats	199
8	Visualização	217
8.1	O pacote ggplot2	217
8.2	Extensões do pacote ggplot2	259

Chapter 1

Instalação

Nesta seção, abordaremos como instalar o R e o RStudio no Linux e no Windows. Também discutiremos sobre a instalação de pacotes no R.

1.1 Instalação do R

A instalação padrão do R é feita a partir do CRAN, uma rede de servidores espalhada pelo mundo que armazena versões idênticas e atualizadas de códigos e documentações para o R.

Sempre que for instalar algo do CRAN, utilize o servidor (*mirror*) mais próximo de você.

1.1.1 No Windows

Para instalar o R no Windows, siga os seguintes passos:

1. Acesse o CRAN: <https://www.r-project.org/>
2. No menu à esquerda, encontre a opção **Download** e clique em **CRAN**.
3. Escolha a opção de servidor (*mirror*) mais próxima de você.
4. Clique na opção **base**.
5. Na nova página, clique em **Download R x.x.x for Windows*, sendo x.x.x o número da versão que será baixada. Se você teve algum problema com o download, tente escolher outro servidor no passo 3.

6. Feito o download, clique duas vezes no arquivo baixado e siga as instruções para instalação.

Na etapa de escolher a pasta de destino da instalação, se você escolher um local que não esteja dentro da sua pasta de usuário, você precisará de acesso de administrador. Se escolher uma pasta dentro da sua paste de usuário, não precisará.

1.1.2 No Linux

Como a instalação no Linux depende da distribuição utilizada e, em geral, pessoas que utilizam Linux são mais experientes, vamos informar apenas as coordenadas até as instruções/arquivos. Se você tiver alguma dificuldade durante o processo, por favor envie a sua dúvida para a nossa comunidade.

Faremos o possível para ajudar.

1. Acesse o CRAN: <https://cran.r-project.org/>
2. No menu à esquerda, encontre a opção **Download** e clique em **CRAN**.
3. Escolha a opção de servidor (*mirror*) mais próxima de você.
4. Clique em *Download R for Linux*.
5. Clique no link referente à distribuição que você utiliza.
6. Siga as instruções contidas na página para instalar o R. Se você teve algum problema com o download, tente escolher outro servidor no passo 3.

1.1.3 No MacOS

Para instalar o R no MacOS, siga os seguintes passos:

1. Acesse o CRAN: <https://www.r-project.org/>
2. No menu à esquerda, encontre a opção **Download** e clique em **CRAN**.
3. Escolha a opção de servidor (*mirror*) mais próxima de você.
4. Na nova página, clique em **Download R for (Mac) OS X*.
5. Clique na versão do R que você quer baixar (geralmente queremos baixar a mais recente). O objetivo aqui é baixar um arquivo do tipo “R-x.x.x.pkg”, sendo x.x.x o número da versão que vamos instalar. Se você teve algum problema com o download, tente escolher outro servidor no passo 3.
6. Feito o download, clique duas vezes no arquivo baixado e siga as instruções para instalação.

1.2 Instalação do RStudio

Agora vamos instalar a versão *open source* do RStudio, a IDE que utilizaremos para escrever e executar códigos em R.

Para instalar o RStudio no Windows, siga os seguintes passos:

1. Acesse a página de downloads da RStudio: <https://rstudio.com/products/rstudio/download/#download>
 - Se você tiver acesso administrador, baixe a versão referente ao seu sistema operacional que está na lista de *All Installers*.
 - Se você não tiver acesso de administrador, baixe a versão referente ao seu sistema operacional que está na lista de *Zip/Tarballs*.

Instalando se você for administrador

2. Clique duas vezes no arquivo que você baixou da página do RStudio e siga as instruções de instalação.

Instalação se você não for administrador

2. Descompacte o arquivo baixado no passo anterior. Geralmente isso pode ser feito clicando no arquivo compactado com o botão direito do mouse e clicando na opção *descompactar* ou *extrair*.
3. Após a descompactação ter sido finalizada, você terá uma pasta chamada: **RStudio-x.x.x**, em que x.x.x é o número da versão baixada. Abra essa pasta e entre na subpasta com nome **bin**.
4. Procure pelo arquivo chamado **rstudio** e clique duas vezes. Isso abrirá o RStudio. Recomendamos fixar o programa na barra de tarefas para não precisar repetir essa etapa sempre que for abrir o programa.

Observação: se você excluir a pasta que extraímos, o RStudio irá parar de funcionar.

1.3 Instalação de softwares adicionais

Os softwares descritos anteriormente são suficientes para realizar as tarefas básicas em R. Porém, principalmente quando tratamos de relatórios com R, alguns softwares adicionais podem ser necessários em casos específicos! A seguir citamos alguns destes softwares, em quais casos eles são necessários e/ou úteis, e instruções sobre como instalá-los.

1.3.1 LaTeX

O LaTeX é um software para renderização de documentos PDF e também uma linguagem de programação. É bastante usada na academia para produção de artigos científicos.

No R, o LaTeX aparece quando queremos gerar relatórios e apresentações em PDF, a partir de um documento RMarkdown. Na prática, o RMarkdown é transformado para LaTeX através do software *Pandoc*¹, que por sua vez é processado para gerar o PDF. Portanto, para gerar um relatório em PDF através do RMarkdown, precisamos ter alguma ferramenta de LaTeX instalada.

A recomendação das pessoas que desenvolvem o RMarkdown é a utilização do *tinytex*, uma distribuição leve do LaTeX.

```
install.packages('tinytex')
tinytex::install_tinytex() # instalar o TinyTeX
```

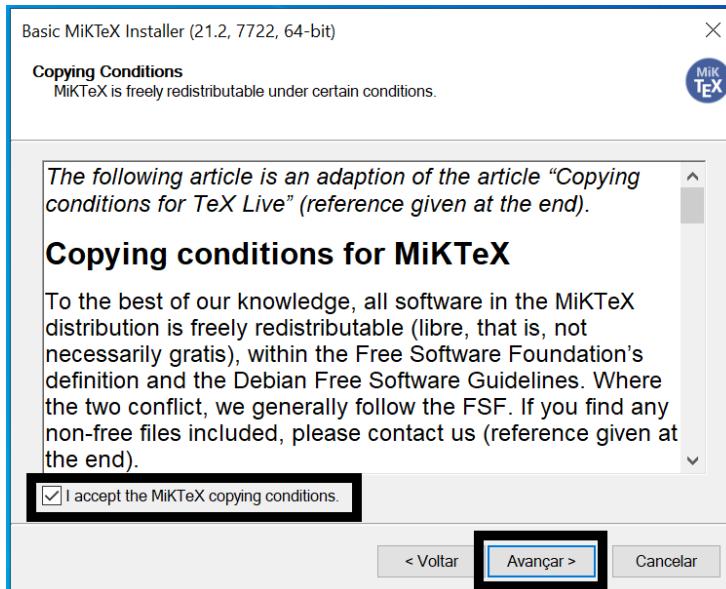
Caso você encontre algum problema para instalar o TinyTeX, também é possível instalar outros editores de LaTeX. A seguir estão apresentadas algumas opções!

1.3.1.1 No Windows

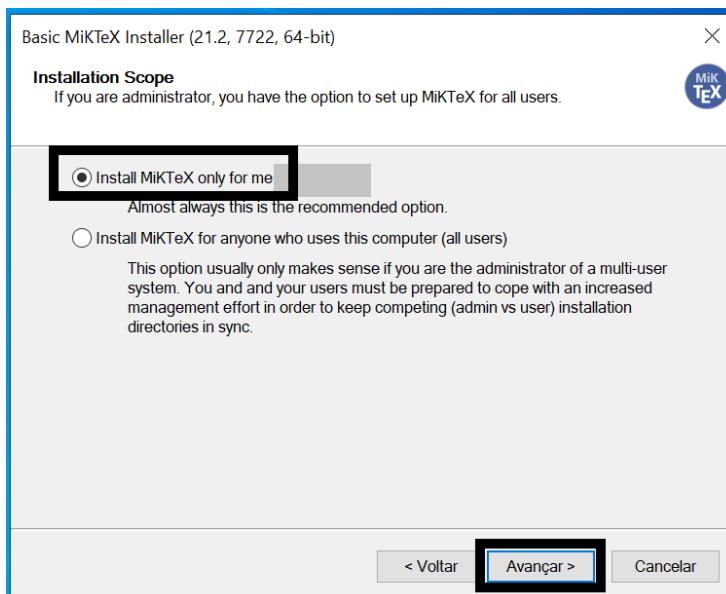
Uma distribuição de LaTeX que apresenta versão para Windows é o MiKTeX.

1. Para instalá-lo, primeiramente faça o download do arquivo .exe através deste link, escolhendo a versão de instalação para Windows.
2. Abra o arquivo baixado. Uma janela irá abrir, e a primeira etapa de instalação é a “Copying Conditions”. Nesta etapa você deve selecionar a opção “**I accept the MiKTeX copying conditions**” e clicar em “Avançar”.

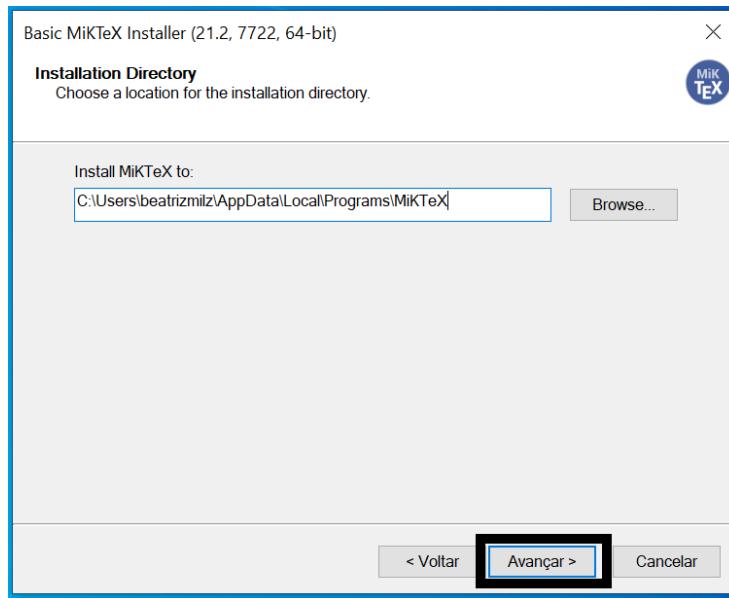
¹O Pandoc já vem dentro do próprio RStudio, então não é necessário instalar este *software*.



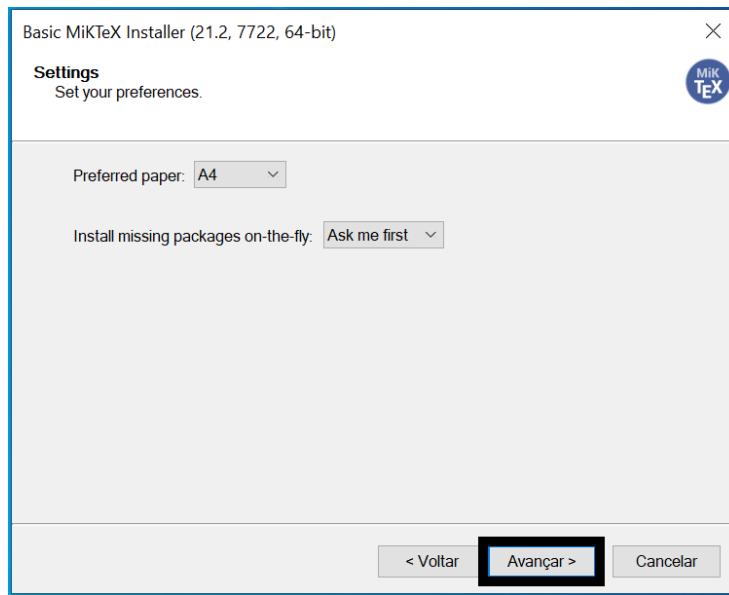
3. A próxima etapa da instalação, chamada de “Installation Scope”, você deve escolher em qual usuário o software deve ser instalado. Selecione a primeira opção (“Install MiKTeX only for me.”) para que o MiKTeX seja instalado apenas no seu usuário atual no computador. Essa é a opção recomendada. Então clique em “Avançar”.



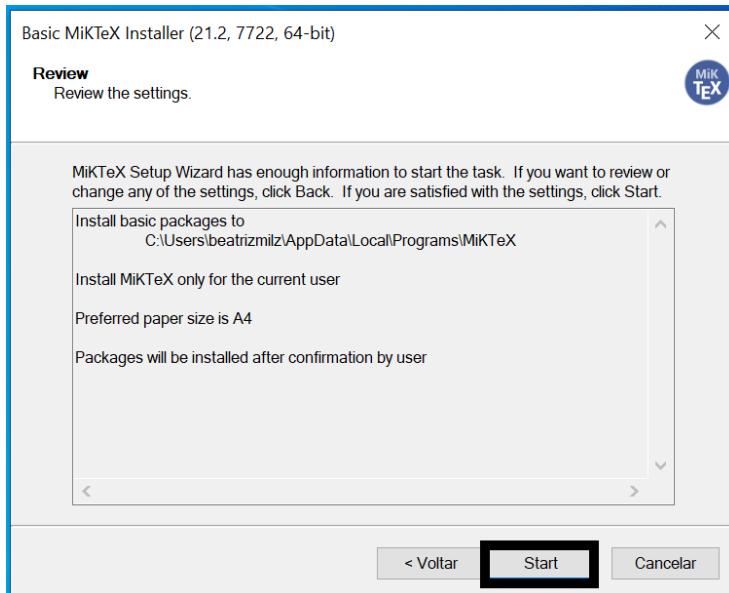
4. Na etapa “Installation Directory”, escolha o diretório em que o MiKTeX será instalado. Caso tenha dúvidas sobre isso, não altere nada e utilize o caminho padrão de instalação. Então clique em “Avançar”.



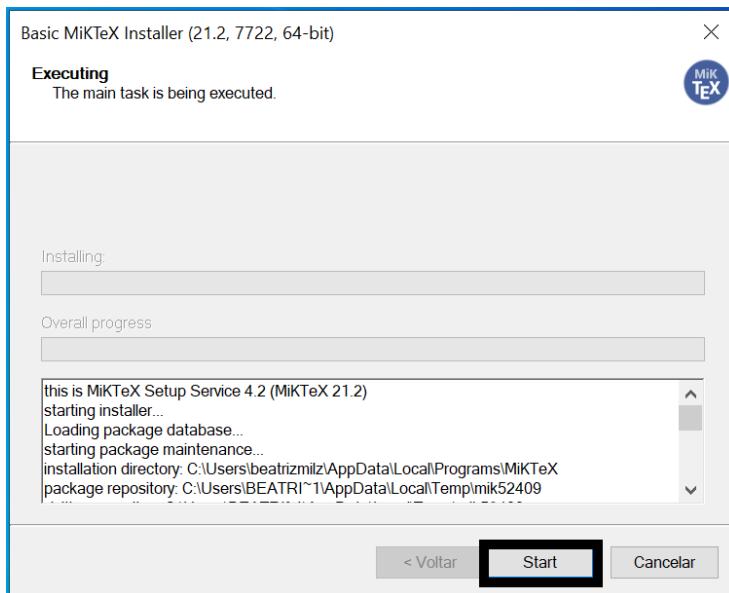
5. Na etapa “Settings”, utilize as configurações padrão e clique em “Avançar”.



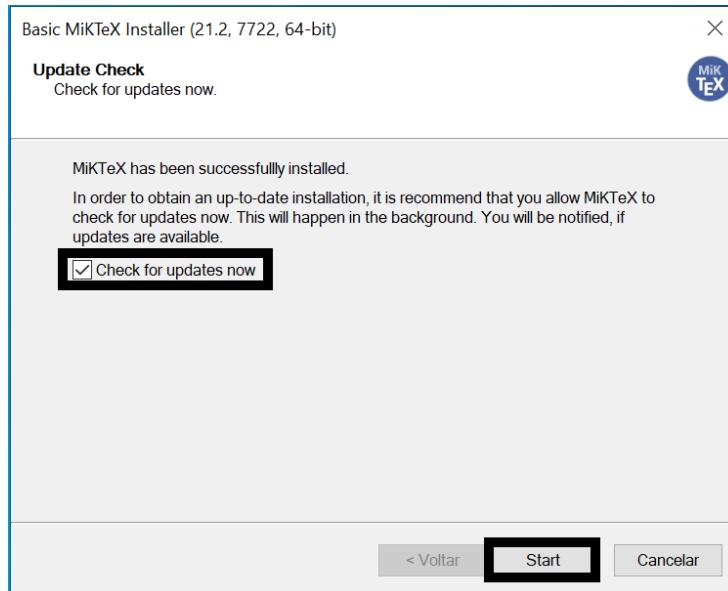
6. Na etapa “Review”, é apresentado as configurações escolhidas anteriormente. Clique em **Start** para que a instalação seja começada.



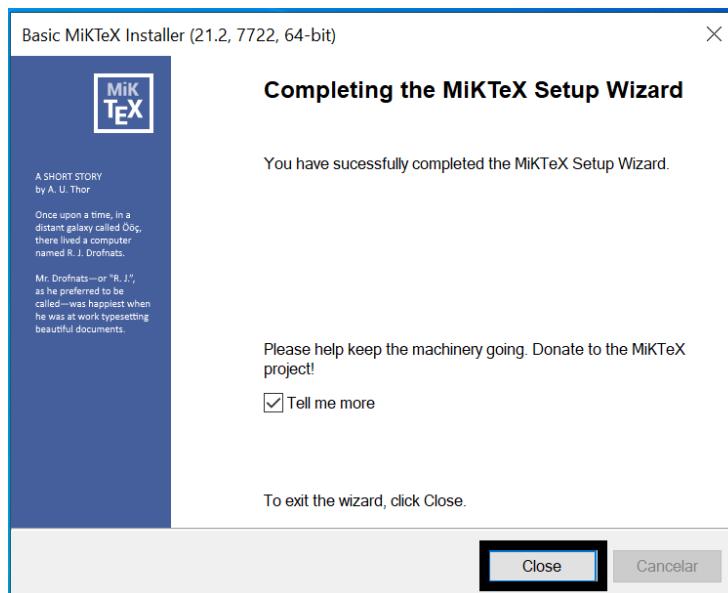
7. O processo de instalação será realizado, e pode demorar alguns minutos. Duas barras verdes aparecerão para demonstrar a porcentagem da instalação já realizada. Quando essa etapa terminar, as barras ficarão cinza e o você deverá clicar no botão **Start**.



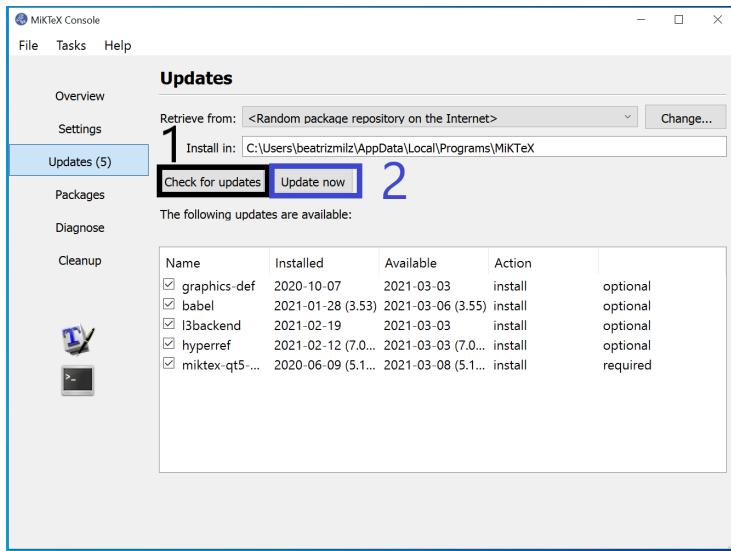
8. Na etapa “Update Check”, apresentará a opção de buscar por atualizações. Mantenha o botão “**Check for updates now**” selecionado, e clique no botão **Start**.



