

Introdução à Ciência de Dados em R

Gustavo Jun Yakushiji

Novembro, 2021

Sumário

| | | |
|----------|---|-----------|
| 1 | Introdução | 5 |
| 2 | Ciência de dados e R | 7 |
| 2.1 | O que é Ciência de Dados? | 7 |
| 2.2 | R / RStudio | 8 |
| 2.3 | Etapas da Ciência de Dados | 13 |
| 2.4 | Pacote <code>tidyverse</code> | 14 |
| 3 | Noções básicas em R | 17 |
| 3.1 | Projetos | 17 |
| 3.2 | Ajuda | 18 |
| 3.3 | Comentários | 19 |
| 3.4 | Operações matemáticas | 20 |
| 3.5 | Objetos | 22 |
| 3.6 | Funções | 23 |
| 3.7 | Classes | 24 |
| 3.8 | Data frames | 25 |
| 3.9 | Vetores | 28 |
| 3.10 | Fatores | 31 |
| 3.11 | Operações lógicas | 33 |
| 3.12 | Valores especiais | 36 |
| 3.13 | Listas | 38 |
| 4 | Importação | 45 |
| 4.1 | Pacote <code>readr</code> | 45 |
| 4.2 | Pacote <code>readxl</code> | 52 |
| 4.3 | Banco de dados | 55 |

| | | |
|----------|--------------------------------------|------------|
| 5 | Organização | 57 |
| 5.1 | Tibbles | 57 |
| 5.2 | Pacote <code>tidyr</code> | 64 |
| 6 | Transformação | 73 |
| 6.1 | Pacote <code>dplyr</code> | 73 |
| 7 | Visualização | 89 |
| 7.1 | Gráfico de Dispersão | 90 |
| 7.2 | Gráfico de Barras | 124 |
| 7.3 | Gráfico de Setores (Pizza) | 136 |
| 7.4 | Gráfico de Linhas | 141 |
| 7.5 | Gráficos de medidas-resumo | 163 |
| 7.6 | Juntar gráficos diferentes | 171 |
| | Referências | 177 |

Capítulo 1

Introdução

A proposta desta apostila é compartilhar experiências sobre ciência de dados na linguagem de programação R, visando auxiliar àqueles que estão iniciando sua jornada nesta empreitada.

Antes de começarmos, levantaremos algumas perguntas norteadoras:

- O que é ciência de dados (*Data Science*)?
- Quais são os primeiros passos a serem dados em R?
- Quais as principais ferramentas aplicadas à ciência de dados em R?

Ao longo da apostila, construiremos as respostas para estas perguntas, cujo intuito principal é criar uma base sólida em programação na linguagem R aplicada à ciência de dados, a fim de dar maior autonomia ao leitor para que possa prosseguir em seus estudos.

Capítulo 2

Ciência de dados e R

2.1 O que é Ciência de Dados?

A ciência de dados, como o próprio termo sugere, consiste no estudo e análise de dados, com o objetivo de extrair informações relevantes, utilizando técnicas e conhecimentos multidisciplinares. Por mais que o termo tenha se popularizado fortemente nos últimos anos devido à massiva geração de dados em elevadas quantidades, diversidades e velocidades, sua concepção se origina no século passado, seja por nomes notáveis como o do matemático e estatístico [John W. Tukey](#), mas também por aqueles que atuavam nas áreas de negócios e de pesquisa que, sem a pretensão de nomear ou organizar uma nova área do conhecimento, poderiam ser considerados cientistas de dados.

A concepção recente de ciência de dados abrange pelo menos três grandes áreas do conhecimento, podendo ser descrita por um diagrama de Venn, idealizado em 2010, por [Drew Conway](#):



Figura 2.1: Diagrama de Venn da ciência de dados.

O diagrama é composto pelo conjunto de habilidades **computacionais**, conhecimento de **matemática e estatística** e domínio da **área de conhecimento**. Assim, as intersecções entre os conjuntos resultam em certas habilidades, descritas da seguinte maneira:

- **Aprendizado de máquinas:** do termo em inglês *machine learning*, consiste na intersecção entre as habilidades **computacionais** e de **matemática e estatística**. Utiliza estas bases para entender os modelos utilizados e detectar os padrões que serão replicados, a partir dos artifícios da programação, com o intuito de colocar em prática os algoritmos.
- **Pesquisa tradicional:** é a intersecção entre as áreas da **matemática e estatística** e **área de conhecimento**. Consiste na aplicação das bases matemáticas e estatísticas para solucionar problemas de uma área de atuação específica, sendo uma prática comum e tradicional no meio da pesquisa, principalmente acadêmica.
- **Zona de perigo:** a intersecção entre **habilidades computacionais** e **área de conhecimento** resulta em uma chamada *zona de perigo*, pois quem se encontra nesta situação consegue resolver problemas aplicando algoritmos, porém sem ter bases teóricas para compreender ou averiguar os resultados.
- **Ciência de dados:** a ciência de dados é o resultado da intersecção entre as três áreas - **habilidades computacionais**, **matemática e estatística** e **área de conhecimento**. Em teoria, um cientista de dados não possui total domínio destas três áreas, ou senão, possui especialização em alguma das três, contudo sabe aplicá-las para resolver problemas.

Tendo em vista as bases que definem um cientista de dados, entraremos no âmbito da programação, conhecendo um pouco mais sobre o software R.

2.2 R / RStudio

O R é uma das linguagens de programação mais utilizadas por cientistas de dados. Foi desenvolvido por Ross Ihaka e Robert Gentleman, na Universidade de Auckland, Nova Zelândia, em 1993. Iniciou como uma linguagem focada em programação estatística, mas que, ao longo do tempo, tornou-se cada vez mais encorpada e diversificada. Atualmente, o *R Development Core Team* atua na manutenção e no desenvolvimento da linguagem, sendo composto por diversos membros, dentre eles, seus idealizadores.

Por ser um software gratuito de código aberto (*Open source*), possibilitou a formação de uma comunidade que atua diretamente no desenvolvimento do programa, promovendo constantes facilidades, melhorias e inovações acessíveis ao público em geral. O compartilhamento de um conjunto de funções é dado através de **pacotes**, os quais devemos instalar para podermos utilizá-los. Detalharemos a instalação de pacotes na seção 2.4.

E, justamente, uma das principais contribuições idealizadas é o **RStudio**. O RStudio é uma IDE (*Integrated Development Environment*), ou seja, um ambiente de trabalho que executa o R a partir de uma interface gráfica mais agradável e com diversas funcionalidades (Figura 2.3), o que nos proporciona um maior conforto quando comparado ao R original, composto basicamente pelas janelas de script e console, como mostra a figura 2.2.

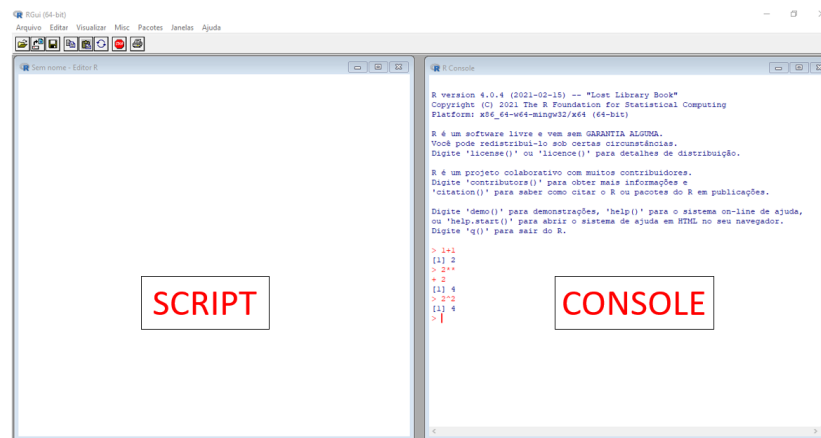


Figura 2.2: Tela do R original. Composto apenas pelo script e console.

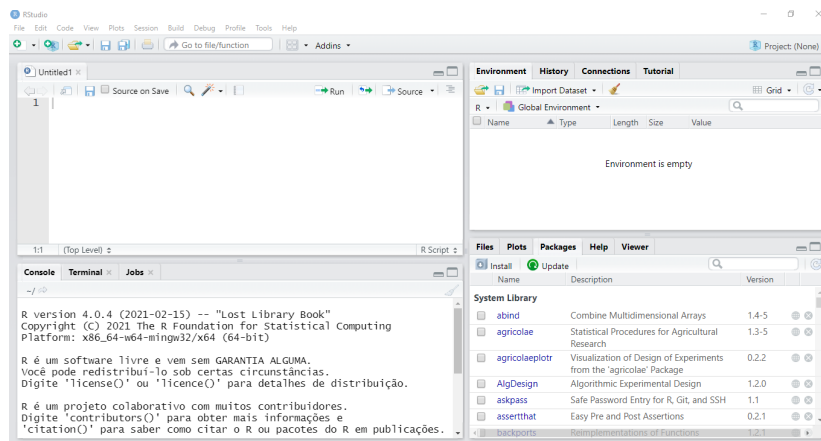


Figura 2.3: Tela do RStudio. Como podemos perceber, bem diferente do R original.

Mais adiante, na subseção 2.2.3, entraremos em mais detalhes sobre o ambiente do RStudio.

Vale salientar que o R pode ser utilizado sem o RStudio, porém o RStudio não funciona sem o R. No nosso caso, utilizaremos o RStudio para desenvolver nossas análises. Assim, precisamos ter instalados ambos os programas.

2.2.1 Instalando o R

O R está disponível para todos os sistemas operacionais. Sua instalação é feita via CRAN (*Comprehensive R Archive Network*), ou seja, uma rede com diversos servidores localizados em várias regiões do mundo, os quais armazenam versões idênticas e atualizadas de códigos e documentações para o R. Assim, para instalar o R, recomenda-se selecionar o servidor mais próximo à sua região. A seguir está o passo a passo para o *download*.

1. Acessar: <https://www.r-project.org/>;
2. No canto superior esquerdo, clicar em **CRAN**;

3. Selecionar o servidor (*mirror*) mais próximo a você (perceba que há um servidor da ESALQ/USP-Piracicaba);
4. Escolha o link referente ao seu sistema operacional;
5. Sistemas operacionais:
 - **Windows:** após clicar em ‘*Download R for Windows*’, selecione a opção ‘*base*’ e, posteriormente, ‘*Download R x.x.x for Windows*’, sendo ‘*x.x.x*’ a versão mais recente a ser baixada;
 - **Linux:** após clicar em ‘*Download R for Linux*’, selecione a distribuição que você utiliza e siga as instruções da página para instalar o R;
 - **MacOS:** após clicar em ‘*Download R for macOS*’, selecione a opção mais recente do R, a partir do link ‘*R-x.x.x.pkg*’, sendo ‘*x.x.x*’ a versão mais recente a ser baixada;
6. Feito o download, abra o arquivo baixado e siga as instruções para a instalação. Uma vez que utilizaremos o R a partir do RStudio, não há necessidade de criar um ícone de inicialização do R na área de trabalho, portanto, apenas instale o R em seu computador.

2.2.2 Instalando o RStudio

Uma vez feita a instalação do R, precisamos instalar o RStudio. Também está disponível para todos os sistemas operacionais e sua instalação pode ser feita a partir do link: <https://www.rstudio.com/products/rstudio/download/#download>. Escolha a versão referente ao seu sistema operacional e siga as instruções para baixar a IDE em seu computador.








| OS | Download | Size | SHA-256 |
|---------------------|--|-----------|----------|
| Windows 10 |  RStudio-2021.09.0-351.exe | 156.88 MB | f698d4a2 |
| macOS 10.14+ |  RStudio-2021.09.0-351.dmg | 196.28 MB | f8e97ced |
| Ubuntu 18/Debian 10 |  rstudio-2021.09.0-351-amd64.deb | 116.53 MB | 0d7ef262 |
| Fedora 19/Red Hat 7 |  rstudio-2021.09.0-351-x86_64.rpm | 133.82 MB | 3d858521 |
| Fedora 28/Red Hat 8 |  rstudio-2021.09.0-351-x86_64.rpm | 133.84 MB | 1043943b |
| Debian 9 |  rstudio-2021.09.0-351-amd64.deb | 116.79 MB | 309b7d7c |
| OpenSUSE 15 |  rstudio-2021.09.0-351-x86_64.rpm | 119.27 MB | 108dfae4 |

Figura 2.4: Na página referente ao link acima, vá até a seção ilustrada na figura. Lá encontraremos as versões disponíveis do RStudio, de acordo com o sistema operacional.

2.2.3 Ambiente RStudio

Agora que temos o R e o RStudio instalados, vamos conhecer mais sobre o ambiente do RStudio.

Janelas

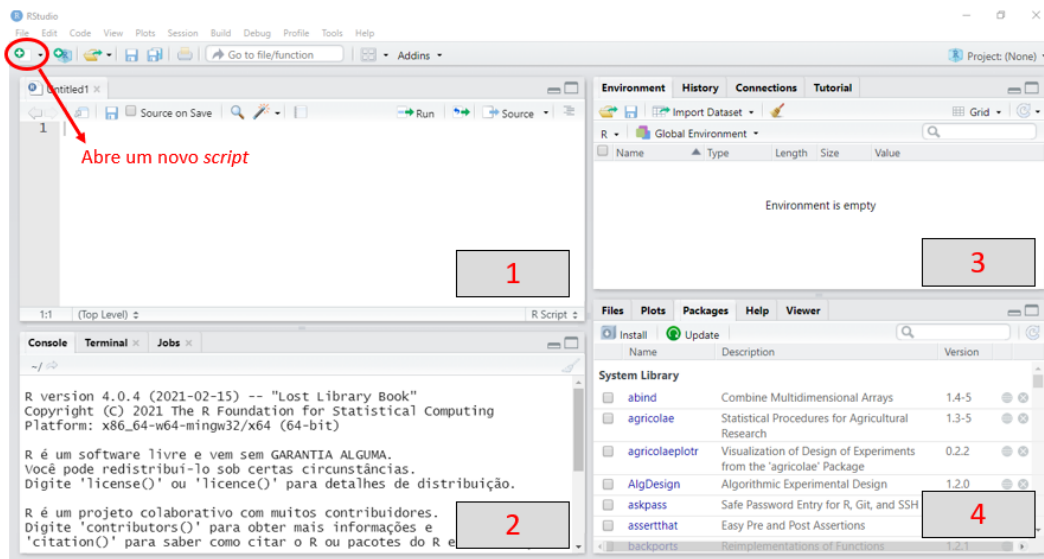


Figura 2.5: O RStudio apresenta 4 janelas principais, algumas com abas específicas, cada qual apresentando funcionalidades particulares.

A figura 2.5 ilustra as quatro janelas presentes no RStudio, cada qual com suas particularidades e funções.

1. *Script*: é a janela na qual escreveremos os códigos e comandos. Para abrir um novo *script*, clique no ícone logo abaixo da aba file, no canto esquerdo superior;
2. *Console*: é onde o código roda e apresenta as saídas dos códigos redigidos no *script*. Também podemos escrever comandos no *console*, porém, ao contrário do *script*, não há a possibilidade de edição, sendo necessário reescrevê-lo, caso preciso.
3. *Environment*: é onde se localiza e armazena os objetos criados. O ícone da vassoura (presente ao lado do ícone **Import Dataset**) exclui os objetos criados. Esta janela contém outras abas, porém a *Environment* é a principal dentre essas.
4. *File, Plots, Packages, Help e Viewer*: esta janela contém cinco abas.
 - *File*: apresenta os arquivos presentes no diretório do seu computador;
 - *Plots*: permite a visualização dos gráficos gerados;
 - *Packages*: mostra todos os pacotes instalados em seu RStudio;
 - *Help*: retorna documentações referentes a funções as quais podemos saber mais detalhes sobre elas;
 - *Viewer*: apresenta os resultados gerados a partir do R Markdown, Bookdown, dentre outras extensões relacionadas a execução de relatórios e documentos diversos.

Aparência

Podemos alterar a aparência do RStudio acessando a aba **Tools**, presente no menu superior, clicar em **Global Options...** e, posteriormente, na aba **Appearance**. Nela pode-se alterar o tema de fundo, regular o *zoom* do ambiente como um todo ou somente dos textos e alterar a fonte dos textos. Na figura 2.6 esta ilustrado um exemplo de configuração da aparência do RStudio, e na figura 2.7, o resultado dessa alteração.

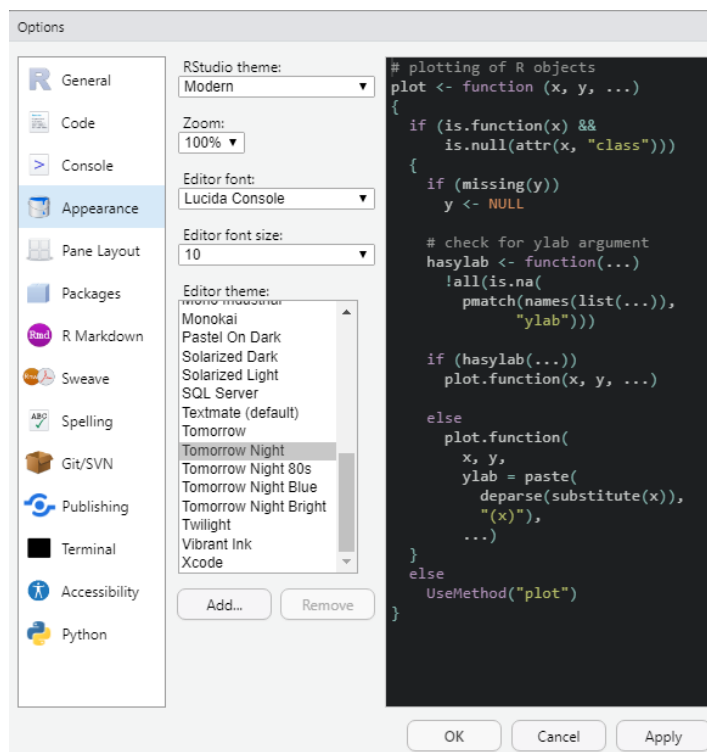


Figura 2.6: Podemos configurar a aparência do RStudio em diversos aspectos. Faça alguns testes e veja qual lhe agrada mais.

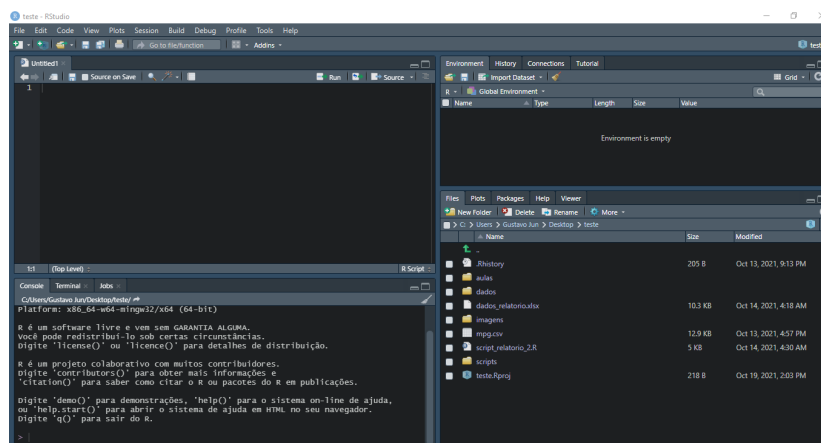


Figura 2.7: Um exemplo de alteração na aparência do RStudio.

2.3 Etapas da Ciência de Dados

Agora que temos uma melhor noção sobre ciência de dados e o software R, vamos explorar as etapas que compõem o seu processo.

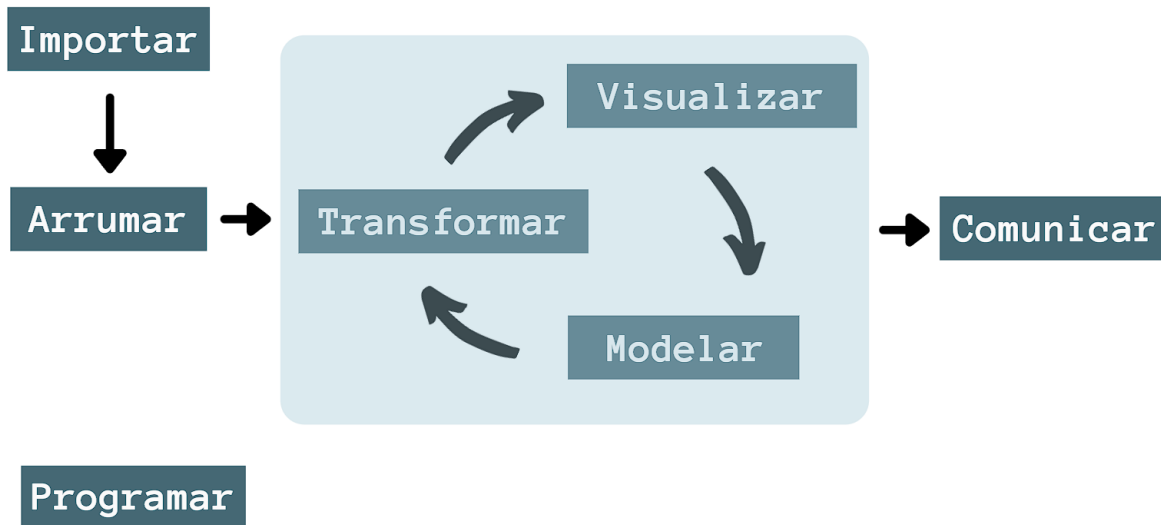


Figura 2.8: Etapas do trabalho em ciência de dados. Perceba que o ato de programar abrange todos os processos do fluxograma.

O fluxograma da figura 2.8 representa as etapas que compõem o trabalho de um cientista de dados. A seguir, descreveremos brevemente as etapas, para termos noção sobre a relevância de cada uma delas.

- **Importar** (*Import*): é a importação dos dados brutos para dentro do R, seja a partir de banco de dados presentes na web ou coletados pelo próprio cientista de dados. Basicamente é a etapa *sine qua non* da ciência de dados, pois sem dados, não há o que analisar;
- **Limpar/Arrumar** (*Tidy*): limpar ou arrumar os dados significa organizá-los em uma estrutura consistente, que esteja de acordo com a semântica de um conjunto de dados, para que não haja problemas ao realizar as análises. Mais adiante, veremos como estruturar os dados de maneira desejável, designando cada variável a uma coluna e cada observação a uma linha, semelhante a uma planilha Excel;
- **Transformar** (*Transform*): a transformação consiste em selecionar as observações de interesse no banco de dados. Em outras palavras, reduzir o banco de dados para conter somente as informações de interesse. Podemos também criar novas variáveis em função das variáveis já existentes originalmente, além de gerar descrições estatísticas como média, variância, porcentagens, dentre outras;
- **Visualizar** (*Visualisation*): a visualização gráfica dos dados permite enxergar as informações com mais clareza, levantar novos questionamentos e até mesmo indicar se a pesquisa está no caminho correto ou não;
- **Modelar** (*Models*): os modelos são usados para responder as perguntas norteadoras, depois que a pergunta norteadora estiver suficientemente precisa. Entra em cena a matemática, estatística e a computação como ferramentas para sua realização.

- **Comunicar** (*Communication*): é a parte crítica de um projeto analítico (*Data analysis*), pois é necessário expor os resultados de maneira inteligível para o público, seja ele técnico ou leigo;
- **Programar** (*Programming*): a programação circunda todas as etapas citadas anteriormente. Em ciência de dados, não precisamos ter um domínio avançado para começarmos um projeto, mas quanto mais se sabe, mais automático ficam as tarefas comuns e mais facilmente se resolve novos problemas.

Por último, podemos destacar o termo *Wrangling*, que abrange as etapas de **Arrumar** e **Transformar**. Traduzindo o termo, podemos entender que essas etapas do processo são, literalmente, uma *luta* para que se consiga deixar os dados de forma mais natural para serem analisados.

Na seção 2.4, vamos conhecer mais sobre o pacote `tidyverse`, o qual contém as principais funções a serem utilizadas ao longo desta apostila. Detalharemos os pacotes específicos para cada uma das etapas descritas anteriormente.

2.4 Pacote tidyverse

O `tidyverse` é um “pacote mestre” que abrange diversos outros, cada qual apresentando diversas funcionalidades específicas para cada uma das etapas apresentadas no fluxograma do tópico anterior. O esquema a seguir relaciona as etapas que constituem o trabalho do cientista de dados com os respectivos pacotes.

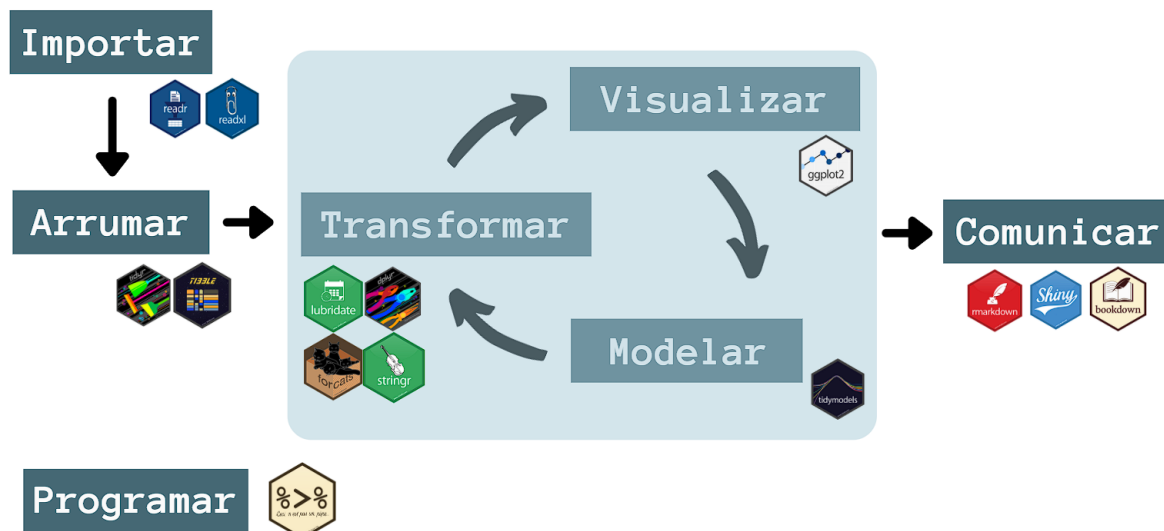


Figura 2.9: Para cada etapa do fluxograma de trabalho da ciência de dados, existem pacotes específicos no R.

Nesta apostila, focaremos no pacote `tidyverse` aplicado às etapas de **Importar**, **Arrumar**, **Transformar** e **Visualizar**, apresentando as principais ferramentas a serem utilizadas. Apenas os pacotes relacionados às etapas de **Modelar** e **Comunicar** não estão presentes no `tidyverse`.

Assim sendo, vamos instalar o nosso primeiro pacote, o `tidyverse`:

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

A função `install.packages("")` instala o requerido pacote. Atente-se ao fato que o nome do pacote deve estar entre aspas ("`nome_do_pacote`").

Uma vez instalado, devemos acionar o pacote com a função `library()`, para que possamos utilizar as suas funcionalidades. Agora, o nome do pacote não precisa estar entre aspas. Esta função deve ser executada a cada nova seção inicializada no R.

Lembrando que, para executar um comando, devemos escrever os respectivos códigos no *script* ou no *console*.

Para rodar estas funções (além das demais outras que rodaremos), devemos selecionar a linha de código que se deseja executar e clicar no ícone *Run*, presente no canto superior direito da própria janela do *script*, ou utilizar o atalho `ctrl + Enter` no teclado. Perceba que temos que rodar linha por linha de código ou selecionar todas as linhas do *script* para então rodar o código completamente.

Para se ter uma visão geral de quais pacotes estão presentes no `tidyverse`, utilizaremos a função `tidyverse_packages()`.

```
tidyverse_packages()
```

```
[1] "broom"          "cli"            "crayon"         "dbplyr"
[5] "dplyr"          "dtplyr"         "forcats"        "googledrive"
[9] "googlesheets4" "ggplot2"        "haven"          "hms"
[13] "httr"           "jsonlite"       "lubridate"      "magrittr"
[17] "modelr"         "pillar"         "purrr"          "readr"
[21] "readxl"         "reprex"         "rlang"          "rstudioapi"
[25] "rvest"          "stringr"        "tibble"         "tidyr"
[29] "xml2"           "tidyverse"
```

Perceba que o pacote `tidyverse` contém outros 30 pacotes. Dentre estes, utilizaremos o `readr` e o `readxl` para importarmos os dados; o `tidyr` e o `tibble` para arrumar; o `dplyr`, `stringr`, `forcats` e `lubridate` para transformar; e por último, o `ggplot2` para visualizar.

Caso o leitor tenha curiosidade em saber mais detalhes sobre o `tidyverse`, acesse o link da página oficial do pacote: <https://www.tidyverse.org/packages/>.

Nos capítulos a seguir, abordaremos as etapas de **Importar**, **Arrumar**, **Transformar** e **Visualizar**, apresentando as principais utilidades e funções de cada um dos respectivos pacotes presentes no `tidyverse`. Mas antes, veremos alguns conceitos básicos para programarmos em R.

Capítulo 3

Noções básicas em R

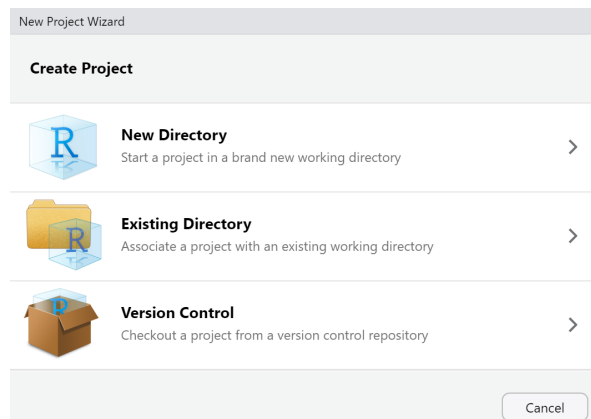
Para trabalhar com ciência de dados em R, devemos ter algumas noções básicas de programação nessa linguagem. Os conceitos discutidos neste capítulo serão a base para aplicarmos as demais ferramentas ao longo da apostila.

3.1 Projetos

Uma funcionalidade importante do RStudio são os projetos. Ao criar um projeto, uma nova pasta é criada em seu computador. Nela, podemos (e devemos) direcionar os arquivos a serem utilizados para o projeto, além de abrigar os novos arquivos criados para a análise.

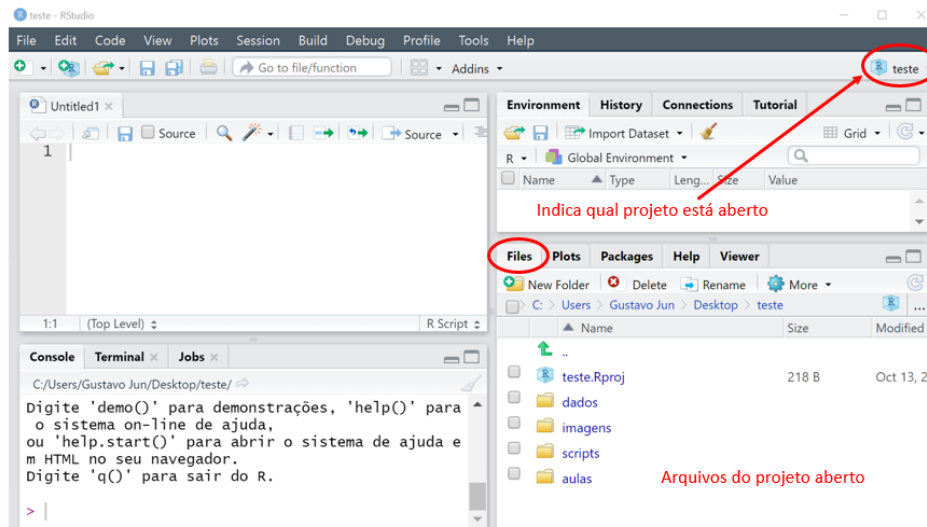
Com isso, a criação de projetos nos proporciona uma melhor organização dos arquivos, separando-os de acordo com o projeto realizado no R. Além disso, facilita a importação de dados para dentro do R, como veremos no capítulo 4, referente à importação de dados.

Para criar um projeto, clique em **File**, presente no menu superior, depois em **New Project...** Então, abrirá uma janela como o da imagem a seguir:



Clique em **New Directory**, **New Project** e nomeie seu projeto em **Directory name**. Em **Create project** as **subdirectory of**: escolha o diretório em seu computador (pasta) no qual seu novo projeto será alocado. Finalize clicando em **Create Project**.

Pronto, seu projeto está criado. No canto superior direito aparecerá o nome do projeto. Além disso, na aba **Files**, estarão todos os arquivos contidos na pasta referente ao projeto. Portanto, direcione todos os arquivos que você utilizará para a respectiva pasta do projeto em uso.



Clicando no mesmo ícone o qual aparece o nome do projeto, podemos criar um novo projeto e abrir um projeto já existente. Por tanto, atente-se a qual projeto se encontra ativo.

3.2 Ajuda

Cada ferramenta presente no R contém uma documentação que explica a sua utilização. Para acessarmos tais documentos, podemos prosseguir das seguintes maneiras:

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

Neste exemplo, queremos saber mais sobre a função **mean**, ou seja, função que calcula a média aritmética. Ao rodar um destes comandos, a documentação referente à função será aberta na aba **Help**. Nela conterá algumas descrições importantes, como a noção geral de uso da função, os argumentos aceitos e exemplos de utilização.

Portanto, caso tenha dúvidas sobre qualquer outra ferramenta - seja funções, *data frames*, listas ou pacotes -, utilize o `help(nome_da_ferramenta)` ou o `?nome_da_ferramenta`. Ainda, pode-se utilizar o atalho F1 do teclado, selecionando uma ferramenta presente no *script* e clicar em F1.

Outra via de auxílio são as *folhas de cola*, ou **Cheatsheets**. Basicamente, trazem resumos sobre as principais funções contidas em determinados pacotes. Seu acesso pode ser realizado em: <https://www.rstudio.com/resources/cheatsheets/> ou clicando na página inicial da aba **Help**.

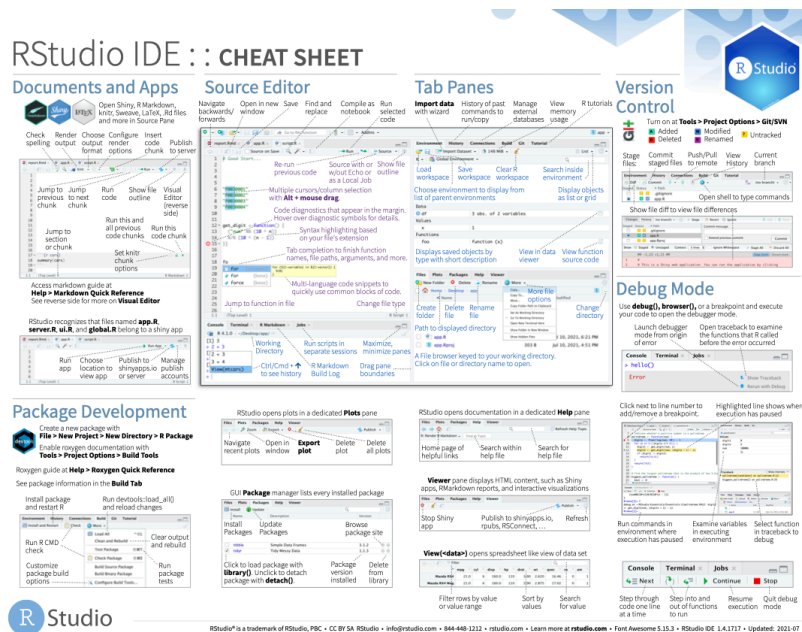


Figura 3.1: Cheatsheet do RStudio. Nela podemos verificar, de maneira geral, as principais funcionalidades presentes no ambiente do RStudio.

Caso ainda tenha dúvidas, não existe em fazer uma busca no Google, encontrar tutoriais explicativos - seja no YouTube ou no próprio site do [RStudio](https://www.rstudio.com/) -, acessar fóruns de perguntas e respostas - como o [Stack Overflow](https://stackoverflow.com/) e acessar o [Rseek](https://rseek.info/) ou o [Search R-project](https://search.r-project.org/), que são buscadores específicos para assuntos relacionados ao R.

3.3 Comentários

Podemos inserir comentários dentro do *script*, sem que estes interfiram na execução dos códigos. Fazer comentários ao longo do *script* é muito importante para quem está começando, pois assim permite fazer registros para, posteriormente, revisar a utilidade de certas funcionalidades ou realizar alguma manutenção no código. Além disso é muito relevante para que se possa compartilhar um código inteligível com outras pessoas.

Para inserir um comentário, basta colocar o símbolo `#` antes da parte comentada.


```
# Potência (possibilidade 1)
2 ^ 3
```

```
[1] 8
```

```
# Potência (possibilidade 2)
2 ** 3
```

```
[1] 8
```

```
# Raiz quadrada
4 ^ (1/2)
```

```
[1] 2
```

```
# Resto da divisão
7 %% 3
```

```
[1] 1
```

```
# Parte inteira de uma divisão
7 %/% 3
```

```
[1] 2
```

```
# Ordem de precedência
1 + 2 * 5 - (4 - 2) / 2
```

```
[1] 10
```

No caso do exemplo da ordem de precedência, assim como na matemática, o R calcula primeiro a multiplicação e divisão, além dos valores entre parênteses, para, posteriormente, calcular a adição e a subtração.

Outra informação relevante a ser dita quando tratamos de números no R é que os decimais são delimitados por pontos e não por vírgulas, portanto, diferente do padrão adotado no Brasil. Essa informação é importante para que possamos escrever números decimais da maneira a qual o R aceita.

Uma função útil para tratar de números decimais é a `round()`. Ela arredonda números decimais de acordo com o número de casas decimais informadas no argumento `digits`.

```
round(10.456783452, digits = 3)
```

```
[1] 10.457
```

```
round(pi, digits = 2)
```

```
[1] 3.14
```

3.5 Objetos

Objetos são nomes que recebem um determinado valor. Para criar um objeto, utilizamos o operador `<-`, cujo atalho no teclado é `Alt + -` (tecla `Alt`, junto com o sinal de menos).

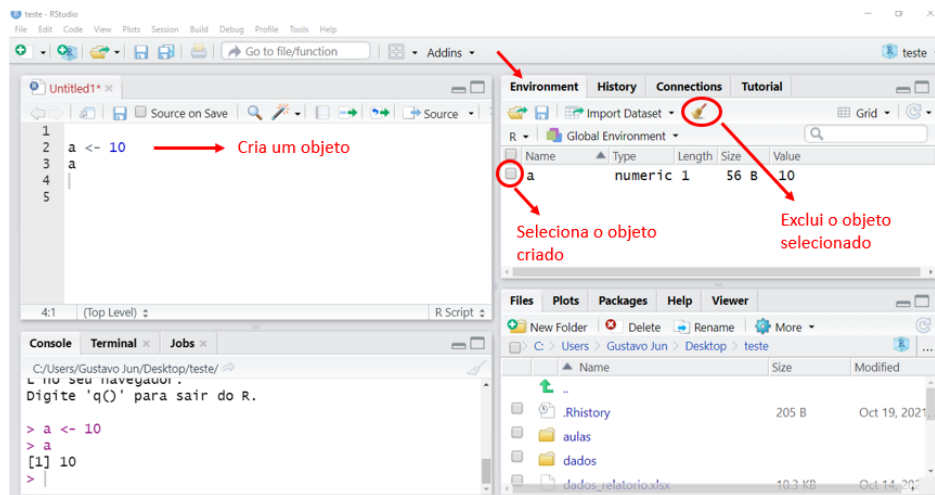
No exemplo a seguir, salvaremos o valor 10 dentro do nome `a`. Ao rodar o objeto `a`, o R retorna o valor 10.

```
# O número '10' será armazenado em 'a'
a <- 10

# Rodando o objeto 'a', retorna o valor '10'
a
```

```
[1] 10
```

Perceba que ao criar um objeto, esse será armazenado na janela **Environment**. Para excluir objetos, selecione os que deseja excluir e clique no ícone da vassoura.



Devemos nos atentar a alguns outros detalhes ao criarmos um objeto. Primeiramente, o R diferencia letras maiúsculas e minúsculas:

```
A <- 10
a <- 50

A
```

```
[1] 10
```

```
a
```

```
[1] 50
```

Além disso, não podemos nomear um objeto começando por números, *underline* (), ponto (.) e traço (-), sendo esses, **nomes de sintaxe inválida**.

```
# Nomes não permitidos!
13v <- 1
_objeto <- 2
-objeto <- 3
nomear-objeto <- 4
.objeto <- 5
```

Contudo, podemos utilizar números, *underline* e pontos, desde que não estejam no início do nome.

```
# Permitido
x1 <- 7
nomear_objeto <- 25
nomear.objeto <- 52
```

Neste primeiro momento, criamos objetos que recebem um único valor. Ao longo da apostila, criaremos objetos mais complexos, cada qual apresentando tipos diferentes, como os vetores, *data frames* e listas.

3.6 Funções

As funções são nomes que guardam um código em R. Portanto, cada função apresenta certas ferramentas específicas que nos trazem alguma resposta.

Dentro dos parênteses de uma função estão os **argumentos**. Estes argumentos são separados por vírgulas e não há um limite de argumentos que uma função pode receber. Por tanto, uma função executa determinado comando, em resposta aos argumentos especificados dentro dela.

```
sum(1, 2, 10)
```

```
[1] 13
```

No exemplo, 1, 2 e 10 são argumentos da função `sum` (ou seja, função soma). Portanto, a função realizou a operação de soma dos argumentos especificados dentro da função, retornando o resultado da operação, igual a 13.

Alguns argumentos de funções possuem nomes, que podemos ou não explicitar em uma função. Utilizaremos como exemplo a função `seq()`.

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

```
[1] 2 4 6 8 10
```

A função `seq()` cria uma sequência numérica de acordo com os argumentos. O `from` indica por qual número se inicia a sequência, o `to`, em qual número termina e o `by`, de quanto em quanto a sequência será construída. Assim, no exemplo, criamos uma sequência que começa do 2, termina no 10 e que vai de 2 em 2.

Temos a possibilidade de não explicitar os nomes dos argumentos, desde que se respeite a ordem em que os argumentos aparecem.

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

```
[1] 2 4 6 8 10
```

Para saber qual a ordem dos argumentos da função `seq()`, acessamos a sua documentação com o comando `?seq`.

Caso seja explicitado o nome dos argumentos, a ordem não interfere no resultado final.

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

```
[1] 2 4 6 8 10
```

Mas caso os nomes não sejam explicitados, a ordem incorreta acarreta em outro resultado.

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

```
[1] 2
```

Além de conferir as documentações referentes às funções, podemos utilizar a função `args(nome_da_função)` para verificar todos os argumentos presentes em uma função específica.

```
args(lm)
```

```
function (formula, data, subset, weights, na.action, method = "qr",  
  model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,  
  contrasts = NULL, offset, ...)  
NULL
```

As funções serão a base para realizarmos cada etapa do fluxograma da ciência de dados, assim, nos depararemos com diversas funções e argumentos específicos, cada qual presente em um pacote ou sendo nativo do R.

3.7 Classes

As classes de objetos nos indicam qual o **tipo** de valor que está armazenado em um determinado objeto. São divididas em quatro principais tipos:

- *numeric*: apresenta valores numéricos, sejam inteiros (*integer*) ou decimais (*double*);
- *character*: valores do tipo caractere. Também podemos chamá-los de valores do tipo texto, categóricos ou *string*, nome mais comum no meio da programação;
- *factor*: apresentam variáveis qualitativas possíveis de serem agrupadas em categorias. Veremos com mais detalhes na seção 3.10;
- *logical*: valores lógicos do tipo verdadeiro ou falso (TRUE/FALSE). Também são conhecidos como valores booleanos. Abordaremos mais detalhadamente esta classe na seção 3.11.

Para verificarmos a classe de um objeto, utilizamos a função `class()`, tendo como argumento o nome do objeto. Vejamos alguns exemplos:

```
# Classe numérica  
a <- 10  
class(a)
```

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

```
# Classe caractere  
b <- "a"  
class(b)
```

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

Para criarmos um objeto com a classe do tipo caractere, devemos escrevê-lo entre aspas `"`. As aspas servem para diferenciar **nomes** (objetos, funções e pacotes) de **textos** (letras e palavras). No exemplo anterior, perceba que na classe numérica criamos um **objeto** de **nome** `a` que recebe o valor 10, enquanto que na classe caractere, criamos um objeto de nome `b` que recebe o **texto** `"a"`.

Portanto, resumindo: no primeiro caso, criamos um objeto chamado `a`, enquanto no outro exemplo, criamos um objeto que contém o caractere `"a"`.

Conhecer a classe de objetos e valores é importante para definirmos os procedimentos e operações possíveis de serem realizadas. Por exemplo, podemos realizar uma operação matemática com números, porém não podemos com caracteres.

```
# Números  
10^2
```

```
[1] 100
```

```
# Caracteres  
"a" + "b"
```

```
Error in "a" + "b": argumento não-numérico para operador binário
```

```
"1" + "1"
```

```
Error in "1" + "1": argumento não-numérico para operador binário
```

No último exemplo, perceba que os números 1 foram escritos entre aspas, logo deixam de ser um tipo numérico para se apresentar como um tipo caractere. Por este motivo não conseguimos realizar a soma desses.

3.8 Data frames

O *data frame* é o objeto que armazena os dados importados para dentro do R. São estruturados a partir de linhas e colunas, sendo que cada coluna representa uma variável e cada linha, uma observação, estrutura muito semelhante a uma planilha Excel.

```
PlantGrowth
```

```

      weight group
1      4.17  ctrl
2      5.58  ctrl
3      5.18  ctrl
4      6.11  ctrl
5      4.50  ctrl
6      4.61  ctrl
7      5.17  ctrl
8      4.53  ctrl
9      5.33  ctrl
10     5.14  ctrl
11     4.81 trt1
12     4.17 trt1
13     4.41 trt1
14     3.59 trt1
15     5.87 trt1
16     3.83 trt1
17     6.03 trt1
18     4.89 trt1
19     4.32 trt1
20     4.69 trt1
21     6.31 trt2
22     5.12 trt2
23     5.54 trt2
24     5.50 trt2
25     5.37 trt2
26     5.29 trt2
27     4.92 trt2
28     6.15 trt2
29     5.80 trt2
30     5.26 trt2

```

O *data frame* `PlantGrowth` é nativo do R e contém dados sobre o crescimento de plantas sob 2 tipos diferentes de tratamentos (para mais informações, consulte a documentação do *data frame*, rodando `?PlantGrowth`). Possui 30 linhas (observações) e 2 colunas (variáveis).

Portanto, podemos dizer que o objeto de nome `PlantGrowth` guarda um *data frame* com 30 linhas e 2 colunas.

Podemos aplicar algumas funções em *data frames*, a fim de visualizarmos melhor sua estrutura e elementos presentes. A seguir, demonstraremos algumas delas aplicadas no *data frame* `PlantGrowth`.

```

# Mostra as 6 primeira linhas
head(PlantGrowth)

```

```

      weight group
1      4.17  ctrl
2      5.58  ctrl
3      5.18  ctrl

```

```
4  6.11  ctrl
5  4.50  ctrl
6  4.61  ctrl
```

```
# Mostra as 6 últimas linhas
tail(PlantGrowth)
```

```
      weight group
25    5.37  trt2
26    5.29  trt2
27    4.92  trt2
28    6.15  trt2
29    5.80  trt2
30    5.26  trt2
```

```
# Mostra as dimensões (nº de linhas x nº de coluna)
dim(PlantGrowth)
```

```
[1] 30  2
```

```
# Nomes das colunas (variáveis)
names(PlantGrowth)
```

```
[1] "weight" "group"
```

```
# Estrutura do data frame com informações como o tipo, dimensão e classes
str(PlantGrowth)
```

```
'data.frame':  30 obs. of  2 variables:
 $ weight: num  4.17 5.58 5.18 6.11 4.5 4.61 5.17 4.53 5.33 5.14 ...
 $ group : Factor w/ 3 levels "ctrl","trt1",...: 1 1 1 1 1 1 1 1 1 1 ...
```

```
# Retorna algumas medidas-resumo
summary(PlantGrowth)
```

```
      weight      group
Min.   :3.590  ctrl:10
1st Qu.:4.550  trt1:10
Median :5.155  trt2:10
Mean    :5.073
3rd Qu.:5.530
Max.    :6.310
```

Os *data frames* serão o nosso principal objeto de estudo para aplicarmos ciência de dados, uma vez que guardam os dados a serem analisados. Estudaremos os *data frames* com mais detalhes na subseção 3.13.1. Mas antes, devemos conhecer alguns outros conceitos importantes, como é o caso dos **vetores**, assunto do tópico a seguir.

3.9 Vetores

Os vetores nada mais são do que um conjunto de valores unidos em um só objeto. Em um linguajar mais técnicos, vetor é um conjunto de valores indexados. Para criarmos um vetor, devemos utilizar a função `c()`, cujos argumentos devem estar separados por vírgulas.

```
vetor_numerico <- c(2, 6, -10, 14, 18, 22)
vetor_numerico
```

```
[1]  2  6 -10 14 18 22
```

```
class(vetor_numerico)
```

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

```
vetor_texto <- c("g", "j", "y")
vetor_texto
```

```
[1] "g" "j" "y"
```

```
class(vetor_texto)
```

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

Perceba que a função `class()` nos retorna o tipo de classe que um vetor apresenta. Um vetor só pode guardar um tipo de classe. Caso misturemos um vetor com números e caracteres, os números serão convertidos para texto. Esse comportamento é conhecido como **coerção**.

```
# Vetor misto
vetor_misto <- c(1, 5, "a")
vetor_misto
```

```
[1] "1" "5" "a"
```

```
class(vetor_misto)
```

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

Os números 1 e 5, dentro de um vetor que contém o caractere "a", são convertidos para texto, resultando em um vetor com os textos "1", "5" e "a", como constatado ao utilizar a função `class()`, que nos retorna uma classe do tipo caractere.

Portanto, devemos ter em mente que, para vetores com valores de classes diferentes, os caracteres serão dominantes em relação aos números. Para criar um conjunto de valores com classes diferentes, devemos criar uma **lista**, assunto que veremos mais adiante na seção 3.13.

Podemos criar um vetor com uma sequência numérica utilizando o operador `:`.

```
# Vetor de 1 a 15
sequencia_numerica <- 1:15
sequencia_numerica
```

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

Ao criarmos um vetor, cada valor ocupa uma posição dentro do vetor. A posição é dada pela ordem em que estão no vetor. Portanto, podemos encontrar determinados valores de acordo com a posição em que estão localizados no vetor. Essa operação é conhecida como *subsetting*. Para isso, colocamos o número da posição que desejamos acessar dentro de colchetes [], associado ao objeto que desejamos analisar.

```
posicao_vetor <- c(11, 22, 33, 44)
posicao_vetor[1]
```

```
[1] 11
```

```
posicao_vetor[2]
```

```
[1] 22
```

```
posicao_vetor[3]
```

```
[1] 33
```

```
posicao_vetor[4]
```

```
[1] 44
```

```
posicao_vetor[5]
```

```
[1] NA
```

O objeto de nome `posicao_vetor` é um vetor com 4 valores (ou 4 argumentos). Com o comando `posicao_vetor[1]`, temos o valor 11, contido na primeira posição do vetor, seguindo a mesma lógica para as demais posições. Perceba que o comando `posicao_vetor[5]` nos retorna o valor NA, pois não existe esta posição dentro do vetor. Mais adiante, na seção 3.12, trataremos sobre o valor NA.

Também podemos inserir um conjunto de posições dentro dos colchetes, o que nos retorna um subconjunto de valores dentro de um vetor. Para isso, utilizamos a função `c()`, tendo como argumentos as posições que se deseja acessar.

```
v <- c("w", "x", "y", "z")
v[c(2, 4)]
```

```
[1] "x" "z"
```

Ainda, podemos realizar operações matemáticas com vetores de classe numérica:

```
vetor <- c(8, 19, 24, 25)
```

```
vetor + 1
```

```
[1]  9 20 25 26
```

```
vetor - 1
```

```
[1]  7 18 23 24
```

```
vetor / 2
```

```
[1]  4.0  9.5 12.0 12.5
```

```
vetor * 2
```

```
[1] 16 38 48 50
```

```
vetor ^ 2
```

```
[1]  64 361 576 625
```

Perceba que as operações matemáticas são executadas para cada um dos elementos do vetor.

Também podemos fazer operações entre vetores:

```
vetor1 <- c(1, 2, 3, 4)
```

```
vetor2 <- c(5, 6, 7, 8)
```

```
vetor1 + vetor2
```

```
[1]  6  8 10 12
```

Para realizar a operação, ambos os vetores são alinhados, sendo somados os valores de acordo com a posição correlata entre os elementos dos vetores. Portanto, o elemento que ocupa a primeira posição no **vetor1** é somado com o primeiro elemento do **vetor2**, seguindo a mesma lógica para os demais elementos.

No caso de vetores com tamanhos diferentes, ocorre o processo de **reciclagem**.

```
vetor3 <- c(1, 3)
```

```
vetor4 <- c(11, 22, 33, 44)
```

```
vetor3 + vetor4
```

```
[1] 12 25 34 47
```

A operação entre vetores de tamanhos diferentes segue a mesma lógica citada anteriormente: ambos os vetores são alinhados, porém, por apresentarem diferentes dimensões, é realizada uma repetição (*reciclagem*) do **vetor3** para que esse fique com o mesmo tamanho do **vetor4**, assim, possibilitando a operação matemática. Portanto, é como se o **vetor3** tivesse a dimensão de `c(1, 3, 1, 3)`.

Vale destacar que o comportamento de reciclagem foi aplicado quando fizemos as operações matemáticas em um só vetor. Por exemplo, quando somamos 1 ao vetor de dimensão `c(8, 19, 24, 25)`, o R reciclou o número 1 - que nada mais é do que um vetor de tamanho 1, igual a `c(1)` - formando um vetor `c(1, 1, 1, 1)` para que fosse possível realizar a soma.

Até então, fizemos operações entre vetores com comprimentos múltiplos entre si. Ao realizar operações entre vetores cujos tamanhos não são múltiplos, a reciclagem atua da seguinte maneira:

```
vetor5 <- c(1, 2, 3)
vetor6 <- c(10, 20, 30, 40, 50)

vetor5 + vetor6
```

```
Warning in vetor5 + vetor6: comprimento do objeto maior não é múltiplo do
comprimento do objeto menor
```

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

Nessa situação, foi realizada a reciclagem do **vetor5**, até que ele adquirisse a mesma dimensão do **vetor6**. Assim, o **vetor5** se apresenta da seguinte maneira após a reciclagem: `c(1, 2, 3, 1, 2)`. Perceba que o último valor do **vetor5** não foi reciclado na operação, pois sua presença na reciclagem ultrapassaria a dimensão do **vetor6**. Normalmente, esse tipo de operação não é desejada, devido a não reciclagem de certos valores de um vetor, o que pode causar problemas nas análises. E, justamente, por ser um processo incomum, o R gera uma mensagem de aviso (*warning*) no *console*, alertando o ocorrido.

Guarde com carinho os conceitos explicados nesta seção, pois os utilizaremos com muita frequência nos próximos capítulos para trabalharmos com os *data frame*, uma vez que cada coluna de um *data frame* é um vetor.

3.10 Fatores

As variáveis do tipo **fator** são um caso especial de classe de objetos que representam variáveis qualitativas possíveis de serem agrupadas em **categorias**, como, por exemplo, o sexo e grau de escolaridade. As possíveis categorias presentes em um fator são indicadas pelo atributo **levels**, como os **levels** masculino e feminino, no caso do sexo, e ensino fundamental, médio e superior, no caso do grau de escolaridade.

Normalmente, este tipo de variável é criada ou importada como texto, sendo necessário transformá-la em fator, utilizando a função `as.factor()`.

```
# Classe do tipo caractere
sexo <- c("F", "F", "M", "F", "M", "M")
class(sexo)
```

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

```
# Classe do tipo fator
as.factor(sexo)
```

```
[1] F F M F M M
Levels: F M
```

Criando o objeto `sexo`, sendo os argumentos `F` para o sexo feminino e `M` para o masculino, temos um vetor de classe do tipo `caractere`. Como o `sexo` é uma variável possível de ser categorizada, transformamos essa variável para a classe `fator`, a partir da função `as.factor()`. Perceba que na classe do tipo `fator`, o R nos retorna os `levels`, ou seja, o conjunto de categorias presentes no objeto `sexo`, no caso, `F` e `M`.

Ainda, podemos criar um vetor do tipo `fator` utilizando a função `factor()`. A função `class()` mostra o tipo de classe do objeto `sexo_fator` e a `levels()`, indica quais são as categorias presentes no objeto de classe do tipo `fator`.

```
sexo_fator <- factor(c("F", "F", "M", "F", "M", "M"))
class(sexo_fator)
```

```
[1] "factor"
```

```
levels(sexo_fator)
```

```
[1] "F" "M"
```

Por padrões, os `levels` são ordenados por ordem alfabética. No exemplo anterior, a categoria `F` vem antes da `M`. Para reordená-las, utilizamos o argumento `levels` na função `factor()`, ordenando as categorias de acordo com sua posição no vetor.

```
sexo_fator <- factor(sexo_fator, levels = c("M", "F"))
levels(sexo_fator)
```

```
[1] "M" "F"
```

3.10.1 Diferenças entre fatores e caracteres

Apesar dos objetos do tipo `fator` serem representados por letras ou palavras, o R os enxerga como números inteiros, diferentemente dos objetos da classe `caractere`, que são puramente textos. Podemos notar essas diferenças ao tentar convertê-las em classe numérica.

```
# Classe caractere
sexo <- c("F", "F", "M", "F", "M", "M")
class(sexo)
```

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



```
as.numeric(sexo)
```

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

```
# Classe fator
sexo_fator <- factor(c("F", "F", "M", "F", "M", "M"))
class(sexo_fator)
```

```
[1] "factor"
```

```
as.numeric(sexo_fator)
```

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

Podemos notar que não foi possível converter o vetor do tipo caractere para um vetor numérico, pois o R não consegue atribuir uma classificação numérica para textos. Porém, no caso do vetor tipo fator, foi possível transformá-lo para um tipo numérico, sendo representado como 1 o nível F e como 2, o M.