9. A próxima janela mostrará uma mensagem dizendo que a instalação foi completa. Clique no botão **Close** para fechar essa janela.



10. Abra o MiKTeX console através do menu Iniciar do Windows. No menu lateral, escolha a opção “Updates”. Clique em “**Check for updates**”, e o MiKTeX buscará informações sobre as possíveis atualizações necessárias. Então clique em “**Update now**”, para que as atualizações sejam baixadas e instaladas. Essa etapa também pode demorar um pouco!



11. Após instalar as atualizações, o MiKTeX avisará que ele precisa ser fechado. Abra-o novamente. Após este processo, o MiKtex estará pronto para uso.

1.3.1.2 No Linux

No linux, a forma mais direta de garantir que temos o LaTeX na máquina com todas as dependências é instalando o `texlive-full`. No Ubuntu, basta rodar

```
sudo apt install texlive-full
```

Importante: o `texlive-full` é uma “bruta” de instalar tudo do LaTeX, e pode ocupar mais do que 5GB do seu disco! Então tome cuidado e procure alternativas, como as descritas neste site.

1.3.1.3 No MacOS

Uma distribuição de LaTeX que apresenta versão para MacOS é o MikTex. Para instalá-lo, primeiramente faça o download do arquivo `.dmg` através deste

link. Este tutorial apresenta todos os passos de instalação no MacOS, porém abaixo também vamos descrever os passos:

1. Faça o download do arquivo `.dmg` através deste link, escolhendo a versão de instalação para MacOS.
2. Abra o arquivo baixado. Uma janela irá abrir, e você deve arrastar o ícone “MiKTeX Console” para a pasta “Applications” (como mostrado na animação abaixo, sendo necessário arrastar apenas uma vez).
3. No Launchpad (menu de arquivos instalados), procure o ícone do “MiKTeX Console” e abra-o. Você também pode usar a pesquisa do MacOS (utilizando o atalho Command + Barra de espaço), pesquisando por MiKTeX para abrir o software.
4. Ao abrir pela primeira vez, o MiKTeX solicitará que você selecione uma opção para que a instalação seja completada. Selecione “**Finish private setup**”, e isso fará com que o MiKTeX esteja disponível apenas no seu usuário do computador. Após selecionar essa opção, ele terminará a configuração, e aparecerá uma mensagem dizendo que o MiKTeX será reiniciado. Espere até que ele reinicie. Ao final, aparecerá uma mensagem dizendo que é necessário fazer atualizações no MiKTeX.
5. No menu lateral, escolha a opção “Updates”. Clique em “**Check for updates**”, e o MiKTeX buscará informações sobre as possíveis atualizações necessárias. Então clique em “**Update now**”, para que as atualizações sejam baixadas e instaladas.
6. Após instalar e atualizar, o MiKTeX está pronto para uso!

1.3.2 Office

No R, os arquivos do pacote `office` também são gerados a partir do Pandoc. Por isso, não é necessário instalar nenhum software adicional para gerar os documentos.

No entanto, para abrir documentos do Office, como `.pptx` e `.docx`, é necessário ter o Office ou alguma alternativa aberta instalados.

Além disso, por conta das particularidades desses tipos de documentos, muitas vezes é necessário instalar pacotes adicionais, que não vêm com o `{rmarkdown}`. Abaixo, segue uma lista de pacotes usualmente utilizados nesse contexto:

- `{officedown}`.
- `{officer}`, que é uma dependência do `{officedown}`.
- `{flextable}`, para geração de tabelas.

- `{svglite}`, para renderizar gráficos em SVG.

Uma dica adicional: É possível gerar gráficos *editáveis* nos documentos Office! Dessa forma, você pode gerar o relatório bruto e depois editar manualmente. Para isso, siga o tutorial deste vídeo do YouTube. Referência: [link](#).

1.3.2.1 No Windows

O LibreOffice pode ser instalado através deste link, escolhendo o sistema operacional “Windows” e clicando em “Baixar” e executando o arquivo baixado.

1.3.2.2 No Linux

No linux, a forma mais fácil de abrir documentos provenientes do Office é utilizando o LibreOffice:

```
sudo apt install libreoffice
```

O LibreOffice possui algumas limitações quando comparado ao software da Microsoft. Por exemplo, as tabelas que saem bonitas no Office podem ficar desformatadas no LibreOffice.

o LibreOffice 7.1 não está ainda no aptitude. Se quiser instalar essa versão, é possível baixar o arquivo `.deb` no site da ferramenta.

1.3.2.3 No MacOS

O sistema operaciona MacOS disponibiliza o iWork, que é composto pelas ferramentas Numbers, Pages e Keynote. Esses programas possibilitam abrir arquivos provenientes do Office.

Outra opção é a utilização do LibreOffice.

1. Primeiro, faça o download do arquivo `.dmg` de instalação, através deste link, escolhendo o sistema operacional “MacOS” e clicando em “Baixar”.
2. Após baixar o arquivo, abra-o. Uma janela irá abrir (como na imagem a seguir). Você deve arrastar o ícone “LibreOffice” para a pasta “Applications” (como mostrado na animação abaixo, sendo necessário arrastar apenas uma vez).
3. O LibreOffice será instalado e você poderá utilizá-lo abrindo através do Launchpad (menu de arquivos instalados), ou também utilizando a pesquisa do Mac (utilizando o atalho Command + Barra de espaço), pesquisando por LibreOffice para abrir o software.

1.3.3 Pagedown e Chrome

O `{pagedown}` é uma alternativa recente para produzir arquivos em HTML utilizando o `paged.js`, uma biblioteca em JavaScript que torna um arquivo HTML muito parecido com um PDF. Com a ajuda do Chrome, é possível, inclusive, exportar o arquivo para PDF.

O `{pagedown}`, portanto, apresenta o melhor de dois mundos: a praticidade da extensão HTML e a portabilidade de arquivos PDF. Dessa forma, é possível gerar documentos PDF com formatos agradáveis sem depender do LaTeX.

No entanto, para gerar documentos PDF, é preciso utilizar um navegador. O navegador recomendado pelo `{pagedown}` é o Chrome, tanto que o pacote já vem com a função `pagedown::chrome_print()`, que pode ser utilizada para transformar um arquivo HTML ou RMarkdown em PDF.

Se você já tiver o Chrome instalado na sua máquina, provavelmente tudo funcionará sem problemas. Se não tiver e não quiser instalar, é possível instalar somente o headless Chrome.

No linux, uma alternativa é instalar o projeto Chromium, que é quase que totalmente open source.

1.3.4 Renderizando HTML widgets

Os HTML widgets são formas de ligar bibliotecas do JavaScript com pacotes do R. Isso permite gerar relatórios dinâmicos em HTML diretamente do R. O pacote `{htmlwidgets}` possui um conjunto de melhores práticas para criação dessas soluções.

No entanto, é comum utilizar HTML widgets em relatórios estáticos. Para isso, o R precisa de uma estratégia para tirar fotos estáticas dos widgets. Isso é realizado através do pacote `{webshot}`.

Você pode instalar o `{webshot}` executando o código abaixo:

```
install.packages("webshot")
```

Para que o `{webshot}` funcione, no entanto, também é necessário ter o PhantomJS instalado na máquina. O PhantomJS é basicamente um navegador que roda totalmente no plano de fundo, possibilitando a captura de tela.

Para instalar o PhantomJS, execute o código abaixo:

```
webshot::install_phantomjs()
```

1.3.5 Blogdown

Para criar blogs utilizando o pacote `{blogdown}`, também é necessário instalar o Hugo, programa que renderiza sites estáticos.

Felizmente, essa tarefa é bem tranquila pois o `{blogdown}` já vem com um helper para isso. Basta executar o código abaixo e reiniciar a sua sessão do R para que o Hugo seja instalado:

```
blogdown::install_hugo()
```

1.3.6 Ferramentas de desenvolvimento

Em algumas situações (como por exemplo, no desenvolvimento de pacotes), é solicitado a instalação de ferramentas de desenvolvimento. Essas ferramentas dependem do sistema operacional utilizado.

No caso do desenvolvimento de pacotes, você pode verificar se já possui as ferramentas de desenvolvimento instaladas usando a seguinte função: `devtools::has-devel()`. Caso a mensagem retornada seja “*Your system is ready to build packages!*”, significa que você já tem as ferramentas necessárias instaladas.

```
devtools::has-devel()  
#> Your system is ready to build packages!
```

1.3.6.1 No Windows - Rtools

O software Rtools é apenas necessário para computadores que utilizem o sistema operacional Windows. Para instalá-lo, clique neste link e faça o download do arquivo instalador referente à sua versão do Windows (32 ou 64 bits). Abra o arquivo baixado para realizar a instalação.

1.3.6.2 No Linux - r-base-dev

No Linux, caso seja necessário, você também pode instalar o pacote `r-base-dev`. Por exemplo, no Ubuntu você pode instalá-lo executando o seguinte código no terminal:

```
sudo apt install r-base-dev
```

1.3.6.3 No Mac OS - Xcode command line tools

O Xcode command line tools é uma ferramenta de desenvolvimento oficial da Apple. Para instalar essa ferramenta, primeiramente é preciso acessar o Terminal. Você pode encontrar o Terminal procurando na lista de programas instalados (*Launchpad*), ou então na opção de busca (usando o atalho: **command + barra de espaço**) e escrevendo Terminal.

No terminal, escreva o seguinte código e aperte Enter, o que iniciará a instalação:

```
xcode-select --install
```

Siga as instruções apresentadas para instalar.

Caso a seguinte mensagem apareça, significa que você já tem a ferramenta instalada: `error: command line tools are already installed, use "Software Update" to install updates.`

Caso queira confirmar que a instalação foi bem sucedida, execute no terminal o código a seguir. Caso a mensagem de retorno seja um caminho (por exemplo: `/Library/Developer/CommandLineTools`), significa que a instalação foi bem sucedida, e esse é o caminho onde a ferramenta foi instalada.

```
xcode-select -p
```

1.3.7 Git

O Git é um programa para linha de comando que possibilita realizar o controle de versões dos arquivos em um diretório (ou seja, em uma pasta). Para instalar esse sistema de controle de versões distribuído, acesse a página sobre Download do Git, escolha o seu sistema operacional, siga as instruções apresentadas para instalação.

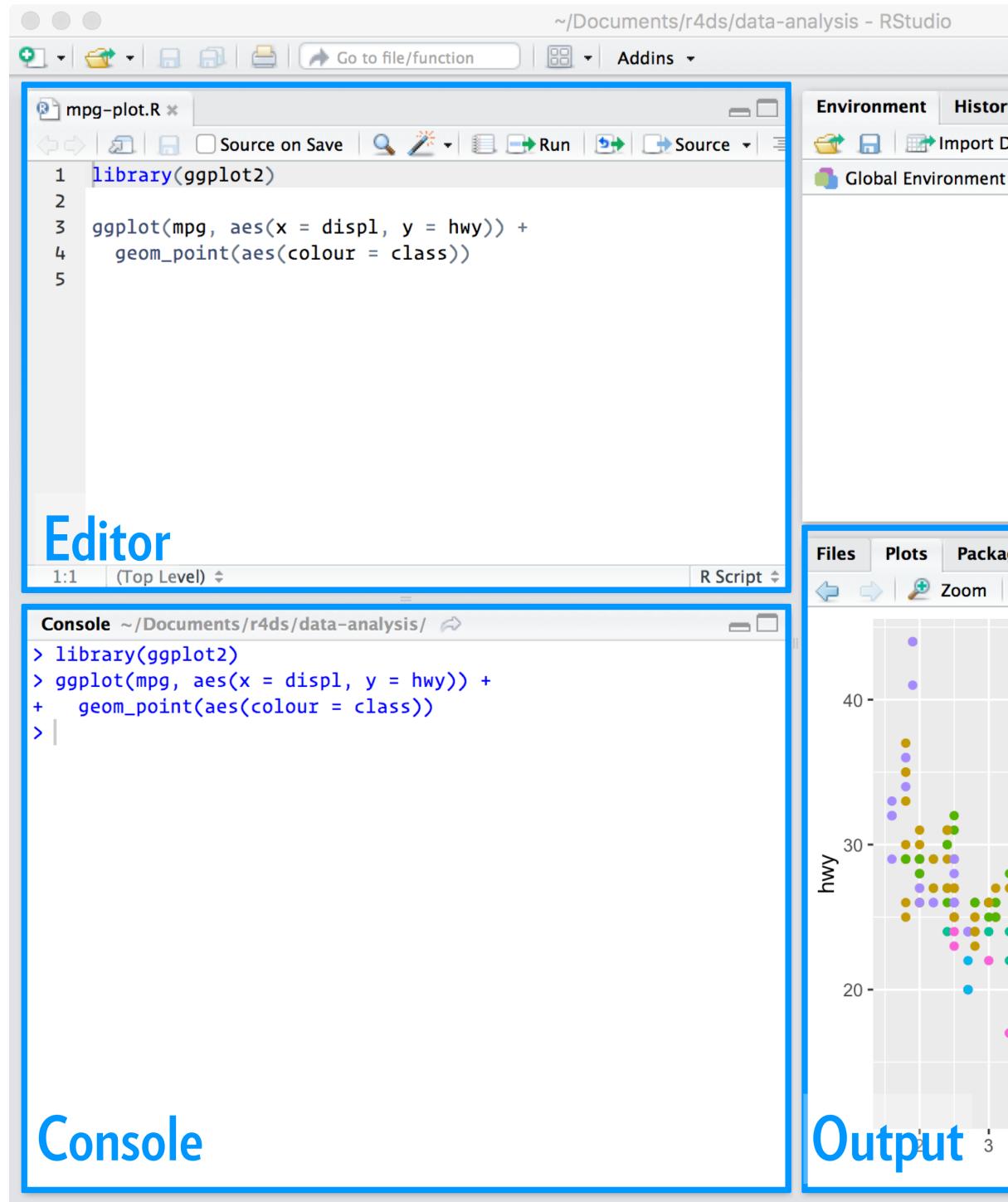
Chapter 2

RStudio

O RStudio é sem dúvida o mais completo ambiente de desenvolvimento para programação em R. Descubra aqui as funcionalidades do RStudio que nos permitem escrever códigos e analisar resultados de forma muito mais eficiente.

2.1 Telas

Ao abrir o RStudio, você verá 4 quadrantes. Observe a figura abaixo.



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, como olhar a documentação de funções, analisar os objetos criados em uma sessão do R, procurar e organizar os arquivos que compõem a nossa análise, armazenar e analisar os gráficos criados e muito mais.

- **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.

2.2 Atalhos

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

- **CTRL+ENTER:** avalia 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:** (`%>%`) operador *pipe*. Guarde esse atalho, você o usará bastante.
- **CTRL+1:** altera cursor para o script.
- **CTRL+2:** altera cursor para o console.
- **CTRL+ALT+I:** cria um chunk no R Markdown.
- **CTRL+SHIFT+K:** compila um arquivo no R Markdown.
- **ALT+SHIFT+K:** janela com todos os atalhos disponíveis.

No MacBook, os atalhos geralmente são os mesmos, substituindo o **CTRL** por **command** e o **ALT** por **option**.

2.3 Projetos

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

Um projeto nada mais é do que uma pasta no seu computador. Nessa pasta, estarão todos os arquivos que você usará 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 reproduutíveis e compartilhar o nosso trabalho.

Você que está começando agora no R, já se habitue a criar um novo 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**.

New Project

Create Project

**New Directory**
Start a project in a brand new working directory

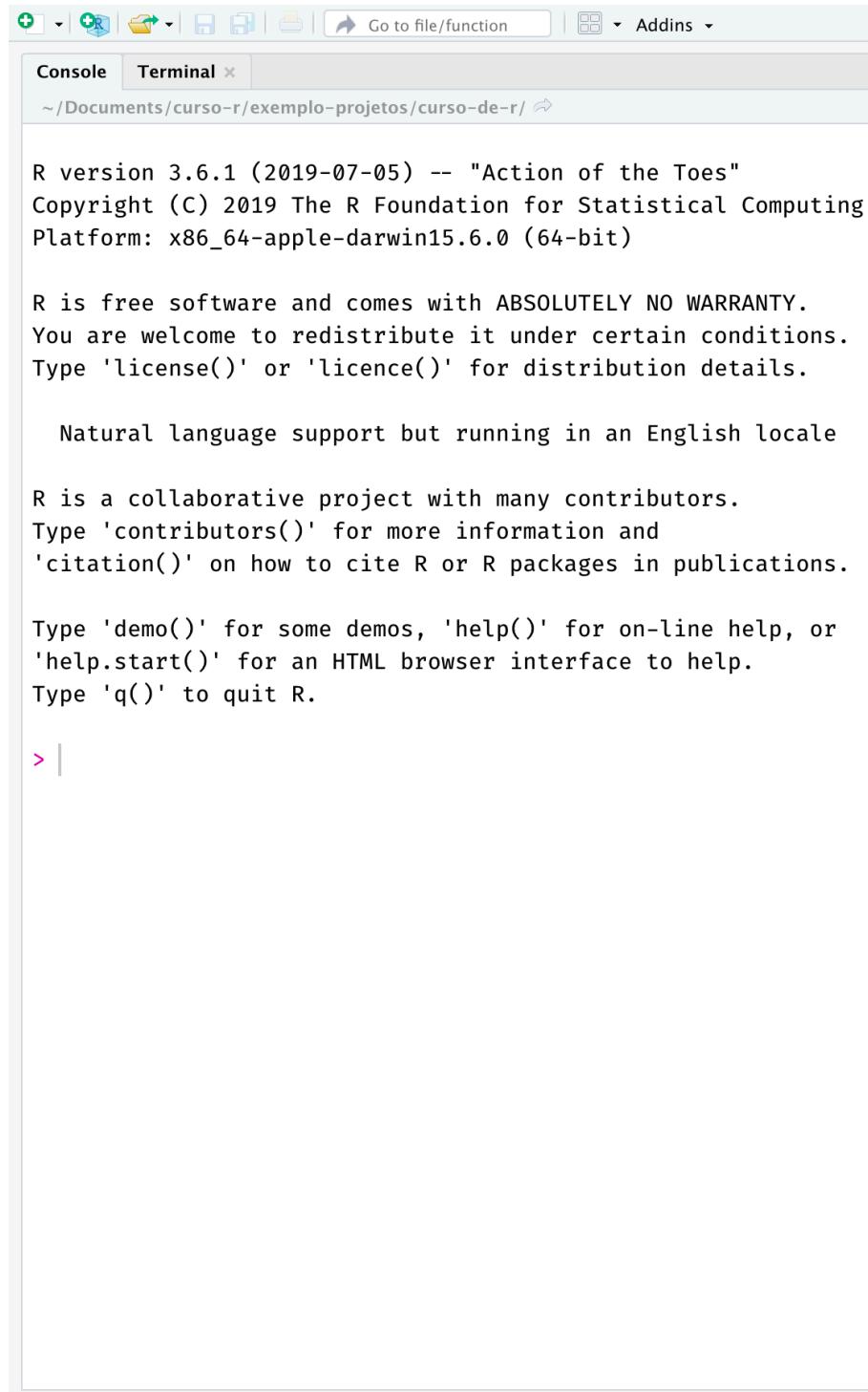
**Existing Directory**
Associate a project with an existing working directory

**Version Control**
Checkout a project from a version control repository

Cancel

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.

Quando um projeto estiver aberto no RStudio, o seu nome aparecerá no canto superior direito da tela. Na aba **Files**, aparecerão todos os arquivos contidos no projeto.



The screenshot shows the RStudio interface with the 'Console' tab selected. The console window displays the standard R startup message, including the version number (R version 3.6.1), copyright information, platform details, and licensing terms. It also includes a note about natural language support and collaborative development. At the bottom of the console window, there is a single character '>' followed by a vertical cursor line, indicating where the user can type commands.

```
R version 3.6.1 (2019-07-05) -- "Action of the Toes"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

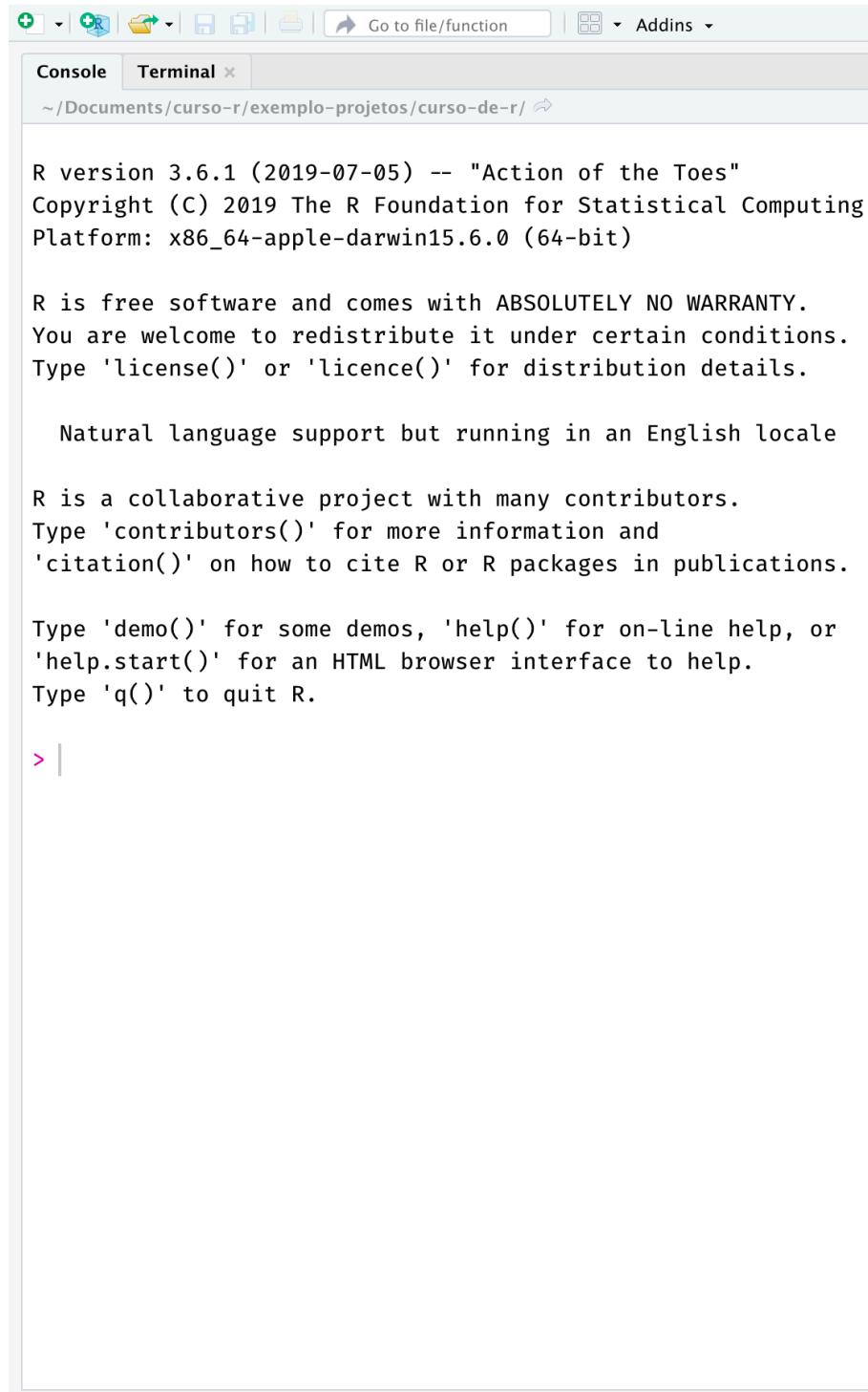
Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
```

Você pode criar livremente novas pastas dentro da pasta do projeto. Por padrão, o R sempre começará a procurar arquivos na pasta raiz do projeto (é a pasta que contém o `nome-do-projeto.Rproj`).

Uma maneira fácil de navegar entre projetos é utilizar o menu disponibilizado quando clicamos no nome do projeto. Veja a figura a seguir.



The screenshot shows the RStudio interface with the 'Console' tab selected. The console window displays the standard R startup message, which includes the version number (R version 3.6.1), copyright information, platform details, and various informational messages about the software's nature and usage. The message ends with a prompt '> |' indicating where the user can begin their own R session.

```
R version 3.6.1 (2019-07-05) -- "Action of the Toes"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

Nesse menu, além de podermos criar novos projetos ou abrir projetos já existentes, também temos um acesso rápido a projetos abertos recentemente.

Basta clicar em qualquer um deles para trocar de projeto, isto é, deixar de trabalhar em uma análise e começar a trabalhar em outra.

A seguir, apresentamos algumas estruturas de organização de projetos no RStudio.

Estrutura 1. Por extensão de arquivo.

```
nome_do_projeto/
- .Rprofile      # códigos para rodar assim que abrir o projeto
- R/              # Código R, organizado com a-carrega.R, b-prepara bd.R, c-vis.R, d-modela, ...
- RData/          # Dados em formato .RData
- csv/            # Dados em .csv
- png/            # gráficos em PNG
- nome_do_projeto.Rproj
```

Estrutura 2. Típico projeto de análise estatística.

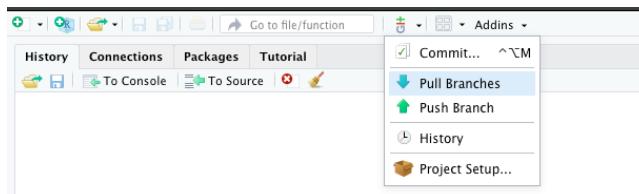
```
project/
- README.Rmd    # Descrição do pacote
- set-up.R       # Pacotes etc
- R/              # Código R, organizado com 0-load.R, 1-tidy.R, 2-vis.R, ...
- data/           # Dados (estruturados ou não)
- figures/        # gráficos (pode ficar dentro de output/)
- output/         # Relatórios em .Rmd, .tex etc
- project.Rproj
```

Estrutura 3. Pacote do R.

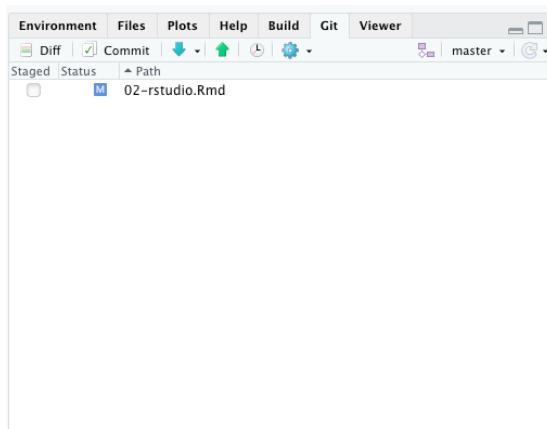
```
project/
- README.md      # Descrição do pacote
- DESCRIPTION     # Metadados estruturados do pacote e dependências
- NAMESPACE       # importações e exportações do pacote
- vignettes/      # Relatórios em .Rmd
- R/              # Funções do R
- data/            # Dados estruturados (tidy data)
- data-raw/        # Dados não estruturados e arqs 0-load.R, 1-tidy.R, 2-vis.R, ...
- project.Rproj
```

2.4 Git e versionamento

O RStudio possui funcionalidades para quem trabalha com o programa *Git* para versionar arquivos.



Além disso, quando você inicia o Git na pasta raiz de um projeto, o RStudio criará uma nova abinha chamada *Git*, onde você pode confirmar os arquivos modificados que estão à espera de um *commit*.



Para saber mais sobre Git e versionamento de arquivos, leia esse excelente capítulo do Zen do R.

2.5 Cheatsheets

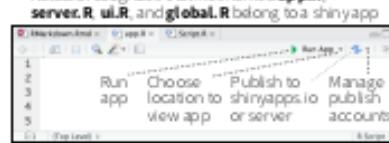
O RStudio tem à disposição algumas *folhas de cola*, as **cheatsheets**. Elas trazem um resumão de como utilizar diversos pacotes e até o próprio RStudio.

Para acessá-las, basta clicar no menu **Help** e então em **Cheatsheets**.

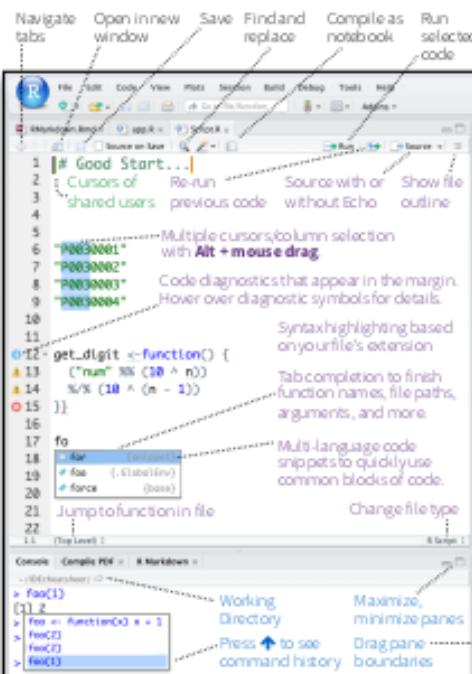
Outra forma de acessá-las é através do site da RStudio, onde existem muitas opções de **cheatsheets**, incluindo algumas traduzidas para o Português.

RStudio IDE :: CHEAT SHEET

Documents and Apps



Write Code



R Support



Debug Mode

Open with `debug()`, `browser()`, or a breakpoint. RStudio will open the debugger mode when it encounters a breakpoint while executing code.

Click next to line number to add/remove a breakpoint.

Highlighted line shows where execution has paused

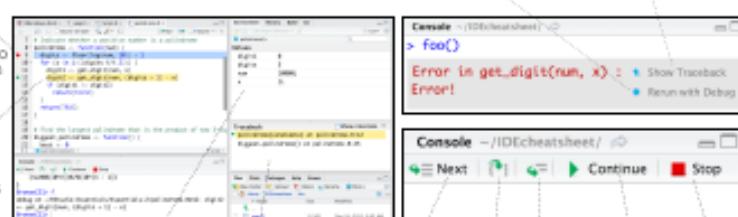
Run commands in environment where execution has paused

Examine variables in executing environment

Select function in traceback to debug

Launch debugger mode from origin of error

Open traceback to examine the functions that R called before the error occurred

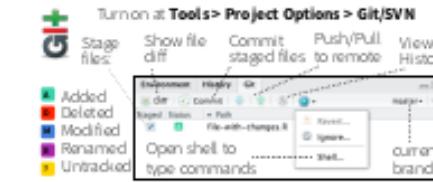


Step through code one line at a time

Step into and out of functions to run

Resume / Quit debug execution mode

Version Control

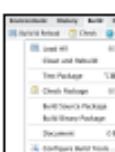


Package Writing

File > New Project > New Directory > R Package

Turn project into package, Enable roxygen documentation with Tools > Project Options > Build Tools.

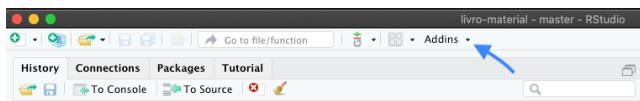
Roxygen guide at Help > Roxygen Quick Reference



R Studio

2.6 Addins

Addins são cápsulas de código R que podem ser executados interativamente a partir do menu **Addins**.

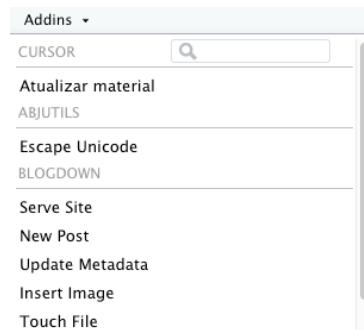


Eles servem para fazer diversos tipos de tarefas, como identar seu código, ajudar nas tarefas de copiar e colar, inserir códigos no seu script e muito mais.

Os addins são criados pelos pacotes que instalamos, então cada pacote vai nos disponibilizar addins que facilitam algum tipo de tarefa.

Veja aqui um exemplo de um addin que formata um código e aqui um addin que filtra uma base.

O pacote `{CursoR}` da Curso-R, por exemplo, tem um addin que auxilia quem faz nossos cursos a atualizar o material de cada aula.



2.7 Snippets

Snippets são atalhos que podemos criar para gerar pedaços rotineiros de código.

O RStudio já vem com vários snippets criados. Você pode visualizar os snippets existentes ou criar novos acessando **Tools > Global Options... > Code > Editting > Snippets > Edit snippets**.

No arquivo de texto aberto, basta seguir o padrão dos snippets já existentes para criar novos. Você também pode editar os snippets padrão do RStudio.

```

snippet ei
else if (${1:condition}) {
| ${0}
}

snippet fun
${1:name} ← function(${2:variables}) {
| ${0}
}

snippet for
for (${1:variable} in ${2:vector}) {
| ${0}
}

snippet while
while (${1:condition}) {
| ${0}
}

snippet switch
switch (${1:object},
| ${2:case} = ${3:action}
)

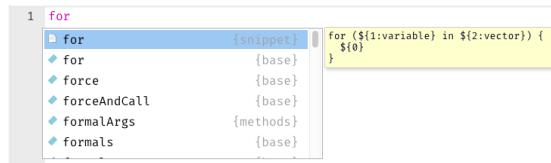
snippet apply
apply(${1:array}, ${2:margin}, ${3: ... })

snippet lapply
lapply(${1:list}, ${2:function})

snippet sapply
sapply(${1:list}, ${2:function})

```

Cada snippet terá um *nome*, que deverá ser utilizado no script para ativar o código guardado por trás.



Basta escrever o nome, apertar a teclar TAB e *voilà*.

```

1 for (variable| in vector) {
2
3 }

```


Chapter 3

R Básico

Introduziremos aqui os principais conceitos de programação em R. Indicamos a leitura deste capítulo a quem nunca teve contato com uma linguagem de programação ou a quem gostaria de entender um pouco melhor a estrutura de objetos, funções e classes do R.

Os tópicos discutidos aqui são especialmente importantes para entendermos o que é um *data frame*, a nossa base de dados dentro do R, e quais operações estão sendo realizadas por trás das cortinas quando estivermos filtrando suas linhas ou modificando suas colunas. Também são importantes para começarmos a criar as nossas próprias funções, o que deixa nossos códigos muito mais organizados, eficientes e *compartilháveis*.

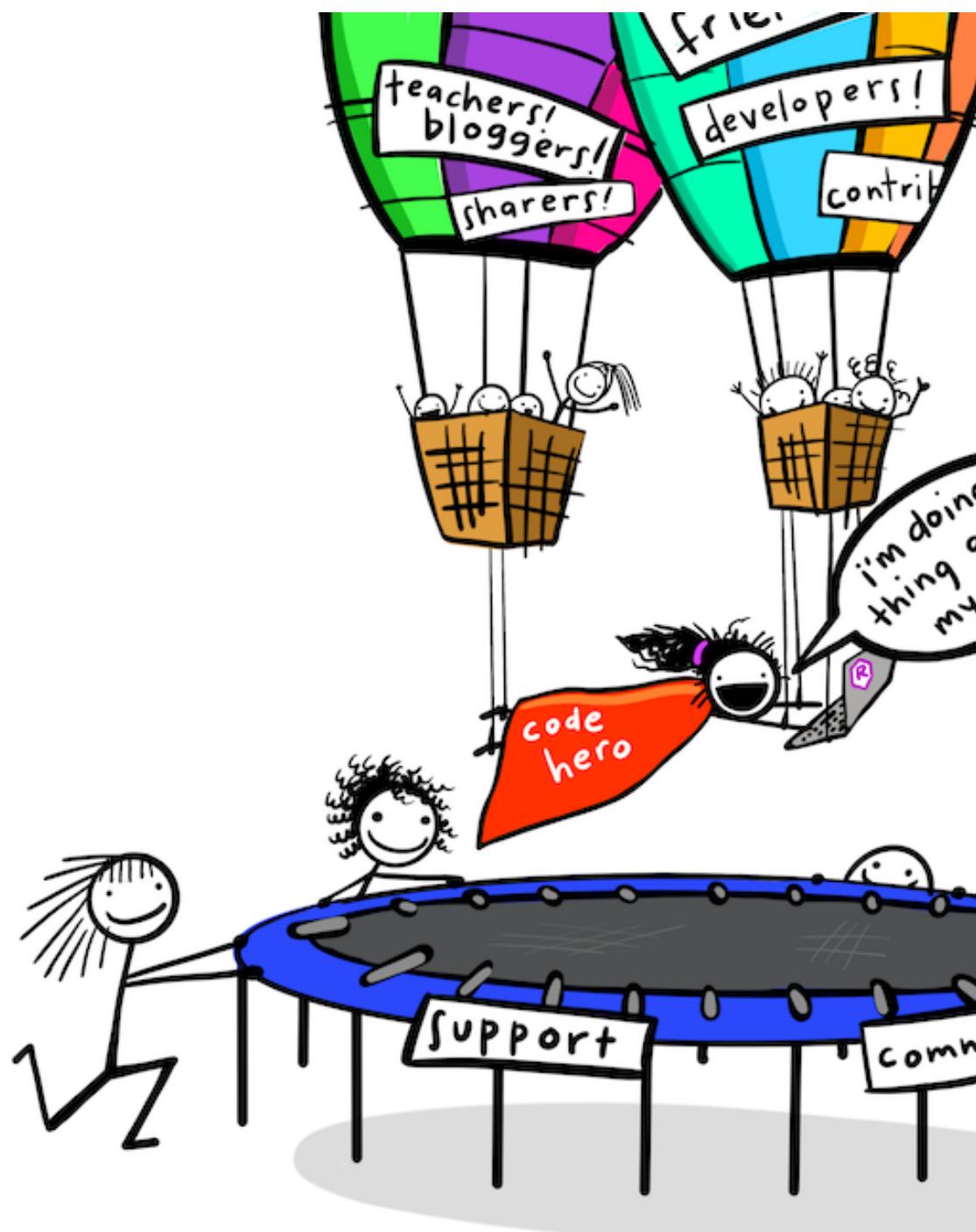
3.1 Pedindo Ajuda

A linguagem R é bem intuitiva. É possível fazer bastante coisa à base da tentativa e erro. Além disso, grande parte do conhecimento é escalável, isto é, aprender a utilizar uma função é meio caminho andado para aprender todas as outras funções que operam de forma semelhante¹.

No entanto, a intuição não infalível, e recorrentemente vamos precisar de ajuda para rodar alguma função ou descobrir como fazer alguma tarefa no R. Felizmente, a comunidade R é bem ativa e existem vários lugares para buscar respostas. Nesta seção, vamos apresentar as principais maneiras algumas dessas maneiras.

\begin{figure}

¹Essa ideia é um dos princípios por trás do **tidyverse**.



}

\caption{Arte por Allison Horst (?). Veja nas Referências onde encontrá-la.}
\end{figure}

No R, há quatro principais entidades para se pedir ajuda:

- Help/documentação do R
- Google
- Stack Overflow
- Coleguinha

A busca por ajuda é feita preferencialmente, mas não necessariamente, na ordem acima.

3.1.1 Documentação do R

A documentação do R serve para você aprender a usar uma determinada função. Se você não sabe o que é uma função, não se preocupe. Discutiremos esse tópico nas Seções 3.3 e 3.11.

Você pode acessar a documentação de uma função² das seguintes maneiras:

```
?mean  
help(mean)
```

Algumas dicas:

1. Leia a seção *Usage* para ter noção de como usar a função.
2. Os parâmetros da função estão descritos em *Arguments*.
3. Os exemplos no final são particularmente úteis.
4. Caso essa função não atenda às suas necessidades, a seção *See Also* sugere funções relacionadas.

Alguns pacotes possuem tutorias de uso mais completos. Esses textos são chamados de *vignettes* e podem ser acessados com a função `vignette(package = 'nomeDoPacote')`. Por exemplo, `vignette(package = 'dplyr')`.

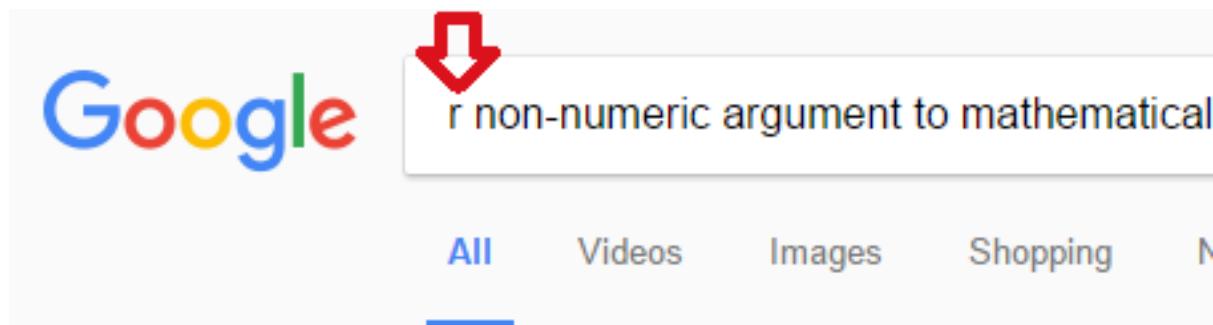
²Bases de dados presentes em pacotes também têm documentação, e geralmente é possível encontrar o significado de cada variável nela. Por exemplo, `help(mtcars)`.

3.1.2 Google

Há uma comunidade gigantesca de usuários de R gerando diariamente uma infinidade de conteúdos e discussões. Não raramente, você irá encontrar discussões sobre o seu problema simplesmente o descrevendo no Google. Pesquisas em inglês aumentam consideravelmente a chance de encontrar uma resposta.

Quando você recebe um erro na tentativa de rodar algum código no R e não sabe o que está errado, uma boa estratégia é pesquisar a mensagem de erro no Google. Essa deve ser sua primeira tentativa para resolver o problema. Repare na imagem abaixo o ‘r’ adicionado na busca. Isso ajuda bastante a encontrar uma solução.