Portanto, para o R, os **levels** dos fatores são números inteiros sequenciais, começando do 1, atribuídos conforme a ordem alfabética dos argumentos no vetor.

3.11 Operações lógicas

As operações lógicas nos retornam valores do tipo verdadeiro ou falso, representados no R por **TRUE** e **FALSE** (em letras maiúsculas), respectivamente. Portanto, a classe atribuída a estes tipos de valores é a **logical** - como vimos na seção 3.7 - aceitando somente estes dois valores.

```
class(TRUE)
```

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

```
class(FALSE)
```

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

Para aplicarmos testes lógicos, podemos utilizar o operador **==** (duas vezes o sinal de igual) para verificar se dois valores são iguais, ou o operador **!=** (exclamação + igual) para ver se os valores são diferentes.

```
# Resultados verdadeiros
52 == 52
```

```
[1] TRUE
```

```
"x" == "x"
```

```
[1] TRUE
```

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

```
[1] TRUE
```

```
1 != 2
```

```
[1] TRUE
```

```
# Resultados falsos  
52 != 52
```

```
[1] FALSE
```

```
"x" != "x"
```

```
[1] FALSE
```

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

```
[1] FALSE
```

```
1 == 2
```

```
[1] FALSE
```

Para verificar se um valor é maior que o outro, podemos utilizar 4 operadores:

- < se um valor é **menor** ao outro;
- > se um valor é **maior** ao outro;
- <= se um valor é **menor ou igual** ao outro;
- >= se um valor é **maior ou igual** ao outro.

```
# Menor  
3 < 5
```

```
[1] TRUE
```

```
3 < 2
```

```
[1] FALSE
```

```
# Maior  
3 > 1
```

```
[1] TRUE
```

```
4 > 7
```

```
[1] FALSE
```

```
# Menor ou igual  
3 <= 3
```

```
[1] TRUE
```

```
2 <= 1
```

```
[1] FALSE
```

```
# Maior ou igual  
10 >= 5
```

```
[1] TRUE
```

```
1 >= 6
```

```
[1] FALSE
```

O operador `%in%` verifica se um dado valor pertence a um vetor, ou seja, se um dado valor está contido dentro de um conjunto de valores.

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

```
[1] TRUE
```

```
"a" %in% c("x", "y")
```

```
[1] FALSE
```

Ainda tratando dos vetores, observe o seguinte exemplo:

```
vet <- c(1, 0, 6, -9, 10, 52, 3)
```

```
vet > 3
```

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

```
vet[vet > 3]
```

```
[1] 6 10 52
```

Aqui teremos que relembrar alguns conceitos expostos anteriormente. Primeiramente, criamos um vetor de nome `vet`, que recebe 7 valores. Posteriormente, utilizamos um teste lógico para verificar quais valores de `vet` são maiores que 3, cuja resposta é dada por `TRUE` ou `FALSE`. Nessa situação, ocorre uma **reciclagem** (conceito visto na seção 3.9) do valor 3, portanto, resultando em um vetor igual a `c(3, 3, 3, 3, 3, 3, 3)`. A partir disso, o R alinha o vetor `c(1, 0, 6, -9, 10, 52, 3)` com o vetor `c(3, 3, 3, 3, 3, 3, 3)` e testa a lógica proposta elemento por elemento (`1 > 3`, `0 > 3`, `6 > 3`, `-9 > 3`, `10 > 3`, `52 > 3` e `3 > 3`), formando um vetor de verdadeiros e falsos.

Toda essa explicação embasa a operação lógica `vet[vet > 3]`, que nos retorna apenas os valores do vetor que são maiores que 3, ou seja, todos os valores iguais a `TRUE`. Esse tipo de operação será muito utilizada nos capítulos seguintes, mais especificamente no capítulo 6 ao utilizarmos a função `filter`, essa muito mais simples de operar do que em relação ao apresentado anteriormente, porém seguindo a mesma lógica.

3.12 Valores especiais

Nesta seção, explicaremos alguns valores particulares presentes no R.

3.12.1 NA

O NA representa a **ausência de informação**, ou seja, a informação existe, porém não se sabe qual é. Em bases de dados, é comum que algumas informações não tenham registro, sendo assim, representados pelo valor NA. Com isso, devemos saber interpretar e tratar estes valores.

Reforçando o significado de NA, podemos exemplificar com uma coleta de dados em que certos indivíduos não informaram suas idades. Isso não significa que os entrevistados não possuem idade, mas simplesmente que o registro de algumas idades é uma informação ausente na pesquisa.

Sabendo do significado de NA, podemos aprofundar sua utilização em testes lógicos, como representa o exemplo a seguir:

```
idade_gustavo <- 24
idade_joao <- NA
idade_cristian <- NA

idade_gustavo == idade_joao
```

```
[1] NA
```

```
idade_joao == idade_cristian
```

```
[1] NA
```

Conhecemos apenas a idade do Gustavo, mas não sabemos a idade do João e do Cristian, logo, a idade dos dois últimos são representados pelo valor **NA**. Ao aplicar o teste lógico `idade_gustavo == idade_joao`, o retorno é o valor **NA**, ou seja, o R não sabe responder se a idade do Gustavo é a mesma de João, uma vez que a idade do João não foi informada. No teste `idade_joao == idade_cristian`, também nos é retornado o valor **NA**, pois ambas as idades não foram informadas, logo tanto a idade do João pode ser a mesma do Cristian, como pode ser distinta. Portanto, o R não “chutará” uma resposta e simplesmente responderá: **NA** (*não sei*).

Dito isso, temos que nos atentar aos valores **NA** presentes nos nossos *data frames*, pois em algumas operações, como a média (`mean()`), não conseguimos executá-la na presença desses valores. No caso da função `mean()`, ela possui o argumento `na.rm = TRUE` para excluir os **NA** da operação matemática. Todavia, algumas outras funções não possuem um argumento semelhante, sendo necessário realizar outras manipulações de dados, as quais veremos nos capítulos mais adiantes.

3.12.2 NaN

O NaN (*not a number*) representa indeterminações matemáticas.

```
0/0
```

```
[1] NaN
```

```
log(-1)
```

```
[1] NaN
```

Também podemos utilizar testes lógicos com a função `is.nan()`.

```
nao_numero <- 0/0
is.nan(nao_numero)
```

```
[1] TRUE
```

3.12.3 Inf

O **Inf** (infinito) representa um valor muito grande, o qual o R não consegue retratar. Também pode representar um limite matemático.

```
# Valor grande
100 ^ 200
```

```
[1] Inf
```

```
# Limite matemático
1 / 0
```

```
[1] Inf
```

```
-1 / 0
```

```
[1] -Inf
```

Novamente, podemos utilizar testes lógicos para identificar se um objeto apresenta valor infinito. Para isso, utilizamos a função `is.infinite()`.

```
valor_infinito <- 1 / 0
is.infinite(valor_infinito)
```

```
[1] TRUE
```

3.13 Listas

As listas são objetos semelhantes a um vetor, porém, com algumas diferenças. Como citado na seção 3.9, não podemos misturar objetos de classes distintas em um único vetor, contudo, nas **listas** podemos realizar essa mescla de classes.

Para criarmos uma lista, utilizamos a função `list()`, tendo como argumentos os valores desejados.

```
lista <- list(5, "x", FALSE)
lista
```

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

```
[[2]]
[1] "x"
```

```
[[3]]
[1] FALSE
```

```
class(lista)
```

```
[1] "list"
```

Nesse caso, criamos uma lista com elementos de classes numérica, caractere e lógica, sem que ocorresse coerção, ou seja, não houve a conversão do objeto para uma única classe, como vimos ocorrer com os vetores (seção 3.9). Assim, as listas nos permite unir classes distintas em um mesmo objeto.

Outro diferencial é o fato de que **cada elemento de uma lista também é uma lista**. Portanto, para acessarmos um elemento de uma lista, devemos utilizar dois colchetes `[[]]`.

```
lista <- list(5, "x", FALSE)

# Utilizando 1 colchete, nos retorna uma classe do tipo lista
lista[2]
```

```
[[1]]
[1] "x"
```

```
class(lista[2])
```

```
[1] "list"
```

```
# Utilizando 2 colchetes, nos retorna a classe do elemento
lista[[2]]
```

```
[1] "x"
```

```
class(lista[[2]])
```

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

Assim, utilizando um colchete, é retornado a classe da lista que contém um único elemento. Já com dois colchetes, nos é retornado a classe do elemento que está na lista, no caso do exemplo, a classe do x.

O fato de cada elemento ser uma lista dentro de uma lista é importante para podermos colocar vetores de tamanhos diferentes em cada posição. Isso faz das listas objetos muito flexíveis para armazenar dados.

```
listas_sao_flexiveis <- list(1:5, c("a", "b", "c"), c(TRUE, FALSE, TRUE, FALSE))
```

```
listas_sao_flexiveis[1]
```

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

```
listas_sao_flexiveis[2]
```

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

```
listas_sao_flexiveis[3]
```

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

Podemos nomear cada posição de uma lista. Para isso, colocamos dentro da função `list()` os argumentos com as respectivas denominações e valores.

```
dados_estudantes <- list(nome = c("José", "Joao", "Maria"),
                          sexo = c("M", "M", "F"),
                          idade = c(26, 19, 20))
```

```
dados_estudantes
```

```
$nome
[1] "José" "Joao" "Maria"
```

```
$sexo
[1] "M" "M" "F"
```

```
$idade
[1] 26 19 20
```

Quando a posição de uma lista possui um nome, podemos acessar seus valores com o operador `$`. Esse operador é equivalente a `dados_alunos[[]]`.

```
# Equivalente a dados_estudantes[[1]]
dados_estudantes$nome
```

```
[1] "José" "Joao" "Maria"
```

```
# Equivalente a dados_estudantes[[2]]
dados_estudantes$sexo
```

```
[1] "M" "M" "F"
```

```
# Equivalente a dados_estudantes[[3]]
dados_estudantes$idade
```

```
[1] 26 19 20
```

3.13.1 Data frames e listas

Depois de apresentarmos o que são listas (além dos demais conceitos presentes nas seções anteriores), vamos aprofundar os nossos conhecimentos sobre os *data frames*.

A relação entre listas e *data frames* é que, basicamente, os *data frames* são um tipo de lista. Assim, as propriedades expostas sobre as listas se aplicam aos *data frames*.

No capítulo 3.8, apresentamos o básico sobre os *data frames*, exemplificado com o `PlantGrowth`. Assim, para entendermos a equivalência entre listas e *data frames*, converteremos o `PlantGrowth` em uma lista.

```
# Data frame
head(PlantGrowth)
```

```
  weight group
1   4.17  ctrl
2   5.58  ctrl
3   5.18  ctrl
4   6.11  ctrl
5   4.50  ctrl
6   4.61  ctrl
```



```
# Lista
as.list(PlantGrowth)

$weight
 [1] 4.17 5.58 5.18 6.11 4.50 4.61 5.17 4.53 5.33 5.14 4.81 4.17 4.41 3.59 5.87
[16] 3.83 6.03 4.89 4.32 4.69 6.31 5.12 5.54 5.50 5.37 5.29 4.92 6.15 5.80 5.26

$group
 [1] ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl trt1 trt1 trt1 trt1 trt1
[16] trt1 trt1 trt1 trt1 trt1 trt1 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2
Levels: ctrl trt1 trt2
```

Perceba que os nomes das colunas do *data frame* se tornam o nome das posições de uma lista (`$weight` e `$group`) e cada valor das colunas são convertidos em elementos da respectiva lista.

Portanto, podemos dizer que cada coluna de um *data frame* também é um *data frame*, assim como no caso das listas.

```
# Classe do data frame PlantGrowth
class(PlantGrowth)
```

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

```
# Classe da primeira coluna (weight)
class(PlantGrowth[1])
```

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

Além disso, podemos utilizar o operador `$` para acessar os elementos de uma coluna (ou lista).

```
PlantGrowth$weight

 [1] 4.17 5.58 5.18 6.11 4.50 4.61 5.17 4.53 5.33 5.14 4.81 4.17 4.41 3.59 5.87
[16] 3.83 6.03 4.89 4.32 4.69 6.31 5.12 5.54 5.50 5.37 5.29 4.92 6.15 5.80 5.26
```

```
PlantGrowth$group

 [1] ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl trt1 trt1 trt1 trt1 trt1
[16] trt1 trt1 trt1 trt1 trt1 trt1 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2
Levels: ctrl trt1 trt2
```

Contudo, temos de destacar que todo *data frame* é um tipo de lista, porém nem toda lista é um *data frame*. Assim, o *data frame* possui algumas propriedades particulares que o torna um tipo de lista especial:

- Todas as colunas precisam ter a mesma dimensão, ou seja, ter o mesmo número de linhas;
- Todas as colunas precisam ser nomeadas;

- Possuir 2 dimensões.

Essas propriedades nos indicam que um *data frame* tem que receber uma base de dados em formato retangular (análoga a uma planilha Excel), com o mesmo número de linhas (observações) em cada coluna (variável), sendo necessário a presença um nome específico para cada uma das colunas.

Um exemplo da propriedade *Todas as colunas precisam ter a mesma dimensão* é a tentativa (falha) de converter uma lista com vetores de comprimentos diferentes em um *data frame*.

```
listas <- list(1:5, c("a", "b", "c"), c(TRUE, FALSE, TRUE, FALSE))
as.data.frame(listas)
```

```
Error in (function (... , row.names = NULL, check.rows = FALSE, check.names = TRUE, : arguments imply
```

A propriedade *Todas as colunas precisam ser nomeadas* não é um impeditivo para se criar um *data frame*, porém, não se obtém um bom resultado ao ignorar esta propriedade, pois é necessário ter um nome para cada coluna e o R trata de criá-los de uma maneira pouco agradável.

```
dados_estudantes <- list(c("José", "Joao", "Maria"),
                          c("M", "M", "F"),
                          c(45, 19, 26))

as.data.frame(dados_estudantes)
```

```
  c..José....Joao....Maria.. c..M....M....F.. c.45..19..26.
1                José                M                45
2                Joao                M                19
3                Maria                F                26
```

Já a propriedade *Possuir 2 dimensões* indica que o *data frame* possui linhas e colunas, o que os diferencia das listas, uma vez que essas não possuem dimensão. A função `dim()` retorna as dimensões de um objeto e comprova que as listas não possuem dimensão.

```
# Data frame
dim(PlantGrowth)
```

```
[1] 30  2
```

```
# Lista
dim(as.list(PlantGrowth))
```

```
NULL
```

Assim, a função `dim()` nos indica que o *data frame* `PlantGrowth` apresenta 30 linhas e 2 colunas. Por outro lado, ao convertermos o *data frame* `PlantGrowth` em uma lista, a mesma função nos diz que o objeto é ausente de dimensão (`NULL`).

Além disso, caso haja valores faltantes na base de dados (como, por exemplo, células vazias no Excel), esses serão representados por `NA`, sendo preservada a estrutura do *data frame*.

Por possuir duas dimensões, para acessarmos valores em um *data frame*, devemos especificar as linhas e colunas dentro de colchetes, na seguinte ordem: `[numero_linha, numero_coluna]`.

```
# Acessa o elemento posicionado na 10ª linha da 1ª coluna
PlantGrowth[10, 1]
```

```
[1] 5.14
```

Podemos pegar todas as linhas de uma coluna ou todas as colunas de uma linha deixando um dos argumentos vazios:

```
# Todas as linhas da 2ª coluna
PlantGrowth[,2]
```

```
[1] ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl ctrl trt1 trt1 trt1 trt1 trt1
[16] trt1 trt1 trt1 trt1 trt1 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2 trt2
Levels: ctrl trt1 trt2
```

```
# Todas as colunas da 2ª linhas
PlantGrowth[2,]
```

```
weight group
2 5.58 ctrl
```

E, devido ao fato de que cada coluna do *data frame* é um vetor, podemos aplicar testes lógicos para filtrar linhas, assim como fizemos na seção 3.11.

```
PlantGrowth[PlantGrowth$weight > 5.5, ]
```

```
weight group
2 5.58 ctrl
4 6.11 ctrl
15 5.87 trt1
17 6.03 trt1
21 6.31 trt2
23 5.54 trt2
28 6.15 trt2
29 5.80 trt2
```

Neste caso, a condição lógica imposta à coluna **weight** nos retorna todas as linhas que apresentem plantas com peso seco maior que 5,5.

Com as listas, finalizamos as noções básicas em R. Para aqueles que entraram em contato pela primeira vez com conceitos teóricos de programação, seja na linguagem que for, a teoria pode parecer complicada e maçante. Mas a não compreensão de alguns conceitos expostos neste capítulo não será um impeditivo para continuar os estudos em ciência de dados em R. Sinta-se livre em replicar os códigos dos próximos capítulos, mesmo que não compreenda 100% do que está sendo feito. Em algumas ocasiões, você compreenderá melhor um conceito ao aplicá-lo na prática. Além disso, recomendo que você revise este capítulo caso tenha alguma dúvida conceitual, ou senão, quando estiver mais familiarizado na programação em R.

Portanto, nos próximos capítulos, começaremos a aplicar ciência de dados na prática, começando pela importação de dados ao R.

Capítulo 4

Importação

Neste capítulo, iniciaremos a primeira etapa para começarmos a aplicar ciência de dados: a importação dos dados para o R. Deve-se ter em mente que um conjunto de dados pode estar em diversos formatos, seja em arquivo de texto, planilha Excel ou extensões de outros programas. Assim, para cada formato, haverá uma maneira específica de importá-los ao R. Nesta apostila, trataremos dos formatos mais usuais a serem importados: os arquivos texto e as planilhas Excel.

Como citado na seção 3.1, referente à criação de projetos no RStudio, devemos criar um projeto para cada trabalho realizado, logo, direcionar os arquivos a serem importados para o respectivo diretório, a fim de manter uma organização e facilitar o nosso acesso aos documentos necessários. Portanto, caso ainda não tenha criado um projeto, confira o capítulo referente ao tema.

A seguir, mostraremos como importar os dados em formato texto e planilha Excel. Para os exemplos, utilizaremos a base de dados referente a produção de cereais, extraída da FAOSTAT. Faça o *download* da pasta [clikando aqui](#). Nela contém a mesma base de dados em diversos formatos de arquivos para que você possa acompanhar os exemplos a seguir.

4.1 Pacote readr

Com o pacote `readr`, podemos ler arquivos em formato de texto, como os `.txt` e os `.csv`.

```
library(readr)
```

O `readr` transforma os arquivos em `tibbles`. No capítulo 5 veremos mais detalhes sobre as `tibbles`. Neste momento, focaremos em como importar os dados para o R, para então, entendermos os diferentes formatos de dados. A seguir, estão apresentadas as funções a serem utilizadas de acordo com o formato do arquivo:

- `read_csv()`: arquivos separados por vírgula;
- `read_csv2()`: arquivos separados por ponto-e-vírgula;
- `read_tsv()`: arquivos separados por tabulação;
- `read_delim()`: arquivos separados por um delimitador genérico. Requer o argumento `delim =` para indicar qual o caractere que separa as colunas do arquivo texto;

- `read_table()`: arquivos com colunas separadas por espaço.

Para entendermos melhor o que são separadores, basicamente, em um arquivo texto, as colunas do Excel são delimitadas por um operador, ou seja, um separador de colunas, podendo ser vírgulas, ponto-e-vírgulas, espaços, TAB ou um delimitador genérico. Para isso, o `readr` apresenta diferentes funções para conseguir importar estes arquivos no formato correto, de acordo com o tipo de arquivo e delimitador utilizado no arquivo.

4.1.1 Ler arquivos texto

Importaremos arquivos texto nos formatos `.csv` e `.txt`.

Para ler um arquivo em `.csv`, cujo separador de colunas são as vírgulas, utilizamos a função `read_csv()`.

```
cereais_csv <- read_csv(file = "cereais.csv")

-- Column specification -----
cols(
  `Domain Code` = col_character(),
  Domain = col_character(),
  `Area Code` = col_double(),
  Area = col_character(),
  `Element Code` = col_double(),
  Element = col_character(),
  `Item Code` = col_double(),
  Item = col_character(),
  `Year Code` = col_double(),
  Year = col_double(),
  Unit = col_character(),
  Value = col_double(),
  Flag = col_character(),
  `Flag Description` = col_character()
)
```

Como resultado da função, para cada variável (coluna) um tipo de classe foi atribuída, indicando se contém caracteres (`col_character()`) ou valores numéricos (`col_double()`). Devemos nos atentar a este passo, pois em algumas situações, a classificação das variáveis pode estar errada ou imprópria para o uso correto nas análises. Logo abaixo, apresentaremos o argumento `col_types` = para reclassificar as variáveis.

O argumento `file` = representa o caminho até o arquivo. Perceba que não foi preciso indicar o diretório do arquivo, pois esse se localiza na mesma pasta do projeto em uso. Caso o arquivo esteja em uma outra pasta presente no diretório do projeto em uso, devemos especificá-la dentro do caminho até o arquivo. Exemplificaremos o processo com a pasta de nome `dados_importar`.

```
cereais_csv <- read_csv(file = "dados_importar/cereais.csv")
```

Em algumas situações, as colunas dos arquivos `.csv` são separadas por ponto-e-vírgula, pois os separadores decimais são as vírgulas, como é o caso da sintaxe utilizada no Brasil. Nesse caso, deve-se utilizar a função `read_csv2()`.

```
cereais_csv2 <- read_csv2(file = "cereais2.csv")
```

Já os arquivos `.txt` são lidos a partir da função `read_delim()`, sendo necessário indicar qual caractere é utilizado para separar as colunas do arquivo a ser importado.

No exemplo a seguir, importaremos um arquivo `.txt`, separado por tabulação, sendo que o código `\t` representa a tecla TAB.

```
cereais_txt <- read_delim(file = "cereais.txt", delim = "\t")
```

Também podemos ler o mesmo arquivo anterior pela função `read_tsv()`, pois o separador utilizado é a tabulação. Assim, não precisamos utilizar o argumento `delim` = nesse caso, pois, por padrão dessa função, ela entende que o separador utilizado é a tabulação.

```
cereais2_tab <- read_tsv(file = "cereais.txt")
```

4.1.2 Outros argumentos

Alguns outros argumentos estão presentes na maioria das funções de leitura de arquivos do pacote `readr`. Essas funções auxiliam na organização do banco de dados, pois visam arrumar alguns detalhes antes mesmo de importá-los ao R. A seguir, citaremos alguns deles:

- `skip` =: pula linhas do começo do arquivo antes da importação. Muito útil para evitar possíveis textos presentes no início do arquivo. Indique no argumento a quantidade de linhas a serem puladas (por exemplo, pular as duas primeiras linhas: `skip = 2`);
- `comment` =: no caso de arquivos que possuem algum caractere padrão que precede os comentários, usamos esse argumento para indicar qual o caractere utilizado. Por exemplo, caso o caractere `#` venha antes de todos os comentários presentes em um arquivo, utilizamos o `comment = "#"` para que o arquivo importado venha sem a parte comentada;
- `na` =: atribui valor NA a determinado caractere especificado no argumento. Aceita um vetor como objeto do argumento. Como exemplo, atribuiremos valor NA ao texto `Brazil`, presente na coluna `Area`, a fim de ilustrar o argumento.

```
cereais_csv <- read_csv(file = "cereais.csv",
                        na = "Brazil")
```

```
head(cereais_csv)
```

```
# A tibble: 6 x 14
  `Domain Code` Domain `Area Code` Area `Element Code` Element `Item Code`
  <chr>          <chr>    <dbl> <lg1>          <dbl> <chr>          <dbl>
1 QC           Crops      21 NA           5312 Area harves~    44
2 QC           Crops      21 NA           5312 Area harves~    44
3 QC           Crops      21 NA           5312 Area harves~    44
4 QC           Crops      21 NA           5312 Area harves~    44
5 QC           Crops      21 NA           5312 Area harves~    44
6 QC           Crops      21 NA           5312 Area harves~    44
# ... with 7 more variables: Item <chr>, Year Code <dbl>, Year <dbl>,
#   Unit <chr>, Value <dbl>, Flag <chr>, Flag Description <chr>
```

- `col_names` =: por padrão, a função assume que a primeira linha da base de dados é o nome das colunas. Caso a base de dados não venha com os nomes das colunas, utilizamos o argumento `col_names = FALSE`. Além disso, também pode ser utilizado para (re)nomear as colunas.

```
cereais_csv <- read_csv(file = "cereais.csv",
                        col_names = c("cd", "cod", "ac", "pais", "ec",
                                      "elemento", "ic", "cultura", "yc",
                                      "ano", "unidade", "valor", "flag",
                                      "descricao"),
                        skip = 1)

head(cereais_csv)
```

```
# A tibble: 6 x 14
   cd   cod   ac pais   ec elemento   ic cultura   yc   ano unidade valor
  <chr> <chr> <dbl> <chr> <dbl> <chr>   <dbl> <chr>   <dbl> <dbl> <chr>   <dbl>
1 QC   Crops  21 Braz~ 5312 Area ha~ 44 Barley 1961 1961 ha   31511
2 QC   Crops  21 Braz~ 5312 Area ha~ 44 Barley 1962 1962 ha   28454
3 QC   Crops  21 Braz~ 5312 Area ha~ 44 Barley 1963 1963 ha   30443
4 QC   Crops  21 Braz~ 5312 Area ha~ 44 Barley 1964 1964 ha   31164
5 QC   Crops  21 Braz~ 5312 Area ha~ 44 Barley 1965 1965 ha   33550
6 QC   Crops  21 Braz~ 5312 Area ha~ 44 Barley 1966 1966 ha   41175
# ... with 2 more variables: flag <chr>, descricao <chr>
```

Como exemplo, renomeamos as colunas com o argumento `col_names = c()`, sendo que cada coluna deve ser, obrigatoriamente, renomeada ou conter o mesmo nome anterior, porém reescrito dentro do vetor, na ordem correta. Perceba que temos que pular a primeira linha (`skip = 1`), pois após a renomeação, a primeira linha do *data frame* será o antigo nome das colunas. Por outro lado, caso a base de dados tenha vindo sem nome, o argumento `skip` não precisa estar presente.

- `col_types` =: no caso de alguma coluna ser importada com a classe errada, pode-se usar esse argumento para alterar a classe. Será exemplificada a transformação da variável `Element` em fator e da variável `Year` em números inteiros.

```
cereais_csv2 <- read_csv2(file = "cereais2.csv",
                         col_types = cols(
                           Element = col_factor(),
                           Year = col_integer()))

class(cereais_csv2$Element)
```

```
[1] "factor"
```

```
class(cereais_csv2$Year)
```

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

- `locale` =: esse argumento define opções de formatações de certas localidades, como o idioma, formato de datas e horas, separador decimal e *encoding*. É utilizada dentro da função `read_`, tendo como objeto a função de mesmo nome, `locale()`. A seguir, veremos alguns exemplos de aplicações desse argumento.


```
# Função locale() - verifica quais padrões estão sendo utilizados
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
```

```
# Troca os dias e meses para português
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
```

```
# Troca o separador decimal de ponto para vírgula
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
```

Outro problema que podemos resolver com o argumento é o *encoding* de arquivos. *Encoding* é a forma como o computador traduz os caracteres replicados no R para valores binários. Há diversos

tipos de *encoding*, sendo que o Windows utiliza um diferente em relação ao Linux e Mac. Com isso, é comum termos problemas de *encoding* no Windows quando um arquivo é criado em um desses sistemas operacionais, havendo a desconfiguração de letras com acentos e outros caracteres especiais após a importação para o R.

A seguir, demonstraremos o que são os problemas de *encoding* no Windows.

```
poema_drummond <- "E agora, José? A festa acabou, a luz apagou,
o povo sumiu,a noite esfriou, e agora, José? (...)"

# Verificando encoding no Windows
Encoding(poema_drummond)
```

```
[1] "latin1"
```

```
# Forçando problema de encoding no Windows
Encoding(poema_drummond) <- "UTF-8"
poema_drummond
```

```
[1] "E agora, Jos<e9>? A festa acabou, a luz apagou, \no povo sumiu,a noite esfriou, e agora, Jos<e9>"
```

Para corrigir o *encoding* no Windows, devemos atribuir o *encoding* latin1.

Assim, dependendo de qual sistema operacional um arquivo foi executado, devemos converter para diferentes *encoding*.

```
# Dados via Windows: usuário de Windows, converter para "latin1"
read_csv("dados_via_Windows.csv", locale = locale(encoding = "latin1"))

# Dados via Linux/Mac: usuário de Windows, converter para "UTF-8"
read_csv("dados_via_linux_mac.csv", locale = locale(encoding = "UTF-8"))
```

Outra funcionalidade do argumento `locale =` é o de *parsear* valores. Esse termo, comum no meio da programação, significa arrumar ou formatar um “valor A” para um “valor B”, a partir das similaridades entre ambos.

Como exemplo, podemos converter números que estão em formato de caractere para o formato de números, efetivamente:

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

```
[1] 5 5 50 5 5
```

Podemos realizar o mesmo procedimento especificando o *parseamento*:

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

```
[1] 5
```

Ainda, podemos *parsear* datas, de acordo com o idioma:

```
# Inglês
parse_date("08/July/2021",
           format = "%d/%B/%Y")
```

```
[1] "2021-07-08"
```

```
# Português
parse_date("08/Julho/2021",
           format = "%d/%B/%Y",
           locale = locale(date_names = "pt"))
```

```
[1] "2021-07-08"
```

Para conferir todos os argumentos presentes nas funções do pacote `readr`, podemos utilizar a função `args(nome_da_função)`. Exemplificaremos com a função `read_csv()`.

```
args(read_csv)

function (file, col_names = TRUE, col_types = NULL, locale = default_locale(),
         na = c("", "NA"), quoted_na = TRUE, quote = "\"", comment = "",
         trim_ws = TRUE, skip = 0, n_max = Inf, guess_max = min(1000,
         n_max), progress = show_progress(), skip_empty_rows = TRUE)
NULL
```

4.1.3 Escrever arquivos texto

Também temos a opção de salvar uma base de dados contida no R para um formato específico de arquivo. Para tanto, utilizamos a função `write_`, acompanhada dos argumentos `x =` e `file =`, referentes, respectivamente, ao objeto a ser escrito e ao nome do arquivo a ser criado.

Nos exemplos a seguir, escreveremos dados para os formatos `.csv` e `.txt`. Utilizaremos os dados do `mtcars`, presente no banco de dados nativo do R.

```
# Arquivo .csv separado por vírgula
write_csv(x = mtcars, file = "mtcars.csv")

# Arquivo .csv separado por ponto-e-vírgula
write_csv2(x = mtcars, file = "mtcars.csv")

# Arquivo .txt, separado por tabulação
write_delim(x = mtcars, file = "mtcars.txt", delim = "\t")
```

O arquivo escrito estará localizado no diretório referente ao projeto em atividade. Caso queira definir outro local para armazenar o arquivo, especifique-o no argumento `file =`. Por exemplo, caso se deseje salvar o arquivo em uma pasta chamada `banco_de_dados`, localizada no diretório do projeto em uso, devemos prosseguir da seguinte maneira:

```
# Arquivo .csv separado por vírgula
write_csv(x = mtcars, file = "banco_de_dados/mtcars.csv")

# Arquivo .csv separado por ponto-e-vírgula
write_csv2(x = mtcars, file = "banco_de_dados/mtcars.csv")

# Arquivo .txt, separado por tabulação
write_delim(x = mtcars, file = "banco_de_dados/mtcars.txt", delim = "\t")
```

4.2 Pacote readxl

4.2.1 Ler arquivos Excel

O pacote `readxl` lê e importa planilhas Excel em formato `.xlsx` e `.xls`. Para tanto, utilizamos a função `read_excel()`. Esta função identifica automaticamente qual a extensão do arquivo, seja `.xlsx` ou `.xls`.

```
library(readxl)

cereais_xlsx <- read_excel("cereais.xlsx")
```

Neste mesmo pacote, estão presentes alguns exemplos de arquivos nos formatos `.xlsx` e `.xls`. Para conferi-los, utilizamos a função `readxl_example()`.

```
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"
```

Para utilizar um destes dados, devemos verificar em qual diretório estão localizados. Vamos exemplificar com o arquivo `"deaths.xlsx"`, salvando sua localização no objeto `local_arquivo`.

```
local_arquivo <- readxl_example("deaths.xlsx")
local_arquivo
```

```
[1] "C:/R/R-4.0.4/library/readxl/extdata/deaths.xlsx"
```

Em alguns casos, um arquivo Excel pode conter diversas planilhas. Para verificar quais planilhas estão presentes, utilizamos a função `excel_sheets()`.

```
excel_sheets(local_arquivo)
```

```
[1] "arts"  "other"
```

Perceba que o arquivo Excel `deaths` apresenta duas planilhas: “arts” e “other”. Para selecionar a planilha desejada, devemos utilizar o argumento `sheet` = dentro da função `read_excel()`. Caso não seja utilizado o argumento, por padrão, será selecionada a primeira planilha contida no arquivo.

```
# Primeira planilha
read_excel(local_arquivo)
```

```
# A tibble: 18 x 6
  `Lots of people`    ...2      ...3    ...4    ...5      ...6
  <chr>              <chr>      <chr> <chr> <chr>      <chr>
1 simply cannot resist w~ <NA>      <NA> <NA> <NA>      some notes
2 at                  the         top    <NA> of      their sprea~
3 or                  merging    <NA> <NA> <NA>      cells
4 Name                Profession Age      Has k~ Date of ~ Date of dea~
5 David Bowie         musician  69      TRUE  17175    42379
6 Carrie Fisher       actor     60      TRUE  20749    42731
7 Chuck Berry         musician  90      TRUE  9788     42812
8 Bill Paxton         actor     61      TRUE  20226    42791
9 Prince              musician  57      TRUE  21343    42481
10 Alan Rickman        actor     69      FALSE 16854    42383
11 Florence Henderson actor     82      TRUE  12464    42698
12 Harper Lee          author    89      FALSE 9615     42419
13 Zsa Zsa Gábor       actor     99      TRUE  6247     42722
14 George Michael     musician  53      FALSE 23187    42729
15 Some               <NA>      <NA> <NA> <NA>      <NA>
16 <NA>               also like to wr~ <NA> <NA> <NA>      <NA>
17 <NA>               <NA>      at the botto~ <NA> <NA>
18 <NA>               <NA>      <NA> <NA> <NA>      too!
```

```
# Segunda planilha
read_excel(local_arquivo, sheet = 2)
```

```
# A tibble: 18 x 6
  `For the sake`    ...2      ...3    ...4    ...5      ...6
  <chr>              <chr>      <chr> <chr> <chr>      <chr>
1 <NA>              of consistency <NA> <NA> in the    data layout,
2 which is really <NA>      <NA> <NA> a        beautiful th~
3 I will            keep making notes <NA> <NA> <NA>      up here.
4 Name                Profession Age      Has kids Date of bi~ Date of death
5 Vera Rubin         scientist  88      TRUE  10432    42729
6 Mohamed Ali        athlete   74      TRUE  15358    42524
7 Morley Safer       journalist 84      TRUE  11635    42509
8 Fidel Castro       politician 90      TRUE  9722     42699
9 Antonin Scalia     lawyer    79      TRUE  13220    42413
10 Jo Cox             politician 41      TRUE  27202    42537
11 Janet Reno         lawyer    78      FALSE 14082    42681
12 Gwen Ifill         journalist 61      FALSE 20361    42688
13 John Glenn         astronaut 95      TRUE  7880     42712
14 Pat Summit         coach     64      TRUE  19159    42549
15 This              <NA>      <NA> <NA> <NA>      <NA>
16 <NA>              has been really fun~ <NA> <NA> <NA>      <NA>
```

```
17 we're signing <NA> <NA> <NA> <NA>
18 <NA> <NA> off <NA> now! <NA>
```

```
# Pega a segunda planilha, agora pelo seu nome
read_excel(local_arquivo, sheet = "other")
```

```
# A tibble: 18 x 6
  `For the sake` ...2 ...3 ...4 ...5 ...6
  <chr> <chr> <chr> <chr> <chr> <chr>
1 <NA> of consistency <NA> <NA> in the data layout,
2 which is really <NA> <NA> <NA> a beautiful th~
3 I will keep making notes <NA> <NA> <NA> up here.
4 Name Profession Age Has kids Date of bi~ Date of death
5 Vera Rubin scientist 88 TRUE 10432 42729
6 Mohamed Ali athlete 74 TRUE 15358 42524
7 Morley Safer journalist 84 TRUE 11635 42509
8 Fidel Castro politician 90 TRUE 9722 42699
9 Antonin Scalia lawyer 79 TRUE 13220 42413
10 Jo Cox politician 41 TRUE 27202 42537
11 Janet Reno lawyer 78 FALSE 14082 42681
12 Gwen Ifill journalist 61 FALSE 20361 42688
13 John Glenn astronaut 95 TRUE 7880 42712
14 Pat Summit coach 64 TRUE 19159 42549
15 This <NA> <NA> <NA> <NA> <NA>
16 <NA> has been really fun~ <NA> <NA> <NA> <NA>
17 we're signing <NA> <NA> <NA> <NA> <NA>
18 <NA> <NA> off <NA> now! <NA>
```

Podemos verificar outros argumentos presentes nas funções do pacote `readxl` com a função `args(nome_da_função)`. Como exemplo, veremos os argumentos presentes na função `read_excel()`.

```
args(read_excel)
```

```
function (path, sheet = NULL, range = NULL, col_names = TRUE,
  col_types = NULL, na = "", trim_ws = TRUE, skip = 0, n_max = Inf,
  guess_max = min(1000, n_max), progress = readxl_progress(),
  .name_repair = "unique")
NULL
```

Como se pode notar, há diversos argumentos idênticos aos demonstrados na subseção 4.1.2, referente ao pacote `readr`. Contudo, na função `read_excel()` não temos o argumento `locale`, sendo uma falta relevante para que possamos resolver problemas relacionados à temática. Como alternativa, podemos priorizar a importação de arquivos texto ao R, ao invés de planilhas Excel, sendo facilmente resolvido salvando o Excel em formato texto, seja em `.csv` ou `.txt`.

4.2.2 Escrever arquivos Excel

Também podemos escrever um arquivo Excel em formato `.xlsx` utilizando a função `write_xlsx`, contida no pacote `writexl`. Utilizaremos novamente os dados `mtcars` para demonstração.

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

write_xlsx(mtcars, "mtcars.xlsx")
```

4.3 Banco de dados

A seguir, listaremos alguns sites que fornecem uma grande variedade de base de dados públicos. Confira algum que apresente dados de seu interesse, para que você possa aplicá-los futuramente no R.

- FAOSTAT: <https://www.fao.org/faostat/en/#data>
- IBGE: <https://sidra.ibge.gov.br/acervo#/A/Q>
- USDA: <https://apps.fas.usda.gov/psdonline/app/index.html#/app/advQuery>
- DataBank: <https://databank.worldbank.org/home.aspx>
- Kaggle: <https://www.kaggle.com/datasets>;
- Dados do Agro: <https://dados.agr.br/fonte-de-dados/>

Como pudemos perceber, a importação de dados para o R não é uma tarefa difícil. Devemos nos atentar ao formato em que o arquivo está, a fim de utilizarmos a função de importação correta. Dependendo da fonte dos dados, esses podem vir com algumas configurações indesejadas, o que pode ser resolvido ainda antes de importarmos os dados, utilizando funções e argumentos específicos para cada necessidade. Por fim, também conhecemos funções que escrevem os dados do R para diversos formatos, sejam arquivos texto ou Excel.

No próximo capítulo, trataremos da etapa de **arrumar** os dados. Serão apresentadas as **tibbles**, um tipo de *data frame* que utilizaremos muito, além de explorarmos as funcionalidades presentes no pacote **tidyr**, a fim de obtermos uma base de dados organizada e fácil de se trabalhar.

Capítulo 5

Organização

Neste capítulo, trataremos a fundo a manipulação dos dados. Esse processo tem origem no conceito *tidy data*, no qual devemos ajustar os nossos dados em uma estrutura consistente e padronizada. Isso nos permite visualizar, analisar e modelar os dados de maneira mais fácil e rápida. Normalmente, costuma ser trabalhoso o processo de organizar a base de dados, mas, com o auxílio das ferramentas presentes no **tidyverse**, conseguimos obter bons resultados que, em longo prazo, faz o esforço valer a pena.

Como o próprio nome do pacote sugere, o **tidyverse** é baseado nos princípios de uma *tidy data*. Por isso, seus pacotes foram desenvolvidos para funcionarem seguindo tal princípio e serem complementares entre si.

No capítulo 4, demos início ao processo de manipulação dos dados quando alteramos a classe das variáveis, renomeamos as colunas, corrigimos problemas de *encoding*, além de outras ações demonstradas a partir das diversas funções e argumentos.

Neste capítulo, veremos como começar a arrumar uma base de dados para, posteriormente, transformar variáveis de acordo com os interesses de uma análise (6) e visualizar os dados de forma gráfica (7).

Para mais detalhes sobre *tidy data*, recomendo dois documentos que abordam o tema. Ambos são de autoria de Hadley Wickham, o idealizador do pacote **tidyverse**. O primeiro é o [The tidy tools manifesto](#), que aborda os princípios que norteiam o *tidyverse*. O outro documento é o artigo [Tidy Data](#), que trata de maneira teórica os conceitos de dados arrumados.

Assim sendo, vamos começar a organizar os nossos dados. A seguir, apresentaremos os pacotes **tibble** e **tidyr**.

5.1 Tibbles

Nesta seção, veremos o que são as *tibbles* e suas funcionalidades. No capítulo 4 vimos que o pacote **readr** apresenta funções que convertem diretamente os arquivos importados para o formato *tibble*. Isso se deve ao fato de que os pacotes presentes no **tidyverse** utilizam como padrão o formato *tibble*, ao invés do formato tradicional de *data frame*. Porém, a maioria dos outros pacotes que não estão no **tidyverse** ainda utilizam o formato tradicional de *data frame*.

Tibbles nada mais são do que uma versão mais atualizada dos *data frames*, apresentando ajustes importantes que facilitam o trabalho do cientista de dados. Dessa forma, os conceitos que vimos nas seções 3.8 e 3.13.1 são válidos para as *tibbles*. Elas fazem parte do pacote **tibble**, assim, devemos carregar o seu pacote.

```
library(tibble)
```

5.1.1 Criando *Tibbles*

Para converter *data frames* em *tibbles*, utilizamos a função `as_tibble()`. Como exemplo, transformaremos a base de dados nativa do R *iris* para o formato *tibble*.

```
as_tibble(iris)
```

```
# A tibble: 150 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
      <dbl>      <dbl>      <dbl>      <dbl> <fct>
1         5.1         3.5         1.4         0.2 setosa
2         4.9         3         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         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         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
```

Perceba que, por padrão, apenas as 10 primeiras linhas são apresentadas. Caso o número de colunas não couber na largura da tela, essas serão ocultadas da apresentação. Além disso, as *tibbles* mostram as dimensões da tabela (no caso, 150 x 5) e a classe de todas as colunas (entre < >).

Já os *data frames*, não apresentam as dimensões da tabela e nem as respectivas classes das colunas. Além disso, sua saída não é a muito boa quando trabalhamos com bases de dados extensas, como podemos ver e comparar a seguir.

```
class(iris)
```

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

```
iris
```

| | Sepal.Length | Sepal.Width | Petal.Length | Petal.Width | Species |
|----|--------------|-------------|--------------|-------------|---------|
| 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 |

| | | | | | |
|----|-----|-----|-----|-----|------------|
| 11 | 5.4 | 3.7 | 1.5 | 0.2 | setosa |
| 12 | 4.8 | 3.4 | 1.6 | 0.2 | setosa |
| 13 | 4.8 | 3.0 | 1.4 | 0.1 | setosa |
| 14 | 4.3 | 3.0 | 1.1 | 0.1 | setosa |
| 15 | 5.8 | 4.0 | 1.2 | 0.2 | setosa |
| 16 | 5.7 | 4.4 | 1.5 | 0.4 | setosa |
| 17 | 5.4 | 3.9 | 1.3 | 0.4 | setosa |
| 18 | 5.1 | 3.5 | 1.4 | 0.3 | setosa |
| 19 | 5.7 | 3.8 | 1.7 | 0.3 | setosa |
| 20 | 5.1 | 3.8 | 1.5 | 0.3 | setosa |
| 21 | 5.4 | 3.4 | 1.7 | 0.2 | setosa |
| 22 | 5.1 | 3.7 | 1.5 | 0.4 | setosa |
| 23 | 4.6 | 3.6 | 1.0 | 0.2 | setosa |
| 24 | 5.1 | 3.3 | 1.7 | 0.5 | setosa |
| 25 | 4.8 | 3.4 | 1.9 | 0.2 | setosa |
| 26 | 5.0 | 3.0 | 1.6 | 0.2 | setosa |
| 27 | 5.0 | 3.4 | 1.6 | 0.4 | setosa |
| 28 | 5.2 | 3.5 | 1.5 | 0.2 | setosa |
| 29 | 5.2 | 3.4 | 1.4 | 0.2 | setosa |
| 30 | 4.7 | 3.2 | 1.6 | 0.2 | setosa |
| 31 | 4.8 | 3.1 | 1.6 | 0.2 | setosa |
| 32 | 5.4 | 3.4 | 1.5 | 0.4 | setosa |
| 33 | 5.2 | 4.1 | 1.5 | 0.1 | setosa |
| 34 | 5.5 | 4.2 | 1.4 | 0.2 | setosa |
| 35 | 4.9 | 3.1 | 1.5 | 0.2 | setosa |
| 36 | 5.0 | 3.2 | 1.2 | 0.2 | setosa |
| 37 | 5.5 | 3.5 | 1.3 | 0.2 | setosa |
| 38 | 4.9 | 3.6 | 1.4 | 0.1 | setosa |
| 39 | 4.4 | 3.0 | 1.3 | 0.2 | setosa |
| 40 | 5.1 | 3.4 | 1.5 | 0.2 | setosa |
| 41 | 5.0 | 3.5 | 1.3 | 0.3 | setosa |
| 42 | 4.5 | 2.3 | 1.3 | 0.3 | setosa |
| 43 | 4.4 | 3.2 | 1.3 | 0.2 | setosa |
| 44 | 5.0 | 3.5 | 1.6 | 0.6 | setosa |
| 45 | 5.1 | 3.8 | 1.9 | 0.4 | setosa |
| 46 | 4.8 | 3.0 | 1.4 | 0.3 | setosa |
| 47 | 5.1 | 3.8 | 1.6 | 0.2 | setosa |
| 48 | 4.6 | 3.2 | 1.4 | 0.2 | setosa |
| 49 | 5.3 | 3.7 | 1.5 | 0.2 | setosa |
| 50 | 5.0 | 3.3 | 1.4 | 0.2 | setosa |
| 51 | 7.0 | 3.2 | 4.7 | 1.4 | versicolor |
| 52 | 6.4 | 3.2 | 4.5 | 1.5 | versicolor |
| 53 | 6.9 | 3.1 | 4.9 | 1.5 | versicolor |
| 54 | 5.5 | 2.3 | 4.0 | 1.3 | versicolor |
| 55 | 6.5 | 2.8 | 4.6 | 1.5 | versicolor |
| 56 | 5.7 | 2.8 | 4.5 | 1.3 | versicolor |
| 57 | 6.3 | 3.3 | 4.7 | 1.6 | versicolor |
| 58 | 4.9 | 2.4 | 3.3 | 1.0 | versicolor |
| 59 | 6.6 | 2.9 | 4.6 | 1.3 | versicolor |
| 60 | 5.2 | 2.7 | 3.9 | 1.4 | versicolor |
| 61 | 5.0 | 2.0 | 3.5 | 1.0 | versicolor |
| 62 | 5.9 | 3.0 | 4.2 | 1.5 | versicolor |

| | | | | |
|-----|-----|-----|-----|----------------|
| 63 | 6.0 | 2.2 | 4.0 | 1.0 versicolor |
| 64 | 6.1 | 2.9 | 4.7 | 1.4 versicolor |
| 65 | 5.6 | 2.9 | 3.6 | 1.3 versicolor |
| 66 | 6.7 | 3.1 | 4.4 | 1.4 versicolor |
| 67 | 5.6 | 3.0 | 4.5 | 1.5 versicolor |
| 68 | 5.8 | 2.7 | 4.1 | 1.0 versicolor |
| 69 | 6.2 | 2.2 | 4.5 | 1.5 versicolor |
| 70 | 5.6 | 2.5 | 3.9 | 1.1 versicolor |
| 71 | 5.9 | 3.2 | 4.8 | 1.8 versicolor |
| 72 | 6.1 | 2.8 | 4.0 | 1.3 versicolor |
| 73 | 6.3 | 2.5 | 4.9 | 1.5 versicolor |
| 74 | 6.1 | 2.8 | 4.7 | 1.2 versicolor |
| 75 | 6.4 | 2.9 | 4.3 | 1.3 versicolor |
| 76 | 6.6 | 3.0 | 4.4 | 1.4 versicolor |
| 77 | 6.8 | 2.8 | 4.8 | 1.4 versicolor |
| 78 | 6.7 | 3.0 | 5.0 | 1.7 versicolor |
| 79 | 6.0 | 2.9 | 4.5 | 1.5 versicolor |
| 80 | 5.7 | 2.6 | 3.5 | 1.0 versicolor |
| 81 | 5.5 | 2.4 | 3.8 | 1.1 versicolor |
| 82 | 5.5 | 2.4 | 3.7 | 1.0 versicolor |
| 83 | 5.8 | 2.7 | 3.9 | 1.2 versicolor |
| 84 | 6.0 | 2.7 | 5.1 | 1.6 versicolor |
| 85 | 5.4 | 3.0 | 4.5 | 1.5 versicolor |
| 86 | 6.0 | 3.4 | 4.5 | 1.6 versicolor |
| 87 | 6.7 | 3.1 | 4.7 | 1.5 versicolor |
| 88 | 6.3 | 2.3 | 4.4 | 1.3 versicolor |
| 89 | 5.6 | 3.0 | 4.1 | 1.3 versicolor |
| 90 | 5.5 | 2.5 | 4.0 | 1.3 versicolor |
| 91 | 5.5 | 2.6 | 4.4 | 1.2 versicolor |
| 92 | 6.1 | 3.0 | 4.6 | 1.4 versicolor |
| 93 | 5.8 | 2.6 | 4.0 | 1.2 versicolor |
| 94 | 5.0 | 2.3 | 3.3 | 1.0 versicolor |
| 95 | 5.6 | 2.7 | 4.2 | 1.3 versicolor |
| 96 | 5.7 | 3.0 | 4.2 | 1.2 versicolor |
| 97 | 5.7 | 2.9 | 4.2 | 1.3 versicolor |
| 98 | 6.2 | 2.9 | 4.3 | 1.3 versicolor |
| 99 | 5.1 | 2.5 | 3.0 | 1.1 versicolor |
| 100 | 5.7 | 2.8 | 4.1 | 1.3 versicolor |
| 101 | 6.3 | 3.3 | 6.0 | 2.5 virginica |
| 102 | 5.8 | 2.7 | 5.1 | 1.9 virginica |
| 103 | 7.1 | 3.0 | 5.9 | 2.1 virginica |
| 104 | 6.3 | 2.9 | 5.6 | 1.8 virginica |
| 105 | 6.5 | 3.0 | 5.8 | 2.2 virginica |
| 106 | 7.6 | 3.0 | 6.6 | 2.1 virginica |
| 107 | 4.9 | 2.5 | 4.5 | 1.7 virginica |
| 108 | 7.3 | 2.9 | 6.3 | 1.8 virginica |
| 109 | 6.7 | 2.5 | 5.8 | 1.8 virginica |
| 110 | 7.2 | 3.6 | 6.1 | 2.5 virginica |
| 111 | 6.5 | 3.2 | 5.1 | 2.0 virginica |
| 112 | 6.4 | 2.7 | 5.3 | 1.9 virginica |
| 113 | 6.8 | 3.0 | 5.5 | 2.1 virginica |
| 114 | 5.7 | 2.5 | 5.0 | 2.0 virginica |

| | | | | | |
|-----|-----|-----|-----|-----|-----------|
| 115 | 5.8 | 2.8 | 5.1 | 2.4 | virginica |
| 116 | 6.4 | 3.2 | 5.3 | 2.3 | virginica |
| 117 | 6.5 | 3.0 | 5.5 | 1.8 | virginica |
| 118 | 7.7 | 3.8 | 6.7 | 2.2 | virginica |
| 119 | 7.7 | 2.6 | 6.9 | 2.3 | virginica |
| 120 | 6.0 | 2.2 | 5.0 | 1.5 | virginica |
| 121 | 6.9 | 3.2 | 5.7 | 2.3 | virginica |
| 122 | 5.6 | 2.8 | 4.9 | 2.0 | virginica |
| 123 | 7.7 | 2.8 | 6.7 | 2.0 | virginica |
| 124 | 6.3 | 2.7 | 4.9 | 1.8 | virginica |
| 125 | 6.7 | 3.3 | 5.7 | 2.1 | virginica |
| 126 | 7.2 | 3.2 | 6.0 | 1.8 | virginica |
| 127 | 6.2 | 2.8 | 4.8 | 1.8 | virginica |
| 128 | 6.1 | 3.0 | 4.9 | 1.8 | virginica |
| 129 | 6.4 | 2.8 | 5.6 | 2.1 | virginica |
| 130 | 7.2 | 3.0 | 5.8 | 1.6 | virginica |
| 131 | 7.4 | 2.8 | 6.1 | 1.9 | virginica |
| 132 | 7.9 | 3.8 | 6.4 | 2.0 | virginica |
| 133 | 6.4 | 2.8 | 5.6 | 2.2 | virginica |
| 134 | 6.3 | 2.8 | 5.1 | 1.5 | virginica |
| 135 | 6.1 | 2.6 | 5.6 | 1.4 | virginica |
| 136 | 7.7 | 3.0 | 6.1 | 2.3 | virginica |
| 137 | 6.3 | 3.4 | 5.6 | 2.4 | virginica |
| 138 | 6.4 | 3.1 | 5.5 | 1.8 | virginica |
| 139 | 6.0 | 3.0 | 4.8 | 1.8 | virginica |
| 140 | 6.9 | 3.1 | 5.4 | 2.1 | virginica |
| 141 | 6.7 | 3.1 | 5.6 | 2.4 | virginica |
| 142 | 6.9 | 3.1 | 5.1 | 2.3 | virginica |
| 143 | 5.8 | 2.7 | 5.1 | 1.9 | virginica |
| 144 | 6.8 | 3.2 | 5.9 | 2.3 | virginica |
| 145 | 6.7 | 3.3 | 5.7 | 2.5 | virginica |
| 146 | 6.7 | 3.0 | 5.2 | 2.3 | virginica |
| 147 | 6.3 | 2.5 | 5.0 | 1.9 | virginica |
| 148 | 6.5 | 3.0 | 5.2 | 2.0 | virginica |
| 149 | 6.2 | 3.4 | 5.4 | 2.3 | virginica |
| 150 | 5.9 | 3.0 | 5.1 | 1.8 | virginica |

Caso queira ter uma visão completa dos dados, a melhor forma de fazê-la é utilizando a função `view()`. Teste o seguinte comando:

```
view(iris)
```

Também podemos criar uma *tibble* dos zero, a partir de vetores individuais com a função `tibble()`.

```
tibble(
  nomes = c("José", "João", "Maria", "Ana"),
  sexo = c("M", "M", "F", "F"),
  idade = 21:24,
  idade_ao_quadrado = idade ^ 2,
  filhos = 0)
```

```
# A tibble: 4 x 5
```

| | nomes | sexo | idade | idade_ao_quadrado | filhos |
|---|-------|-------|-------|-------------------|--------|
| | <chr> | <chr> | <int> | <dbl> | <dbl> |
| 1 | José | M | 21 | 441 | 0 |
| 2 | João | M | 22 | 484 | 0 |
| 3 | Maria | F | 23 | 529 | 0 |
| 4 | Ana | F | 24 | 576 | 0 |

No exemplo acima, veja que foi possível criar a coluna `idade_ao_quadrado` em função de outra coluna, a `idade`. Na coluna `filhos`, passamos um vetor de tamanho 1, ocorrendo o processo de reciclagem de vetor, ou seja, foi atribuído o valor 0 para todas as demais observações até que se igualasse o número de linhas da *tibble*.

É possível colocar nomes de colunas com caracteres de sintaxe inválida (tema tratado na seção 3.5), desde que os nomes estejam entre acentos graves (`). No caso dos *data frames*, teríamos dificuldades em trabalhar com esse tipo de sintaxe, pois os nomes seriam convertidos para um formato que se enquadre na sintaxe válida, portanto, ocorreria uma alteração dos nomes designados originalmente.

A seguir, criaremos um exemplo de *tibble* (puramente a título de demonstração) com nomes que contêm espaços, começam com números e possuem caracteres especiais.

```
tibble(
  `nomes dos estudantes` = c("José", "João", "Maria", "Ana"),
  `2sexo` = c("M", "M", "F", "F"),
  `:)idade` = 21:24,
  `idade^2` = `:)idade` ^ 2,
  `nº filhos` = 0)
```

```
# A tibble: 4 x 5
  `nomes dos estudantes` `2sexo` `:)idade` idade^2 `nº filhos`
  <chr>                 <chr>    <int>    <dbl>      <dbl>
1 José                  M        21      441        0
2 João                  M        22      484        0
3 Maria                  F        23      529        0
4 Ana                    F        24      576        0
```

Por consequência, quando trabalharmos com as variáveis nomeadas dessa forma, precisaremos colocá-las sempre entre acentos graves, como foi o caso da construção da coluna `idade^2`, em que tivemos que colocar a coluna `:)idade` entre acentos graves para que conseguíssemos elevar seus valores ao quadrado.

Por último, podemos criar *tibbles* com a função `tribble()`. A `tribble()` é construída definindo os nomes das colunas por fórmulas (começando com ~), cujos valores são separados por vírgulas, sendo uma forma mais visual e intuitiva para construir pequenas *tibbles*.