```
log("5")
## Error in log("5"): non-numeric argument to mathematical function
```



3.1.3 Stack Overflow

O Stack Overflow e o Stack Overflow em Português são sites de Pergunta e Resposta amplamente utilizados por todas as linguagens de programação, e o

R é uma delas. Nos EUA, chegam até a usar a reputação dos usuários dentro da plataforma como diferencial no currículo!

Provavelmente o Google lhe indicará uma página deles quando você estiver procurando ajuda. E quando todas as fontes possíveis de ajuda falharem, o Stack Overflow lhe dará o espaço para **criar sua própria pergunta**.

Um ponto importante: como fazer uma *boa pergunta* no Stack Overflow?

No site, existe um tutorial com uma lista de boas práticas, que se encontra aqui. Resumindo, as principais dicas são

- ser conciso;
- ser específico;
- ter mente aberta; e
- ser gentil.

Porém, no caso do R, há outro requisito que vai aumentar muito sua chance de ter uma boa resposta: **exemplinho minimal e reproduzível**.

- Ser **minimal**: usar bancos de dados menores e utilizar pedaços de códigos apenas suficientes para apresentar o seu problema. Não precisa de banco de dados de um milhão de linhas e nem colocar o seu código inteiro para descrever a sua dúvida.
- Ser **reproduzível**: o seu código deve rodar fora da sua máquina. Se você não fornecer uma versão do seu problema que rode (ou que imite seu erro), as pessoas vão logo desistir de te ajudar. Por isso, nunca coloque bancos de dados que só você tem acesso. Use bancos de dados que já vem no R ou disponibilize um exemplo (possivelmente anonimizado) em .csv na web para baixar. E se precisar utilizar funções de algum pacote, especifique os pacotes que você usou.

3.2 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.

Vamos começar com o exemplo mais simples possível:

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

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

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.

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

Pronto! Você já é capaz de pedir ao R para fazer qualquer uma das quatro operações aritméticas básicas. A seguir, apresentamos uma lista resumindo como fazer as principais operações no R.

```
# adição
1 + 1
## [1] 2

# subtração
4 - 2
## [1] 2

# multiplicação
2 * 3
## [1] 6

# divisão
5 / 3
## [1] 1.666667

# potência
4 ^ 2
## [1] 16

# resto da divisão de 5 por 3
5 %% 3
## [1] 2

# parte inteira da divisão de 5 por 3
5 %/% 3
## [1] 1
```

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 os parênteses nunca são demais!

Uma outra forma de executar uma expressão é escrever o código em um **script**, deixar o cursor em cima da linha e usar o atalho **Ctrl + Enter**.

Assim, o comando é enviado para o **Console**, onde é diretamente executado.

Essa operação é chamada de **avaliar**, **executar** ou **rodar** o código.

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
```

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
```

Exercícios

1. Qual a diferença entre o R e o RStudio?
2. Podemos usar o RStudio sem o R? E o R sem o RStudio?
3. Precisamos compilar nossos códigos de R?
4. Calcule o número de ouro no R. Dica: o número de ouro é dado pela expressão $\frac{1+\sqrt{5}}{2}$.
5. Por que é preferível escrevermos sempre o nosso código no *script* e não no *Console*?

3.3 Objetos e funções

O R te permite salvar valores dentro de um **objeto**. Um objeto é simplesmente 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
```

Existem algumas regras para dar nomes aos objetos. A mais importante é: o nome deve começar com uma letra³. 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
```

Atenção!

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
```

³Ou com um ponto.

O objeto mais importante para o cientista de dados é, claro, a base de dados. No R, uma base de dados é representada por objetos chamados de *data frames*.

Na próxima seção, vamos entender o que são esses objetos.

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

A sintaxe para usar uma função é a seguinte:

```
nome_da_funcao(arg1, arg2, argn)
```

Entre parênteses, após o nome da função, temos o que chamamos de *argumentos*. Uma função pode ter qualquer número de argumentos e eles são sempre separados por vírgula.

Basicamente, uma função recebe seus argumentos, executa uma ação sobre ou a partir deles e devolve um resultado. Por exemplo

```
sum(1, 2)
## [1] 3
```

A função `sum()` recebeu os argumentos `1` e `2`, somou os dois valores e devolveu o resultado dessa operação: o valor `3`.

Falaremos mais sobre funções na Seção 3.11.

Exercícios

- Qual a diferença entre os códigos abaixo?

```
# Código 1
33 / 11

# Código 2
divisao <- 33 / 11
```

- Multiplique a sua idade por meses e salve o resultado em um objeto chamado `idade_em_meses`. Em seguida, multiplique esse objeto por 30 e salve o resultado em um objeto chamado `idade_em_dias`.
- Por que o nome `meu-objeto` não pode ser utilizado para criar um objeto?
O que significa a mensagem de erro resultante?

```
meu-objeto <- 1
## Error in meu - objeto <- 1: object 'meu' not found
```

3.4 Data frames

Os *data frames* são de extrema importância no R, pois são os objetos que guardam os nossos dados. Eles são equivalentes a uma tabela do SQL ou uma planilha do Excel.

A principal característica de um *data frame* é possuir linhas e colunas⁴. Veja o exemplo abaixo:

	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
## Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
## Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
## Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
## Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
## Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
## Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
## Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
## 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
## Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
## AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
## Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
## Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
## 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

⁴Você pode construir um *data frame* vazio, com 0 linha e 0 coluna. No entanto, a estrutura de linhas e colunas estará presente.

<code>## Ford Pantera L</code>	<code>15.8</code>	<code>8</code>	<code>351.0</code>	<code>264</code>	<code>4.22</code>	<code>3.170</code>	<code>14.50</code>	<code>0</code>	<code>1</code>	<code>5</code>	<code>4</code>
<code>## Ferrari Dino</code>	<code>19.7</code>	<code>6</code>	<code>145.0</code>	<code>175</code>	<code>3.62</code>	<code>2.770</code>	<code>15.50</code>	<code>0</code>	<code>1</code>	<code>5</code>	<code>6</code>
<code>## Maserati Bora</code>	<code>15.0</code>	<code>8</code>	<code>301.0</code>	<code>335</code>	<code>3.54</code>	<code>3.570</code>	<code>14.60</code>	<code>0</code>	<code>1</code>	<code>5</code>	<code>8</code>
<code>## Volvo 142E</code>	<code>21.4</code>	<code>4</code>	<code>121.0</code>	<code>109</code>	<code>4.11</code>	<code>2.780</code>	<code>18.60</code>	<code>1</code>	<code>1</code>	<code>4</code>	<code>2</code>

O `mtcars` é um *data frame* nativo do R que contém informações sobre diversos modelos de carros. Ele possui 32 linhas e 11 colunas⁵. Se você quiser saber mais sobre o `mtcars`, veja a documentação dele rodando `?mtcars` no **Console**.

Nos próximos capítulos, os *data frames* serão o nosso principal objeto de estudo. Aprenderemos a selecionar, criar e modificar colunas, filtrar e ordenar linhas, juntar dois *data frames* e, a partir deles, construiremos gráficos e ajustaremos modelos.

Mas, da mesma forma que é muito mais fácil aprendermos a fritar um ovo após entendermos o que é o fogo, uma frigideira e um ovo, vamos estudar nas próximas seções quais são as estruturas que formam os *data frames* e como manipulá-las.

3.5 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.

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 e são popularmente chamados de *strings* no contexto de programação.

⁵A 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.

```
a <- 10

# O objeto `a`, sem aspas
a
## [1] 10

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

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"
```

As classes mais básicas dentro do R são:

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

Geralmente serão utilizados como sinônimos:

- texto, string, *character*, caracteres
- número, valor real, *numeric*, *double*
- lógico, *logical*, booleano, valor TRUE/FALSE

Veja alguns exemplos:

```
# numeric
1
0.10
0.95
pi

# characters
```

```
"a"  
"1"  
"positivo"  
  
# logical  
TRUE  
FALSE
```

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**.

3.5.1 Exercícios

1. Guarde em um objeto chamado **nome** uma *string* contendo o seu nome completo.
2. Guarde em um objeto chamado **cidade** o nome da cidade onde você mora. Em seguida, guarde em um objeto chamado **estado** o nome do estado onde você mora. Usando esses objetos, resolva os itens abaixo:

- a. Utilize a função **nchar()** para contar o número de caracteres em cada string.
- b. Interprete o resultado do seguinte código:

```
paste(cidade, estado)
```

- c. Interprete o resultado do seguinte código:

```
paste(cidade, estado, sep = " - ")
```

- d. Desafio. Como você reproduziria o mesmo resultado do item (b) sem utilizar o argumento **sep**?
- e. Qual a diferença entre as funções **paste()** e **paste0()**?

3.6 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 no R 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"
```

Os objetos `vetor1` e `vetor2` são vetores.

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
```

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[2]
## [1] "b"
vetor[3]
## [1] "c"
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"
```

Essa operação é conhecida como *subsetting*, pois estamos pegando subconjuntos de valores de um vetor.

Se você tentar acessar uma posição do vetor que não existe, ele retornará `NA`, indicando que esse valor não existe. Discutiremos o que são `NA`'s na Seção 3.8.

```
vetor[5]
## [1] NA
```

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"
```

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. Agora temos um vetor com o texto "1", o texto "2" e o texto "a". 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.

Falaremos bastante de coerção nas próximas seções e capítulos, trazendo exemplos de quando ela ajuda e de quando ela atrapalha.

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.

```
vetor - 1
vetor / 2
vetor * 10
```

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**. Isso pode ficar um pouco confuso quando os dois vetores não possuem o mesmo tamanho. Tente adivinhar qual será a saída do código a seguir:

```
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 nada mais é do que um vetor de tamanho 1, isto é, 1 é igual a `c(1)`.

Usaremos esse comportamento no R o tempo todo e 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:

- 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)`.
- 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.

Exercícios

- Guarde em um objeto a sequência de números de 0 a 5 e resolva os itens abaixo.
 - a.** Use subsetting para fazer o R devolver o primeiro número dessa sequência. Em seguida, faça o R devolver o último número da sequência.
 - b.** Multiplique todos os valores do vetor por -1. Guarde o resultado em um novo objeto chamado `vetor_negativo`.

2. Crie um vetor com o nome de tres **frutas**, guarde em um objeto chamado **frutas** e resolva os itens abaixo.
 - a. Utilize a função **length()** para verificar o tamanho do vetor.
 - b. Inspecione a saída de **paste("eu gosto de", frutas)** e responda se o tamanho do vetor mudou.
3. O que é reciclagem? Escreva um código em R que exemplifique esse comportamento.
4. O que é coerção? Escreva um código em R que exemplifique esse comportamento.
5. Por que a coerção pode ser um problema na hora de importarmos bases de dados para o R?
6. Use a função **sum()** para somar os valores de 1 a 100.
7. Considere o vetor booleano a seguir:

```
dolar_subiu <- c(TRUE, TRUE, FALSE, FALSE, TRUE, FALSE, TRUE)
```

Este vetor tem informação de uma semana (7 dias, começando no domingo) indicando se o dólar subiu (TRUE) ou não subiu (FALSE) no respectivo dia.

Interprete o resultado dos códigos abaixo:

- a. **length(dolar_subiu)**
- b. **dolar_subiu[2]**
- c. **sum(dolar_subiu)**
- d. **mean(dolar_subiu)**

3.7 Testes lógicos

Poder fazer qualquer tipo de operação lógica é um dos motivos pelos quais programar nos deixar mais eficientes. Dê bastante atenção a elas, pois usaremos comparações lógicas o tempo todo!

Uma operação lógica nada mais é do que 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
```

Checando a classe desses valores, vemos que são lógicos⁶. 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. Começando pelo mais simples: 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
```

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
```

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

⁶Também conhecidos como valores binários ou booleanos

```

# Maior
3 > 3
## [1] FALSE
3 > 2
## [1] TRUE

# Maior ou igual
3 > 4
## [1] FALSE
3 >= 3
## [1] TRUE

# Menor
3 < 3
## [1] FALSE
3 < 4
## [1] TRUE

# Menor ou igual
3 < 2
## [1] FALSE
3 <= 3
## [1] TRUE

```

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

```

Nós começamos essa seção dizendo que usaremos testes lógicos o tempo todo. O motivo para isso é que eles 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 aqui, 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 muito simples: **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.

A seguir, apresentamos uma tabela com os principais operadores lógicos:

Para finalizar, listamos na tabela abaixo os principais operadores lógicos.

Operador	Descrição
$x < y$	x menor que y ?
$x \leq y$	x menor ou igual a y ?
$x > y$	x maior que y ?
$x \geq y$	x maior ou igual a y ?
$x == y$	x igual a y ?
$x != y$	x diferente de y ?
$!x$	Negativa de x
$x y$	x ou y são verdadeiros?
$x & y$	x e y são verdadeiros?
$x \%in% y$	x percente a y ?
$xor(x, y)$	x ou y são verdadeiros (apenas um deles)?

Por fim, veja algumas diferenças entre comparações lógicas no SQL e no R:

- **Igualdade:** no SQL é só um sinal de igual: $2 = 1$. No R são dois: $2 == 1$.
- **Diferença:** no SQL, usamos $<>$. No R usamos $!=$.
- **Negação:** em vez de usar a palavra NOT igual ao SQL, usamos $!$ no R. Por exemplo, `id not in ('1', '2', '3')` fica `!(id %in% c(1, 2, 3))`.

Exercícios

1 O código abaixo vai guardar no objeto `segredo` um número inteiro entre 0 e 10. Sem olhar qual número foi guardado no objeto, resolva os itens a seguir:

```
segredo <- round(runif(1, min = 0, max = 10))
```

- a. Teste se o segredo é maior ou igual a 0.
- b. Teste se o segredo é menor ou igual a 10.
- c. Teste se o segredo é maior que 5.
- d. Teste se o segredo é par.
- e. Teste se `segredo * 5` é maior que a sua idade.
- f. Desafio. Escreva um teste para descobrir o valor do segredo.

2. Escreva um código em R que devolva apenas os valores maiores ou iguais a 10 do vetor abaixo:

```
vetor <- c(4, 8, 15, 16, 23, 42)
```

3. Use o vetor numeros abaixo para responder as questões seguintes.

```
numeros <- -4:2
```

- a. Escreva um código que devolva apenas valores positivos do vetor `numeros`.
- b. Escreva um código que devolva apenas os valores pares do vetor `numeros`.
- c. Filtre o vetor para que retorne apenas aqueles valores que, quando elevados a 2, são menores do que 4.

3.8 Valores especiais

Vimos anteriormente que se você tentar acessar uma posição que não existe dentro de um vetor, ele retorna um valor estranho.

```
vetor <- c(1, 2, 3)
vetor[4]
## [1] NA
```

Esse valor, o `NA`, é tratado de forma especial no R. Ele representa a *ausência de informação*, isto é, a informação existe, mas nós (e o R) não sabemos qual é.

O `NA` para o R nada mais é do que o valor faltante ou omissão da Estatística. O famoso *missing*. Geralmente, quando temos uma base com valores faltando, como a idade para alguns indivíduos da nossa amostra, não significa que a idade deles não existe. Significa apenas que não temos essa informação.

Esse conceito é muito importante para entender o resultado da expressão abaixo.

```
5 == NA
## [1] NA
```

Em um primeiro momento, poderíamos esperar que o resultado fosse `TRUE`. Mas, sabendo o significado por trás do `NA` — um valor desconhecido —, a verdadeira pergunta que estamos fazendo é: 5 é igual a um valor que existe, mas que não sei qual é? É como se eu perguntasse se eu tenho 5 moedas na mão, mas lhe mostrasse a mão fechada. A resposta para isso é *não sei* ou, dentro do R, `NA`.

Um outro exemplo:

```

idade_ana <- 30
idade_beto <- NA
idade_carla <- NA

idade_ana == idade_beto
## [1] NA

idade_beto == idade_carla
## [1] NA

```

Eu posso saber a idade da Ana, mas se eu não souber a idade do Beto, não sei se os dois tem a mesma idade. Por isso, `NA`. Da mesma forma, se não sei nem a idade do Beto nem da Carla, também não tenho como saber se os dois têm a mesma idade. Outra vez `NA`.

Mas e quando queremos saber se um valor é `NA` ou não? Para fazer esse teste, temos que rodar `is.na(valor-ou-objeto)`.

```

is.na(NA)
## [1] TRUE

is.na(idade_ana)
## [1] FALSE

is.na(idade_beto)
## [1] TRUE

```

Repare que essa função também é vetorizada.

```

is.na(c(idade_ana, idade_beto, idade_carla))
## [1] FALSE  TRUE  TRUE

```

Assim como o `NA`, existem outros valores especiais muito comuns no R.

O `NaN` (*not a number*) representa indefinições matemáticas.

```

0/0
## [1] NaN

log(-1)
## Warning in log(-1): NaNs produced
## [1] NaN

```

O `Inf` (infinito) representa um número muito grande (que o computador não consegue representar) ou um limite matemático.

```
# O computador não consegue representar um número tão grande.
# O número é então """arredondado"" para infinito.
10^310
## [1] Inf

# Limite matemático.
1 / 0
## [1] Inf

# O "menos infinito" também existe.
-1 / 0
## [1] -Inf
```

O `NULL` (nulo) representa a ausência de um objeto. Ele não tem significado prático para a análise dados. Está mais em sintonia com comportamentos de lógica de programação. Muitas vezes vamos definir um objeto como nulo para dizer ao R que não queremos dar um valor para ele. Muito utilizado em funções.

Da mesma forma que utilizados `is.na()` para testar se um objeto é `NA`, utilizamos `is.nan()`, `is.infinite()` ou `is.null()` para testar se um objeto é `NaN`, infinito ou nulo.

```
nao_sou_um_numero <- NaN
objeto_infinito <- Inf
objeto_nulo <- NULL

is.nan(nao_sou_um_numero)
## [1] TRUE

is.infinite(objeto_infinito)
## [1] TRUE

is.null(objeto_nulo)
## [1] TRUE
```

Exercícios

1. Quais as diferenças entre `NaN`, `NULL`, `NA` e `Inf`? Digite expressões que retornem cada um desses valores.
2. Escreva um código que conte o número de `NAs` do vetor `b`.

```
b <- c(1, 0, NA, NA, NA, NA, 7, NA, NA, NA, NA, NA, 2, NA, NA, 10, 1, 1, NA)
```

3.9 Listas

Chegamos ao último tópico antes de voltarmos aos data frames: as listas.

Listas são objetos muito importantes dentro do R. Primeiro porque **todo data frame é uma lista**. Segundo porque elas são bem parecidas com vetores, mas com uma diferença essencial: você pode misturar diferentes classes de objetos dentro dela.

Para criar uma lista, rodamos `list(valor1, valor2, valor3)`.

```
list(1, "a", TRUE)
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
```

Veja que não houve coerção. Ainda temos um valor número, um texto e um valor lógico dentro da lista.

O *subsetting* de listas é um pouco diferente do que o de vetores. Isso porque **cada elemento de uma lista também é uma lista**. Veja o que acontece se tentarmos usar `[]` para pegar um elemento de uma lista.

```
lista <- list(1, "a", TRUE)

lista[1]
## [[1]]
## [1] 1

class(lista[1])
## [1] "list"
```

O R nos retorna uma lista com apenas aquele elemento. Se quisermos o elemento de fato dentro de cada posição, precisamos usar dois colchetes:

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

class(lista[[1]])
## [1] "numeric"
```

Cada elemento de uma lista ser uma lista é importante pois isso nos permite colocar vetores de tamanhos diferentes em cada posição. Isso faz das listas uma estrutura bem flexível para guardar dados.

```
lista <- list(1:3, "a", c(TRUE, TRUE, FALSE, FALSE))

lista
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE TRUE FALSE FALSE

lista[1]
## [[1]]
## [1] 1 2 3

lista[2]
## [[1]]
## [1] "a"

lista[3]
## [[1]]
## [1] TRUE TRUE FALSE FALSE
```

É muito comum darmos nomes para cada posição de uma lista.

```
dados_cliente <- list(cliente = "Ana Silva", idade = 25, estado_civil = NA)

dados_cliente
## $cliente
## [1] "Ana Silva"
##
## $idade
## [1] 25
##
```

```
## $estado_civil
## [1] NA
```

Agora, dentro da lista, o valor `Ana Silva`, por exemplo, está sendo atribuído ao nome `cliente`. Esse nome só existirá dentro da lista.

Um detalhe importante: os iguais utilizados nas atribuições dos nomes dentro da lista **não podem** ser substituídos por `<-`.

Quando as posições de uma lista tem nome, podemos acessar seus valores diretamente utilizando o operador `$`.

```
dados_cliente$cliente
## [1] "Ana Silva"

dados_cliente$idade
## [1] 25

dados_cliente$estado_civil
## [1] NA
```

Repare que o R devolve o valor dentro de cada posição, e não uma lista.

```
dados_cliente[1]
## $cliente
## [1] "Ana Silva"

dados_cliente$cliente
## [1] "Ana Silva"

class(dados_cliente[1])
## [1] "list"

class(dados_cliente$cliente)
## [1] "character"
```

Isto implica que, nesse exemplo, `dados_cliente$cliente` é equivalente a `dados_cliente[[1]]`.

Conforme ficamos mais e mais proficientes na linguagem R, as listas passam a ficar cada vez mais frequentes. Voltaremos a falar delas diversas vezes nos próximos capítulos, em especial no Capítulo 10.

3.10 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*.

Na seção anterior, nós dissemos que *data frames* são listas. Isso é importante pois todas as propriedades de uma lista valem para um *data frame*.

A melhor forma de entender essa equivalência é ver um *data frame* representado como uma lista.

```
as.list(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 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
##
## $cyl
## [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 4 8 6 8 4
##
## $disp
## [1] 160.0 160.0 108.0 258.0 360.0 225.0 360.0 146.7 140.8 167.6 167.6 275.8
## [13] 275.8 275.8 472.0 460.0 440.0 78.7 75.7 71.1 120.1 318.0 304.0 350.0
## [25] 400.0 79.0 120.3 95.1 351.0 145.0 301.0 121.0
##
## $hp
## [1] 110 110 93 110 175 105 245 62 95 123 123 180 180 180 205 215 230 66 52
## [20] 65 97 150 150 245 175 66 91 113 264 175 335 109
##
## $drat
## [1] 3.90 3.90 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 3.92 3.07 3.07 3.07 2.93
## [16] 3.00 3.23 4.08 4.93 4.22 3.70 2.76 3.15 3.73 3.08 4.08 4.43 3.77 4.22 3.62
## [31] 3.54 4.11
##
## $wt
## [1] 2.620 2.875 2.320 3.215 3.440 3.460 3.570 3.190 3.150 3.440 3.440 4.070
## [13] 3.730 3.780 5.250 5.424 5.345 2.200 1.615 1.835 2.465 3.520 3.435 3.840
## [25] 3.845 1.935 2.140 1.513 3.170 2.770 3.570 2.780
##
## $qsec
## [1] 16.46 17.02 18.61 19.44 17.02 20.22 15.84 20.00 22.90 18.30 18.90 17.40
## [13] 17.60 18.00 17.98 17.82 17.42 19.47 18.52 19.90 20.01 16.87 17.30 15.41
## [25] 17.05 18.90 16.70 16.90 14.50 15.50 14.60 18.60
##
## $vs
## [1] 0 0 1 1 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 0 1 0 0 0 1
```

```
##  
## $am  
## [1] 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1  
##  
## $gear  
## [1] 4 4 4 3 3 3 3 4 4 4 4 3 3 3 3 3 4 4 4 3 3 3 3 3 4 5 5 5 5 5 4  
##  
## $carb  
## [1] 4 4 1 1 2 1 4 2 2 4 4 3 3 3 4 4 4 1 2 1 1 2 2 4 2 1 2 2 4 6 8 2
```

O código acima nos permite ver o *data frame* `mtcars` representado como uma lista. Veja que cada coluna da base se transforma em um elemento da lista. E o nome de cada coluna vira o nome de cada posição. Isso é interessante, pois podemos usar nos *data frames* as mesmas operações que aprendemos para listas.

Por exemplo, podemos usar o operador `$` para acessar cada elemento da lista, isto é, cada coluna do *data frame*.

```
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 10.4  
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7  
## [31] 15.0 21.4
```

E assim como cada elemento de uma lista também é uma lista, cada elemento (coluna) de um *data frame* também é um *data frame.

```
mtcars[1]  
##          mpg  
## Mazda RX4     21.0  
## Mazda RX4 Wag 21.0  
## Datsun 710    22.8  
## Hornet 4 Drive 21.4  
## Hornet Sportabout 18.7  
## Valiant      18.1  
## Duster 360    14.3  
## Merc 240D     24.4  
## Merc 230      22.8  
## Merc 280      19.2  
## Merc 280C     17.8  
## Merc 450SE    16.4  
## Merc 450SL    17.3  
## Merc 450SLC   15.2  
## Cadillac Fleetwood 10.4  
## Lincoln Continental 10.4
```

```

## Chrysler Imperial    14.7
## Fiat 128            32.4
## Honda Civic          30.4
## Toyota Corolla       33.9
## Toyota Corona         21.5
## Dodge Challenger     15.5
## AMC Javelin          15.2
## Camaro Z28           13.3
## Pontiac Firebird      19.2
## Fiat X1-9            27.3
## Porsche 914-2         26.0
## Lotus Europa          30.4
## Ford Pantera L        15.8
## Ferrari Dino          19.7
## Maserati Bora          15.0
## Volvo 142E             21.4

class(mtcars[1])
## [1] "data.frame"

```

Mas se *data frames* são listas, por que existe a classe *data frame*? Na verdade, *data frames* são um tipo especial de listas, que têm as seguintes propriedades:

1. Todos os seus elementos (colunas) precisam ter o mesmo comprimento (número de linhas).
2. Todos os seus elementos (colunas) precisam ser nomeados.
3. *Data frames* têm 2 dimensões.

As propriedades (1) e (2) se devem ao formato das bases de dados. Elas são retangulares⁷ — observamos as mesmas variáveis (colunas) para todas as unidades amostrais (linhas)⁸ —, e precisam ter algum nome especificando as colunas.

Da mesma forma que podemos ver um *data frame* como uma lista, também podemos fazer o inverso.

```

dados_cliente <- list(
  cliente = c("Ana Silva", "Beto Pereira", "Carla Souza"),
  idade = c(25, 30, 23),
  estado_civil = c(NA, "Solteiro", "Casada")

```

⁷Também existem bases não retangulares, como dados de imagens por exemplo, mas não trataremos dessas estruturas neste livro.

⁸Mesmo quando uma variável não existe para uma unidade amostral, representamos esse valor como um *missing*


```

## 2                               Beto Pereira      30
## 3                               Carla Souza      23
##   c.NA...Solteiro....Casada...
## 1                               <NA>
## 2                               Solteiro
## 3                               Casada

```

A propriedade (3) é atribuída aos *data frames* para que possamos aproveitar melhor dessa estrutura retangular dentro do R. Na prática, essas duas dimensões representam nada mais que as linhas e as colunas da base. Essa é a maior diferença entre uma lista e um *data frame*.

```

class(mtcars)
## [1] "data.frame"

dim(mtcars)
## [1] 32 11

```

O resultado do código `dim(mtcars)` nos dá as seguintes informações:

- O *data frame* `mtcars` tem duas dimensões (como todo *data frame*).
- 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.

Veja a seguir que listas não têm dimensão.

```

mtcars_como_lista <- as.list(mtcars)

class(mtcars_como_lista)
## [1] "list"

dim(mtcars_como_lista)
## NULL

```

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:

```
# Toda 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 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4

# 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
```

Por fim, lembrando que dentro de cada coluna temos um vetor, podemos usar os testes lógicos para filtrar as linhas do nosso *data frame* conforme alguma regra.

```
mtcars$cyl == 4
## [1] FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE FALSE FALSE FALSE
## [25] 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 simples: linha com `TRUE` é retornada, linha com `FALSE` não.

Encerramos aqui a nossa primeira conversa sobre *data frames* para falar de outra estrutura muito importante dentro do R: as funções.

Exercícios

1. Quais códigos abaixo retornam **um vetor** com a coluna `mpg` do data frame `mtcars`?

- a. `mtcars$mpg`
- b. `mtcars[, 3]`
- c. `mtcars("mpg")`
- d. `mtcars[, "mpg"]`
- e. `mtcars.mpg`
- f. `mtcars[, 1]`
- g. `mtcars[1, 1]`
- h. `mpg$mtcars`

2. Para que serve a função `str()`. Dê um exemplo do seu uso.

3. Para que serve a função `names()`. Dê um exemplo do seu uso.

4. Use o data frame `airquality` para responder às questões abaixo:

- a. Quantas colunas `airquality` tem?
 - b. Quantas linhas `airquality` tem?
 - c. O que a função `head()` retorna?
 - d. Quais são os nomes das colunas?
 - e. Qual é a classe da coluna `Ozone`?
5. Desafio. Calculando desvio-padrão no R. Use o data frame `airquality` para responder às questões abaixo:
- a. Tire a média da coluna `Ozone` e guarde em um objeto.
 - b. Guarde em um objeto o vetor correspondente à coluna `Ozone` subtraída da sua própria média (calculada em no item a).
 - c. Eleve o vetor calculado em (b) ao quadrado. Salve o resultado em um novo objeto.
 - d. Tire a média do vetor calculado em (c) e salve o resultado em um objeto chamado `variancia`. Em seguida, calcule a raiz quadrada desse valor e salve em um objeto chamado `desvio_padrao`.

- e. Compare o valor de `desvio_padrao` com `sd(airquality$Ozone)` e pesquise por que os valores não são iguais. Dica: veja a documentação da função `sd()`.

6. Use o *data frame* `airquality` para responder às questões abaixo.

- a. Conte quantos `NAs` tem na coluna `Solar.R`.
- b. Filtre a tabela `airquality` com apenas linhas em que `Solar.R` é `NA`.
- c. Filtre a tabela `airquality` com apenas linhas em que `Solar.R` **não** é `NA`.
- d. Filtre a tabela `airquality` com apenas linhas em que `Solar.R` **não** é `NA` e `Month` é igual a 5.

3.11 Mais sobre funções

Funções são tão comuns e intuitivas (provavelmente você já usou funções no Excel), que mesmo sem termos abordado o tema com detalhes, nós conseguimos utilizar várias 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 família de funções `is.na()`, `is.nan()`, `is.infinite()` e `is.null` foram utilizadas para testar se um valor é `NA`, `NaN`, infinito ou `NULL`, respectivamente.

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
```

Como cada coluna de um *data frame* é um vetor, podemos calcular a média de uma coluna fazendo:

```
# Podemos passar esse vetor para a função mean()
mean(mtcars$mpg)
## [1] 20.09062
```

Também podemos usar argumentos para modificar o comportamento de uma função. O que acontece se algum elemento do vetor for `NA`?

```
mean(c(1, 3, NA))
## [1] NA
```

Como a função não sabe o valor do terceiro elemento do vetor, ela não sabe qual é a média desses 3 elementos e, então, devolve `NA`. Como é muito comum termos `NA` nas nossas bases de dados, é muito comum tentarmos calcular a média de uma coluna que tem `NA` e recebermos `NA` como resposta.

Na grande maioria dos casos, queremos saber a média de uma coluna apesar dos `NAs`. Isto é, queremos retirar os `NAs` e então calcular a média com os valores que conhecemos. Para isso, podemos utilizar o argumento `na.rm = TRUE` da função `mean()`.

```
mean(c(1, 3, NA), na.rm = TRUE)
## [1] 2
```

Esse argumento diz à função para remover os NAs antes de calcular a média.
Assim, a média calculada é: $(1 + 3)/2$.

Claro que cada função tem os seus próprios argumentos e nem toda função terá o argumento `na.rm=`. Para saber quais são e como usar os argumentos de uma função, basta acessar a sua documentação:

```
help(mean)
```

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
```

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.

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`, `xe` e `y` são descartados após a função ser executada.

Claro que, na prática, é sempre bom criarmos funções que retornem na tela os
seus resultados, para evitar esse passo a mais se quisermos apenas ver o
resultado no console. Assim, a função `minha_soma()` costuma ser preferível
com relação à função `minha_nova_soma()`.

Exercícios

1. Qual dos códigos abaixo devolverá um erro se for avaliado?
 - a. `3 * 5 + 10`
 - b. `function <- 10`
 - c. `mean(1, 10)`
 - d. `(soma <- sum(1, 1))`
2. Crie uma função que receba um número e retorne o quadrado deste número.
3. Crie uma função que receba 2 números e devolva a raiz quadrada da soma
desses números.
4. Crie uma função que receba dois valores (numéricos) e devolva o maior
deles.
5. Use a função `runif()` para criar uma função que retorne um número
aleatório inteiro entre 0 e 10 (0 e 10 inclusive). Caso você não conheça a
função `runif()`, rode `help(runif)` para ler a sua documentação.
6. Rode `help(sample)` para descobrir o que a função `sample()` faz. Em
seguida
 - a. use-a para escrever uma função que devolva uma linha aleatória de um
data frame;
 - b. generalize a função para retornar um número qualquer de linhas, es-
colhido pelo usuário.

3.12 Controle de Fluxo

Como toda boa linguagem de programação, o R possui estruturas de `if`, `else`,
`for` e `while`. Esses **controles de fluxo** são muito importantes na hora de

programar, pois nos permitem manipular de modo eficiente as ações do computador.

A seguir, explicaremos para que servem e como utilizar cada uma dessas estruturas.

3.12.1 Condicionamento: if e else

As estruturas `if` e `else` servem para executarmos um código apenas se uma condição (teste lógico) for satisfeita.

No código abaixo, a função `Sys.time()`, que retorna a data/hora no momento da execução, só será avaliada se o objeto `x` for igual a 1.

```
# Não vai executar a função Sys.time()
x <- 2

if (x == 1) {
  Sys.time()
}

# Vai executar a função Sys.time()
x <- 1

if (x == 1) {
  Sys.time()
}
## [1] "2021-07-14 14:31:30 -03"
```

O R só vai executar o que está na expressão dentro das chaves {} se a expressão que estiver dentro dos parênteses () retornar TRUE. Veja outro exemplo:

```
# Vai fazer a soma
x <- c(1, 3, 10, 15)

if (class(x) == "numeric") {
  sum(x)
}
## [1] 29

# Não vai fazer a soma
x <- c("a", "b", "c")

if (class(x) == "numeric") {
```

```
    sum(x)
}
```

Nesse exemplo, a soma só é executada se a classe do objeto `x` for numérica, isto é, se `x` for um vetor de números. Essa verificação poderia ser colocada dentro de uma função para evitarmos que ela retorne um erro.

```
minha_soma <- function(x, y) {
  if (class(x) == "numeric" & class(y) == "numeric") {
    x + y
  }
}

# Retorna a soma
minha_soma(1, 2)
## [1] 3

# Não retorna nada
minha_soma("a", "b")
```

Nesses casos, é muito comum o uso das funções `return()` e `stop()` para, respectivamente, retornar um resultado antecipadamente ou para a execução da função e devolver ao usuário uma mensagem de erro personalizada.

Um exemplo usando `return()`.

```
# Devolvendo um resultado antecipadamente
minha_soma_NA <- function(x, y) {
  if (class(x) == "numeric" & class(y) == "numeric") {
    soma <- x + y
    return(soma)
  }

  NA
}

# Retorna a soma
minha_soma_NA(1, 2)
## [1] 3

# Retorna NA
minha_soma_NA("a", "b")
## [1] NA

# Retorna NA
```

```
minha_soma_NA(1, "b")
## [1] NA
```

Na função `minha_soma_NA()`, a soma só é calculada e retornada se `x` e `y` forem numéricos. Caso pelo menos um dos dois não seja, o código dentro do `if` não é executado e o valor retornado é o `NA`.

Agora, usando `stop()`.

```
# Agora, devolvendo um erro
minha_soma_erro <- function(x, y) {
  if (class(x) != "numeric" | class(y) != "numeric") {
    stop("A classe dos objetos x e y deve ser numérica.")
  }

  x + y
}

# Retorna a soma
minha_soma_erro(1, 2)
## [1] 3

# Retorna erro
minha_soma_erro("a", "b")
## Error in minha_soma_erro("a", "b"): A classe dos objetos x e y deve ser numérica.

# Retorna erro
minha_soma_erro(1, "b")
## Error in minha_soma_erro(1, "b"): A classe dos objetos x e y deve ser numérica.
```

Na função `minha_soma_erro()`, testamos no `if` se a classe de `x` ou a classe de `y` é diferente de `numeric`, isto é, se pelo menos um dos dois não é um número.

Se esse teste retornar `TRUE`, a função para a sua execução e devolve para o usuário a seguinte mensagem de erro: “A classe dos objetos `x` e `y` deve ser numérica.”. Se o teste retorna `FALSE`, a soma é realizada e seu resultado nos é retornado.

O `else` funciona como uma extensão do `if`, dando uma alternativa caso o teste executado seja falso.