```
tribble(
  ~nome, ~id, ~sexo,
  #-----/---/-----
  "João", 25, "M",
  "José", 30, "M",
  "Ana", 23, "F"
)
```

```
# A tibble: 3 x 3
  nome      id sexo
  <chr> <dbl> <chr>
1 João     25 M
2 José     30 M
3 Ana      23 F
```

Veja que é possível até mesmo adicionar um comentário (#) para criar uma delimitação entre o cabeçalho e os valores, tornando o código ainda mais visual.

5.1.2 Outras funções

Podemos adicionar novas linhas e colunas à *tibble* com as funções `add_row()` e `add_column()`, respectivamente.

```
tib <- tibble(
  nomes = c("José", "João", "Maria"),
  sexo = c("M", "M", "F")
)

tib_col <- add_column(tib, idade = c(26, 30, 19))

tib_col
```

```
# A tibble: 3 x 3
  nomes sexo idade
  <chr> <chr> <dbl>
1 José M      26
2 João M      30
3 Maria F      19
```

```
tib_row <- add_row(tib_col,
  nomes = c("Ana", "Beatriz"),
  sexo = "F",
  idade = c(20, 23))

tib_row
```

```
# A tibble: 5 x 3
  nomes  sexo idade
  <chr>  <chr> <dbl>
1 José   M     26
2 João   M     30
3 Maria  F     19
4 Ana    F     20
5 Beatriz F     23
```

5.2 Pacote `tidyr`

Nesta seção, abordaremos as principais ferramentas de organização de dados presentes no pacote `tidyr`. Portanto, precisamos rodá-lo no R.

```
library(tidyr)
```

A seguir, aplicaremos as ferramentas do `tidyr` em uma mesma base de dados, porém em diversas versões desarrumadas ¹. Os dados são referentes a produção (em toneladas) e a área colhida (em hectares) da cultura do milho, no Brasil, China e Índia, nos anos de 2000 e 2019. Para fazer o *download* da pasta contendo as bases, [clique aqui](#).

5.2.1 Base de dados *tidy*

Antes de abordarmos as bases desarrumadas, vamos tratar da base de dados arrumada.

Para alcançarmos a tão desejada base de dados arrumada, devemos ter em mente as três principais propriedades de uma *tidy data*:

- Cada variável possui sua própria coluna;
- Cada observação possui sua própria linha;
- Cada célula contém somente um único valor.

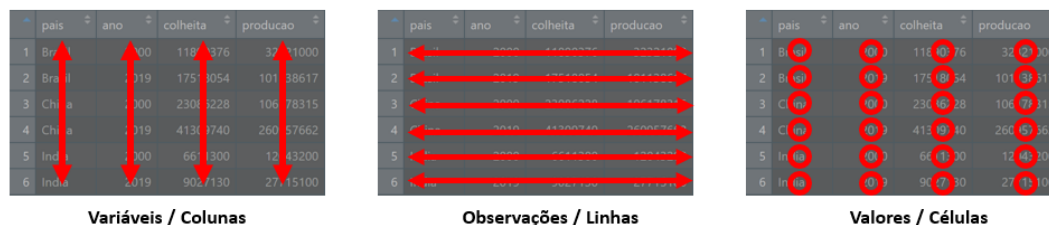


Figura 5.1: Seguindo os princípios da tidy data, cada variável possui uma coluna, cada observação está em uma linha e cada célula contém somente um valor.

Como citamos em capítulos anteriores, cada coluna de um *data frame* (ou *tibble*) é um vetor. Designando uma variável à uma única coluna, significa que podemos trabalhar com as informações a partir de vetores individualizados. Portanto, os conceitos vistos na seção 3.9, referente aos vetores, são aplicáveis a cada uma das colunas de uma base de dados organizada. É dessa maneira que os pacotes do `tidyverse` trabalham.

```
tidy <- read_csv("dados_tidy/tidy.csv")
```

¹O termo *desarrumada* não é o mais apropriado para nos referirmos aos demais formatos de dados, pois o formato *tidy* é um dos possíveis para se trabalhar com dados, principalmente quando trabalhamos com dados retangulares, utilizando o pacote `tidyverse`. Portanto, quando nos referirmos a dados *desarrumados* nessa apostila, entenda como dados fora do padrão *tidy data* ou *não-tidy*. Para entender mais sobre os dados *não-tidy*, recomendo o post do [Jeff Leek](#) sobre o assunto.


```
tidy
```

```
# A tibble: 6 x 4
  pais      ano colheita producao
<chr> <dbl>   <dbl>   <dbl>
1 Brasil  2000 11890376 32321000
2 Brasil  2019 17518054 101138617
3 China   2000 23086228 106178315
4 China   2019 41309740 260957662
5 India   2000  6611300 12043200
6 India   2019  9027130 27715100
```

Portanto, esse é um exemplo de uma base de dados organizada. Cada coluna é uma variável, cada observação está em uma linha e cada célula contém um único valor. Assim, sempre que se deparar com uma nova base de dados, observe primeiro quais elementos são variáveis e quais são observações, e se uma célula contém um ou mais valores.

Nos próximos capítulos, veremos como utilizar uma *tidy data* para realizar transformações e gráficos. Mas antes, temos que tratar dos dados desarrumados.

5.2.2 Pivotagem

Uma forma de organizarmos os nossos dados é realizando a pivotagem. Este método converte as observações que estão como nome de colunas, para linhas; e variáveis que estão em linhas, para as colunas.

Pivot longer

A função `pivot_longer()` converte as observações que estão como nome das colunas, para linhas. Aplicaremos sua função no arquivo `tidy1a_prod.csv`.

```
plonger1 <- read_csv("dados_tidy/tidy1a_prod.csv")

plonger1
```

```
# A tibble: 3 x 3
  pais      `2000`      `2019`
<chr>   <dbl>   <dbl>
1 Brasil 32321000 101138617
2 China 106178315 260957662
3 India 12043200 27715100
```

Veja que as observações referentes aos anos (2000 e 2019) estão como nome das colunas, cada qual com os valores referentes a produção de milho nos respectivos anos. Para arrumá-las, devemos criar duas novas colunas, uma para alocar os anos e outra para os valores de produção de milho.

```
plonger1 <- plonger1 %>%
  pivot_longer(col = c(`2000`, `2019`),
               names_to = "ano",
               values_to = "producao")
plonger1
```

```
# A tibble: 6 x 3
  pais   ano   producao
<chr> <chr>   <dbl>
1 Brasil 2000   32321000
2 Brasil 2019   101138617
3 China  2000   106178315
4 China  2019   260957662
5 India  2000    12043200
6 India  2019    27715100
```

Assim, na função `pivot_longer()`, utilizamos o argumento `col` para selecionar as colunas que desejamos pivotar. Já no argumento `names_to` devemos dizer para qual coluna os nomes selecionados devem ir (no caso, os nomes 2000 e 2019), portanto, são direcionados para a nova coluna `ano`. Por fim, no argumento `values_to`, dizemos que os valores contidos nas colunas 2000 e 2019 devem ir para uma única nova coluna `producao`.

Perceba que os valores se mantêm associados às antigas colunas 2000 e 2019, agora como valores de uma observação.

Podemos fazer o mesmo com os dados referentes à área colhida - presente no arquivo `tidy1b_col.csv` - somente alterando o nome do argumento `values_to` para `colheita`:

```
plonger2 <- read_csv("dados_tidy/tidy1b_col.csv")
plonger2
```

```
# A tibble: 3 x 3
  pais   `2000` `2019`
<chr>   <dbl>   <dbl>
1 Brasil 11890376 17518054
2 China  23086228 41309740
3 India   6611300  9027130
```

```
plonger2 <- plonger2 %>%
  pivot_longer(col = c(`2000`, `2019`),
               names_to = "ano",
               values_to = "colheita")
plonger2
```

```
# A tibble: 6 x 3
  pais   ano   colheita
<chr> <chr>   <dbl>
1 Brasil 2000   11890376
2 Brasil 2019   17518054
```

```
3 China 2000 23086228
4 China 2019 41309740
5 India 2000 6611300
6 India 2019 9027130
```

Para juntar ambas as tabelas, utilizamos a `dplyr::left_join()`, presente no pacote `dplyr`, a qual veremos com mais detalhes no capítulo 6.

```
plonger <- left_join(plonger1, plonger2)
plonger
```

```
# A tibble: 6 x 4
  pais   ano   producao colheita
  <chr> <chr>   <dbl>   <dbl>
1 Brasil 2000   32321000 11890376
2 Brasil 2019  101138617 17518054
3 China 2000  106178315 23086228
4 China 2019  260957662 41309740
5 India 2000   12043200 6611300
6 India 2019   27715100 9027130
```

Pivot wider

A função `pivot_wider()` é a oposta da `pivot_longer()`. Ela é usada para converter as variáveis que estão nas linhas, para o nome das colunas. Como exemplo, utilizaremos o arquivo `tidy2.csv`.

```
pwider <- read_csv(file = "dados_tidy/tidy2.csv")
pwider
```

```
# A tibble: 12 x 4
  pais   ano tipo      valor
  <chr> <dbl> <chr>   <dbl>
1 Brasil 2000 colheita 11890376
2 Brasil 2019 colheita 17518054
3 Brasil 2000 producao 32321000
4 Brasil 2019 producao 101138617
5 China 2000 colheita 23086228
6 China 2019 colheita 41309740
7 China 2000 producao 106178315
8 China 2019 producao 260957662
9 India 2000 colheita 6611300
10 India 2019 colheita 9027130
11 India 2000 producao 12043200
12 India 2019 producao 27715100
```

Como podemos observar, as variáveis `colheita` e `producao` estão como valores de observações. Portanto, devemos transformá-las em nomes de colunas, recebendo os respectivos valores associados à coluna `valor`.

```

pwider %>%
  pivot_wider(names_from = tipo,
              values_from = valor)

```

```

# A tibble: 6 x 4
  pais      ano colheita producao
<chr>  <dbl>    <dbl>    <dbl>
1 Brasil  2000 11890376 32321000
2 Brasil  2019 17518054 101138617
3 China   2000 23086228 106178315
4 China   2019 41309740 260957662
5 India   2000  6611300 12043200
6 India   2019  9027130 27715100

```

Assim, na função `pivot_wider()`, utilizamos o argumento `names_from` para dizer em qual coluna (`tipo`) estão os nomes das novas variáveis (`producao` e `colheita`) e o `values_from` para indicar em qual coluna estão localizados os respectivos valores das novas colunas criadas.

Portanto, podemos dizer que a `pivot_longer()` torna a base de dados mais longa (reduz o número de colunas e aumenta o número de linhas) e a `pivot_wider()`, deixa mais larga (aumenta o número de colunas e diminui o número de linhas).

Para conferir todos os argumentos das funções `pivot_longer()` e `pivot_wider()`, utilize as funções `args(pivot_longer)` e `args(pivot_wider)`.

5.2.3 Separar e Unir

Para tratar das funções de separar e unir, exemplificaremos com o formato de dados do arquivo `tidy3.csv`.

```

sep_unir <- read_csv(file = "dados_tidy/tidy3.csv")

sep_unir

```

```

# A tibble: 6 x 3
  pais      ano produtividade
<chr>  <dbl> <chr>
1 Brasil  2000 32321000/11890376
2 Brasil  2019 101138617/17518054
3 China   2000 106178315/23086228
4 China   2019 260957662/41309740
5 India   2000 12043200/6611300
6 India   2019 27715100/9027130

```

Separar

Em relação aos dados organizados, essa base apresenta a coluna `produtividade`, cujos valores são `produção/área colhida`. Portanto, precisamos separá-las em duas colunas, pois temos duas variáveis em uma mesma coluna e dois valores em uma mesma célula. Para isso utilizaremos a função `separate()`.

```
sep_unir %>%
  separate(col = produtividade,
           into = c("producao", "colheita"))
```

```
# A tibble: 6 x 4
  pais      ano producao colheita
<chr> <dbl> <chr>      <chr>
1 Brasil  2000 32321000 11890376
2 Brasil  2019 101138617 17518054
3 China   2000 106178315 23086228
4 China   2019 260957662 41309740
5 India   2000 12043200 6611300
6 India   2019 27715100 9027130
```

Na função `separate()`, indicamos o nome da coluna a ser separada - no caso, a `produtividade` -, e com o argumento `into`, dizemos o nome das novas colunas que direcionaremos os valores da coluna separada.

Por padrão, a função `separate()` irá quebrar os valores quando perceber que entre eles há um operador não numérico ou não textual, como foi o caso do exemplo acima, que apresentava uma `/` separando os valores. Caso for necessário especificar qual o separador utilizado, devemos utilizar o argumento `sep`.

```
sep_unir %>%
  separate(col = produtividade,
           into = c("producao", "colheita"),
           sep = "/")
```

```
# A tibble: 6 x 4
  pais      ano producao colheita
<chr> <dbl> <chr>      <chr>
1 Brasil  2000 32321000 11890376
2 Brasil  2019 101138617 17518054
3 China   2000 106178315 23086228
4 China   2019 260957662 41309740
5 India   2000 12043200 6611300
6 India   2019 27715100 9027130
```

Perceba que a classe das novas colunas vieram como tipo caractere. Por padrão, a função `separate()` sempre adotará essa classe. Para consertarmos a classe, podemos usar o argumento `convert = TRUE`, ou seja, pedir para função adotar uma classe mais apropriada para os valores contidos nas colunas.

```
sep_unir %>%
  separate(col = produtividade,
           into = c("producao", "colheita"),
           convert = TRUE)
```

```
# A tibble: 6 x 4
  pais      ano producao colheita
<chr> <dbl>    <int>    <int>
1 Brasil  2000 32321000 11890376
```

```
2 Brasil 2019 101138617 17518054
3 China 2000 106178315 23086228
4 China 2019 260957662 41309740
5 India 2000 12043200 6611300
6 India 2019 27715100 9027130
```

Feito isso, temos a classe `int` (números inteiros) para as colunas `producao` e `colheita`.

Também podemos separar valores informando uma quantidade de caracteres a serem considerados na separação. Para isso, informamos ao argumento `sep` a quantidade dos primeiros caracteres que desejamos quebrar. Por exemplo, para separar os valores da coluna `ano` em `seculo` e `decada`, podemos passar o argumento `sep = 2`.

```
separar <- sep_unir %>%
  separate(col = ano,
           into = c("seculo", "decada"),
           sep = 2)

separar
```

```
# A tibble: 6 x 4
  pais   seculo decada produtividade
<chr> <chr> <chr> <chr>
1 Brasil 20    00    32321000/11890376
2 Brasil 20    19    101138617/17518054
3 China 20    00    106178315/23086228
4 China 20    19    260957662/41309740
5 India 20    00    12043200/6611300
6 India 20    19    27715100/9027130
```

Nesse caso, o argumento `sep = 2` pegou os dois primeiros número e os separou dos demais. Caso houver um número negativo, o argumento considera o sinal de negativo como o primeiro caractere informado no argumento.

Unir

A função `unite()` é a inversa de `separate()`. Portanto, ela combina múltiplas colunas em uma única. Usaremos a `unite()` para juntar novamente as colunas `seculo` e `decada` para apenas a coluna `novamente_ano`.

```
unir <- separar %>%
  unite(col = "novamente_ano",
        seculo, decada,
        sep = "")

unir
```

```
# A tibble: 6 x 3
  pais   novamente_ano produtividade
<chr> <chr>          <chr>
```

| | | | |
|---|--------|------|--------------------|
| 1 | Brasil | 2000 | 32321000/11890376 |
| 2 | Brasil | 2019 | 101138617/17518054 |
| 3 | China | 2000 | 106178315/23086228 |
| 4 | China | 2019 | 260957662/41309740 |
| 5 | India | 2000 | 12043200/6611300 |
| 6 | India | 2019 | 27715100/9027130 |

No argumento `col` dizemos qual o nome da nova coluna, em seguida, as colunas a serem juntadas (`seculo` e `decada`), e o `sep`, para informar qual o separador utilizado. Nesse caso, precisávamos juntar os valores sem qualquer caractere ou espaço, portanto, nosso argumento recebe apenas duas aspas `""`. Caso não fosse informado o argumento `sep`, por padrão, a função adota o *underline* (`_`) como separador.

Para conferir todos os argumentos das funções `separate()` e `unite()`, utilize as funções `args(separate)` e `args(unite)`.

De maneira geral, essas são as principais ferramentas quando o tema é arrumar os dados. A seguir, agora com os dados arrumados, iremos transformar as nossas bases de dados para que contenha apenas as variáveis de nosso interesse e também criar novas variáveis, de acordo com as existentes.

Capítulo 6

Transformação

A etapa de transformação dos dados consiste em selecionar as variáveis e observações de interesse no nosso banco de dados, a fim de gerar medidas úteis para a análise. Podemos realizar operações entre colunas de acordo com determinada variável, calcular a média, mediana, contagem e porcentagens, além de selecionar, filtrar e criar novas variáveis.

Os principais pacotes relacionados ao tema, presentes no **tidyverse**, são o **dplyr**, **stringr**, **forcats** e **lubridate**, cada qual apresentando funções particulares e específicas para trabalhar com os dados. Nessa apostila, abordaremos apenas o pacote **dplyr**, cujas funções conseguem resolver a maioria dos problemas relacionados a essa etapa.

Os demais pacotes tratam de assuntos específicos na transformação de dados. Caso você precise tratar de algum problema que o **dplyr** não consiga resolver, descreveremos, brevemente, as características desses pacotes.

- **stringr**: manipula as variáveis categóricas a partir de **expressões regulares**;
- **forcats**: apresenta funções que lidam com variáveis do tipo fator (*factor*). Caso queira entender melhor sobre esse tipo de classe, confira a seção [3.10](#);
- **lubridate**: pacote específico para trabalhar com variáveis do tipo data e tempo.

A seguir, trataremos com detalhes as funcionalidades presentes no pacote **dplyr**. Para tanto, precisamos rodar o pacote.

```
library(dplyr)
```

6.1 Pacote dplyr

O pacote **dplyr** possui ferramentas simples, porém muito importantes para realizar as devidas transformações na base de dados. A seguir, apresentaremos as principais funções do pacote, que nos permitem resolver a maioria dos problemas relacionados à etapa de transformação dos dados.

A base de dados utilizada para os exemplos é referente à produção de milho, soja, trigo e arroz, nos países da América do Sul, entre 1961 e 2019. Para fazer o *download* dos dados, [clique aqui](#).

```
graos <- read_csv("dados_transf/prod_graos.csv")
```

```
graos
```

```
# A tibble: 5,510 x 14
```

```
  `Domain Code` Domain      `Area Code (FAO~ Area  `Element Code` Element
    <chr>          <chr>          <dbl> <chr>          <dbl> <chr>
1 QCL            Crops and liv~      9 Argen~      5312 Area har~
2 QCL            Crops and liv~      9 Argen~      5312 Area har~
3 QCL            Crops and liv~      9 Argen~      5312 Area har~
4 QCL            Crops and liv~      9 Argen~      5312 Area har~
5 QCL            Crops and liv~      9 Argen~      5312 Area har~
6 QCL            Crops and liv~      9 Argen~      5312 Area har~
7 QCL            Crops and liv~      9 Argen~      5312 Area har~
8 QCL            Crops and liv~      9 Argen~      5312 Area har~
9 QCL            Crops and liv~      9 Argen~      5312 Area har~
10 QCL           Crops and liv~      9 Argen~      5312 Area har~
```

```
# ... with 5,500 more rows, and 8 more variables: Item Code (FAO) <dbl>,
```

```
#   Item <chr>, Year Code <dbl>, Year <dbl>, Unit <chr>, Value <dbl>,
```

```
#   Flag <lgl>, Flag Description <chr>
```

O banco de dados possui 5510 observações e 14 variáveis. Como podemos observar, muitas das variáveis são referentes à códigos de identificação, os quais não nos interessam para a realização das análises. Assim, a seguir, veremos como selecionar somente as variáveis de interesse.

6.1.1 Selecionar

Para selecionar colunas, utilizamos a função `select()`, tendo como primeiro argumento a base de dados utilizada, sendo os demais argumentos referentes aos nomes das colunas que se deseja selecionar.

Nos dois exemplos a seguir, perceba que podemos selecionar uma ou mais de uma coluna.

```
# Selecionando somente a coluna "Item"
```

```
select(graos,
       Item)
```

```
# A tibble: 5,510 x 1
```

```
  Item
  <chr>
1 Maize
2 Maize
3 Maize
4 Maize
5 Maize
6 Maize
7 Maize
8 Maize
9 Maize
10 Maize
```

```
# ... with 5,500 more rows
```

```
# Selecionando mais de uma coluna
select(graos,
       Area, Item, Value)
```

```
# A tibble: 5,510 x 3
  Area      Item      Value
  <chr>    <chr>    <dbl>
1 Argentina Maize 2744400
2 Argentina Maize 2756670
3 Argentina Maize 2645400
4 Argentina Maize 2970500
5 Argentina Maize 3062300
6 Argentina Maize 3274500
7 Argentina Maize 3450500
8 Argentina Maize 3377700
9 Argentina Maize 3556000
10 Argentina Maize 4017330
# ... with 5,500 more rows
```

Podemos selecionar várias colunas consecutivas com o operador `:`. Basta informar o nome ou a posição da primeira e da última coluna que se deseja selecionar.

```
# Selecionando colunas consecutivas a partir dos nomes
select(graos,
       Area:Year)
```

```
# A tibble: 5,510 x 7
  Area      `Element Code` Element      `Item Code (FAO~ Item      `Year Code`   Year
  <chr>          <dbl> <chr>          <dbl> <chr>          <dbl> <dbl>
1 Argenti~      5312 Area harves~      56 Maize      1961 1961
2 Argenti~      5312 Area harves~      56 Maize      1962 1962
3 Argenti~      5312 Area harves~      56 Maize      1963 1963
4 Argenti~      5312 Area harves~      56 Maize      1964 1964
5 Argenti~      5312 Area harves~      56 Maize      1965 1965
6 Argenti~      5312 Area harves~      56 Maize      1966 1966
7 Argenti~      5312 Area harves~      56 Maize      1967 1967
8 Argenti~      5312 Area harves~      56 Maize      1968 1968
9 Argenti~      5312 Area harves~      56 Maize      1969 1969
10 Argenti~      5312 Area harves~      56 Maize      1970 1970
# ... with 5,500 more rows
```

```
# Selecionando colunas consecutivas a partir das posições
select(graos, 4:10)
```

```
# A tibble: 5,510 x 7
  Area      `Element Code` Element      `Item Code (FAO~ Item      `Year Code`   Year
  <chr>          <dbl> <chr>          <dbl> <chr>          <dbl> <dbl>
1 Argenti~      5312 Area harves~      56 Maize      1961 1961
2 Argenti~      5312 Area harves~      56 Maize      1962 1962
3 Argenti~      5312 Area harves~      56 Maize      1963 1963
```

```

4 Argenti~      5312 Area harves~      56 Maize      1964 1964
5 Argenti~      5312 Area harves~      56 Maize      1965 1965
6 Argenti~      5312 Area harves~      56 Maize      1966 1966
7 Argenti~      5312 Area harves~      56 Maize      1967 1967
8 Argenti~      5312 Area harves~      56 Maize      1968 1968
9 Argenti~      5312 Area harves~      56 Maize      1969 1969
10 Argenti~     5312 Area harves~      56 Maize      1970 1970
# ... with 5,500 more rows

```

A função `select()` possui outras funções que auxiliam na seleção de colunas, sendo elas:

- `starts_with()`: seleciona colunas que começam com um texto padrão;
- `ends_with()`: seleciona colunas que terminam com um texto padrão;
- `contains()`: seleciona colunas que possuem um texto padrão.

```

# starts_with() - Começa com tal palavra
select(graos, starts_with("Year"))

```

```

# A tibble: 5,510 x 2
  `Year Code`  Year
    <dbl> <dbl>
1      1961  1961
2      1962  1962
3      1963  1963
4      1964  1964
5      1965  1965
6      1966  1966
7      1967  1967
8      1968  1968
9      1969  1969
10     1970  1970
# ... with 5,500 more rows

```

```

# ends_with() - Termina com tal palavra
select(graos, ends_with("Code"))

```

```

# A tibble: 5,510 x 3
  `Domain Code` `Element Code` `Year Code`
    <chr>          <dbl>      <dbl>
1 QCL             5312        1961
2 QCL             5312        1962
3 QCL             5312        1963
4 QCL             5312        1964
5 QCL             5312        1965
6 QCL             5312        1966
7 QCL             5312        1967
8 QCL             5312        1968
9 QCL             5312        1969
10 QCL            5312        1970
# ... with 5,500 more rows

```

```
# contains() - Contém tal palavra
select(graos, contains("FAO"))
```

```
# A tibble: 5,510 x 2
  `Area Code (FAO)` `Item Code (FAO)`
      <dbl>         <dbl>
1         9         56
2         9         56
3         9         56
4         9         56
5         9         56
6         9         56
7         9         56
8         9         56
9         9         56
10        9         56
# ... with 5,500 more rows
```

Também podemos retirar uma coluna, inserindo um sinal de menos (-) antes do nome da variável ou das funções auxiliares.

```
select(graos,
  -contains("Code"), -contains("Flag"), -Domain)
```

```
# A tibble: 5,510 x 6
  Area      Element      Item  Year Unit      Value
  <chr>    <chr>      <chr> <dbl> <chr>    <dbl>
1 Argentina Area harvested Maize  1961 ha    2744400
2 Argentina Area harvested Maize  1962 ha    2756670
3 Argentina Area harvested Maize  1963 ha    2645400
4 Argentina Area harvested Maize  1964 ha    2970500
5 Argentina Area harvested Maize  1965 ha    3062300
6 Argentina Area harvested Maize  1966 ha    3274500
7 Argentina Area harvested Maize  1967 ha    3450500
8 Argentina Area harvested Maize  1968 ha    3377700
9 Argentina Area harvested Maize  1969 ha    3556000
10 Argentina Area harvested Maize  1970 ha    4017330
# ... with 5,500 more rows
```

Por último, temos a função `everything()`, utilizada na função `select()` para arrastar determinadas colunas para o início da base de dados.

```
select(graos,
  Value, Unit, everything())
```

```
# A tibble: 5,510 x 14
  Value Unit `Domain Code` Domain      `Area Code (FAO~ Area  `Element Code`
  <dbl> <chr> <chr>      <chr>         <dbl> <chr>      <dbl>
1 2744400 ha    QCL        Crops and ~          9 Arge~      5312
```

```

 2 2756670 ha    QCL      Crops and ~      9 Arge~      5312
 3 2645400 ha    QCL      Crops and ~      9 Arge~      5312
 4 2970500 ha    QCL      Crops and ~      9 Arge~      5312
 5 3062300 ha    QCL      Crops and ~      9 Arge~      5312
 6 3274500 ha    QCL      Crops and ~      9 Arge~      5312
 7 3450500 ha    QCL      Crops and ~      9 Arge~      5312
 8 3377700 ha    QCL      Crops and ~      9 Arge~      5312
 9 3556000 ha    QCL      Crops and ~      9 Arge~      5312
10 4017330 ha    QCL      Crops and ~      9 Arge~      5312
# ... with 5,500 more rows, and 7 more variables: Element <chr>,
#   Item Code (FAO) <dbl>, Item <chr>, Year Code <dbl>, Year <dbl>, Flag <lgl>,
#   Flag Description <chr>

```

6.1.2 Operador *pipe* (%>%)

Na maior parte dos casos, utilizaremos mais de uma função para manipular os nossos dados. Com isso, entra em cena o *pipe* (%>%). O *pipe* está presente no pacote *magrittr*, que está contido no *tidyverse*. Portanto, antes de aprofundarmos na ideia central do *pipe*, devemos carregar o pacote *magrittr*.

```
library(magrittr)
```

A principal função do *pipe* é conectar linhas de códigos que se relacionam, executando-as em sequência, de uma só vez. A essa estrutura de código chamamos de *pipelines*. Como exemplo hipotético, calcularemos a média final de um aluno na disciplina de cálculo II, arredondando-a com uma casa decimal.

```
notas <- c(8.88, 6.84, 7.51)
```

```
# Sem pipe
round(mean(notas),1)
```

```
[1] 7.7
```

```
# Com pipe
notas %>% mean() %>% round(1)
```

```
[1] 7.7
```

Utilizando o *pipe*, evitamos de escrever funções dentro de funções, ordenando-as de acordo com a ordem em que desejamos realizar as operações. No exemplo, calculamos primeiro a média das notas e, posteriormente, arredondamos.

Quando utilizamos o *pipe*, obtemos um código mais legível, claro e compacto, principalmente quando trabalhamos com diversas funções. Isso facilita não somente a leitura, mas também na manutenção do código, caso seja preciso realizar alterações ou consertar possíveis problemas.

Tendo essa noção básica do que é o *pipe*, começaremos a aplicá-lo na manipulação dos dados. Caso queira saber mais sobre o *pipe*, confira o capítulo 18 do livro *R for Data Science*.

6.1.3 Filtrar

Podemos filtrar determinados valores que estão contidos nas colunas, sejam eles quantitativos ou categóricos. Para isso, utilizamos testes lógicos dentro da função `filter()`.

```
filter(graos,
       Area == "Chile")
```

```
# A tibble: 459 x 14
  `Domain Code` Domain      `Area Code (FAO~ Area  `Element Code` Element
  <chr>           <chr>           <dbl> <chr>           <dbl> <chr>
1 QCL            Crops and live~      40 Chile          5312 Area har~
2 QCL            Crops and live~      40 Chile          5312 Area har~
3 QCL            Crops and live~      40 Chile          5312 Area har~
4 QCL            Crops and live~      40 Chile          5312 Area har~
5 QCL            Crops and live~      40 Chile          5312 Area har~
6 QCL            Crops and live~      40 Chile          5312 Area har~
7 QCL            Crops and live~      40 Chile          5312 Area har~
8 QCL            Crops and live~      40 Chile          5312 Area har~
9 QCL            Crops and live~      40 Chile          5312 Area har~
10 QCL           Crops and live~      40 Chile          5312 Area har~
# ... with 449 more rows, and 8 more variables: Item Code (FAO) <dbl>,
#   Item <chr>, Year Code <dbl>, Year <dbl>, Unit <chr>, Value <dbl>,
#   Flag <lgl>, Flag Description <chr>
```

No exemplo anterior, filtramos a coluna `Area` para que nos retornasse somente as observações referentes ao país `Chile`. Para isso, utilizamos o teste lógico `==`, ou seja, pedimos para que nos retornasse somente as observações que apresentem o valor igual a `Chile` na coluna `Area`.

Lembre que a função `filter()` segue a lógica demonstrada na seção 3.11. Mas perceba que, diferentemente do que fora exposto na referente seção, a `filter()` é muito mais simples e intuitiva de ser utilizada.

Também podemos selecionar um conjunto de valores contidos em uma coluna. Para isso, criamos um vetor com os valores desejados e aplicamos o teste `%in%`, ou seja, dentro da coluna `Area`, pedimos para que nos retorne somente os valores que estão contidos no vetor.

```
filter(graos,
       Area %in% c("Brazil", "Argentina", "Chile"))
```

```
# A tibble: 1,403 x 14
  `Domain Code` Domain      `Area Code (FAO~ Area  `Element Code` Element
  <chr>           <chr>           <dbl> <chr>           <dbl> <chr>
1 QCL            Crops and liv~      9 Argen~          5312 Area har~
2 QCL            Crops and liv~      9 Argen~          5312 Area har~
3 QCL            Crops and liv~      9 Argen~          5312 Area har~
4 QCL            Crops and liv~      9 Argen~          5312 Area har~
5 QCL            Crops and liv~      9 Argen~          5312 Area har~
6 QCL            Crops and liv~      9 Argen~          5312 Area har~
7 QCL            Crops and liv~      9 Argen~          5312 Area har~
8 QCL            Crops and liv~      9 Argen~          5312 Area har~
9 QCL            Crops and liv~      9 Argen~          5312 Area har~
```

```

10 QCL           Crops and liv~           9 Argen~           5312 Area har~
# ... with 1,393 more rows, and 8 more variables: Item Code (FAO) <dbl>,
#   Item <chr>, Year Code <dbl>, Year <dbl>, Unit <chr>, Value <dbl>,
#   Flag <lgl>, Flag Description <chr>

```

Por outro lado, podemos retirar valores com o operador `!`.

```

filter(graos,
       !(Area %in% c("Brazil", "Argentina", "Chile")))

```

```

# A tibble: 4,107 x 14
  `Domain Code` Domain      `Area Code (FAO~ Area      `Element Code` Element
  <chr>           <chr>           <dbl> <chr>           <dbl> <chr>
1 QCL           Crops and ~           19 Bolivia (~       5312 Area ha~
2 QCL           Crops and ~           19 Bolivia (~       5312 Area ha~
3 QCL           Crops and ~           19 Bolivia (~       5312 Area ha~
4 QCL           Crops and ~           19 Bolivia (~       5312 Area ha~
5 QCL           Crops and ~           19 Bolivia (~       5312 Area ha~
6 QCL           Crops and ~           19 Bolivia (~       5312 Area ha~
7 QCL           Crops and ~           19 Bolivia (~       5312 Area ha~
8 QCL           Crops and ~           19 Bolivia (~       5312 Area ha~
9 QCL           Crops and ~           19 Bolivia (~       5312 Area ha~
10 QCL          Crops and ~           19 Bolivia (~       5312 Area ha~
# ... with 4,097 more rows, and 8 more variables: Item Code (FAO) <dbl>,
#   Item <chr>, Year Code <dbl>, Year <dbl>, Unit <chr>, Value <dbl>,
#   Flag <lgl>, Flag Description <chr>

```

Da mesma forma, podemos aplicar os filtros para variáveis quantitativas. Note no exemplo a seguir que aplicamos 3 filtros. O primeiro referente à variável categórica `Element`, sendo os outros dois, às variáveis quantitativas `Year` e `Value`.

```

filter(graos,
       Element == "Production", Year > 2010, Value > 10^7)

```

```

# A tibble: 54 x 14
  `Domain Code` Domain      `Area Code (FAO~ Area      `Element Code` Element
  <chr>           <chr>           <dbl> <chr>           <dbl> <chr>
1 QCL           Crops and live~           9 Argen~           5510 Product~
2 QCL           Crops and live~           9 Argen~           5510 Product~
3 QCL           Crops and live~           9 Argen~           5510 Product~
4 QCL           Crops and live~           9 Argen~           5510 Product~
5 QCL           Crops and live~           9 Argen~           5510 Product~
6 QCL           Crops and live~           9 Argen~           5510 Product~
7 QCL           Crops and live~           9 Argen~           5510 Product~
8 QCL           Crops and live~           9 Argen~           5510 Product~
9 QCL           Crops and live~           9 Argen~           5510 Product~
10 QCL          Crops and live~           9 Argen~           5510 Product~
# ... with 44 more rows, and 8 more variables: Item Code (FAO) <dbl>,
#   Item <chr>, Year Code <dbl>, Year <dbl>, Unit <chr>, Value <dbl>,
#   Flag <lgl>, Flag Description <chr>

```


Para melhorar a organização e a apresentação da base de dados, podemos aplicar as funções `filter()` e `select()` juntas. Para tanto, utilizaremos o *pipe* para mesclar ambas as funções em uma *pipeline*.

```
graos %>%
  filter(Element == "Production",
         Area %in% c("Brazil", "Argentina"),
         Year > 2010) %>%
  select(Area, Element, Item, Year, Unit, Value)
```

```
# A tibble: 72 x 6
   Area      Element Item      Year Unit      Value
  <chr>    <chr>    <chr>    <dbl> <chr>    <dbl>
1 Argentina Production Maize    2011 tonnes 23799830
2 Argentina Production Maize    2012 tonnes 21196637
3 Argentina Production Maize    2013 tonnes 32119211
4 Argentina Production Maize    2014 tonnes 33087165
5 Argentina Production Maize    2015 tonnes 33817744
6 Argentina Production Maize    2016 tonnes 39792854
7 Argentina Production Maize    2017 tonnes 49475895
8 Argentina Production Maize    2018 tonnes 43462323
9 Argentina Production Maize    2019 tonnes 56860704
10 Argentina Production Rice, paddy 2011 tonnes 1748075
# ... with 62 more rows
```

Perceba que a aplicação do *pipe* é bem simples e intuitiva. Primeiramente, indicamos a base de dados a ser utilizada para realizar a filtragem e seleção - no caso, a base `graos`. Em seguida, escrevemos o `%>%` para conectar o banco de dados com a função `filter()`; nesse caso, filtramos apenas os valores iguais a "Production" na coluna `Element`, os países `Brazil` e `Argentina` na variável `Area` e os anos maiores que 2010. Novamente, escrevemos o `%>%` para aplicar a `select()` e selecionar as variáveis desejadas.

Note que não foi preciso indicar, como primeiro argumento das funções, qual a base de dados utilizada, pois essa foi especificada na primeira parte do *pipeline*. Além disso, a execução do código é realizada na ordem em que são escritos os comandos. Assim, caso desejarmos filtrar uma coluna e, posteriormente, retirá-la da seleção, devemos nos atentar à ordem dos comandos.

```
graos %>%
  filter(Area == "Brazil",
         Element == "Production",
         Year %in% c(2015:2019)) %>%
  select(Item, Year, Value)
```

```
# A tibble: 20 x 3
   Item      Year      Value
  <chr>    <dbl>    <dbl>
1 Maize    2015 85283074
2 Maize    2016 64188314
3 Maize    2017 97910658
4 Maize    2018 82366531
5 Maize    2019 101138617
6 Rice, paddy 2015 12301201
7 Rice, paddy 2016 10622189
```

| | | | |
|----|-------------|------|-----------|
| 8 | Rice, paddy | 2017 | 12464766 |
| 9 | Rice, paddy | 2018 | 11808412 |
| 10 | Rice, paddy | 2019 | 10368611 |
| 11 | Soybeans | 2015 | 97464936 |
| 12 | Soybeans | 2016 | 96394820 |
| 13 | Soybeans | 2017 | 114732101 |
| 14 | Soybeans | 2018 | 117912450 |
| 15 | Soybeans | 2019 | 114269392 |
| 16 | Wheat | 2015 | 5508451 |
| 17 | Wheat | 2016 | 6834421 |
| 18 | Wheat | 2017 | 4342812 |
| 19 | Wheat | 2018 | 5469236 |
| 20 | Wheat | 2019 | 5604158 |

No exemplo acima, perceba que filtramos as colunas `Area` e `Element`, mas não as selecionamos posteriormente. Caso fosse realizada a seleção antes da filtragem, não seria possível filtrar as devidas variáveis, uma vez que não selecionamos suas colunas para, posteriormente, serem filtradas.

6.1.4 Modificar e criar colunas

Para modificar ou criar novas colunas, utilizamos a função `mutate()`. No exemplo a seguir, transformaremos os valores de produção, em toneladas, para quilogramas.

```
graos %>%
  filter(Element == "Production") %>%
  select(Area, Element, Item, Year, Value) %>%
  mutate(Value = Value*1000)
```

```
# A tibble: 2,756 x 5
   Area      Element Item   Year   Value
  <chr>    <chr>    <chr> <dbl>   <dbl>
1 Argentina Production Maize  1961 4850000000
2 Argentina Production Maize  1962 5220000000
3 Argentina Production Maize  1963 4360000000
4 Argentina Production Maize  1964 5350000000
5 Argentina Production Maize  1965 5140000000
6 Argentina Production Maize  1966 7040000000
7 Argentina Production Maize  1967 8510000000
8 Argentina Production Maize  1968 6560000000
9 Argentina Production Maize  1969 6860000000
10 Argentina Production Maize  1970 9360000000
# ... with 2,746 more rows
```

Anteriormente, transformamos os valores da coluna `Value`, mas também podemos manter a coluna original e criar uma nova coluna com a variável calculada. Basta designar um novo nome à coluna, nesse caso, criamos a `Value(kg)`.

```
graos %>%
  filter(Element == "Production") %>%
  select(Area, Element, Item, Year, Value) %>%
  mutate(`Value(kg)` = Value*1000)
```

```
# A tibble: 2,756 x 6
  Area      Element Item  Year  Value `Value(kg)`
  <chr>    <chr>    <chr> <dbl>  <dbl>      <dbl>
1 Argentina Production Maize  1961 4850000 4850000000
2 Argentina Production Maize  1962 5220000 5220000000
3 Argentina Production Maize  1963 4360000 4360000000
4 Argentina Production Maize  1964 5350000 5350000000
5 Argentina Production Maize  1965 5140000 5140000000
6 Argentina Production Maize  1966 7040000 7040000000
7 Argentina Production Maize  1967 8510000 8510000000
8 Argentina Production Maize  1968 6560000 6560000000
9 Argentina Production Maize  1969 6860000 6860000000
10 Argentina Production Maize  1970 9360000 9360000000
# ... with 2,746 more rows
```

Podemos realizar qualquer operação com a quantidade de colunas que desejarmos. Porém, deve ser retornado um vetor com comprimento igual à quantidade de linhas da base de dados ou com comprimento igual a 1, sendo assim realizado o processo de reciclagem do valor.

```
graos %>%
  filter(Element == "Production") %>%
  select(Area, Element, Item, Year, Value) %>%
  mutate(Value = Value*1000,
         Unit = "kg")
```

```
# A tibble: 2,756 x 6
  Area      Element Item  Year  Value Unit
  <chr>    <chr>    <chr> <dbl>  <dbl> <chr>
1 Argentina Production Maize  1961 4850000000 kg
2 Argentina Production Maize  1962 5220000000 kg
3 Argentina Production Maize  1963 4360000000 kg
4 Argentina Production Maize  1964 5350000000 kg
5 Argentina Production Maize  1965 5140000000 kg
6 Argentina Production Maize  1966 7040000000 kg
7 Argentina Production Maize  1967 8510000000 kg
8 Argentina Production Maize  1968 6560000000 kg
9 Argentina Production Maize  1969 6860000000 kg
10 Argentina Production Maize  1970 9360000000 kg
# ... with 2,746 more rows
```

6.1.5 Resumo de valores

O processo de sumariar consiste em resumir um conjunto de dados a partir de uma medida de interesse. Como exemplo, podemos tirar a média, mediana, frequência e proporção dos valores desejados. Para isso, utilizamos a função `summarise()`. A seguir, faremos a média da produção de milho no Brasil.

```
graos %>%
  filter(Area == "Brazil",
         Element == "Production",
         Item == "Maize") %>%
```

```
select(Area, Item, Year, Value) %>%
  summarise(mean_prod_maize_br = mean(Value, na.rm = TRUE))
```

```
# A tibble: 1 x 1
  mean_prod_maize_br
      <dbl>
1      33776948.
```

Podemos calcular várias medidas diferentes na função `summarise()`.

```
graos %>%
  filter(Area == "Brazil",
         Element == "Production",
         Item == "Maize") %>%
  select(Area, Item, Year, Value) %>%
  summarise(mean_prod_maize_br = mean(Value),
            median_prod_maize_br = median(Value),
            variance_prod_maize_br = var(Value))
```

```
# A tibble: 1 x 3
  mean_prod_maize_br median_prod_maize_br variance_prod_maize_br
      <dbl>          <dbl>          <dbl>
1      33776948.      26589870      5.61e14
```

Há casos em que queremos sumariar uma coluna de acordo com alguma variável categórica de uma outra coluna. Para isso, utilizamos a função `group_by()` para indicar qual coluna desejamos agrupar para realizar a `summarise()`. No exemplo a seguir, agruparemos a variável `Element` para calcular a média da produção (`Production`) e da área colhida (`Area harvested`) de soja, na América do Sul.

```
graos %>%
  filter(Item == "Soybeans") %>%
  group_by(Element) %>%
  summarise(mean_value = mean(Value, na.rm = TRUE))
```

```
# A tibble: 2 x 2
  Element      mean_value
  <chr>      <dbl>
1 Area harvested  2132705.
2 Production     5335899.
```

Podemos agrupar mais de duas variáveis para sumariar. A seguir, agruparemos as colunas `Area` e `Element` para calcular, novamente, a média da produção e da área colhida, mas agora, por país sul-americano.

```
graos %>%
  filter(Item == "Soybeans") %>%
  group_by(Area, Element) %>%
  summarise(mean_value = mean(Value, na.rm = TRUE))
```

```
# A tibble: 26 x 3
# Groups:   Area [13]
  Area                Element    mean_value
  <chr>              <chr>         <dbl>
1 Argentina          Area harvested  6967484.
2 Argentina          Production    17941231.
3 Bolivia (Plurinational State of) Area harvested  472537.
4 Bolivia (Plurinational State of) Production      922200.
5 Brazil              Area harvested 12772243.
6 Brazil              Production    32075459.
7 Chile               Area harvested   984
8 Chile               Production     1043.
9 Colombia            Area harvested  45942.
10 Colombia           Production    93380.
# ... with 16 more rows
```

6.1.6 Ordenar

Podemos ordenar as linhas da base de dados de acordo com algum parâmetro referente aos valores de uma ou mais colunas. Para tanto, utilizamos a função `arrange()`.

```
graos %>%
  filter(Element == "Production") %>%
  select(Area, Item, Year, Value) %>%
  arrange(Value)
```

```
# A tibble: 2,756 x 4
  Area                Item    Year Value
  <chr>              <chr>  <dbl> <dbl>
1 Chile              Soybeans 1988     0
2 Chile              Soybeans 1990     0
3 Chile              Soybeans 1991     0
4 Chile              Soybeans 1992     0
5 French Guyana Maize 1990     0
6 French Guyana Soybeans 1990     0
7 French Guyana Soybeans 1991     0
8 French Guyana Soybeans 1992     0
9 Guyana              Soybeans 1990     0
10 Guyana             Soybeans 1991     0
# ... with 2,746 more rows
```

Por padrão, a função `arrange()` ordena os valores em ordem crescente. Para ordená-las em ordem decrescente, utilizamos a função `desc()` dentro da própria `arrange()`.

```
graos %>%
  filter(Element == "Production") %>%
  select(Area, Item, Year, Value) %>%
  arrange(desc(Value))
```

```
# A tibble: 2,756 x 4
```

```

  Area   Item      Year      Value
  <chr>  <chr>    <dbl>    <dbl>
1 Brazil Soybeans  2018 117912450
2 Brazil Soybeans  2017 114732101
3 Brazil Soybeans  2019 114269392
4 Brazil Maize     2019 101138617
5 Brazil Maize     2017  97910658
6 Brazil Soybeans  2015  97464936
7 Brazil Soybeans  2016  96394820
8 Brazil Soybeans  2014  86760520
9 Brazil Maize     2015  85283074
10 Brazil Maize    2018  82366531
# ... with 2,746 more rows

```

Além disso, podemos ordenar a base de dados de acordo com duas variáveis.

```

graos %>%
  filter(Element == "Production",
           Item == "Rice, paddy") %>%
  select(Area, Year, Value) %>%
  arrange(Year, desc(Value))

```

```

# A tibble: 754 x 3
  Area      Year      Value
  <chr>    <dbl>    <dbl>
1 Brazil  1961 5392477
2 Colombia 1961  473600
3 Peru     1961  331877
4 Guyana   1961  215103
5 Ecuador  1961  203000
6 Argentina 1961  149000
7 Chile    1961  104720
8 Venezuela (Bolivarian Republic of) 1961   80658
9 Suriname 1961   71562
10 Uruguay 1961   60866
# ... with 744 more rows

```

Perceba que a ordem da declaração das variáveis na função `arrange()` altera a prioridade da ordenação.

```

graos %>%
  filter(Element == "Production",
           Item == "Rice, paddy") %>%
  select(Area, Year, Value) %>%
  arrange(desc(Value), Year)

```

```

# A tibble: 754 x 3
  Area      Year      Value
  <chr>    <dbl>    <dbl>
1 Brazil  2011 13476994
2 Brazil  2004 13277008

```

```

3 Brazil 2005 13192863
4 Brazil 2009 12651144
5 Brazil 2017 12464766
6 Brazil 2015 12301201
7 Brazil 2014 12175602
8 Brazil 2008 12061465
9 Brazil 2018 11808412
10 Brazil 1988 11806450
# ... with 744 more rows

```

Nesse último exemplo, priorizamos a ordenação pelos valores de produção, em quanto que no outro, ordenamos a base a partir dos anos e, posteriormente, dos valores de produção.

6.1.7 Mudar nomes de colunas

Podemos alterar os nomes das colunas com a função `rename()`. Basta inserir o nome desejado e indicar, após o sinal de `=`, qual coluna da base de dados se deseja alterar o nome.

```

graos %>%
  select(Area, Element, Item, Year, Unit, Value) %>%
  rename(`país` = Area, tipo = Element, cultura = Item, ano = Year,
         unidade = Unit, valor = Value)

```

```

# A tibble: 5,510 x 6
  país      tipo      cultura  ano unidade  valor
  <chr>    <chr>    <chr>  <dbl> <chr>    <dbl>
1 Argentina Area harvested Maize  1961 ha      2744400
2 Argentina Area harvested Maize  1962 ha      2756670
3 Argentina Area harvested Maize  1963 ha      2645400
4 Argentina Area harvested Maize  1964 ha      2970500
5 Argentina Area harvested Maize  1965 ha      3062300
6 Argentina Area harvested Maize  1966 ha      3274500
7 Argentina Area harvested Maize  1967 ha      3450500
8 Argentina Area harvested Maize  1968 ha      3377700
9 Argentina Area harvested Maize  1969 ha      3556000
10 Argentina Area harvested Maize  1970 ha      4017330
# ... with 5,500 more rows

```

Partindo de uma base de dados cujas medidas de interesse foram selecionadas e calculadas, podemos representá-los em gráficos, de acordo com o tipo de dado a ser representado. Com isso, no próximo capítulo, veremos como fazer gráficos a partir do pacote `ggplot2`, com o intuito de enxergarmos os nossos dados por outra perspectiva.

Capítulo 7

Visualização

Nesta seção, focaremos na **Visualização de dados** (*Data Visualization*). A visualização consiste em uma etapa importante tanto para enxergar informações relevantes em nossas análises, como para apresentar os resultados obtidos. Para tanto, utilizaremos os recursos disponíveis no pacote `ggplot2`.

O `ggplot2` foi idealizado por Hadley Wickham em sua tese de doutorado, em 2010, intitulada *A Layered Grammar of Graphics*. Desde então, tornou-se um dos pacotes mais populares para a confecção de gráficos elegantes e versáteis, tendo como base a *gramática de gráficos*.

O conceito de *gramática de gráficos* foi proposto originalmente no livro *The Grammar of Graphics*, por Leland Wilkinson, em 2005. Sua lógica se assemelha à gramática linguística na qual, para formularmos uma frase inteligível, devemos seguir uma ordem coerente de palavras. De modo semelhante, para construirmos gráficos a partir do `ggplot2`, devemos ter em mente quais são os seus fundamentos gramaticais.

Basicamente, a *gramática de gráficos* está dividida em 7 camadas, podendo, ou não, estarem juntas, simultaneamente, em um só gráfico. A sua construção variará de acordo com o tipo de gráfico, além da necessidade e subjetividade do cientista de dados. As camadas podem ser classificadas como:

- **Dados** (*Data*): se refere ao *data frame* a ser utilizado para a confecção gráfica. É relevante ter noção sobre os tipos de variáveis que o compõe (qualitativas ou quantitativas), a fim de utilizá-las de maneira correta;
- **Estéticas** (*Aesthetics*): caracterizada pelo mapeamento em formas visuais, como a disposição das variáveis no plano cartesiano e a designação de cores, formas e tamanhos às variáveis;
- **Geometrias** (*Geometries*): representação geométrica do gráfico, seja em pontos, linhas, barras, caixas etc.;
- **Facetas** (*Facets*): é a forma de exibição dos gráficos de acordo com uma variável de interesse, podendo ser divididos em duas grades, em múltiplas grades ou simplesmente de forma individualizada;
- **Estatísticas** (*Statistics*): são os elementos de estatística calculados e presentes no gráfico, podendo ser a média, uma linha de tendência etc.;
- **Coordenadas** (*Coordinates*): definição das dimensões das coordenadas de acordo com o interesse;
- **Tema** (*Theme*): Cores, fonte do texto, tamanhos, formatações, legendas.

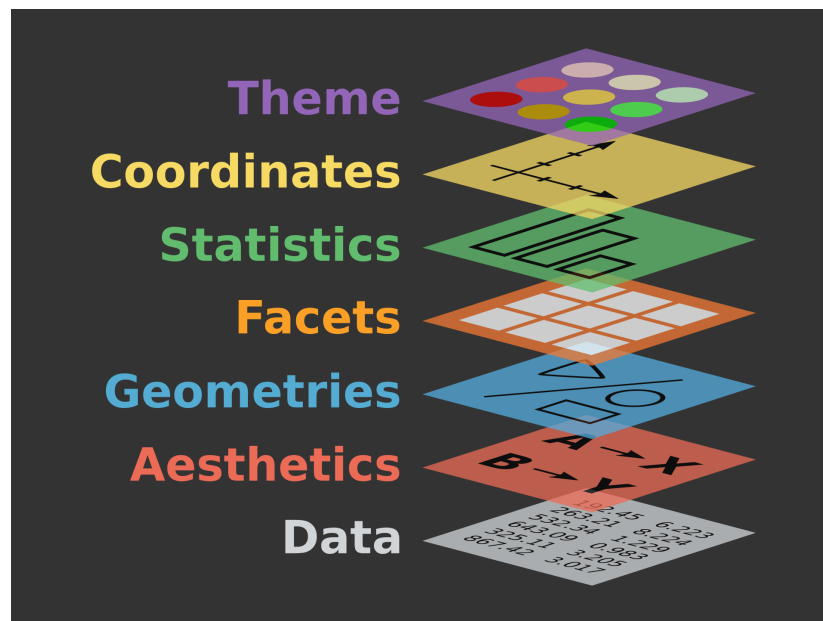


Figura 7.1: Representação das 7 camadas presentes na gramática de gráficos, as quais utilizamos no ggplot2. Fonte: The Grammar of Graphics, 2005.

Portanto, a essência do `ggplot2` é a construção gráfica em camadas. A página [The R Graph Gallery](#) compila uma série de exemplos gráficos possíveis de serem realizados no R, a partir do `ggplot2`, disponibilizando os códigos para que outros possam reproduzi-los e se inspirarem em novas criações.

A seguir, veremos as aplicações das diversas possibilidades gráficas e a lógica gramatical por trás de tudo. Para isso, devemos rodar o pacote `ggplot2`:

```
library(ggplot2)
```

Caso você não tenha instalado o `ggplot2`, prossiga da seguinte maneira:

```
install.packages("ggplot2")
library(ggplot2)
```

Para verificar todos os conteúdos presentes no `ggplot2`, execute o seguinte comando:

```
ls("package:ggplot2")
```

Se você rodou o comando, notou que o `ggplot2` possui mais de 1000 funcionalidades! Mas, apesar da complexidade que este pacote carrega, suas ferramentas básicas são fáceis de serem assimiladas. A seguir, daremos os primeiros passos para construir os principais gráficos em `ggplot2`, para que, posteriormente, o leitor possa explorar com propriedade estas vastas funcionalidades presentes no pacote.

7.1 Gráfico de Dispersão

O primeiro gráfico que construiremos será o de dispersão. Esse tipo de gráfico é muito útil para observarmos a correlação entre duas variáveis contínuas. Para tanto, utilizaremos dados de alunos da

disciplina Estatística Aplicada, que pode ser baixado [clikando aqui](#).

```
library(readr)

dados_alunos <- read_csv("dados_ggplot2/dados_alunos.csv")
```

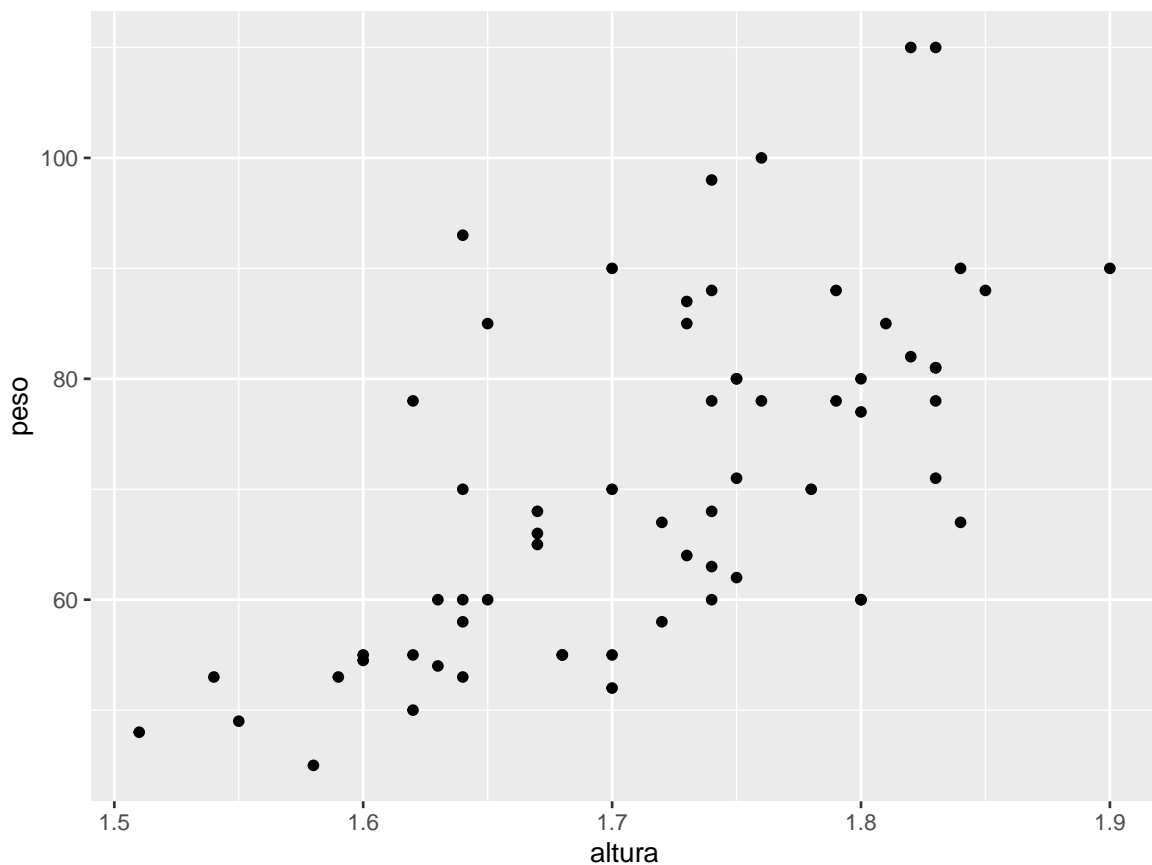
```
dados_alunos
```

```
# A tibble: 64 x 7
  sexo  idade altura  peso horas_estudo media_ponderada futuro
  <chr> <dbl>  <dbl> <dbl>      <dbl>          <dbl> <chr>
1 M      23   1.75   80         2           7.5 academico
2 F      19   1.67   65         2           8.3 mercado
3 M      19   1.7    90         3           6.9 mercado
4 M      22   1.73   87         3           7.1 academico
5 M      19   1.83   71         2           6.5 mercado
6 M      19   1.8    80         3           8.6 mercado
7 M      20   1.9    90         2           7.8 academico
8 F      20   1.6    55         1           8  mercado
9 F      24   1.62   55         2           8.2 academico
10 F     18   1.64   60         2           7.3 mercado
# ... with 54 more rows
```

A base de dados possui 64 observações e 7 variáveis, as quais informam o sexo, idade, altura (em metros), peso (em kg), horas de estudo (por dia), média ponderada no curso e perspectiva futura após a graduação.

Neste primeiro momento, utilizaremos as variáveis altura e peso para construir o gráfico de pontos.