```
# Vai fazer a soma
x <- c(1, 3, 10, 15)

if (class(x) == "numeric") {
  sum(x)
```

```

} else {
  NA
}
## [1] 29

# Vai retornar NA
x <- c(1, 3, 10, "15")

if (class(x) == "numeric") {
  sum(x)
} else {
  NA
}
## [1] NA

```

Também podemos usar o `else` para encadear vários `ifs`. Teste o código abaixo com valores positivos e negativos para `x`.

```

x <- 0

if(x < 0) {

  "negativo"

} else if(x == 0) {

  "neutro"

} else if(x > 0) {

  "positivo"
}
## [1] "neutro"

```

Repare que o `if` no último `else` poderia ser omitido.

```

x <- 0

if(x < 0) {

  "negativo"

} else if(x == 0) {

  "neutro"
}

```

```

} else {

  "positivo"
}
## [1] "neutro"

```

3.12.2 Iteradores: for e while

O **for** pode ser utilizado para fazer os famosos *loopings* de programação, isto é, repetir uma mesma tarefa para um conjunto de valores diferentes. Cada repetição é chamada de iteração e o objeto que muda de valor em cada interação é chamado de **iterador**.

```

numero_de_colunas <- ncol(mtcars)

for (coluna in 1:numero_de_colunas) {
  media <- mean(mtcars[,coluna])

  print(media)
}
## [1] 20.09062
## [1] 6.1875
## [1] 230.7219
## [1] 146.6875
## [1] 3.596563
## [1] 3.21725
## [1] 17.84875
## [1] 0.4375
## [1] 0.40625
## [1] 3.6875
## [1] 2.8125

```

O código acima vai calcular a média de cada coluna do *data frame* `mtcars`.
Alguns pontos importantes:

- No exemplo, temos 11 iterações e o objeto `coluna` é o iterador.
- Como `numero_de_colunas` é igual a 11, a expressão `1:numero_de_colunas` cria uma sequência de números de 1 a 11.
- A expressão `coluna in 1:numero_de_colunas` indica que o valor de `coluna` será 1 na primeira iteração, 2 na segunda iteração, 3 na terceira e assim por diante.

- O código dentro do `for` não é retornado para o usuário ao fim de cada iteração. Por isso, para ver os resultados no Console, usamos a função `print()`.

Também podemos salvar as médias em um vetor.

```
numero_de_colunas <- ncol(mtcars)

# Antes, criamos um vetor vazio.
medias <- c()

for (coluna in 1:numero_de_colunas) {
  medias[coluna] <- mean(mtcars[,coluna])
}

medias
## [1] 20.090625 6.187500 230.721875 146.687500 3.596563 3.217250
## [7] 17.848750 0.437500 0.406250 3.687500 2.812500
```

Assim como o `for`, o `while` também é um iterador.

O código a seguir irá imprimir na tela o valor de `i` enquanto este objeto for menor que 3. No momento em que a condição dentro das chaves {} não for mais respeitada, o processo será interrompido.

```
i <- 1

while (i < 3){
  print(i)
  i <- i + 1
}
## [1] 1
## [1] 2
```

É importante que o valor de `i` seja atualizado em cada interação, caso contrário a função entrará em um loop infinito. Por isso fazemos `i <- i + 1` após o `print`.

Exercícios

1. Por que o código abaixo retorna erro? Arrume o código para retornar o valor `TRUE`.

```
x <- 4
if(x == 4) {
  TRUE
}
```

2. Usando `if` e `else`, escreva um código que retorne a string "número" caso o valor seja da classe `numeric` ou `integer`; a string "palavra" caso o valor seja da classe `character`; e `NA` caso contrário.
3. Usando apenas `for` e a função `length()`, construa uma função que calcule a média de um vetor número qualquer. Construa uma condição para a função retornar `NULL` caso o vetor não seja numérico.
4. Utilize o vetor `a` para resolver as questões a seguir:

```
a <- c(10, 3, 5, -1, 3, -4, 8, 9, -10)
```

- a. Utilize o `for` para imprimir as médias acumuladas do vetor `a`, isto é, primeiro vamos imprimir 10, depois a média entre 10 e 3, depois a média entre 10, 3 e 5 e assim por diante.
- b. Adapte o laço que você fez no item anterior para ignorar os valores negativos, isto é, em caso de valor negativo, o laço não deve calcular a média e não imprimir nada.

3.13 Outros tópicos

Nesta seção, apresentamos alguns tópicos extras. Alguns deles serão retomados em capítulos posteriores.

3.13.1 Matrizes

As matrizes no R podem ser tratadas como vetores com duas dimensões. Por serem vetores, elas só podem conter elementos de uma mesma classe. Por possuírem duas dimensões, as operações de *subsetting* devem ser realizadas utilizando a sintaxe `matriz[linha, coluna]`.

Para criar uma matriz, utilizamos a função `matrix()`. Precisamos definir quais elementos formarão a matriz e qual será o número de linhas e colunas.

```
# Uma matriz de 2 linhas e 3 colunas
m <- matrix(1:9, nrow = 3, ncol = 3)

m
```

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

dim(m)
## [1] 3 3
```

Repare que os números de 1 a 9 foram dispostos na matriz coluna por coluna (*column-wise*), ou seja, preenchendo de cima para baixo e depois da esquerda para a direita. Esse comportamento pode ser alterado se utilizarmos o argumento `byrow=`.

```
matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
##      [,1] [,2] [,3]
## [1,]     1     2     3
## [2,]     4     5     6
## [3,]     7     8     9
```

Subsetting de matrizes é muito parecido com o de *data frames*.

```
# Seleciona a terceira linha
m[3, ]
## [1] 3 6 9

# Seleciona a segunda coluna
m[, 2]
## [1] 4 5 6

# Seleciona o primeiro elemento da segunda coluna
m[1, 2]
## [1] 4
```

A seguir, apresentamos algumas operações úteis para trabalhar com matrizes.

```
# Matriz transposta
t(m)
##      [,1] [,2] [,3]
## [1,]     1     2     3
## [2,]     4     5     6
## [3,]     7     8     9

# Matriz identidade 3 por 3
n <- diag(3)
```

```

n
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1

# Multiplicação por escalar
n * 2
##      [,1] [,2] [,3]
## [1,]    2    0    0
## [2,]    0    2    0
## [3,]    0    0    2

# Multiplicação matricial
m %*% n
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

# Matriz inversa de m
n2 <- n * 2
solve(n2)
##      [,1] [,2] [,3]
## [1,]  0.5  0.0  0.0
## [2,]  0.0  0.5  0.0
## [3,]  0.0  0.0  0.5

```

3.13.2 Fatores

Fatores são uma classe de objetos no R criada para representar as variáveis categóricas numericamente. Essa classe possui um atributo especial: os **levels**. *Levels* são nada mais do que as categorias possíveis de uma variável categórica.

Como exemplo, imagine que o objeto **sexo** guarde uma coluna que indica o sexo de uma pessoa: F para feminino e M para masculino. Normalmente, essa coluna seria importada para o R como texto. Podemos transformá-la em fator utilizando a função **as.factor()**.

```

# Variável sexo como texto
sexo <- c("F", "M", "M", "M", "F", "F", "M")
sexo
## [1] "F"  "M"  "M"  "M"  "F"  "F"  "M"

```

```
# Variável sexo, como fator
as.factor(sexo)
## [1] F M M M F F M
## Levels: F M
```

Repare que a saída do objeto `sexo` quando o transformamos em fator tem uma informação a mais. Na última linha, visualizamos os `levels` desse fator, isto é, um conjunto das categorias possíveis do fator `sexo` (no caso, F e M).

Por padrão, os **levels são ordenados por ordem alfabética**. Veremos mais adiante que isso pode fazer diferença na construção de gráficos e na aplicação de modelos.

A diferença entre fatores e texto dentro do R é como eles são representados internamente. Enquanto objetos da classe `character` realmente são representados como texto, fatores são representados como números inteiros.

```
# Em geral, não é possível transformar textos em números
as.numeric(sexo)
## Warning: NAs introduced by coercion
## [1] NA NA NA NA NA NA NA NA

# Mas podemos transformar fatores em inteiros
fator <- as.factor(sexo)
as.numeric(fator)
## [1] 1 2 2 2 1 1 2
```

Internamente, cada level de um fator é representado como um inteiro. No exemplo anterior, o level F está sendo representado como 1 e o level M como 2.

Se um fator tiver 10 levels, teremos os inteiros de 1 a 10 representando esse fator.

```
# letters é um objeto nativo do R
letras <- letters[1:10]

fator <- as.factor(letras)
fator
## [1] a b c d e f g h i j
## Levels: a b c d e f g h i j

as.numeric(fator)
## [1] 1 2 3 4 5 6 7 8 9 10
```

O texto que vemos quando avaliamos um fator (F e M em vez de 1 e 2, por exemplo) é apenas uma “etiqueta” que o R coloca em cima dos inteiros. As diferentes etiquetas de um fator são justamente os `levels`.

Como fatores são sempre representados internamente por inteiros sequencias começando do 1 (1, 2, 3, ...) e esses inteiros são sempre atribuídos conforme a ordem alfabética dos `levels`, um erro muito comum é tentar transformar levels numéricos em números:

```
# Texto
vetor <- c("10", "55", "55", "12", "10", "-5", "-90")
vetor
## [1] "10"  "55"  "55"  "12"  "10"  "-5"  "-90"

# Fator
fator <- as.factor(vetor)
fator
## [1] 10 55 55 12 10 -5 -90
## Levels: -5 -90 10 12 55

# Número
as.numeric(fator)
## [1] 3 5 5 4 3 1 2
```

Quando transformamos o objeto `vetor` em um fator, o R não enxerga os “números” dentro dele. Para o R, é tudo texto. Então, como nos outros exemplos, cada “número” será representado por um inteiro, atribuído pela ordem alfabética.

Uma forma de evitar esse problema é transformar o fator em texto antes de transformá-lo em número.

```
# Voltando para texto
texto <- as.character(fator)
texto
## [1] "10"  "55"  "55"  "12"  "10"  "-5"  "-90"

# Agora sim os números originais
as.numeric(texto)
## [1] 10 55 55 12 10 -5 -90
```

3.13.3 Gráficos (R base)

O R já vem com funções básicas que fazem gráficos estatísticos de todas as naturezas.

- Vantagens: são rápidas e simples.
- Desvantagens: os gráficos são simplórios e geralmente é difícil gerar gráficos mais elaborados.

Nesta seção, mostraremos como construir alguns tipos de gráficos usando as funções base do R, mas o nosso foco em visualização de dados está nas funções do pacote `ggplot2`, apresentadas no Capítulo 8.

Gráfico de dispersão e linhas

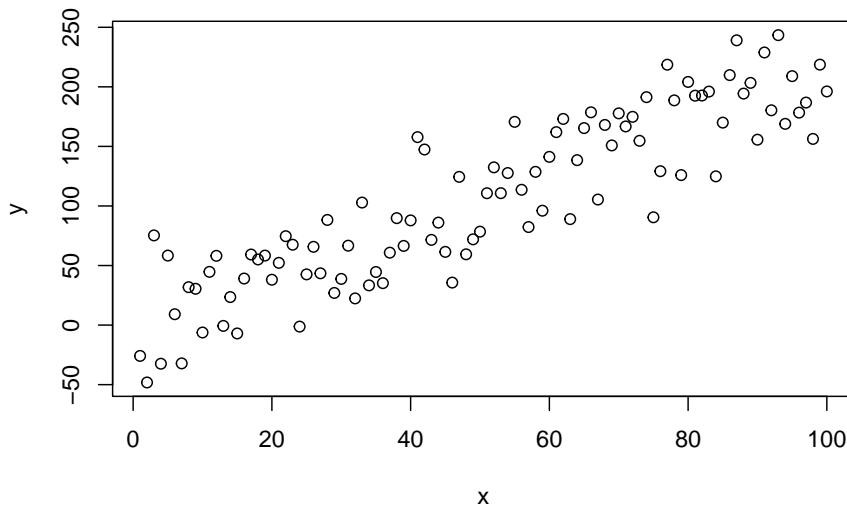
Para construir um gráfico de dispersão, utilizamos a função `plot()`. Seus principais parâmetros são:

- `x, y`: vetores para representarem os eixos x e y.
- `type`: tipo de gráfico. Pode ser pontos, linhas, escada, entre outros.

Para mais detalhes sobre os argumentos, ver `help(plot)`.

```
N <- 100
x <- 1:N
y <- 5 + 2 * x + rnorm(N, sd = 30)

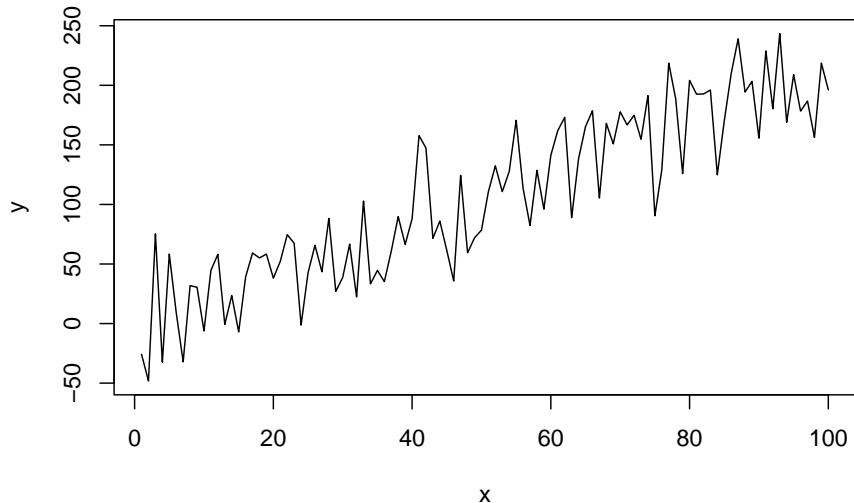
plot(x, y)
```



No código acima, a função `rnorm()` gera uma amostra aleatória da distribuição Normal com média 0 e desvio-padrão 30.

O parâmetro `type = "l"` indica que queremos que os pontos sejam interligados por linhas.

```
plot(x, y, type = "l")
```

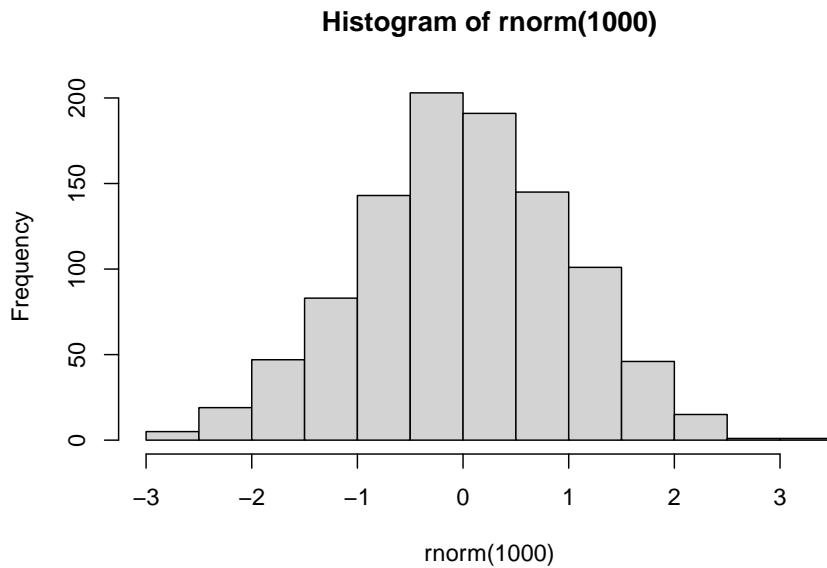


Histograma

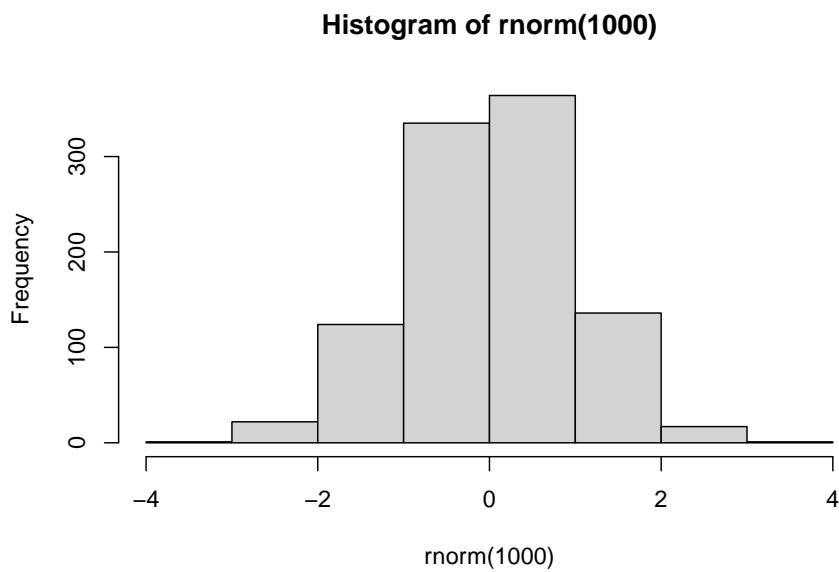
Para construir histogramas, utilizamos a função `hist()`. Os principais parâmetros são:

- `x`: o vetor numérico para o qual o histograma será construído.
- `breaks`: número (aproximado) de retângulos.

```
hist(rnorm(1000))
```



```
hist(rnorm(1000), breaks = 6)
```

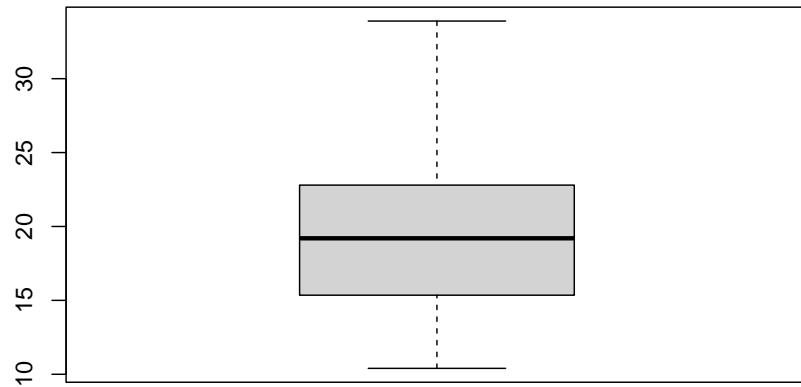


Boxplot

Para construir boxplots, utilizamos a função `boxplot()`. Os principais parâmetros são:

- `x`: o vetor numérico para o qual o boxplot será construído.

```
boxplot(mtcars$mpg, col = "lightgray")
```



Observe que o argumento `col=` muda a cor da caixa do boxplot.

Para mapear duas variáveis ao gráfico, passamos um *data frame* para o argumento `data=` e utilizamos a seguinte sintaxe `var_numerica ~ var_categorica`.

```
boxplot(mpg ~ cyl, data = mtcars, col = "purple")
```

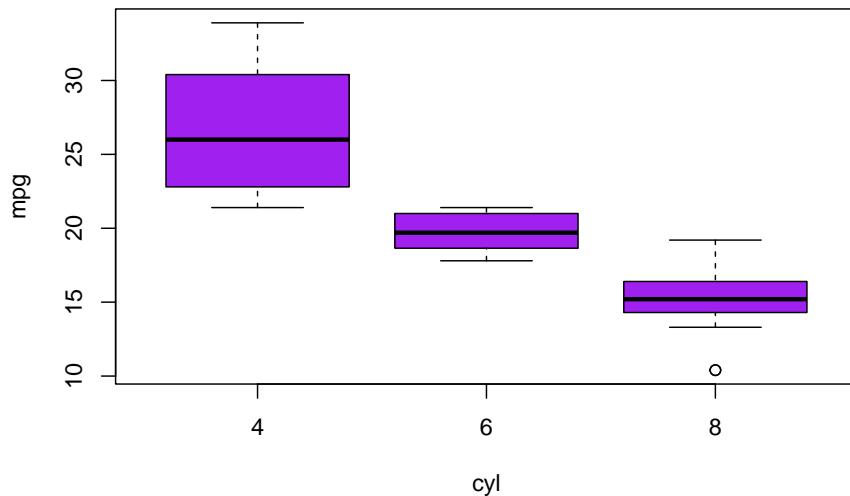
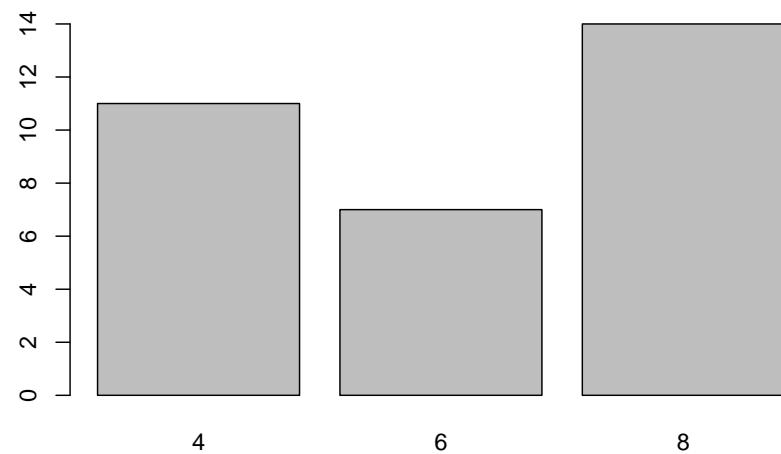


Gráfico de barras

Para construir gráficos de barras, precisamos combinar as funções `table()` e `barplot()`.

No gráfico abaixo, primeiro criamos uma tabela de frequências com a função `table()` e, em seguida, construímos o gráfico com a função `barplot()`.

```
tabela <- table(mtcars$cyl)
tabela
## 
##  4   6   8
## 11   7 14
barplot(tabela)
```



Chapter 4

Pacotes

A primeira aparição da linguagem de programação R foi em 1993 e, apesar de lá para cá muita coisa ter sido desenvolvida e atualizada, é muito difícil fazer mudanças na base da linguagem sem quebrar os códigos já existentes. Por essa razão, as maiores inovações são realizadas na forma de pacotes.

Um pacote é um conjunto de funções que têm como objetivo resolver um problema específico. São eles que deixam o R poderoso, capaz de enfrentar qualquer tarefa de análise de dados. Assim, fique bastante à vontade para instalar e atualizar muitos e muitos pacotes ao longo da sua experiência com o R.

O legal é que qualquer pessoa pode fazer um novo pacote e disponibilizar para a comunidade, o que acelera bastante o desenvolvimento da ferramenta. Difícilmente você vai fazer uma análise apenas com as funções básicas do R e dificilmente não vai existir um pacote com as funções que você precisa.

4.1 Instalação de pacotes

Existem três principais maneiras de instalar pacotes. Em ordem de frequência, são:

- Via CRAN (Comprehensive R Archive Network): `install.packages("nome-do-pacote")`.
- Via Github: `devtools::install_github("nome-do-repo/nome-do-pacote")`.
- Via arquivo .zip/.tar.gz: `install.packages("C:/caminho/nome-do-pacote.zip", repos = NULL)`.

Para conseguir instalar alguns pacotes no Linux, você pode precisar instalar dependências do sistema manualmente. Por exemplo, se você quer instalar o

pacote devtools no R, será necessário ter as bibliotecas `curl`, `openssl`, `httr` e `git2r`.

Essas dependências geralmente podem ser instaladas no terminal por meio do comando `apt-get install nome-da-biblioteca`. Caso você não consiga instalar um pacote devido a ausência de uma dependência, uma maneira de saber quais bibliotecas você precisa instalar é observar as mensagens que aparecem no console durante a tentativa da instalação do pacote.

4.1.1 Via CRAN

Instale pacotes que não estão na sua biblioteca usando a função `install.packages("nome_do_pacote")`. Por exemplo:

```
install.packages("tidyverse")
```

E, de agora em diante, não precisa mais instalar. Basta carregar o pacote com `library(magrittr)`.

Escreva `nome_do_pacote:::nome_da_funcao()` se quiser usar apenas uma função de um determinado pacote. O operador `::` serve para isso. Essa forma também é útil quando se tem duas funções com o mesmo nome e precisamos garantir que o código vá usar a função do pacote correto.

4.1.2 Via Github

Desenvolvedores costumam disponibilizar a última versão de seus pacotes no Github, e alguns deles sequer estão no CRAN. Mesmo assim ainda é possível utilizá-los instalando diretamente pelo github. O comando é igualmente simples:

```
devtools::install_github("rstudio/shiny")
```

Apenas será necessário o `username` e o nome do repositório (que geralmente tem o mesmo nome do pacote). No exemplo, o `username` foi “rstudio” e o repositório foi “shiny”.

Se você não é familiar com o github, não se preocupe! Os pacotes disponibilizados na plataforma geralmente têm um `README` cuja primeira instrução é sobre a instalação. Se não tiver, provavelmente este pacote não te merece! =)

4.1.3 Via arquivo .zip ou .tar.gz

Se você precisar instalar um pacote que está zipado no seu computador (ou em algum servidor), utilize o seguinte comando:

```
install.packages("C:/caminho/para/o/arquivo/zipado/nome-do-pacote.zip", repos = NULL)
```

É semelhante a instalar pacotes via CRAN, com a diferença que agora o nome do pacote é o caminho inteiro até o arquivo. O parâmetro `repos = NULL` informa que estamos instalando a partir da máquina local.

A aba **Packages** do RStudio também ajuda a administrar os seus pacotes.

4.2 Tidyverse

Muitas pessoas tentam definir *o que é* ciência de dados no mercado e na academia. O problema é que esse termo pode ser descrito de várias formas distintas, seja pela formação específica da pessoa que define ou do interlocutor ao qual ela se comunica. Por isso, a definição de ciência de dados é, de certa forma, vazia.

No entanto, é possível definir *como se faz* ciência de dados. Ou seja, independentemente da definição do termo, o que temos de fazer na prática em projetos reais é algo bastante conhecido.

O “como faz” é definido através do *Ciclo da Ciência de Dados*, descrito na Figura 4.1. Primeiro, os dados brutos são coletados de fontes públicas, como arquivos Excel, portais de dados abertos ou bases de dados internos da companhia. Em seguida, os dados são arrumados, para mitigar problemas de padronização de nomes, obtenção das variáveis de interesse e exclusão de casos que estão fora do escopo de análise, produzindo o que se define como base de dados analítica. A base analítica é então transformada para produzir as tabelas e gráficos e, quando necessário, são utilizadas como insumo para o ajuste de modelos estatísticos. Finalmente, os resultados obtidos são comunicados através de uma série de ferramentas, como relatórios, dashboards interativos, indicadores ou Application Programming Interfaces (API) para automação.

O `{tidyverse}` é um pacote guarda-chuva que consolida uma série de ferramentas que fazem parte o ciclo da ciência de dados. Fazem parte do `{tidyverse}` os pacotes `{ggplot2}`, `{dplyr}`, `{tidyverse}`, `{purrr}`, `{readr}`, entre muitos outros, como é possível observar na Figura 4.2. Veremos as características principais desses pacotes nas próximas Seções.

O `{tidyverse}` traz consigo o *manifesto tidy*. Trata-se de um documento que formaliza uma série de princípios que norteiam o desenvolvimento do tidyverse. Como os pacotes do `{tidyverse}` compartilham os mesmos princípios, podem ser utilizados naturalmente em conjunto.

Importar



Arrumar

(Armazenar os dados
consistentemente)



Transformar

(Criar novas variáveis
agregações)

Comunicar

Figure 4.1: O Ciclo da Ciência de Dados.

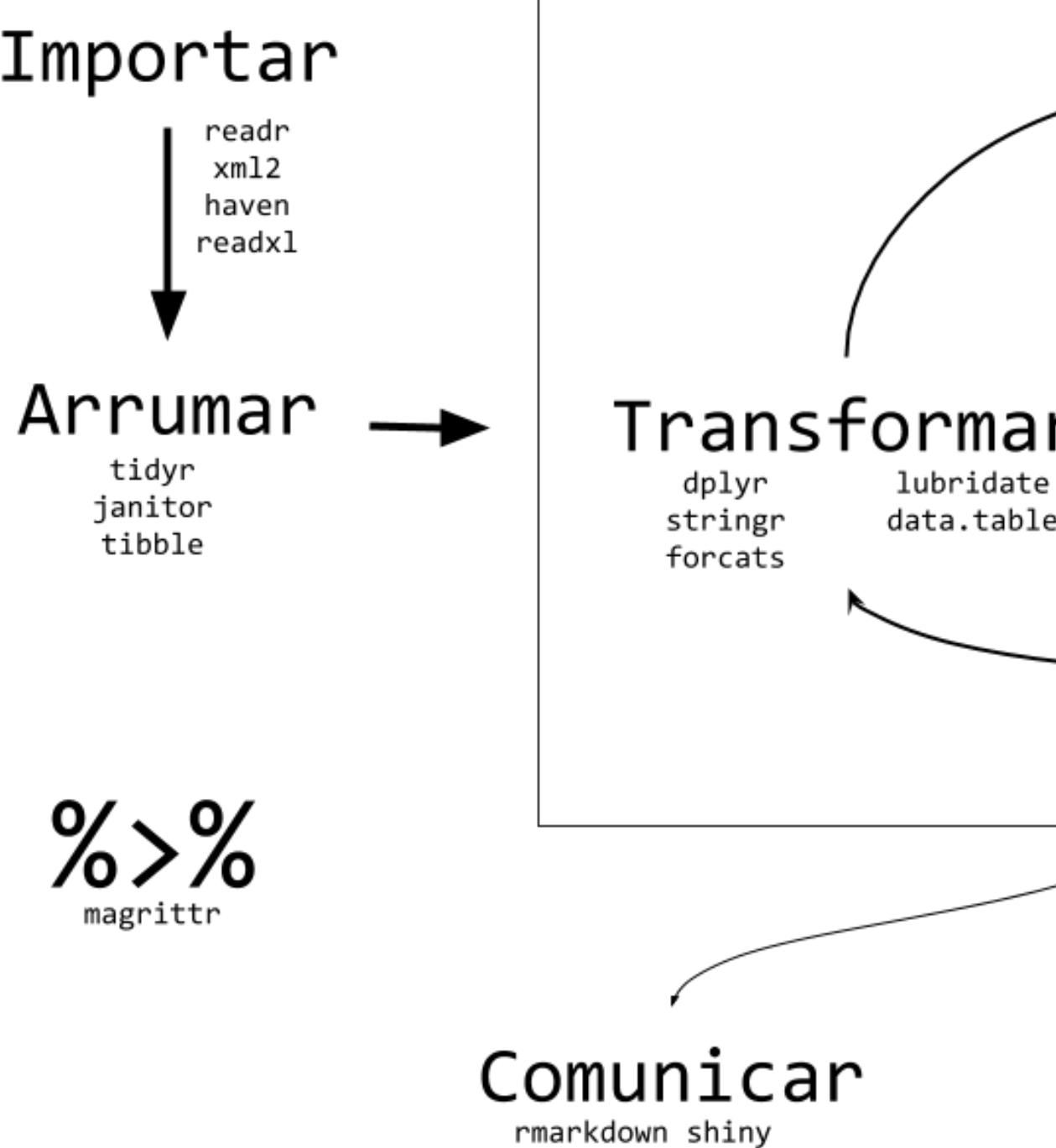


Figure 4.2: A implementação do Ciclo da Ciência de Dados, através do tidyverse. Pela definição estrita do tidyverse, na imagem não fazem parte do tidyverse os pacotes janitor, data.table e os pacotes descritos nas partes de modelagem, comunicação e automatização. No entanto, a maioria desses pacotes também seguem os princípios tidy e podem ser usados em conjunto com o tidyverse sem grandes dificuldades.

Pode-se dizer que existe uma linguagem R antes e outra depois do `{tidyverse}`. A linguagem mudou muito, a comunidade abraçou uso desses princípios e criou centenas de novos pacotes que conversam uns com os outros dessa forma.¹

Os princípios fundamentais do tidyverse são:

1. Reutilizar estruturas de dados existentes.
2. Organizar funções simples usando o *pipe* (Seção 6).
3. Aderir à programação funcional (Seção 10).
4. Projetado para ser usado por seres humanos.

No texto do manifesto tidy cada um dos lemas é descrito de forma detalhada. No nosso blog, selecionamos os aspectos que achamos mais importante de cada um deles.

¹Usar a filosofia tidy não é a única forma de fazer pacotes do R. Existem muitos pacotes excelentes que não utilizam essa filosofia. O próprio manifesto diz “O contrário de tidyverse não é o messyverse, e sim muitos outros universos de pacotes interconectados.”



Na prática, carregar o (veja o código abaixo) é o mesmo que carregar os seguintes pacotes:

- `{tibble}` para *data frames* repaginados;
- `{readr}` para importarmos bases para o R;
- `{tidyverse}` e `{dplyr}` para arrumação e manipulação de dados;
- `{stringr}` para trabalharmos com textos;
- `{forcats}` para trabalharmos com fatores;
- `{ggplot2}` para gráficos;
- `{purrr}` para programação funcional.

Embora o `{tidyverse}` instale diversos outros pacotes, apenas esses são carregados. Dificilmente fazemos uma análise de dados em que não precisamos usá-los. Falaremos com mais detalhes de todos eles neste livro.

```
library(tidyverse)

## -- Attaching packages ----- tidyverse 1.3.1 --

## v ggplot2 3.3.5     v purrr    0.3.4
## v tibble   3.1.2     v dplyr    1.0.7
## v tidyverse 1.1.3     v stringr  1.4.0
## v readr    1.4.0     vforcats  0.5.1

## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()   masks stats::lag()
```

Mensagens de conflito quando carregamos o `{tidyverse}` ou qualquer outro pacote significam que funções anteriormente carregadas foram *mascaradas* por novas funções. No exemplo acima, as funções `filter()` e `lag()` do pacote `stats` foram substituídas na sessão pelas funções `filter()` e `lag()` do pacote `dplyr`. Nesse caso, se quiséssemos usar as funções do pacote `stats` após carregar o `{tidyverse}`, precisaríamos rodar `stats::filter()` e `stats::lag()`.

Se você quiser *descarregar* um pacote, reinicie a sua sessão em `Session > Restart R` ou use a função `detach()` como no exemplo abaixo.

```
detach("package:tidyverse", unload = TRUE)
```

Chapter 5

Importação

Nesta seção, vamos introduzir os principais pacotes para importar dados para o R. Mostraremos como importar dados de arquivos de texto, planilhas do excel e extensões de outros programas estatísticos (SAS e SPSS, por exemplo).

Antes de começarmos, vale a pena destacarmos um ponto importante. As funções de importação do `tidyverse` importam dados em objetos da classe `tibble`, que difere da classe `data.frame` usual em dois pontos importantes:

- imprime os dados na tela (Console) de maneira muito mais organizada, resumida e legível;
- permite a utilização de *list-columns*.

Se você não estiver familiarizado com o conceito de *list-columns*, não se preocupe. Trataremos melhor do assunto no Capítulo 10.

5.1 Caminhos

Um passo importante na tarefa de importação de dados para o R é saber onde está o arquivo que queremos importar. Toda função de importação vai exigir um **caminho**, uma string que representa o endereço do arquivo no computador.

Há duas formas de passarmos o caminho de arquivo: usar o **caminho absoluto** ou usar o **caminho relativo**.

Antes de falarmos sobre a diferença dos dois, precisamos definir o que é o **diretório de trabalho**.

O diretório de trabalho (*working directory*) nada mais é do que a pasta em que o R vai procurar arquivos na hora de ler informações ou gravar arquivos na hora de salvar objetos.

Se você está usando um projeto, o diretório de trabalho da sua sessão será, por padrão, a pasta raiz do seu projeto (é a pasta que contém o arquivo com extensão `.Rproj`). Se você não estiver usando um projeto ou simplesmente não souber qual é o seu diretório de trabalho, você pode descobri-lo usando a seguinte função `getwd()`. Ela vai devolver uma string com o caminho do seu diretório de trabalho.

A função `setwd()` pode ser utilizada para mudar o diretório de trabalho. Como argumento, ela recebe o caminho para o novo diretório.

Caminhos absolutos são aqueles que tem início na pasta raiz do seu computador/usuário. Por exemplo:

```
getwd()
```

```
## [1] "/Users/cliente/Documents/livro-material"
```

Esse é o caminho absoluto para a pasta onde esse livro foi produzido.

Na grande maioria dos casos, caminhos absolutos são uma **má prática**, pois deixam o código irreprodutível. Se você trocar de computador ou passar o script para outra pessoa rodar, o código não vai funcionar, pois o caminho absoluto para o arquivo muito provavelmente será diferente.

Caminhos relativos são aqueles que tem início no diretório de trabalho da sua sessão. Assim, se você quiser acessar um arquivo `minha_base.csv` dentro de uma pasta “dados” existente no seu diretório de trabalho, você poderia passar para o R o caminho `"dados/minha_base.csv"`.

Trabalhar com projetos no RStudio ajuda bastante o uso de caminhos relativos, pois nos incentiva a colocar todos os arquivos da análise dentro da pasta do projeto. Assim, se você usar apenas caminhos relativos e compartilhar a pasta do projeto com alguém, todos os caminhos existentes nos códigos continuarão a funcionar em qualquer computador!

5.2 O pacote `readr`

O pacote `{readr}` do tidyverse é utilizado para importar arquivos de texto, como `.txt` ou `.csv`, para o R. Para carregá-lo, rode o código:

```
library(readr)
```

O `{readr}` transforma arquivos de textos em `tibbles` usando as funções:

- `read_csv()`: para arquivos separados por vírgula.
- `read_tsv()`: para arquivos separados por tabulação.
- `read_delim()`: para arquivos separados por um delimitador genérico. O argumento `delim=` indica qual caracter separa cada coluna no arquivo de texto.
- `read_table()`: para arquivos de texto tabular com colunas separadas por espaço.
- `read_fwf()`: para arquivos compactos que devem ter a largura de cada coluna especificada.
- `read_log()`: para arquivos padrões de log.

Vamos mostrar na próxima seção como importar as extensões mais comuns:

`.csv` e `.txt`.

5.2.1 Lendo arquivos de texto

Como exemplo, utilizaremos uma base de filmes do IMDB, gravada em diversos formatos. O download dos arquivos pode ser realizado clicando [aqui](#).

Primeiro, vamos ler a base em formato `.csv`.

```
imdb_csv <- read_csv(file = "imdb.csv")
```

```
## 
## -- Column specification -----
## cols(
##   titulo = col_character(),
##   ano = col_double(),
##   diretor = col_character(),
##   duracao = col_double(),
##   cor = col_character(),
##   generos = col_character(),
##   pais = col_character(),
##   classificacao = col_character(),
##   orcamento = col_double(),
##   receita = col_double(),
##   nota_imdb = col_double(),
##   likes_facebook = col_double(),
##   ator_1 = col_character(),
##   ator_2 = col_character(),
##   ator_3 = col_character()
## )
```

A mensagem retornada pela função indica qual classe foi atribuída para cada coluna. Repare que o argumento `file=` representa o caminho até o arquivo. Se o arquivo a ser lido não estiver no diretório de trabalho da sua sessão, você precisa especificar o caminho até o arquivo.

```
# Se o arquivo estiver dentro de uma pasta chamada dados.
imdb_csv <- read_csv(file = "dados/imdb.csv")
```

A maioria das funções de leitura do `{readr}` possuem argumentos muito úteis para resolver problemas de importação:

- `col_names=`: indica se a primeira linha da base contém ou não o nome das colunas. Também pode ser utilizado para (re)nomear colunas.
- `col_types=`: caso alguma coluna seja importada com a classe errada (uma coluna de números foi importada como texto, por exemplo), você pode usar esse argumento para especificar a classe das colunas.
- `locale=`: útil para tratar problema de *encoding*.
- `skip=`: pula linhas no começo do arquivo antes de iniciar a importação. Útil para quando o arquivo a ser importado vem com metadados ou qualquer tipo de texto nas primeiras linhas, antes da base.
- `na=`: indica quais *strings* deverão ser consideradas `NA` na hora da importação.