```
ggplot(data = dados_alunos,
       mapping = aes(x = altura,
                     y = peso)) +
  geom_point()
```

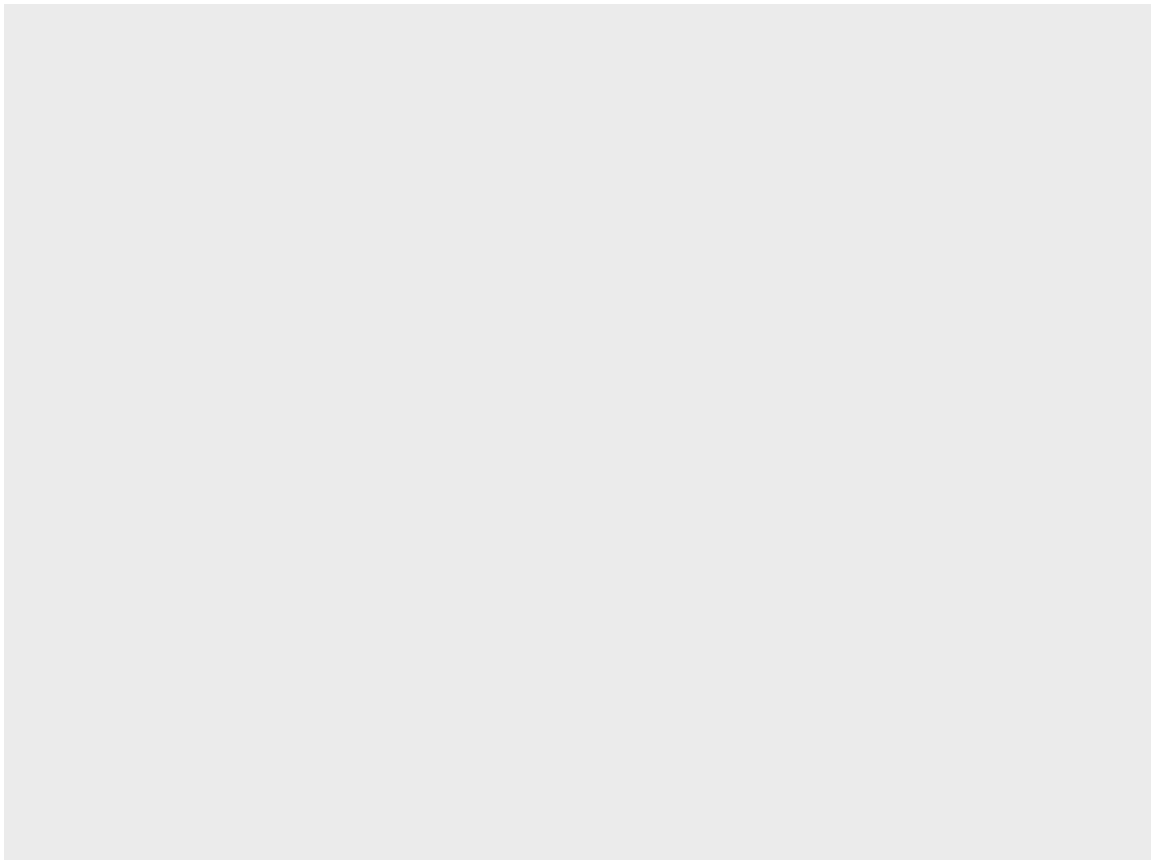


A primeira camada do nosso gráfico é dada pela função `ggplot()`. O argumento `data` = recebe o objeto `dados_alunos`, ou seja, recebe a base de dados que importamos anteriormente e que utilizaremos para construir o gráfico. Em seguida, o argumento `mapping` = define a estética do gráfico a partir da função `aes()`, sendo atribuído ao eixo x a variável `altura` (`x = altura`) e ao eixo y, a variável `peso` (`y = peso`). Por fim, criamos uma última camada referente ao tipo de geometria adotado no gráfico, no caso, a geometria de pontos `geom_point()`.

Perceba que as camadas são unidas por um sinal de `+`. Portanto, a combinação das funções `ggplot()` e `geom_point()` define o tipo de gráfico resultante.

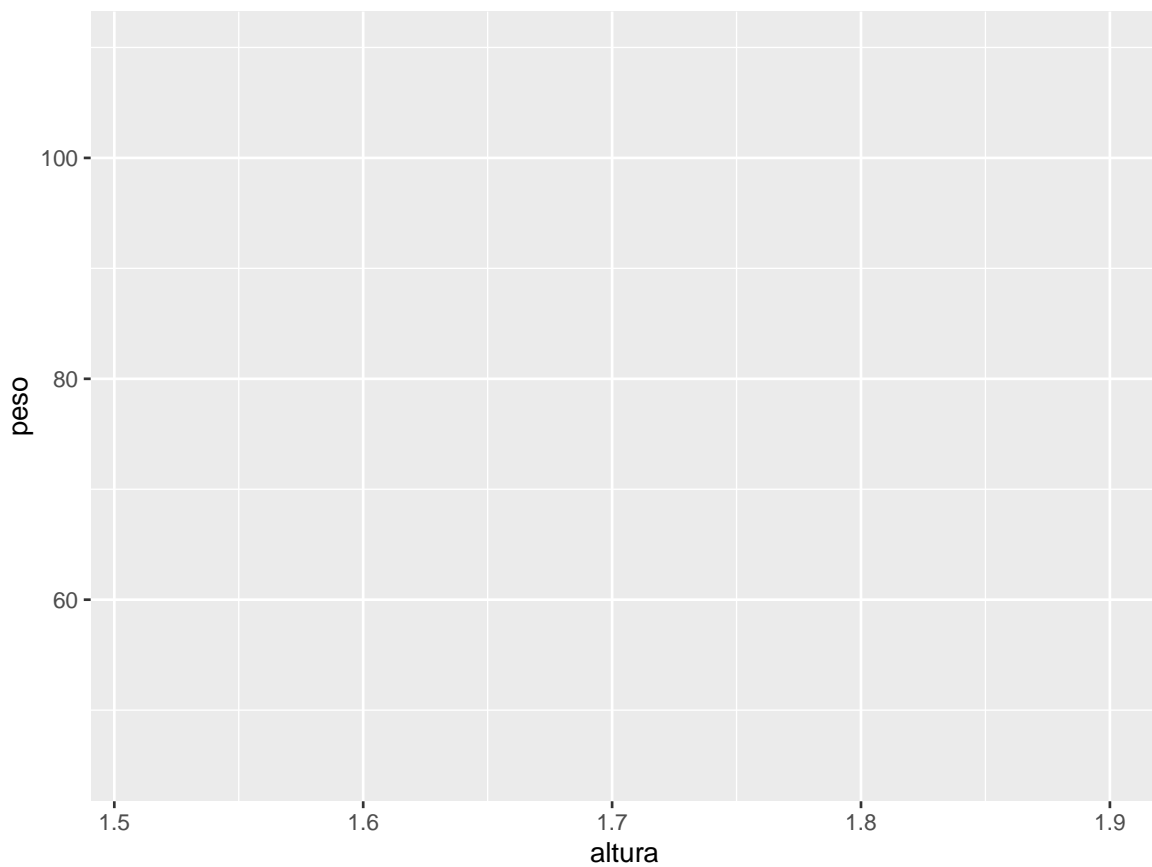
Para se ter uma noção de como foi gerado o gráfico e entender melhor a lógica da *gramática de gráficos*, rodaremos o código anterior por partes.

```
ggplot(data = dados_alunos)
```



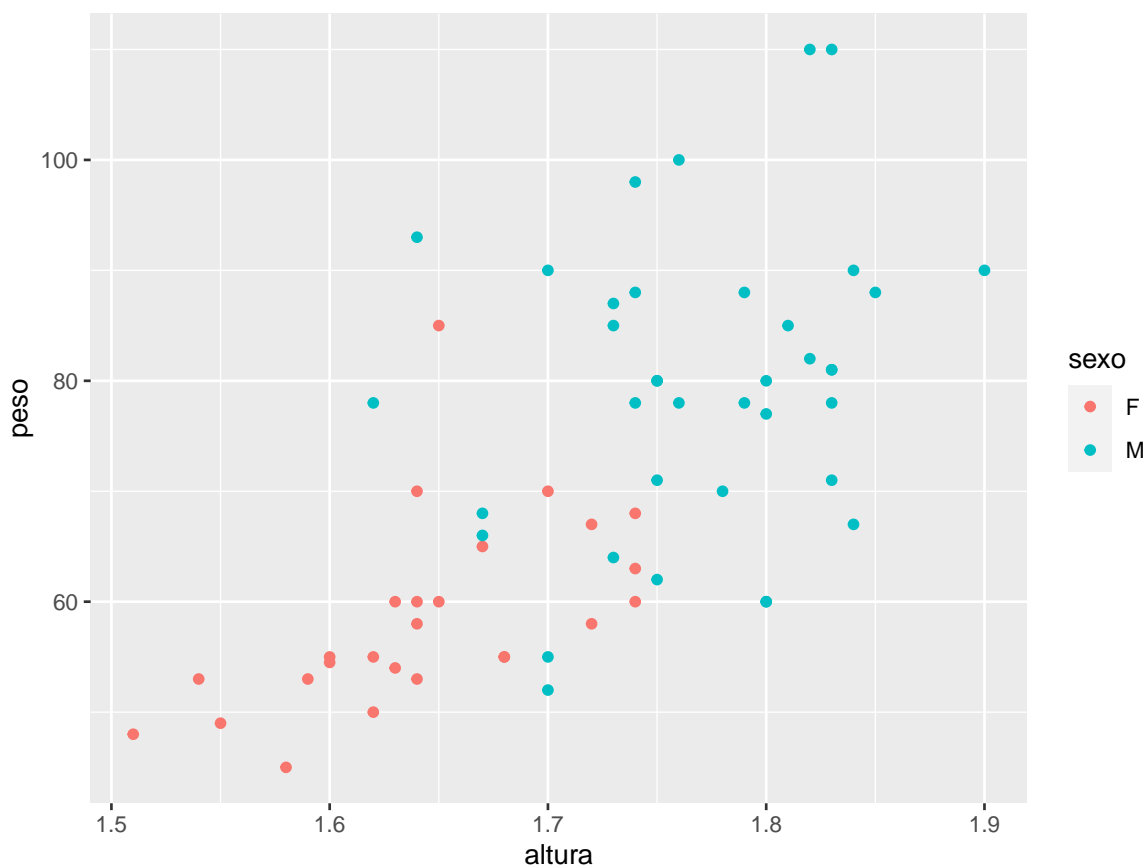
Ao definir somente o argumento `data = dados_alunos` na função `ggplot()`, o `ggplot2` nos retorna um gráfico vazio, pois indicamos apenas qual a base de dados que utilizaremos, sem fornecer informações referentes à estrutura do gráfico.

```
ggplot(data = dados_alunos,  
       mapping = aes(x = altura,  
                     y = peso))
```



Agora, indicando as variáveis mapeadas nos eixos x e y, temos um gráfico com as coordenadas definidas, porém sem ter os dados plotados, pois ainda não definimos qual o tipo de geometria será utilizada.

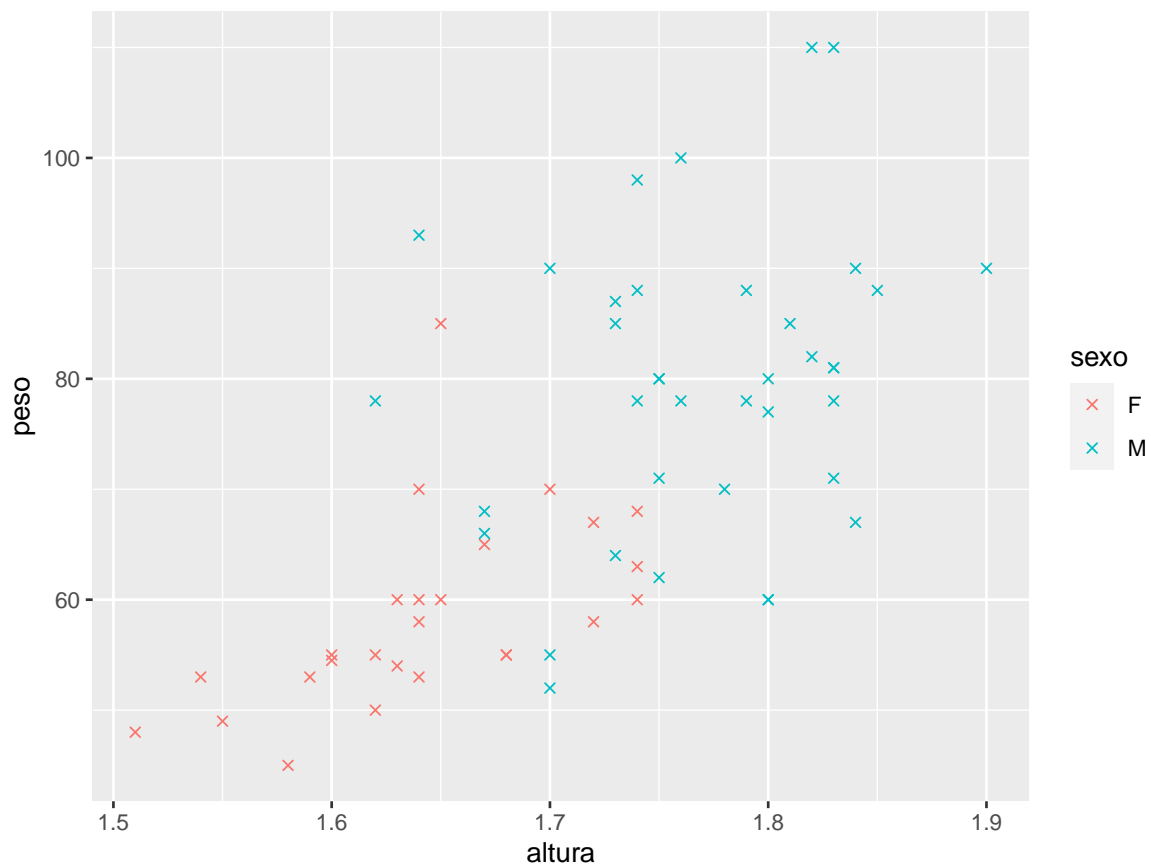
```
ggplot(data = dados_alunos,  
       mapping = aes(x = altura,  
                     y = peso)) +  
  geom_point()
```

7.1.2 Formatos

Podemos realizar algumas modificações em relação à aparência dos pontos, como por exemplo, alterar seu formato utilizando o argumento `shape` = dentro da função `geom_point()`.

```
ggplot(data = dados_alunos,  
       mapping = aes(x = altura,  
                     y = peso,  
                     color = sexo)) +  
  geom_point(shape = 4)
```

Cada tipo de formato é representado por um número, cujas legendas podem ser conferidas na figura 7.2.

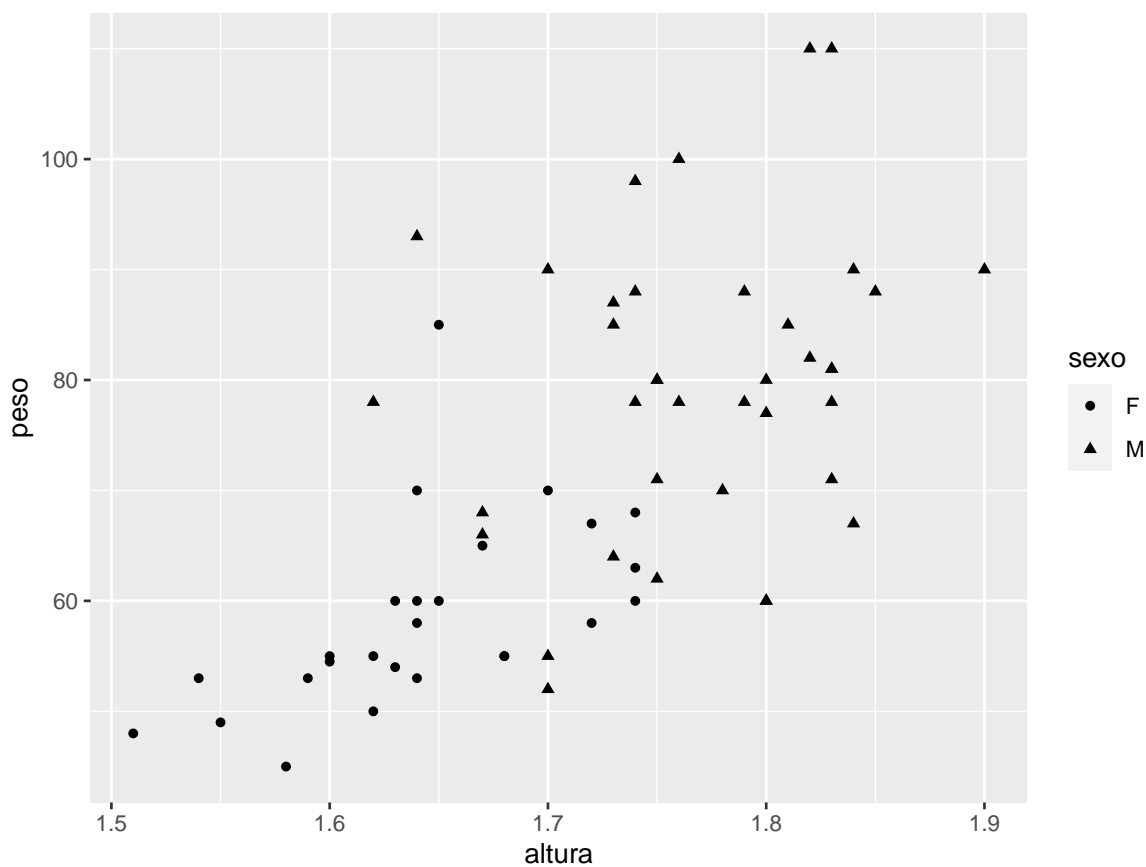
| | | | | |
|-----|-----|------|------|------|
| □ 0 | ✕ 4 | ⊕ 10 | ■ 15 | ■ 22 |
| ○ 1 | ▽ 6 | ☆ 11 | ● 16 | ● 21 |
| △ 2 | ⊠ 7 | ⊞ 12 | ▲ 17 | ▲ 24 |
| ◇ 5 | ✳ 8 | ⊗ 13 | ◆ 18 | ◆ 23 |
| ⊕ 3 | ⬢ 9 | ⬢ 14 | ● 19 | ● 20 |

Figura 7.2: Legendas dos tipos de formatos de pontos, indicados no argumento ‘shape’. Fonte: *R for Data Science*, 2017.

Além de atribuir um único tipo de formato aos pontos, podemos distinguir variáveis a partir dos formatos. Para isso, utilizamos o argumento `shape` dentro da função `aes()`, indicando qual variável

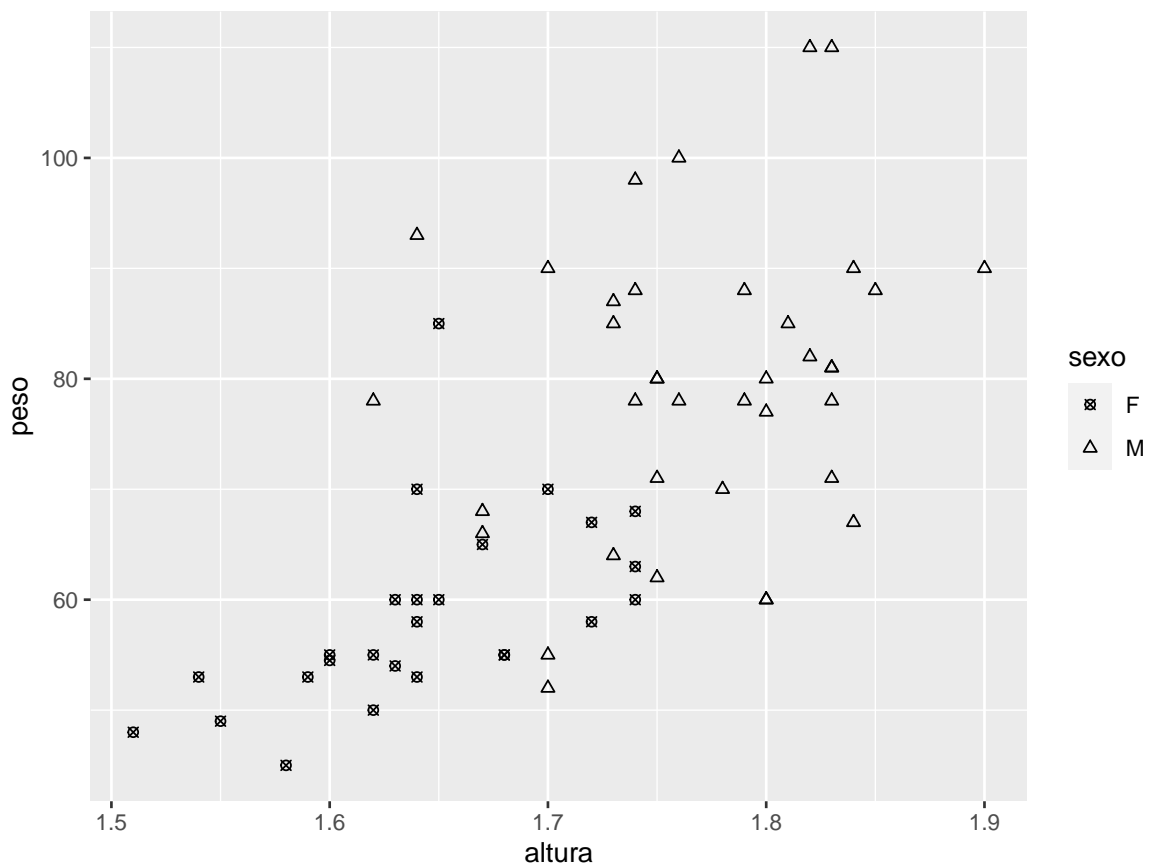
da base de dados será atribuída ao argumento. Como exemplo, utilizaremos novamente a variável `sexo`.

```
ggplot(data = dados_alunos,  
       mapping = aes(x = altura,  
                     y = peso,  
                     shape = sexo)) +  
  geom_point()
```



Como padrão, o argumento atribui aos pontos os formatos 16 e 17. Para alterá-los, utilizamos a função `scale_shape_manual()`, com o argumento `values` recebendo um vetor com os números dos formatos que se deseja atribuir.

```
ggplot(data = dados_alunos,  
       mapping = aes(x = altura,  
                     y = peso,  
                     shape = sexo)) +  
  geom_point()+  
  scale_shape_manual(values = c(13, 24))
```

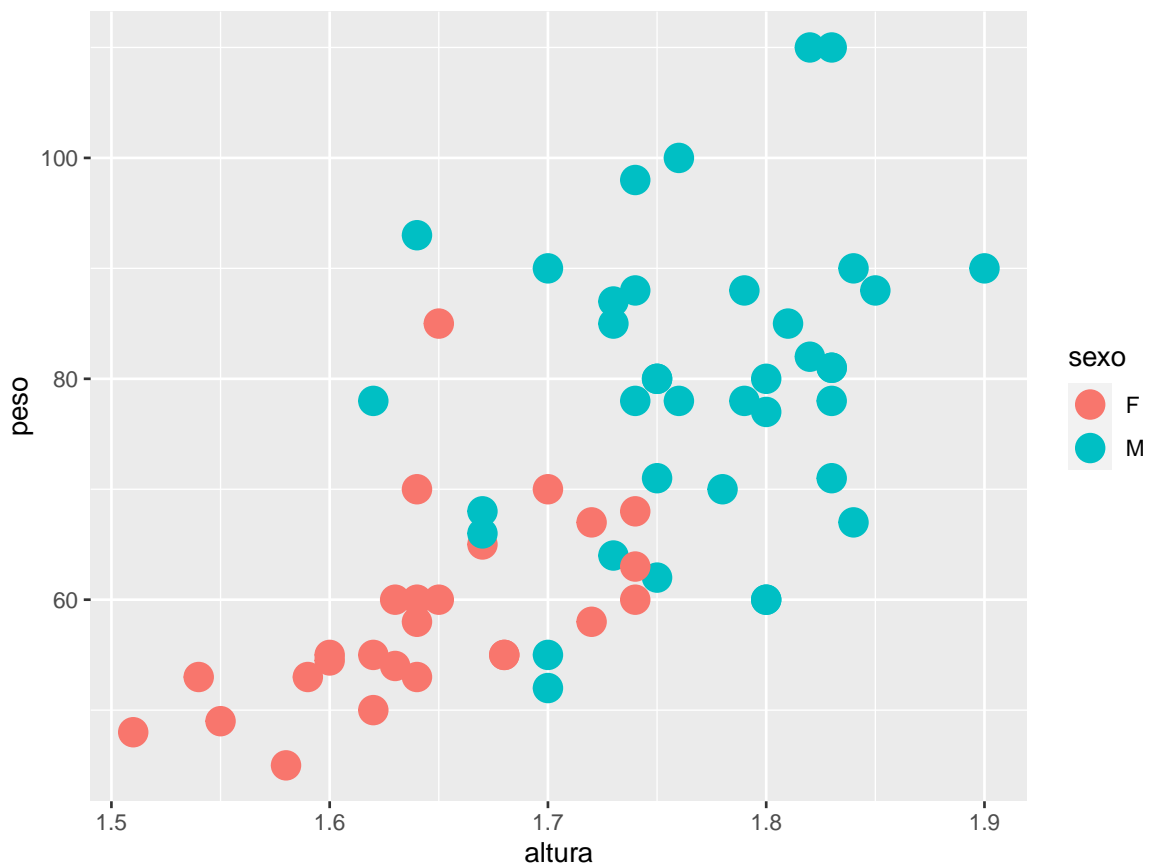


Perceba que a ordem dos números no vetor segue a ordem das variáveis, ou seja, o formato 13 é referente ao sexo feminino (F) e o formato 24, ao sexo masculino (M).

7.1.3 Tamanho

Ainda, podemos alterar o tamanho dos pontos. Para isso, utilizamos o argumento `size` dentro da função `geom_point()`.

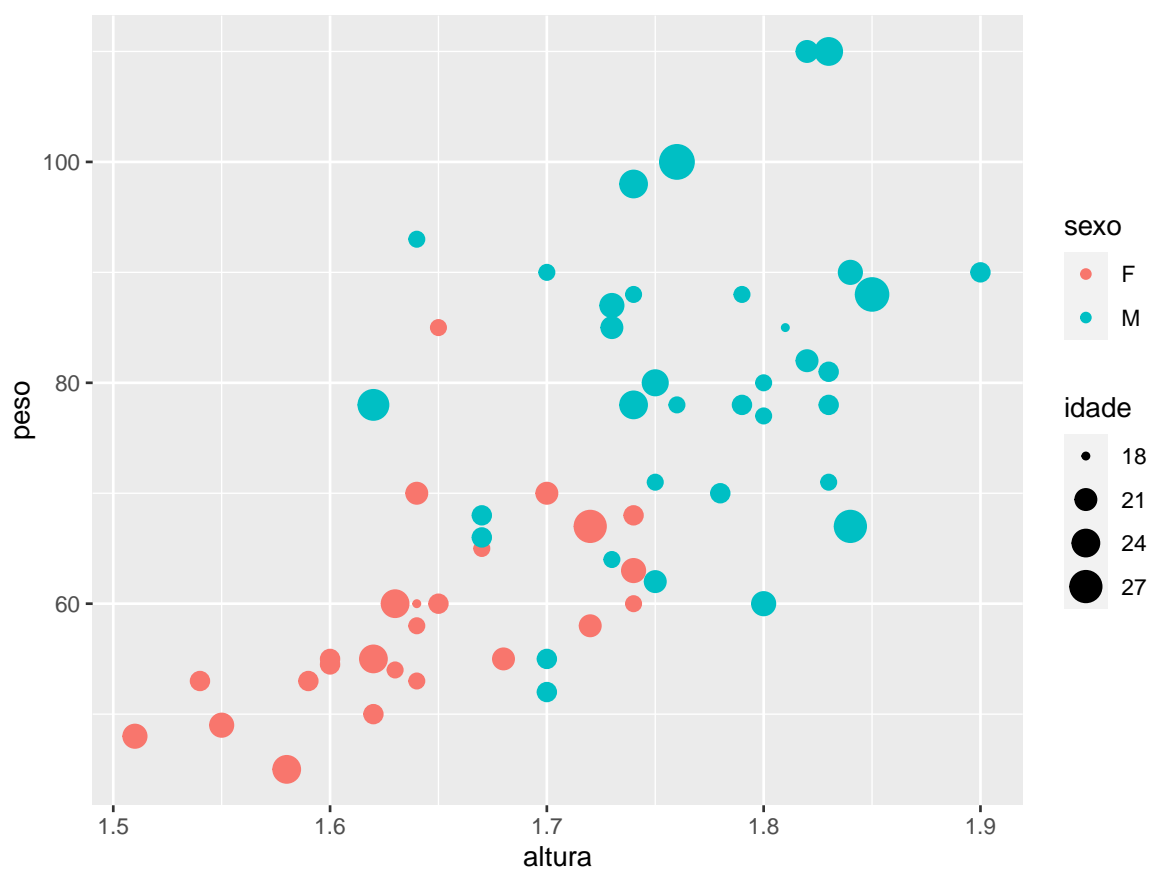
```
ggplot(data = dados_alunos,  
       mapping = aes(x = altura,  
                     y = peso,  
                     color = sexo)) +  
  geom_point(size = 5)
```



Caso o argumento `size` não seja especificado, por padrão, o valor adotado é igual a 1. Assim, podemos gerar pontos maiores designando valores superiores a 1, ou senão, pontos menores, atribuindo valores inferiores a 1.

Outra possibilidade é diferenciar as idades dos alunos pelo tamanho dos pontos. Para isso, o argumento `size` receberá a variável `idade`, dentro a função `aes()`.

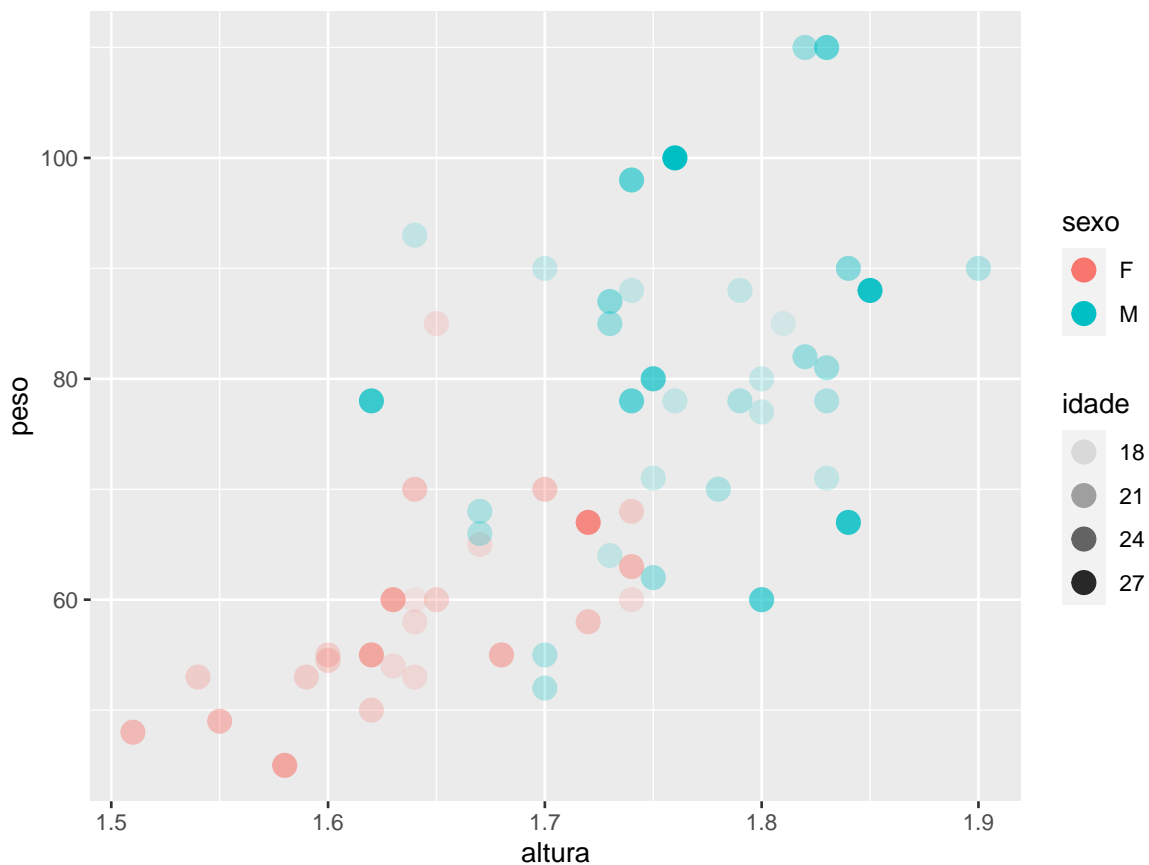
```
ggplot(data = dados_alunos,  
       mapping = aes(x = altura,  
                     y = peso,  
                     color = sexo,  
                     size = idade)) +  
geom_point()
```



7.1.4 Transparência

Por fim, podemos diferenciar as idades pela transparência dos pontos, utilizando o argumento `alpha`.

```
ggplot(data = dados_alunos,  
       mapping = aes(x = altura,  
                     y = peso,  
                     color = sexo,  
                     alpha = idade)) +  
  geom_point(size = 4)
```

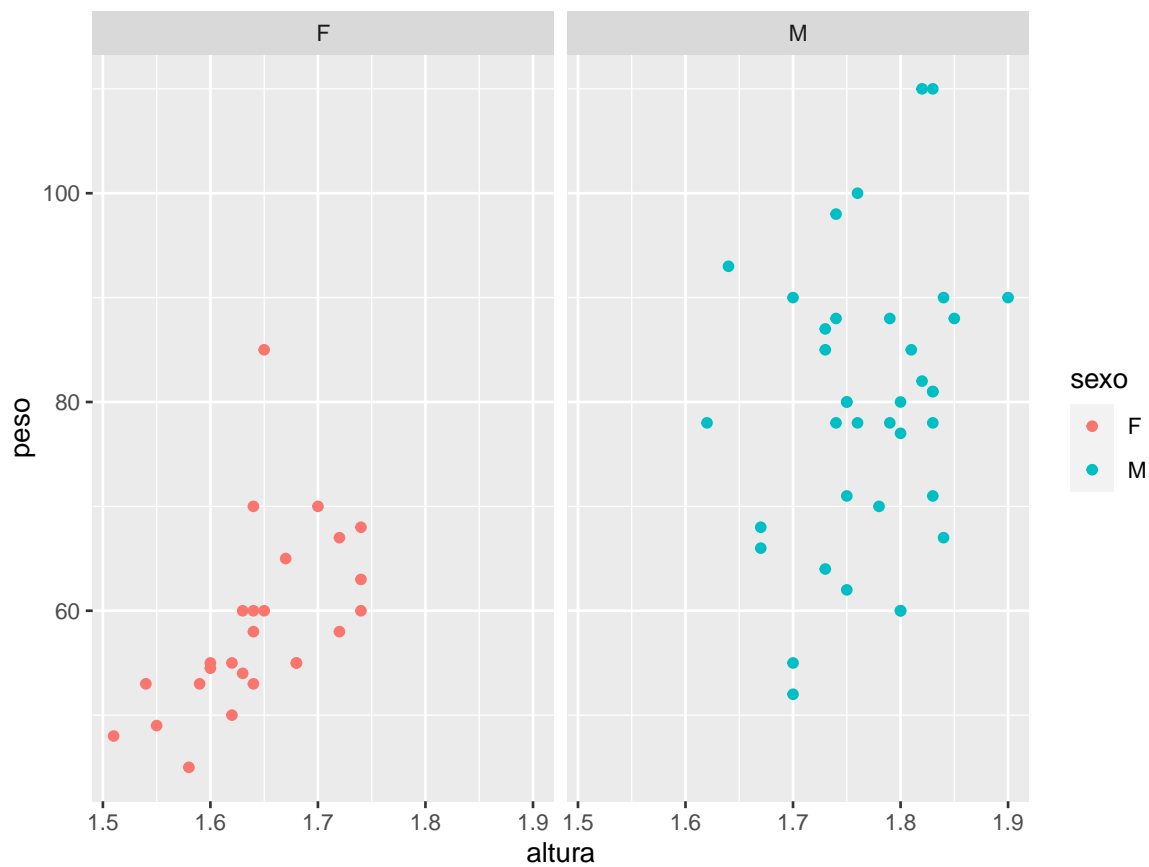


Você deve ter percebido que um mesmo argumento pode ser utilizado de diferentes maneiras. Nos exemplos anteriores, utilizamos os argumentos `shape` e `size` dentro da função `aes()`, mas também na `geom_point()`. Quando utilizamos dentro da `aes()`, o argumento sempre recebe uma **variável** da base de dados. Por outro lado, quando utilizada na `geom_point()`, eles recebem um valor genérico contido em uma determinada escala. Assim, devemos nos atentar a esses detalhes para construirmos os gráficos de acordo com as especificações e posições dos argumentos.

7.1.5 Facetas (*Facets*)

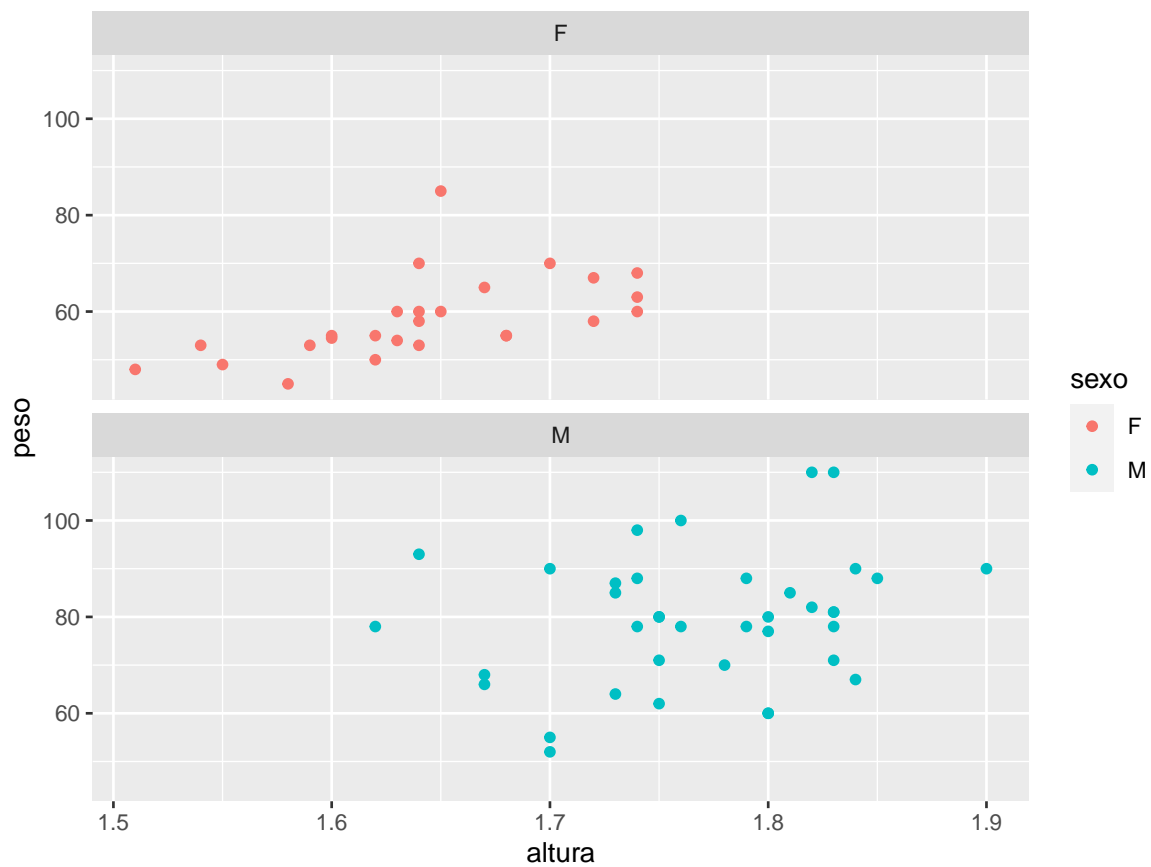
As facetas (*facets*) replicam os gráficos, separando-os em grades (*grids*), de acordo com uma variável categórica do nosso banco de dados. Para ficar mais claro, vamos exemplificar as facetas. Para isso, utilizaremos a função `facet_wrap()`.

```
ggplot(data = dados_alunos,
       mapping = aes(x = altura,
                     y = peso,
                     color = sexo)) +
  geom_point() +
  facet_wrap(~sexo, ncol = 2)
```



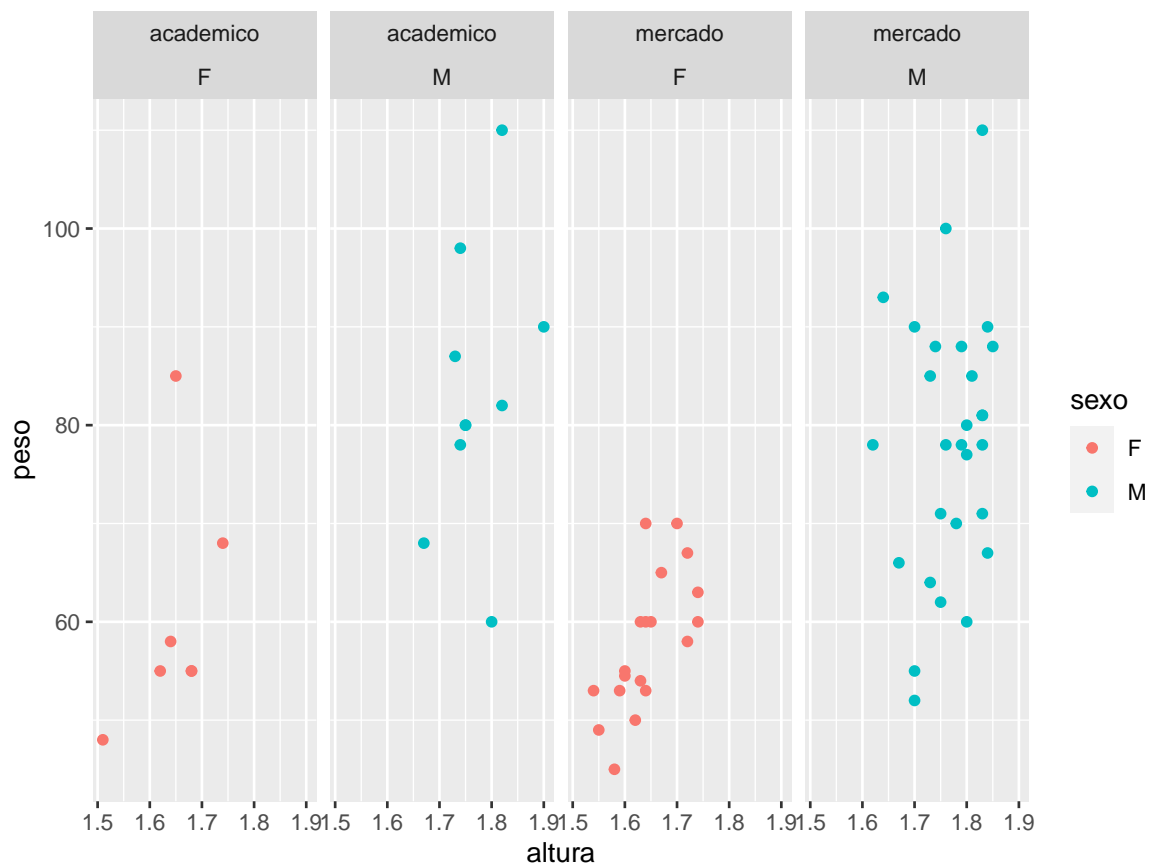
A função `facet_wrap()` entra como uma terceira camada ao gráfico. Como argumento, utilizamos a fórmula `~sexo` para dizer que a variável `sexo` será utilizada como fator para quebrar o gráfico em duas grades. Como se pode perceber, cada grade recebe somente os dados referentes aos respectivos sexos. Logo em seguida, temos um outro argumento que nos indica a disposição dos gráficos. No caso do `ncol`, os gráficos ficam dispostos lado a lado, sendo que para o `nrow` os gráficos se dispõem um embaixo do outro, como podemos ver a seguir.

```
ggplot(data = dados_alunos,  
       mapping = aes(x = altura,  
                     y = peso,  
                     color = sexo)) +  
  geom_point() +  
  facet_wrap(~sexo, nrow = 2)
```



Ainda podemos associar duas variáveis categóricas ao *facet*. Como exemplo, adicionaremos a variável **futuro** à fórmula **futuro~sexo**, a fim de verificarmos as perspectivas dos alunos sobre seus futuros após a graduação, de acordo com o sexo.

```
ggplot(data = dados_alunos,
       mapping = aes(x = altura,
                     y = peso,
                     color = sexo)) +
  geom_point() +
  facet_wrap(futuro~sexo, ncol = 4)
```

Perceba que o número atribuído ao argumento `ncol` = define o número de colunas do *facet*. A mesma lógica é válida para o `nrow` =, porém definindo o número de linhas.

7.1.6 Linhas de referência

Podemos adicionar linhas de referência aos nossos gráficos.

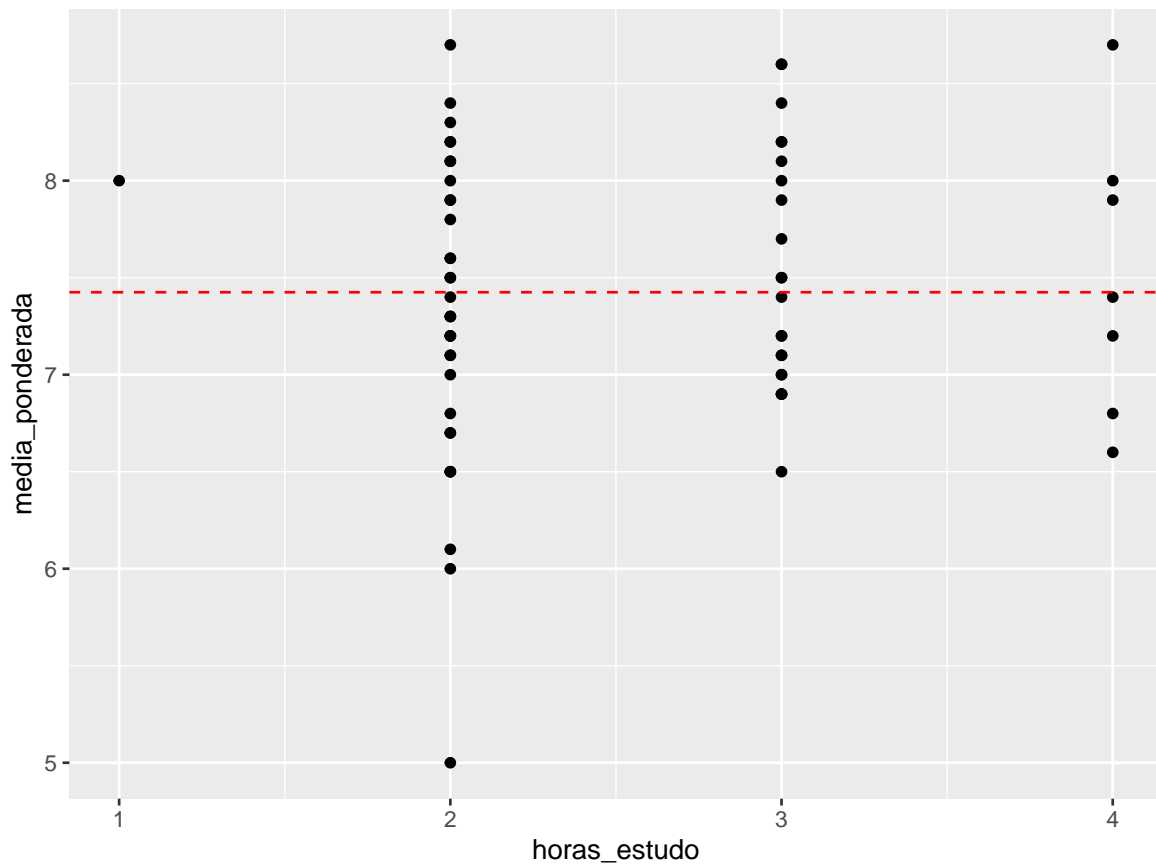
Linhas horizontais

Para criar linhas horizontais, utilizamos a função `geom_hline()`. Como argumento, devemos usar a `yintercept`, que indica em qual ponto do eixo y será traçada a linha de referência horizontal.

No exemplo a seguir, será construído um gráfico de dispersão entre as horas de estudo e a média ponderada dos estudantes. A linha de referência será a média (`mean()`) da média ponderada dos alunos, indicada no argumento `yintercept`.

```
ggplot(data = dados_alunos,
       mapping = aes(x = horas_estudo,
                     y = media_ponderada)) +
  geom_point() +
  geom_hline(mapping = aes(yintercept = mean(media_ponderada)),
```

```
color = "red",  
linetype = 2)
```



Perceba que o argumento `yintercept` foi colocado dentro da função `aes()`, pois utilizamos uma variável da nossa base de dados para construir a linha. Além disso, atribuímos a cor vermelha à linha (`color = "red"`) e definimos seu estilo com o argumento `linetype`. Cada tipo de linha é representado por um número, cuja legenda pode ser conferida na figura 7.3.

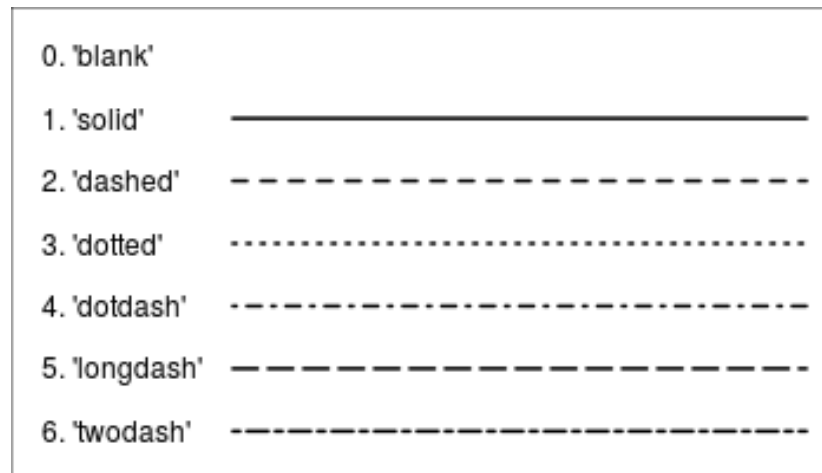
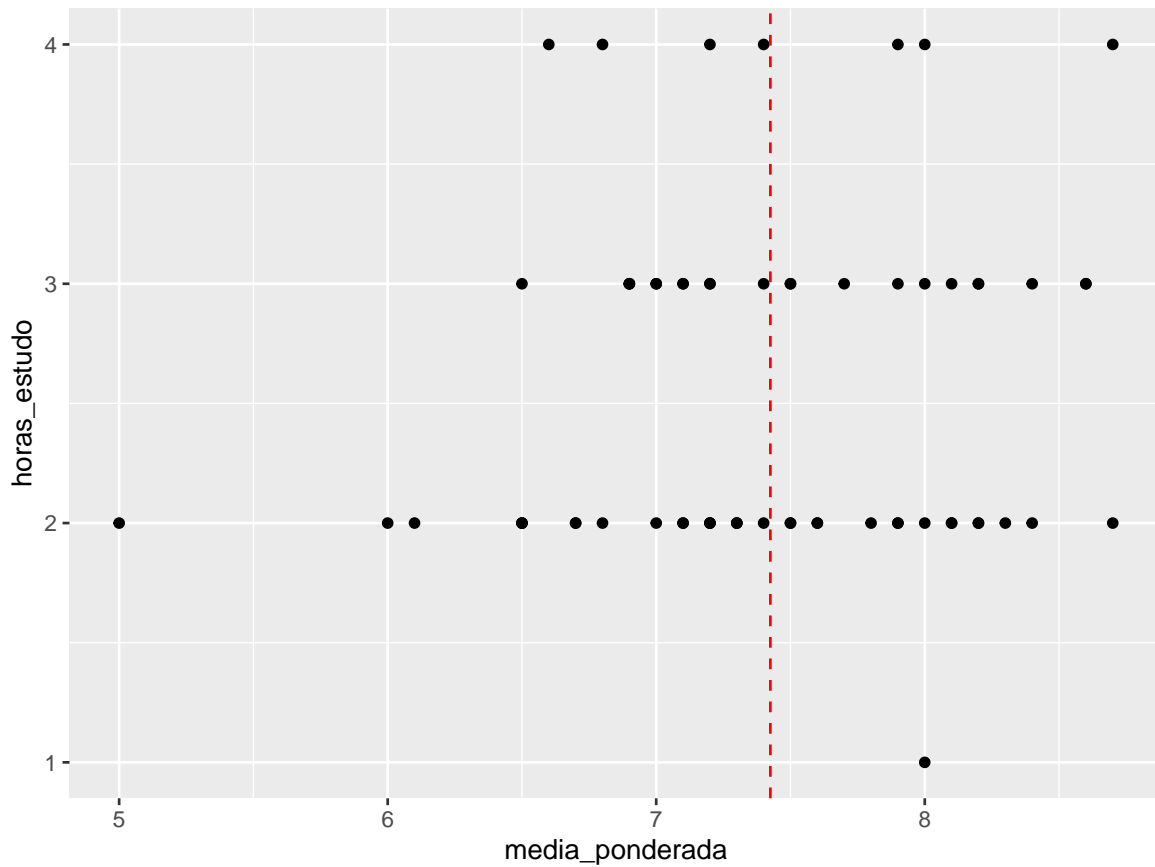


Figura 7.3: Possíveis tipos de linhas a partir do argumento 'linetype'. Fonte: R Graphics Cookbook.

Linhas verticais

As linhas de referência vertical são análogas às linha horizontais. São construídas a partir da função `geom_vline()`, que recebe o argumento `xintercept =` para indicar em qual ponto do eixo x será traçada a linha.

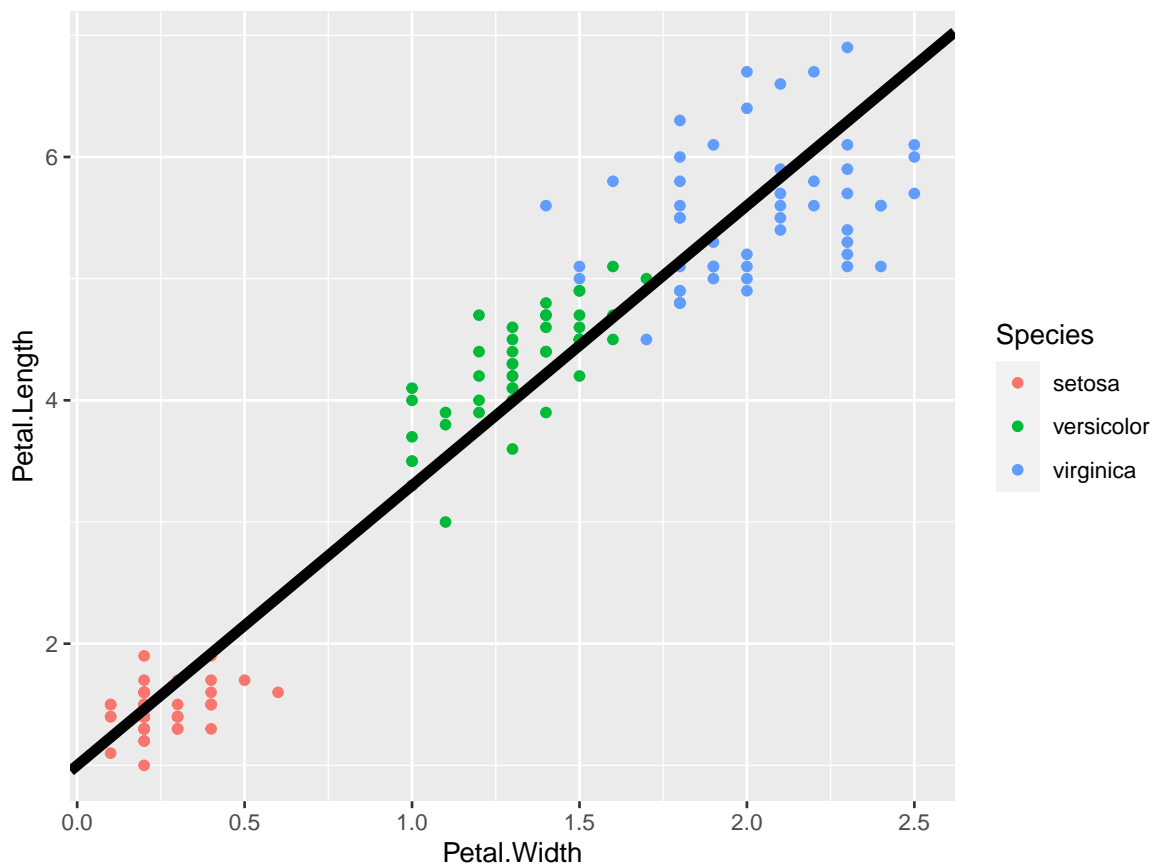
```
ggplot(data = dados_alunos,  
       mapping = aes(x = media_ponderada,  
                     y = horas_estudo))+  
  geom_point()+  
  geom_vline(mapping = aes(xintercept = mean(media_ponderada)),  
            color = "red",  
            linetype = 2)
```



Linhas diagonais

Já as linhas diagonais são feitas com a `geom_abline()`. Essa função desenha qualquer linha que siga a equação $y = a + b \cdot x$, sendo a o ponto onde a reta toca no eixo y , representado pelo argumento `intercept`, e b , o coeficiente angular da reta, indicado pelo argumento `slope`.