Em alguns países, como o Brasil, as vírgulas são utilizadas para separar as casas decimais dos números, inviabilizando o uso de arquivos `.csv`. Nesses casos, quando a vírgula é o separador de decimal, os arquivos `.csv` passam a ser separados por ponto-e-vírgula. Para importar bases de arquivos separados por ponto-e-vírgula no R, basta usar a função `read_csv2()`.

```
imdb_csv2 <- read_csv2("dados/imdb2.csv")
```

Arquivos `.txt` em geral podem ser lidos com a função `read_delim()`. Além do caminho até o arquivo, você também precisa indicar qual é o caractere utilizado para separar as colunas da base. Um arquivo separado por tabulação, por exemplo, pode ser lido utilizando o código abaixo. O código `\t` é uma forma textual de representar a tecla TAB.

```
imdb_txt <- read_delim("dados/imdb.txt", delim = "\t")
```

Repare que a sintaxe é igual a da função `read_csv()`. Em geral, as funções de importação do `{tidyverse}` terão sintaxe e comportamento muito parecidos.

A seguir, vamos falar das funções `parse_()`, muito úteis para tratar problemas na classe das variáveis na hora da importação.

5.2.2 Locale

Muitas funções de importação e formatação possuem um argumento `locale`. Esse argumento é utilizado para definir opções de formatação próprias de uma certa localidade, como idioma, formato de data e hora, fuso horário, separador de decimal e milhar ou encoding.

O pacote `{readr}` possui uma função chamada `locale()`, que pode ser utilizada para definir todos esses atributos. Para saber quais são os padrões atualmente definidos na sua sessão, basta rodar:

```
locale()
## <locale>
## Numbers: 123,456.78
## Formats: %AD / %AT
## Timezone: UTC
## Encoding: UTF-8
## <date_names>
## Days: Sunday (Sun), Monday (Mon), Tuesday (Tue), Wednesday (Wed), Thursday
##          (Thu), Friday (Fri), Saturday (Sat)
## Months: January (Jan), February (Feb), March (Mar), April (Apr), May (May),
##          June (Jun), July (Jul), August (Aug), September (Sep), October
##          (Oct), November (Nov), December (Dec)
## AM/PM: AM/PM
```

Em geral, teremos padrões norte-americanos. Se quisermos que os nomes de dias e meses fiquem em português, podemos fazer:

```
locale(date_names = "pt")
## <locale>
## Numbers: 123,456.78
## Formats: %AD / %AT
## Timezone: UTC
## Encoding: UTF-8
## <date_names>
## Days: domingo (dom), segunda-feira (seg), terça-feira (ter), quarta-feira
##          (qua), quinta-feira (qui), sexta-feira (sex), sábado (sáb)
## Months: janeiro (jan), fevereiro (fev), março (mar), abril (abr), maio (mai),
##          junho (jun), julho (jul), agosto (ago), setembro (set), outubro
##          (out), novembro (nov), dezembro (dez)
## AM/PM: AM/PM
```

Ou trocar o separador de decimal de ponto para vírgula, caso a base a ser importada esteja nesse formato.

```
locale(decimal_mark = ",")
## <locale>
## Numbers: 123.456,78
## Formats: %AD / %AT
## Timezone: UTC
## Encoding: UTF-8
## <date_names>
## Days: Sunday (Sun), Monday (Mon), Tuesday (Tue), Wednesday (Wed), Thursday
##          (Thu), Friday (Fri), Saturday (Sat)
## Months: January (Jan), February (Feb), March (Mar), April (Apr), May (May),
##          June (Jun), July (Jul), August (Aug), September (Sep), October
##          (Oct), November (Nov), December (Dec)
## AM/PM: AM/PM
```

A função `locale()` deve ser utilizada dentro das funções `read_()`, no argumento `locale`. Uma utilização muito comum é a definição do *encoding* do arquivo. O encoding se refere a como o computador traduz os caracteres que vemos na tela para os valores binários que ele utiliza internamente. Existem vários tipos de encoding e isso é um problema principalmente porque o Windows utiliza um encoding diferente do Linux/Mac. Você saberá que tem um problema de encoding quando letras com acento ou outros caracteres especiais ficarem desconfigurados após importar uma base para o R.

```
# Vamos produzir um problema de encoding

frase_com_acentos <- "Você comerá uma maçã amanhã à tarde"

# Vendo encoding. UTF-8 é o padrão do Linux/Mac
Encoding(frase_com_acentos)
## [1] "UTF-8"

# Forçando um novo encoding.
# latin1 é um dos padrões que funcionam no Windows.
Encoding(frase_com_acentos) <- "latin1"

# Agora temos um problema de encoding
frase_com_acentos
## [1] "VocÃª comerÃ¡; uma maÃ§Ã£o amanhÃ£o Ã  tarde"
```

Quando estivermos enfrentando esse problema, devemos dizer à função `read_()` qual o encoding deve ser utilizado no arquivo.

```
read_csv("base_que_veio_do_windows.csv", locale = locale(encoding = "latin1"))
```

O `latin1` é apenas um dos encodings que podem funcionar em arquivos do Windows. Outras sugestões são: `windows-1250`, `windows-1252`, `ISO-8859-2` e `ISO-8859-1`. Se você estiver lendo um arquivo cruado no Linux/Mac no Windows, basta usar o encoding `UTF-8`.

Eu consegui resolver 99% dos meus problemas de encoding quando passei a fingir que o Windows não existe. — Julio Trecenti

Na seção a seguir, mostramos mais alguns exemplos da função `locale()`.

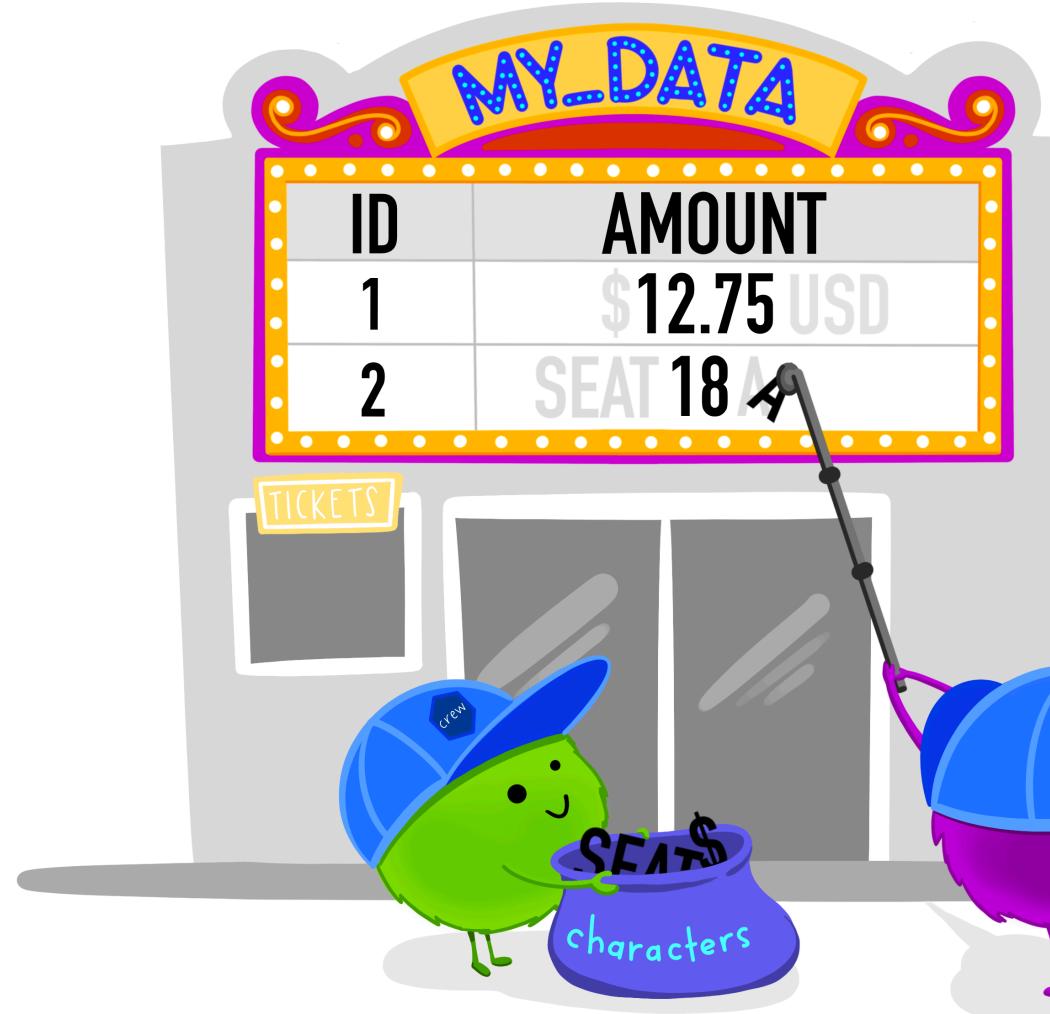
5.2.3 Parseando valores

O pacote `{readr}` possui algumas funções muito úteis para parsear valores. Parsear é um termo muito utilizado em programação e tem o sentido de *arrumar* ou *formatar*. Se estamos parseando um número, por exemplo, estamos pegando um texto que é muito parecido com um número e o transformando em um número de fato.

```
parse_number(c("5", "5.0", "5,0", "R$5.00", "5 a"))
## [1] 5 5 50 5 5
```

\begin{figure}

readr::parse_number
(just give me the number)



{

```
}
```

\caption{Arte por Allison Horst (?). Veja nas Referências onde encontrá-la.}
\end{figure}

Veja que podemos usar o argumento `locale` para especificar coordenadas para o parseamento

```
parse_number("5,0", locale = locale(decimal_mark = ","))  

## [1] 5
```

ou o idioma usado em datas

```
# Inglês  

parse_date(  

  "01/June/2010",  

  format = "%d/%B/%Y"  

)  

## [1] "2010-06-01"  
  

# Português  

parse_date(  

  "01/Junho/2010",  

  format = "%d/%B/%Y",  

  locale = locale(date_names = "pt")  

)  

## [1] "2010-06-01"
```

Você também pode especificar NAs utilizando o argumento `na`.

```
parse_number(c("5", "5.0", "5,0", "R$5.00", "5 a"), na = "5 a")  

## [1] 5 5 50 5 NA
```

Outras funções de parseamento úteis são:

- `readr::parse_integer()`, para parsear números inteiros.
- `readr::parse_character()`, para parsear strings
- `readr::parse_date()`, `readr::parse_time()`, `readr::parse_datetime()` para parsear datas, horas e data/horas.

Todas essas funções são utilizadas nas colunas de uma base quando utilizamos as funções de importação do `{readr}`. Assim, se algum valor foi modificado incorretamente durante a importação, você pode testar importar tudo como texto (`character`) e usar essas funções para reproduzir a transformação que o `{readr}` fez.

5.2.4 Escrevendo arquivos de texto

Para a maioria das funções `read_`, existe uma respectiva função `write_`. Essas funções servem para salvar bases em um formato específico de arquivo. Além do caminho/nome do arquivo a ser criado, você também precisa passar o objeto que será escrito. Para o arquivo criado funcionar corretamente, você precisa especificar a extensão correta no nome do arquivo.

```
# Arquivo .csv
write_csv(x = mtcars, path = "data/mtcars.csv")

# Base separada por tabulação
write_delim(x = mtcars, path = "data/mtcars.txt", delim = "\t")
```

5.2.5 Arquivos .rds

A linguagem R tem uma extensão própria de arquivos binários chamada `RDS` ou `.rds`. Essa extensão pode ser utilizada para guardar qualquer tipo de objeto do R, inclusive bases de dados. Temos duas principais vantagens ao utilizarmos essa extensão para salvarmos as nossas bases:

- ele salva as classes especificadas para as colunas;
- ele pode ser compactado, gerando arquivos muito menores.

A desvantagem é que ele só poderá ser lido dentro do R.

Para criar um arquivo `.rds`, utilize a função `write_rds()`.

```
write_rds(mtcars, path = "mtcars.rds", compress = "gz")
```

O argumento `compress` é opcional e indica qual o tipo de compactação deve ser feito. O padrão é não compactar.

Para ler um arquivo `.rds` de volta para o R, utilizamos a função `read_rds()`. Repare que essa função não possui outros argumentos, pois o objeto importado será exatamente igual ao objeto que foi gravado no arquivo.

```
imdb_rds <- read_rds(path = "imdb.rds")
```

Exercícios

1. Qual a diferença entre as funções `read_csv()` e `read_csv2()`?

- 2.** Leia o arquivo `imdb.csv` utilizando a função `read_delim()`.
- 3.** Escreva a base `mtcars` em um arquivo `mtcars.csv` que não contenha o nome das colunas.
- 4.** Use a função `write_rds()` para salvar em arquivos
 - **a)** Um número.
 - **b)** Um vetor de strings.
 - **c)** Uma lista com valores numéricos, textuais e lógicos.
 - **d)** As 3 primeiras colunas da base `mtcars`.
- 5.** Utilize a função `read_rds()` para importar de volta para o R os arquivos criados no exercício 4.

5.3 Os pacotes `readxl` e `writexl`

Para ler planilhas do Excel (arquivos `.xlsx` ou `.xls`), basta utilizarmos a função `read_excel()` do pacote `readxl`. Instale o pacote antes caso você ainda não o tenha instalado.

```
install.packages("readxl")
library(readxl)

imdb_xlsx <- read_xls("dados/imdb.xls")
imdb_xlsx <- read_xlsx("dados/imdb.xlsx")
```

A função `read_excel()` auto detecta a extensão do arquivo.

```
read_excel(path = "assets/data/imdb.xls")
read_excel(path = "assets/data/imdb.xlsx")
```

O pacote disponibiliza 5 exemplos de arquivos com formato `.xls` e `.xlsx`.

```
readxl_example()

## [1] "clippy.xls"      "clippy.xlsx"     "datasets.xls"    "datasets.xlsx"
## [5] "deaths.xls"      "deaths.xlsx"     "geometry.xls"   "geometry.xlsx"
## [9] "type-me.xls"     "type-me.xlsx"
```

Vamos pegar o caminho até o arquivo `datasets.xlsx` usando a função `readxl_example()`.

```
caminho_datasets <- readxl_example("datasets.xlsx")
caminho_datasets

## [1] "/Library/Frameworks/R.framework/Versions/4.1/Resources/library/readxl/extdata/
```

No Excel, um arquivo pode ter várias planilhas. Esse é o caso do arquivo `datasets.xlsx`. Você pode ver quais planilhas fazem parte do arquivo utilizando a função `excel_sheets()`.

```
excel_sheets(caminho_datasets)

## [1] "iris"      "mtcars"    "chickwts"   "quakes"
```

Por padrão, as funções de leitura trarão apenas a primeira planilha do arquivo. Para trazer outra planilha, basta utilizarmos o argumento `sheet`.

```
# Pega a primeira planilha
read_excel(caminho_datasets)
```

```
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##       <dbl>     <dbl>      <dbl>      <dbl>   <chr>
## 1         5.1      3.5        1.4       0.2  setosa
## 2         4.9      3.0        1.4       0.2  setosa
## 3         4.7      3.2        1.3       0.2  setosa
## 4         4.6      3.1        1.5       0.2  setosa
## 5         5.0      3.6        1.4       0.2  setosa
## 6         5.4      3.9        1.7       0.4  setosa
## 7         4.6      3.4        1.4       0.3  setosa
## 8         5.0      3.4        1.5       0.2  setosa
## 9         4.4      2.9        1.4       0.2  setosa
## 10        4.9      3.1        1.5       0.1 setosa
## # ... with 140 more rows
```

```
# Pega a segunda planilha
read_excel(caminho_datasets, sheet = 2)
```

```
## # 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>
## 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

# Pega a planilha selecionada
read_excel(caminho_datasets, sheet = 'chickwts')

## # A tibble: 71 x 2
##       weight feed
##       <dbl> <chr>
## 1     179 horsebean
## 2     160 horsebean
## 3     136 horsebean
## 4     227 horsebean
## 5     217 horsebean
## 6     168 horsebean
## 7     108 horsebean
## 8     124 horsebean
## 9     143 horsebean
## 10    140 horsebean
## # ... with 61 more rows

```

A seguir, listamos outros argumentos úteis da função `read_excel()`:

- `col_names` indica se a primeira linha representa o nome das colunas;
- `col_types=` para definir a classe das colunas;
- `skip=` para pular linhas no começo da planilha;
- `na=` indica quais strings devem ser interpretadas como NA.

Também podemos escrever um arquivo Excel (com extensão `.xlsx`) utilizando a função `write_xlsx()` do pacote `writexl`.

```

install.packages("writexl")
library(writexl)

write_excel(mtcars, "imdb.xlsx")

```

5.4 haven

Para ler arquivos gerados por outros softwares, como SPSS, SAS e STATA, você pode usar as funções do pacote `haven`. Este pacote faz parte do `tidyverse` e é um wrapper da biblioteca ReadStat, escrita em C.

```
library(haven)

imdb_sas <- read_sas("assets/data/imdb.sas7bdat")
imdb_spss <- read_spss("assets/data/imdb.sav")
imdb_dta <- read_dta("assets/data/imdb.dta")
```

É possível salvar ou escrever bases em SAS e STATA com as funções `write_sas` e `write_dta`.

```
write_dta(mtcars, 'assets/data/mtcars.dta')
```

Quando importamos arquivos gerados pelo SAS SPSS ou STATA para o R, os rótulos de uma variável podem não ser importados de forma correta. O pacote `haven` tem uma solução para este problema.

```
x <- labelled(c(1,1,2,3,2,2,1,2), c(Ruim = 1, Bom = 2, Otimo = 3))
```

`labelled()` adiciona rótulos à valores de uma variável. Para verificar quais são estes rótulos, podemos usar a função `print_labels()`.

```
print_labels(x)
```

```
## 
## Labels:
##   value label
##     1  Ruim
##     2  Bom
##     3  Otimo
```

Existe uma função similar a `labelled()`, exclusiva para o SPSS, que além de rotular as variáveis, também define quais símbolos representam valores faltantes, dado que em SPSS pode haver mais de um tipo de *missing*.

```
x1 <- labelled_spss(c(1,3,0,2,2,1,0,2,4), c(Ruim = 1,Bom = 2, Otimo = 3), na_values =
is.na(x1))
```

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

Agora que já definimos os *missings* “especiais”, podemos transformá-los no *missing* padrão do R, representado pelo símbolo *NA*.

```
x1 <- zap_missing(x1)
x1

## [1] 1 3 NA 2 2 1 NA 2 NA
## attr(,"labels")
## Ruim   Bom Otimo
##      1       2       3
## attr(,"class")
## [1] "haven_labelled"
```

Existem outras funções `zap_` interessantes no pacote.

Após rotular os valores do vetor, podemos convertê-los, por exemplo, em fator. Para isso, usamos uma função do pacote `haven`. A função base `as.factor()` também poderia ser usada, mas quando a usamos, os rótulos não são considerados.

```
x_base <- base::as.factor(x)
levels(x_base)

## [1] "1" "2" "3"

x_factor <- haven::as_factor(x)
levels(x_factor)

## [1] "Ruim"   "Bom"    "Otimo"
```