```
ggplot(iris, aes(x = Petal.Width,
                 y = Petal.Length,
                 color = Species)) +
  geom_point() +
  geom_abline(intercept = 1,
              slope = 2.3,
              size = 2)
```



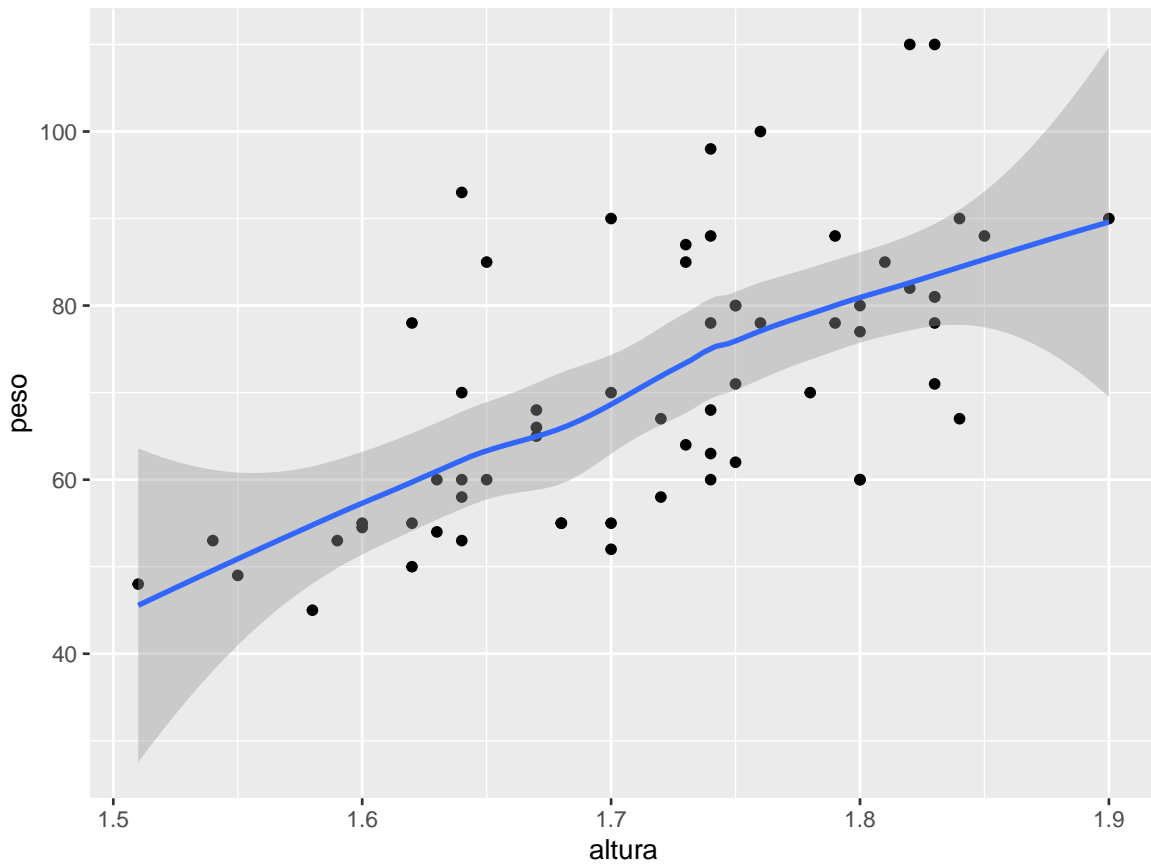
Como exemplo, utilizamos a base de dados nativa do R `iris` para criar um gráfico de dispersão entre a largura e o comprimento das pétalas de flores. Na função `geom_abline()` especificamos os argumentos `intercept = 1` e `slope = 2.3` para informar que a reta começa na coordenada (0,1), possuindo um coeficiente de inclinação igual a 2,3.

Perceba que, em um mesmo gráfico, utilizamos duas camadas geométricas. Nesse último exemplo, a função `geom_point()` foi sobreposta pela `geom_abline()`, pois segundo a lógica da *gramática dos gráficos*, esses são construídos em camadas, portanto, a camada `geom_point()` vem antes da `geom_abline()`, sendo então sobreposta pela última.

7.1.7 Linhas de regressão

Também podemos incluir linhas de regressão ajustadas aos dados, a partir da função `geom_smooth()`.

```
ggplot(data = dados_alunos,
       mapping = aes(x = altura,
                     y = peso)) +
  geom_point() +
  geom_smooth()
```

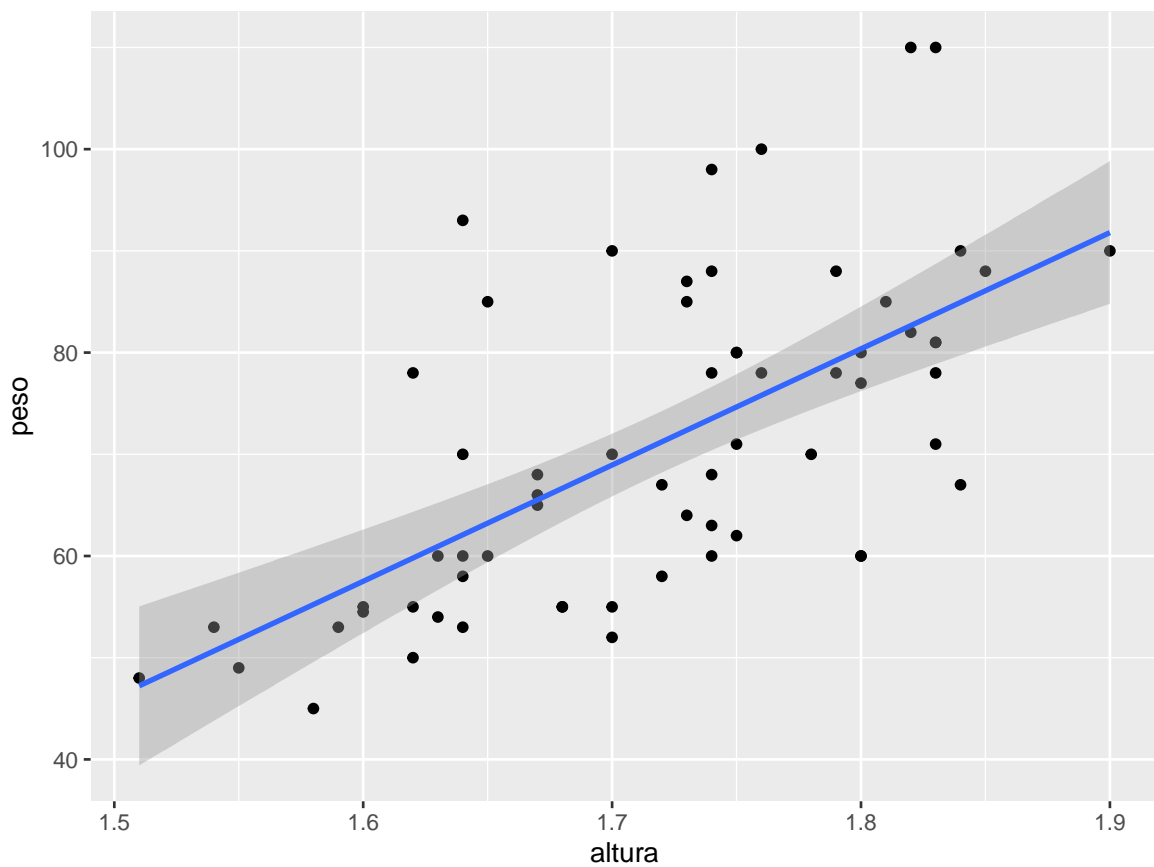


Neste primeiro gráfico, definimos dentro da função `ggplot()` os eixos `x = altura` e `y = peso`. Sendo assim, essa estética é utilizada por ambas as funções geométricas, tanto a `geom_point()`, como a `geom_smooth()`.

Podemos definir na função `geom_smooth()` o tipo de método para gerar a linha de regressão. Por padrão, o método utilizado é o `loess` (sigla de *locally estimated scatterplot smoothing*).

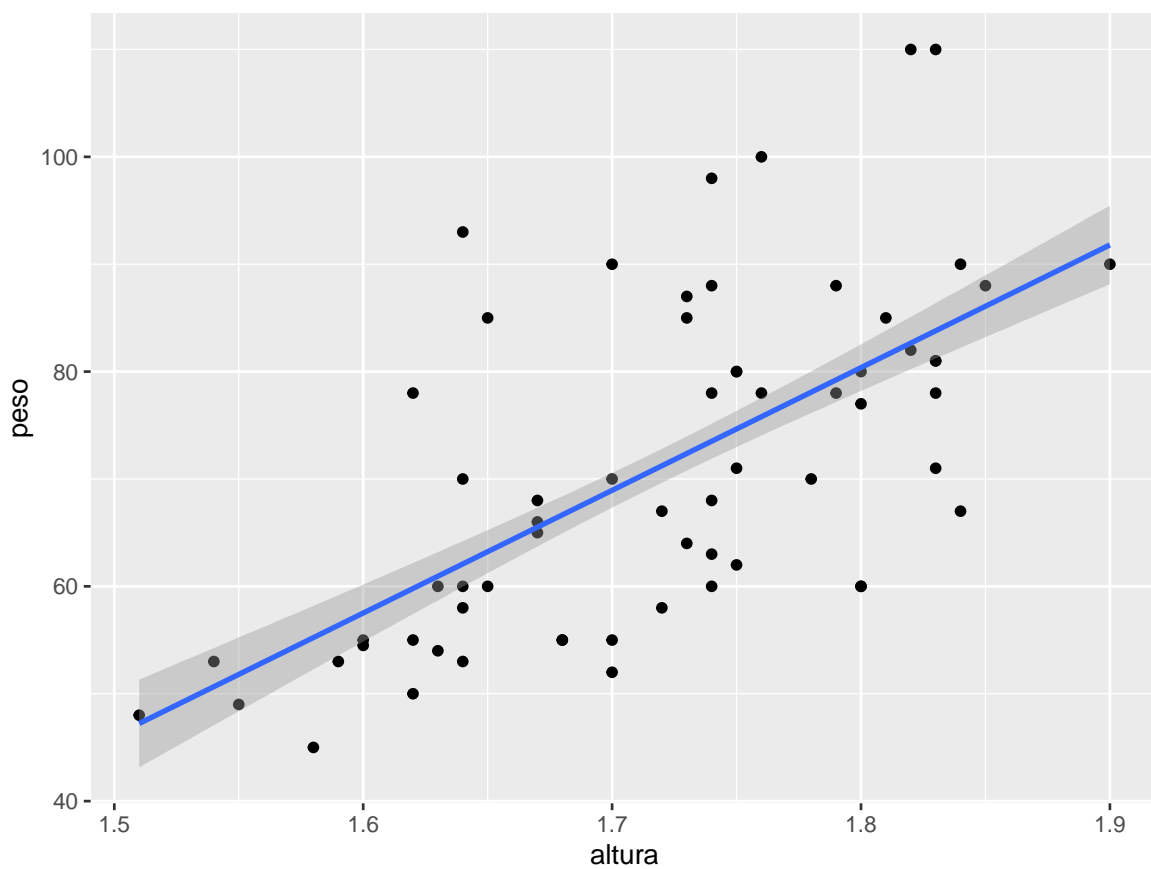
Para alterar o método, utilizamos o argumento `method =` na função `geom_smooth()`. No exemplo a seguir, definimos o método `"lm"`, ou seja, modelo linear (*linear model*).

```
ggplot(data = dados_alunos,  
       mapping = aes(x = altura,  
                     y = peso))+  
  geom_point()+  
  geom_smooth(method = "lm")
```



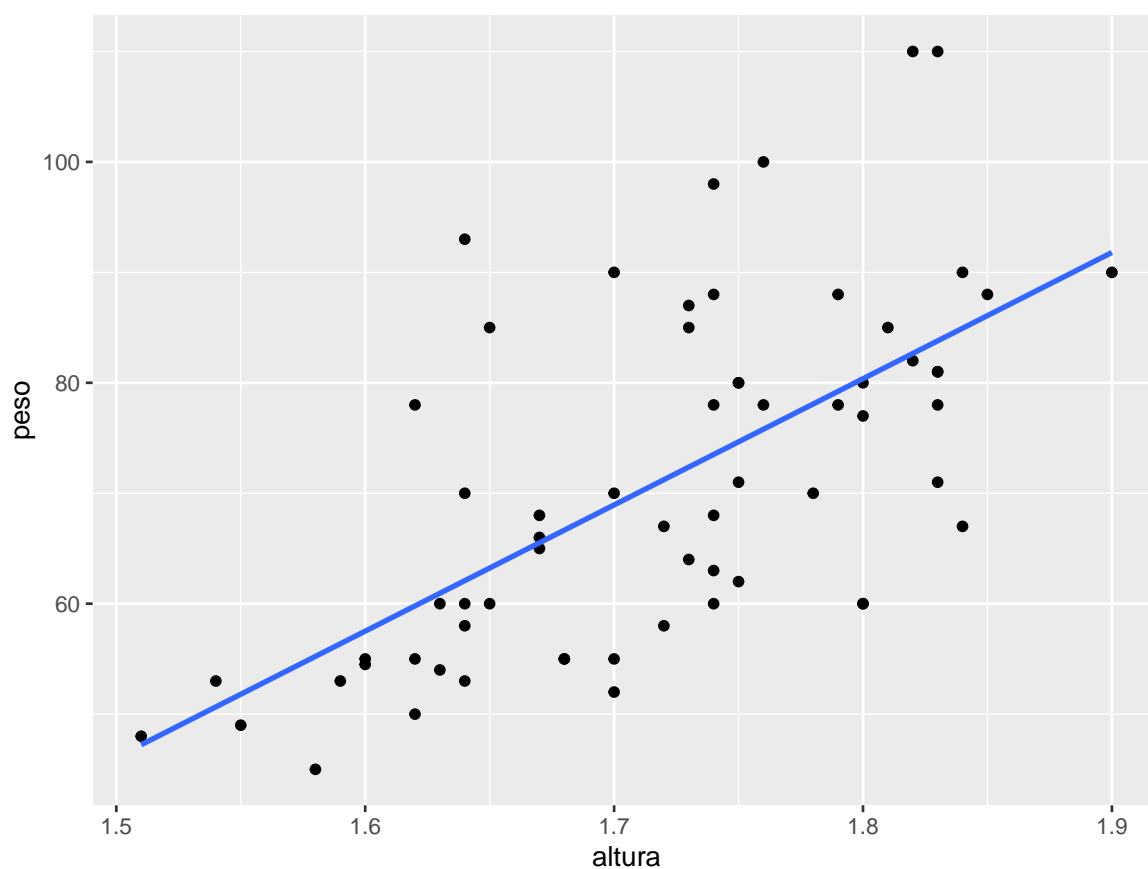
Perceba que a linha de regressão apresenta uma região sombreada em seu entorno. Essa região é o intervalo de confiança que, por padrão, adota-se 95% de confiança. Para alterar o nível de confiança, utilizamos o argumento `level`. No exemplo a seguir, adotaremos 70% de confiança (`level = 0.70`).

```
ggplot(data = dados_alunos,  
       mapping = aes(x = altura,  
                     y = peso)) +  
  geom_point() +  
  geom_smooth(method = "lm",  
             level = 0.70)
```



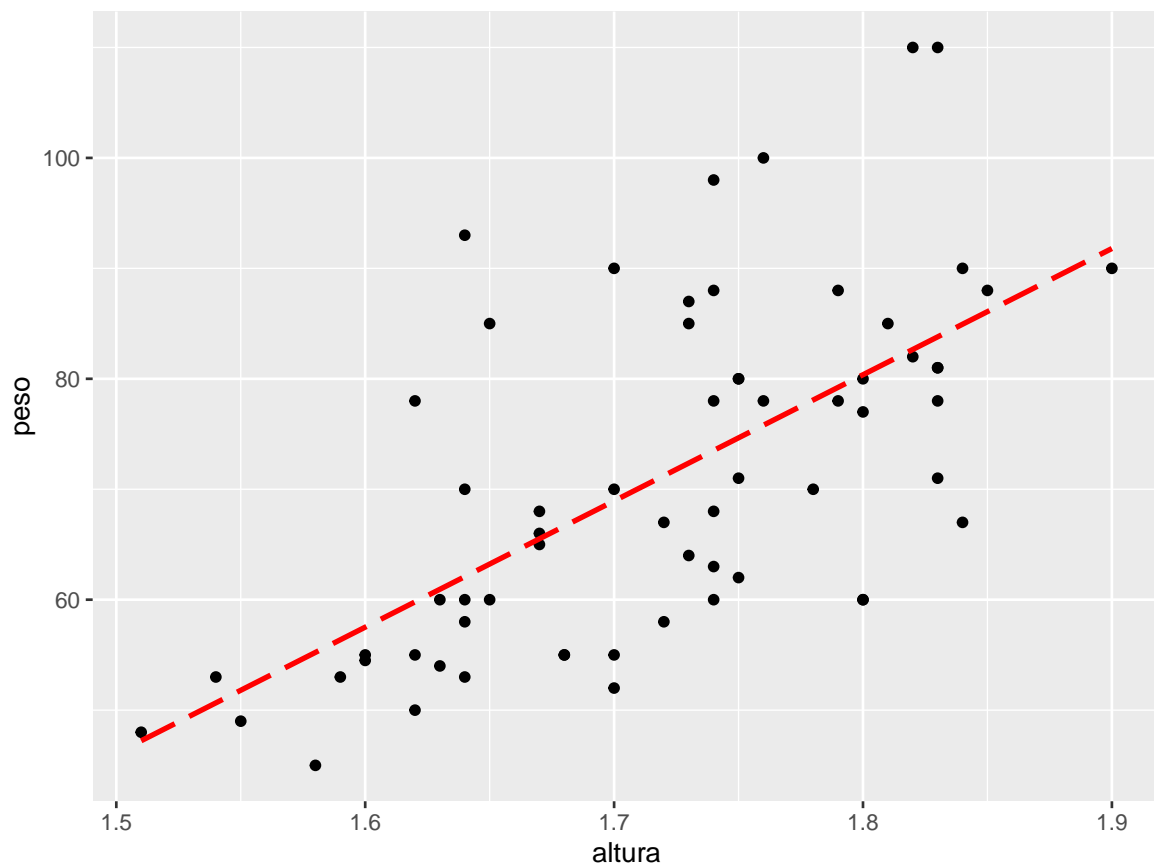
Para desativar o intervalo de confiança, utilizamos o argumento `se = FALSE`.

```
ggplot(data = dados_alunos,  
       mapping = aes(x = altura,  
                     y = peso)) +  
  geom_point() +  
  geom_smooth(method = "lm",  
             se = FALSE)
```

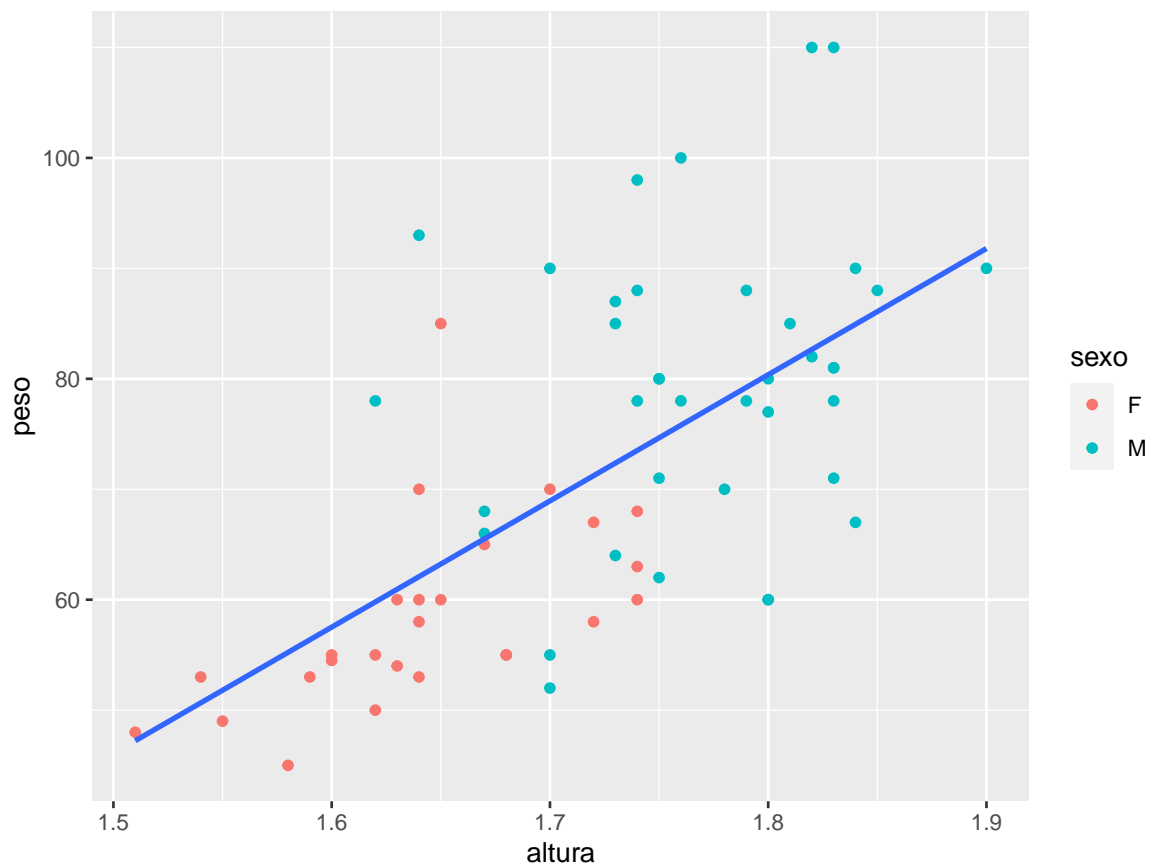
Ainda podemos alterar alguns fatores estéticos da reta, como por exemplo a cor, tamanho e tipo de linha.

```
ggplot(data = dados_alunos,  
       mapping = aes(x = altura,  
                     y = peso)) +  
  geom_point() +  
  geom_smooth(method = "lm",  
             se = FALSE,  
             color = "red",  
             size = 1,  
             linetype = 5)
```



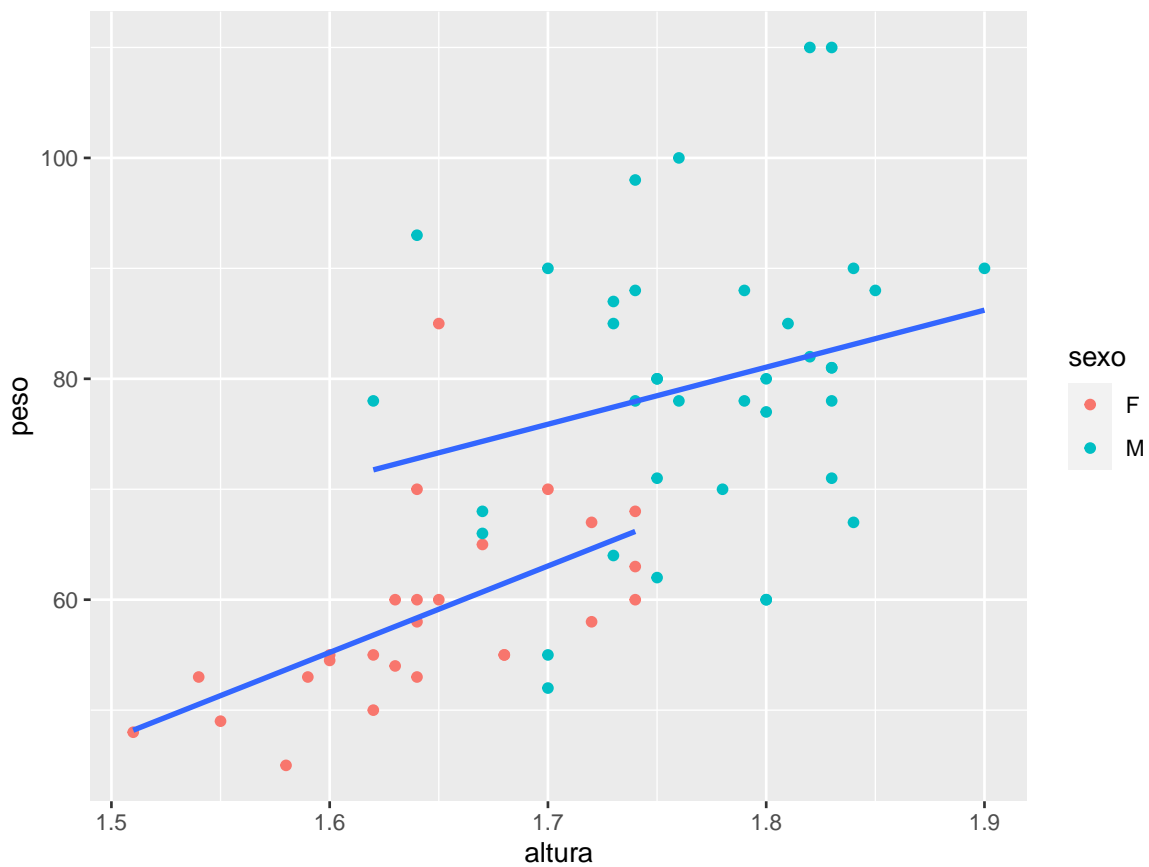
Para construir linhas de regressão para diferentes grupos, em um mesmo gráfico, podemos prosseguir de diferentes maneiras. No exemplo a seguir, distinguiremos os pontos por cores, de acordo com o sexo.

```
ggplot(data = dados_alunos)+  
  geom_point(mapping = aes(x = altura,  
                           y = peso,  
                           color = sexo))+  
  geom_smooth(mapping = aes(x = altura,  
                           y = peso),  
             method = "lm",  
             se = FALSE)
```



Perceba que, apesar de ter distinguido os pontos de acordo com o sexo, foi traçada uma única linha de regressão para ambos os sexos. Para construir uma linha para cada sexo, precisamos agrupar a variável `sexo` na função `aes()` da `geom_smooth()`, a partir do argumento `group =`.

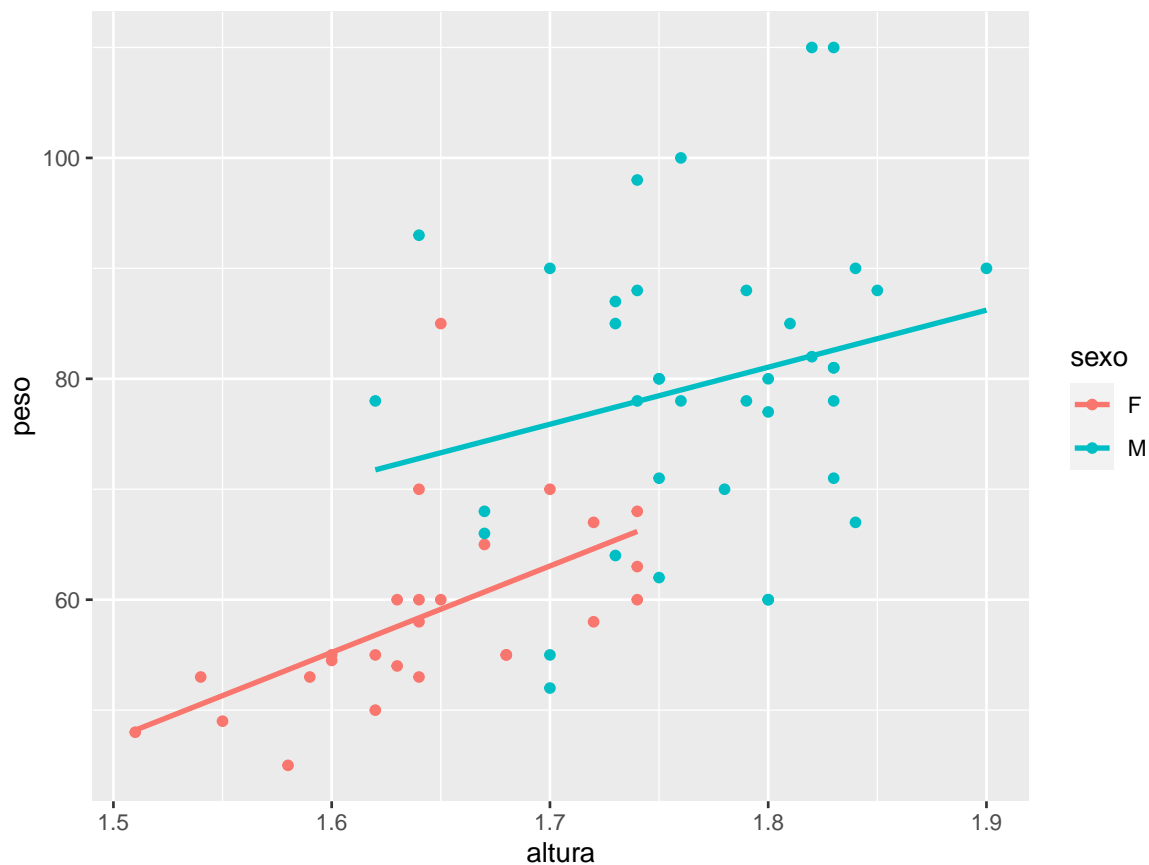
```
ggplot(data = dados_alunos)+  
  geom_point(mapping = aes(x = altura,  
                           y = peso,  
                           color = sexo))+  
  geom_smooth(mapping = aes(x = altura,  
                           y = peso,  
                           group = sexo),  
             method = "lm",  
             se = FALSE)
```



Nesses últimos exemplo, perceba que definimos a estética do gráfico (`aes()`) dentro de cada geometria e não mais na função `ggplot()`. Isso permite definir as estéticas individuais de cada geometria.

Caso contrário, poderíamos definir uma mesma estética para ambas as geometrias, definindo-a como função da `ggplot()`. No exemplo a seguir, traremos uma outra solução para o problema das linhas de regressão por categoria, definindo a `aes()` na função `ggplot()`.

```
ggplot(data = dados_alunos,
       mapping = aes(x = altura,
                     y = peso,
                     color = sexo)) +
  geom_point() +
  geom_smooth(method = "lm",
             se = FALSE)
```



Nesse caso, definimos o fator cor (`color`) para diferenciarmos o sexo, tanto para a `geom_point()`, como para a `geom_smooth()`. Assim, perceba que os pontos e as retas de regressão ficaram com cores diferentes, de acordo com o sexo.

Para saber mais sobre as linhas de regressão no `ggplot2`, confira os capítulos 5.6 a 5.9 do excelente livro [R Graphics Cookbook](#).

7.1.8 Títulos e rótulos

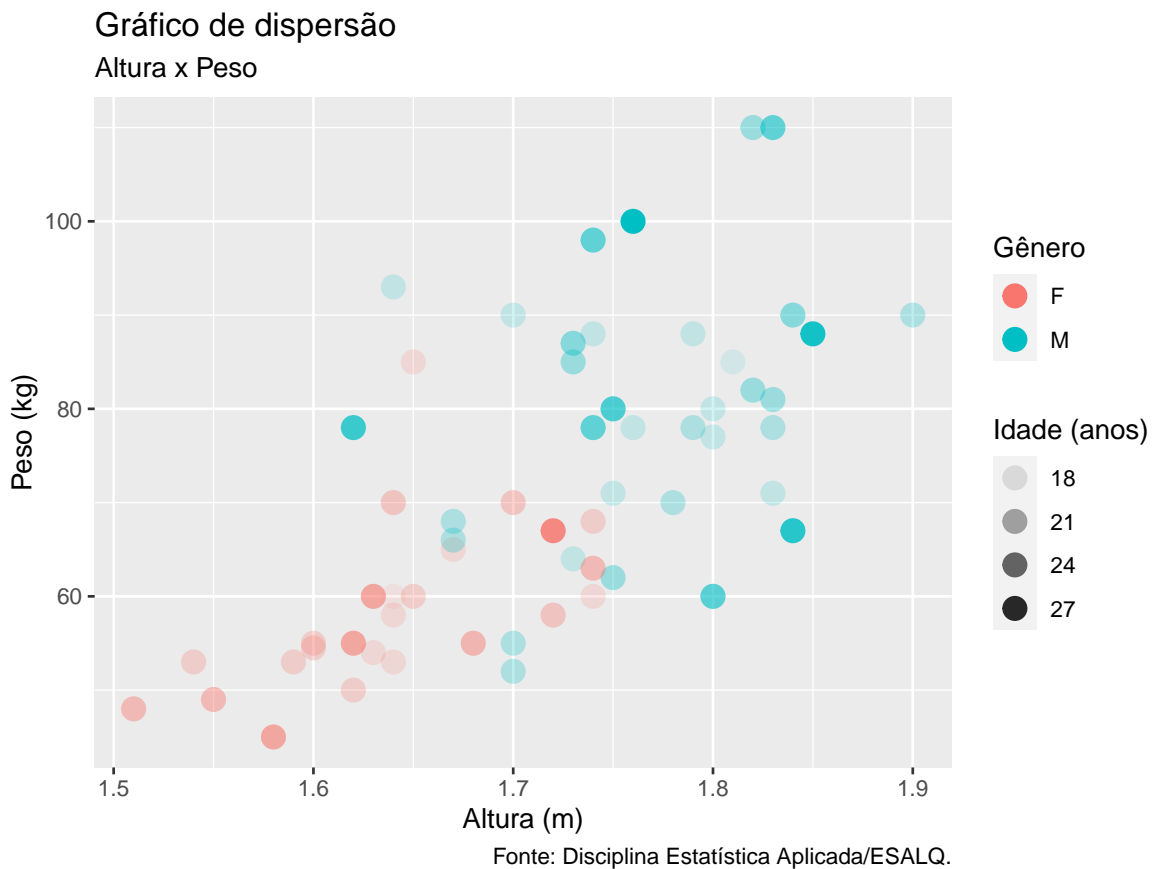
De maneira intuitiva, para colocarmos títulos e rótulos nos gráficos, adicionamos a camada `labs()`. Como argumento, indicamos qual fator desejamos (re)nomear. Atente-se ao fato de que as nomeações devem estar entre aspas.

```
ggplot(data = dados_alunos,
       mapping = aes(x = altura,
                     y = peso,
                     color = sexo,
                     alpha = idade)) +
  geom_point(size = 4) +
  labs(
    x = "Altura (m)",
    y = "Peso (kg)",
```

```

color = "Gênero",
alpha = "Idade (anos)",
title = "Gráfico de dispersão",
subtitle = "Altura x Peso",
caption = "Fonte: Disciplina Estatística Aplicada/ESALQ."
)

```



Podemos adicionar mais alguns ajustes, como centralizar o título e subtítulo, colocar a fonte à esquerda, além de ocultar as legendas.

```

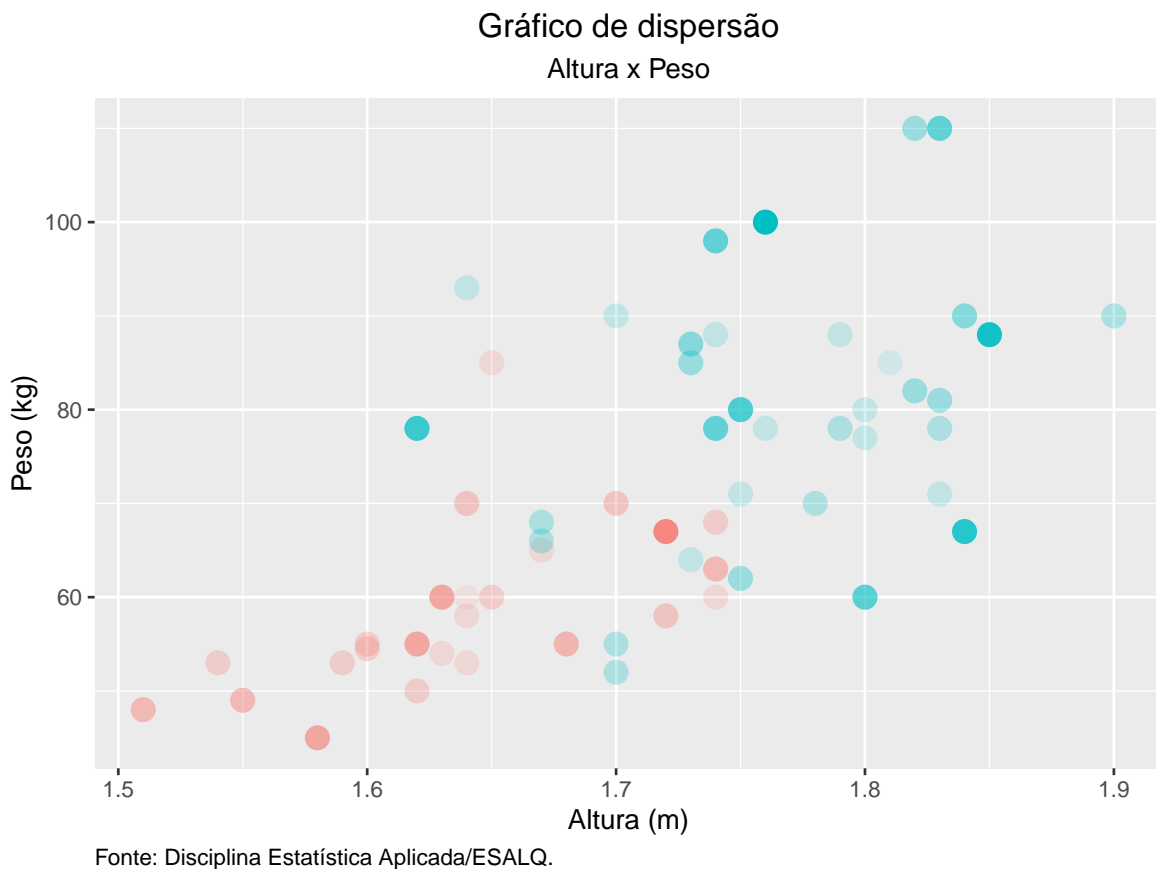
ggplot(data = dados_alunos,
       mapping = aes(x = altura,
                     y = peso,
                     color = sexo,
                     alpha = idade)) +
  geom_point(size = 4) +
  labs(
    x = "Altura (m)",
    y = "Peso (kg)",
    color = "Gênero",
    alpha = "Idade (anos)",
    title = "Gráfico de dispersão",

```

```

  subtitle = "Altura x Peso",
  caption = "Fonte: Disciplina Estatística Aplicada/ESALQ."
)+
theme(plot.title = element_text(hjust = 0.5),
      plot.subtitle = element_text(hjust = 0.5),
      plot.caption = element_text(hjust = 0),
      legend.position = "none")

```



Nesse caso, conjuntamente à função `theme()`, utilizamos outra função, a `element_text()`, para ajustar o posicionamento horizontal do título (`plot.title`) e do subtítulo (`plot.subtitle`), além da fonte (`plot.caption`). O argumento `hjust` apresenta uma escala de 0 a 1, sendo 0 o posicionamento mais à esquerda do gráfico. Assim, para centralizarmos os textos, definimos o ajuste na metade da escala (`hjust = 0.5`).

No caso das legendas, ainda dentro da função `theme()`, utilizamos o argumento `legend.position = "none"` para ocultar todas as legendas presentes no gráfico.

Caso queira ocultar somente uma das legendas, utilizamos a função `guides()`, informando a variável cuja legenda se deseja ocultar.

```

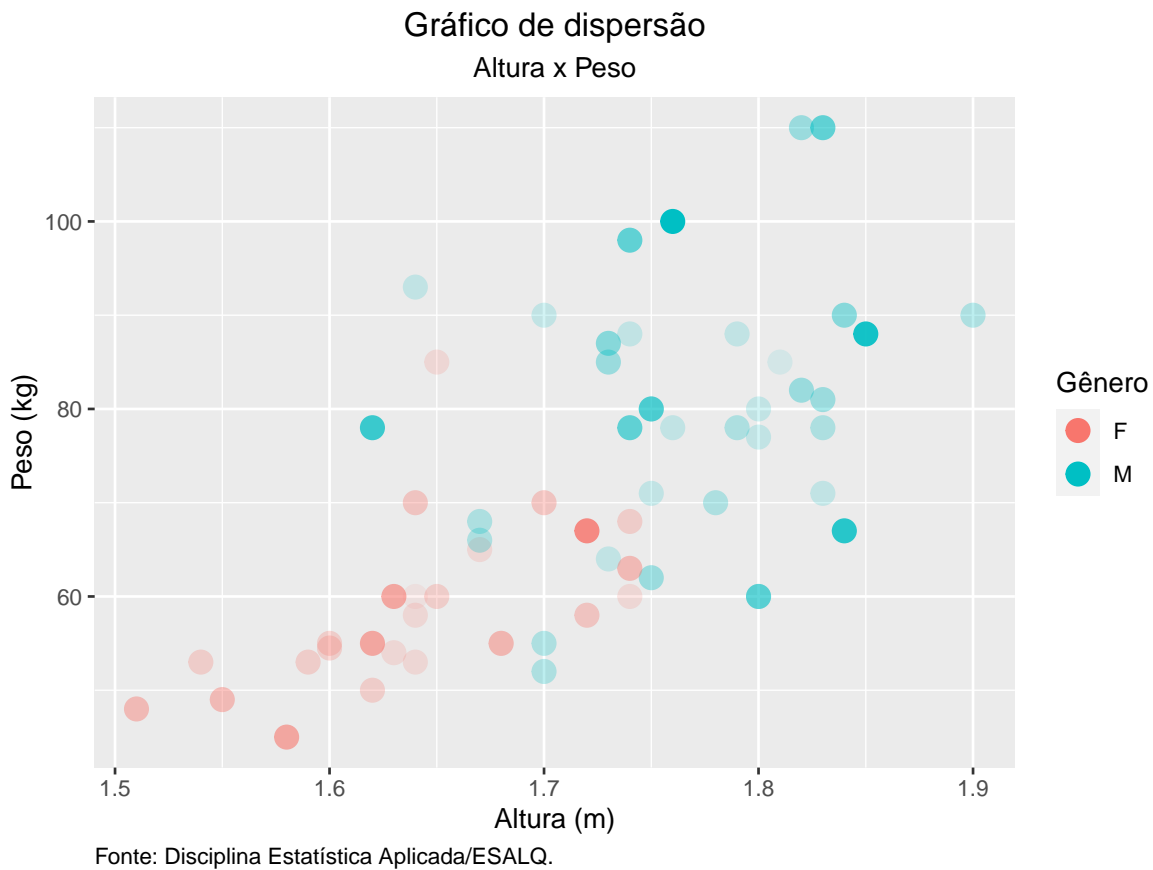
ggplot(data = dados_alunos,
      mapping = aes(x = altura,
                    y = peso,

```

```

        color = sexo,
        alpha = idade)) +
geom_point(size = 4) +
labs(
  x = "Altura (m)",
  y = "Peso (kg)",
  color = "Gênero",
  alpha = "Idade (anos)",
  title = "Gráfico de dispersão",
  subtitle = "Altura x Peso",
  caption = "Fonte: Disciplina Estatística Aplicada/ESALQ."
) +
theme(plot.title = element_text(hjust = 0.5),
      plot.subtitle = element_text(hjust = 0.5),
      plot.caption = element_text(hjust = 0)) +
guides(alpha = FALSE)

```



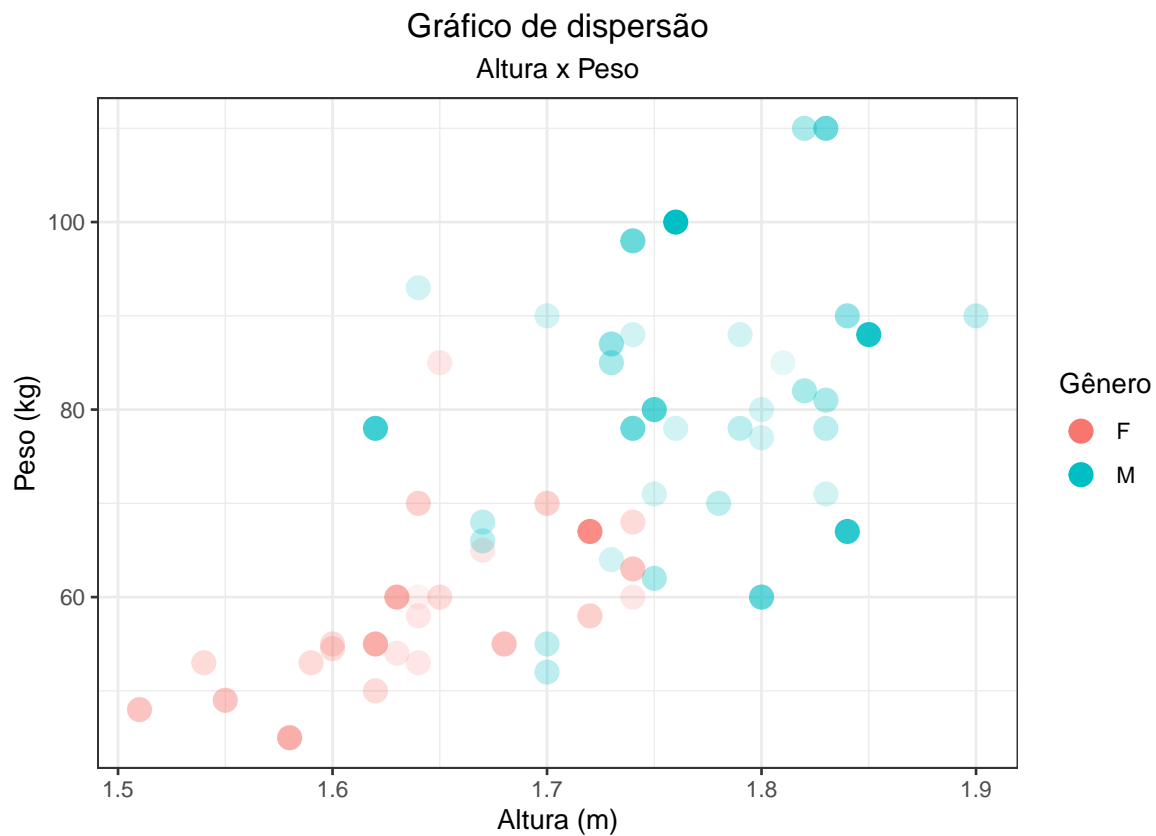
Nesse exemplo, ocultamos apenas a legenda da variável `alpha`, atribuindo o valor lógico `FALSE`.

Para saber mais sobre títulos, rótulos e aparência geral dos gráficos, confira os capítulos 9 e 10 do livro *R Graphics Cookbook*.

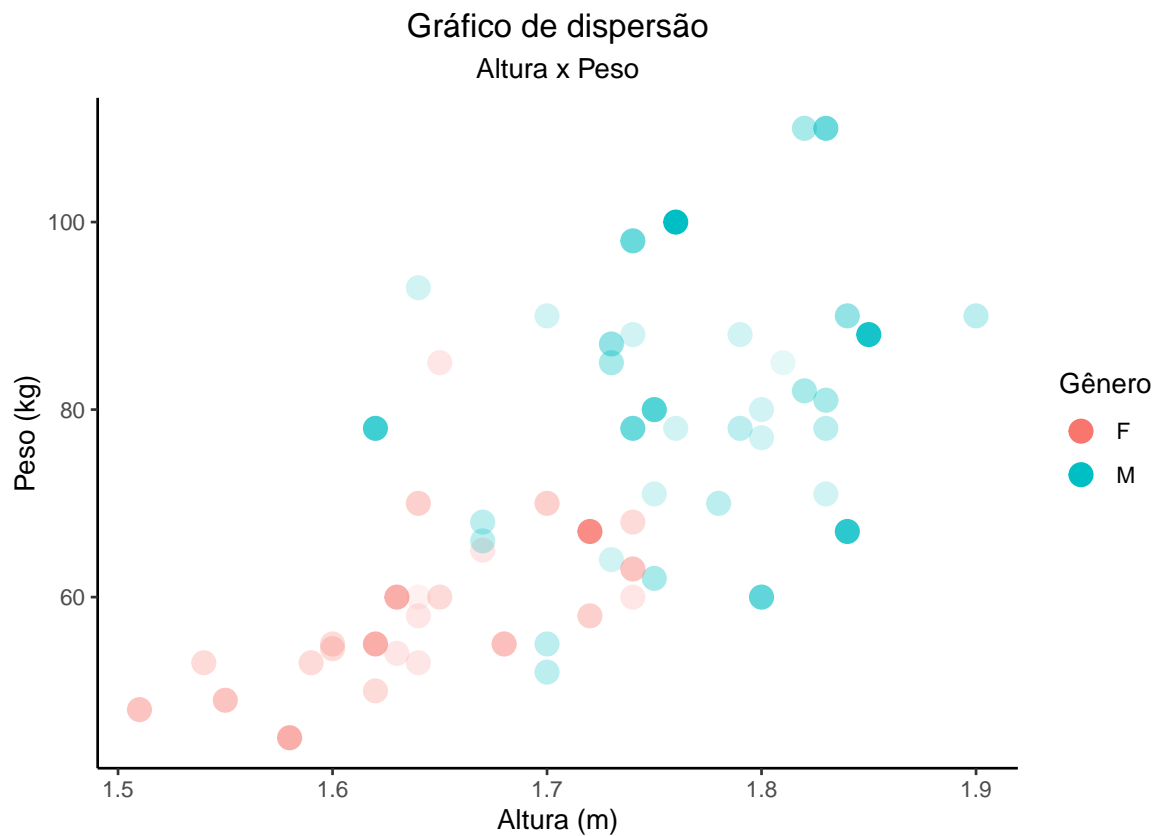
7.1.9 Temas

Existe a possibilidade de escolhermos outros temas para confeccionar nossos gráficos. Podemos criar temas a partir do zero ou senão utilizar aqueles presentes no `ggplot2` pela função `theme_`. A seguir demonstraremos alguns dos temas pré-configurados.

```
ggplot(data = dados_alunos,
       mapping = aes(x = altura,
                     y = peso,
                     color = sexo,
                     alpha = idade))+
  geom_point(size = 4)+
  labs(
    x = "Altura (m)",
    y = "Peso (kg)",
    color = "Gênero",
    alpha = "Idade (anos)",
    title = "Gráfico de dispersão",
    subtitle = "Altura x Peso",
    caption = "Fonte: Disciplina Estatística Aplicada/ESALQ."
  )+
  theme_bw()+
  theme(plot.title = element_text(hjust = 0.5),
        plot.subtitle = element_text(hjust = 0.5),
        plot.caption = element_text(hjust = 0))+
  guides(alpha = FALSE)
```

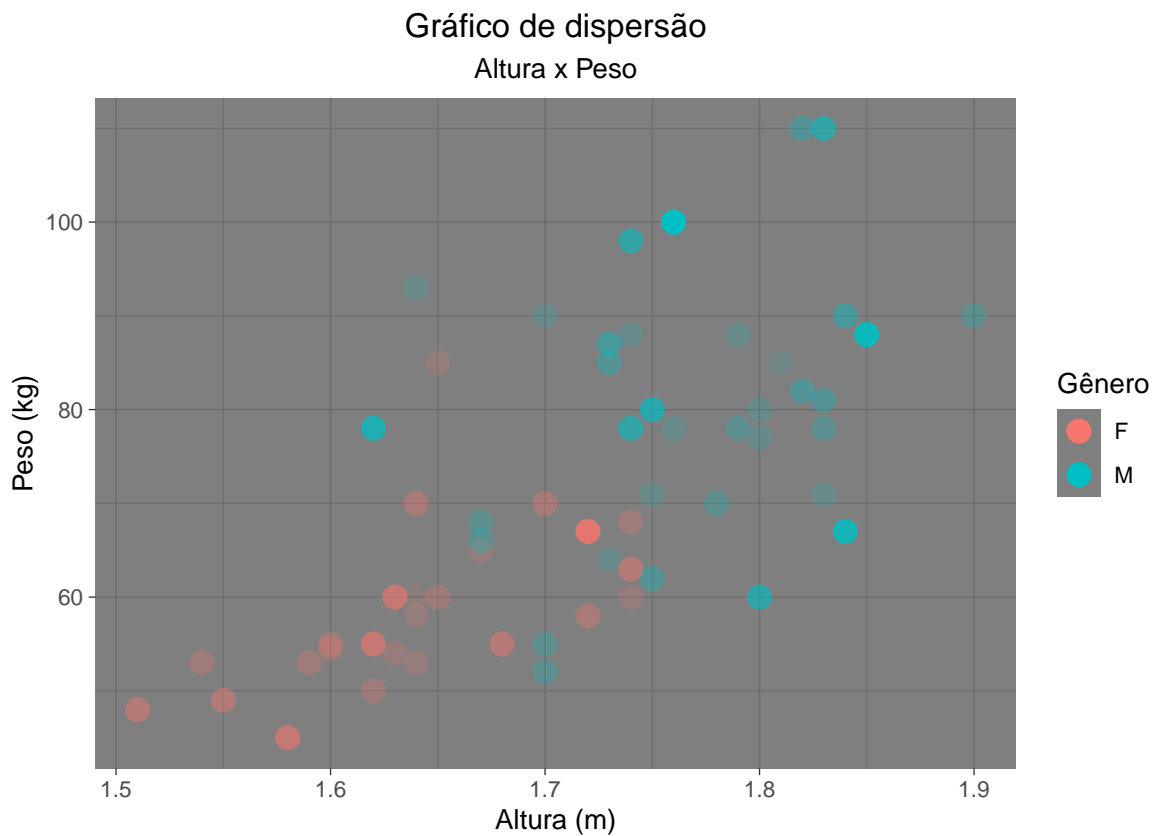


```
ggplot(data = dados_alunos,
       mapping = aes(x = altura,
                     y = peso,
                     color = sexo,
                     alpha = idade)) +
  geom_point(size = 4) +
  labs(
    x = "Altura (m)",
    y = "Peso (kg)",
    color = "Gênero",
    alpha = "Idade (anos)",
    title = "Gráfico de dispersão",
    subtitle = "Altura x Peso",
    caption = "Fonte: Disciplina Estatística Aplicada/ESALQ."
  ) +
  theme_classic() +
  theme(plot.title = element_text(hjust = 0.5),
        plot.subtitle = element_text(hjust = 0.5),
        plot.caption = element_text(hjust = 0)) +
  guides(alpha = FALSE)
```



Fonte: Disciplina Estatística Aplicada/ESALQ.

```
ggplot(data = dados_alunos,
       mapping = aes(x = altura,
                     y = peso,
                     color = sexo,
                     alpha = idade)) +
  geom_point(size = 4) +
  labs(
    x = "Altura (m)",
    y = "Peso (kg)",
    color = "Gênero",
    alpha = "Idade (anos)",
    title = "Gráfico de dispersão",
    subtitle = "Altura x Peso",
    caption = "Fonte: Disciplina Estatística Aplicada/ESALQ."
  ) +
  theme_dark() +
  theme(plot.title = element_text(hjust = 0.5),
        plot.subtitle = element_text(hjust = 0.5),
        plot.caption = element_text(hjust = 0)) +
  guides(alpha = FALSE)
```



7.2 Gráfico de Barras

Para a construção do gráfico de barras, dependendo do tipo de dados que queremos representar, podemos prosseguir a partir de duas vias. A primeira é utilizando a função `geom_col()`, sendo a outra, a `geom_bar()`. A seguir trataremos com detalhes ambas as funções.

- `geom_col()`

Utilizamos a função `geom_col()` para construir gráficos de barras em que indicamos uma variável categórica ao eixo x e uma variável quantitativa ao eixo y, sendo essa última, a que determinará a altura das barras. Para a elaboração dos exemplos, utilizaremos dados referentes à produção de milho e trigo, disponível para *download* [clcando aqui](#).

```
producao <- read_csv("dados_ggplot2/milho_trigo.csv")
```

```
producao
```

```
# A tibble: 8 x 3
  Local  Cultura Valor
<chr>   <chr>   <dbl>
```

| | | | |
|---|--------|-------|------|
| 1 | Brasil | Milho | 2.6 |
| 2 | Brasil | Trigo | 1.52 |
| 3 | China | Milho | 3.94 |
| 4 | China | Trigo | 3.07 |
| 5 | India | Milho | 1.62 |
| 6 | India | Trigo | 2.08 |
| 7 | EUA | Milho | 7.38 |
| 8 | EUA | Trigo | 2.48 |

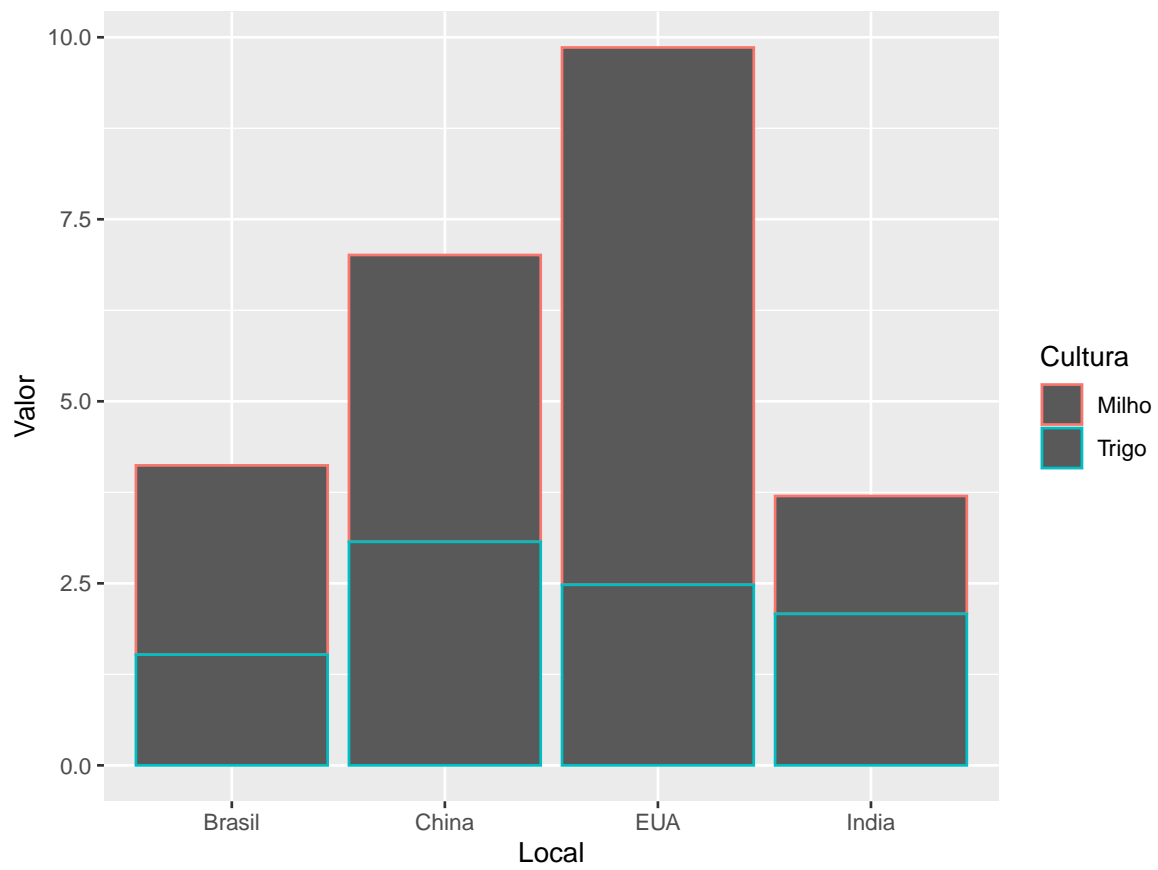
A base de dados apresenta 8 observações e 3 variáveis. A variável **Local** possui os países Brasil, China, Índia e Estado Unidos; a **Cultura** contém as culturas agrícolas milho e trigo; e **Valor** representa a produtividade média anual dos países, em toneladas por hectare, entre 1961 e 2019.

7.2.1 Cores

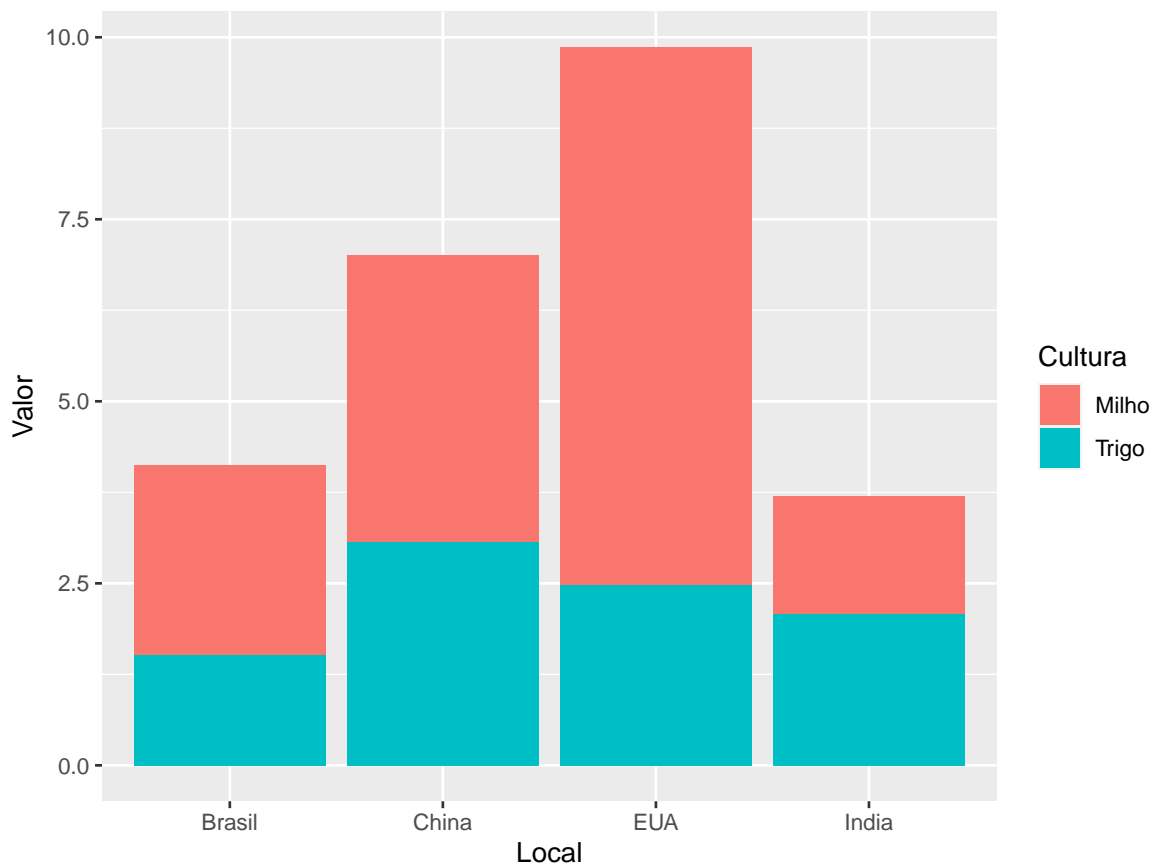
Nesse primeiro exemplo, queremos criar um gráfico de barras para representar a produção média de milho e trigo em cada um dos países. Para isso, utilizamos a função `geom_col()`. Perceba que, para diferenciarmos as porções da barra que representam as produções de cada cereal, podemos utilizar os argumentos estéticos `color = Item` e `fill = Item`.

O argumento `color` = somente colore as bordas das colunas, enquanto que o `fill` = preenche a barra com cores, de acordo os valores das variáveis milho e trigo.

```
# Argumento "color ="
ggplot(data = producao,
       aes(x = Local,
           y = Valor,
           color = Cultura))+
  geom_col()
```



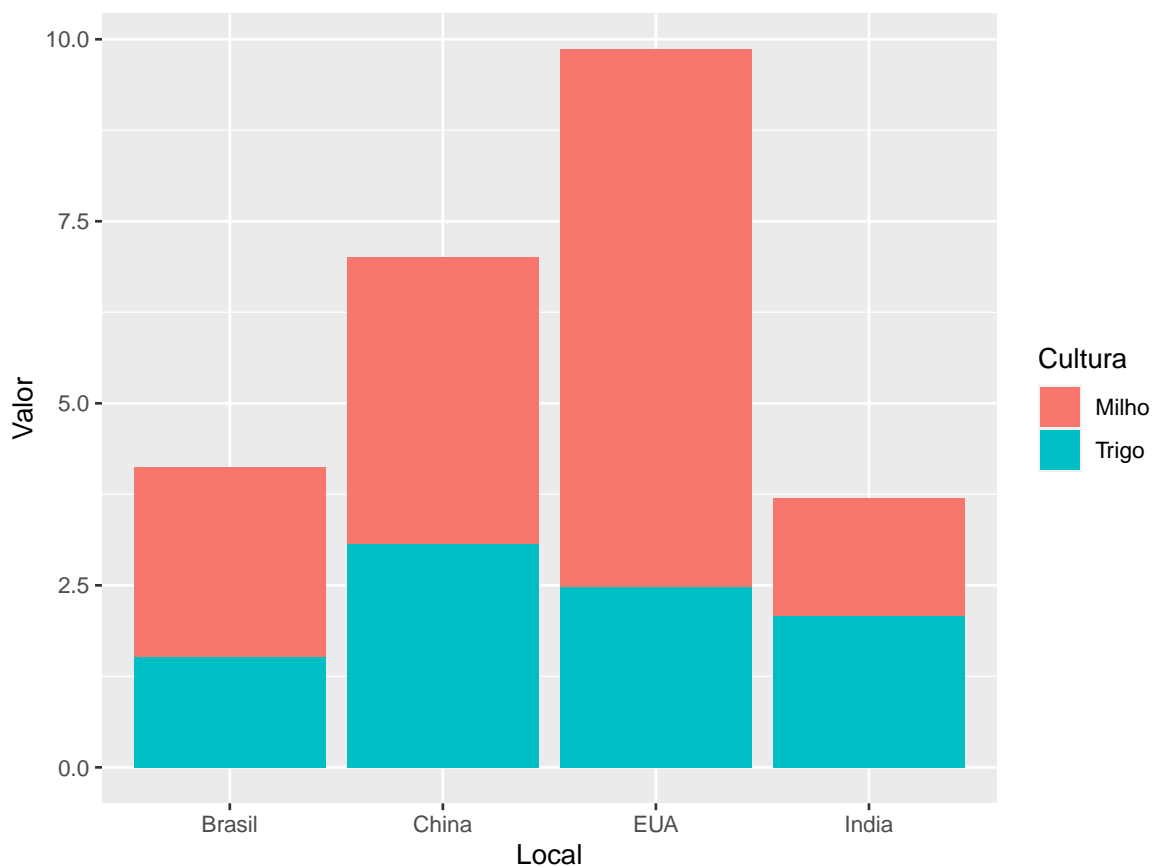
```
#Argumento "fill ="  
ggplot(data = producao,  
       aes(x = Local,  
           y = Valor,  
           fill = Cultura)) +  
  geom_col()
```



7.2.2 Posição

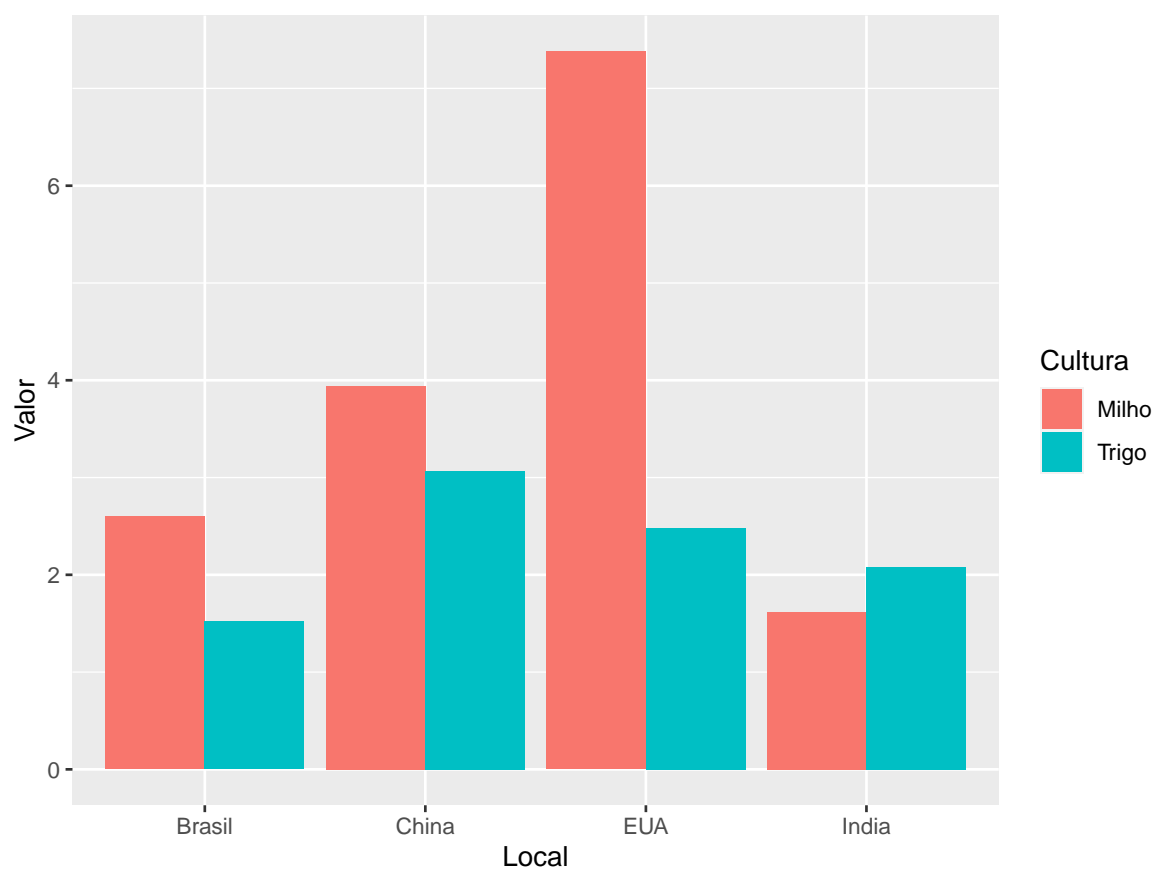
Caso queira alterar a disposição das colunas, podemos utilizar o argumento `position =` e especificar o ajuste desejado, tendo como opções: `"stack"`, `"dodge"` e `"fill"`.

```
# Posição "stack"
ggplot(data = producao,
       aes(x = Local,
           y = Valor,
           fill = Cultura)) +
  geom_col(position = "stack")
```



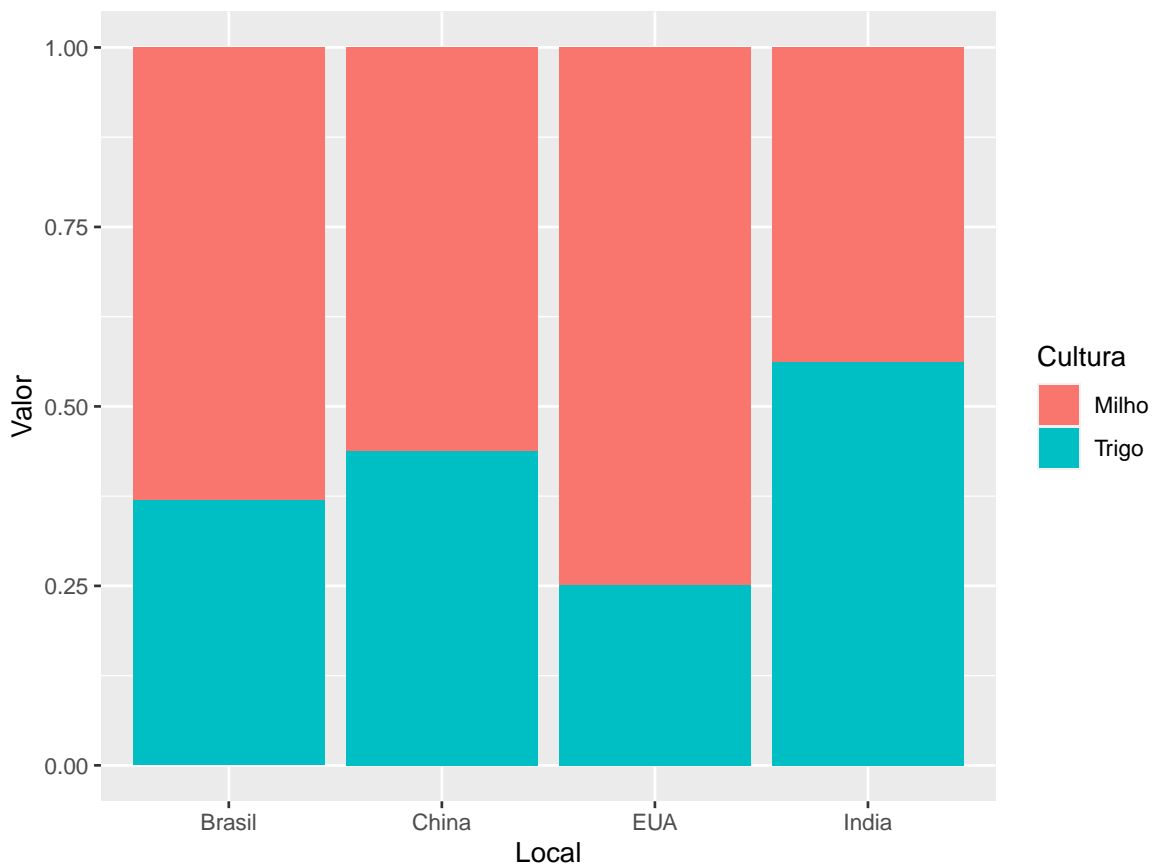
Perceba que a posição **"stack"** é a opção padrão do argumento **position =**, portanto, caso o argumento não seja especificado, a disposição do gráfico de colunas será a **"stack"**. Essa disposição de barras representa os valores absolutos de produtividade das culturas em apenas uma barra, de acordo com o país.

```
# Posição "dodge"
ggplot(data = producao,
       aes(x = Local,
           y = Valor,
           fill = Cultura)) +
  geom_col(position = "dodge")
```

Já a posição "dodge" representa as cultura em colunas separadas, de acordo com o país, ainda representando valores absolutos de produtividade.

```
# Posição "fill"
ggplot(data = producao,
       aes(x = Local,
           y = Valor,
           fill = Cultura)) +
  geom_col(position = "fill")
```

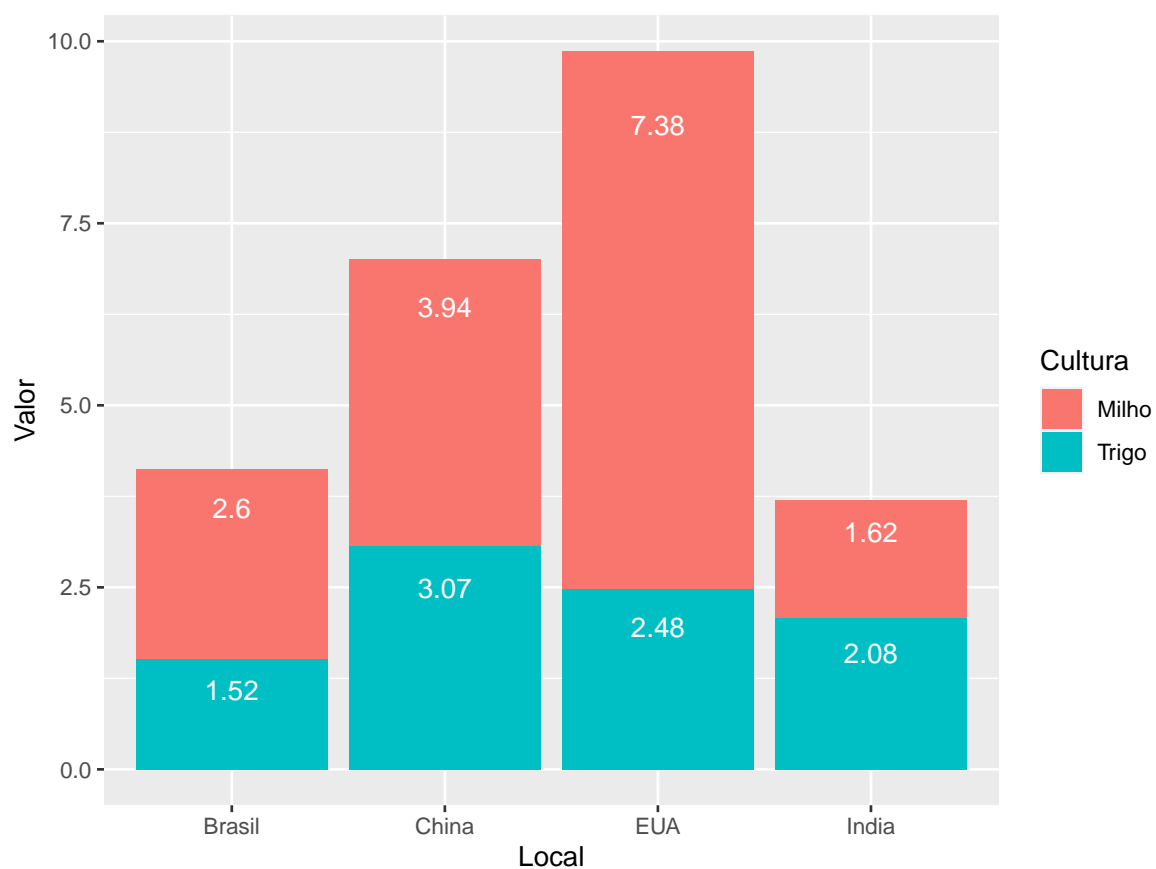


Por último, a posição "fill" constrói barras iguais, com escala de 0 a 1, preenchendo-as com os valores absolutos das produções de milho e trigo, de acordo com o país.

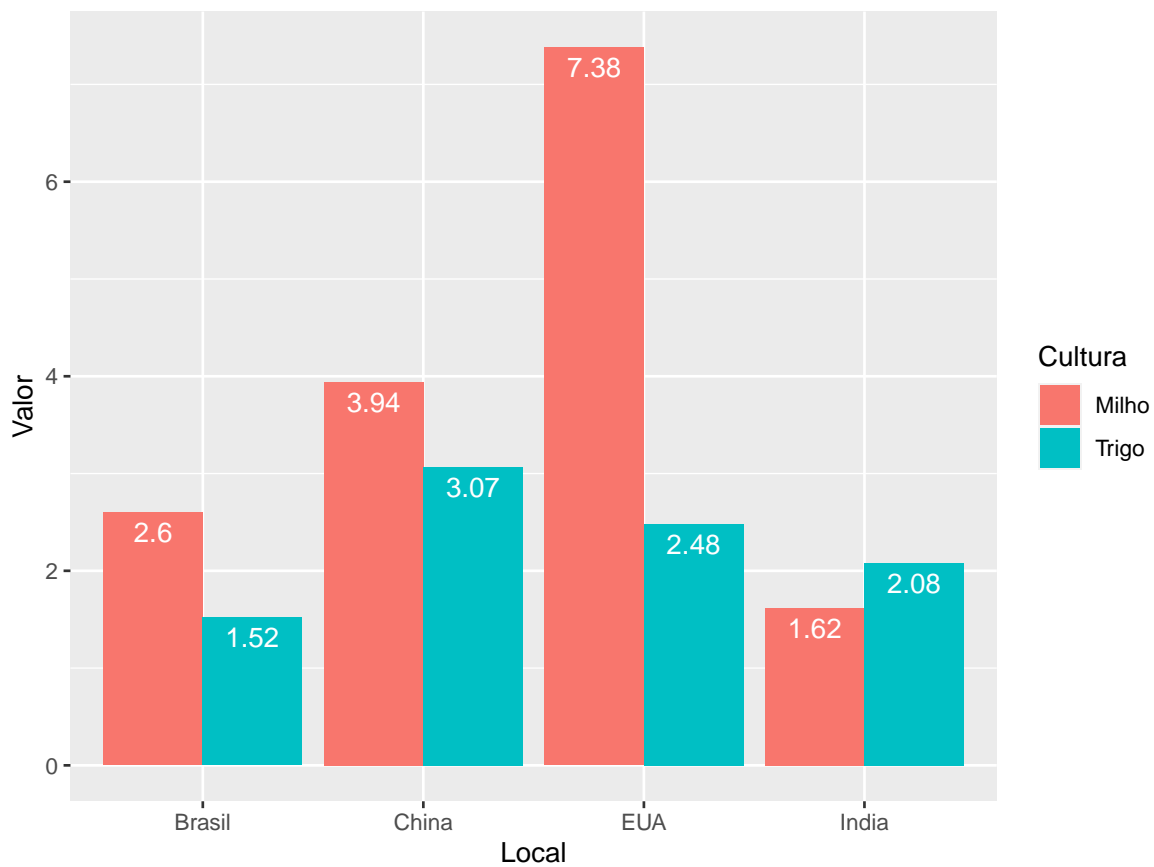
7.2.3 Legendas

Podemos adicionar legendas às nossas colunas, a partir da função `geom_text()`. Nos seguintes caso, demonstraremos como inserir as legendas nos três tipos de posição, indicando os valores de produtividades referentes a cada cultura e país.

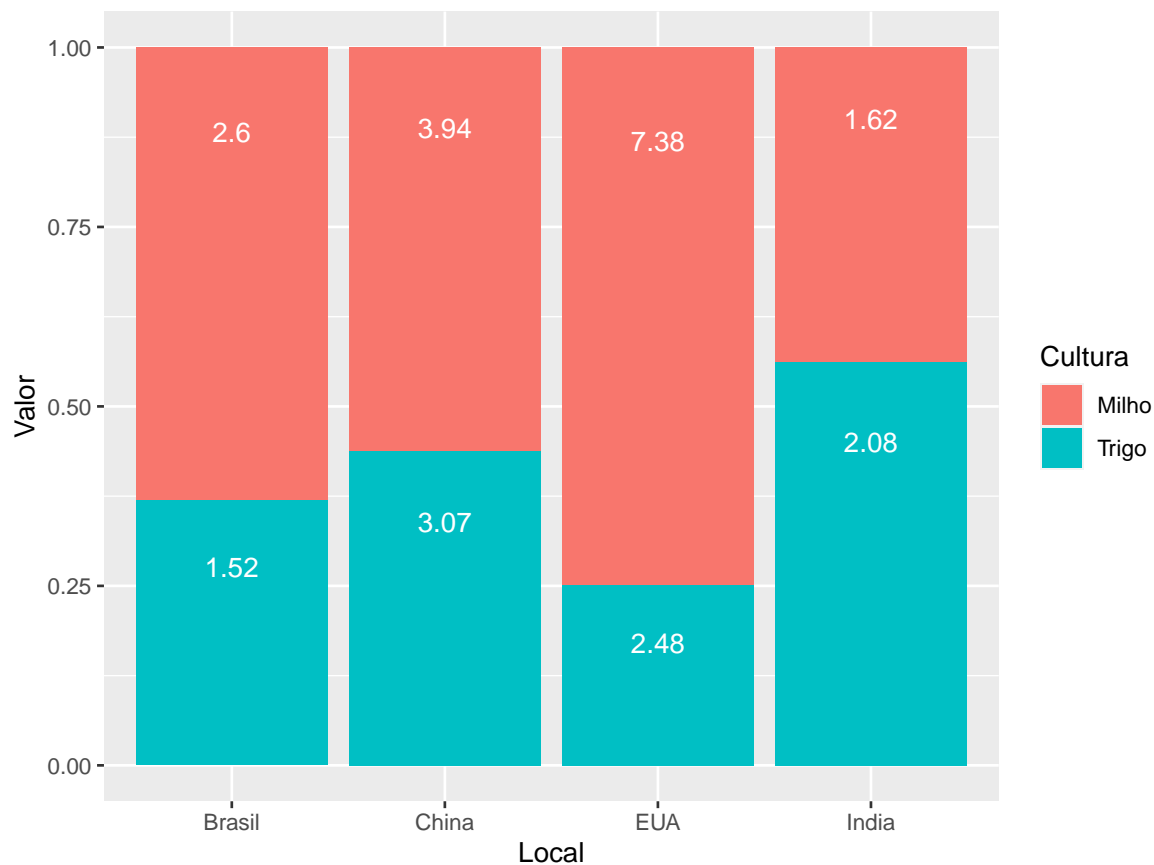
```
# Posição "stack"
ggplot(data = producao,
       aes(x = Local,
           y = Valor,
           fill = Cultura)) +
  geom_col(position = "stack") +
  geom_text(aes(label = Valor),
            color = "white",
            vjust = 1.5,
            position = position_stack(0.9))
```



```
# Posição "dodge"
ggplot(data = producao,
       aes(x = Local,
           y = Valor,
           fill = Cultura)) +
  geom_col(position = "dodge") +
  geom_text(aes(label = Valor),
            color = "white",
            vjust = 1.5,
            position = position_dodge(0.9))
```



```
# Posição "fill"
ggplot(data = producao,
       aes(x = Local,
           y = Valor,
           fill = Cultura)) +
  geom_col(position = "fill") +
  geom_text(aes(label = Valor),
           color = "white",
           vjust = 2.5,
           position = position_fill(0.9))
```



Note que a estrutura da função `geom_text()` é quase idêntica para as três posições, apenas alterando alguns valores e especificações. Primeiramente, definimos na função `aes()` a variável utilizada para ilustrar as legendas, no caso, a `Valor`. Em seguida, definimos a cor da legenda, sendo definida como branca (`color = "white"`). O argumento `vjust =` ajusta o posicionamento vertical das legendas, tendo como referência cada uma das barras; nesse caso, faça os ajustes testando valores, sendo possível atribuir valores negativos, fazendo com que a legenda suba. Por fim, a função `position_`, contida no argumento `position =`, ajusta o posicionamento das legendas em cada um dos setores da barra, de acordo com o tipo de posição adotada no `geom_col()`, seja o `stack` (`position_stack()`), `dodge` (`position_dodge()`) ou `fill` (`position_fill()`).

- `geom_bar()`

A função `geom_bar()` constrói gráficos de barras a partir da contagem de valores presentes em uma variável categórica. Como exemplo, utilizaremos a base de dados `diamonds`, presente no próprio pacote `ggplot2`.

```
diamonds
```

```
# A tibble: 53,940 x 10
```

```
  carat cut      color clarity depth table price      x      y      z
  <dbl> <ord>    <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
1  0.23 Ideal    E     SI2     61.5   55   326  3.95  3.98  2.43
```

```

2  0.21 Premium    E    SI1    59.8    61   326   3.89   3.84   2.31
3  0.23 Good      E    VS1    56.9    65   327   4.05   4.07   2.31
4  0.29 Premium    I    VS2    62.4    58   334   4.2    4.23   2.63
5  0.31 Good      J    SI2    63.3    58   335   4.34   4.35   2.75
6  0.24 Very Good J    VVS2    62.8    57   336   3.94   3.96   2.48
7  0.24 Very Good I    VVS1    62.3    57   336   3.95   3.98   2.47
8  0.26 Very Good H    SI1    61.9    55   337   4.07   4.11   2.53
9  0.22 Fair      E    VS2    65.1    61   337   3.87   3.78   2.49
10 0.23 Very Good H    VS1    59.4    61   338   4      4.05   2.39
# ... with 53,930 more rows

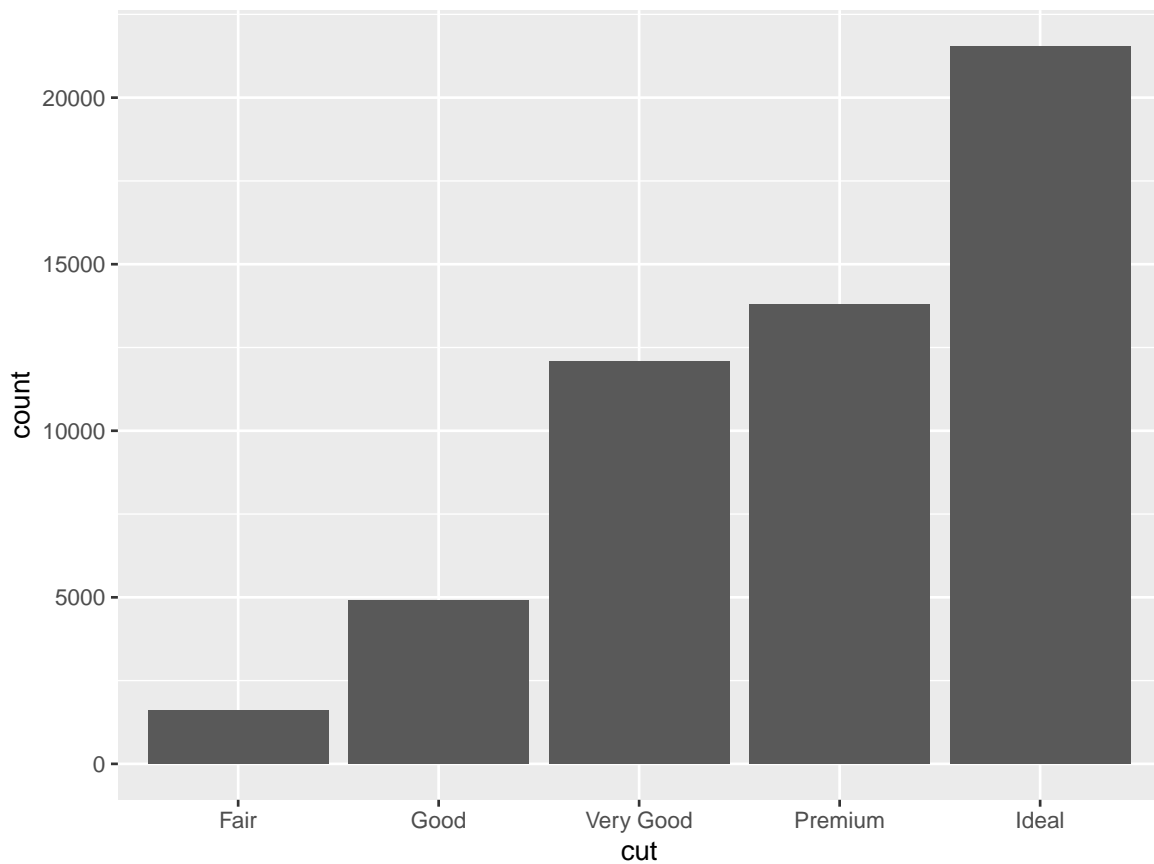
```

Essa *tibble* apresenta 53.940 observações e 10 variáveis, referentes às características de uma amostra de diamantes. Dessas variáveis, utilizaremos apenas a *cut*, que informa a qualidade de corte dos diamantes, sendo categorizadas em *Fair*, *Good*, *Very Good*, *Premium* e *Ideal*. Assim sendo, faremos um gráfico de barras que conta a quantidade de diamantes que se encaixam em cada uma dessas categorias.

```

ggplot(diamonds,
       aes(x = cut)) +
  geom_bar()

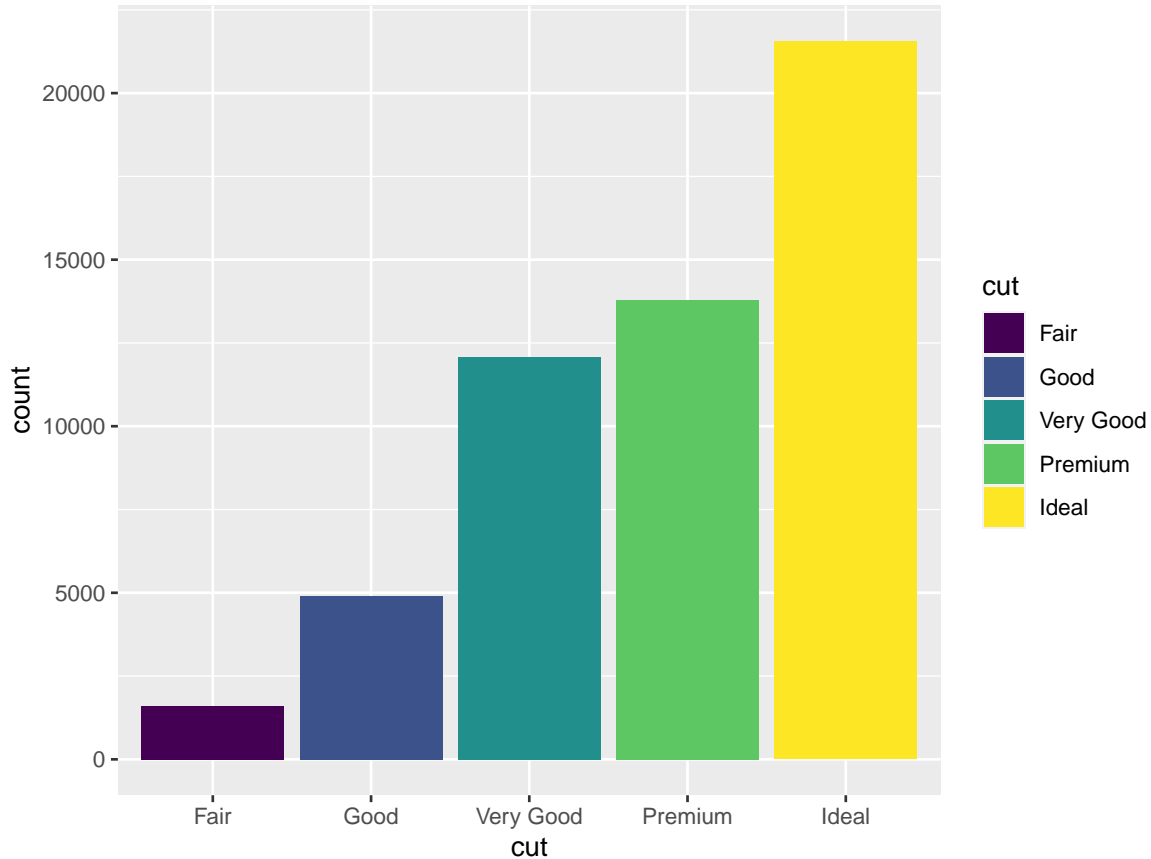
```



Perceba que foi preciso informar apenas a variável categórica *cut* no eixo x, sendo o y construído, automaticamente, a partir da contagem dos diamantes que se encaixam nas respectivas categorias.

Podemos melhorar a apresentação do gráfico atribuindo cores às categorias. Para isso, utilizamos o argumento `fill` na função `aes()`.

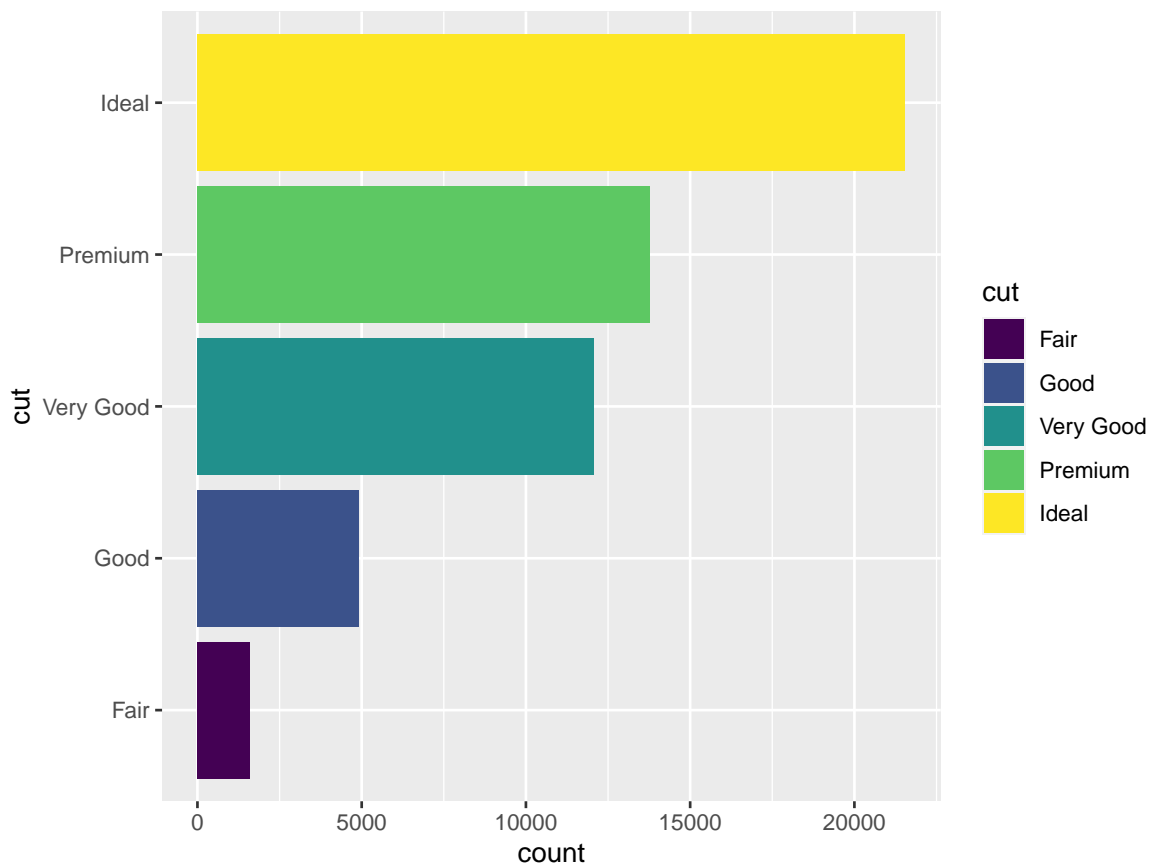
```
ggplot(diamonds,  
  aes(x = cut,  
      fill = cut))+  
  geom_bar()
```



7.2.4 Coordenadas

O sistema de coordenadas padrão do `ggplot2` é o cartesiano, onde os eixos `x` e `y` atuam de maneira independente para determinar a localização de cada ponto. De maneira simples, podemos inverter os eixos `x` e `y` utilizando a função `coord_flip()`, assim, dispomos as barras no sentido horizontal. O exemplo a seguir ilustra o caso.

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



Podemos utilizar outras coordenadas para contruir diferentes tipos de gráficos, como é o caso dos gráficos de setores, os quais veremos a seguir.

7.3 Gráfico de Setores (Pizza)

Mais conhecido como gráfico de pizza, esse tipo gráfico é muito popular e simples de ser compreendido. Apesar disso, sua utilização deve ser cautelosa para não sobrecarregar em informação ou utilizá-lo de maneira inadequada. Normalmente, um gráfico de pizza visa representar a frequência relativa de valores, de acordo com uma variável categórica.

Para construí-lo no `ggplot2`, utilizamos do artifício `coord_polar()`, o qual altera a coordenada do gráfico de barras. A seguir, veremos uma demonstração com dados hipotéticos sobre o grau de instrução de indivíduos.

```
educ <- tibble(
  Instrucao = c("Fundamental", "Médio", "Superior"),
  Quantidade = c(150, 200, 60),
  Porcentagem = round((Quantidade/sum(Quantidade)*100),2)
)

educ
```

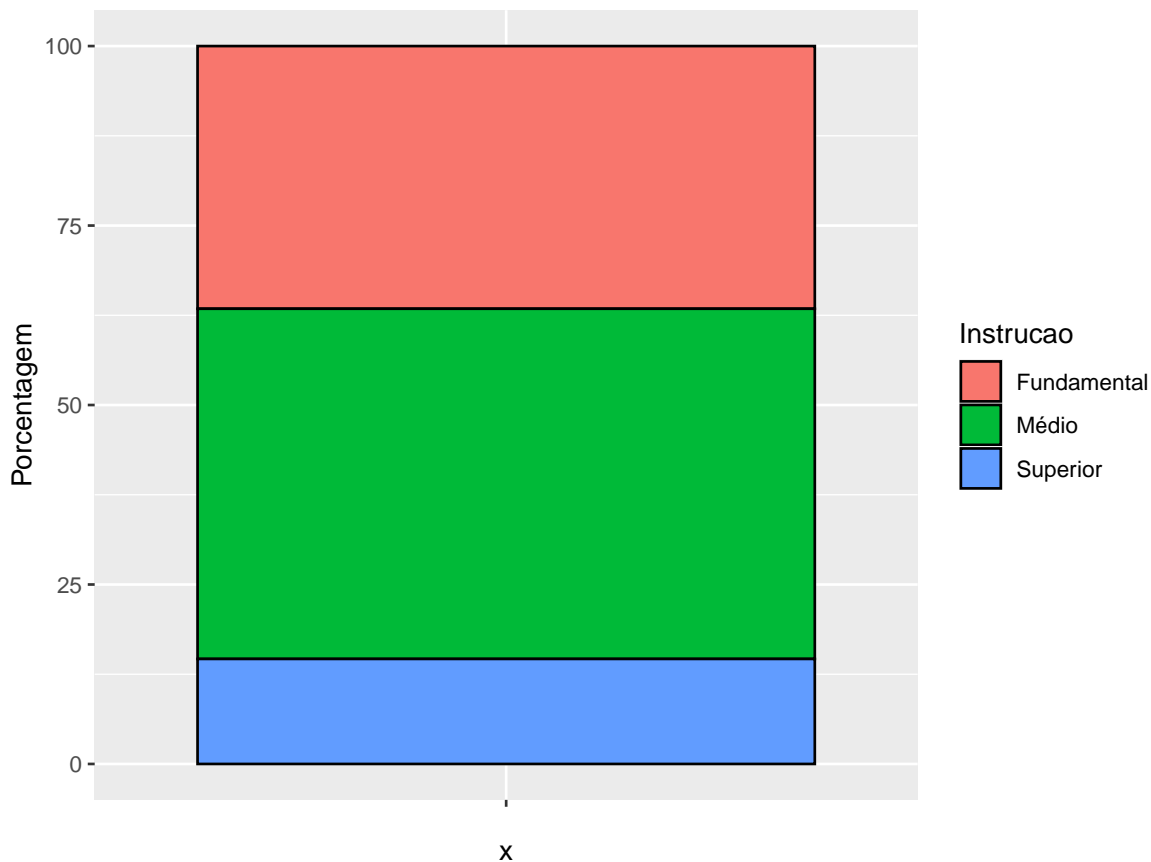


```
# A tibble: 3 x 3
  Instrucao  Quantidade Porcentagem
  <chr>      <dbl>      <dbl>
1 Fundamental    150      36.6
2 Médio         200      48.8
3 Superior       60      14.6
```

Assim, devemos proceder da seguinte maneira para construir o gráfico de setores:

1. Montar o gráfico de barras: devemos deixar vazio o eixo x (`x = ""`), definir a porcentagem no eixo y (`y = Porcentagem`) e preencher a barra com a variável categórica (`fill = Instrucao`). Feito isso, atribuímos a geometria de barras (`geom_col()`), colorindo suas bordas com a cor preta (`color = "black"`);

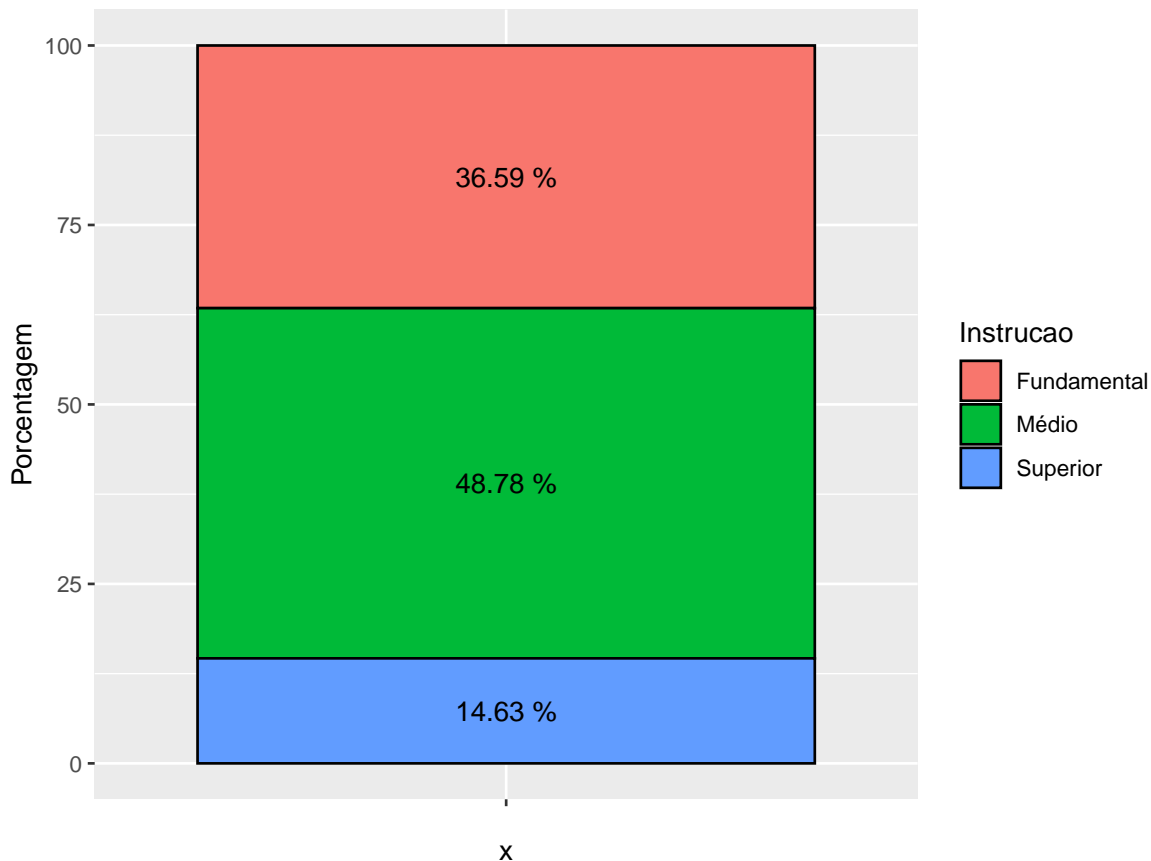
```
ggplot(data = educ,
       mapping = aes(x = "",
                     y = Porcentagem,
                     fill = Instrucao)) +
  geom_col(color = "black")
```



2. Legendas: utilizamos a `geom_text()` para definir a legenda das porcentagens dentro de cada setor da barra. Dentro da função `aes()`, definimos como rótulo (`label =`) a variável `Porcentagem`, sendo

que a função `paste(Porcentagem, "%")` insere o símbolo de % logo após os valores de porcentagem. Por último, o argumento `vjust = 0.5` ajusta a posição das legendas;

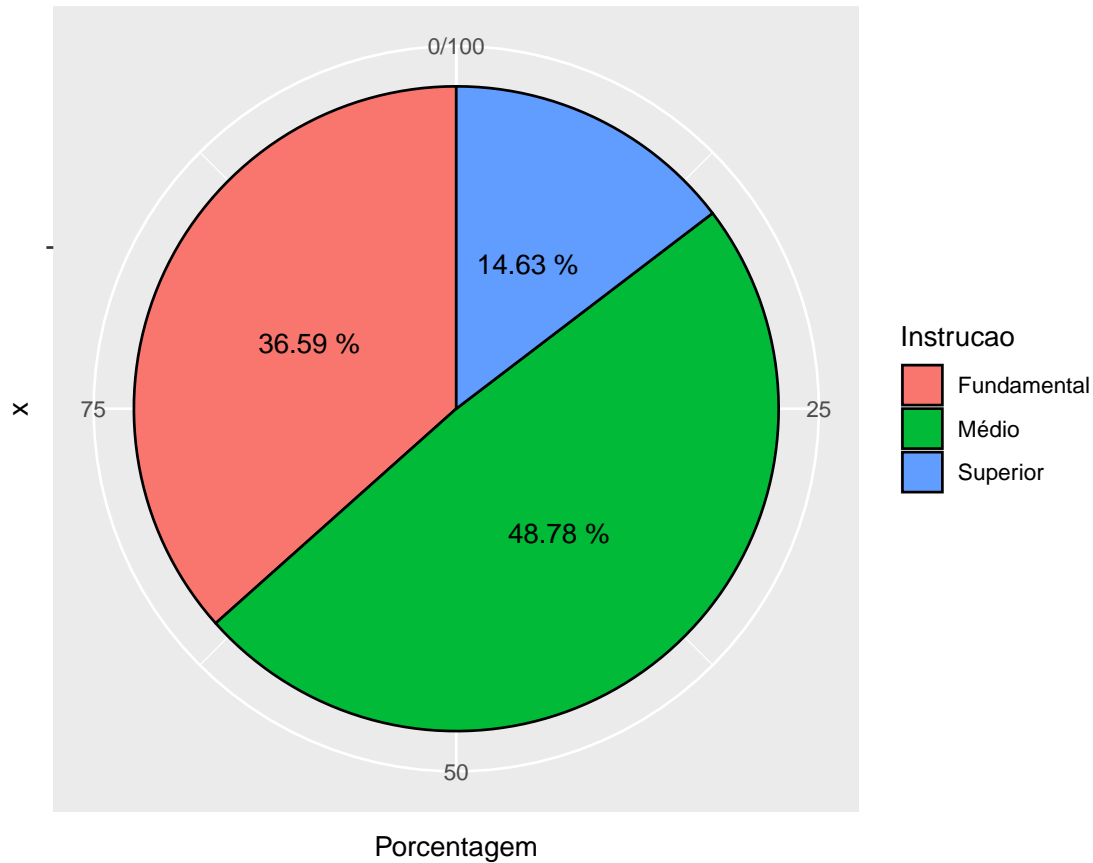
```
ggplot(data = educ,
       mapping = aes(x = "",
                     y = Porcentagem,
                     fill = Instrucao)) +
  geom_col(color = "black") +
  geom_text(aes(label = paste(Porcentagem, "%")),
           position = position_stack(vjust = 0.5))
```



3. Coordenada: nesse ponto, utilizamos a função `coord_polar()` para tornar nosso gráfico redondo. O argumento `theta = "y"` indica que o eixo y deve ser adotado como referência para a alteração da coordenada e o `start =` indica por qual valor o gráfico deve começar (teste outros valores para ver a diferença);

```
ggplot(data = educ,
       mapping = aes(x = "",
                     y = Porcentagem,
                     fill = Instrucao)) +
  geom_col(color = "black") +
  geom_text(aes(label = paste(Porcentagem, "%")),
```

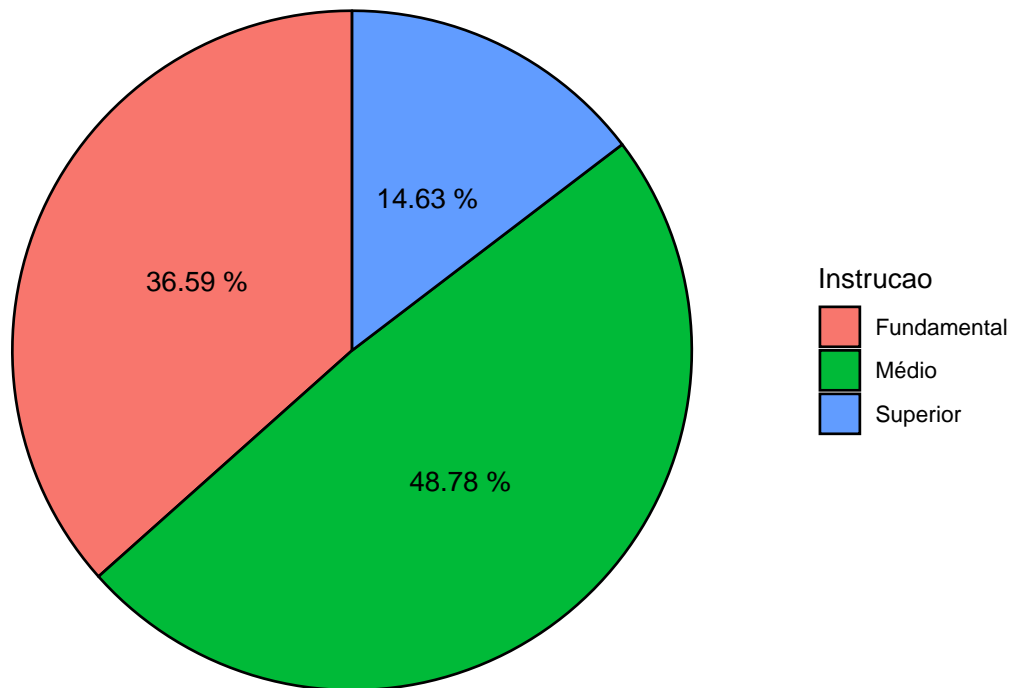
```
position = position_stack(vjust = 0.5))+
coord_polar(theta = "y",
            start = 0)
```



4. Retirar elementos: até agora, temos um gráfico de pizza, porém poluído devido a presença das escalas, nome dos eixos e cor de fundo inadequado. Para alterar o fundo cinza para um branco, escolhemos o tema `theme_minimal()`. Em seguida, na função `theme()`, atribuímos a alguns argumentos o `element_blank()`, ou seja, função que retira os elementos de cena. Portanto, o `axis.title = element_blank()` retira os nomes dos eixos, o `axis.text = element_blank()`, exclui o restante de texto presente no gráfico e o `panel.grid = element_blank()` omite o restante das linhas gráficas.

```
ggplot(data = educ,
       mapping = aes(x = "",
                     y = Porcentagem,
                     fill = Instrucao))+
  geom_col(color = "black")+
  geom_text(aes(label = paste(Porcentagem, "%")),
            position = position_stack(vjust = 0.5))+
  coord_polar(theta = "y",
            start = 0)+
  theme_minimal()+
  theme(
```

```
axis.title = element_blank(),
axis.text = element_blank(),
panel.grid = element_blank()
)
```



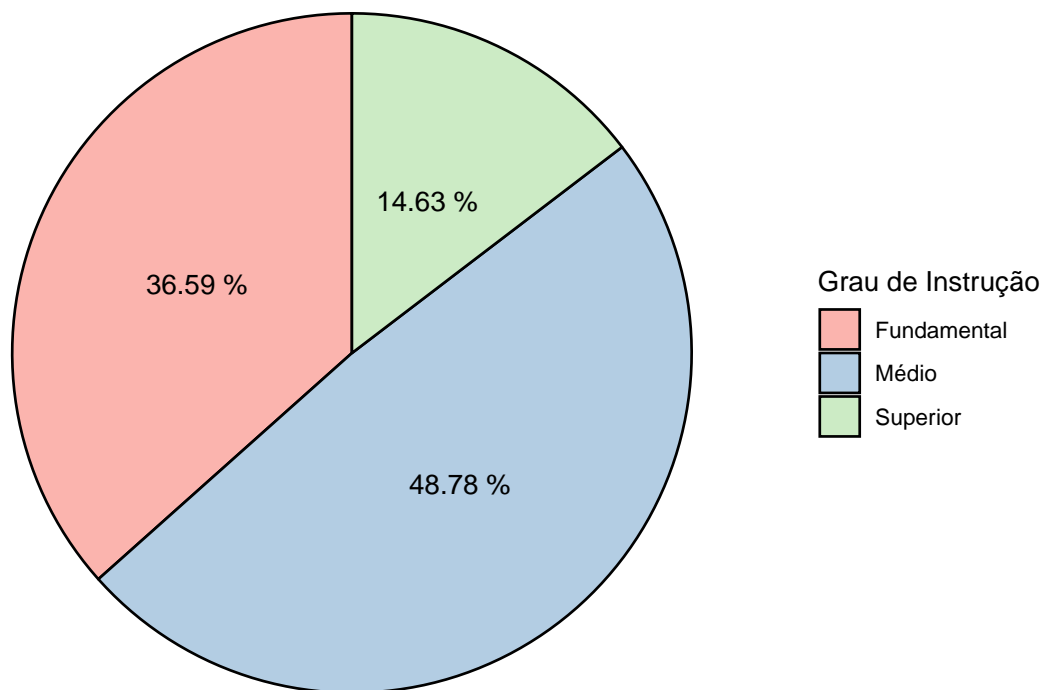
5. Personalização: agora temos um gráfico de pizza autêntico. Podemos realizar mais algumas modificações estéticas, como definir uma paleta de cores com a `scale_fill_brewer()` e ajustar o nome na legenda com o `labs()`.

```
ggplot(data = educ,
       mapping = aes(x = "",
                     y = Porcentagem,
                     fill = Instrucao)) +
  geom_col(color = "black") +
  geom_text(aes(label = paste(Porcentagem, "%")),
            position = position_stack(vjust = 0.5)) +
  coord_polar(theta = "y",
              start = 0) +
  theme_minimal() +
  theme(
    axis.title = element_blank(),
    axis.text = element_blank(),
```

```

panel.grid = element_blank()
)+
scale_fill_brewer(palette = "Pastel1")+
labs(fill = "Grau de Instrução")

```



7.4 Gráfico de Linhas

Os gráficos de linhas são muito utilizados para representar séries temporais, ou seja, a progressão de valores ao longo do tempo. Utilizamos a função `geom_line()` para construí-los. Para exemplificação, utilizaremos os dados de produtividade de milho entre 1961 e 2019. Faça o *download* dos dados [clicando aqui](#).

```

library(readr)
produtiv_milho <- read_csv("dados_ggplot2/produtiv_milho.csv")

produtiv_milho

```

```

# A tibble: 236 x 3
  Local    Ano Valor
  <chr>   <dbl> <dbl>

```

```

1 Brasil 1961 1.31
2 Brasil 1962 1.30
3 Brasil 1963 1.31
4 Brasil 1964 1.16
5 Brasil 1965 1.38
6 Brasil 1966 1.31
7 Brasil 1967 1.38
8 Brasil 1968 1.34
9 Brasil 1969 1.31
10 Brasil 1970 1.44
# ... with 226 more rows

```

A base de dados apresenta 236 observações e 3 variáveis. A variável `Local` possui os países Brasil, China, Índia e Estado Unidos; a `Ano` dispõe de dados entre 1961 e 2019; e `Valor` representa a produtividade da cultura do milho, em toneladas por hectare.

Para o primeiro exemplo, utilizaremos apenas os dados referentes ao Brasil. Para isso, utilizaremos a função `dplyr::filter`, do pacote `dplyr`.

```

produtiv_br <- produtiv_milho %>%
  filter(Local == "Brasil")

produtiv_br

```

```

# A tibble: 59 x 3
   Local    Ano Valor
   <chr> <dbl> <dbl>
1 Brasil 1961 1.31
2 Brasil 1962 1.30
3 Brasil 1963 1.31
4 Brasil 1964 1.16
5 Brasil 1965 1.38
6 Brasil 1966 1.31
7 Brasil 1967 1.38
8 Brasil 1968 1.34
9 Brasil 1969 1.31
10 Brasil 1970 1.44
# ... with 49 more rows

```

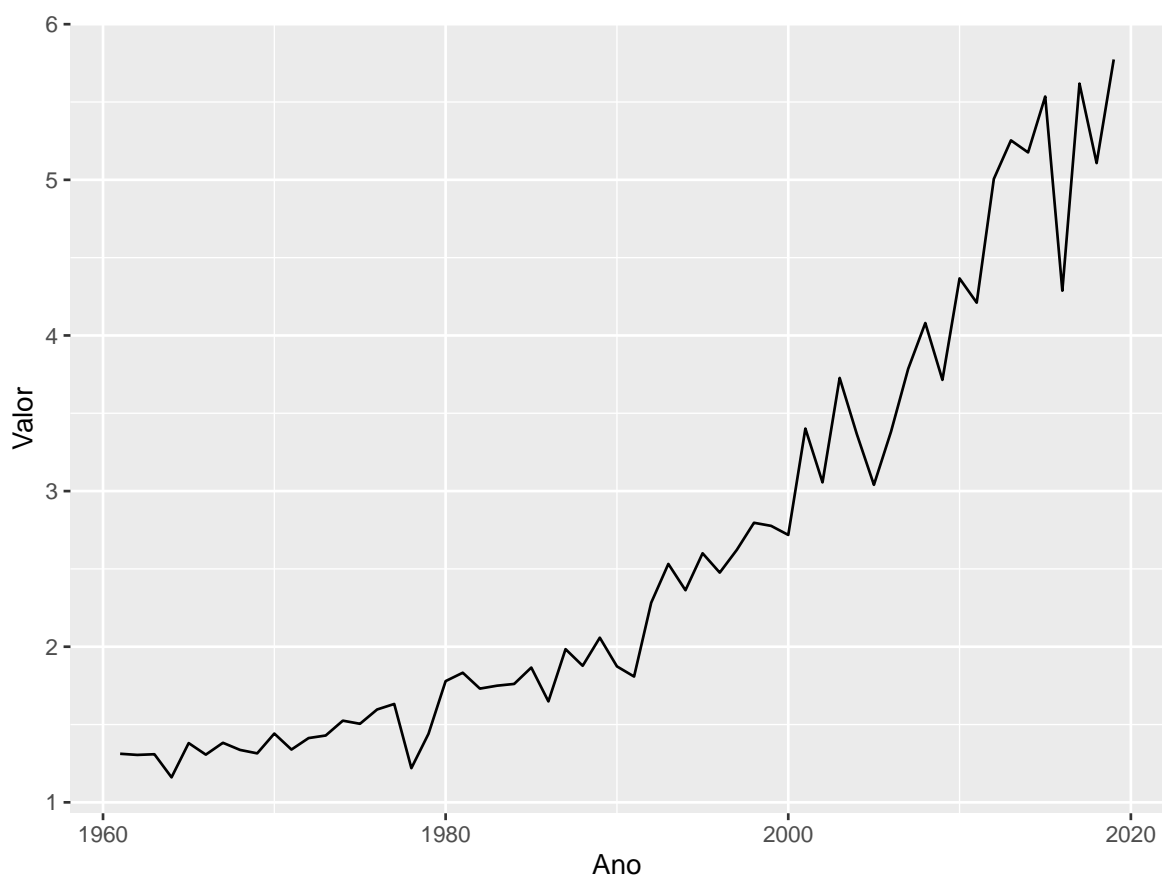
Portanto, o objeto `produtiv_br` possui apenas o país Brasil, apresentando 59 observações e 3 variáveis.

Para construir o gráfico de linhas, atribuiremos ao eixo x a variável `Ano` e ao eixo y, a variável `Valor`, além de definir a geometria de linha, ou seja, a `geom_line()`.

```

ggplot(produtiv_br)+
  geom_line(aes(x = Ano,
                y = Valor))

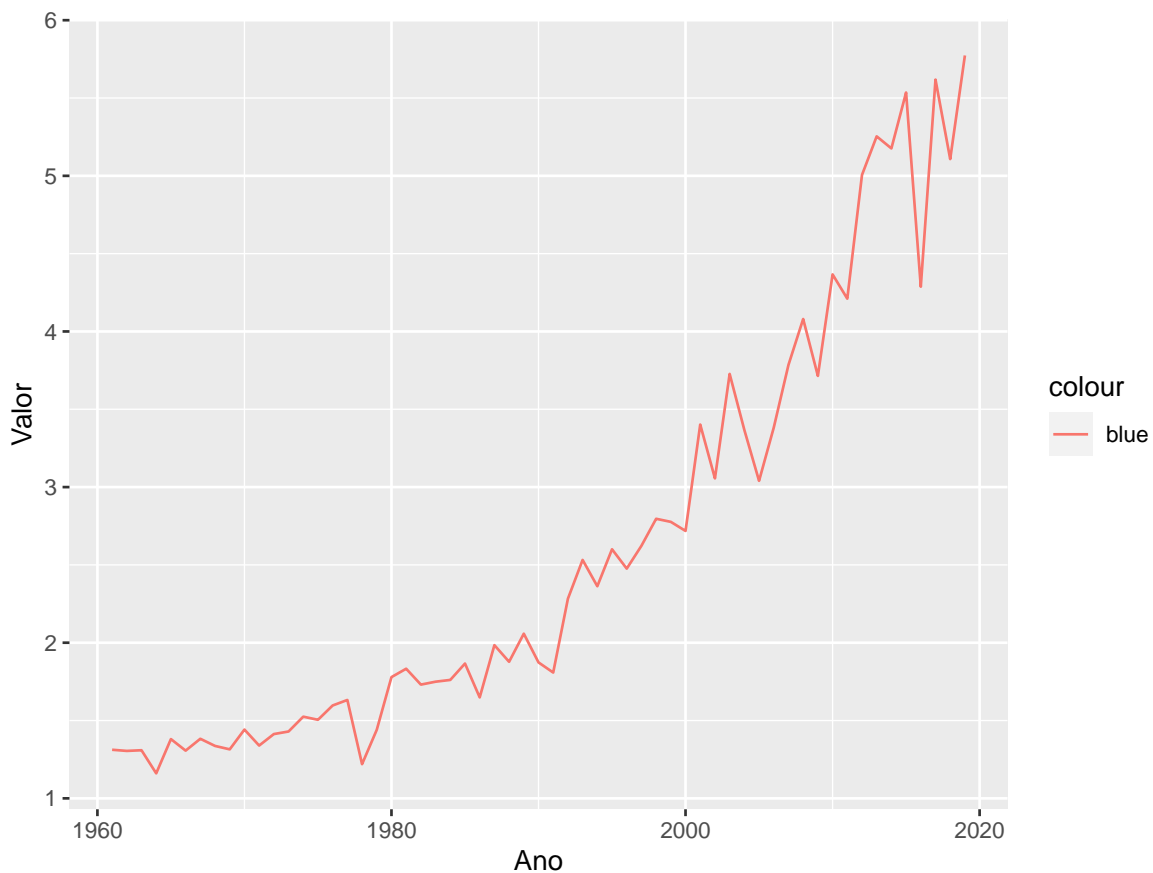
```



7.4.1 Cores

Podemos definir a cor da linha de maneira manual, utilizando o argumento `color =`. Porém, devemos nos atentar a alguns detalhes.

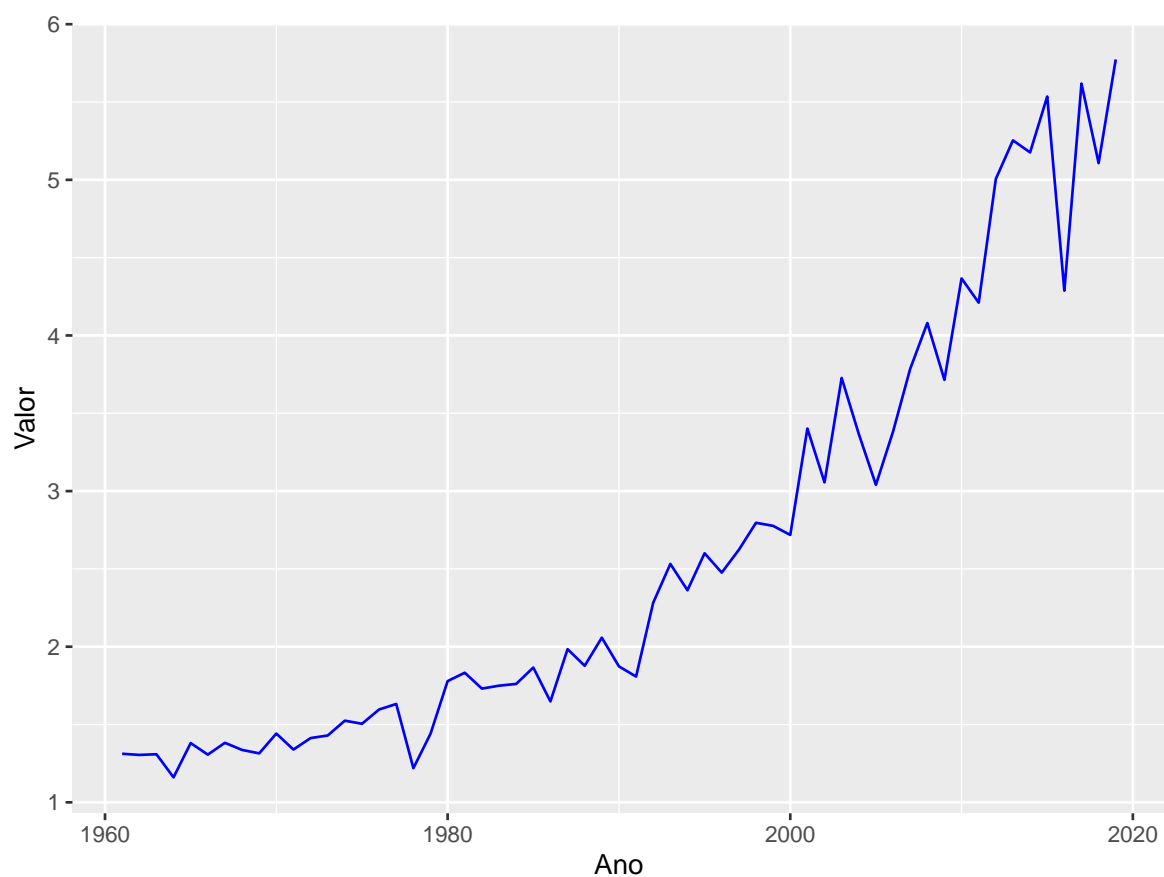
```
ggplot(produktiv_br)+  
  geom_line(aes(x = Ano,  
                y = Valor,  
                color = "blue"))
```



Perceba que o argumento `color = "blue"` nos retornou uma linha de coloração vermelha e não azul. Isto aconteceu pois o argumento foi colocado dentro da função `aes()` e esta espera uma coluna do banco de dados para mapear, assim, o valor `"blue"` é tratado como uma nova variável pertencente a todas as observações. Portanto, a linha é colorida de vermelho (padrão do `ggplot2`) associada à nova categoria `"blue"`.

Portanto, para colorirmos a linha de azul, devemos colocar o atributo `color =` fora da função `aes()`.

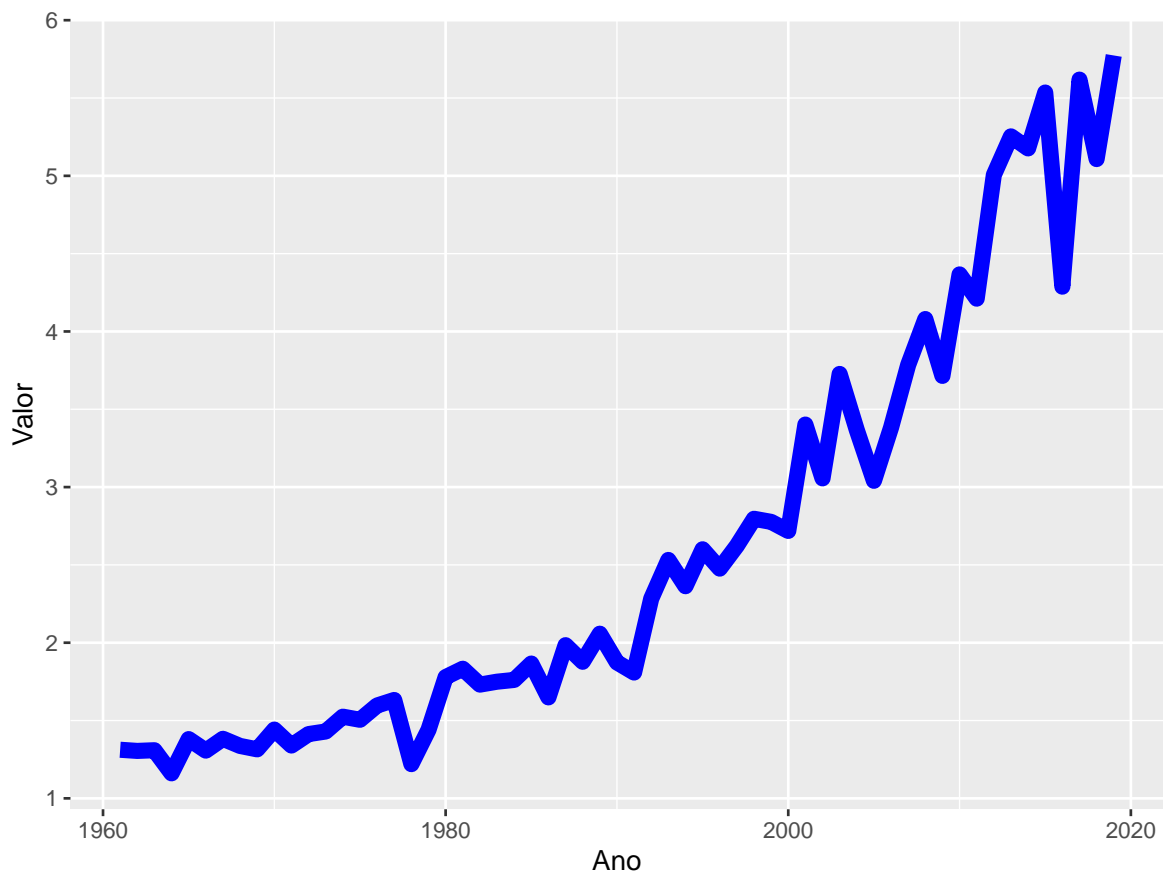
```
ggplot(produtiv_br)+  
  geom_line(aes(x = Ano,  
                y = Valor),  
            color = "blue")
```

7.4.2 Tamanho

Para alterarmos o tamanho da linha, utilizamos o argumento `size =`. Agora que aprendemos com o exemplo anterior, devemos colocá-lo fora da função `aes()`.

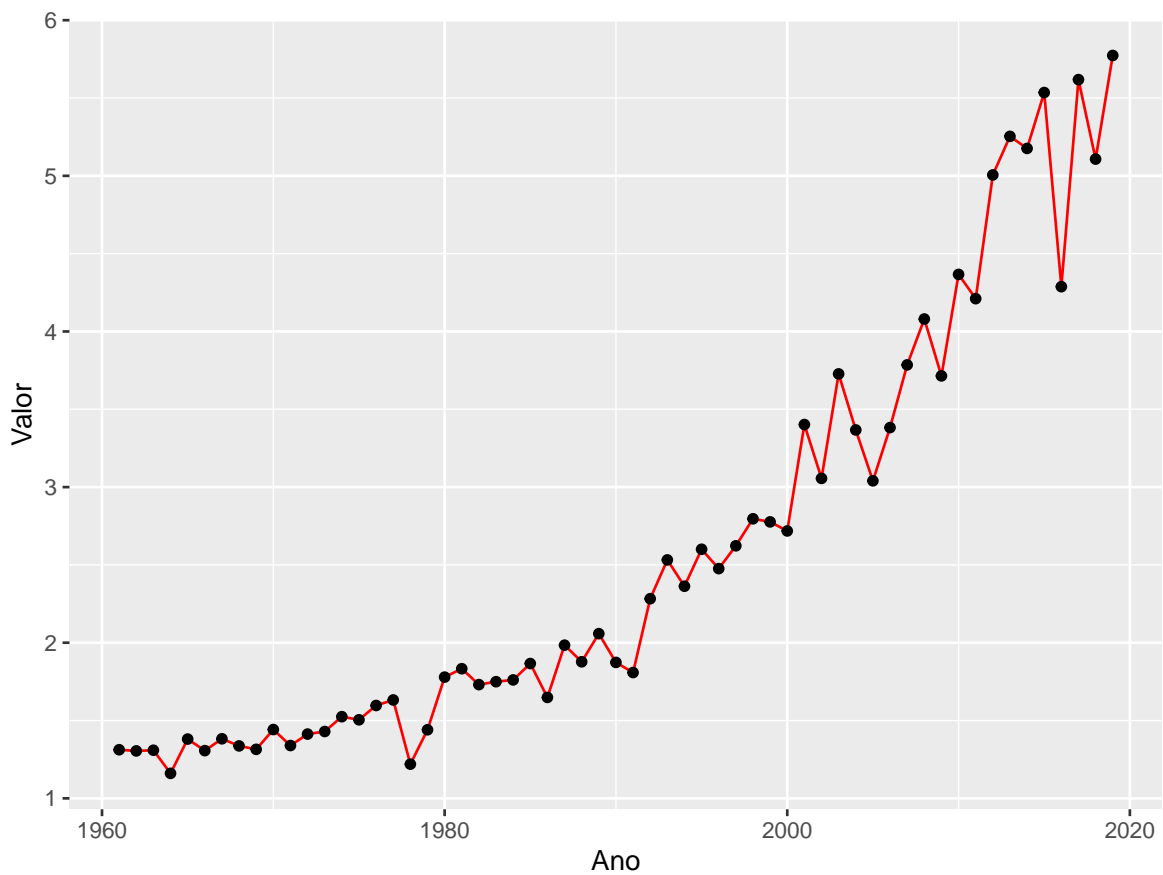
```
ggplot(produtiv_br)+  
  geom_line(aes(x = Ano,  
                y = Valor),  
            color = "blue",  
            size = 3)
```



7.4.3 Geometrias

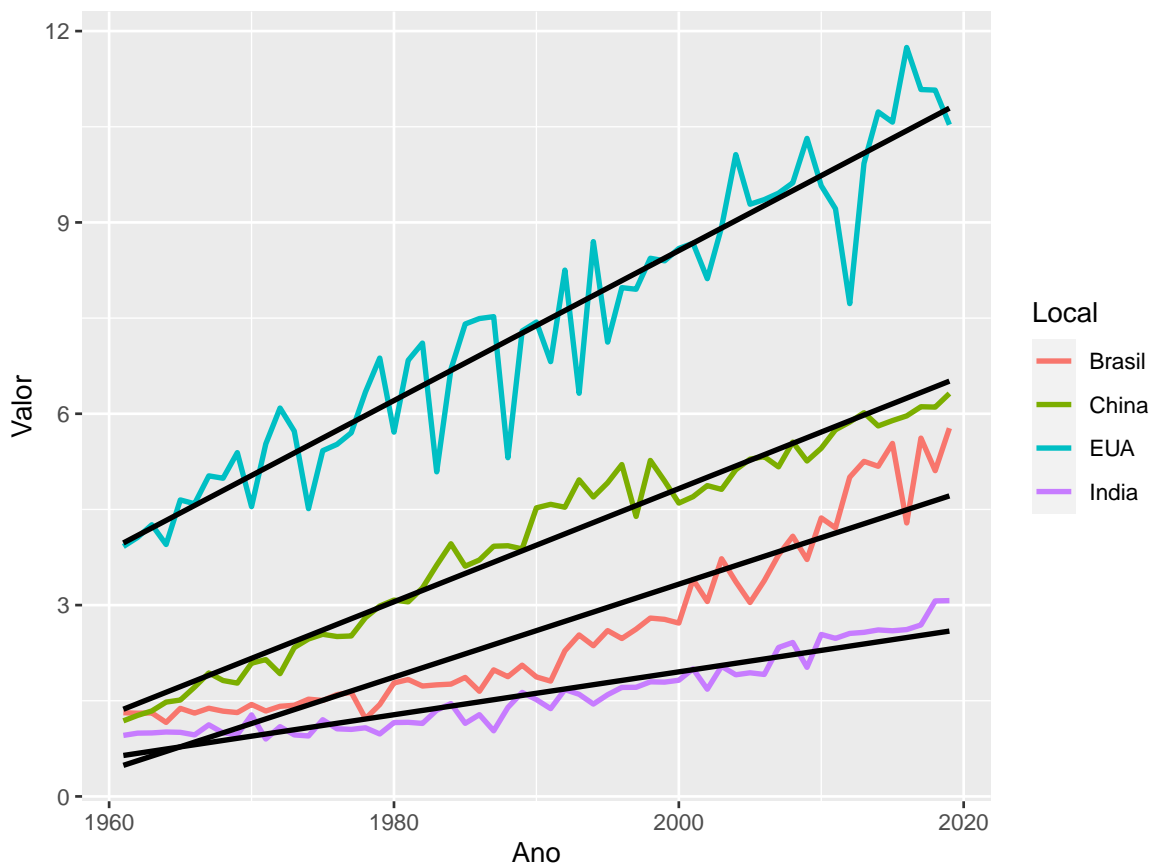
Nos gráficos de linhas, podemos mesclar diversas geometrias. A seguir, demonstraremos alguns exemplos.

```
ggplot(produtiv_br,  
  aes(x = Ano,  
      y = Valor))+  
  geom_line(color = "red")+  
  geom_point()
```



Aqui, podemos ver a associação do gráfico de linhas com o gráfico de pontos. Perceba que a estética (`aes()`) foi definida na função `ggplot()`, servindo tanto para o `geom_line()`, como para o `geom_point()`.

```
ggplot(produtiv_milho,
  aes(x = Ano,
      y = Valor))+
  geom_line(aes(color = Local),
    size = 1)+
  geom_smooth(aes(group = Local),
    color = "black",
    method = "lm",
    se = FALSE)
```

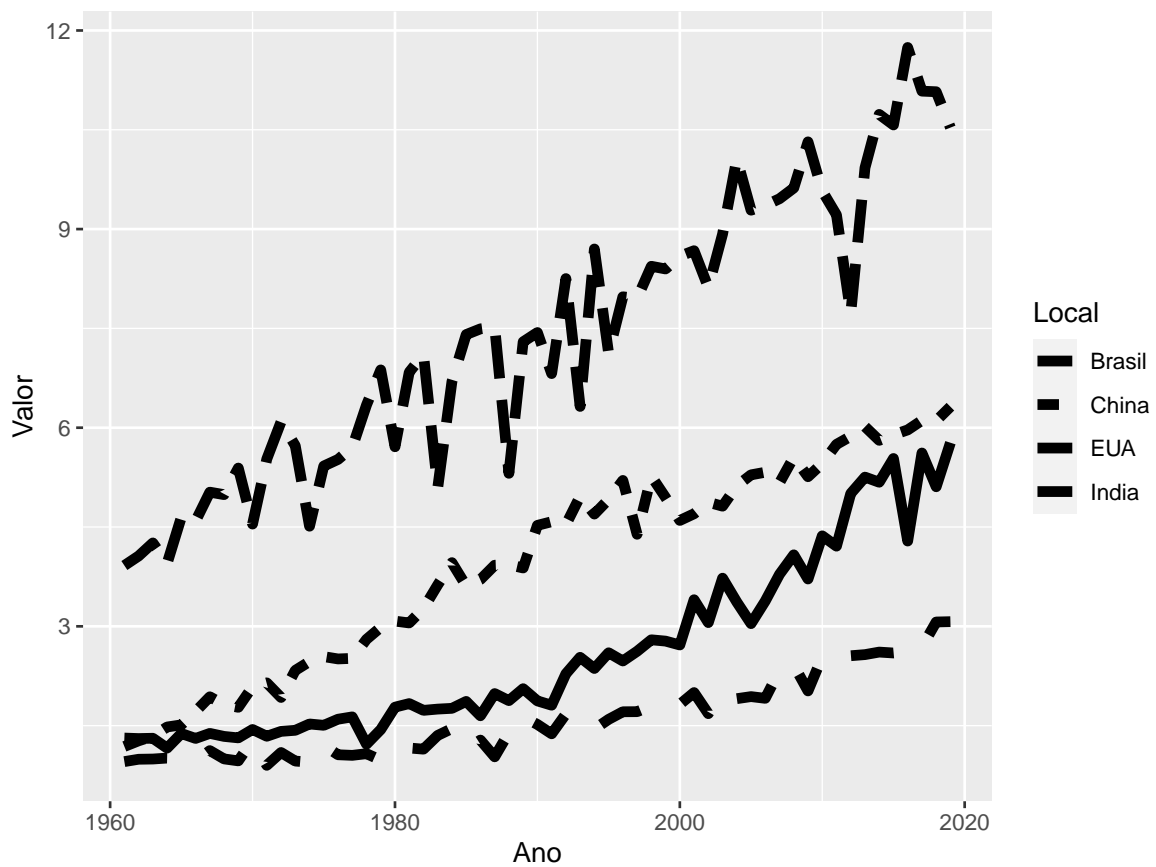


Agora, utilizando a base de dados `produtiv_milho`, unimos os quatro países em um mesmo gráfico. Designamos a variável `Local` (ou seja, os países) ao argumento `color =` para distingui-los com cores diferentes. Por se tratar de uma variável do nosso banco de dados, colocamos o argumento dentro da função `aes()`, contida na `geom_line()`. A outra camada geométrica é referente a reta de regressão, onde agrupamos a variável `Local` para que fosse possível traçar uma linha de regressão linear para cada variável.

7.4.4 Formatos

Também podemos diferenciar variáveis pelo formato das linhas, aplicando o argumento `linetype =` dentro da função `aes()`.

```
ggplot(produtiv_milho,
  aes(x = Ano,
      y = Valor,
      linetype = Local))+
  geom_line(size = 2)
```



Ademais, poderíamos associar formatos e cores em um mesmo gráfico, a fim de diferenciar as variáveis.

```
ggplot(produktiv_milho,  
  aes(x = Ano,  
      y = Valor,  
      color = Local,  
      linetype = Local)) +  
  geom_line(size = 2)
```



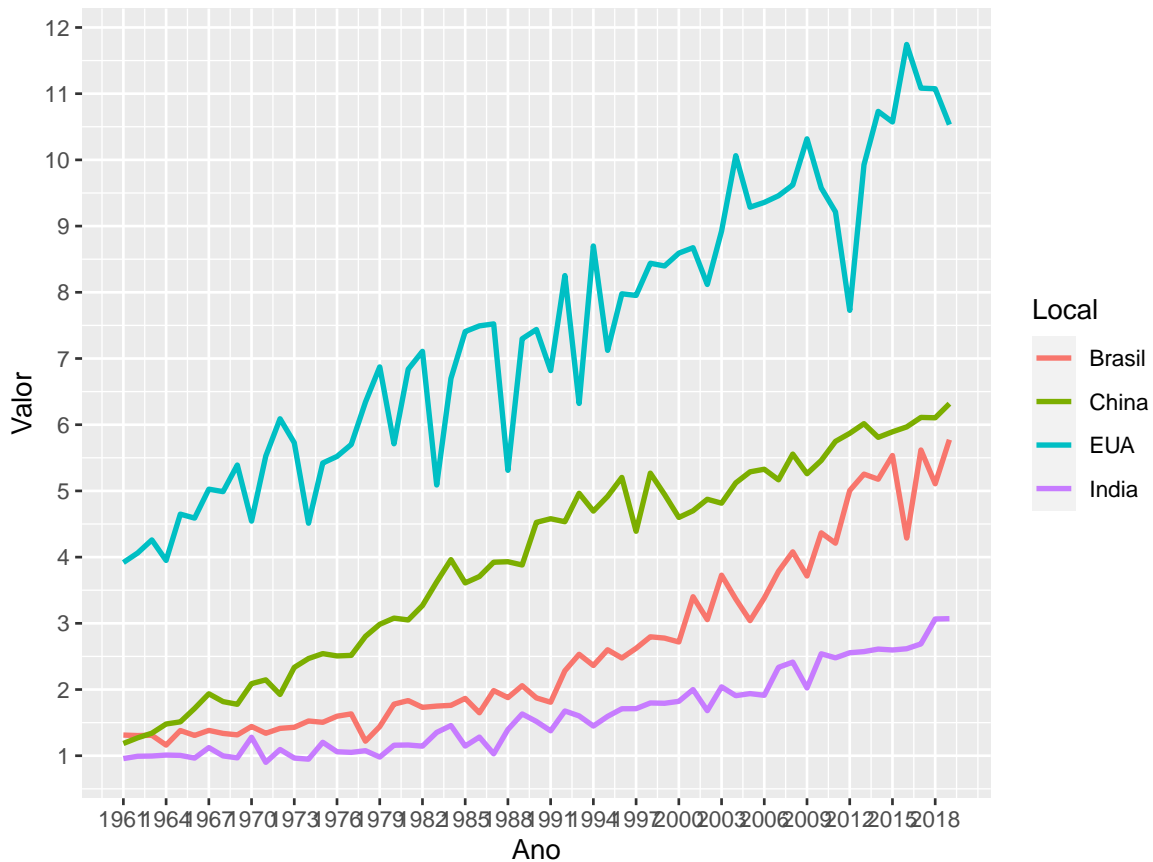
7.4.5 Escalas

A família de funções `scale_` confere propriedades para mudar as escalas de gráficos, cada qual com funções específicas. A seguir demonstraremos algumas delas.

Eixos

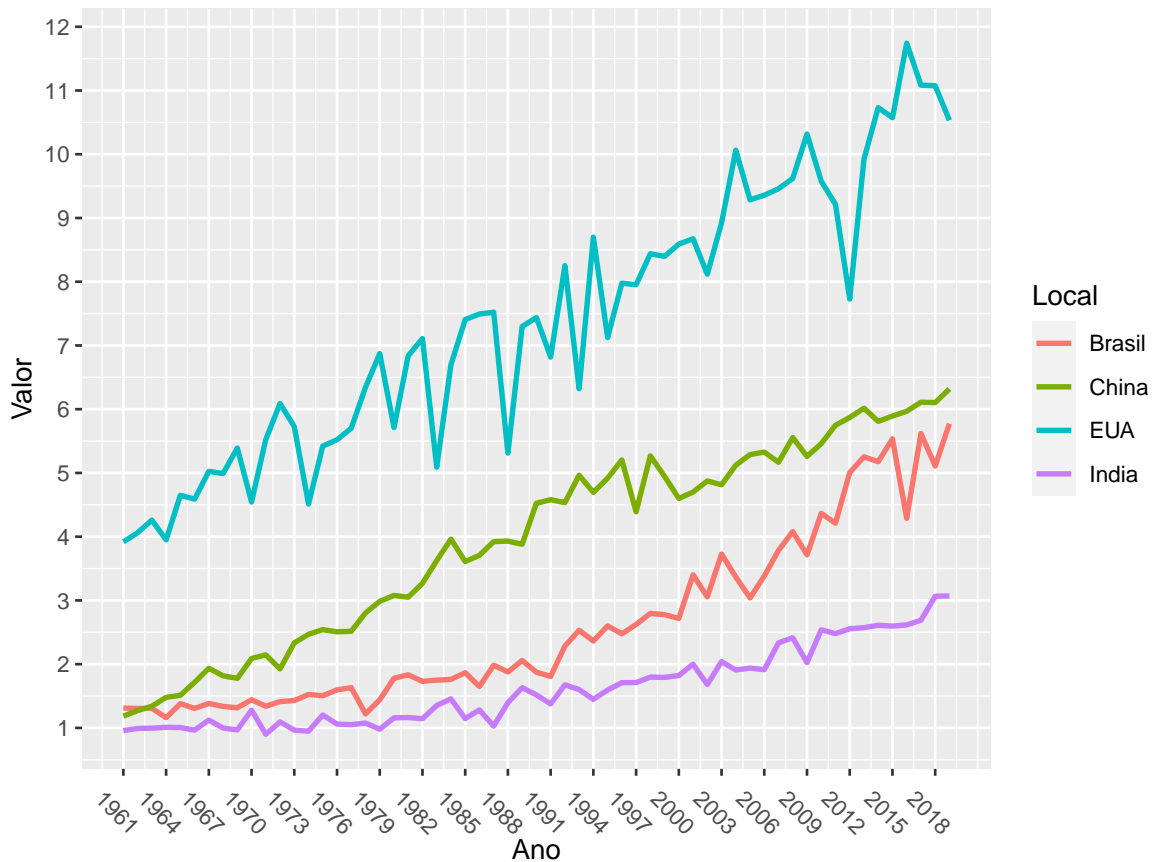
Para quebrarmos (`breaks =`) as escalas dos eixos `x` e `y` e redefinirmos outra sequência (`seq()`), utilizamos as funções `scale_x_continuous()` e `scale_y_continuous()`. Como fatores do argumento `seq()`, definimos o limite inferior (`from =`), o limite superior (`to =`) e a sequência da escala (`by =`).

```
ggplot(produtiv_milho,
  aes(x = Ano,
      y = Valor,
      color = Local))+
  geom_line(size = 1)+
  scale_x_continuous(breaks = seq(from = 1961, to = 2019, by = 3))+
  scale_y_continuous(breaks = seq(from = 0, to = 12, by = 1))
```



Perceba que, no eixo x, os anos ficaram apertados e mal apresentados. Para melhorar sua aparência, podemos alterar a angulação do texto (`angle =`) com o argumento `axis.text.x = element_text()`, dentro da função `theme()`.

```
ggplot(produtiv_milho,
  aes(x = Ano,
      y = Valor,
      color = Local))+
  geom_line(size = 1)+
  scale_x_continuous(breaks = seq(from = 1961, to = 2019, by = 3))+
  scale_y_continuous(breaks = seq(from = 0, to = 12, by = 1))+
  theme(axis.text.x = element_text(angle = -45))
```



Ainda podemos inserir uma segunda escala aos nossos gráficos, utilizando a função `scale_y_continuous()`. Uma segunda escala é útil para representar, em um mesmo gráfico, variáveis que apresentem escala numérica diferente. Como exemplo, utilizaremos a base de dados sobre o PIB do Brasil, disponível para download [clikando aqui](#).

```
pib_br <- read_csv("dados_ggplot2/pib_br.csv")
```

```
pib_br
```

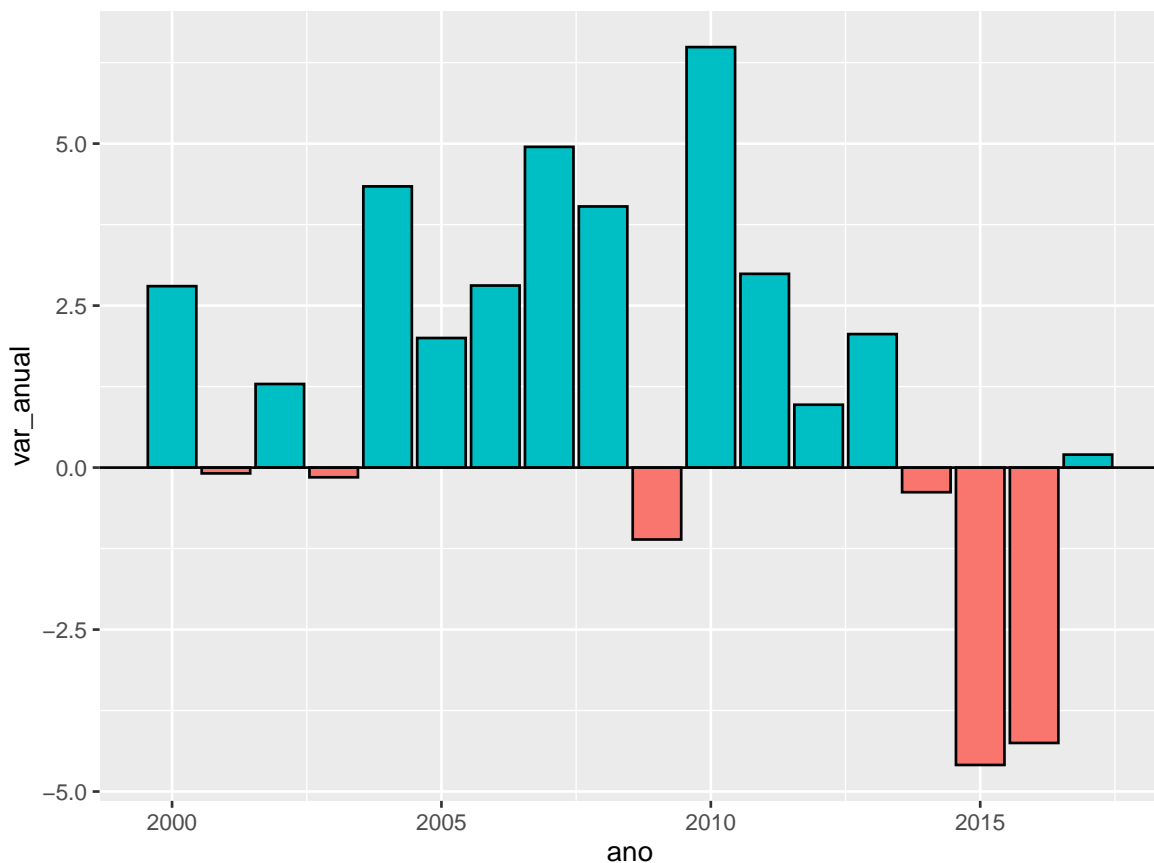
```
# A tibble: 18 x 3
  ano var_anual valor
<dbl> <dbl> <dbl>
1 2000 2.8 3722.
2 2001 -0.09 3155.
3 2002 1.29 2842.
4 2003 -0.15 3063.
5 2004 4.34 3623.
6 2005 2 4770.
7 2006 2.81 5860.
8 2007 4.95 7314.
9 2008 4.03 8788.
10 2009 -1.11 8553.
```


| | | | |
|----|------|-------|--------|
| 11 | 2010 | 6.49 | 11224. |
| 12 | 2011 | 2.99 | 13167. |
| 13 | 2012 | 0.97 | 12292. |
| 14 | 2013 | 2.06 | 12217. |
| 15 | 2014 | -0.38 | 12027. |
| 16 | 2015 | -4.59 | 8750. |
| 17 | 2016 | -4.25 | 8645. |
| 18 | 2017 | 0.2 | 9812. |

A *tibble* apresenta 18 observações e 3 variáveis, sendo elas o `ano`, entre 2000 e 2019; a variação anual do PIB (`var_anual`); e o valor bruto do PIB, em US\$ (`valor`).

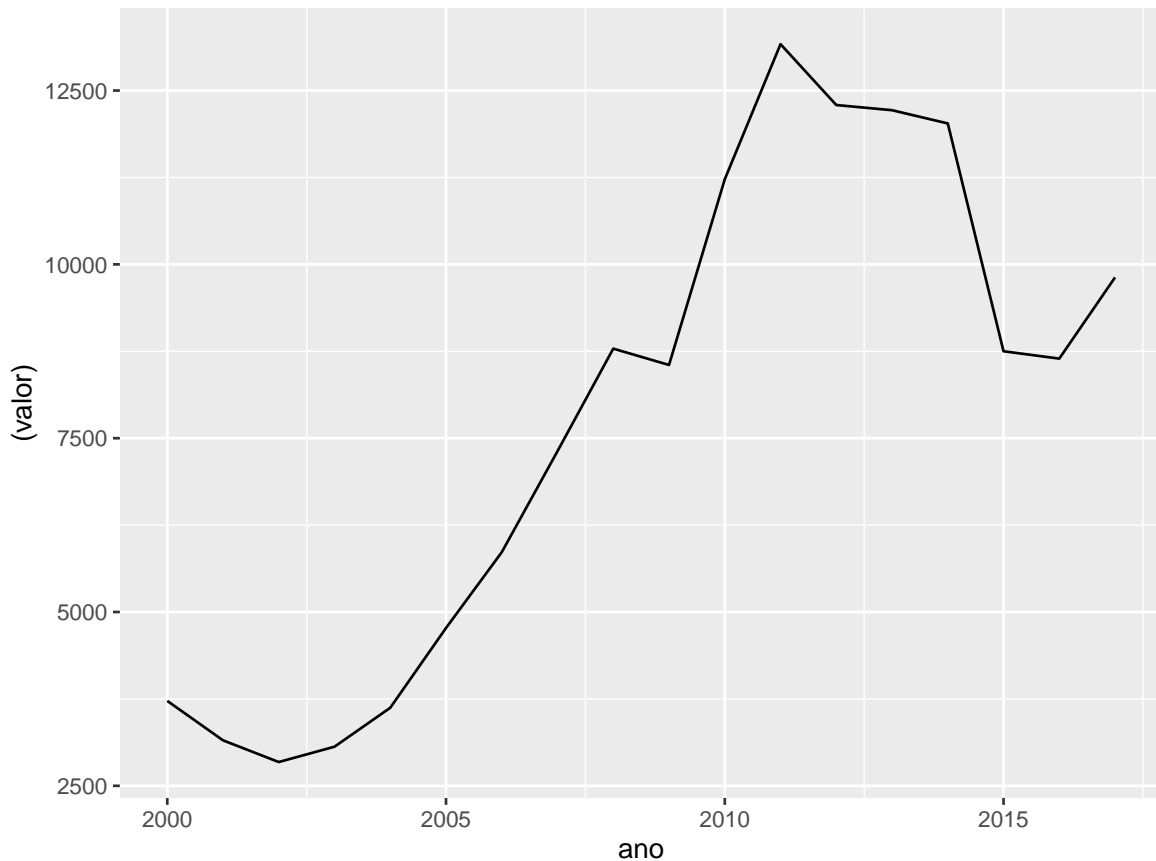
Primeiramente, plotaremos uma variável por vez, a fim de observarmos a escala do eixo y de cada uma, começando pela variação anual do PIB.

```
ggplot(pib_br,
       aes(x=ano))+
  geom_col(aes(y = var_anual,
              fill = var_anual>0),
          show.legend = FALSE,
          color = "black")+
  geom_hline(yintercept = 0, color = "black")
```



Para preencher as barras com cores, utilizamos o critério no qual valores positivos recebem uma cor e negativos, outra cor. Perceba que a escala da variação anual do PIB varia entre, aproximadamente, -5 e 6.

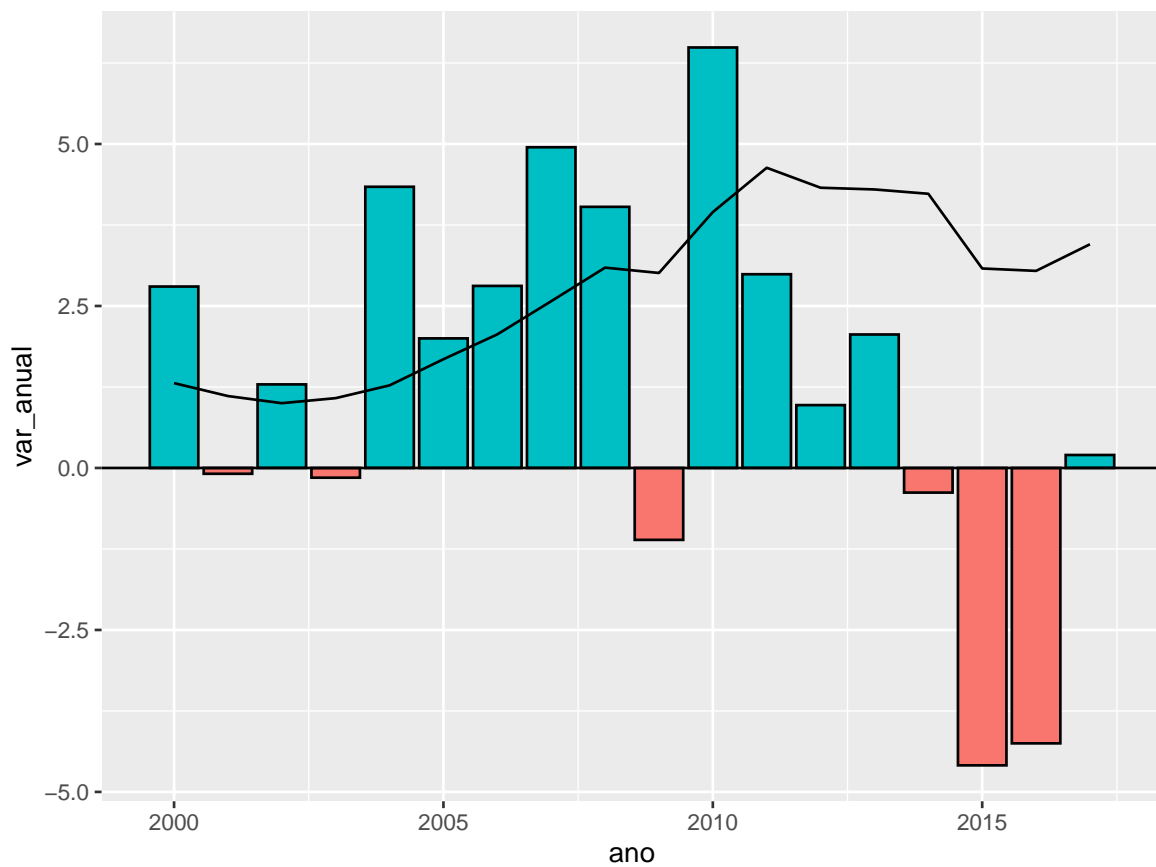
```
ggplot(pib_br,
       aes(x=ano))+
  geom_line(aes(y = (valor)))
```



Já no gráfico referente ao valor bruto do PIB, a escala varia entre 2500 e 13000.

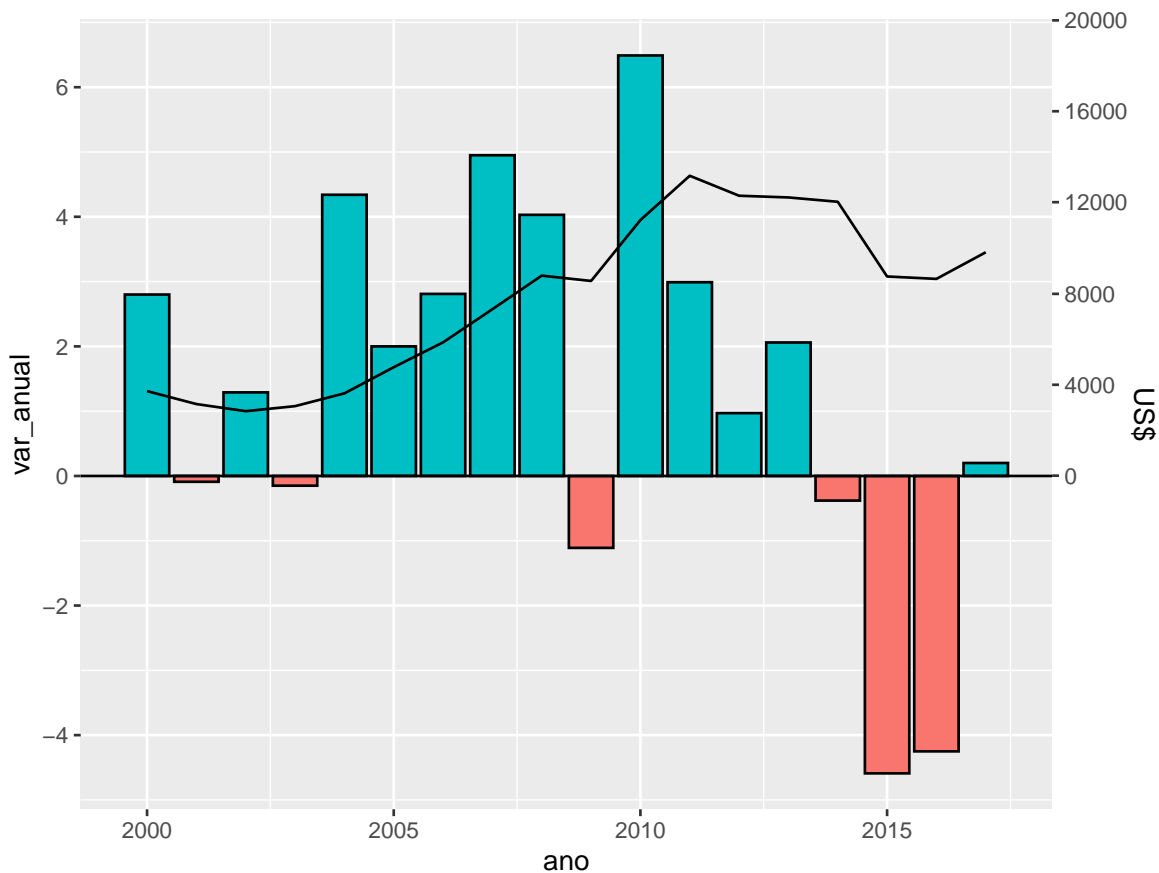
Assim, para unirmos ambas as medidas, precisaremos criar duas escalas no eixo y. Para isso, precisamos redefinir a escala da variável `valor`. A tática a ser utilizada será dividir os valores pelo menor valor apresentado na variável referente à variação anual do PIB.

```
ggplot(pib_br,
       aes(x=ano))+
  geom_col(aes(y = var_anual,
              fill = var_anual>0),
          show.legend = FALSE,
          color = "black")+
  geom_hline(yintercept = 0, color = "black")+
  geom_line(aes(y = (valor/2842)))
```



Dessa maneira, conseguimos representar ambas as variáveis em um mesmo gráfico. Contudo, precisamos definir um novo eixo y, a fim de representarmos os valores da variável `var_anual`. Para isso, utilizaremos a função `scale_y_continuous()`.

```
ggplot(pib_br,
  aes(x=ano))+
  geom_col(aes(y = var_anual,
    fill = var_anual>0),
    show.legend = FALSE,
    color = "black")+
  geom_hline(yintercept = 0, color = "black")+
  geom_line(aes(y = (valor/2842)))+
  scale_y_continuous(breaks = seq(-4.0, 6.0, by = 2),
    sec.axis = sec_axis(trans = ~. *2842,
      name = "US$",
      breaks = seq(0, 20000, by = 4000)))
```



Na função `scale_y_continuous()`, o primeiro argumento (`breaks =`) trata de redefinir a escala da variável `var_anual`. Posteriormente, o `sec.axis =` lida com a criação do segundo eixo y para representar a escala referente ao valor bruto do PIB brasileiro. Essa segunda escala toma como base os valores da escala primária do eixo y. O primeiro argumento é o `trans =`, que recebe uma fórmula, a fim de realizar uma operação matemática para alterar a escala do eixo y secundário. No nosso exemplo, multiplicaremos pelo valor 2842, o mesmo o qual dividimos anteriormente os valores do PIB bruto, para representar ambas as variáveis em um mesmo gráfico. Em seguida, nomeamos a nova escala com o argumento `names =` e definimos sua escala com `breaks =`.

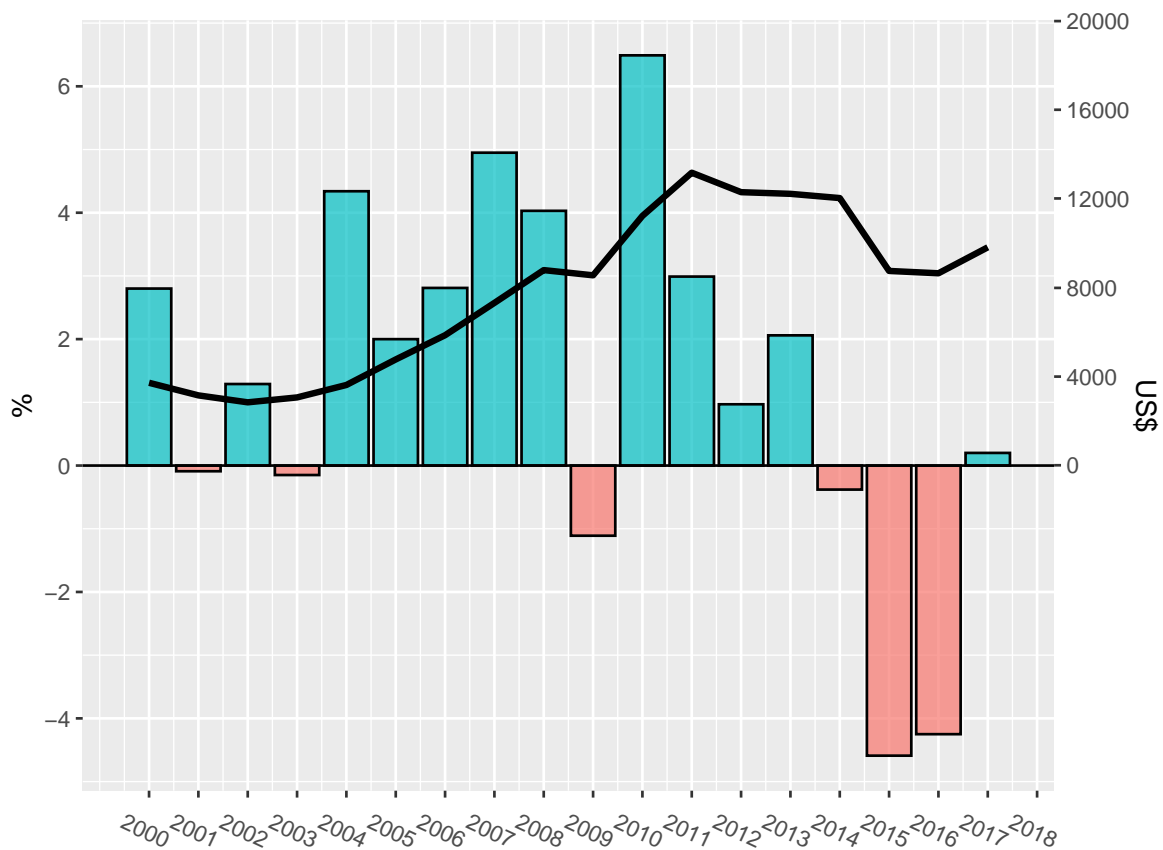
Por fim, podemos realizar mais alguns incrementos ao gráfico para deixá-lo mais apresentável.

```
ggplot(pib_br,
  aes(x=ano))+
  geom_col(aes(y = var_anual,
    fill = var_anual>0),
    show.legend = FALSE,
    color = "black",
    alpha = 0.7)+
  geom_hline(yintercept = 0, color = "black")+
  geom_line(aes(y = (valor/2842)),
    size = 1.2)+
  scale_y_continuous(name = "%",
    breaks = seq(-4.0, 6.0, by = 2),
```

```

sec.axis = sec_axis(trans = ~. *2842,
                     name = "US$",
                     breaks = seq(0, 20000, by = 4000)))+
scale_x_continuous(name = "",
                   breaks = seq(2000, 2019, 1))+
theme(axis.text.x = element_text(angle = -25))

```



Esse exemplo representa um gráfico mais completo e de nível elevado. Caso não tenha entendido algum passo, rode o código por camada, para acompanhar a progressão de construção do gráfico.

Cores

Para mudarmos as escalas de cores, utilizamos a `scale_color_` e `scale_fill_`.

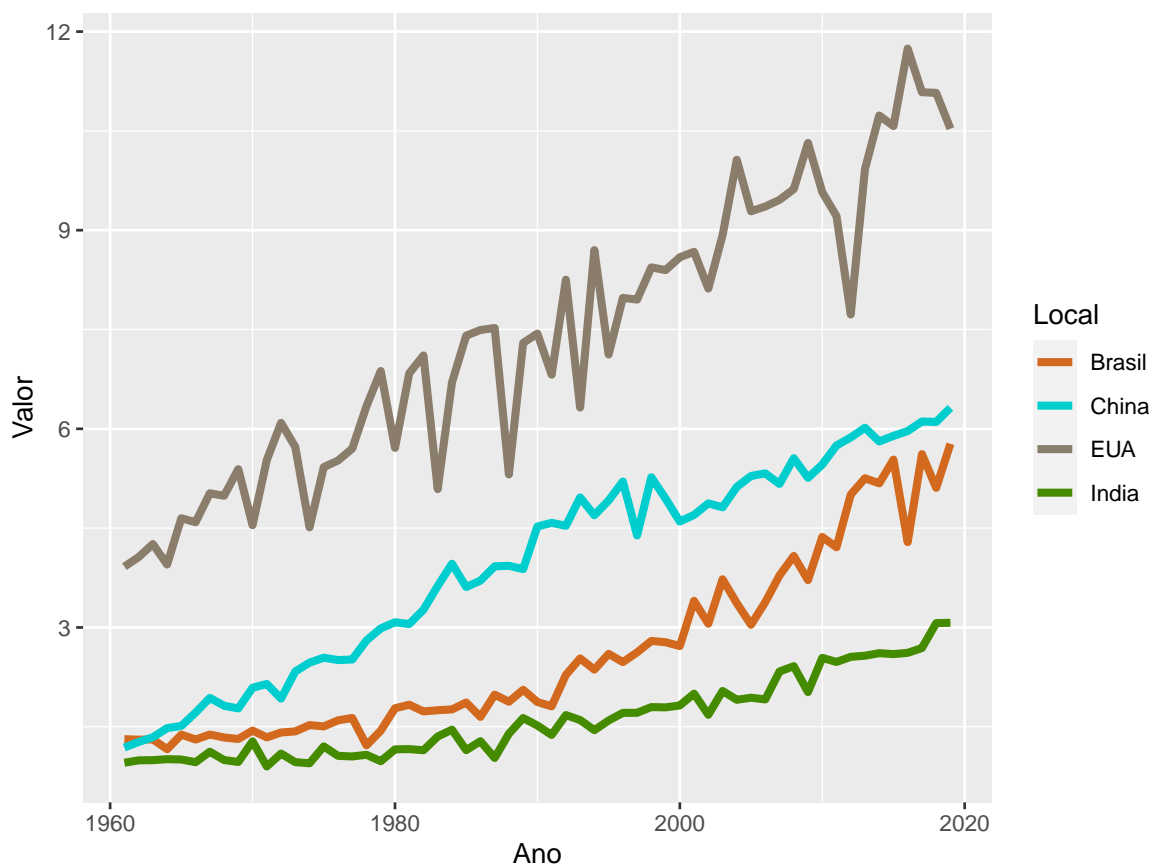
Para alterá-las manualmente, utiliza-se as funções `scale_color_manual()` e `scale_fill_manual()`, tendo como argumento o `values =`, que recebe um vetor com o nome das cores, cuja ordem no vetor diz respeito à ordem das variáveis às quais se quer atribuir tais cores.

```

ggplot(produtiv_milho,
       aes(x = Ano,
           y = Valor,

```

```
color = Local)))+  
geom_line(size = 1.5)+  
scale_color_manual(values = c("chocolate", "cyan3", "bisque4", "chartreuse4"))
```



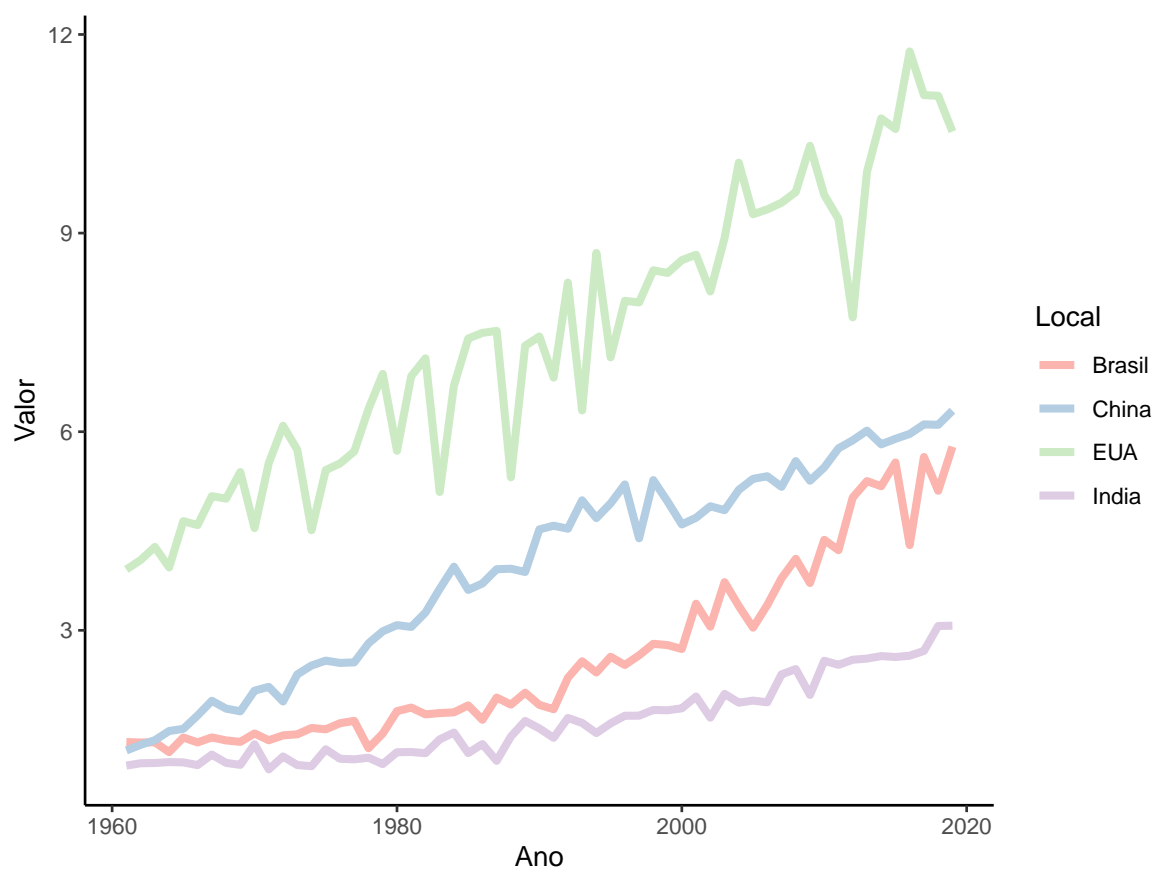
A figura 7.4 mostra os nomes das possíveis cores.

| | | | | |
|-----------------|-----------------|-----------------|-----------------|----------------|
| white | aliceblue | antiquewhite | antiquewhite1 | antiquewhite2 |
| antiquewhite3 | antiquewhite4 | aquamarine | aquamarine1 | aquamarine2 |
| aquamarine3 | aquamarine4 | azure | azure1 | azure2 |
| azure3 | azure4 | beige | bisque | bisque1 |
| bisque2 | bisque3 | bisque4 | | blanchedalmond |
| blue | blue1 | blue2 | blue3 | blue4 |
| blueviolet | brown | brown1 | brown2 | brown3 |
| brown4 | burlywood | burlywood1 | burlywood2 | burlywood3 |
| burlywood4 | cadetblue | cadetblue1 | cadetblue2 | cadetblue3 |
| cadetblue4 | chartreuse | chartreuse1 | chartreuse2 | chartreuse3 |
| chartreuse4 | chocolate | chocolate1 | chocolate2 | chocolate3 |
| chocolate4 | coral | coral1 | coral2 | coral3 |
| coral4 | cornflowerblue | cornsilk | cornsilk1 | cornsilk2 |
| cornsilk3 | cornsilk4 | cyan | cyan1 | cyan2 |
| cyan3 | cyan4 | darkblue | darkcyan | darkgoldenrod |
| darkgoldenrod1 | darkgoldenrod2 | darkgoldenrod3 | darkgoldenrod4 | darkgray |
| darkgreen | darkgrey | darkkhaki | darkmagenta | darkolivegreen |
| darkolivegreen1 | darkolivegreen2 | darkolivegreen3 | darkolivegreen4 | darkorange |
| darkorange1 | darkorange2 | darkorange3 | darkorange4 | darkorchid |
| darkorchid1 | darkorchid2 | darkorchid3 | darkorchid4 | darkred |
| darksalmon | darkseagreen | darkseagreen1 | darkseagreen2 | darkseagreen3 |
| darkseagreen4 | darkslateblue | darkslategray | darkslategray1 | darkslategray2 |
| darkslategray3 | darkslategray4 | darkslategrey | darkturquoise | darkviolet |
| deeppink | deeppink1 | deeppink2 | deeppink3 | deeppink4 |
| deepskyblue | deepskyblue1 | deepskyblue2 | deepskyblue3 | deepskyblue4 |

Figura 7.4: Nome das possíveis cores a serem definidas para os gráficos. Fonte: *The R Graph Gallery.*

Também, podemos alterar as colorações a partir de paletas de cores pré-definidas, utilizando a função `scale_color_brewer()`, tendo como argumento a `palette =`.

```
ggplot(produtiv_milho,
  aes(x = Ano,
      y = Valor,
      color = Local))+
  geom_line(size = 1.5)+
  scale_color_brewer(palette = "Pastel1")+
  theme_classic()
```



A figura 7.5 mostra as possíveis paletas de cores.



Figura 7.5: Nome das paletas de cores disponíveis para aplicarmos em nossos gráficos. Fonte: *The R Graph Gallery*.

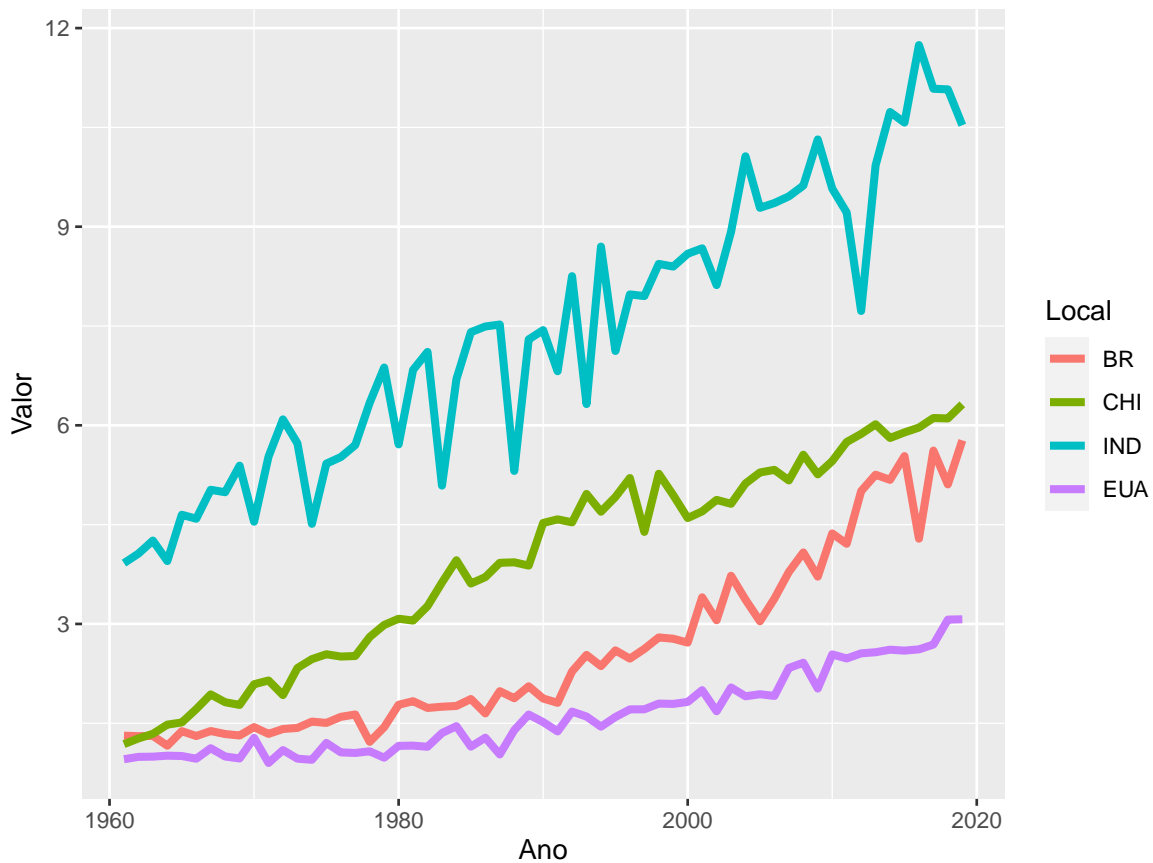
Caso queira saber mais sobre as escalas de cores existentes no R, confira o post do site *The R Graph Gallery*: [Dealing with colors in ggplot2](#).

Rótulos

Para trocar o nome das categorias na legenda, usa-se `scale_"argumento"_discrete()`, junto ao argumento `labels`. A seguir, citamos algumas das possibilidades:

- `scale_color_discrete()`: para alterar o nome das variáveis contidas no argumento `color`;
- `scale_fill_discrete()`: para alterar o nome das variáveis contidas no argumento `fill`;
- `scale_alpha_discrete()`: para alterar o nome das variáveis contidas no argumento `alpha`;
- `scale_size_discrete()`: para alterar o nome das variáveis contidas no argumento `size`.

```
ggplot(produktiv_milho,  
  aes(x =Ano,  
      y = Valor,  
      color = Local))+  
geom_line(size = 1.5)+  
scale_color_discrete(labels = c("BR", "CHI", "IND", "EUA"))
```



7.5 Gráficos de medidas-resumo

Voltemos aos dados dos alunos de Estatística Aplicada para fazermos alguns gráficos de medidas-resumo, importantes para observarmos a distribuição de valores.

```
dados_alunos
```

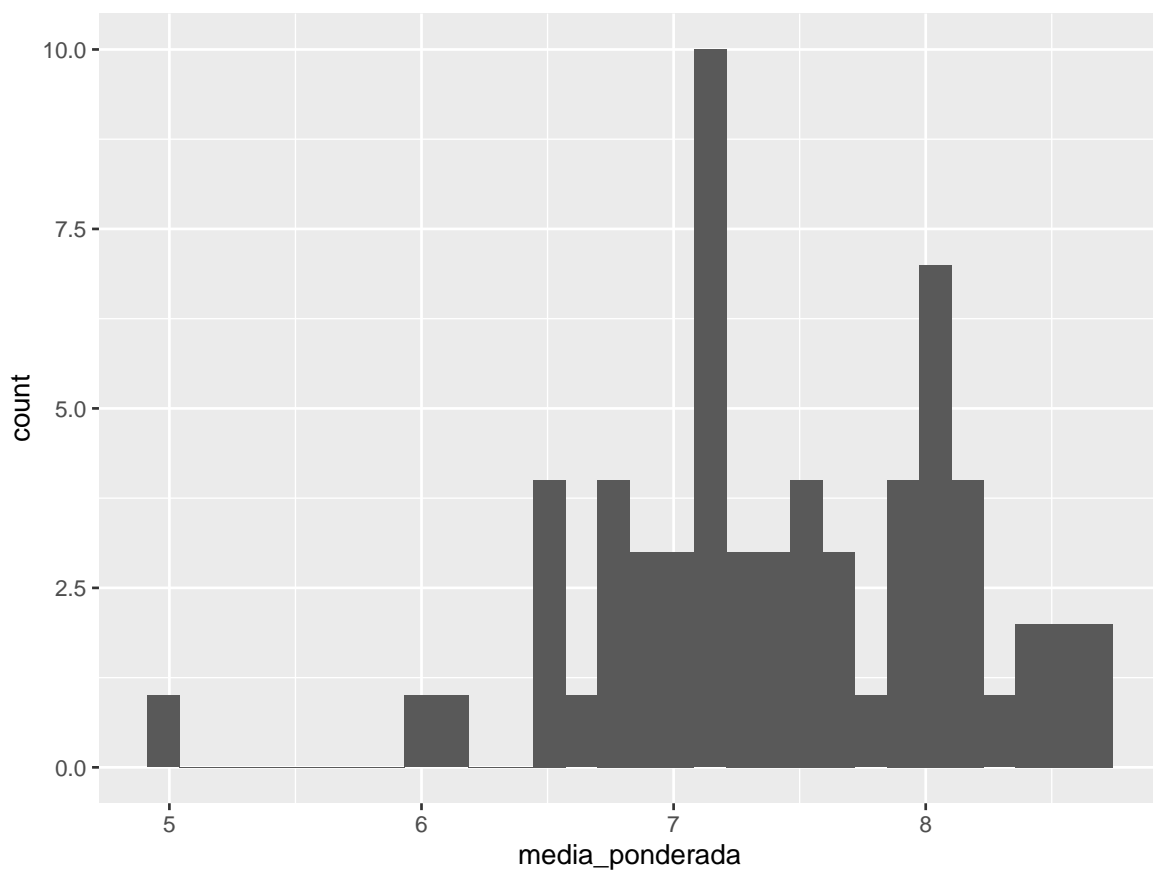
```
# A tibble: 64 x 7
  sexo  idade altura  peso horas_estudo media_ponderada futuro
  <chr> <dbl>  <dbl> <dbl>      <dbl>      <dbl> <chr>
1 M      23   1.75   80         2         7.5 academico
2 F      19   1.67   65         2         8.3 mercado
3 M      19   1.7    90         3         6.9 mercado
4 M      22   1.73   87         3         7.1 academico
5 M      19   1.83   71         2         6.5 mercado
6 M      19   1.8    80         3         8.6 mercado
7 M      20   1.9    90         2         7.8 academico
8 F      20   1.6    55         1         8  mercado
9 F      24   1.62   55         2         8.2 academico
10 F     18   1.64   60         2         7.3 mercado
# ... with 54 more rows
```

7.5.1 Histogramas

Neste primeiro caso, faremos um histograma referente à média ponderada dos alunos. Esse tipo de gráfico é útil para verificar a frequência de uma variável e a sua distribuição.

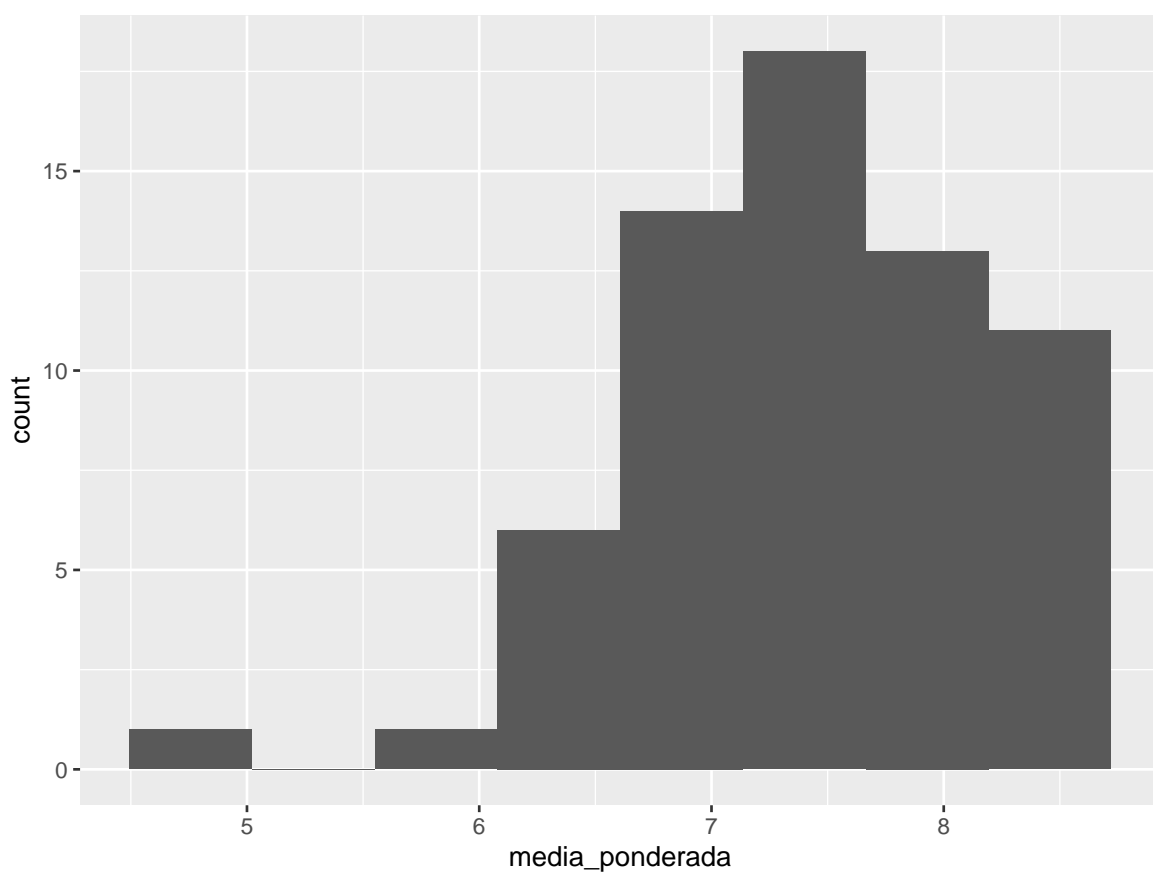
Para isso, utilizamos a função `geom_histogram()`, sendo necessário indicar somente o atributo `x =`, pois o eixo `y` será construído, automaticamente, a partir da contagem dos valores correspondentes às variáveis `x`.

```
ggplot(dados_alunos)+
  geom_histogram(aes(x = media_ponderada))
```



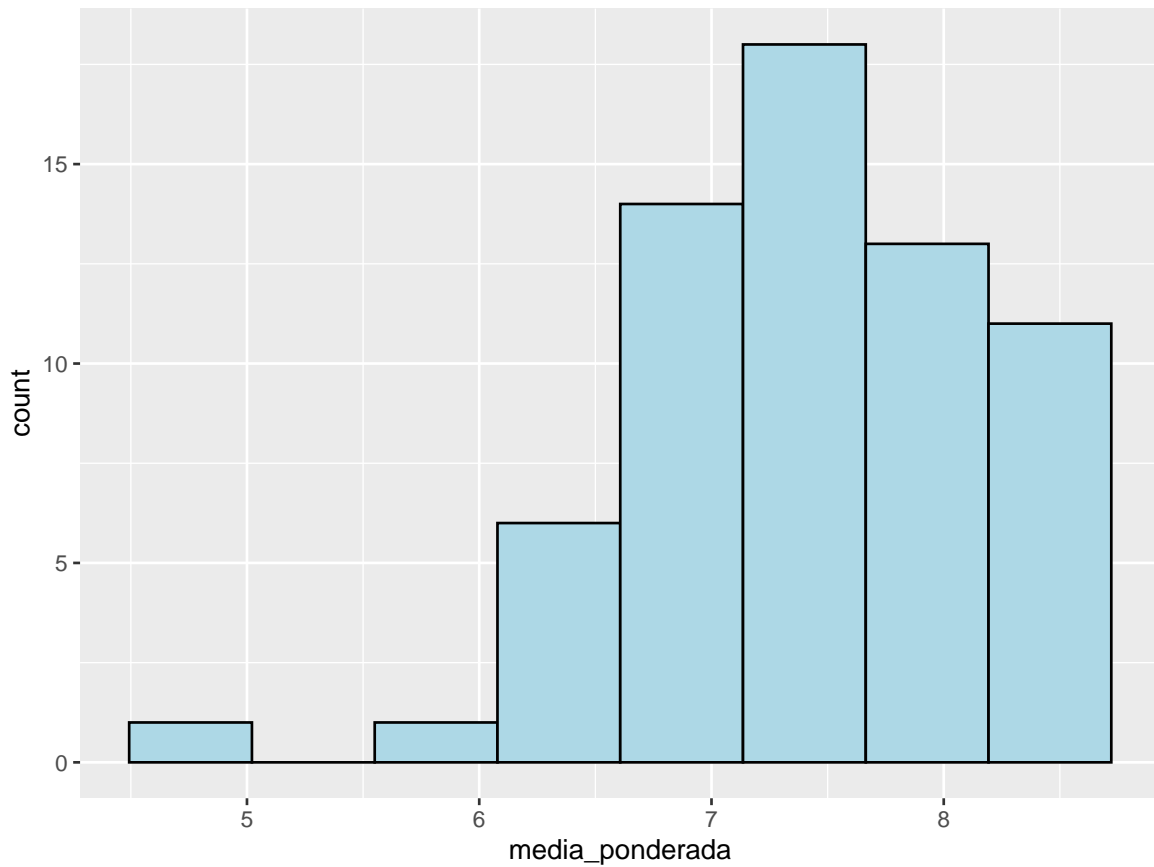
Por padrão, a função define que o histograma apresenta 30 intervalos. Porém, este número não é o ideal para representar os nossos dados. Para redefini-lo, utilizamos o argumento `bins`.

```
ggplot(dados_alunos)+  
  geom_histogram(aes(x = media_ponderada),  
                 bins = 8)
```



Podemos melhorar a aparência do histograma utilizando os argumentos `color = -` para colorir a borda das barras - e o `fill =`, para colorir o seu interior.

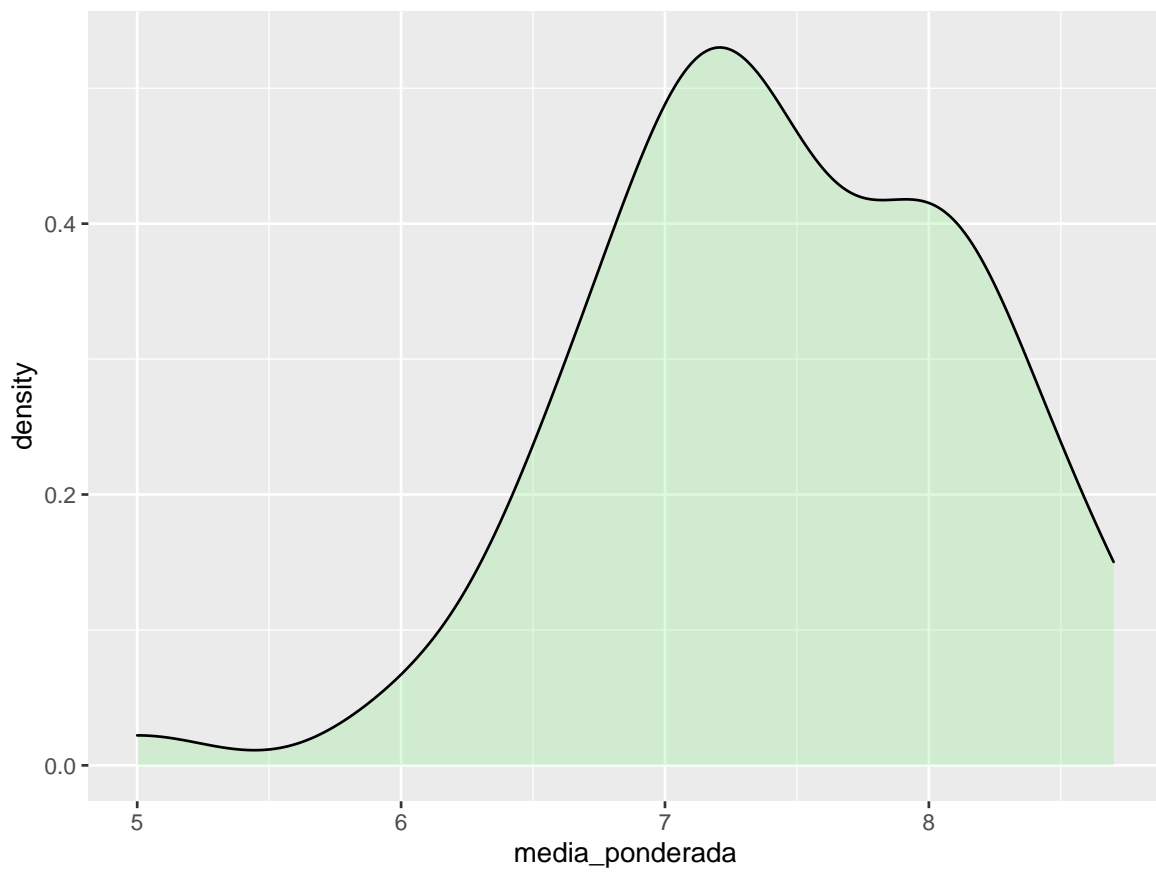
```
ggplot(dados_alunos)+  
  geom_histogram(aes(x = media_ponderada),  
                 bins = 8,  
                 color = "black",  
                 fill = "light blue")
```



7.5.2 Gráfico de densidade

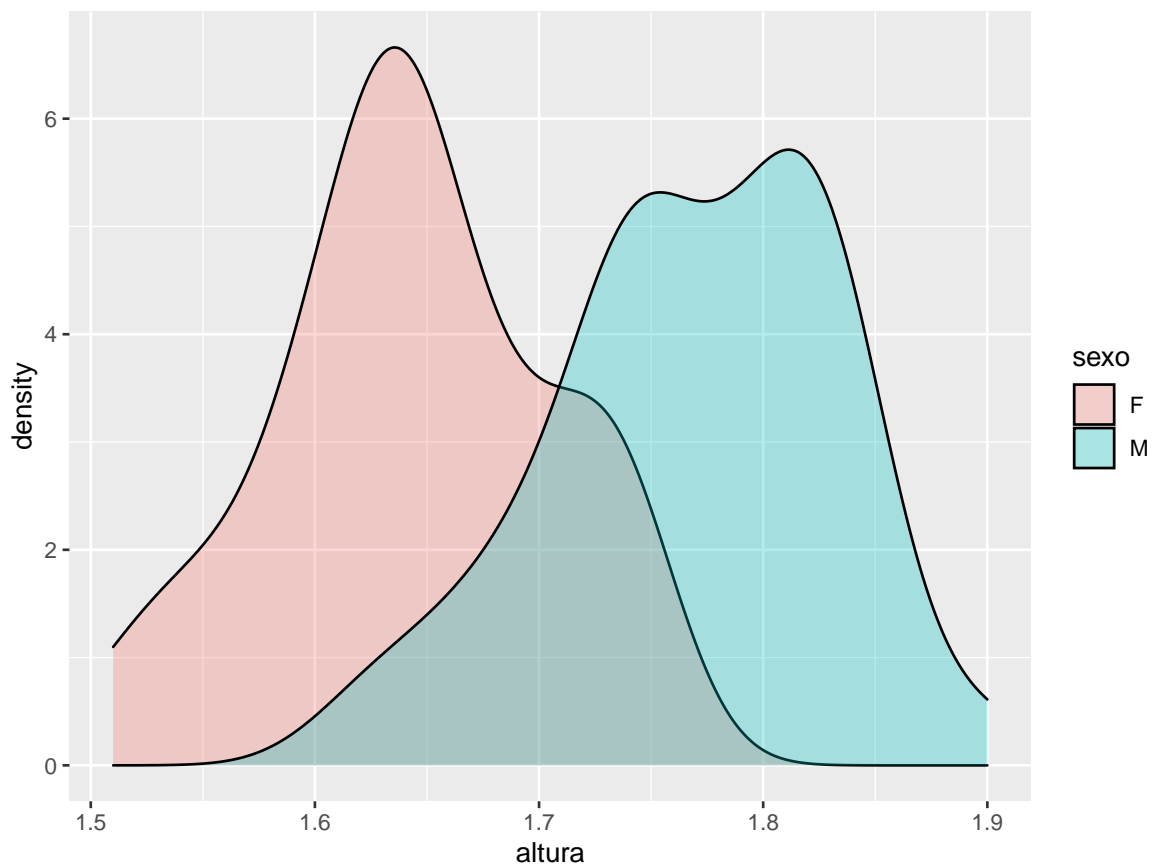
A construção do gráfico de densidade é semelhante ao histograma. Utilizamos a função `geom_density()`, atribuindo apenas uma variável contínua ao eixo x, sendo o eixo y construído automaticamente, de acordo com os valores da variável do eixo x. Ademais, podemos preencher seu interior com o argumento `fill =` e alterar a transparência da cor com o argumento `alpha`, cuja escala vai de 0 a 1, sendo 0 o valor máximo de transparência.

```
ggplot(dados_alunos,  
  aes(x = media_ponderada))+  
  geom_density(fill = "light green",  
    alpha = 0.3)
```



Podemos representar mais de uma densidade em um mesmo gráfico. Basta atribuir uma variável categórica ao argumento `fill =` ou `color =`, a fim de distinguir as densidades.

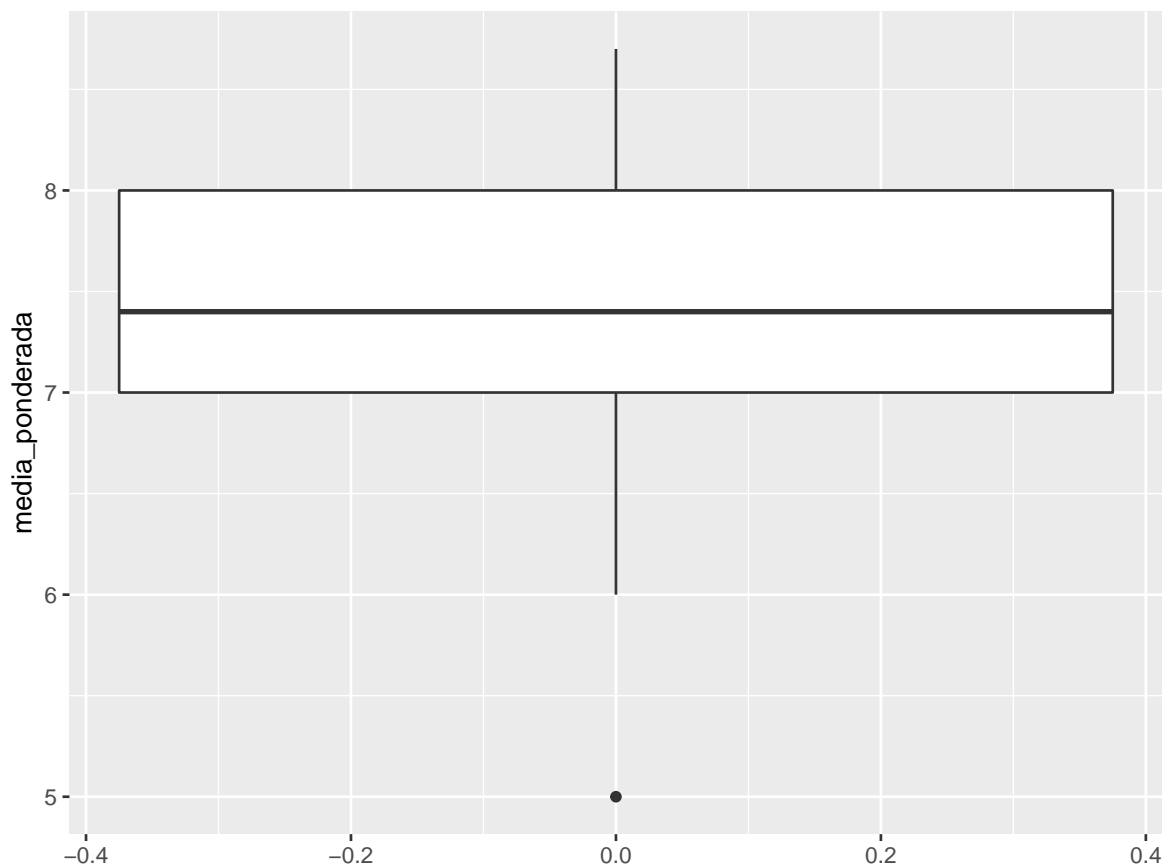
```
ggplot(dados_alunos,  
  aes(x = altura,  
      fill = sexo))+  
  geom_density(alpha = 0.3)
```



7.5.3 Boxplot

Por fim, temos o boxplot, muito útil para observarmos a distribuição de valores de uma variável. Para fazermos os boxplots, utilizamos a função `geom_boxplot()`.

```
ggplot(dados_alunos,  
       aes(y = media_ponderada))+  
  geom_boxplot()
```

Nesse primeiro caso, fizemos um boxplot da média ponderada dos alunos, atribuindo os valores da variável no eixo y. Para conferir os valores dos quartis, além da amplitude, podemos utilizar a função `summary()`. Para calcular a distância interquartil, utilizamos a função `IQR()`.

```
summary(dados_alunos$media_ponderada)
```

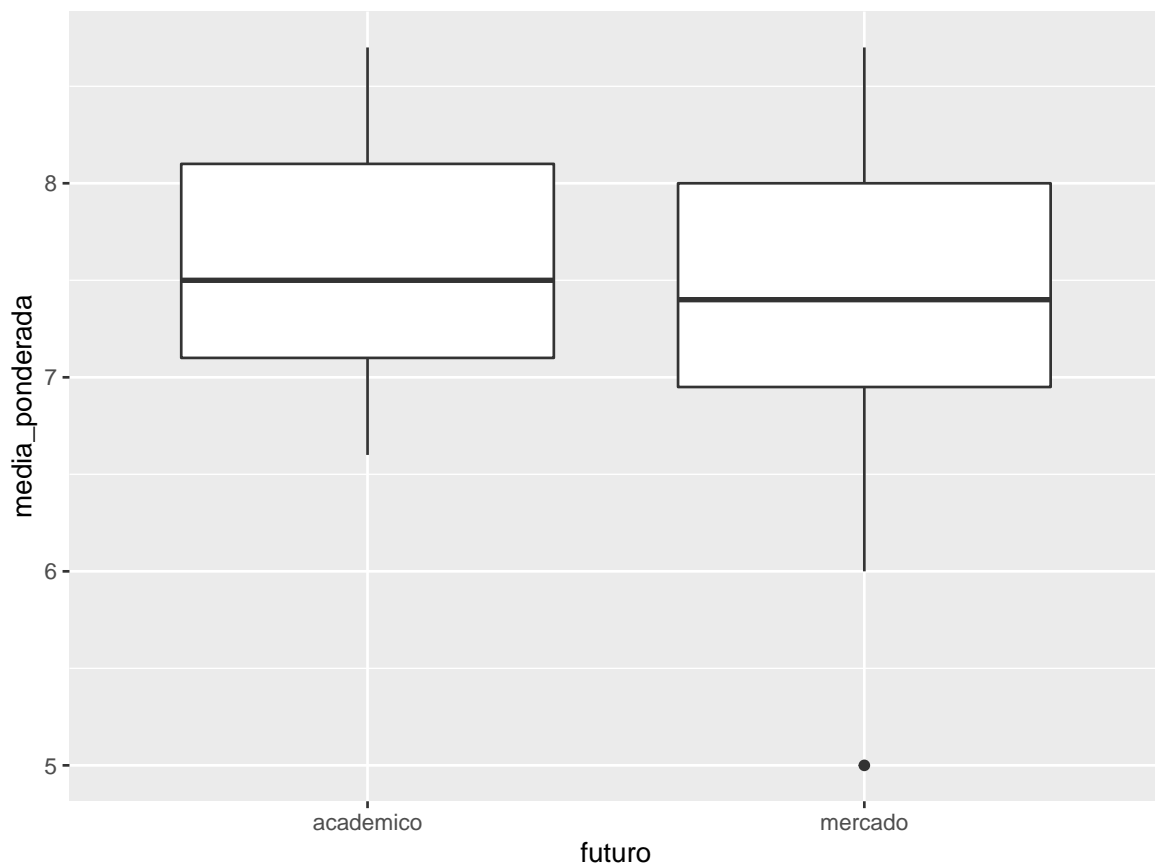
| Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|-------|---------|--------|-------|---------|-------|
| 5.000 | 7.000 | 7.400 | 7.425 | 8.000 | 8.700 |

```
IQR(dados_alunos$media_ponderada)
```

```
[1] 1
```

Podemos incluir outras variáveis ao nosso boxplot, como, por exemplo, a variável `futuro`.

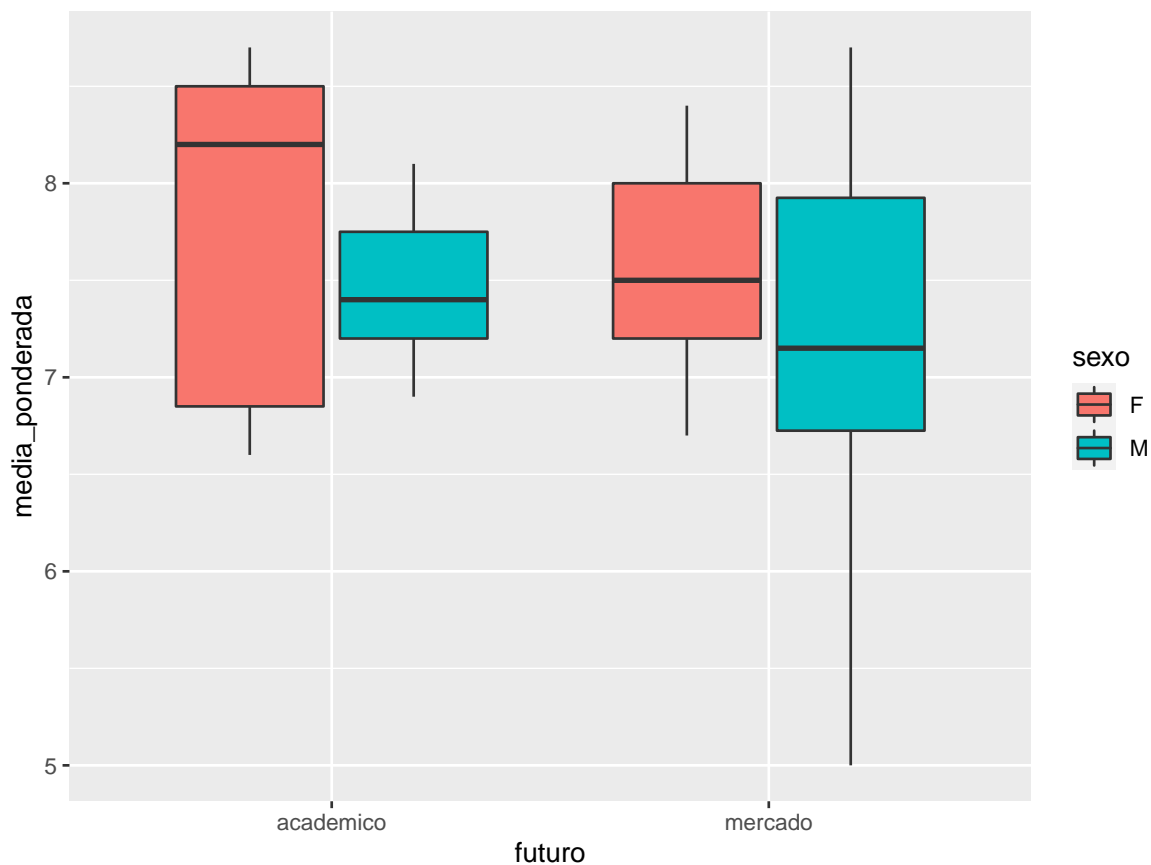
```
ggplot(dados_alunos,
       aes(x = futuro,
           y = media_ponderada)) +
  geom_boxplot()
```



Neste caso, podemos ver a distribuição da média ponderada dos alunos e alunas de acordo com as perspectivas futuras de cada um. Perceba que a variável categórica **futuro** foi atribuída ao eixo x.

Agora, caso se queira dividir os boxplots anteriores de acordo com o sexo, podemos atribuí-la ao argumento **fill**.

```
ggplot(dados_alunos,  
  aes(x = futuro,  
      y = media_ponderada,  
      fill = sexo))+  
  geom_boxplot()
```



Assim, podemos observar a distribuição das médias ponderadas de acordo com o sexo e a perspectiva futura.

7.6 Juntar gráficos diferentes

Temos a possibilidade de juntar gráficos diferentes em uma mesma apresentação. Para isso, utilizamos o pacote `patchwork`.

```
install.packages("patchwork")
```

```
library(patchwork)
```

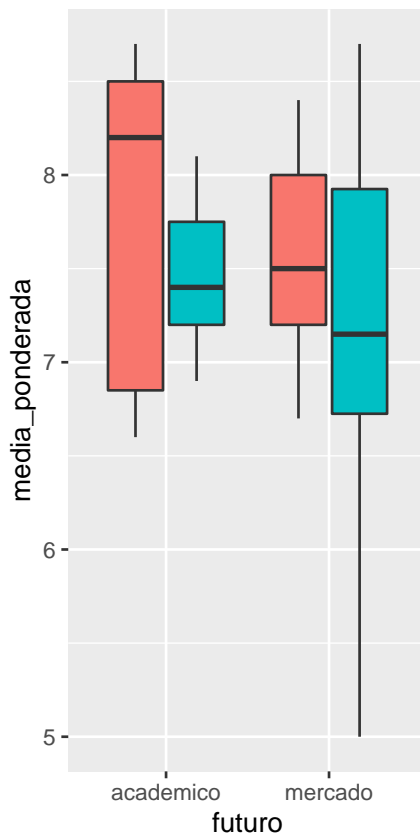
O pacote funciona de maneira bem simples. Para juntarmos gráficos, devemos salvá-los em um objeto e, posteriormente, uni-los com um sinal de `+`. Veja o exemplo a seguir:

```
# Boxplot salvo no objeto "g1"
g1 <- ggplot(dados_alunos,
             aes(x = futuro,
                 y = media_ponderada,
                 fill = sexo)) +
```

```
geom_boxplot()

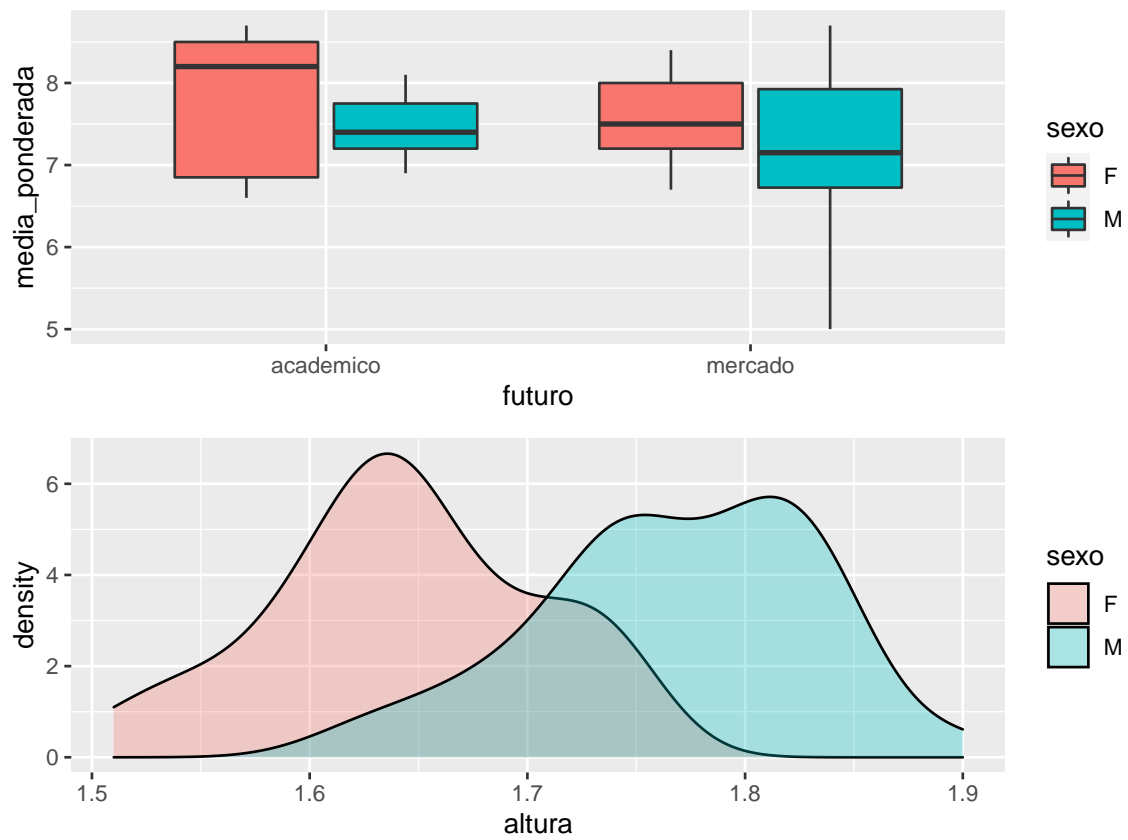
# Gráfico de densidade salvo no objeto "g2"
g2 <- ggplot(dados_alunos,
             aes(x = altura,
                 fill = sexo))+
  geom_density(alpha = 0.3)

# Unindo os gráficos "g1" e "g2"
g1 + g2
```



Podemos dispô-los um embaixo do outro, com o operador `/`.

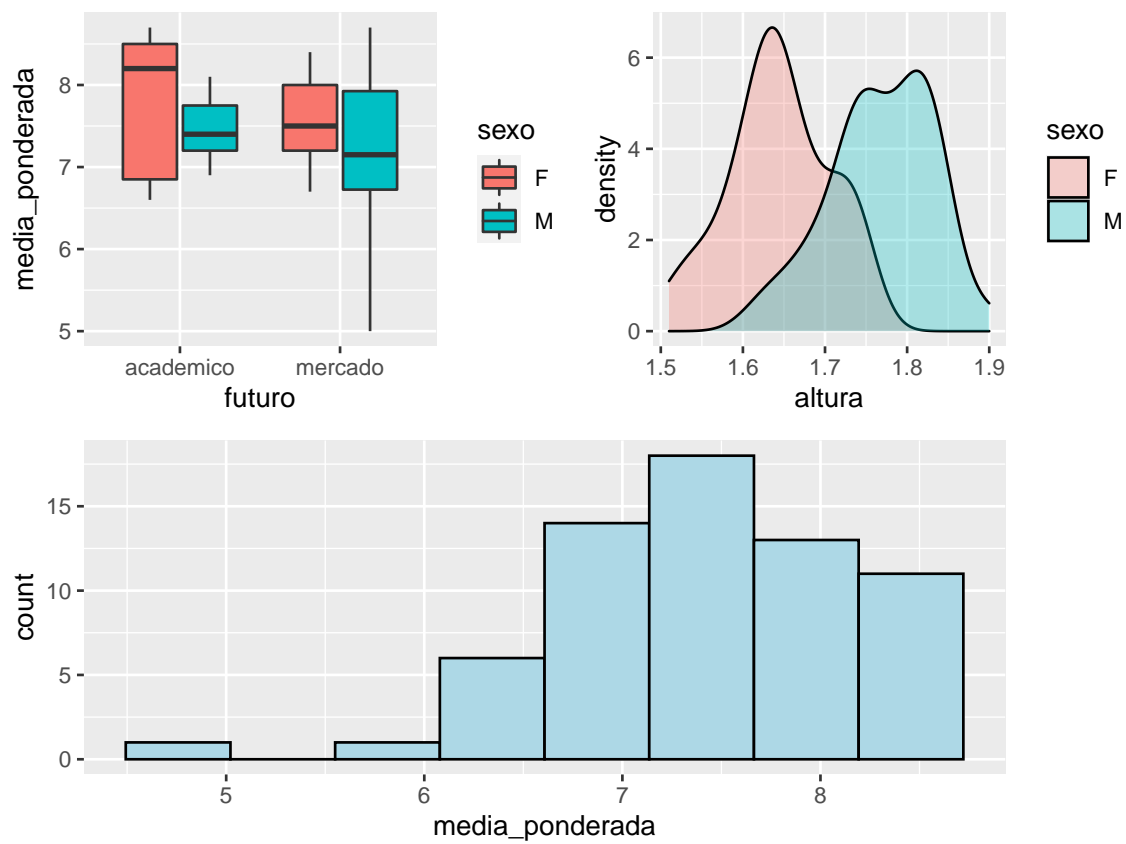
```
g1 / g2
```

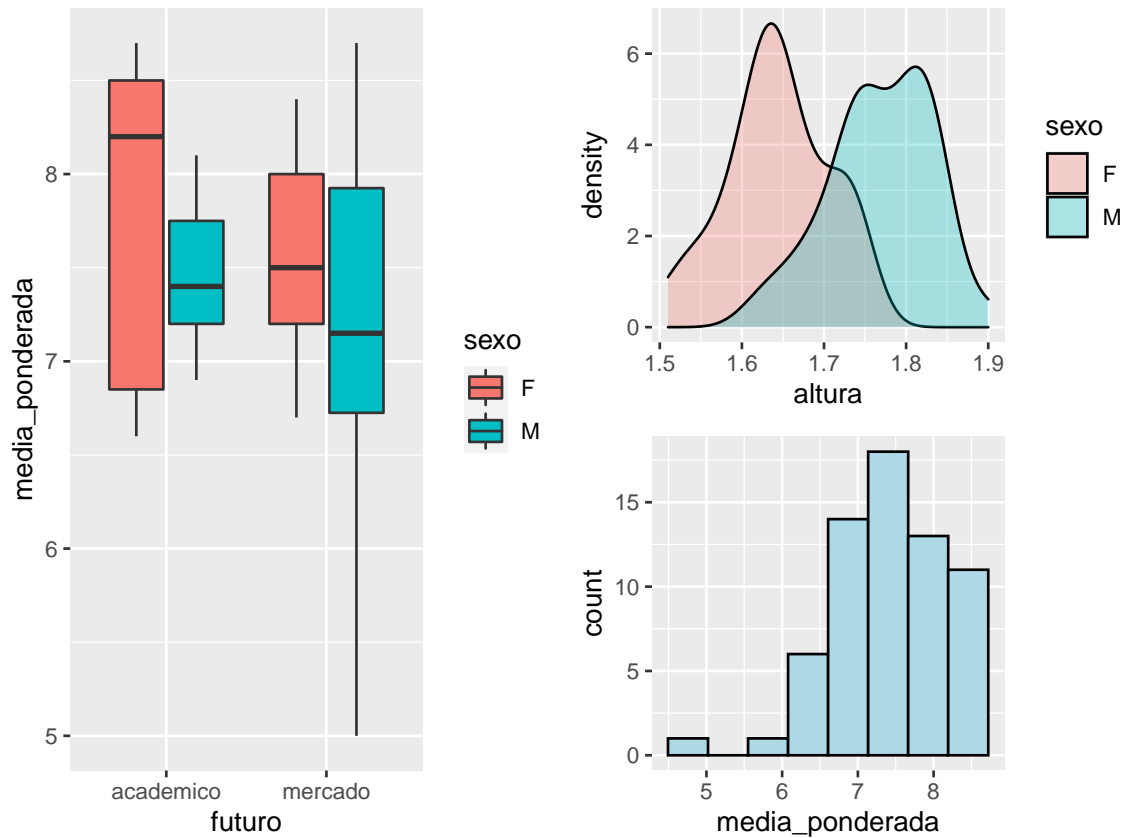


Além disso, podemos inserir mais gráficos ao conjunto. Nesse caso, incluiremos o histograma.

```
g3 <- ggplot(dados_alunos)+
  geom_histogram(aes(x = media_ponderada),
    bins = 8,
    color = "black",
    fill = "light blue")

(g1 + g2) / g3
```





Assim, dependendo da combinação de operações entre objetos, a partir do pacote `patchwork`, é possível dispor os gráficos de diferentes formas.

Como visto ao longo deste capítulo, percebemos que o pacote `ggplot2` possui ferramentas poderosas e versáteis para lidar com gráficos. Há diversas outras funcionalidades presentes no pacote, as quais podem (e devem) ser exploradas. Contudo, neste primeiro momento, o que foi exposto se apresenta como uma base para possibilitar a execução dos primeiros gráficos em R, além de ser a porta de entrada ao leitor para que possa aprofundar e aprimorar seus gráficos de maneira mais independente.

Para os leitores que desejam aprofundar o conhecimento no pacote `ggplot2`, deixo como recomendação alguns livros, todos disponíveis gratuitamente na *web*:

- [R Graphics Cookbook](#) - Winston Chang;
- [ggplot2: Elegant Graphics for Data Analysis](#) - Hadley Wickham;
- [R Gallery Book](#) - Kyle W. Brown.

Referências

Chang, Winston. 2021. R Graphics Cookbook. 2nd ed. Beijing: O'Reilly Media. <https://r-graphics.org/>.

Damiani, Athos; Lente, Caio; Milz, Beatriz; Falbel, Daniel; Correa, Fernando; Trecenti, Julio; Ludovice, Nicole; Amorim, William. 2021. Ciência de Dados em R. Curso-R. <https://livro.curso-r.com/index.html>.

Wickham, Hadley; Golemund, Garrett. 2017. R for Data Science. 1st ed. Sebastopol, California: O'Reilly Media. <https://r4ds.had.co.nz/index.html>.

Xie, Yihui. 2015. Dynamic Documents with R and Knitr. 2nd ed. Boca Raton, Florida: Chapman; Hall/CRC. <http://yihui.org/knitr/>. 2021. Bookdown: Authoring Books and Technical Documents with r Markdown. <https://CRAN.R-project.org/package=bookdown>.